



Practical Angular

Build apps while learning
the new concepts

[Techiediaries.com](https://www.techiediaries.com)

Ahmed Bouchefra

Practical Angular: Build your first web apps with Angular 8

This book will walk you through building several apps with Angular while learning new concepts and practices along the way. This book is beginners friendly and doesn't assume any prior experience with Angular. You'll learn Angular from scratch building easy and relatively more complex apps step by step.

Ahmed Bouchefra

This book is for sale at <http://leanpub.com/practical-angular>

This version was published on 2019-08-28



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2019 Ahmed Bouchefra - techiediaries.com

Contents

| | |
|---|-----------|
| Introduction to Angular | 1 |
| What is TypeScript | 1 |
| Introducing Angular | 2 |
| Angular Concepts | 3 |
| Angular Libraries | 5 |
| Why Would you Use Angular | 6 |
| Getting Started with Angular | 7 |
| A Primer on Angular CLI | 10 |
| Conclusion | 14 |
| Building a Calculator App with Angular | 15 |
| Prerequisites | 15 |
| Generating our Angular Calculator Project | 15 |
| Angular Modules & Components | 16 |
| Creating our Calculator UI | 18 |
| Angular Property Binding by Example | 24 |
| Angular Event Binding by Example | 25 |
| Conclusion | 26 |
| Build your First Angular Web Application with Firebase and Bootstrap | 27 |
| Setting up your Project | 27 |
| Installing Angular CLI | 28 |
| Creating an Angular Project | 28 |
| Styling our Components with Bootstrap 4 | 32 |
| Angular Modules | 35 |
| Child & Nested Routing | 39 |
| Adding Firebase Authentication | 42 |
| Securing the UI with Router Guards | 55 |
| Build a Contact Form with Angular | 62 |
| Building a Template-Based Form | 62 |
| Building a Reactive Form | 66 |
| Conclusion | 68 |
| Building a News App with Angular - HttpClient | 69 |

CONTENTS

| | |
|--|-----------|
| Prerequisites | 69 |
| Creating an Angular Project | 69 |
| Adding an Angular Service | 70 |
| How the <code>HttpClient.get()</code> Method Works | 72 |
| Creating an Angular Component | 72 |
| Conclusion | 75 |
| Building a Full-Stack App with Angular, PHP and MySQL | 76 |
| Prerequisites | 76 |
| Creating the PHP Application | 77 |
| Implementing the Read Operation | 79 |
| Implementing the Create Operation | 80 |
| Implementing the Update Operation | 81 |
| Implementing the Delete Operation | 82 |
| Serving the PHP REST API Project | 83 |
| Creating the MySQL Database | 83 |
| Wrap-up | 84 |
| Creating the Angular Front-End | 84 |
| Installing Angular CLI | 84 |
| Generating a New Project | 84 |
| Setting up <code>HttpClient</code> & Forms | 85 |
| Creating an Angular Service | 86 |
| Creating the Angular Component | 89 |
| Conclusion | 96 |

Introduction to Angular

Before diving into the practical steps for developing your first web app(s) with Angular from scratch, let's first learn about the basics of Angular.

Throughout this book's chapters, you'll learn how to use Angular to build client-side web applications for mobile and desktop.

You'll learn:

- How to use the Angular CLI for quickly creating a front-end project, generating components, pipes, directives, and services, etc.
- How to add routing and navigation using the Angular router.
- How to build and submit forms, using reactive and template-based approaches.
- How to use Angular Material for building professional-grade UIs.

This first chapter is an in-depth introduction to Angular aimed at new developers who have little experience with JavaScript client-side frameworks and want to learn the essential concepts of Angular.

Let's get started!

What is TypeScript

TypeScript is a strongly-typed superset of JavaScript developed by Microsoft. This means three things:

- TS provides more features to the original JavaScript language.
- TS doesn't get in the way if you still want to write plain JavaScript.
- TypeScript does also integrate well with most used JavaScript libraries.

TypeScript is not the first attempt to create a super-set of JavaScript but it's by far the most successful one. It provides powerful OOP (Object Oriented Programming) features like inheritance interfaces and classes, a declarative style, static typing, and

modules. Although many of these features are already in JavaScript they are different as JS follows a prototypical-based OOP, not class-based OOP.

TS features make it easy for developers to create complex and large JavaScript apps that are easier to main and debug.

TypeScript is supported by two big companies in the software world; Microsoft, obviously because it's the creator but also by Google as it was used to develop Angular from v2 up to Angular 7 (the current version). It's also the official language and the recommended language to build Angular 2+ apps.

TypeScript is a compiled language which means we'll need to transpile it into JavaScript to be able to run it in web browsers which do only understand one programming language. Fortunately, the TS transpiler integrates well with the majority of build systems and bundlers.

You can install the TypeScript compiler using [npm](#) and then you can call it by running the `tsc source-file.ts` command from your terminal. This will generate a `source-file.js` JavaScript file with the same name. You can control many aspects of the compilation process using a `tsconfig.json` configuration file. We can specify the module system to compile to and where to output the compiled file(s) etc.

For large projects, you need to use advanced tools like task runners like Gulp and Grunt and code bundlers like the popular Webpack.

You can use [grunt-typescript](#) and [gulp-typescript](#) plugins for integrating TypeScript with Gulp and Grunt which will allow you to pass the compiler options from your task runners.

For Webpack, you can use the [loader](#) to work with TypeScript.

More often than not, you'll need to use external JavaScript libraries in your project. You'll also need to use **type definitions**.

Type definitions are files that end with the `.d.ts` extension – They allow us to use TypeScript interfaces created by other developers for different JavaScript libraries to integrate seamlessly with TypeScript. These definitions are available from the DefinitelyTyped registry, from where we can install them.

To install them you need to use [Typings](#). It has its configuration file, which is called `typings.json`, where you need configure to specify paths for type definitions.

Introducing Angular

[AngularJS](#) was the most popular client-side framework among JavaScript developers

for many years. Google introduced AngularJS in 2012. It's based on a variation of the very popular [Model-View-Controller pattern](#) which is called Model-View-*.

The AngularJS framework was built on top of JavaScript to decouple the business logic of an application from the low-level DOM manipulation and create dynamic websites. Developers could use it to either create full-fledged SPAs and rich web applications or simply control a portion of a web page which makes it suitable for different scenarios and developer requirements.

Data Binding

Among the powerful concepts that were introduced by AngularJS, is the concept of [data binding](#) which enables the view to get automatically updated whenever the data (the model layer) is changed and vice versa.

Directives

The concept of [directives](#) was also introduced by AngularJS, which allows developers to create their own custom HTML tags i.e extend HTML.

Dependency Injection

Another concept is [Dependency Injection](#), which allows developers to inject services (singletons that encapsulate a unique and re-usable functionality within an application) into other components. This encourages reusability of the code.

Angular Concepts

Angular is a component-based framework with many new concepts that encourage DRY and separation of concerns principles. In this section, we'll briefly explain the most commonly used concepts in Angular.

Components

Components are the basic building of an Angular 7 application. A component controls a part of the app's UI. It's encapsulated and reusable.

You can create a component by creating a TypeScript class and decorate with the `@Component` decorator available from the Angular core package (`@angular/core`)

A component's view is built using a unique HTML template associated with the component's class and also a stylesheet file that's used to style the HTML view.

This is an example of an Angular component:

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.css']
7 })
8 export class AppComponent {
9   title = 'angular7-router-demo';
10 }
```

We start by importing `Component` from the Angular Core package and we use it to decorate a TypeScript class.

The `@Component` decorator takes some meta-information about the component:

- `selector`: It's used to call the component from an HTML template e.g. `<app-component></app-component>` just like any other HTML tag.
- `templateUrl`: It's used to specify the relative path to an HTML file that will be used as the component's template
- `styleUrls`: It's an array that specifies one or more stylesheets that can be used to style the component's view.

An Angular's component has a life-cycle from its creation to destruction. There are many events that you can listen to for executing code at these events.

Services

Angular services are singleton TypeScript classes that have only one instance throughout the app and its lifetime. They provide methods that maintain data from the start of the application to its end.

A service is used to encapsulate the business logic that can be repeated in many areas of your code. This helps the developers to follow the DRY (Don't Repeat Yourself) software concept.

The service can be called by components and even other services in the app. It's injected in the component's constructor via Dependency Injection.

Services are used to achieve DRY and separation of concerns into an Angular application. Along with components, they help make the application into reusable and maintainable pieces of code that can be separated and even used throughout other apps.

Let's suppose that your application has many components that need to fetch data from a remote HTTP resource.

If you are making an HTTP call to fetch the remote resource from a server in your component. This means that each component is repeating a similar code for getting the same resource. Instead, you can use a service that encapsulates the part of the code that only deals with fetching the remote resources (The server address and the specific resource to fetch can be passed via parameters to a service method). Then we can simply inject the service wherever we want to call the fetching logic. This is what's called **Separation of Concerns** that states that components are not responsible for doing specific tasks (in our case fetch data), instead a service can do the task and pass data back to the components.

Angular Libraries

Angular provides many libraries out of the box. Let's see the most important ones:

HttpClient

Angular has its powerful HTTP client that can be used to make calls to remote API servers so you don't need to use external libraries like Axios for example or even the standard Fetch API. The HttpClient is based on the old [XMLHttpRequest](#) interface available in all major browsers.

HttpClient is an Angular Service that's available from the `@angular/common/http`.

Angular Router

The Angular router is a powerful client-side routing library that allows you to add build SPAs or Single Page Apps. It provides advanced features such as multiple router outlets, auxiliary paths, and nested routing.

Angular 7 didn't add many features to the router, except for some warnings that notify the user when routing is activated outside the zone.

Angular Forms

Angular provides developers with powerful APIs to create and work with forms and two approaches that you can choose from when you are dealing with forms: **Template-based** and **model-based or reactive** forms.

Again Angular 7, didn't add any features to the forms APIs.

Angular Material

[Angular Material](#) is a modern UI library based on Google's Material Design spec which provides common internationalized and themable UI components that work across the web, mobile, and desktop. It's built by the Angular team and integrate well with Angular ecosystem.

In Angular 7 (and v6 also), you can use CLI `ng add` command for quickly add the required dependencies and configure Material into your project:

```
1 ng add @angular/material
```

Angular 7 has added new features to this library including drag and drop support so you don't need to use an external library anymore and also virtual scrolling which allows you to efficiently scroll large set of items without performance issues, particularly on mobile devices.

Why Would you Use Angular

Angular is an open-source and TypeScript-based platform for building client-side web applications as Single Page Applications. Angular provides features such as declarative templates, dependency injection and best patterns to solve everyday development problems. But precisely, why Angular? Because:

- It provides support for most platform and web browsers such as web, mobile, and desktop.
- It's powerful and modern with a complete ecosystem,
- It can be used to develop native mobile apps with frameworks such as NativeScript and Ionic
- It's convenient and can be used with Electron to develop native desktop apps etc.
- Angular provides you with the tools and also with powerful software design patterns to easily manage your project.
- It's using TypeScript instead of plain JavaScript, a strongly typed and OOP-based language created by Microsoft which provides features like strong types, classes, interfaces, and even decorators, etc.
- It's batteries-included which means you don't have to look for a separate tool for different tasks. With Angular, you have built-in libraries for routing, forms, and HTTP calls, etc. You have templates with powerful directives and pipes. You can use the forms APIs to easily create, manipulate and validate forms.
- Angular uses RxJS which is a powerful reactive library for working with Observables.
- Angular is a component-based framework which means decoupled and re-usable components are the basic building of your application.
- In Angular DOM, manipulation is abstracted with a set of powerful APIs.
- Angular is a powerful framework that can be also used to build PWAs (Progressive Web Apps).

Getting Started with Angular

Now let's see how we can start using the latest Angular 7 version.

Prior knowledge of Angular is not required for this tutorial series but you'll need to have a few requirements:

- Prior working experience or understanding of HTML and CSS.
- Familiarity with TypeScript/JavaScript.

Using the GitHub Repository To Generate a Project

You can clone a quick-start Angular project from GitHub to generate a new project. If you have Git installed on your system, simply run the following command:

```
1 git clone https://github.com/angular/quickstart my-angular-project
2 cd my-angular-project
3 npm install
4 npm start
```

You can find more information [here](#).

In the next chapter, we'll use the Angular CLI to generate our Angular front-end project. It's also the recommended way by the Angular team.

Generating a New Angular Project with Angular CLI

Developers can use different ways to start a new project; such as:

- Installing Angular by hand in a new project generated with **npm init**,
- Installing and using the CLI to generate a new project,
- Upgrading from an existing Angular project or any previous version (refer to the sections on top for more information).
- The best way though is using the Angular CLI which is recommended by the Angular team. A project generated via the CLI has many features and tools built-in like testing for example which makes easy to start developing enterprise-grade apps in no time and without dealing with complex configurations and tools like Webpack.

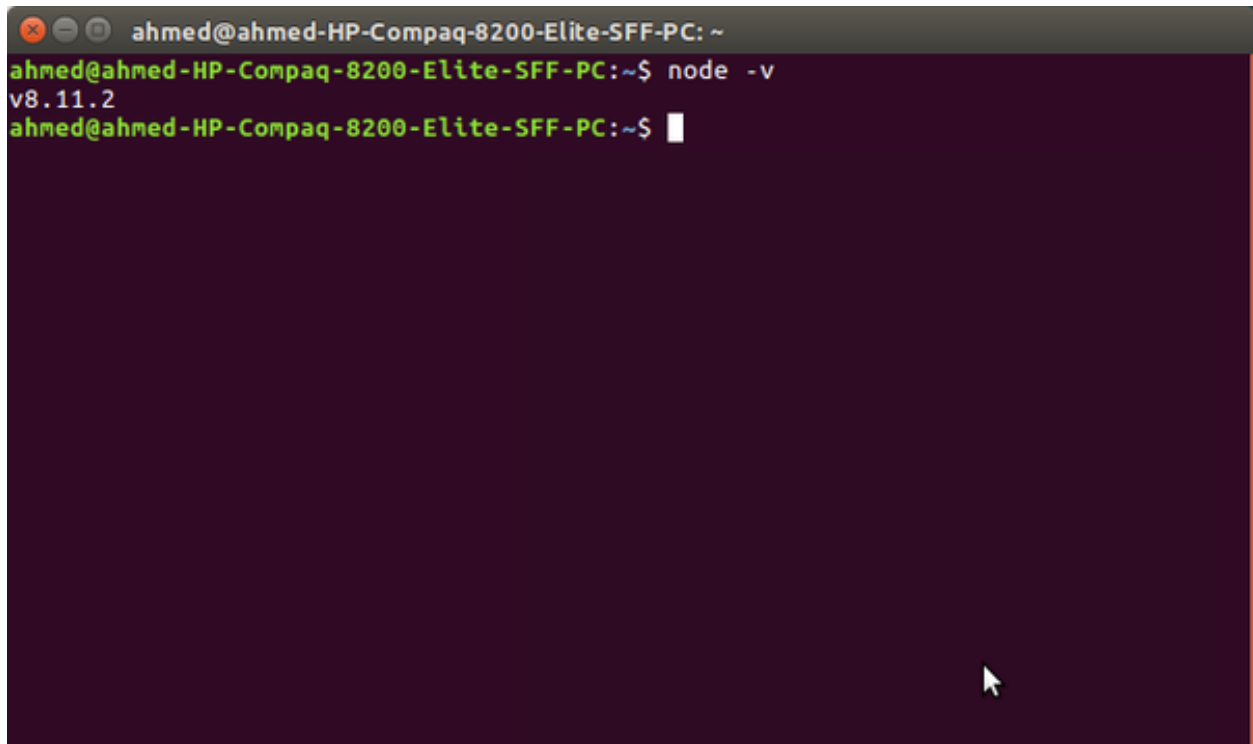
Requirements

This chapter has a few requirements. Angular CLI depends on Node.js so you need to have Node and NPM—Node 8.9 or higher, together with NPM 5.5.1—installed on your development machine. The easy way is to go [their official website](#) and get the appropriate installer for your operating system.

Now, just to make sure you have Node.js installed, open a terminal and run the following command:

```
1 node -v
```

You should get the version of the installed **Node.js 8.9+** platform.

A terminal window with a dark purple background and green text. The window title is 'ahmed@ahmed-HP-Compaq-8200-Elite-SFF-PC: ~'. The prompt is 'ahmed@ahmed-HP-Compaq-8200-Elite-SFF-PC:~\$'. The user has entered 'node -v' and the output is 'v8.11.2'. The prompt is now 'ahmed@ahmed-HP-Compaq-8200-Elite-SFF-PC:~\$' with a cursor.

```
ahmed@ahmed-HP-Compaq-8200-Elite-SFF-PC: ~
ahmed@ahmed-HP-Compaq-8200-Elite-SFF-PC:~$ node -v
v8.11.2
ahmed@ahmed-HP-Compaq-8200-Elite-SFF-PC:~$
```

Angular 7 tutorial - node version

Node version ~8.9+

Installing Angular CLI

The Angular CLI is a powerful command-line utility built by the Angular team to make it easy for developers to generate Angular projects without dealing with the complex Webpack configurations or any other tool. It provides a fully-featured tool for working with your project from generating artifacts like components, pipes, and services to serving and building production-ready bundles.

To use the Angular CLI—you first need to install it via the npm package manager. Head over to your terminal and enter the following command:

```
1 $ npm install -g @angular/cli
```

Note: Depending on your npm configuration, you may need to add **sudo** to install global packages.

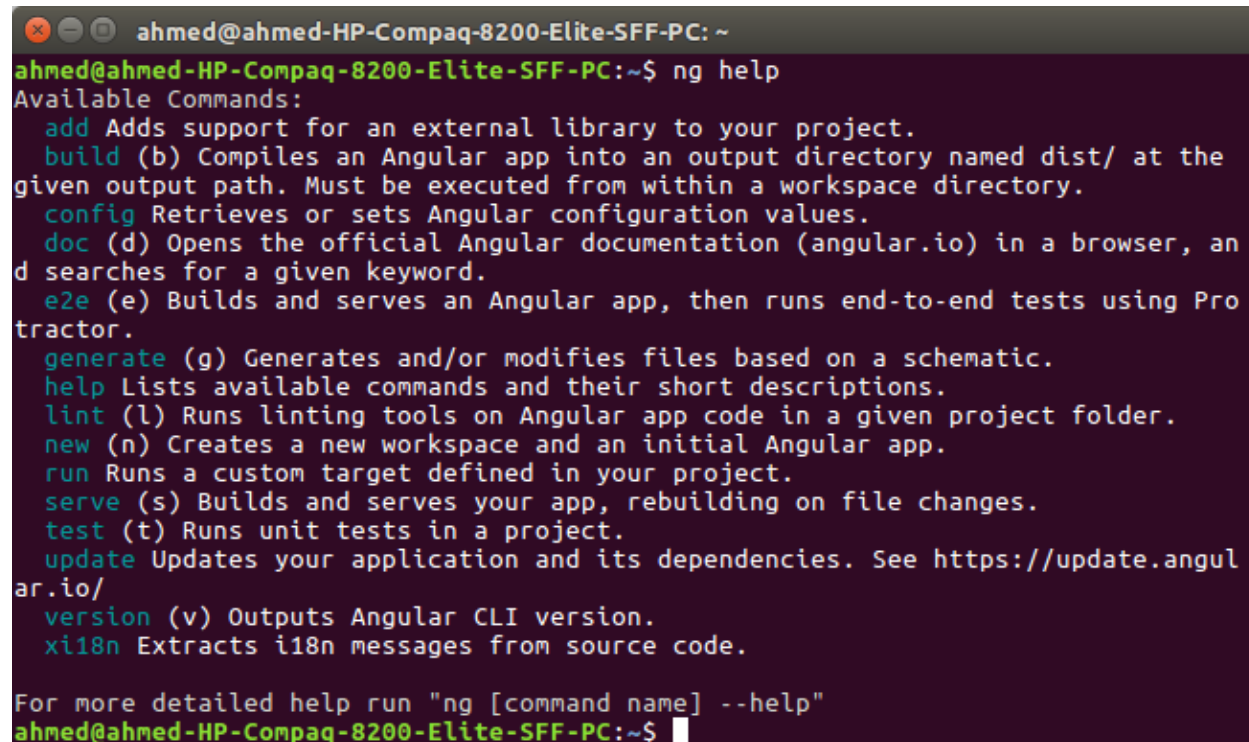
A Primer on Angular CLI

After installing Angular CLI, you'll have various commands at your disposal. Let's start by checking the version of the installed CLI:

```
1 $ ng version
```

A second command that you might need to run is the **help** command to get a complete usage help:

```
1 $ ng help
```



```
ahmed@ahmed-HP-Compaq-8200-Elite-SFF-PC: ~
ahmed@ahmed-HP-Compaq-8200-Elite-SFF-PC:~$ ng help
Available Commands:
  add Adds support for an external library to your project.
  build (b) Compiles an Angular app into an output directory named dist/ at the
given output path. Must be executed from within a workspace directory.
  config Retrieves or sets Angular configuration values.
  doc (d) Opens the official Angular documentation (angular.io) in a browser, an
d searches for a given keyword.
  e2e (e) Builds and serves an Angular app, then runs end-to-end tests using Pro
tractor.
  generate (g) Generates and/or modifies files based on a schematic.
  help Lists available commands and their short descriptions.
  lint (l) Runs linting tools on Angular app code in a given project folder.
  new (n) Creates a new workspace and an initial Angular app.
  run Runs a custom target defined in your project.
  serve (s) Builds and serves your app, rebuilding on file changes.
  test (t) Runs unit tests in a project.
  update Updates your application and its dependencies. See https://update.angular.io/
  version (v) Outputs Angular CLI version.
  xi18n Extracts i18n messages from source code.

For more detailed help run "ng [command name] --help"
ahmed@ahmed-HP-Compaq-8200-Elite-SFF-PC:~$
```

Angular 7 tutorial - CLI help

Angular CLI Usage ~ ng help

The CLI provides the following commands:

- **add**: Adds support for an external library to your project.
- **build (b)**: Compiles an Angular app into an output directory named `dist/` at the given output path. Must be executed from within a workspace directory.

- `config`: Retrieves or sets Angular configuration values.
- `doc (d)`: Opens the official Angular documentation (angular.io) in a browser, and searches for a given keyword.
- `e2e (e)`: Builds and serves an Angular app, then runs end-to-end tests using Protractor.
- `generate (g)`: Generates and/or modifies files based on a schematic.
- `help`: Lists available commands and their short descriptions.
- `lint (l)`: Runs linting tools on Angular app code in a given project folder.
- `new (n)`: Creates a new workspace and an initial Angular app.
- `run`: Runs a custom target defined in your project.
- `serve (s)`: Builds and serves your app, rebuilding on file changes.
- `test (t)`: Runs unit tests in a project.
- `update`: Updates your application and its dependencies. See <https://update.angular.io/>
- `version (v)`: Outputs Angular CLI version.
- `xi18n`: Extracts i18n messages from source code.

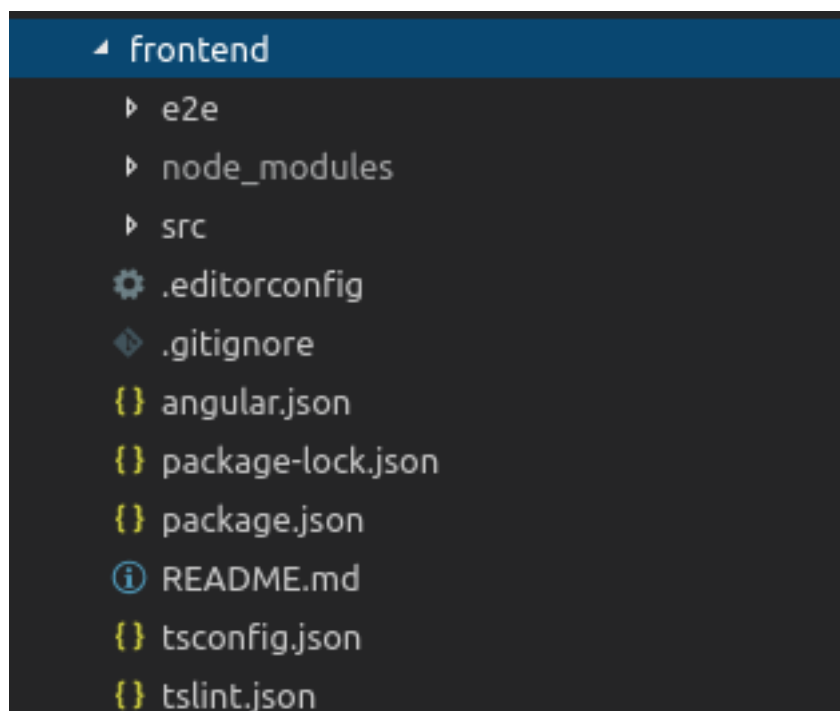
Angular CLI — Generating a New Project from Scratch

You can use Angular CLI to quickly generate your Angular project by running the following command in your terminal:

```
1 $ ng new frontend
```

frontend is the name of the project. You can obviously choose any valid name for your project. I'm using *frontend* as a name for the front-end application.

As mentioned earlier, the CLI will prompt you if *Would you like to add Angular routing?*, you can answer by *y* (Yes) or *No* which is the default option. It will also ask you for the stylesheet format, you want to use (such as *CSS*). Choose your options and hit *Enter* to continue.



Angular 7 project structure

After that, you'll have your project created with the directory structure and a bunch of configurations and source code files, mostly in TypeScript and JSON format. Let's see the role of each file:

- `/e2e/`: This folder contains end-to-end (simulating user behavior) tests of the website.
- `/node_modules/`: All 3rd party libraries are installed to this folder using `npm install`.
- `/src/`: It contains the source code of the application. Most work will be done here.
 - `/app/`: It contains modules and components.
 - `/assets/`: It contains static assets like images, icons, and styles etc.
 - `/environments/`: It contains the environment (production and development) specific configuration files.
 - `browserslist`: Needed by autoprefixer for CSS support.
 - `favicon.ico`: The favicon.
 - `index.html`: The main HTML file.
 - `karma.conf.js`: The configuration file for Karma (a testing tool)
 - `main.ts`: The main starting file from where the `AppModule` is bootstrapped.
 - `polyfills.ts`: Polyfills needed by Angular.
 - `styles.css`: The global stylesheet file for the project.
 - `test.ts`: This is a configuration file for Karma

- `tsconfig.*.json`: The configuration files for TypeScript.
- `angular.json`: It contains the configurations for CLI
- `package.json`: It contains the basic information of the project (name, description and dependencies, etc.)
- `README.md`: A Markdown file that contains a description of the project.
- `tsconfig.json`: The configuration file for TypeScript.
- `tslint.json`: The configuration file for TSLint (a static analysis tool)

Angular CLI — Serving your Project with a Development Server

Angular CLI provides a complete tool-chain for developing front-end apps on your local machine. As such, you don't need to install a local server to serve your project—you can simply, use the `ng serve` from your terminal to serve your project locally.

First, navigate inside your project's folder and run the following commands:

```
1 $ cd frontend
2 $ ng serve
```

You can now navigate to the `http://localhost:4200/` address to start playing with your front-end application. The page will automatically live-reload if you change any of the source files.

You can also use a different host address and port other than the default ones by using the `--host` and `--port` switches. For example:

```
1 $ ng serve --host 0.0.0.0 --port 8080
```

Angular CLI — Generating Components, Directives, Pipes, Services and Modules

To bootstrap your productivity, Angular CLI provides a **generate** command for generating components, directives, pipes, services, modules, and other artifacts. For example to generate a component run the following command:

```
1 $ ng generate component account-list
```

`account-list` is the name of the component. You can also use just **g** instead of **generate**. The Angular CLI will automatically add a reference to components, directives and pipes in the `src/app/app.module.ts` file.

If you want to add your component, directive or pipe to another module, other than the main application module, you can simply prefix the name of the component with the module name and a slash like a path:

```
1 $ ng g component account-module/account-list
```

`account-module` is the name of an existing module

Conclusion

Thanks to **Angular CLI**, you can get started with Angular by quickly generating a new project with a variety of flags to customize and control the generation process. As a recap, we have seen various ways to create a new Angular project.

We have also seen some of the important new features of all Angular versions up to v8 and the basic concepts of Angular.

In the next chapter, you'll start building your first Angular web application.

Building a Calculator App with Angular

In this chapter, you will learn to build your first Angular application from scratch.

You'll get started with the basic concepts like modules, components, and directives. You'll also learn about event and property bindings. We'll be building a simple calculator application to demonstrate all these concepts.

Prerequisites

Let's get started with the prerequisites. You will need to have:

- Basic knowledge of TypeScript, HTML, and CSS.
- Node and NPM installed on your system.
- Angular CLI installed on your system.

Generating our Angular Calculator Project

After setting up your environment by installing Node and Angular CLI in your system, let's create a project. So open a terminal and execute the following command:

```
1 $ ng new ngcalculator
```

The CLI will ask you if you would like to add routing to your project - You can say **No** as we will not need routing in this demo. For the stylesheets format, choose **CSS**.

You can then wait for the CLI to generate your project and install the required dependencies from npm.

After that, you can start a live development server using the `ng serve` command:

```
1 $ cd ./ngcalculator
2 $ ng serve
```

Wait for the CLI to compile your project and start the server at `http://localhost:4200`.

Angular Modules & Components

Angular follows a **modular** and **component-based** architecture. In fact, these are two separate concepts:

- The modular architecture in which you build your application as a set of modules. The general rule is that you need to use a module for each feature of your application.
- The component-based architecture in which, you build your application as a set of components.

Note: In our Angular project generated with the CLI, we already have a root module which is conventionally called `AppModule` and a root component which is conventionally called `AppComponent`.

The root module and component are the first things bootstrapped by the Angular application (In the `main.js` and `app.module.ts` files)

According to the [Angular docs](#), this is the definition of a module:

Angular apps are modular and Angular has its own modularity system called `NgModules`. `NgModules` are containers for a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. They can contain components, service providers, and other code files whose scope is defined by the containing `NgModule`. They can import functionality that is exported from other `NgModules`, and export selected functionality for use by other `NgModules`.

Since we are building a simple calculator application we don't need more than one module, so let's keep it simple and use the root module for implementing our feature.

Note: You can generate a new module using the CLI: `ng generate module <name>`.

What About Components?

A component controls a part of the screen. It's simply a TypeScript class (decorated with `@Component`) with an HTML template that displays the view.

We also don't need many components in our calculator app. Let's create one component for encapsulating the calculator view and logic.

Open a new terminal, navigate to your project's folder and run the following command:

```
1 $ ng generate component calculator --skipTests
```

We added the `--skipTests` option to tell the CLI to skip generating a file for component tests as we will not add any tests in this tutorial.

The CLI has generated the following files in the `src/app/calculator` folder:

- `src/app/calculator/calculator.component.css` for CSS styles.
- `src/app/calculator/calculator.component.html` for the component's template or the view.
- `src/app/calculator/calculator.component.ts` for the component logic.

Open the `src/app/calculator/calculator.component.ts` file:

```
1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-calculator',
5   templateUrl: './calculator.component.html',
6   styleUrls: ['./calculator.component.css']
7 })
8 export class CalculatorComponent implements OnInit {
9
10  constructor() { }
11
12  ngOnInit() {
13  }
14
15 }
```

The `CalculatorComponent` class knows about the template and CSS file thanks to the `@Component()` decorator which takes the following metadata:

- `selector` that allows us to give the component a tag name that can be used to reference the component from other templates just like standard HTML tags.
- `templateUrl` that points to the HTML template that renders the view of the component. You can also use an inline template with the `template` property instead.
- `styleUrls` that allows us to associate one or multiple stylesheets to our component.

Since we didn't include routing in our application, we need a way to access our calculator component from our root component. That's where the `selector` property of the component comes handy - we can call our calculator component using the `<app-calculator>` tag. Open the `src/app/app.component.html` file, remove the existing content and add:

```
1 <app-calculator></app-calculator>
```

Creating our Calculator UI

We'll be using the HTML and CSS code from [this JS fiddle](#) to create our calculator UI.

Open the `src/app/calculator/calculator.component.html` file and add the following code:

```
1 <div class="calculator">
2
3   <input type="text" class="calculator-screen" value="0" disabled />
4
5   <div class="calculator-keys">
6
7     <button type="button" class="operator" value="+">+</button>
8     <button type="button" class="operator" value="-">-</button>
9     <button type="button" class="operator" value="*">&times;</button>
10    <button type="button" class="operator" value="/">&divide;</button>
11
12    <button type="button" value="7">7</button>
13    <button type="button" value="8">8</button>
```

```
14     <button type="button" value="9">9</button>
15
16
17     <button type="button" value="4">4</button>
18     <button type="button" value="5">5</button>
19     <button type="button" value="6">6</button>
20
21
22     <button type="button" value="1">1</button>
23     <button type="button" value="2">2</button>
24     <button type="button" value="3">3</button>
25
26
27     <button type="button" value="0">0</button>
28     <button type="button" class="decimal" value=".">.</button>
29     <button type="button" class="all-clear" value="all-clear">AC</button>
30
31     <button type="button" class="equal-sign" value="=">=</button>
32
33 </div>
34 </div>
```

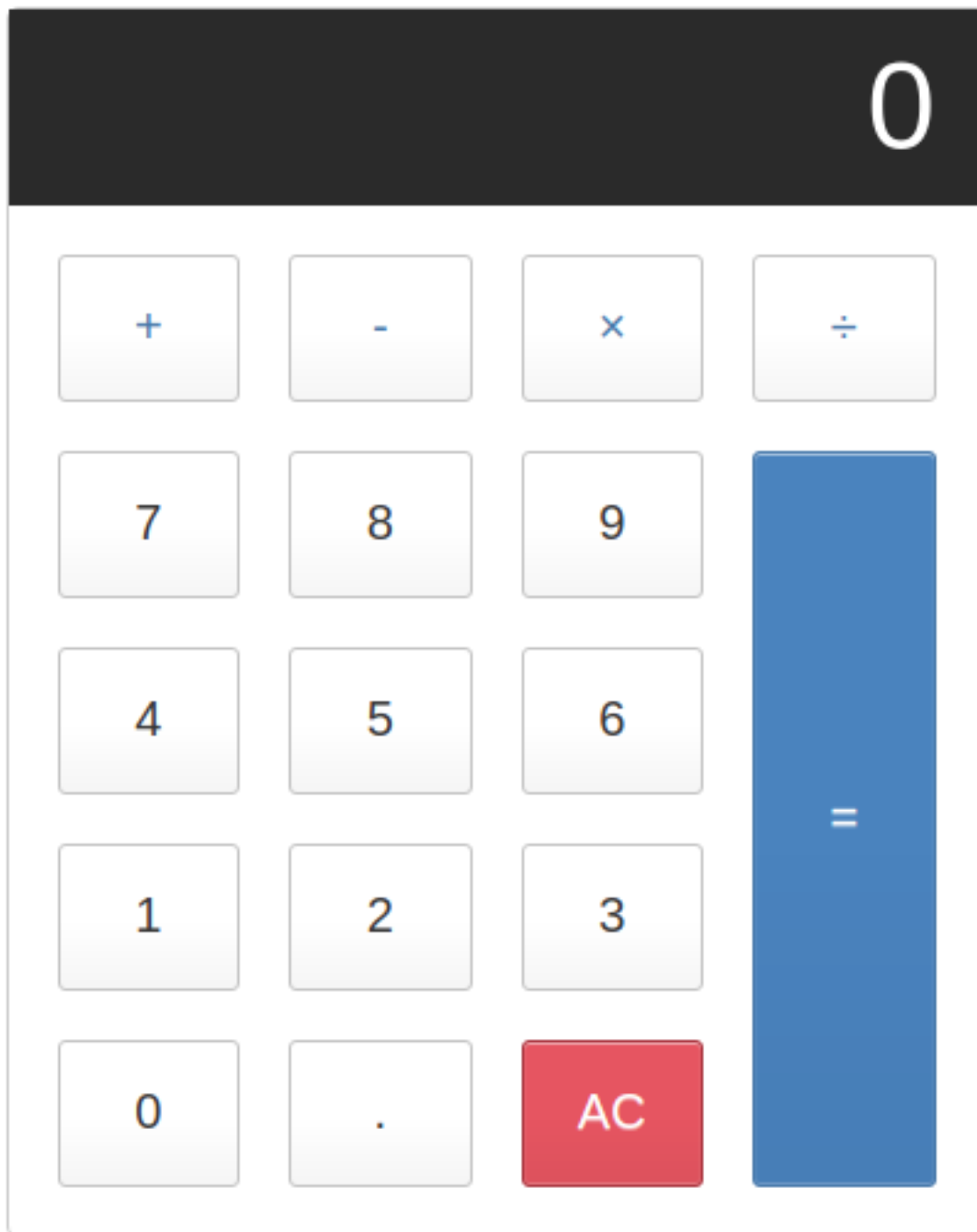
Next, open the `src/app/calculator/calculator.component.css` file and add the following CSS styles:

```
<script src="https://gist.github.com/techiediaries/c5f68e66acbca784cb2350863aa4e4f0.js"></script>
```

We also need to add some global styling, so open the `src/styles.css` file and add:

```
1  html {
2      font-size: 62.5%;
3      box-sizing: border-box;
4  }
5
6  *, *::before, *::after {
7      margin: 0;
8      padding: 0;
9      box-sizing: inherit;
10 }
```

Now if you go to your web browser and navigate to your app, you should see the following interface:



Angular 8 Example Calculator

Now, we need to use the Angular magic to turn this template to a working calculator.

Up until now, our template is just plain HTML but Angular provides other constructs that we can use in the templates such as data binding (interpolation, event and property binding).

Simply put, data binding is a fundamental concept in Angular that allows developers to make communication between a component and its view or more precisely the DOM. This way you don't need to manually push data from your component to the DOM and back.

Angular provides four types of data binding and they are essentially different in the way data flows i.e from the component to the DOM, from the DOM to the component or both ways:

- **Interpolation:** Data flows from the component to the DOM - It's used to display the value of a component member variable in the associated template, e.g. `{{ foobar }}`. We use curly braces for interpolation.
- **Property binding:** Data flows from the component to a property of an element in the DOM. It's used to bind a component member variable to an attribute of a DOM such as the value attribute of an `<input>` tag (For example: `<input type="text" [value]="foobar">`). We use brackets for property binding.
- **Event binding:** Data flows from the DOM to the component. When a DOM event, such as a `click`, is triggered, the bound method from the component is called. For example: `<button (click)="sayHi()">Hi</button>` - The `sayHi()` method will be called so it needs to be defined in the component class. We use parentheses for event binding.
- **Two-way data binding:** Data flows both ways. For example: `<input type="text" [(ngModel)]="foobar">` (The `foobar` variable needs to be defined in the component). The input element and `foobar` will have the same value and when one changes, the other one changes to the same value accordingly. We use the banana in the box syntax which combines brackets and parentheses for two-way data binding. `ngModel` is a special directive that binds to the `value` attribute of the `<input>` and `<textarea>` elements but you can construct two-way data binding for any property in the DOM or component. Two-way data binding = property binding + event binding.

Equipped with this information, let's implement our calculator application.

We have DOM elements for operations and numbers and where to display the result of the operation(s).

We need to get the value of the number or the type of the operation when the user clicks on the corresponding DOM element, calculate the result and display it in the results element.

In the template, we have four sets of keys:

- digits (0-9),
- operators (+, -, *, /, =),
- a decimal point (.)
- and a reset key.

Let's see how we can use Angular to listen for clicks on the calculator and determine what type of key was pressed.

Open the `src/app/calculator/calculator.component.ts` file and start by defining the following member variables of the component:

```
1 export class CalculatorComponent implements OnInit {  
2  
3   currentNumber = '0';  
4   firstOperand = null;  
5   operator = null;  
6   waitForSecondNumber = false;
```

- The `currentNumber` variable holds the string that will be displayed in the result input element.
- The `firstOperand` variable holds the value of the first operand of the operation.
- The `operator` variable holds the operation.
- The `waitForSecondNumber` variable holds a boolean value indicating if the user has finished typing the first operand and ready to enter the second operand of the operation.

Next, define the `getNumber()` method that will be used to get the current number:

```
1 public getNumber(v: string){  
2   console.log(v);  
3   if(this.waitForSecondNumber)  
4   {  
5     this.currentNumber = v;  
6     this.waitForSecondNumber = false;  
7   }else{  
8     this.currentNumber === '0'? this.currentNumber = v: this.currentNumber += v;  
9  
10  }  
11 }
```

Next, define the `getDecimal()` method which appends the decimal point to the current number:

```
1  getDecimal(){
2      if(!this.currentNumber.includes('.')){
3          this.currentNumber += '.';
4      }
5  }
```

Next, define the `doCalculation()` method which performs the calculation depending on the operator type:

```
1  private doCalculation(op , secondOp){
2      switch (op){
3          case '+':
4              return this.firstOperand += secondOp;
5          case '-':
6              return this.firstOperand -= secondOp;
7          case '*':
8              return this.firstOperand *= secondOp;
9          case '/':
10             return this.firstOperand /= secondOp;
11         case '=':
12             return secondOp;
13     }
14 }
```

Next, define the `getOperation()` that will be used to get the performed operation:

```
1  public getOperation(op: string){
2      console.log(op);
3
4      if(this.firstOperand === null){
5          this.firstOperand = Number(this.currentNumber);
6      }
7      else if(this.operator){
8          const result = this.doCalculation(this.operator , Number(this.currentNumber))
9          this.currentNumber = String(result);
10         this.firstOperand = result;
11     }
12     this.operator = op;
```

```
13     this.waitForSecondNumber = true;
14
15     console.log(this.firstOperand);
16
17 }
```

Finally, define the `clear()` method that will be used to clear the result area and reset the calculations:

```
1  public clear(){
2      this.currentNumber = '0';
3      this.firstOperand = null;
4      this.operator = null;
5      this.waitForSecondNumber = false;
6  }
```

Now, you need to use data binding to bind these methods to the template.

Angular Property Binding by Example

We have defined variables and methods in the component. Now, we'll need to bind them to the template.

Let's start with the `currentNumber` variable which holds the value of the currently typed number. Let's use property binding to bind `currentNumber` to the `value` attribute of the `<input>` element as follows:

```
1  <div class="calculator">
2
3      <input type="text" class="calculator-screen" [value]="currentNumber" disabled />
4
5      <!-- [...] -->
```

We use brackets around the `value` attribute to create a property binding.

Now, our current number will be displayed in our calculator and when the value of the `currentNumber` variable changes, the displayed value will change accordingly without having to manually add any code to update the DOM.

Angular Event Binding by Example

Next, when a digit button is clicked we need to call the `getNumber()` method to append the digit to the current number string. For this, we can use Angular event binding to bind the `getNumber()` method to the `click` event of buttons displaying the digits. Change your component template as follows:

```
1 <div class="calculator">
2
3   <input type="text" class="calculator-screen" [value]="currentNumber" disabled />
4
5   <div class="calculator-keys">
6
7     <!-- [...] -->
8
9     <button type="button" (click) = "getNumber('7')" value="7">7</button>
10    <button type="button" (click) = "getNumber('8')" value="8">8</button>
11    <button type="button" (click) = "getNumber('9')" value="9">9</button>
12
13
14    <button type="button" (click) = "getNumber('4')" value="4">4</button>
15    <button type="button" (click) = "getNumber('5')" value="5">5</button>
16    <button type="button" (click) = "getNumber('6')" value="6">6</button>
17
18
19    <button type="button" (click) = "getNumber('1')" value="1">1</button>
20    <button type="button" (click) = "getNumber('2')" value="2">2</button>
21    <button type="button" (click) = "getNumber('3')" value="3">3</button>
22
23
24    <button type="button" (click) = "getNumber('0')" value="0">0</button>
25    <!-- [...] -->
26  </div>
27 </div>
```

We use the parentheses around the `click` event to create an event binding.

Next, let's bind the `getOperation()`, `getDecimal()` and `clear()` methods to the `click` event of their respective buttons:

```
1   <div class="calculator-keys">
2
3       <button type="button" (click) = "getOperation('+')" class="operator" value="+">+\
4   </button>
5       <button type="button" (click) = "getOperation('-')" class="operator" value="-">-\
6   </button>
7       <button type="button" (click) = "getOperation('*')" class="operator" value="*">&\
8   times;</button>
9       <button type="button" (click) = "getOperation('/')" class="operator" value="/">&\
10  divide;</button>
11
12       <!-- [...] -->
13
14       <button type="button" (click) = "getDecimal()" class="decimal" value=".">.</butt\
15   on>
16       <button type="button" (click) = "clear()" class="all-clear" value="all-clear">A\
17   C</button>
18
19       <button type="button" (click) = "getOperation('=')" class="equal-sign" value="=">\
20   >=</button>
21
22   </div>
23 </div>
```

That's it our calculator is ready!

Conclusion

In this chapter, we've built a simple calculator application with Angular. We've learned about the types of data bindings and we've seen examples of event and property binding in Angular.

Build your First Angular Web Application with Firebase and Bootstrap

In this chapter, we'll be covering Angular from scratch with routing and navigation by building a complete example.

You will be building a simple web application that you can use to show your portfolio projects and which you can host in the web and make it accessible to your potential clients. You'll use Firebase for authentication, Firestore for storing and fetching the projects and Bootstrap 4 for styling the UI.

You'll learn about the prerequisites for working with Angular, how to install Angular CLI and use it to create your project and the various artifacts that you'll need throughout the development of your project.

You'll learn how to work with route parameters, child routes, and nested routing.

You'll learn about the `router-outlet` component, used by the Angular router to render the component(s) matching the current path. How to navigate using the `routerLink` and `routerLinkActive` directives or also the `router.navigate()` method.

You'll also learn about the router events, animations and the router `resolve` property for fetching data.

You'll also learn how to create Angular modules to organize your code, create components for controlling various UI parts, add CRUD operations and style the UI with Bootstrap 4.

Setting up your Project

You'll start the tutorial by installing Angular CLI, the official CLI for creating and working with Angular projects and workspaces, next you'll generate your project based on the latest version as of this writing, next you'll proceed to create your app's pages and components.

After that, you'll set up Bootstrap 4 in your project and style your Angular components with the Bootstrap UI components. Finally, you'll set up routing in your project and add navigation between the various components composing your application.

Prerequisites

To follow along with this chapter, you need a few prerequisites, such as:

- Recent versions of Node.js and NPM installed on your system,
- Working knowledge of JavaScript and TypeScript.

If you have these requirements, you are good to go!

Installing Angular CLI

Let's start by installing Angular CLI from npm using the following command:

```
1 $ npm install @angular/cli -g
```

As of the time of this writing, this will install **angular cli v8** globally on your system.

Note: You may need to use an elevated administrator CMD in Windows or add **sudo** to your command in macOS and Debian based systems if you want to install packages globally. If you want to install packages without the admin access, you simply need to fix your npm permissions.

Creating an Angular Project

We'll start with the first step which is creating a new Angular project using the Angular CLI. Head over to your terminal and run the following command:

```
1 $ ng new angular-portfolio
```


Now, you need to pay attention to this step – The Angular CLI will prompt you if you **Would like to add Angular routing?** Answer by **y** (**No** is the default option) to tell the CLI to generate the necessary files for routing like the `src/app/app-routing.module.ts` file with all the required code for setting up the router. It will also add a `<router-outlet>` component inside the `src/app/app.component.html` file which is where the router renders any matched component(s) depending on the current path.

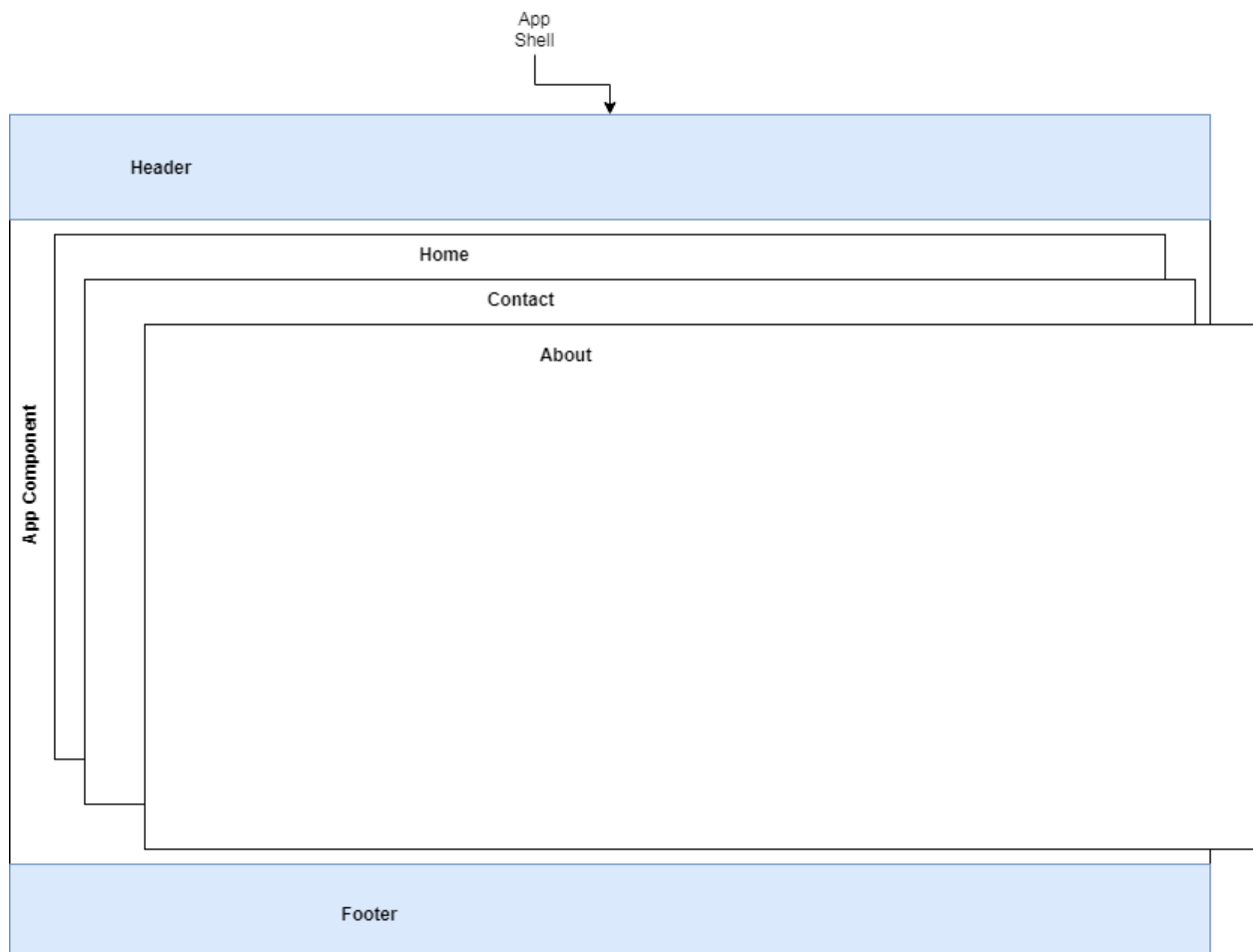
Note: The Angular CLI will also prompt you to choose the stylesheets format you want to use in your projects such as CSS, SCSS, Less, Stylus and Sass. You can simply choose **CSS** if you prefer no other format.

That's it. If you manage to pass this step without any errors you will have an Angular project set up with best practices and routing all done automatically by the CLI on your behalf.

Creating Angular Components

Angular uses components everywhere so to have multiple pages, you need to create components. You can also use components for specific parts of your page(s) that can be unique in each page or common between multiple pages.

For example, we could have a **home**, **about** and **contact** components that represent the corresponding pages and **header** and **footer** components that are common between all the other components so we'll create them once and have Angular load them all the time whatever the current path is by simply adding them to our **application shell** which is simply the main application component with the associated `src/app/app.component.html` template that hosts `<router-outlet>`:



Head back to your terminal, navigate inside your project's root folder and run these commands to create the components:

```
1 $ cd angular-portfolio
2 $ ng g c header
3 $ ng g c footer
4 $ ng g c home
5 $ ng g c about
6 $ ng g c contact
```

The CLI will generate the files for these components and the minimal required code for declaring each component and its corresponding template.

For example, for the header component, the CLI will create the `src/app/header/header.component.html`, `src/app/header/header.component.spec.ts`, `src/app/header/header.component.ts` and `src/app/header/header` files.

The components that will be created are:

- HeaderComponent in the src/app/header/ folder,
- FooterComponent in the src/app/footer/ folder,
- HomeComponent in the src/app/home/ folder,
- AboutComponent in the src/app/about/ folder,
- ContactComponent in the src/app/contact/ folder.

Those components will be imported and added in the main application module in the src/app/app.module.ts file by the CLI in your behalf.

This is the content of the src/app/app.module.ts file:

```
1  import { BrowserModule } from '@angular/platform-browser';
2  import { NgModule } from '@angular/core';
3  import { AppRoutingModule } from './app-routing.module';
4  import { AppComponent } from './app.component';
5  import { HeaderComponent } from './header/header.component';
6  import { FooterComponent } from './footer/footer.component';
7  import { HomeComponent } from './home/home.component';
8  import { ContactComponent } from './contact/contact.component';
9  import { AboutComponent } from './about/about.component';
10
11  @NgModule({
12    declarations: [
13      AppComponent,
14      HeaderComponent,
15      FooterComponent,
16      HomeComponent,
17      ContactComponent,
18      AboutComponent
19    ],
20    imports: [
21      BrowserModule,
22      AppRoutingModule
23    ],
24    providers: [],
25    bootstrap: [AppComponent]
26  })
27  export class AppModule { }
```

The previous components will be public i.e they could be accessed from any visitor of your web application but let's suppose we want to be able to list our latest portfolio projects in our home page. We want to add the portfolio items from a protected admin page that will be only accessed from the admin of the web application.

Styling our Components with Bootstrap 4

For styling our components we'll be using Bootstrap 4. The most popular CSS framework in the world. In your terminal run the following command to install Bootstrap 4 from npm:

```
1 $ npm install bootstrap --save
```

Note: This will install **bootstrap v4.2.1** as the time of this tutorial.

Open the `angular.json` file and add `./node_modules/bootstrap/dist/css/bootstrap.min.css` to the `styles` array:

```
1 "styles": [  
2   "src/styles.css",  
3   "./node_modules/bootstrap/dist/css/bootstrap.min.css"  
4 ],
```

Open the `src/app/header/header.component.html` file and create a Bootstrap 4 nav header using the following code:

```
1 <nav class="navbar navbar-expand-lg navbar-light" style="background-color: #b3cbdd\  
2 ;">  
3 <a class="navbar-brand" href="#">Angular Developer</a>  
4 <div class="collapse navbar-collapse" id="navbarText">  
5 <ul class="navbar-nav">  
6 <li class="nav-item">  
7 <a class="nav-link" href="#">Home</a>  
8 </li>  
9 <li class="nav-item">  
10 <a class="nav-link" href="#">About</a>  
11 </li>  
12 <li class="nav-item">  
13 <a class="nav-link" href="#">Contact</a>  
14 </li>  
15 <li class="nav-item">  
16 <a class="nav-link" href="#">Admin</a>  
17 </li>  
18 </ul>  
19 </div>  
20 </nav>
```

Next, you need to add the header component to the application shell. Open the `src/app/app.component.html` file and update it accordingly:

```
1 <app-header></app-header>
2 <div class="container">
3     <router-outlet></router-outlet>
4 </div>
```

You include a component in other components using the `selector` property of the `@Component` decorator:

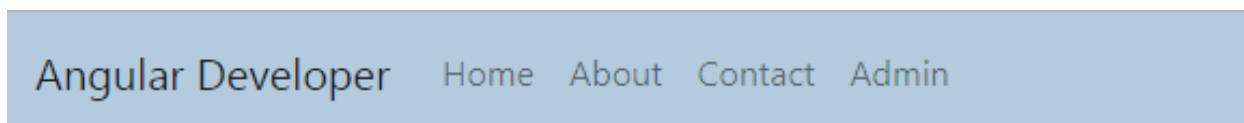
```
1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-header',
5   templateUrl: './header.component.html',
6   styleUrls: ['./header.component.css']
7 })
8 export class HeaderComponent implements OnInit {
9   constructor() { }
10  ngOnInit() {
11  }
12 }
```

At this point, we can serve our application using the following command:

```
1 $ ng serve
```

You can then see your application up and running by visiting the `localhost:4200` address in your web browser.

This is a screenshot of our header:



Angular 7 & Bootstrap 4 header

Adding Routing & Navigation

We want to load our home, about, contact and admin components when we click on the links on the navigation header.

First we need to assign the components to their corresponding paths. You need to open the `src/app-routing.module.ts` file and add the following imports:

```
1 // [...]  
2 import { HomeComponent } from './home/home.component';  
3 import { AboutComponent } from './about/about.component';  
4 import { ContactComponent } from './contact/contact.component';
```

Next in the same file, add paths to various components inside the already-declared routes array:

```
1 const routes: Routes = [  
2 {  
3 path: 'home',  
4 component: HomeComponent  
5 },  
6 {  
7 path: 'about',  
8 component: AboutComponent  
9 },  
10 {  
11 path: 'contact',  
12 component: ContactComponent  
13 }  
14 ];
```

Each object in the array defines a route. The `path` property contains the path that will be used to access a component and the `component` property contains the name of the component.

Next we need to add navigation in our header component. Open the `src/app/header/header.component.ts` file and update it with the convenient links using the `routerLink` directive:

```
1 <nav class="navbar navbar-expand-lg navbar-light" style="background-color: #b3cbdd\  
2 ;">  
3 <a class="navbar-brand" href="#">Angular Developer</a>  
4 <div class="collapse navbar-collapse" id="navbarText">  
5 <ul class="navbar-nav">  
6 <li class="nav-item">  
7 <a class="nav-link" routerLink="/home">Home</a>  
8 </li>  
9 <li class="nav-item">  
10 <a class="nav-link" routerLink="/about">About</a>  
11 </li>  
12 <li class="nav-item">  
13 <a class="nav-link" routerLink="/contact">Contact</a>  
14 </li>  
15 </ul>  
16 </div>  
17 </nav>
```

Recap

As a recap of what you achieved, you have installed Angular CLI and created a project for your developer portfolio web application, you have setup routing and Bootstrap 4 in your project then created the various pages (components that represent whole pages of your app and linked to unique routes) and components of your application and created the routes and navigation links.

In the next section, you'll learn about modules including the existing modules in your project and you'll create a feature module for encapsulating the code of the admin part of your application.

Angular Modules

Angular modules are containers of code parts that implement related domain requirements. They let developers create apps with a modular architecture and reusable code just like components. Angular uses NgModules to create modules and submodules which are different from JavaScript/ES6 modules.

What's a NgModule

NgModules are simply TypeScript classes decorated with the `@NgModule` decorator imported from the `@angular/core` package.

Modules provide a way for developers to organize their code and they are particularly helpful for managing large apps.

You can either create your modules or use the built-in modules for importing the various Angular APIs such as:

- `[FormsModule]` (<https://angular.io/api/forms/FormsModule>) for working with forms,
- `[HttpClientModule]` (<https://angular.io/api/common/http/HttpClientModule>) for sending HTTP requests,
- and `[RouterModule]` (<https://angular.io/api/router/RouterModule>) for providing routing mechanisms to your Angular application.

Each Angular module can contain components, directives, pipes, and services and may be lazy-loaded by the router.

Your Angular application has at least one module which is called the **root** module. You need to bootstrap the root module to start your application.

Creating the Admin Feature Sub-Module & CRUD Interface

Now, let's create the admin CRUD interface for listing, creating, updating and deleting the portfolio projects.

Create an admin module with four components:

- `ProjectComponent`,
- `ProjectListComponent`,
- `ProjectCreateComponent`,
- `ProjectUpdateComponent`.

First, run the following command to create a module called `admin`:

```
1 $ ng g module admin
```

This will create a `src/app/admin/admin.module.ts` file with the following content:


```
1 import { NgModule } from '@angular/core';
2 import { CommonModule } from '@angular/common';
3
4 @NgModule({
5   declarations: [],
6   imports: [
7     CommonModule
8   ]
9 })
10 export class AdminModule { }
```

Next, run the following commands to create the components inside the admin module:

```
1 $ ng g c admin/project-list
2 $ ng g c admin/project-create
3 $ ng g c admin/project-update
4 $ ng g c admin/project
```

This is the content of the `src/app/admin/admin.module.ts` file:

```
1 import { NgModule } from '@angular/core';
2 import { CommonModule } from '@angular/common';
3 import { ProjectListComponent } from '../project-list/project-list.component';
4 import { ProjectCreateComponent } from '../project-create/project-create.component';
5 import { ProjectUpdateComponent } from '../project-update/project-update.component';
6 import { ProjectComponent } from '../project/project.component';
7
8 @NgModule({
9   declarations: [ProjectListComponent, ProjectCreateComponent, ProjectUpdateComponent, \
10     ProjectComponent],
11   imports: [
12     CommonModule
13   ]
14 })
15 export class AdminModule { }
```

In the NgModule metadata, we specify:

- The components, directives, and pipes that belong to the module. In our case, the four components that we created i.e ProjectListComponent, ProjectCreateComponent, ProjectUpdateComponent and ProjectComponent.

- The components, directives, and pipes that we want to export. In our case, none.
- The modules that we need to import in our current module. In our case `CommonModule`
- The services that we need to use. In our case none.

`CommonModule` is a built in module that exports all the basic Angular directives and pipes, such as `[NgIf]` (<https://angular.io/api/common/NgIf>), `[NgForOf]` (<https://angular.io/api/common/NgForOf>), `[DecimalPipe]` (<https://angular.io/api/common/DecimalPipe>), etc.

Next, we need to import the admin module in the main module. Open the `src/app/app.module.ts` file and update it accordingly:

```
1 // [...]
2 import { AdminModule } from './admin/admin.module';
3
4 @NgModule({
5 // [...]
6 imports: [
7   BrowserModule,
8   AppRoutingModule,
9   AdminModule
10 ],
11 providers: [],
12 bootstrap: [AppComponent]
13 })
14 export class AppModule { }
```

This is the main module of our application. In the imports array, we added `AdminModule`. You can see two other arrays:

- The `providers` array which can be used to include the services we want to provide to our components,
- The `bootstrap` array which specifies the component(s) to bootstrap.

Recap

In this section, you have learned about the concept of `NgModule` in Angular, you have created the `admin` submodule of your portfolio web application and the various components of the submodule which are needed to build a CRUD interface for creating and manipulating your portfolio's projects.

In your next section, you'll be adding routing in your `admin` module using a nested router outlet and child routes.

Child & Nested Routing

In the previous section, you have seen what `NgModule` is and you created the `admin` module of your developer's portfolio web application. Now, let's add routing to our module using a routing module, a nested router-outlet and child routes.

You can create a nested routing by defining child routes using the `children` property of a route (alongside a `path` and `component` properties). You also need to add a nested `router-outlet` in the HTML template related to the component linked to the parent route (In our case it's the `admin` route).

To create nested routing, you need to create a routing submodule for the module you want to provide routing for, you next need to define a parent route and its child routes and provide them to the router configuration via a `forChild()` method.

Let's see this, step by step. First, inside the `admin` module, create an `admin-routing.module.ts` file and add a submodule for implementing child routing in our `admin` module:

```
1 import { NgModule } from '@angular/core';
2 import { Routes, RouterModule } from '@angular/router';
3
4 import { ProjectComponent } from '../project/project.component';
5 import { ProjectListComponent } from '../project-list/project-list.component';
6 import { ProjectCreateComponent } from '../project-create/project-create.component';
7 import { ProjectUpdateComponent } from '../project-update/project-update.component';
8
9 const routes: Routes = [
10 {
11   path: 'admin',
12   component: ProjectComponent,
13   children: [
14     {
15       path: 'list',
16       component: ProjectListComponent
17     },
18     {
19       path: 'create',
20       component: ProjectCreateComponent
21     },
22     {
23       path: 'update',
24       component: ProjectUpdateComponent
```

```
25 }
26 ]
27 }
28 ];
29 @NgModule({
30   imports: [RouterModule.forChild(routes)],
31   exports: [RouterModule]
32 })
33 export class AdminRoutingModule { }
```

This is an example of a module which has `imports` and `exports` meta information;

- The `imports` array which contains the modules that we need to import and use in the current module. In this case it's `RouterModule.forChild(routes)`,
- The `exports` array which contains what we need to export.

To provide our child routes to the router module, we use the `forChild()` method of the module because we want to add routing in the `admin` submodule. if this is used in the root module you need to use the `forRoot()` method instead. See more differences of `forChild()` VS `forRoot()` from the [official docs](#).

The `forChild()` and `forRoot()` methods are static methods that are used to configure modules in Angular. They are not specific to `RouterModule`.

We are creating a parent `admin` route and its child routes using the `children` property of the route which takes an array of routes.

You can respectively access the `ProjectListComponent`, `ProjectCreateComponent` and `ProjectCreateComponent` using the `/admin/list`, `/admin/create` and `/admin/update` paths.

Next, open the `src/app/admin/admin.module.ts` file and import the routing module:

```
1 // [...]
2 import { AdminRoutingModule } from './admin-routing.module';
3
4 @NgModule({
5   // [...]
6   imports: [
7     CommonModule,
8     AdminRoutingModule
9   ]
10 })
11 export class AdminModule { }
```

Next open the `src/app/admin/project/project.component.html` file and add a nested router outlet:

```
1 <h2>Admin Interface</h2>
2 <router-outlet></router-outlet>
```

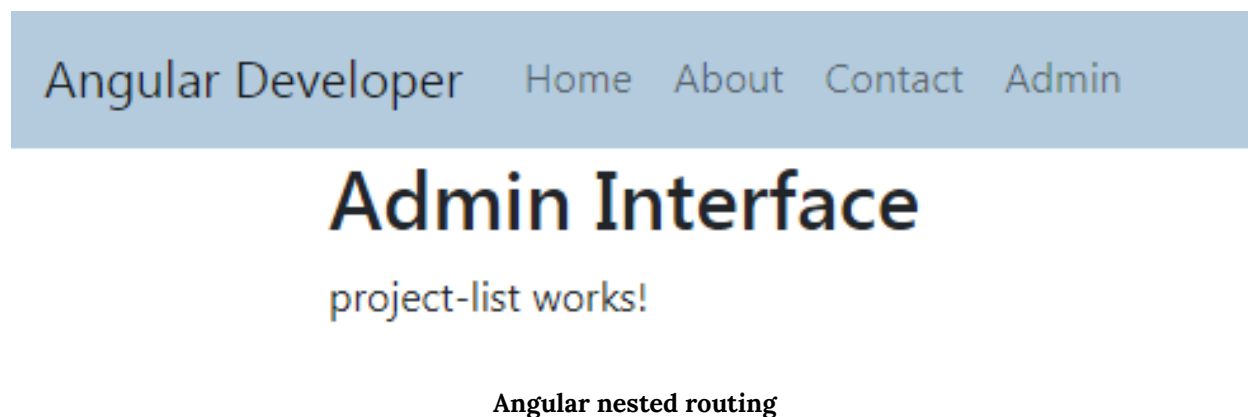
This is a nested router-outlet that will be only used to render the components of the admin module i.e `ProjectListComponent`, `ProjectCreateComponent` and `ProjectCreateComponent`.

Note: If you don't add a nested router outlet in the parent route, child components will be rendered in the parent router outlet of the application.

Next in the `src/app/header/header.component.html` file, add a link to access the admin interface:

```
1 <li class="nav-item">
2 <a class="nav-link" routerLink="/admin/list">Admin</a>
3 </li>
```

At this point, if you click on the admin link in the header, you should see the following interface:



Recap

In this section, you have added nested routing in your Angular application by creating a routing submodule for the admin module and adding a nested router-outlet and child routes for the `/admin` parent route.

In the next section, you'll secure the admin interface using Firebase authentication with email and password.

Adding Firebase Authentication

In the previous section, you have added routing to your `admin` module. You will now add email and password authentication with Firebase.

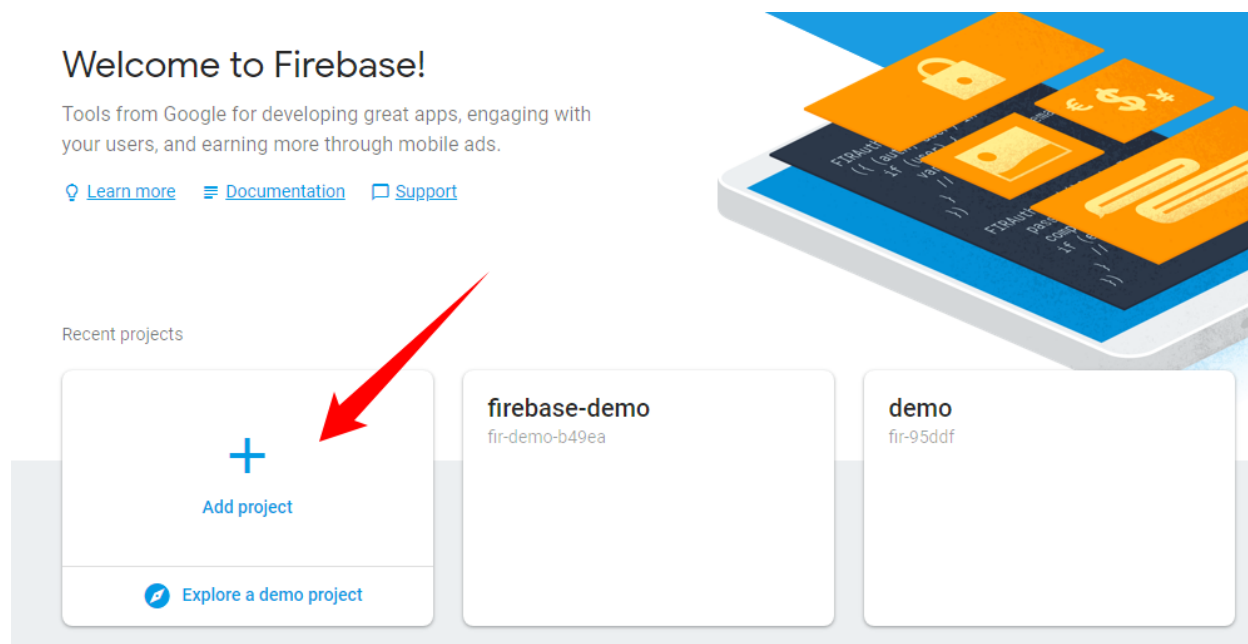
Note: Since we only need to allow admin access to the portfolio owner we only need to provide the login and logout functionality in our application without allowing registration. A user with email and password will be manually created in our Firebase console.

In this section, we'll be adding the following functionalities:

- Email and password authentication with Firebase,
- Securing your Angular application with the Router Guards,
- Storing and accessing the authentication state using the browser's `localStorage` and Angular Observables

Setting up a Firebase Project

If you don't already have a Firebase setup, you simply need to head to your [Firebase console](#) and click on **Add project** then follow the steps to create a Firebase project:



Firebase add project

Enter a name for your project, accept the terms and click on the blue **Create project** button:

Add a project ✕

Project name
angular-portfolio ▼ Tip: Projects span apps across platforms ?

Project ID ?
angular-portfolio-2fbca ✎

Locations ?
United States (Analytics) ✎
us-central (Cloud Firestore)

☒ Use the default settings for sharing Google Analytics for Firebase data

- ✓ Share your Analytics data with all Firebase features
- ✓ Share your Analytics data with Google to improve Google Products and Services
- ✓ Share your Analytics data with Google to enable technical support
- ✓ Share your Analytics data with Google to enable Benchmarking
- ✓ Share your Analytics data with Google Account Specialists

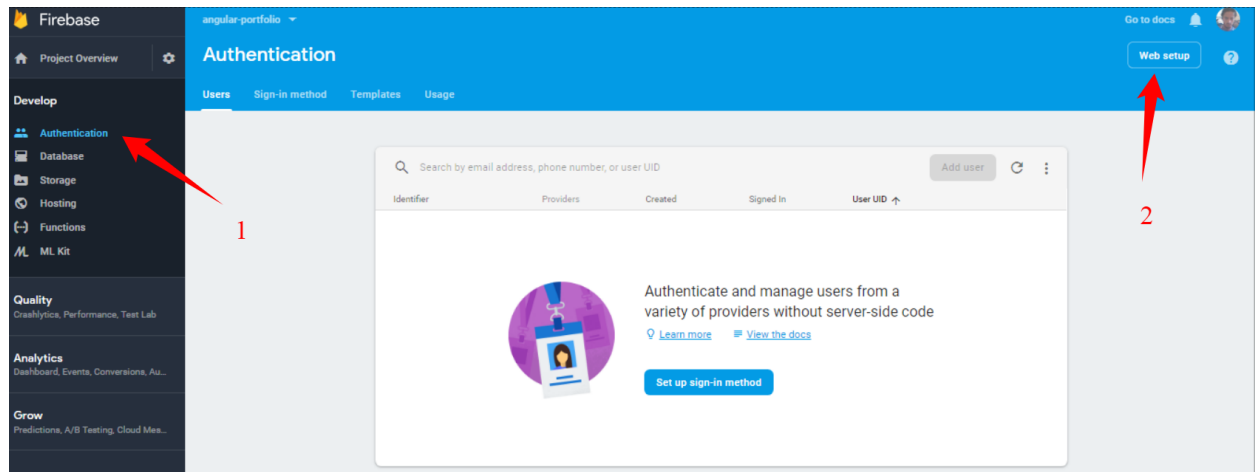
☒ I accept the [controller-controller terms](#). This is required when sharing Analytics data to improve Google Products and Services. [Learn more](#)

Cancel Create project

Create a firebase project

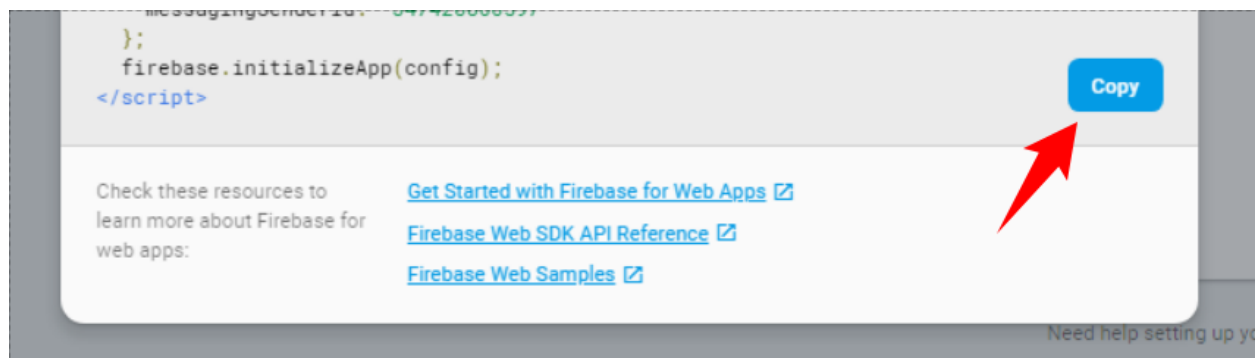
Once your project is created, you need to click on it to go to the admin dashboard for that particular project.

On the dashboard, go to Develop > Authentication and click on the **Web setup** button:



Firebase web setup

A popup window will be opened that contains your firebase credentials:



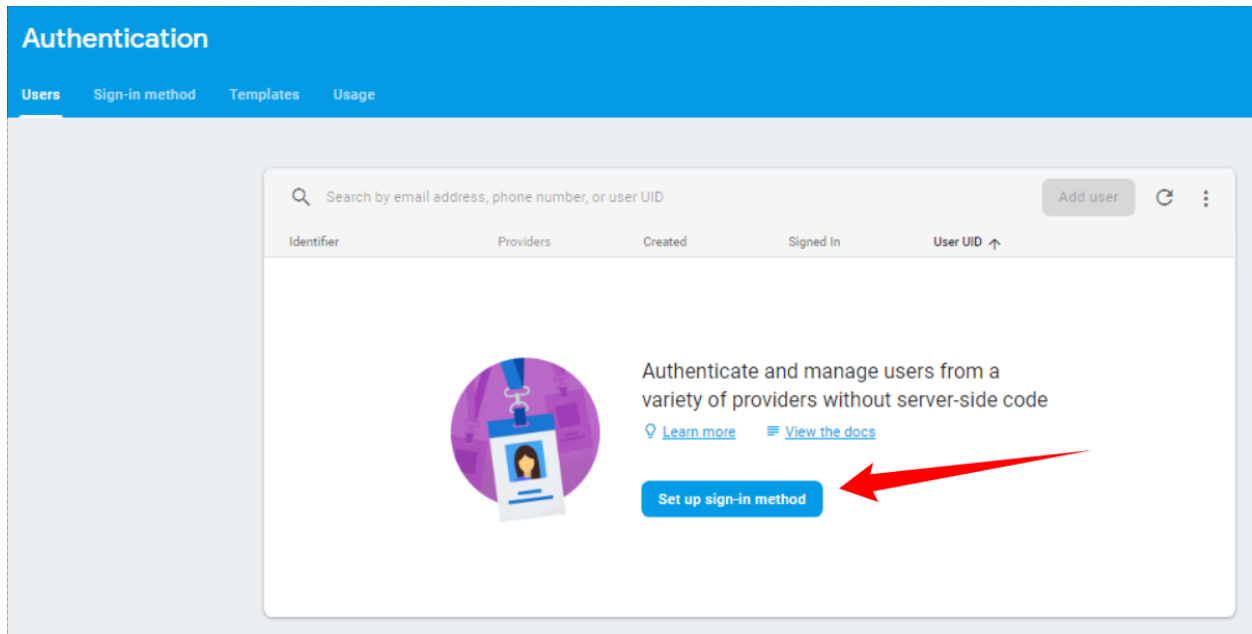
Click on the **Copy** button to copy all code with your credentials in your clipboard:

```
1 <script src="https://www.gstatic.com/firebasejs/5.7.1/firebase.js"></script>
2 <script>
3   // Initialize Firebase
4   var config = {
5     apiKey: "YOUR_API_KEY",
6     authDomain: "YOUR_AUTH_DOMAIN",
7     databaseURL: "YOUR_DATABASE_URL",
8     projectId: "YOUR_PROJECT_ID",
9     storageBucket: "YOUR_STORAGE_BUCKET",
10    messagingSenderId: "YOUR_MESSAGING_SENDER_ID"
11  };
12  firebase.initializeApp(config);
```

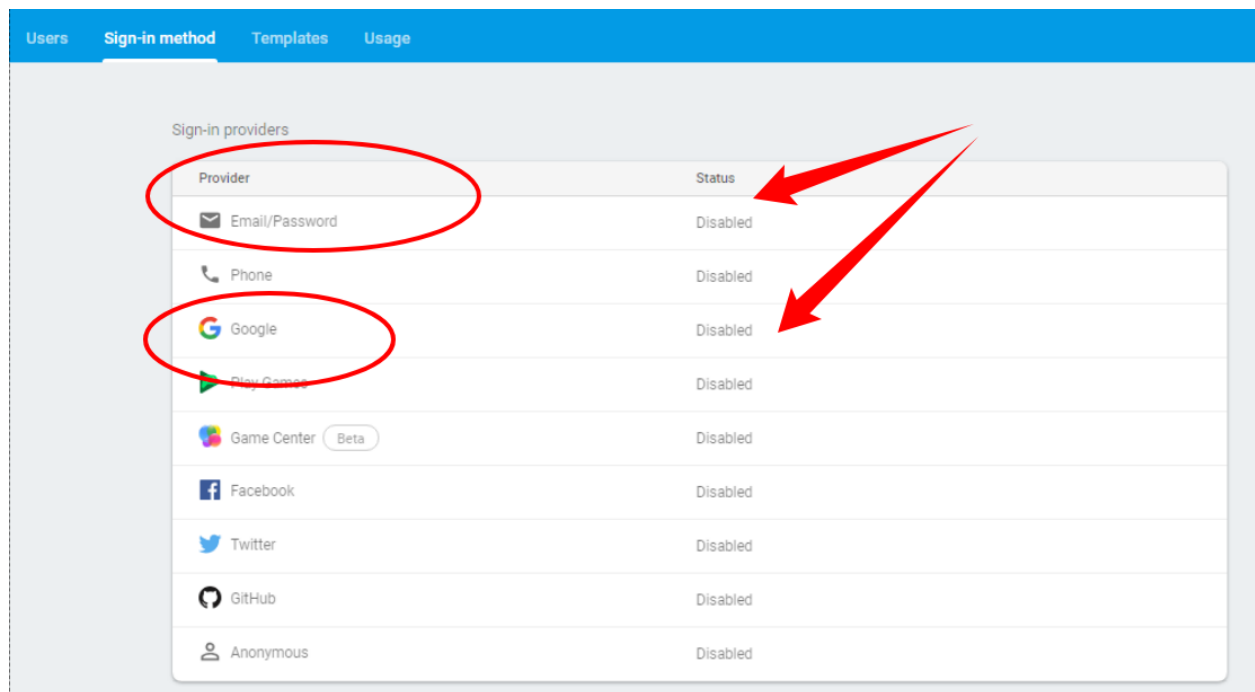
13 </script>

In our case, we only need the values of the `config` object because we'll be installing Firebase SDK from npm.

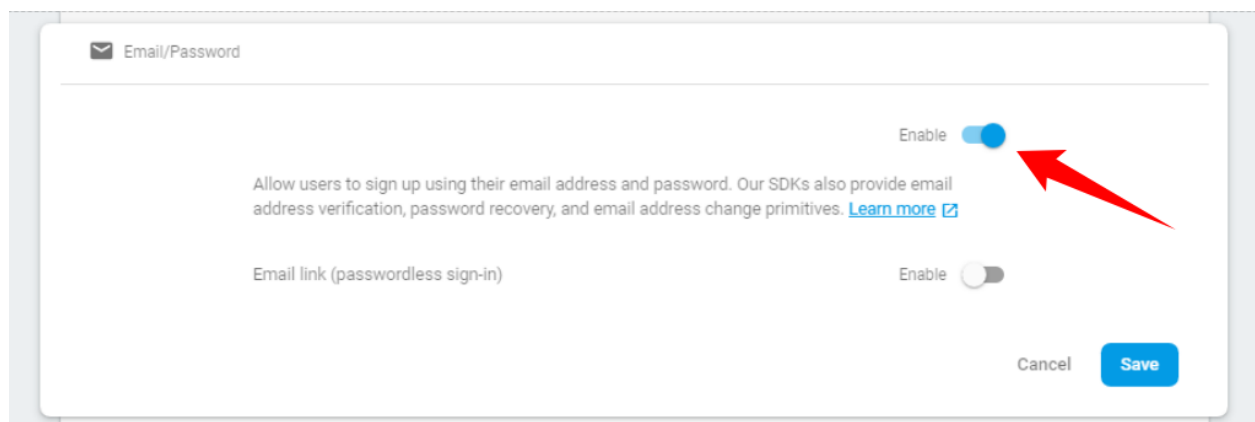
Next, you'll need to enable Email authentication from the **authentication > Sign-in** method tab:



When you click on the **Set up sign-in method** button, you'll be taken to the following tab:



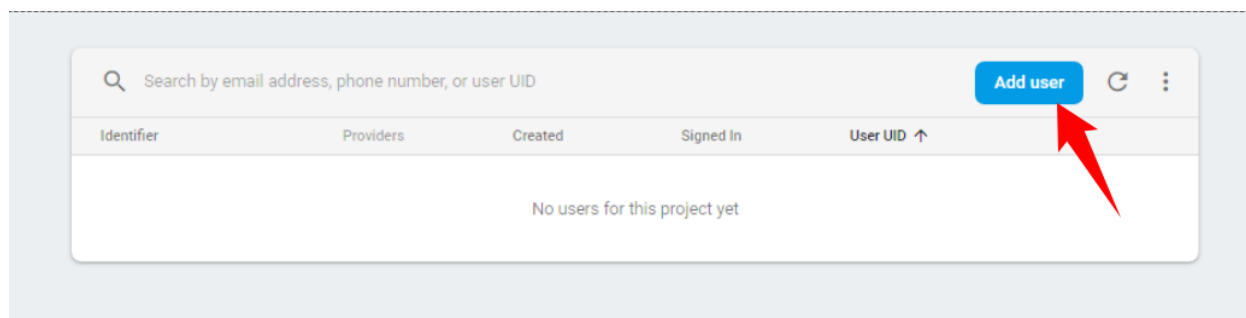
Click on the Email/Password row and then on **Enable** :



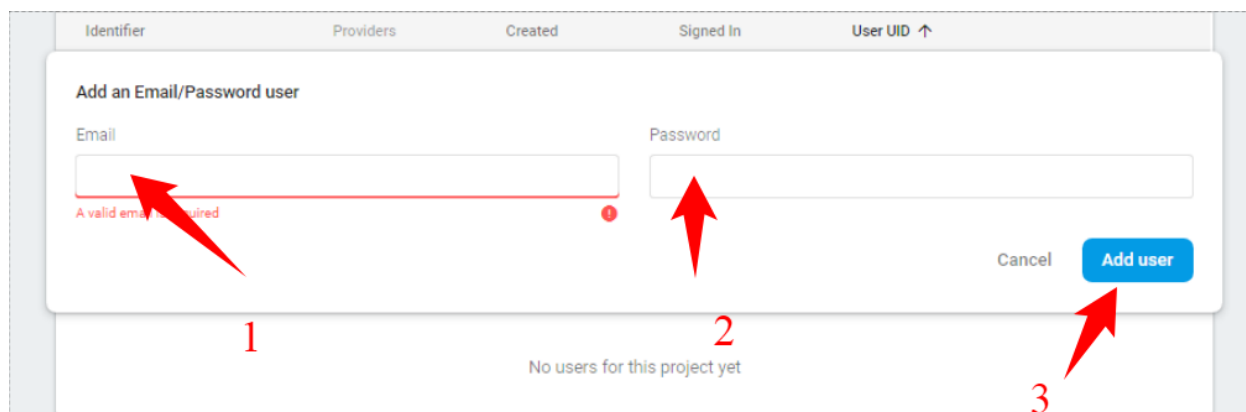
Finally, click on the **Save** button.

One last thing that you need to do from the console is creating a user with email and password that you'll use to login because we will not allow registration from our web application. Only the website admin will be able to access the admin interface to create their portfolio.

Go to the **authentication > Users** tab and click on the **Add user** button:



Enter your user's credentials and click on the **Add user** button:



Installing AngularFire2

Head back to your terminal, make sure you are inside your project's root folder and run the following command to install Firebase SDK and AngularFire2 from npm:

```
1 $ npm install firebase @angular/fire --save
```

As of this writing **firebase v5.7.1** and **angular/fire v5.1.1** will be installed.

Once the library is installed, you need to add it to your application main module. Open the `src/app/app.module.ts` file and update it accordingly:

```
1 // [...]
2 import { AngularFireModule } from "@angular/fire";
3 import { AngularFireAuthModule } from "@angular/fire/auth";
4
5 var config = {
6   apiKey: "YOUR_API_KEY",
7   authDomain: "YOUR_AUTH_DOMAIN",
8   databaseURL: "YOUR_DATABASE_URL",
9   projectId: "YOUR_PROJECT_ID",
10  storageBucket: "YOUR_STORAGE_BUCKET",
11  messagingSenderId: "YOUR_MESSAGING_SENDER_ID"
12 };
13
14 @NgModule({
15   imports: [
16     AngularFireModule.initializeApp(config),
17     AngularFireAuthModule
18   ]
19 })
```

Creating the Authentication (Login) UI Component

After setting up Firebase authentication in our project using AngularFire2 v5, we'll now proceed to create a login UI.

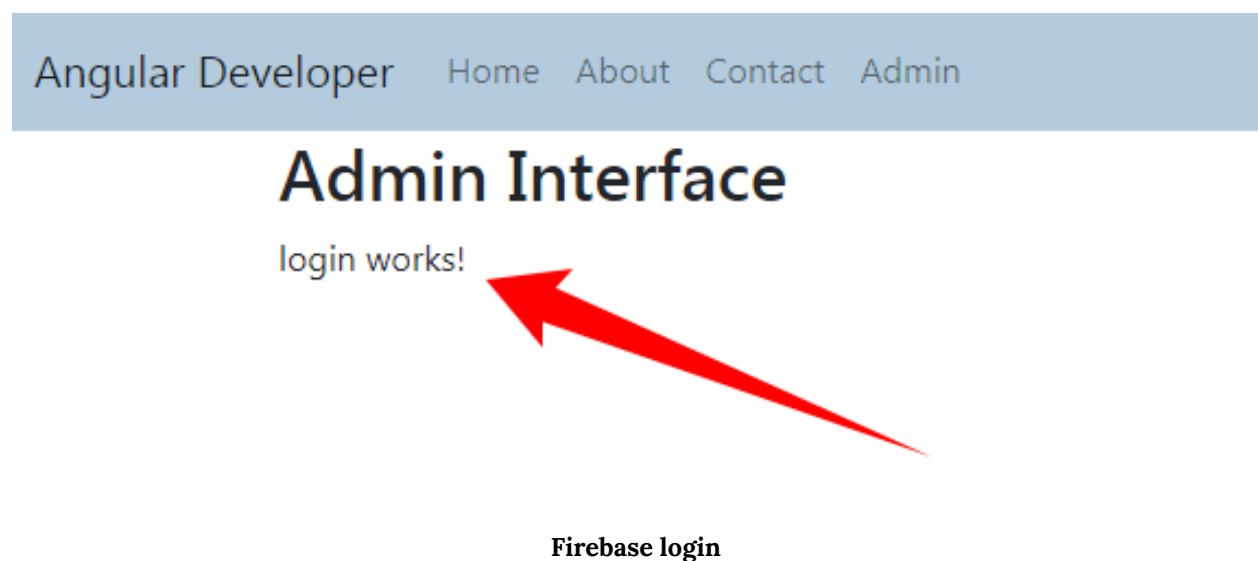
Previously, we have created the `admin` module with a bunch of components to create, update and list the developer's portfolio projects. Now, let's create one more component for user login. In your terminal, run this command:

```
1 $ ng g c admin/login
```

Open the `src/app/admin/admin-routing.module.ts` file and add a path to the component to be able to navigate to it:

```
1 // [...]
2 import { LoginComponent } from './login/login.component';
3
4 const routes: Routes = [
5 {
6   path: 'admin',
7   component: ProjectComponent,
8
9   children: [
10     // [...]
11     { path: 'login', component: LoginComponent }
12   ]
13 }
14 ];
```

All these routes are children of the admin route. So, for example, you can access the login page from the `http://127.0.0.1:4200/admin/login` route:



Creating the Firebase Authentication Service

To abstract all the interactions with Firebase authentication, we will create an Angular service using this command:

```
1 $ ng g s auth/auth.service
```

Open the `src/app/auth/auth.service.ts` file:

```
1 import { Injectable } from '@angular/core';
2
3 @Injectable({
4   providedIn: 'root'
5 })
6 export class AuthService {
7   constructor() { }
8 }
```

Start by adding the following imports:

```
1 import { Router } from '@angular/router';
2 import { auth } from 'firebase/app';
3 import { AngularFireAuth } from '@angular/fire/auth';
4 import { User } from 'firebase';
```

Next, add a variable to store user data:

```
1 export class AuthService {
2   user: User;
```

Next, inject the Firebase authentication service and the router via the service's constructor:

```
1 @Injectable({
2   providedIn: 'root'
3 })
4 export class AuthService {
5   user: User;
6   constructor(public afAuth: AngularFireAuth, public router: Router) { }
7 }
```

Next, in the constructor, we subscribe to the authentication state; if the user is logged in, we add the user's data to the browser's local storage; otherwise, we store a null user

```
1  this.afAuth.authState.subscribe(user => {
2    if (user) {
3      this.user = user;
4      localStorage.setItem('user', JSON.stringify(this.user));
5    } else {
6      localStorage.setItem('user', null);
7    }
8  })
```

Next, add the `login()` method that will be used to login users with email and password:

```
1  async login(email: string, password: string) {
2
3    try {
4      await this.afAuth.auth.signInWithEmailAndPassword(email, password)
5      this.router.navigate(['admin/list']);
6    } catch (e) {
7      alert("Error!" + e.message);
8    }
9  }
```

Next, add the `logout()` method:

```
1  async logout(){
2    await this.afAuth.auth.signOut();
3    localStorage.removeItem('user');
4    this.router.navigate(['admin/login']);
5  }
```

Next, add the `isLoggedIn()` property to check if the user is logged in:

```
1  get isLoggedIn(): boolean {
2    const user = JSON.parse(localStorage.getItem('user'));
3    return user !== null;
4  }
```

We are done with the authentication service. Next, we need to create the login and register UIs.

Implementing the Login UI

Open the `src/app/admin/login/login.component.ts` file and import then inject the authentication service:

```
1 import { Component, OnInit } from '@angular/core';
2 import { AuthService } from '../auth/auth.service';
3
4 @Component({
5   selector: 'app-login',
6   templateUrl: './login.component.html',
7   styleUrls: ['./login.component.css']
8 })
9 export class LoginComponent implements OnInit {
10   constructor(private authService: AuthService) { }
11   ngOnInit() {}
12 }
```

Next open the `src/app/admin/login/login.component.html` file and add the following HTML code:

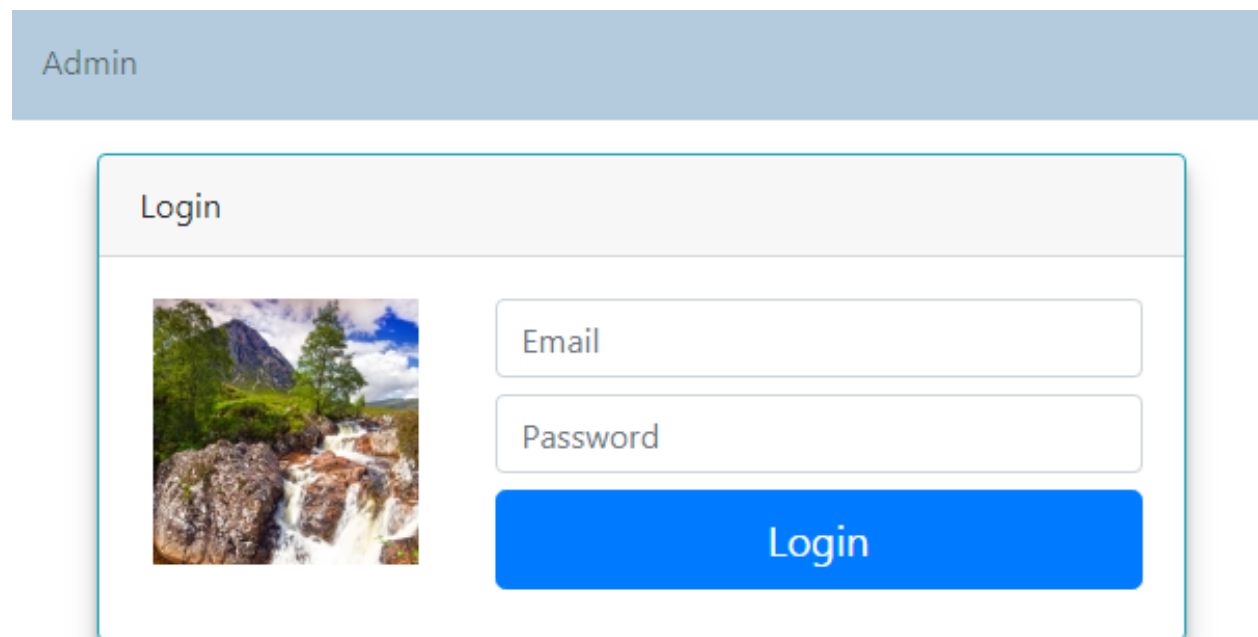
```
1 <div class="container pt-3">
2   <div class="row justify-content-sm-center">
3     <div class="col-sm-10 col-md-6">
4       <div class="card border-info">
5         <div class="card-header">Login</div>
6         <div class="card-body">
7           <div class="row">
8             <div class="col-md-4 text-center">
9               
10             </div>
11             <div class="col-md-8">
12
13               <input type="text" class="form-control mb-2" placeholder="Email" #userEmail required autofocus>
14               <input type="password" class="form-control mb-2" placeholder="Password" #userPassword required>
15               <button class="btn btn-lg btn-primary btn-block mb-1">Login</button>
16
17             </div>
18           </div>
19         </div>
20       </div>
```

```
21     </div>
22   </div>
23 </div>
24 </div>
25 </div>
```

Let's also add some styling for adding a shadow to the form in the `src/app/admin/login/login.component` file:

```
1 .card {
2   box-shadow: 0 10px 20px rgba(0,0,0,0.19), 0 6px 6px rgba(0,0,0,0.23);
3 }
```

If you visit the `http://localhost:4200/admin/login` address, you'll see the following UI:



Bootstrap 4 login form

Next, you need to bind the form inputs and button:

```
1 <button class="btn btn-lg btn-primary btn-block mb-1" (click)="authService.login(u\
2 serEmail.value, userPassword.value)">Login</button>
```

Adding the Logout Button

Open the `src/app/header/header.component.ts` file and inject `AuthService`:

```
1 import { Component, OnInit } from '@angular/core';
2 import { AuthService } from '../admin/auth/auth.service';
3
4 @Component({
5   selector: 'app-header',
6   templateUrl: './header.component.html',
7   styleUrls: ['./header.component.css']
8 })
9 export class HeaderComponent implements OnInit {
10   constructor(private authService: AuthService) { }
11   ngOnInit() {}
12 }
```

Open the `src/app/header/header.component.html` file and add a logout out button:

```
1 <li *ngIf="authService.isLoggedIn" class="nav-item">
2   <a class="nav-link" (click)="authService.logout()">Logout</a>
3 </li>
```

Recap

In this section, you have seen how you can add email and password authentication in your web application using Google's Firebase.

In the next section, you'll use Angular 7 guards to protect the admin interface from unauthorized access i.e you will only be able to activate the route if you have successfully logged in with your email and password that you provided when you created a user in the Firebase console.

Securing the UI with Router Guards

In the previous section, we have added routing in our developer portfolio web application created with Angular 7. Let's now secure the UI with router guards.

We'll be learning how to use `Router Guards` and `UrlTree` data structures to protect the UI if the user is not logged in and redirect them to the login interface if they don't have access to a specific route.

The admin interface can be only accessed by the website admin so we need to use Guards to protect the components of the admin module and only allow access to them if the user is logged in.

First, you need to create a guard. Run the following command in your terminal to generate a guard service:

```
1 $ ng g guard admin/admin
```

Note: We prefix the guard name with the `admin/` path to generate it inside the admin folder for the matter of code organization.

Two `src/app/admin/admin.guard.spec.ts` and `src/app/admin/admin.guard.ts` files will be generated. Open the `src/app/admin/admin.guard.ts` file, you should see the following code:

```
1 import { Injectable } from '@angular/core';
2 import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/r\
3 outer';
4 import { Observable } from 'rxjs';
5
6 @Injectable({
7   providedIn: 'root'
8 })
9 export class AdminGuard implements CanActivate {
10   canActivate(
11     next: ActivatedRouteSnapshot,
12     state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
13     return true;
14   }
15 }
```

From the code, you see that a guard is simply a service that implements the `CanActivate` interface and overrides the `canActivate()` method. In this case, it always returns `true` which means access will be always granted to the user when this guard is applied to a route.

Note: There are other types of Guards such as:

- CanActivateChild: used to allow or disallow access to child routes.
- CanDeactivate: used to allow or deny exit from the route.
- Resolve: used for doing operations (resolve data) just before route activation etc.

Let's change the method to only allow access if the user is logged in. First, you need to import AuthService and inject it via the AdminGuard service and next you need to call the isLoggedIn property in the canActivate() method to check if the user is logged in and return true or false;

```
1 import { Injectable } from '@angular/core';
2 import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/r\
3 outer';
4 import { Observable } from 'rxjs';
5
6 import { AuthService } from '../auth/auth.service';
7
8 @Injectable({
9   providedIn: 'root'
10 })
11 export class AdminGuard implements CanActivate {
12
13   constructor(private authService: AuthService){}
14
15   canActivate(
16     next: ActivatedRouteSnapshot,
17     state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
18     return this.authService.isLoggedIn;
19   }
20 }
```

The canActivate() method will return true if the user is logged in or false otherwise.

The canActivate() method is passed many arguments which makes it easy to determine if the guard needs to allow or disallow access to certain route(s):

1. next: ActivatedRouteSnapshot which is the next route that will be activated if the guard is allowing access,
2. state: RouterStateSnapshot which is the next router state if the guard is allowing access.

Now, you need to apply the guard to the routes you need to protect using the `canActivate` property of the path object. Open the `src/app/admin/admin-routing.module.ts` file and update it accordingly:

```
1  import { NgModule } from '@angular/core';
2  import { Routes, RouterModule } from '@angular/router';
3
4  import { ProjectComponent } from '../project/project.component';
5  import { ProjectListComponent } from '../project-list/project-list.component';
6  import { ProjectCreateComponent } from '../project-create/project-create.component';
7  import { ProjectUpdateComponent } from '../project-update/project-update.component';
8  import { LoginComponent } from '../login/login.component';
9  import { AdminGuard } from '../admin.guard';
10
11  const routes: Routes = [
12    {
13      path: 'admin',
14      component: ProjectComponent,
15      children: [
16        {
17          path: 'list',
18          component: ProjectListComponent,
19          canActivate: [AdminGuard]
20        },
21        {
22          path: 'create',
23          component: ProjectCreateComponent,
24          canActivate: [AdminGuard]
25        },
26        {
27          path: 'update',
28          component: ProjectUpdateComponent,
29          canActivate: [AdminGuard]
30        },
31        {
32          path: 'login',
33          component: LoginComponent
34        }
35      ]
36    }
37 ];
38
39
```

```
40 @NgModule({
41   imports: [RouterModule.forChild(routes)],
42   exports: [RouterModule]
43 })
44 export class AdminRoutingModule { }
```

We protected the `ProjectListComponent`, `ProjectCreateComponent` and `ProjectUpdateComponent` components of the admin module from non logged in users.

Note: The `canActivate` property of the path object takes an array which means you can register multiple guards.

Before Angular 7.1, route guards can only return a boolean, `Promise<boolean>` or `Observable<boolean>` (asynchronous boolean objects) to tell the router if the route can be activated or not. But now, you can also return an `UrlTree` variable which provides the new router state (route) that should be activated.

According to the [Angular docs](#) an `UrlTree` is a data structure that represents a parsed URL.

Note: You can create an `UrlTree` by calling the `parseUrl()` or `createUrlTree()` method of the `Router` object.

Now, let's change our router guard to redirect the users to the `/admin/login` route if they try to access the protected admin components without being logged in first:

- First, we import and inject the router,
- Next, we update the `canActivate()` method to return an `UrlTree` corresponding to the `/admin/login` route.

This is the full code of the `AdminGuard` service:

```
1 import { Injectable } from '@angular/core';
2 import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, Router, UrlTree } \
3   from '@angular/router';
4 import { Observable } from 'rxjs';
5 import { AuthService } from '../auth/auth.service';
6
7 @Injectable({
8   providedIn: 'root'
9 })
10 export class AdminGuard implements CanActivate {
11   constructor(private authService: AuthService, private router: Router)
12   {}
13   canActivate(
14     next: ActivatedRouteSnapshot,
15     state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean | \
16   UrlTree {
17     if(this.authService.isLoggedIn){
18       return true;
19     }
20     else{
21       return this.router.parseUrl("/admin/login");
22     }
23
24   }
25 }
```

We check if the user is logged in and we return true, otherwise, we return the result from the `parseUrl("/admin/login")` method of the injected router instance which is an `UrlTree` of the `/admin/login` route.

Now, go to your application, if you visit any protected route without logging in you will be redirected to the `/admin/login` route where you can log in.

Recap

As a recap, we've seen how to use route guards new feature introduced in Angular v7.1+ which enables you to redirect to another route by using a `UrlTree` parsed route.

In this section, we've used Angular Guards and `UrlTree` structures that correspond to parsed routes to disallow access to certain routes if users are not logged in.

In the next section, we'll proceed by implementing the CRUD operations of the admin interface which allow the portfolio owner to add projects to their website. We'll be using Firestore as our persistence layer.

Build a Contact Form with Angular

In this chapter, we'll learn how to build and work with forms in Angular by creating a simple contact form example.

We'll be using the online IDE from <https://stackblitz.com/>.

Angular provides two methodologies for working with forms:

- The template-based method,
- The reactive (or model-based) method which makes use of Reactivity and RxJS behind the scenes.

For small projects, there is no better approach, just choose the one most convenient to you!

For bigger projects, it's recommended to use reactive forms as they scale better. Check out this [in-depth article](#) for more information.

You can register for an account using your GitHub account and **START A NEW APP** based on Angular from ready templates. Your Angular project will be based on the latest Angular version.

Building a Template-Based Form

Now, let's start with the template-based approach by building an example form. Let's do some configurations.

Open the `src/app/app.module.ts` file and import the `FormsModule` then we add it to the imports array:

```
1 import { NgModule } from '@angular/core';
2 import { BrowserModule } from '@angular/platform-browser';
3 import { FormsModule } from '@angular/forms';
4
5 import { AppComponent } from './app.component';
6
7 @NgModule({
8   imports:      [ BrowserModule, FormsModule ],
9   declarations: [ AppComponent ],
10  bootstrap:    [ AppComponent ]
11 })
12 export class AppModule { }
```

Note: If you are using Stackblitz, FormsModule is already imported in the project.

That's all that you need to add to be able to work with template-based forms.

Next, open the `src/app/app.component.html` file and add the following content:

```
1 <h1> Template-Based Contact Form Example </h1>
2
3 <form #myform = "ngForm" (ngSubmit) = "onSubmit(myform)" >
4   <input type = "text" name = "fullName" placeholder = "Your full name" ngModel>
5   <br/>
6
7   <input type = "email" name = "email" placeholder = "Your email" ngModel>
8   <br/>
9
10  <textarea name = "message" placeholder = "Your message" ngModel></textarea>
11  <br/>
12  <input type = "submit" value = "Send">
13 </form>
```

We can create our form completely in our template. We first add a template reference variable to the form and assign the `ngForm` key to it using the `#myform = "ngForm"` syntax. This will allow us to access the form via the `myform` reference.

Note: The `#myform = "ngForm"` syntax doesn't create a template-based form but only a local template variable that will allow us to work with the form object. In fact, the form was automatically created when you imported `FormsModule` in your project.

Next, we bind the `ngSubmit` event to the `onSubmit()` method (Which we'll add to our component next) and we pass in the form object (via the local template variable)

Next, we register the child controls with the form. We simply add the `NgModel` directive and a `name` attribute to each element.

According to the [docs](#):

`NgForm` creates a top-level `FormGroup` instance and binds it to a form to track aggregate form value and validation status. This is done automatically when `FormsModule` is imported.

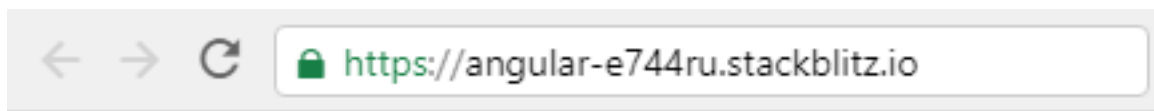
Next, add the `onSubmit()` method to the component. Open the `src/app/app.component.ts` file and add the following code:

```
1  import { Component } from '@angular/core';
2  import { NgForm } from '@angular/forms';
3
4  @Component({
5    selector: 'my-app',
6    templateUrl: './app.component.html',
7    styleUrls: [ './app.component.css' ]
8  })
9  export class AppComponent {
10
11    onSubmit(form: NgForm) {
12      console.log('Your form data : ', form.value);
13    }
14  }
```

We passed in the reference to the `NgForm` object that represents our form to the `onSubmit()` method and we can use it to access various properties like `value` which provides a plain JS object that contains the attributes of the form and their values. In this example, we simply print the form value in the console but in a real-world situation, we can use it to send the data to a server via a POST request.

You can see all the available methods of `NgForm` from the [docs](#).

After, adding some CSS styles from this [pen](#), this is a screenshot of our form UI:



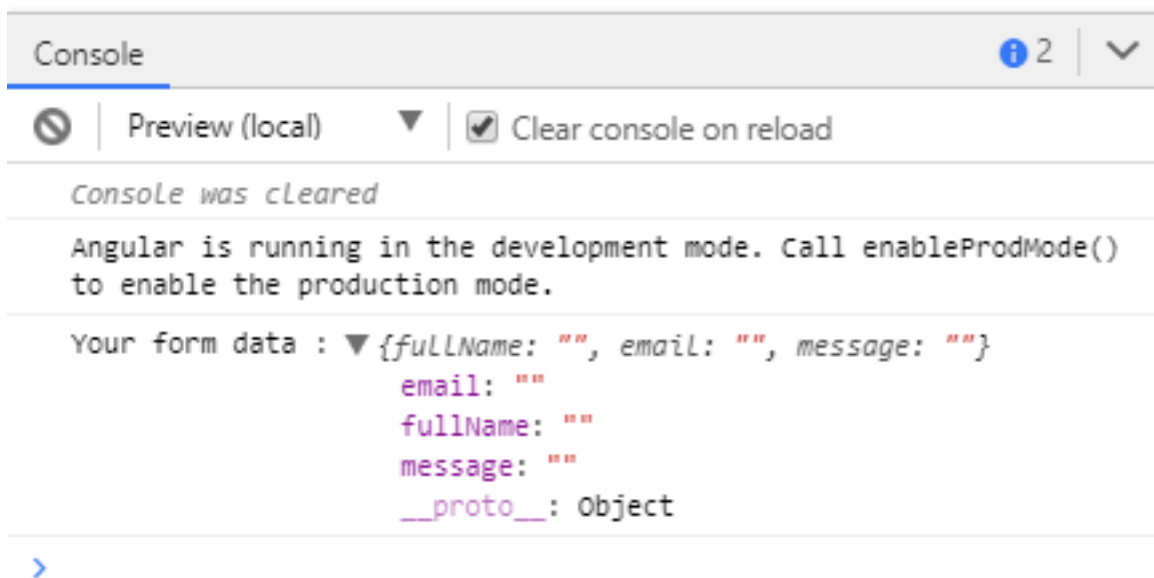
Template-Based Contact Form Example

Your full name

Your email

Your message

Send



Building a Reactive Form

After building our contact form with the template-based approach, let's now see how we can build the same example with the reactive based approach.

Open the `src/app/app.module.ts` file and import `FormsModule` and `ReactiveFormsModule`:

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

Next, add them to the `imports` array of the app module:

```
1 // [...]
2
3 @NgModule({
4   imports:      [ BrowserModule, FormsModule, ReactiveFormsModule ],
5   declarations: [ AppComponent ],
6   bootstrap:    [ AppComponent ]
7 })
8 export class AppModule { }
```

Next, open the `src/app/app.component.ts` file and import `FormGroup` and `FormBuilder`:

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

Next, define the `contactForm` variable which will be used to hold our form object (instance of `FormGroup`) and inject `FormBuilder` via the component constructor:

```
1 export class AppComponent {
2
3   contactForm: FormGroup;
4   constructor(private FormBuilder: FormBuilder) {
5   }
6 }
```

Next, define a `createContactForm()` method where we'll build our form:

```
1  createContactForm(){
2    this.contactForm = this.formBuilder.group({
3      fullName: [''],
4      email: [''],
5      message: ['']
6    });
7  }
```

We call the `group()` method of the injected instance of `FormBuilder` to create a `FormGroup` of three controls, `fullName`, `email` and `message`. Each control can take an optional array of options and validation rules.

Next, we call the method on the constructor:

```
1  constructor(private FormBuilder: FormBuilder) {
2    this.createContactForm();
3  }
```

Finally, add the following method which will be called when we submit our form:

```
1  onSubmit() {
2    console.log('Your form data : ', this.contactForm.value );
3  }
```

Now, we need to bind this form object to our HTML form. Open the `src/app/app.component.html` file and add the following code:

```
1  <h1> Reactive Contact Form Example </h1>
2
3  <form [formGroup]="contactForm" (ngSubmit)="onSubmit()">
4    <input type = "text" name = "fullName" placeholder = "Your full name" formControlName="fullName" >
5    <br/>
6
7    <input type = "email" name = "email" placeholder = "Your email" formControlName="email" >
8    <br/>
9
10   <textarea name = "message" placeholder = "Your message" formControlName="message" \
11   ></textarea>
```

```
14     <br/>
15     <input type = "submit" value = "Send">
16 </form>
```

We use property binding to bind the form using the `formGroup` property. Next, we use `formControlName` to sync the `FormControl` objects in `contactForm` with the HTML form controls by name. See the [docs](#) for more details. Finally, we bind the `ngSubmit` event of the form to the `onSubmit()` method.

Conclusion

In this chapter, we've built an example contact form in Angular 8 using the template-based approach and the reactive (model-based) approach

Building a News App with Angular - HttpClient

In this tutorial, you'll learn by example how to send GET requests to REST API servers in your Angular application using `HttpClient`. We'll also learn how to use the basic concepts of Angular like components and services and how to use the `ngFor` directive to display collections of data.

We'll be consuming a JSON API available from [NewsAPI.org](https://newsapi.org/)

Throughout this chapter, we are going to build a simple example from scratch using Angular CLI and we'll see how to use `HttpClient` to send GET requests to third-party REST API servers and how to consume and display the returned JSON data.

In more details, we'll learn:

- How to create an Angular project using Angular CLI,
- How to quickly set up routing in our project,
- How to create Angular components and services,
- How to subscribe to Observables,
- How to use the `ngFor` directive in templates to iterate over data.

Prerequisites

Before getting started, you need a few requirements. You need to have the following tools installed on your development machine:

- Node.js and npm. You can install both of them from the [official website](https://nodejs.org/en/).
- Angular CLI (You can install it from npm using: `npm install -g @angular/cli`)

Creating an Angular Project

Now let's create our Angular project. Open a new terminal and run the following command:

```
1 $ ng new angular-httpclient-demo
```

The CLI will prompt you if **Would you like to add Angular routing? (y/N)**, type **y**. And **Which stylesheet format would you like to use?** Choose CSS and type **Enter**.

Next, you can serve your application locally using the following commands:

```
1 $ cd ./angular-httpclient-demo
2 $ ng serve
```

Your application will be running from `http://localhost:4200`.

Getting News Data

Before you can fetch the news data from [NewsAPI.org](https://newsapi.org/) which offers a free plan for open source and development projects, you first need to go the [register](#) page for getting an API key.

Adding an Angular Service

Next, let's create a service that will take care of getting data from the news API. Open a new terminal and run the following command:

```
1 $ ng generate service api
```

Setting up HttpClient

Next, open the `src/app/app.module.ts` file then import `HttpClientModule` and add it to the `imports` array:

```
1 // [...]  
2 import { HttpClientModule } from '@angular/common/http';  
3  
4 @NgModule({  
5   declarations: [AppComponent],  
6   entryComponents: [],  
7   imports: [  
8     // [...]  
9     HttpClientModule,  
10  ],  
11  // [...]  
12 })  
13 export class AppModule {}
```

That's all, we are now ready to use the `HttpClient` in our project.

Injecting HttpClient in The Angular Service

Next, open the `src/app/api.service.ts` file and inject `HttpClient` via the service constructor:

```
1 import { Injectable } from '@angular/core';  
2 import { HttpClient } from '@angular/common/http';  
3  
4 @Injectable({  
5   providedIn: 'root'  
6 })  
7 export class ApiService {  
8  
9   constructor(private httpClient: HttpClient) { }  
10 }
```

Sending GET Request for Fetching Data

Next, define an `API_KEY` variable which will hold your API key from the News API:

```
1 export class ApiService {  
2   API_KEY = 'YOUR_API_KEY';
```

Finally, add a method that sends a GET request to an endpoint for TechCrunch news:

```
1   public getNews(){  
2     return this.httpClient.get(`https://newsapi.org/v2/top-headlines?sources=techcrunch&apiKey=${this.API_KEY}`);  
3   }  
4 }
```

That's all we need to add for the service.

How the `HttpClient.get()` Method Works

The `HttpClient.get()` method is designed to send HTTP GET requests. The syntax is as follows:

```
1 get(url: string, options: {  
2   headers?: HttpHeaders;  
3   observe: 'response';  
4   params?: HttpParams;  
5   reportProgress?: boolean;  
6   responseType?: 'json';  
7   withCredentials?: boolean;  
8 }): Observable<HttpResponse<Object>>;
```

It takes a REST API endpoint and an optional `options` object and returns an `Observable` instance.

Creating an Angular Component

Now, let's create an Angular 8 component for displaying the news data. Head back to your terminal and run the following command:

```
1 $ ng generate component news
```

Injecting ApiService in Your Component

Next, open the `src/app/news/news.component.ts` file and start by importing `ApiService` in your component:

```
1 import { ApiService } from '../api.service';
```

Next, you need to inject `ApiService` via the component's constructor:

```
1 import { Component, OnInit } from '@angular/core';
2 import { ApiService } from '../api.service';
3 @Component({
4   selector: 'app-news',
5   templateUrl: './news.component.html',
6   styleUrls: ['./news.component.css']
7 })
8 export class NewsComponent implements OnInit {
9
10   constructor(private apiService: ApiService) { }
11 }
```

Sending the GET Request & Subscribing to The Observable

Next, define an `articles` variable and call the `getNews()` method of the API service in the `ngOnInit()` method of the component:

```
1 export class NewsComponent implements OnInit {
2   articles;
3
4   constructor(private apiService: ApiService) { }
5   ngOnInit() {
6     this.apiService.getNews().subscribe((data)=>{
7       console.log(data);
8       this.articles = data['articles'];
9     });
10  }
11 }
```

This will make sure our data is fetched once the component is loaded.

We call the `getNews()` method and subscribe to the returned Observable which will send a GET request to the news endpoint.

Displaying Data in The Template with NgFor

Let's now display the news articles in our component template. Open the `src/app/news.component.html` file and update it as follows:

```
1 <div *ngFor="let article of articles">
2   <h2>{{article.title}}</h2>
3
4   <p>
5     {{article.description}}
6   </p>
7   <a href="{{article.url}}">Read full article</a>
8 </div>
```

Adding the Angular Component to The Router

Angular CLI 8 has automatically added routing for us, so we don't need to set up anything besides adding the component(s) to our Router configuration. Open the `src/app/app-routing.module.ts` file and start by importing the news component as follows:

```
1 import { NewsComponent } from './news/news.component';
```

Next, add the component to the `routes` array:

```
1 import { NgModule } from '@angular/core';
2 import { Routes, RouterModule } from '@angular/router';
3 import { NewsComponent } from './news/news.component';
4 const routes: Routes = [
5   {path: 'news', component: NewsComponent}
6 ];
7 @NgModule({
8   imports: [RouterModule.forRoot(routes)],
9   exports: [RouterModule]
10 })
11 export class AppRoutingModule { }
```

You can now access your component from the `/news` path.

Conclusion

In this chapter, we used Angular to build a simple news application that retrieves data from a JSON REST API using the `get()` method of `HttpClient`. We've seen how to subscribe to the RxJS Observable returned from the `get()` method and how to use the `*ngFor` directive to iterate over the fetched data in the template.

Building a Full-Stack App with Angular, PHP and MySQL

In this chapter, you'll create an example REST API CRUD Angular application with PHP and a MySQL backend.

You will be creating a simple RESTful API that supports GET, POST, PUT and DELETE requests and allow you to perform CRUD operations against a MySQL database to create, read, update and delete records from a database.

For the application design, It's a simple interface for working with vehicle insurance policies. For the sake of simplicity, you are only going to add the following attributes to the `policies` database table:

- `number` which stores to the insurance policy number,
- `amount` which stores the insurance amount.

This is of course far from being a complete database design for a fully working insurance system. Because at least you need to add other tables like employees, clients, coverage, vehicles, and drivers, etc. And also the relationships between all these entities.

Prerequisites

In this chapter, we'll assume you have the following prerequisites:

- The MySQL database management system installed on your development machine,
- PHP installed on your system (both these first requirements are required by the back-end project),
- Node.js 8.9+ and NPM installed in your system. This is only required by your Angular project.
- You also need to have a working experience of PHP and the various functions that will be used to create the SQL connection, getting the GET and POST data and returning JSON data in your code.

- You need to be familiar with TypeScript.
- Basic knowledge of Angular is preferable but not required since you'll go from the first step until you create a project that communicates with a PHP server.

Also read [PHP Image/File Upload Tutorial and Example \[FormData and Angular 7 Front-End\]](#)

Creating the PHP Application

Let's start by creating a simple PHP script that connects to a MySQL database and listens to API requests then responds accordingly by either fetching and returning data from the SQL table or insert, update and delete data from the database.

Go ahead and create a folder structure for your project using the following commands:

```
1 $ mkdir angular-php-app
2 $ cd angular-php-app
3 $ mkdir backend
```

We create the `angular-php-app` that will contain the full front-end and back-end projects. Next, we navigate inside it and create the `backend` folder that will contain a simple PHP script for implementing a simple CRUD REST API against a MySQL database.

Next, navigate into your `backend` project and create an `api` folder.

```
1 $ cd backend
2 $ mkdir api
```

Inside the `api` folder, create the following files:

```
1 $ cd api
2 $ touch database.php
3 $ touch read.php
4 $ touch create.php
5 $ touch update.php
6 $ touch delete.php
```

Open the `backend/api/database.php` file and add the following PHP code step by step:

```
1 <?php
2 header("Access-Control-Allow-Origin: *");
3 header("Access-Control-Allow-Methods: PUT, GET, POST, DELETE");
4 header("Access-Control-Allow-Headers: Origin, X-Requested-With, Content-Type, Accept\
5 ");
```

These lines are used to add response headers such as CORS and the allowed methods (PUT, GET, DELETE and POST).

Setting CORS to * will allow your PHP server to accept requests from another domain where the Angular 7 server is running from without getting blocked by the browser because of the **Same Origin Policy**. In development, you'll be running the PHP server from `localhost:8080` port and Angular from `localhost:4200` which are considered as two distinct domains.

Next, add:

```
1 define('DB_HOST', 'localhost');
2 define('DB_USER', 'root');
3 define('DB_PASS', 'YOUR_PASSWORD');
4 define('DB_NAME', 'mydb');
```

These variables hold the credentials that will be used to connect to the MySQL database and the name of the database.

Note: Make sure you change them to your actual MySQL credentials. Also make sure you have created a database with a policies table that has two number and amount columns.

Next, add:

```
1 function connect()
2 {
3     $connect = mysqli_connect(DB_HOST ,DB_USER ,DB_PASS ,DB_NAME);
4
5     if (mysqli_connect_errno($connect)) {
6         die("Failed to connect:" . mysqli_connect_error());
7     }
8
9     mysqli_set_charset($connect, "utf8");
10
```

```
11     return $connect;
12 }
13
14 $con = connect();
```

This will allow you to create a connection to the MySQL database using the `mysqli` extension.

That's all for the `database.php` file

Implementing the Read Operation

Now, let's implement the read operation. Open the `backend/api/read.php` file and add the following code:

```
1 <?php
2 /**
3  * Returns the list of policies.
4  */
5 require 'database.php';
6
7 $policies = [];
8 $sql = "SELECT id, number, amount FROM policies";
9
10 if($result = mysqli_query($con,$sql))
11 {
12     $i = 0;
13     while($row = mysqli_fetch_assoc($result))
14     {
15         $policies[$i]['id'] = $row['id'];
16         $policies[$i]['number'] = $row['number'];
17         $policies[$i]['amount'] = $row['amount'];
18         $i++;
19     }
20
21     echo json_encode($policies);
22 }
23 else
24 {
25     http_response_code(404);
26 }
```

This will fetch the list of policies from the database and return them as a JSON response. If there is an error it will return a 404 error.

Implementing the Create Operation

Let's now implement the create operation. Open the `backend/api/create.php` file and add the following code:

```
1  <?php
2  require 'database.php';
3
4  // Get the posted data.
5  $postdata = file_get_contents("php://input");
6
7  if(isset($postdata) && !empty($postdata))
8  {
9      // Extract the data.
10     $request = json_decode($postdata);
11
12
13     // Validate.
14     if(trim($request->number) === '' || (float)$request->amount < 0)
15     {
16         return http_response_code(400);
17     }
18
19     // Sanitize.
20     $number = mysqli_real_escape_string($con, trim($request->number));
21     $amount = mysqli_real_escape_string($con, (int)$request->amount);
22
23
24     // Create.
25     $sql = "INSERT INTO `policies`(`id`,`number`,`amount`) VALUES (null,'{$number}','{\
26 $amount}')";
27
28     if(mysqli_query($con,$sql))
29     {
30         http_response_code(201);
31         $policy = [
32             'number' => $number,
33             'amount' => $amount,
```

```
34     'id'      => mysqli_insert_id($con)
35 ];
36 echo json_encode($policy);
37 }
38 else
39 {
40     http_response_code(422);
41 }
42 }
```

Implementing the Update Operation

Open the backend/api/update.php file and add the following code:

```
1  <?php
2  require 'database.php';
3
4  // Get the posted data.
5  $postdata = file_get_contents("php://input");
6
7  if(isset($postdata) && !empty($postdata))
8  {
9      // Extract the data.
10     $request = json_decode($postdata);
11
12     // Validate.
13     if ((int)$request->id < 1 || trim($request->number) == '' || (float)$request->amount < 0) {
14         return http_response_code(400);
15     }
16
17
18     // Sanitize.
19     $id      = mysqli_real_escape_string($con, (int)$request->id);
20     $number   = mysqli_real_escape_string($con, trim($request->number));
21     $amount   = mysqli_real_escape_string($con, (float)$request->amount);
22
23     // Update.
24     $sql = "UPDATE `policies` SET `number`='{$number}', `amount`='{$amount}' WHERE `id` = '\{ $id }' LIMIT 1";
25
26
```

```
27     if(mysqli_query($con, $sql))
28     {
29         http_response_code(204);
30     }
31     else
32     {
33         return http_response_code(422);
34     }
35 }
```

Implementing the Delete Operation

Open the backend/api/delete.php file and add the following code:

```
1  <?php
2
3  require 'database.php';
4
5  // Extract, validate and sanitize the id.
6  $id = ($_GET['id'] !== null && (int)$_GET['id'] > 0)? mysqli_real_escape_string($con\
7  , (int)$_GET['id']) : false;
8
9  if(!$id)
10 {
11     return http_response_code(400);
12 }
13
14 // Delete.
15 $sql = "DELETE FROM `policies` WHERE `id`='{$id}' LIMIT 1";
16
17 if(mysqli_query($con, $sql))
18 {
19     http_response_code(204);
20 }
21 else
22 {
23     return http_response_code(422);
24 }
```

In all operations, we first require the database.php file for connecting to the MySQL database and then we implement the appropriate logic for the CRUD operation.

Serving the PHP REST API Project

You can next serve your PHP application using the built-in development server using the following command:

```
1 $ php -S 127.0.0.1:8080 -t ./angular-php-app/backend
```

This will run a development server from the 127.0.0.1:8080 address.

Creating the MySQL Database

In your terminal, run the following command to start the MySQL client:

```
1 $ mysql -u root -p
```

The client will prompt for the password that you configured when installing MySQL in your system.

Next, run this SQL query to create a `mydb` database:

```
1 mysql> create database mydb;
```

Creating the `policies` SQL Table

Next create the `policies` SQL table with two `number` and `amount` columns:

```
1 mysql> create table policies( id int not null auto_increment, number varchar(20), am\
2 ount float, primary key(id));
```

Now, you are ready to send GET, POST, PUT and DELETE requests to your PHP server running from the 127.0.0.1:8080 address.

For sending test requests, you can use REST clients such as Postman or cURL before creating the Angular UI.

Leave your server running and open a new terminal.

Wrap-up

In this section, you have created a PHP RESTful API that can be used to execute CRUD operations against a MySQL database to create, read, update and delete insurance policies.

You have enabled CORS so you can use two domains `localhost:8000` and `localhost:4200` for respectively serving your PHP and Angular apps and being able to send requests from Angular to PHP without getting blocked by the Same Origin Policy rule in modern web browsers.

Creating the Angular Front-End

In this section, you'll learn how to use `HttpClient` to make HTTP calls to a REST API and use template-based forms to submit data.

Now, that you've created the RESTful API with a PHP script, you can proceed to create your Angular project.

Installing Angular CLI

The recommended way of creating Angular projects is through using Angular CLI, the official tool created by the Angular team. The latest and best version yet is Angular CLI 8 so head back to another terminal window and run the following command to install the CLI:

```
1 $ npm install -g @angular/cli
```

Note: This will install Angular CLI globally so make sure you have configured npm to install packages globally without adding `sudo` in Debian systems and macOS or using an administrator command prompt on Windows. You can also just fix your [npm permissions](#) if you get any issues

Generating a New Project

That's it! You can now use the CLI to create an Angular project using the following command:


```
1 $ cd angular-php-app
2 $ ng new frontend
```

The CLI will ask you if **Would you like to add Angular routing?** type **y** because we'll need routing setup in our application. And **Which stylesheet format would you like to use?** Select **CSS**.

Wait for the CLI to generate and install the required dependencies and then you can start your development server using:

```
1 $ cd frontend
2 $ ng serve
```

You can access the frontend application by pointing your browser to the <http://localhost:4200> address.

Setting up HttpClient & Forms

Angular provides developers with a powerful HTTP client for sending HTTP requests to servers. It's based on the `XMLHttpRequest` interface supported on most browsers and has a plethora of features like the use of RxJS Observable instead of callbacks or promises, typed requests and responses and interceptors.

You can set up `HttpClient` in your project by simply importing the `HttpClientModule` in your main application module.

We'll also be using a template-based form in our application so we need to import `FormsModule`.

Open the `src/app/app.module.ts` file and import `HttpClientModule` then add it to the imports array of `@NgModule`:

```
1 import { BrowserModule } from '@angular/platform-browser';
2 import { NgModule } from '@angular/core';
3 import { HttpClientModule } from '@angular/common/http';
4 import { FormsModule } from '@angular/forms';
5
6 import { AppRoutingModule } from './app-routing.module';
7 import { AppComponent } from './app.component';
8
9 @NgModule({
10   declarations: [
11     AppComponent
12   ],
13   imports: [
14     BrowserModule,
15     HttpClientModule,
16     FormsModule,
17     AppRoutingModule
18   ],
19   providers: [],
20   bootstrap: [AppComponent]
21 })
22 export class AppModule { }
```

That's it! You can now use `HttpClient` in your components or services via dependency injection.

Creating an Angular Service

Let's now, create an Angular service that will encapsulate all the code needed for interfacing with the RESTful PHP backend.

Open a new terminal window, navigate to your `frontend` project and run the following command:

```
1 $ ng generate service api
```

This will create the `src/app/api.service.spec.ts` and `src/app/api.service.ts` files with a minimum required code.

Open the `src/app/api.service.ts` and start by importing and injecting `HttpClient`:

```
1 import { Injectable } from '@angular/core';
2 import { HttpClient } from '@angular/common/http';
3
4 @Injectable({
5   providedIn: 'root'
6 })
7 export class ApiService {
8
9   constructor(private httpClient: HttpClient) {}
10 }
```

We inject `HttpClient` as a private `httpClient` instance via the service constructor. This is called **dependency injection**. If you are not familiar with this pattern. Here is the definition from [Wikipedia](#):

In software engineering, dependency injection is a technique whereby one object (or static method) supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it. The service is made part of the client's state.[1] Passing the service to the client, rather than allowing a client to build or find the service, is the fundamental requirement of the pattern.

Also, this is what [Angular docs](#) says about dependency injection in Angular:

Dependency injection (DI), is an important application design pattern. Angular has its own DI framework, which is typically used in the design of Angular applications to increase their efficiency and modularity. Dependencies are services or objects that a class needs to perform its function. DI is a coding pattern in which a class asks for dependencies from external sources rather than creating them itself.

You can now use the injected `httpClient` instance to send HTTP requests to your PHP REST API.

Creating the Policy Model

Create a `policy.ts` file in the `src/app` folder of your project and add the following TypeScript class:

```
1 export class Policy {
2   id: number;
3   number: number;
4   amount: number;
5 }
```

Defining the CRUD Methods

Next, open the `src/app/api.service.ts` file and import the `Policy` model and the `RxJS Observable` interface:

```
1 import { Policy } from './policy';
2 import { Observable } from 'rxjs';
```

Next, define the `PHP_API_SERVER` variable in the service:

```
1 export class ApiService {
2   PHP_API_SERVER = "http://127.0.0.1:8080";
```

The `PHP_API_SERVER` holds the address of the PHP server.

Next, add the `readPolicies()` method that will be used to retrieve the insurance policies from the REST API endpoint via a GET request:

```
1 readPolicies(): Observable<Policy[]>{
2   return this.httpClient.get<Policy[]>(`${this.PHP_API_SERVER}/api/read.php`);
3 }
```

Next, add the `createPolicy()` method that will be used to create a policy in the database:

```
1 createPolicy(policy: Policy): Observable<Policy>{
2   return this.httpClient.post<Policy>(`${this.PHP_API_SERVER}/api/create.php`, pol\
3 icy);
4 }
```

Next, add the `updatePolicy()` method to update policies:

```
1  updatePolicy(policy: Policy){
2      return this.httpClient.put<Policy>(`${this.PHP_API_SERVER}/api/update.php`, poli\
3  cy);
4  }
```

Finally, add the `deletePolicy()` to delete policies from the SQL database:

```
1  deletePolicy(id: number){
2      return this.httpClient.delete<Policy>(`${this.PHP_API_SERVER}/api/delete.php/?id\
3  =${id}`);
4  }
```

That's all for the service.

Creating the Angular Component

After creating the service that contains the CRUD operations, let's now create an Angular component that will call the service methods and will contain the table to display policies and a form to submit a policy to the PHP backend.

head back to your terminal and run the following command:

```
1  $ ng generate component dashboard
```

Let's add this component to the Router. Open the `src/app/app-routing.module.ts` file and add a `/dashboard` route:

```
1  import { NgModule } from '@angular/core';
2  import { Routes, RouterModule } from '@angular/router';
3  import { DashboardComponent } from '../dashboard/dashboard.component';
4
5
6  const routes: Routes = [
7      { path: 'dashboard', component: DashboardComponent }
8  ];
9
10 @NgModule({
11     imports: [RouterModule.forRoot(routes)],
12     exports: [RouterModule]
13 })
14 export class AppRoutingModule { }
```

You can now access your dashboard component from the `127.0.0.1:4200/dashboard` URL. This is a screenshot of the page at this point:

Welcome to frontend!



Here are some links to help you start:

- [Tour of Heroes](#)
- [CLI Documentation](#)
- [Angular blog](#)

dashboard works!

PHP Angular REST CRUD example

Let's remove the boilerplate content added by Angular CLI. Open the `src/app/app.component.html` file and update accordingly:

```
1 <router-outlet></router-outlet>
```

We only leave the router outlet where Angular router will insert the matched component(s).

Next, open the `src/app/dashboard/dashboard.component.ts` file and import `ApiService` then inject it via the component constructor:

```
1 import { Component, OnInit } from '@angular/core';
2 import { ApiService } from '../api.service';
3
4 @Component({
5   selector: 'app-dashboard',
6   templateUrl: './dashboard.component.html',
7   styleUrls: ['./dashboard.component.css']
8 })
9 export class DashboardComponent implements OnInit {
10
11   constructor(private apiService: ApiService) { }
12
13   ngOnInit() {
14   }
15 }
```

We inject `ApiService` as a private `apiService` instance.

Next, let's define the `policies` array that will hold the insurance policies once we get them from the server after we send a `GET` request and also the `selectedPolicy` variable that will hold the selected policy from the table.

```
1 export class DashboardComponent implements OnInit {
2   policies: Policy[];
3   selectedPolicy: Policy = { id : null , number:null, amount: null};
```

On the `ngOnInit()` method of the component, let's call the `readPolicies()` method of `ApiService` to get the policies:

```
1   ngOnInit() {
2     this.apiService.readPolicies().subscribe((policies: Policy[])=>{
3       this.policies = policies;
4       console.log(this.policies);
5     })
6   }
```

We call the `readPolicies()` which return an `Observable<Policy[]>` object and we subscribe to the `RxJS` `Observable`. We then assign the returned policies to the `policies` array of our component and we also log the result in the console.

Note: The actual `HTTP` request is only sent to the server when you subscribe to the returned `Observable`.

You should see the returned policies in your browser console. This is a screenshot of the output in my case:

dashboard.component.ts:18

```

▼ (4) [{...}, {...}, {...}, {...}] ⓘ
  ▼ 0:
    amount: "400"
    id: "2"
    number: "PL005000"
    ▶ __proto__: Object
  ▶ 1: {id: "3", number: "PL009000", amount: "9000"}
  ▶ 2: {id: "4", number: "PL009000", amount: "9000"}
  ▶ 3: {id: "5", number: "PL000009", amount: "333"}
    length: 4
    ▶ __proto__: Array(0)
> |

```

Angular PHP example

We'll see a bit later how to display these policies in a table in the component template.

Let's add the other methods to create, update and delete policies in our component.

```

1  createOrUpdatePolicy(form){
2    if(this.selectedPolicy && this.selectedPolicy.id){
3      form.value.id = this.selectedPolicy.id;
4      this.apiService.updatePolicy(form.value).subscribe((policy: Policy)=>{
5        console.log("Policy updated" , policy);
6      });
7    }
8    else{
9
10     this.apiService.createPolicy(form.value).subscribe((policy: Policy)=>{
11       console.log("Policy created, ", policy);
12     });
13   }
14
15 }
16
17 selectPolicy(policy: Policy){
18   this.selectedPolicy = policy;

```



```
19   }
20
21   deletePolicy(id){
22     this.apiService.deletePolicy(id).subscribe((policy: Policy)=>{
23       console.log("Policy deleted, ", policy);
24     });
25   }
```

Adding the Table and Form

Let's now add a table and form to display and create the policies in our dashboard component. Open the `src/app/dashboard/dashboard.component.html` and add the following HTML code:

```
1  <h1>Insurance Policy Management</h1>
2  <div>
3
4    <table border='1' width='100%' style='border-collapse: collapse;'>
5      <tr>
6        <th>ID</th>
7        <th>Policy Number</th>
8        <th>Policy Amount</th>
9        <th>Operations</th>
10
11      </tr>
12
13      <tr *ngFor="let policy of policies">
14        <td>{{ policy.id }}</td>
15        <td>{{ policy.number }}</td>
16        <td>{{ policy.amount }}</td>
17        <td>
18          <button (click)="deletePolicy(policy.id)">Delete</button>
19          <button (click)="selectPolicy(policy)">Update</button>
20        </td>
21      </tr>
22    </table>
```

This is a screenshot of the page at this point:

Insurance Policy Management

| ID | Policy Number | Policy Amount | Operations | |
|----|---------------|---------------|------------|--------|
| 2 | PL005000 | 400 | Delete | Update |
| 3 | PL009000 | 9000 | Delete | Update |
| 4 | PL009000 | 9000 | Delete | Update |
| 5 | PL000009 | 333 | Delete | Update |

PHP Angular Example

Next, below the table, let's add a form that will be used to create and update a policy:

```

1 <br>
2 <form #f = "ngForm">
3   <label>Number</label>
4   <input type="text" name="number" [(ngModel)] = "selectedPolicy.number">
5   <br>
6   <label>Amount</label>
7   <input type="text" name="amount" [(ngModel)] = "selectedPolicy.amount">
8   <br>
9   <input type="button" (click)="createOrUpdatePolicy(f)" value="Create or Update P\
10 olicy">
11 </form>

```

This is a screenshot of the page at this point:

Insurance Policy Management

| ID | Policy Number | Policy Amount | Operations | |
|----|---------------|---------------|------------|--------|
| 2 | PL005000 | 400 | Delete | Update |
| 3 | PL009000 | 9000 | Delete | Update |
| 4 | PL009000 | 9000 | Delete | Update |
| 5 | PL000009 | 333 | Delete | Update |

Number

Amount

Angular PHP CRUD Example

Next, open the `src/app/dashboard/dashboard.component.css` file and the following CSS styles:

```
1  input {
2      width: 100%;
3      padding: 2px 5px;
4      margin: 2px 0;
5      border: 1px solid red;
6      border-radius: 4px;
7      box-sizing: border-box;
8  }
9
10 button, input[type=button]{
11     background-color: #4CAF50;
12     border: none;
13     color: white;
14     padding: 4px 7px;
15     text-decoration: none;
16     margin: 2px 1px;
17     cursor: pointer;
18 }
19 th, td {
20     padding: 1px;
21     text-align: left;
22     border-bottom: 1px solid #ddd;
23
24 }
25 tr:hover {background-color: #f5f5f5;}
```

This is a screenshot of the page after adding some minimal styling:

Insurance Policy Management

| ID | Policy Number | Policy Amount | Operations | |
|----|---------------|---------------|------------|--------|
| 2 | PL0020000 | 10000 | Delete | Update |
| 3 | PL009000 | 9000 | Delete | Update |
| 4 | PL009000 | 9000 | Delete | Update |
| 5 | PL000009 | 333 | Delete | Update |
| 6 | POL10010 | 1000 | Delete | Update |
| 8 | PL0080001 | 5 | Delete | Update |

Number

Amount

Create or Update Policy

PHP Angular REST CRUD Example

Conclusion

In this chapter, we learned how to create a RESTful CRUD application with PHP, MySQL, and Angular.