Learn Angular In 15 Easy Steps

A book by techiediaries.com

Learn Angular 8 in 15 Easy Steps

You'll learn to build an Angular 8 web application from scratch and deploy it to Firebase

Ahmed Bouchefra

This book is for sale at http://leanpub.com/learnangular8ineasysteps

This version was published on 2019-10-18

Contents

Learn Angular 8 in 15 Easy Steps

In this book, we'll learn Angular in 15 easy steps by building an example web application that consumes a REST API.

The REST API will be mocked using json-server[1] which allows you to create a fully-working API based on a JSON file that contains sample data.

Introduction

More often than not, modern web development involves multiple developers working in separate front-end and back-end applications. This approach has many advantages, such as the separation of concerns but also introduces a few challenges such as the difficulties in coordination between the front-end and back-end developers. Here comes the role of tools such as JSON-Server to ease these difficulties.

Most of the times when we hear about mocking, we think of unit testing where we need to mock an object instance before we'll be able test it. But actually we can do more with mocking beyond testing.

In this book, we'll show you how you can increase your development speed and quality by mocking. your backend.

How Can You Increase Development Speed with Mocking?

In most cases, a project is developed by two front-end and back-end teams. When a new project is started, we as front-end developers need to wait for the back-end team to create a REST API that we can consume from our app. But how we can make the two teams work in parallel?

A front-end app is mostly about the UI which needs data from the server. If you don't want to wait for the backend API to be ready, you need a way to mock HTTP data. This, generally, can be done in two approaches:

- The backend developers prepare the stubs and simply return some hard-coded data. This still requires some time before the frontend developers can start working on the app.
- The frontend developers create and use hardcoded data which becomes messy fast.

[1]https://github.com/typicode/json-server

Both approaches have many disadvantages but luckily for Angular developers, there is another way which involves using `json-server` to mock a fully-working REST API in no time with nearly zero-lines of code in most scenarios.

As a front-end developer, JSON-Server is such a great tool that allows you to spin up a REST API server with a fully-working API with zero coding.

In this book, we'll show you how to use JSON-Server to simulate a REST API with literally zero lines of code.

As far as Angular is concerned, there is no real difference between consuming a real and fake REST API. This will allow you to start developing your front-end application even when the back-end is not ready yet.

Angular 8 was released on **May 28, 2019**, and comes with various features and improvements to the Angular CLI and the framework. We now have small bundles and new APIs to hook into the `ng add` and `ng build` commands of the CLI but also a new `ng deploy` command. This book is now updated to the latest Angular 8.3 version. We'll see how to use the new `ng deploy` feature in Angular 8.3+ to easily deploy your Angular application from the command-line to Firebase hosting.

What We'll Cover in This Book?

In this book, we'll cover:

- How to create a fake and complete working JSON REST API,
- How to install Angular CLI,
- How to create an Angular 8 project using Angular CLI,
- How to set up Angular Material and style your application with Material Design,
- How to create Angular components, routing and navigation between them,
- How to create and inject Angular services,
- How to send HTTP GET requests to servers using HttpClient,
- How to use the HttpParams class to add URL query strings in your HttpRequest,
- How to subscribe and unsubscribe from RxJS Observables returned by HttpClient,
- How to handle HTTP errors using the `throwError()` and `catchError()` operators,
- How to retry failed HTTP requests using the RxJS `retry()` operator,
- How to unsubscribe from RxJS Observables returned from HttpClient methods using the `takeUntil()` operator when requests are concelled,
- How to build your application for production and deploy it to Firebase hosting using the new `ng deploy` command available from Angular 8.3+

The steps of this book are as follows:

- Step 1 -Setting up Angular CLI 8

- Step 2 -Initializing a New Angular 8 Example Project
- Step 3 -Setting up a (Fake) JSON REST API
- Step 4 -Setting up Angular HttpClient in our Example Project
- Step 5 -Creating Angular Components
- Step 6 -Adding Angular Routing
- Step 7 -Styling the UI with Angular Material
- Step 8 -Consuming the JSON REST API with Angular HttpClient
- Step 9 -Adding HTTP Error Handling with RxJS `catchError()` & `HttpClient`
- Step 10-Retrying Failed HTTP Requests with RxJS `retry()` & `HttpClient`
- Step 11-Unsubscribing from HttpClient Observables with RxJS `takeUntil()`
- Step 12-Adding URL Query Parameters to the HttpClient get() Method
- Step 13-Getting the Full HTTP Response with Angular HttpClient
- Step 14-Requesting a Typed HTTP Response with Angular HttpClient
- Step 15-Building and Deploying your Angular Application to Firebase Hosting

Let's get started!.

Prerequisites

Before getting started you need a few prerequisites:

- Basic knowledge of TypeScript. Particularly the familiarity with Object Oriented concepts such as TypeScript classes and decorators.
- A local development machine with **Node 8.9+**, together with **NPM 5.5.1+** installed. Node is required by the Angular CLI like the most frontend tools nowadays. You can simply go to the downloads page of the official website[2] and download the binaries for your operating system. You can also refer to your specific system instructions for how to install Node using a package manager. The recommended way though is using NVM[3]-Node Version Manager-a POSIX-compliant bash script to manage multiple active Node.js versions.

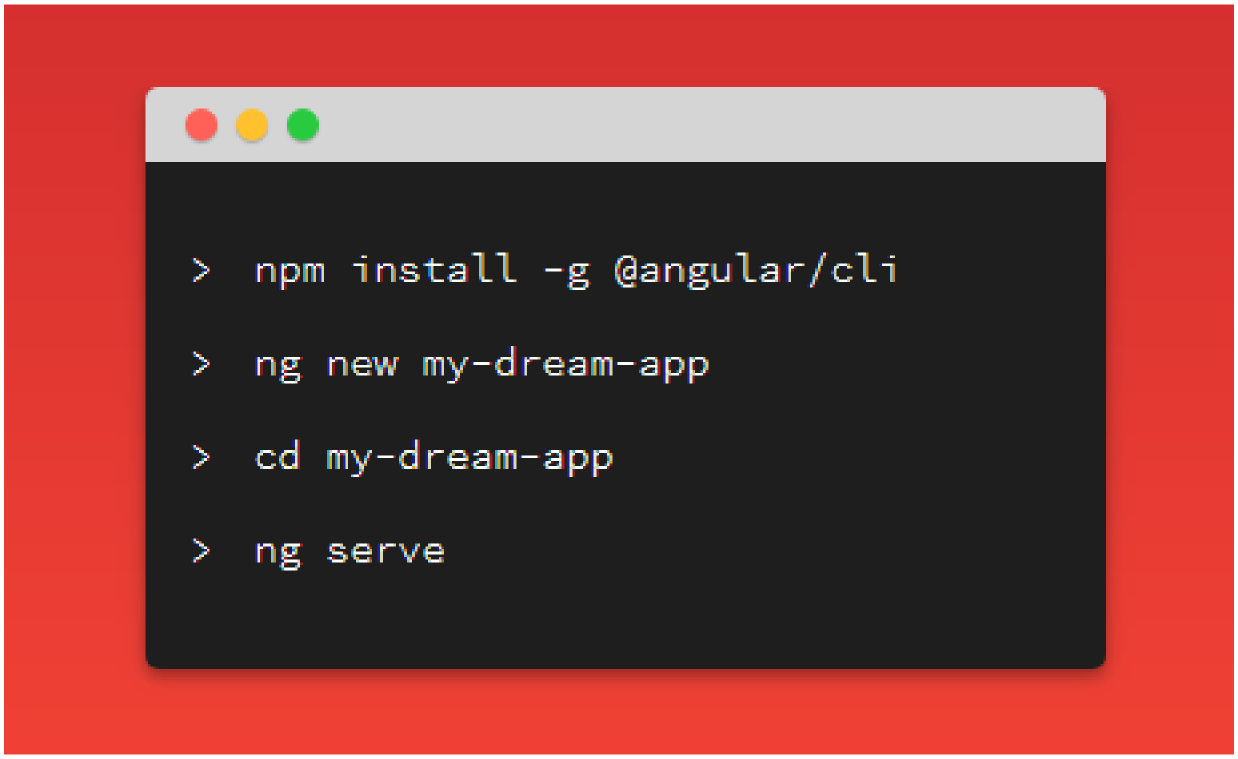
  Note: If you don't want to install a local environment for Angular development but still want to try the code in this book, you can use Stackblitz[4], an online IDE for frontend development that you can use to create an Angular project compatible with Angular CLI.

If you have the previous prerequisites, you are ready for the next steps of our book that will teach you by example how to use Angular HttpClient to send HTTP GET requests for fetching JSON data and the various RxJS operators such as `catchError()`, `tap()`, `retry()`, and `takeUntil()` for implementing advanced features such as error handling, retrying failed HTTP requests and cancelling pending requests. In the first step(s) of our book, we'll see how to install Angular CLI 8 and create an example project from scratch.

[2]https://nodejs.org/downloads
[3]https://github.com/nvm-sh/nvm
[4]https://stackblitz.com/

Step 1 - Setting up Angular CLI 8

In this step, we'll install the latest Angular CLI 8 version (at the time of writing this book).



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

> ng new my-dream-app

> cd my-dream-app

> ng serve
```

Angular CLI

Angular CLI[5] is the official tool for initializing and working with Angular projects. To install it, open a new command-line interface and run the following command:

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

At the time of writing this book, **angular/cli v8.3.2** will be installed on your system. In the next step, we'll learn how to intialize a new example project from the command-line.

Step 2 - Initializing a New Angular 8 Example Project

In this step, we'll proceed to create our example project. Head back to your command-line interface and run the following commands:

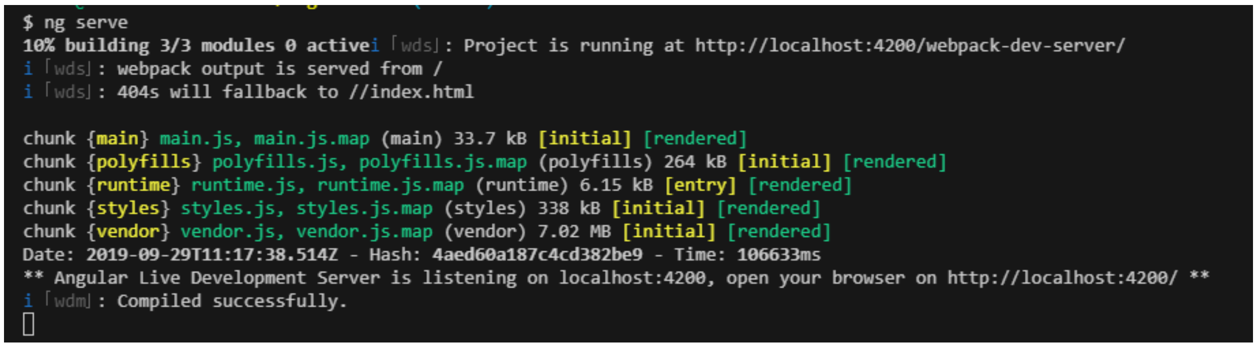
[5] https://cli.angular.io/

```
$ cd ~
$ ng new ngstore
```

The CLI will ask you a couple of questions-If **Would you like to add Angular routing?** Type **y** for Yes and **Which stylesheet format would you like to use?** Choose **CSS**. This will instruct the CLI to automatically set up routing in our project so we'll only need to add the routes for our components to implement navigation in our application. Next, navigate to you project's folder and run the local development server using the following commands:

```
$ cd ngstore
$ ng serve
```

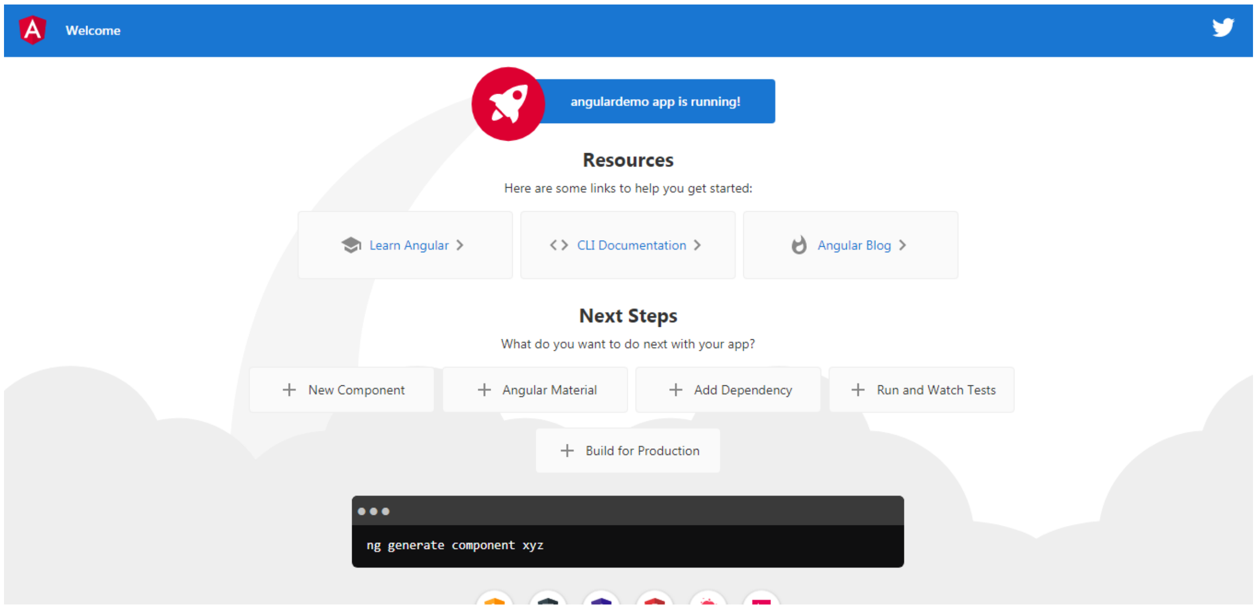
A local development server will start listening on the `http://localhost:4200/` address:



```
$ ng serve
10% building 3/3 modules 0 activei ｢wds｣: Project is running at http://localhost:4200/webpack-dev-server/
i ｢wds｣: webpack output is served from /
i ｢wds｣: 404s will fallback to //index.html

chunk {main} main.js, main.js.map (main) 33.7 kB [initial] [rendered]
chunk {polyfills} polyfills.js, polyfills.js.map (polyfills) 264 kB [initial] [rendered]
chunk {runtime} runtime.js, runtime.js.map (runtime) 6.15 kB [entry] [rendered]
chunk {styles} styles.js, styles.js.map (styles) 338 kB [initial] [rendered]
chunk {vendor} vendor.js, vendor.js.map (vendor) 7.02 MB [initial] [rendered]
Date: 2019-09-29T11:17:38.514Z - Hash: 4aed60a187c4cd382be9 - Time: 106633ms
** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **
i ｢wdm｣: Compiled successfully.
```

Angular CLI Ng Serve

Open your web browser and navigate to the `http://localhost:4200/` address to see your app up and running. This is a screenshot at this point:



angulardemo app is running!

Here are some links to help you get started:

What do you want to do next with your app?

```
ng generate component xyz
```

Angular 8 Project

You should now leave the development server running and start a new command-line interface for running the CLI commands of the next steps.

In the next step, we'll learn how to create a fake JSON REST API that we'll be consuming in our Angular example application.

Step 3-Setting up a (Fake) JSON REST API

Before we proceed to develop our Angular application, we'll need to prepare a JSON REST API that we can consume using `HttpClient`.

We can also consume or fetch JSON data from third-party REST API servers but in this example, we choose to create a fake REST API. Check out this tutorial[6] for a real REST API example. As far as Angular concerned, there is no difference between consuming fake or real REST APIs.

As said, you can either use an external API service, create a real REST API server or create a fake API using `json-server`. In this example we'll use the last approach. So head over to a new command-line interface and start by installing `json-server` from npm in your project:

```
$ cd ~/ngstore
$ npm install-save json-server
```

Next, create a `server` folder in the root folder of your Angular project:

[6]https://www.techiediaries.com/angular-tutorial-example-rest-api-httpclient-get-ngfor

```
$ mkdir server
$ cd server
```

In the `server` folder, create a `database.json` file and add the following JSON object:

```
{
    "products": []
}
```

This JSON file will act as a database for your REST API server. You can simply add some data to be served by your REST API or use Faker.js[7] for automatically generating massive amounts of realistic fake data.

Go back to your command-line, navigate back from the `server` folder, and install `Faker.js` from npm using the following command:

```
$ cd ..
$ npm install faker-save
```

At the time of creating this example, **faker v4.1.0** will be installed. Now, create a `generate.js` file and add the following code:

```
var faker = require('faker');
var database = { products: []};
for (var i = 1; i<= 300; i++) {
 database.products.push({
 id: i,
 name: faker.commerce.productName(),
 description: faker.lorem.sentences(),
 price: faker.commerce.price(),
 imageUrl: "https://source.unsplash.com/1600x900/?product",
 quantity: faker.random.number()
 });
}
console.log(JSON.stringify(database));
```

We first imported faker, next we defined an object with one empty array for products. Next, we entered a *for* loop to create *300* fake entries using faker methods like `faker.commerce.productName()` for generating product names. Check all the available methods[8]. Finally we converted the database object to a string and log it to standard output.

Next, add the `generate` and `server` scripts to the `package.json` file:

[7]https://github.com/marak/Faker.js/
[8]https://github.com/marak/Faker.js/#api-methods

```
"scripts": {
   "ng": "ng",
   "start": "ng serve",
   "build": "ng build",
   "test": "ng test",
   "lint": "ng lint",
   "e2e": "ng e2e",
   "generate": "node ./server/generate.js > ./server/database.json",
   "server": "json-server-watch ./server/database.json"
},
```

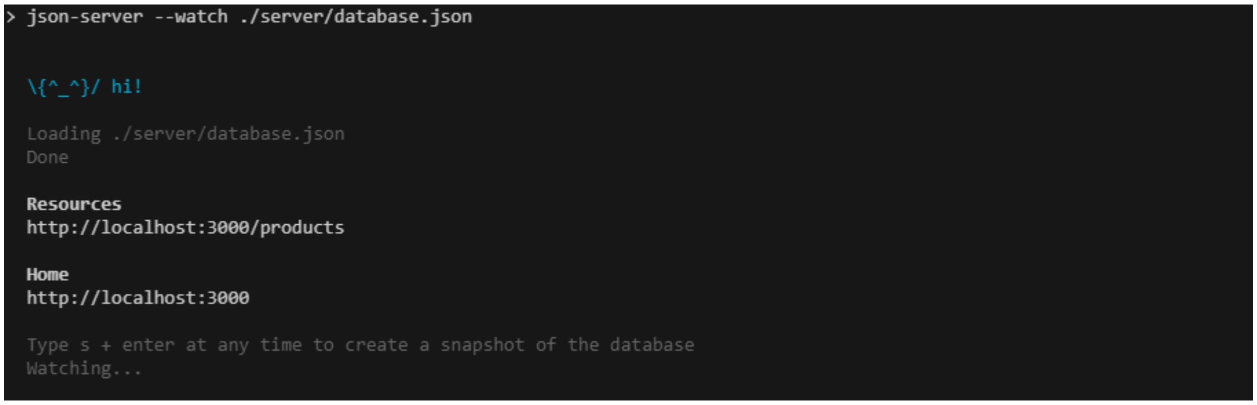
Next, head back to your command-line interface and run the generate script using the following command:

```
$ npm run generate
```

Finally, run the REST API server by executing the following command:

```
$ npm run server
```

You can now send HTTP requests to the server just like any typical REST API server. Your server will be available from the `http://localhost:3000/` address.



```
> json-server --watch ./server/database.json


  \{^_^}/ hi!

  Loading ./server/database.json
  Done

  Resources
  http://localhost:3000/products

  Home
  http://localhost:3000

  Type s + enter at any time to create a snapshot of the database
  Watching...
```

REST API Server

These are the API endpoints we'll be able to use via our JSON REST API server:

- `GET /products` for getting the products,
- `GET /products/<id>` for getting a single product by id,
- `POST /products` for creating a new product,
- `PUT /products/<id>` for updating a product by id,
- `PATCH /products/<id>` for partially updating a product by id,

- `DELETE /products/<id>` for deleting a product by id.

You can use `_page` and `_limit` parameters to get paginated data. In the `Link` header you'll get `first`, `prev`, `next` and `last` links.

For example:

`GET /products?_page=1` for getting the first page of data, `GET /products?_page=1&_limit=5` for getting the first five products of the first page of data.

> **Note**: You can use other features such as filters, sorting and ordering. For more information, check out the docs[9].

Leave the JSON REST API server running and open a new command-line interface for typing the commands of the next steps.

As a summary of what we have done-We installed Angular CLI and initialized a new project based on the latest Angular 8 version. Then, we created a REST API using `json-server` based on a JSON file. In the next step of our book, we'll learn how to set up `HttpClient` in our Angular 8 project.

Step 4-Setting up Angular HttpClient in our Example Project

In this step, we'll proceed to set up the `HttpClient` module in our example.

`HttpClient` lives in a separate Angular module, so we'll need to import it in our main application module before we can use it.

Open your example project with a code editor or IDE. We'll be using Visual Studio Code[10].

Next, open the `src/app/app.module.ts` file, import `HttpClientModule`[11] and add it to the `imports` array of the module as follows:

[9]https://github.com/typicode/json-server
[10]https://code.visualstudio.com
[11]https://angular.io/api/common/http/HttpClientModule#description

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HttpClientModule } from '@angular/common/http';
@NgModule({
 declarations: [
 AppComponent,
 ],
 imports: [
 BrowserModule,
 AppRoutingModule,
 HttpClientModule
 ],
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

That's all, we are now ready to use the `HttpClient` service in our project but before that we need to create a couple of components-The home and about components. This is what we'll learn to do in the next step.

Step 5 - Creating Angular Components

In this step, we'll proceed to create the Angular components that control our application UI. Head back to a new command-line interface and run the following command:

```
$ cd ~/ngstore
$ ng generate component home
```

This is the output of the command:

CREATE src/app/home/home.component.html (19 bytes) CREATE src/app/home/home.component.spec.ts (614 bytes) CREATE src/app/home/home.component.ts (261 bytes) CREATE src/app/home/home.component.css (0 bytes) UPDATE src/app/app.module.ts (467 bytes)

The CLI created four files for the component and added it to the `declarations` array in the `src/app/app.module.ts` file.

Next, let's create the about component using the following command:

```
$ ng generate component about
```

Next, open the `src/app/about/about.component.html` and add the following code:

```
<p style="padding: 13px;">
An Angular 8 example application that demonstrates how to use HttpClient to consume \
REST APIs
</p>
```

We'll update the home component in the following steps. In the next step of our book, we'll add these components to the router.

Step 6-Adding Angular Routing

In this step, we'll proceed to add routing to our example.

Head back to the `src/app/app-routing.module.ts` file, that was automatically created by Angular CLI for routing configuration, and import the components then add the routes as follows:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
const routes: Routes = [
 { path: '', redirectTo: 'home', pathMatch: 'full'},
 { path: 'home', component: HomeComponent },
 { path: 'about', component: AboutComponent },
];
@NgModule({
 imports: [RouterModule.forRoot(routes)],
 exports: [RouterModule]
})
export class AppRoutingModule { }
```

We first imported the home and about components, next we added three routes including a route for redirecting the empty path to the home component, so when the user visits the app, they will be redirected to the home page.

In the next step of our example, we'll set up Angular Material in our project for styling our UI.

Step 7 - Styling the UI with Angular Material

In this step of our book, we'll proceed to add Angular Material to our project and style our application UI. Angular Material[12] provides Material Design components that allow developers to create professional UIs. Setting up Angular Material in our project is much easier now with the new `ng add` command of the Angular CLI v7+. Head back to your command-line interface, and run the following command from the root of your project:

```
$ ng add @angular/material
```

You'll be asked for choosing a theme, choose **Indigo/Pink**. For the other options-**Set up HammerJS for gesture recognition?** and **Set up browser animations for Angular Material?** Simply press **Enter** in your keyboard to choose the default answers.

Next, open the `src/styles.css` file and add a theme:

```
@import "~@angular/material/prebuilt-themes/indigo-pink.css";
```

Each Angular Material component has a separate module that you need to import before you can use the component. Open the `src/app/app.module.ts` file and add the following imports:

```
import { MatToolbarModule,
 MatIconModule,
 MatCardModule,
 MatButtonModule,
 MatProgressSpinnerModule } from '@angular/material';
```

We imported the following modules:

- MatToolbar[13] that provides a container for headers, titles, or actions.
- MatCard[14] that provides a content container for text, photos, and actions in the context of a single subject.
- MatButton[15] that provides a native `<button>` or `<a>` element enhanced with Material Design styling and ink ripples.
- MatProgressSpinner[16] that provides a circular indicator of progress and activity.

Next, you need to include these modules in the `imports` array:

[12] https://material.angular.io
[13] https://material.angular.io/components/toolbar/overview
[14] https://material.angular.io/components/card/overview
[15] https://material.angular.io/components/button/overview
[16] https://material.angular.io/components/progress-spinner/overview

```
@NgModule({
 declarations: [
    AppComponent,
    HomeComponent,
    AboutComponent
 ],
 imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule,
    BrowserAnimationsModule,
    MatToolbarModule,
    MatIconModule,
    MatButtonModule,
    MatCardModule,
    MatProgressSpinnerModule
 ],
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

Next, open the `src/app/app.component.html` file and update it as follows:

```
<mat-toolbar color="primary">
 <h1>
 ngStore
 </h1>
 <button mat-button routerLink="/">Home</button>
 <button mat-button routerLink="/about">About</button>
</mat-toolbar>
<router-outlet></router-outlet>
```

We created the shell of our application containing a top bar with two navigation buttons to the home and about components.

As a summary of what we did until this point of our book-We have setup HttpClient and Angular Material in our project, created the home and about components and configured routing, and finaly added the shell of our application containing a topbar with navigation. In the next step of our book, we'll learn how to fetch the JSON data from our REST API server using `HttpClient`.

Step 8 - Consuming the JSON REST API with Angular HttpClient

In this step, we'll proceed to consume JSON data from our REST API server in our example application.

What is Angular HttpClient?

Front end applications, built using frameworks like Angular communicate with backend servers through REST APIs (which are based on the HTTP protocol) using either the `XMLHttpRequest` interface or the `fetch()` API.

Angular HttpClient makes use of the `XMLHttpRequest` interface that supports both modern and legacy browsers.

The HttpClient is available from the `@angular/common/http` package and has a simplified API interface and powerful features such as easy testability, typed request and response objects, request and response interceptors, reactive APIs with RxJS Observables, and streamlined error handling.

Why Angular HttpClient?

The `HttpClient` builtin service provides many advantages to Angular developers:

- HttpClient makes it easy to send and process HTTP requests and responses,
- HttpClient has many builtin features for implementing test units,
- HttpClient makes use of RxJS Observables for handling asynchronous operations instead of

Promises which simplify common web development tasks such as:

- The concelation of HTTP requests,
- Listenning for the progression of download and upload operations,
- Easy error handling,
- Retrying failed HTTP requests, etc.

Now after introducing HttpClient, let's proceed to building our example application starting with the prerequisites needed to successfully complete the tutorial.

We'll need to create an Angular service for encapsulating the code that deals with consuming data from the REST API server.

A service is a singleton that can be injected by other services and components using the Angular dependency injection.

In software engineering, dependency injection is a technique whereby one object supplies the dependencies of another object. Source[17] Now, let's generate an Angular service that interfaces with the JSON REST API. Head back to your command-line interface and run the following command:

```
$ ng generate service data
```

Next, open the `src/app/data.service.ts` file, import and inject `HttpClient` as follows:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({
 providedIn: 'root'
})
export class DataService {
    private REST_API_SERVER = "http://localhost:3000";
    constructor(private httpClient: HttpClient) { }
}
```

We imported and injected the `HttpClient` service as a private `httpClient` instance. We also defined the `REST_API_SERVER` variable that holds the address of our REST API server. Next, add a `sendGetRequest()` method that sends a GET request to the REST API endpoint to retrieve JSON data:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
@Injectable({
 providedIn: 'root'
})
export class DataService {
    private REST_API_SERVER = "http://localhost:3000";
    constructor(private httpClient: HttpClient) { }
    public sendGetRequest(){
        return this.httpClient.get(this.REST_API_SERVER);
    }
}
```

The method simply invokes the `get()` method of `HttpClient` to send GET requests to the REST API server.

Next, we now need to use this service in our home component. Open the `src/app/home/home.component.ts` file, import and inject the data service as follows:

[17] https://en.wikipedia.org/wiki/Dependency_injection

```
import { Component, OnInit } from '@angular/core';
import { DataService } from '../data.service';
@Component({
 selector: 'app-home',
 templateUrl: './home.component.html',
 styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit {
products = [];
constructor(private dataService: DataService) { }
ngOnInit() {
this.dataService.sendGetRequest().subscribe((data: any[])=>{
 console.log(data);
 this.products = data;
 })
 }
}
```

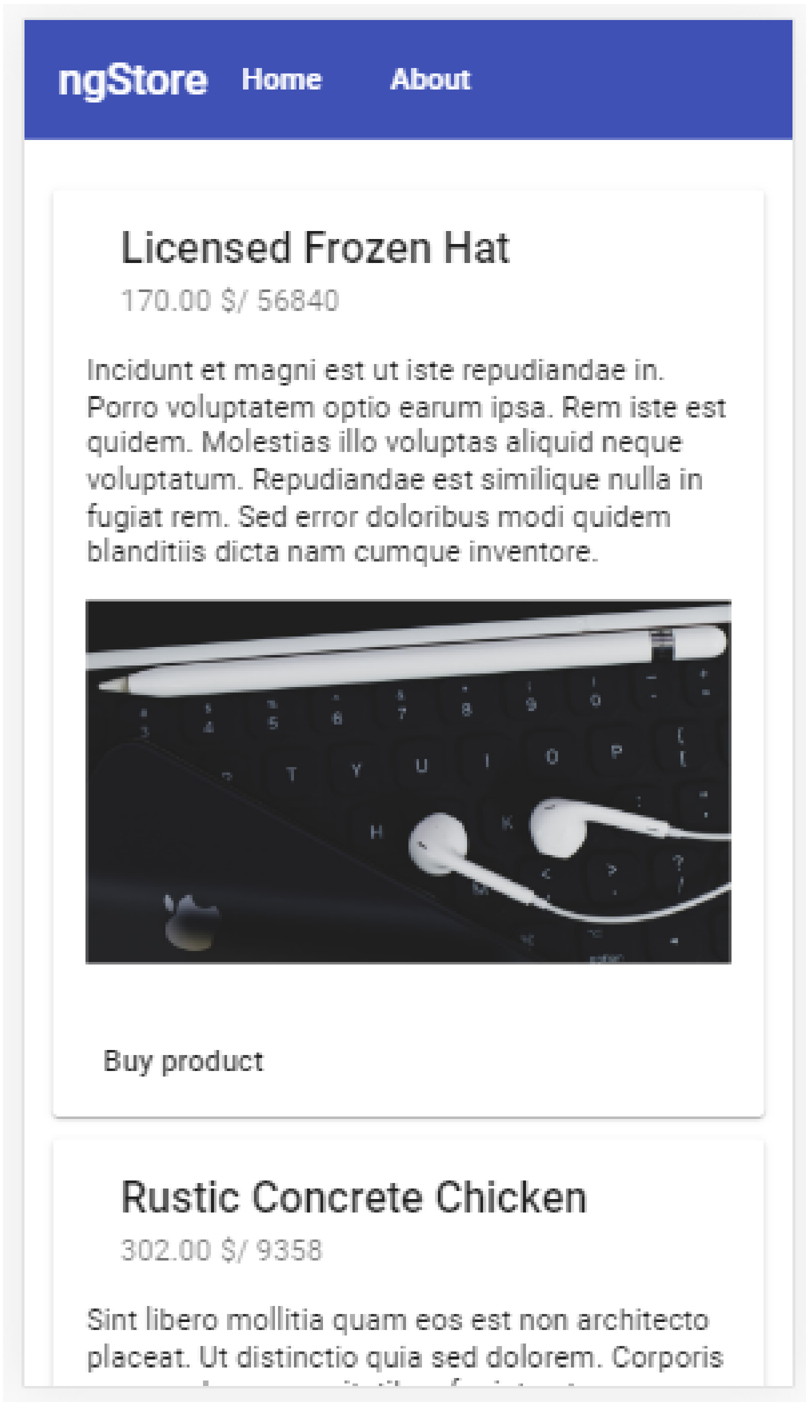
We imported and injected `DataService` as a private `dataService` instance via the component constructor.

Next, we defined a `products` variable and called the `sendGetRequest()` method of the service for fetching data from the JSON REST API server. Since the `sendGetRequest()` method returns the return value of the `HttpClient.get()` method which is an RxJS Observable, we subscribed to the returned Observable to actually send the HTTP GET request and process the HTTP response. When data is received, we added it in the `products` array.

Next, open the `src/app/home/home.component.html` file and update it as follows:

```
<div style="padding: 13px;">
 <mat-spinner *ngIf="products.length === 0"></mat-spinner>
<mat-card *ngFor="let product of products" style="margin-top:10px;">
 <mat-card-header>
 <mat-card-title>{{product.name}}</mat-card-title>
 <mat-card-subtitle>{{product.price}} $/ {{product.quantity}}
 </mat-card-subtitle>
 </mat-card-header>
 <mat-card-content>
 <p>
 {{product.description}}
 </p>
 <img style="height:100%; width: 100%;" src="{{ product.imageUrl }}" />
 </mat-card-content>
 <mat-card-actions>
 <button mat-button> Buy product</button>
 </mat-card-actions>
 </mat-card>
</div>
```

We used the `<mat-spinner>` component for showing a loading spinner when the length of the `products` array equals zero i.e before no data is received from the REST API server. Next, we iterated over the `products` array and used a Material card to display the `name`, `price`, `quantity`, `description` and `image` of each product. This is a screenshot of the home page after JSON data is fetched:



Licensed Frozen Hat

170.00 $/ 56840

Incidunt et magni est ut iste repudiandae in. Porro voluptatem optio earum ipsa. Rem iste est quidem. Molestias illo voluptas aliquid neque voluptatum. Repudiandae est similique nulla in fugiat rem. Sed error doloribus modi quidem blanditiis dicta nam cumque inventore.

Rustic Concrete Chicken

302.00 $/ 9358

Sint libero mollitia quam eos est non architecto placeat. Ut distinctio quia sed dolorem. Corporis

Angular 8 Example

Next, we'll see how to add error handling to our service.

Step 9-Adding HTTP Error Handling with RxJS `catchError()` & `HttpClient`

In this step, we'll proceed to add error handling in our example application.

The Angular's HttpClient methods can be easily used with the `catchError()` operator from RxJS, since they return Observables, via the `pipe()` method for catching and handling errors. We simply need to define a method to handle errors within your service.

There are two types of errors in front-end applications:

- Client-side errors such as network issues and JavaScript syntax and type errors. These errors return `ErrorEvent` objects.
- Server-side errors such as code errors in the server and database access errors. These errors return HTTP Error Responses.

As such, we simply need to check if an error is an instance of `ErrorEvent` to get the type of the error so we can handle it appropriately.

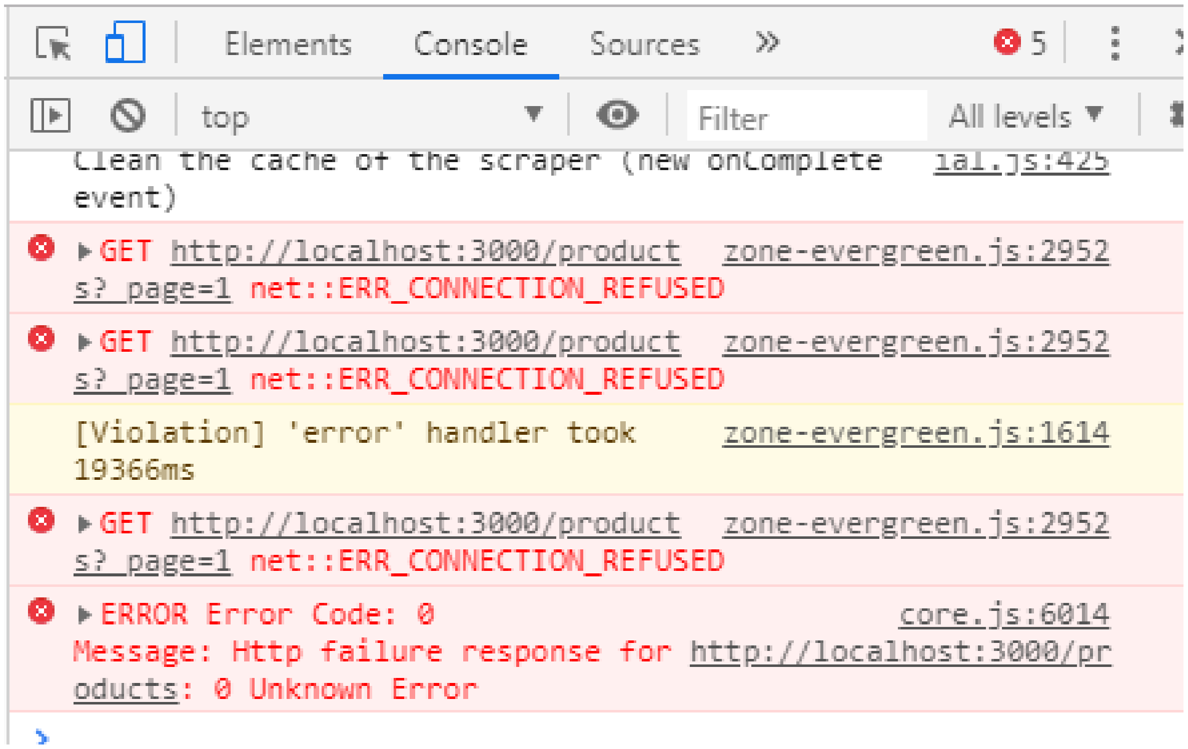
Now, let's see this by example. Open the `src/app/data.service.ts` file and update it accordingly:

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpErrorResponse } from "@angular/common/http";
import { throwError } from 'rxjs';
import { retry, catchError } from 'rxjs/operators';
@Injectable({
 providedIn: 'root'
})
export class DataService {
private REST_API_SERVER = "http://localhost:3000/products";
constructor(private httpClient: HttpClient) { }

 handleError(error: HttpErrorResponse) {
 let errorMessage = 'Unknown error!';
 if (error.error instanceof ErrorEvent) {
 // Client-side errors
 errorMessage = `Error: ${error.error.message}`;
 } else {
 // Server-side errors
 errorMessage = `Error Code: ${error.status}\nMessage: ${error.message}`;
 }
 window.alert(errorMessage);
 return throwError(errorMessage);
```

```
  }

 public sendGetRequest(){
 return this.httpClient.get(this.REST_API_SERVER).pipe(catchError(this.handleError));
  }
}
```

As you can see, this needs to be done for each service in your application which is fine for our example since it only contains one service but once your application starts growing with many services which may all throw errors you need to use better solutions instead of using the `handleError` method per each service which is error-prone. One solution is to handle errors globally in your Angular application using HttpClient interceptors[18].

This is a screenshot of an error on the console if the server is unreachable:



```
Clean the cache of the scraper (new onComplete        ial.js:425
event)
▶GET http://localhost:3000/product    zone-evergreen.js:2952
s?_page=1 net::ERR_CONNECTION_REFUSED
▶GET http://localhost:3000/product    zone-evergreen.js:2952
s?_page=1 net::ERR_CONNECTION_REFUSED
[Violation] 'error' handler took      zone-evergreen.js:1614
19366ms
▶GET http://localhost:3000/product    zone-evergreen.js:2952
s?_page=1 net::ERR_CONNECTION_REFUSED
▶ERROR Error Code: 0                           core.js:6014
Message: Http failure response for http://localhost:3000/pr
oducts: 0 Unknown Error
```

Angular HttpClient Error Example

In the next step, we'll see how to improve our data service by automatically retry sending the failed HTTP requests.

[18]https://angular.io/guide/http#http-interceptors

Step 10 - Retrying Failed HTTP Requests with RxJS `retry()` & `HttpClient`

In this step of our book, we'll see how to use the `retry()` operator of RxJS with HttpClient to automatically resubscribing to the returned Observable which results in resending the failed HTTP requests.

In many cases, errors are temporary and due to poor network conditions so simply trying again will make them go away automatically. For example, in mobile devices network interruptions are frequent so if the user tries again, they may get a successful response. Instead of letting users manually retry, let's see how to do that automatically in our example application.

The RxJS library provides several retry operators. Among them is the `retry()` operator which allows you to automatically re-subscribe to an RxJS Observable a specified number of times. Re-subscribing to the Observable returned from an HttpClient method has the effect of resending the HTTP request to the server so users don't need to repeat the operation or reload the application.

You can use the RxJS `retry()` operator by piping it (using the `pipe()` method) onto the Observable returned from the HttpClient method before the error handler.

Go to the `src/app/data.service.ts` file and import the `retry()` operator:

```
import { retry, catchError } from 'rxjs/operators';
```

Next update the `sendGetRequest()` method as follows:

```
 public sendGetRequest(){
 return this.httpClient.get(this.REST_API_SERVER).pipe(retry(3), catchError(this.han\
dleError));
 }
```

This will retry sending the failed HTTP request three times.

In the next step, we'll see how to unsubscribe from RxJS Observables in our example home component.

Step 11 - Unsubscribing from HttpClient Observables with RxJS `takeUntil()`

In this step of our book, we'll learn about why we need and how to unsubscribe from Observables in our code using the `takeUntil()` operator.

First of all, do you need to unsubscribe from the Observables returned by the `HttpClient` methods? Generally, you need to manually unsubscribe from any subscribed RxJS Observables in your Angular components to avoid memory leaks but in the case of HttpClient, this is automatically handled by Angular by unsubscribing when the HTTP response is received. However, there are some cases when you need to manually unsubscribe for example to cancel pending requests when users are about to leave the component.

We can simply call the `unsubscribe()` method from the `Subscription` object returned by the `subscribe()` method in the `ngOnDestroy()` life-cycle method of the component to unsubscribe from the Observable.

There is also a better way to unsubscribe from or complete Observables by using the `takeUntil()` operator.

The takeUntil()[19] operator emits the values emitted by the source Observable until a notifier Observable emits a value.

Let's see how to use this operator to complete Observables when the component is destroyed.

Open the `src/app/home/home.component.ts` file and update it as follows:

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { DataService } from '../data.service';
import { takeUntil } from 'rxjs/operators';
import { Subject } from 'rxjs';
@Component({
 selector: 'app-home',
 templateUrl: './home.component.html',
 styleUrls: ['./home.component.css']
})
export class HomeComponent implements OnInit, OnDestroy {
products = [];
 destroy$: Subject<boolean> = new Subject<boolean>();
constructor(private dataService: DataService) { }
ngOnInit() {
this.dataService.sendGetRequest().pipe(takeUntil(this.destroy$)).subscribe((data: an\
y[])=>{
 console.log(data);
 this.products = data;
 })
 }
 ngOnDestroy() {
 this.destroy$.next(true);
 // Unsubscribe from the subject
 this.destroy$.unsubscribe();
 }
}
```

[19] https://rxjs.dev/api/operators/takeUntil

We first imported the `OnDestroy` interface, `Subject` and the `takeUntil()` operator. Next, we implemented the `OnDestroy` interface and added the `ngOnDestroy()` lifecycle hook to the component.

Next, we created an instance of `Subject` which can emit boolean values (the type of the value doesn't really matter in this example) that will be used as the notifier of the `takeUntil()` operator.

Next, in the `ngOnInit()` lifecycle hook, we called the `sendGetRequest()` of our data service and called the `pipe()` method of the returned Observable to pipe the `takeUnitl()` operator and finaly subscribed to the combined Observable. In the body of the `subscribe()` method, we added the logic to put the fetched data of the HTTP response in the `products` array. The `takeUntil()` operator allows a notified Observable to emit values until a value is emitted from a notifier Observable.

When Angular destroys a component it calls the `ngOnDestroy()` lifecycle method which, in our case, calls the `next()` method to emit a value so RxJS completes all subscribed Observables. That's it. In this step, we have added the logic to cancel any pending HTTP request by unsubscribing from the returned Observable in case the user descides to navigate away from the component before the HTTP response is received.

In the next step of our book, we'll see how to use URL query parameters with the `get()` method of `HttpClient`.

Step 12-Adding URL Query Parameters to the HttpClient get() Method

In this step, we'll start adding the logic for implementing pagination in our example application.

We'll see how to use URL query parameters via `fromString` and `HttpParams`[20] to provide the appropriate values for the the `_page` and `_limit` parameters of the `/products` endpoint of our JSON REST API server for getting paginated data.

Open the `src/app/data.service.ts` file and start by adding the following the import for `HttpParams`:

```
import { HttpClient, HttpErrorResponse, HttpParams } from "@angular/common/http";
```

Next, update the `sendGetRequest()` method as follows: `ts public sendGetRequest(){ // Add safe, URL encoded_page parameter const options = { params: new HttpParams({fromString: "_-page=1&_limit=20"}) }; return this.httpClient.get(this.REST_API_SERVER, options).pipe(retry(3), catchError(this.handleError)); }`

We used `HttpParams` and `fromString` to create HTTP query parameters from the `_page=1&_limit=20` string. This tells to returns the first page of 20 products.

Now the `sendGetRequest()` will be used to retrieve the first page of data. The received HTTP response will contain a Link header with information about the first, previous, next and last links of data pages.

[20]https://angular.io/guide/http#use-fromstring-to-create-httpparams

In the Link header you'll get first, prev, next and last links. In the next step, we'll see how to extract these pagination links by parsing full HTTP responses.

Step 13 - Getting the Full HTTP Response with Angular HttpClient

In this ste, we'll proceed by implementing the logic for retrieving pagination information from the Link header contained in the HTTP response received from the JSON REST API server.

By default, HttpClient does only provide the response body but in our case we need to parse the Link header for pagination links so we need to tell `HttpClient` that we want the full `HttpResponse`[21] using the `observe` option.

> The Link header in HTTP allows the server to point an interested client to another resource containing metadata about the requested resource.Wikipedia[22]

Go to the src/app/data.service.ts file and import the RxJS `tap()` operator:

```
import { retry, catchError, tap } from 'rxjs/operators';
```

Next, define the following string variables:

```
 public first: string = "";
 public prev: string = "";
 public next: string = "";
 public last: string = "";
```

Next, define the `parseLinkHeader()` method which parses the Link header and populate the previous variables accordingly:

[21]https://angular.io/api/common/http/HttpResponse
[22]https://www.w3.org/wiki/LinkHeader

```
 parseLinkHeader(header) {
 if (header.length == 0) {
 return ;
 }
let parts = header.split(',');
 var links = {};
 parts.forEach( p => {
 let section = p.split(';');
 var url = section[0].replace(/<(.*)>/, '$1').trim();
 var name = section[1].replace(/rel="(.*)"/, '$1').trim();
 links[name] = url;
});

 this.first = links["first"];
 this.last = links["last"];
 this.prev = links["prev"];
 this.next = links["next"];
 }
```

Next, update the `sendGetRequest()` as follows:

```
 public sendGetRequest(){
 // Add safe, URL encoded _page and _limit parameters

 return this.httpClient.get(this.REST_API_SERVER, { params: new HttpParams({fromStri\
ng: "_page=1&_limit=20"}), observe: "response"}).pipe(retry(3), catchError(this.hand\
leError), tap(res => {
 console.log(res.headers.get('Link'));
 this.parseLinkHeader(res.headers.get('Link'));
 }));
 }
```

We added the `observe` option with the `response` value in the options parameter of the `get()` method so we can have the full HTTP response with headers. Next, we use the RxJS `tap()` operator for parsing the Link header before returning the final Observable.

Since the `sendGetRequest()` is now returning an Observable with a full HTTP response, we need to update the home component so open the `src/app/home/home.component.ts` file and import `HttpResponse` as follows:

```
import { HttpResponse } from '@angular/common/http';
```

Next, update the `subscribe()` method as follows:

```
 ngOnInit() {
this.dataService.sendGetRequest().pipe(takeUntil(this.destroy$)).subscribe((res: Htt\
pResponse<any>)=>{
 console.log(res);
 this.products = res.body;
 })
 }
```

We can now access the data from the `body` object of the received HTTP response. Next, go back to the src/app/data.service.ts file and add the following method:

```
 public sendGetRequestToUrl(url: string){
 return this.httpClient.get(url, { observe: "response"}).pipe(retry(3), catchError(t\
his.handleError), tap(res => {
 console.log(res.headers.get('Link'));
 this.parseLinkHeader(res.headers.get('Link'));

 }));
 }
```

This method is similar to `sendGetRequest()` except that it takes the URL to which we need to send an HTTP GET request.

Go back to the `src/app/home/home.component.ts` file and add define the following methods:

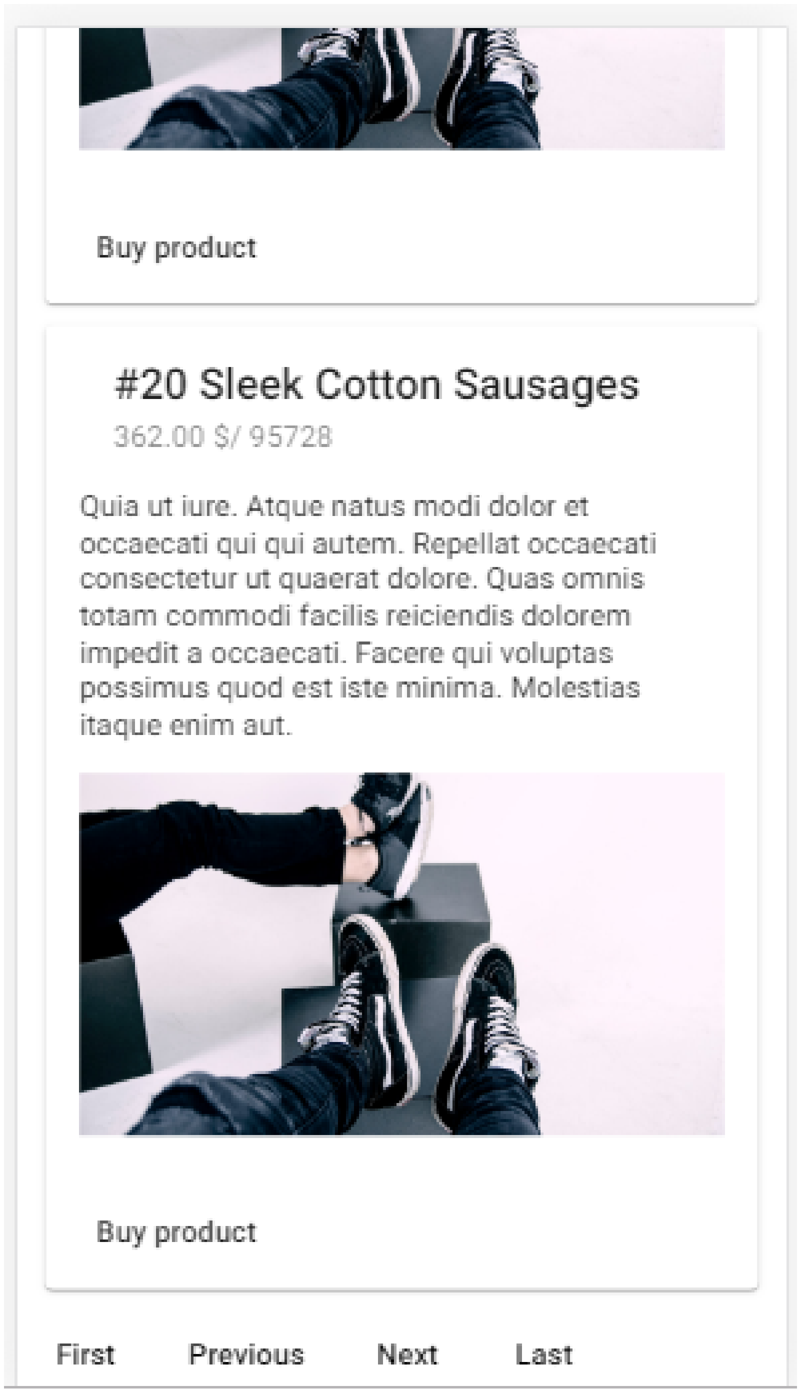
```
 public firstPage() {
 this.products = [];
 this.dataService.sendGetRequestToUrl(this.dataService.first).pipe(takeUntil(this.de\
stroy$)).subscribe((res: HttpResponse<any>) => {
 console.log(res);
 this.products = res.body;
 })
 }
 public previousPage() {
if (this.dataService.prev !== undefined && this.dataService.prev !== '') {
 this.products = [];
 this.dataService.sendGetRequestToUrl(this.dataService.prev).pipe(takeUntil(this.des\
troy$)).subscribe((res: HttpResponse<any>) => {
 console.log(res);
 this.products = res.body;
 })
 }
}
```

```
 public nextPage() {
 if (this.dataService.next !== undefined && this.dataService.next !== '') {
 this.products = [];
 this.dataService.sendGetRequestToUrl(this.dataService.next).pipe(takeUntil(this.des\
troy$)).subscribe((res: HttpResponse<any>) => {
 console.log(res);
 this.products = res.body;
 })
 }
 }
 public lastPage() {
 this.products = [];
 this.dataService.sendGetRequestToUrl(this.dataService.last).pipe(takeUntil(this.des\
troy$)).subscribe((res: HttpResponse<any>) => {
 console.log(res);
 this.products = res.body;
 })
 }
```

Finally, add open the `src/app/home/home.component.html` file and update the template as follows:

```
<div style="padding: 13px;">
 <mat-spinner *ngIf="products.length === 0"></mat-spinner>
<mat-card *ngFor="let product of products" style="margin-top:10px;">
 <mat-card-header>
 <mat-card-title>#{{product.id}} {{product.name}}</mat-card-title>
 <mat-card-subtitle>{{product.price}} $/ {{product.quantity}}
 </mat-card-subtitle>
 </mat-card-header>
 <mat-card-content>
 <p>
 {{product.description}}
 </p>
 <img style="height:100%; width: 100%;" src="{{ product.imageUrl }}" />
 </mat-card-content>
 <mat-card-actions>
 <button mat-button> Buy product</button>
 </mat-card-actions>
 </mat-card>
</div>
<div>
 <button (click) ="firstPage()" mat-button> First</button>
 <button (click) ="previousPage()" mat-button> Previous</button>
 <button (click) ="nextPage()" mat-button> Next</button>
 <button (click) ="lastPage()" mat-button> Last</button>
</div>
```

This is a screenshot of our application:



#20 Sleek Cotton Sausages

362.00 $/ 95728

Quia ut iure. Atque natus modi dolor et occaecati qui qui autem. Repellat occaecati consectetur ut quaerat dolore. Quas omnis totam commodi facilis reiciendis dolorem impedit a occaecati. Facere qui voluptas possimus quod est iste minima. Molestias itaque enim aut.

Step 14 - Requesting a Typed HTTP Response with Angular HttpClient

In this step, we'll see how to use typed HTTP responses in our example application.

Angular HttpClient allows you to specify the type of the response object in the request object, which make consuming the response easier and straightforward. This also enables type assertion during the compile time.

Let's start by defining a custom type using a TypeScript interface with the required properties. Head back to your command-line interface and run the following command from the root of your project:

```
$ ng generate interface product
```

Next, open the `src/app/product.ts` file and update it as follows:

```
export interface Product {
 id: number;
 name: string;
 description: string;
 price: number;
 quantity: number;
 imageUrl: string;
}
```

Next, specify the `Product` interface as the `HttpClient.get()` call's type parameter in the data service. Go back to the `src/app/data.service.ts` file and import the `Product` interface:

```
import { Product } from './product';
```

Next:

```
 public sendGetRequest(){
return this.httpClient.get<Product[]>(this.REST_API_SERVER, { params: new HttpParams\
({fromString: "_page=1&_limit=20"}), observe: "response"}).pipe(retry(3), catchError\
(this.handleError), tap(res => {
 console.log(res.headers.get('Link'));
 this.parseLinkHeader(res.headers.get('Link'));

 }));
 }
public sendGetRequestToUrl(url: string){
 return this.httpClient.get<Product[]>(url, { observe: "response"}).pipe(retry(3), c\
atchError(this.handleError), tap(res => {
 console.log(res.headers.get('Link'));
 this.parseLinkHeader(res.headers.get('Link'));

 }));
 }
```

Next, open the `src/app/home/home.component.ts` file and import the `Product` interface:

```
import { Product } from '../product';
```

Next change the type of the `products` array as follows:

```
export class HomeComponent implements OnInit, OnDestroy {
products: Product[] = [];
```

Next chnage the type of the HTTP response in the `sendGetRequest()` call:

```
 ngOnInit() {
this.dataService.sendGetRequest().pipe(takeUntil(this.destroy$)).subscribe((res: Htt\
pResponse<Product[]>) => {
 console.log(res);
 this.products = res.body;
 })
 }
```

You also need to do the same for the other `firstPage()`, `previousPage()`, `nextPage()` and `lastPage()` methods.

Step 15 - Building and Deploying your Angular Application to Firebase Hosting

In this step, we'll see how to build and deploy our example application to Firebase hosting using the ng deploy command available in Angular 8.3+.

We'll only see how to deploy the frontend application without the fake JSON server. Angular CLI 8.3+ introduced a new `ng deploy` command that makes it more easier than before to deploy your Angular application using the deploy CLI builder assocaited with your project. There are many third-party builders that implement deployment capabilities for different platforms. You can add any of them to your project by running the `ng add` command.

After adding a deployment package it will automatically update your workspace configuration (i.e the `angular.json` file) with a deploy section for the selected project. You can then use the `ng deploy` command to deploy that project.

Let's now see that by example by deploying our project to Firebase hosting.

Head back to your command-line interface, make sure you are inside the root folder of your Angular project and run the following command:

```
$ ng add @angular/fire
```

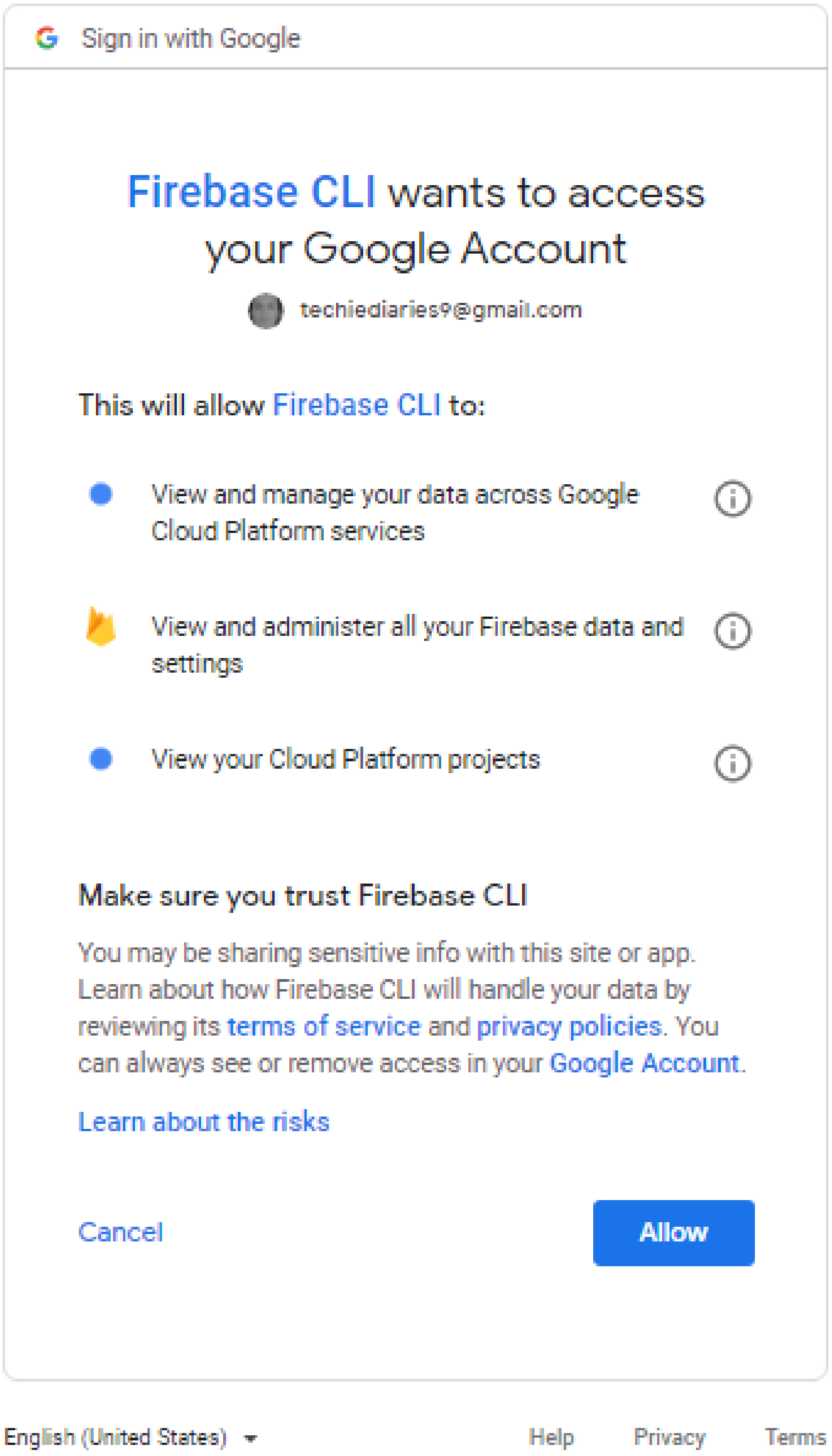
This will add the Firebase deployment capability to your project.

As the time of writing this book, **@angular/fire v5.2.1** will be installed.

The command will also update the `package.json` of our project by adding this section:

```
 "deploy": {
 "builder": "@angular/fire:deploy",
 "options": {}
 }
```

The CLI will prompt you to **Paste authorization code here**: and will open your default web browser and ask you to give Firebase CLI permissions to administer your Firebase account:


Sign in with Google

Firebase CLI wants to access your Google Account

techiediaries9@gmail.com

This will allow Firebase CLI to:

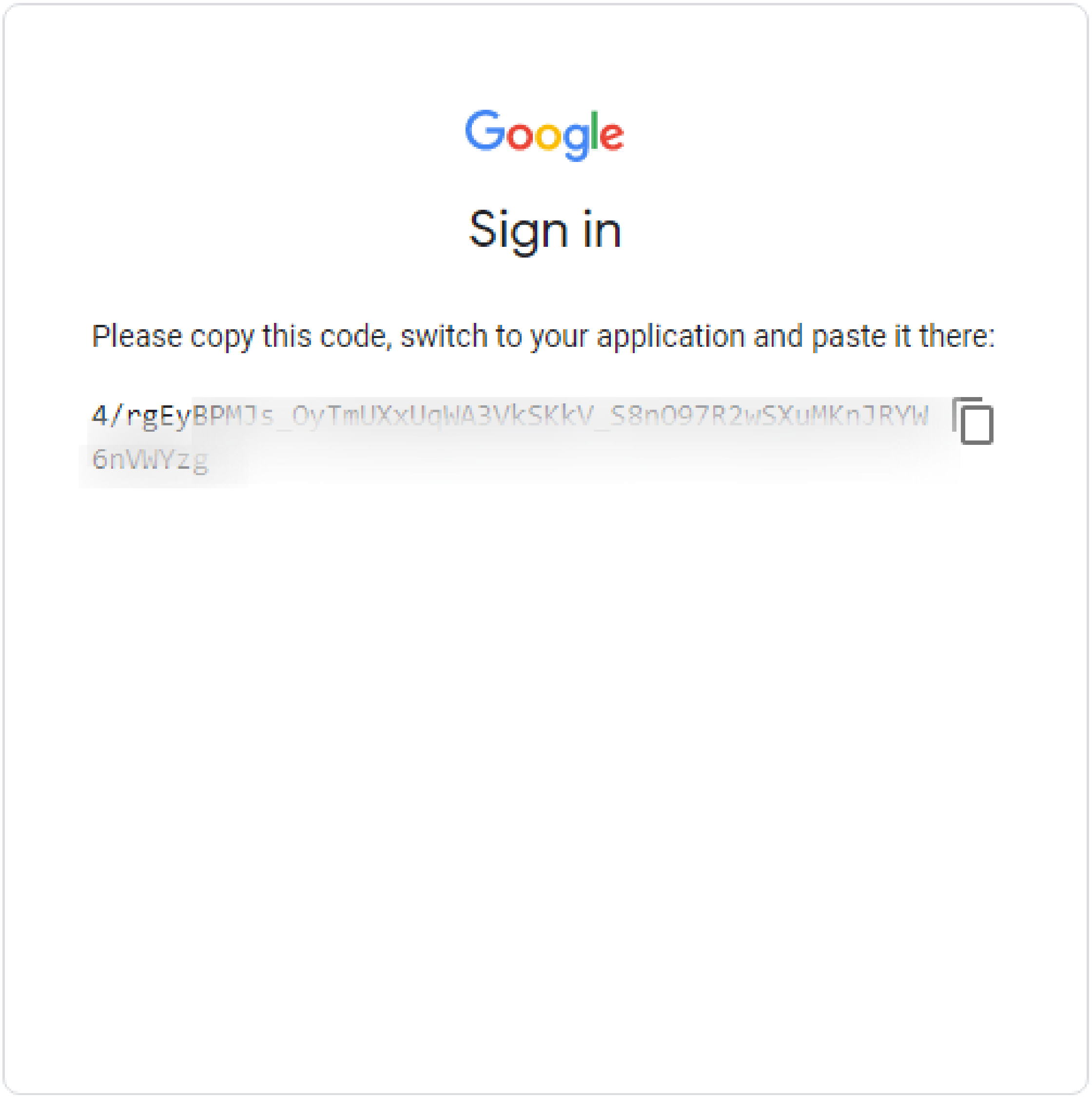
- View and manage your data across Google Cloud Platform services
- View and administer all your Firebase data and settings
- View your Cloud Platform projects

Make sure you trust Firebase CLI

You may be sharing sensitive info with this site or app. Learn about how Firebase CLI will handle your data by reviewing its terms of service and privacy policies. You can always see or remove access in your Google Account.

Learn about the risks

After you signin with the Google account associated with your Firebase account, you'll be given the authorization code:



Sign in

Please copy this code, switch to your application and paste it there:

```
4/rgEy[redacted]
```

Next, you'll be prompted: **Please select a project: (Use arrow keys or type to search)**. You should have created a Firebase project before.

The CLI will create the `firebase.json` and `.firebaserc` files and update the `angular.json` file accordingly.

Next, deploy your application to Firebase, using the following command:

```
$ ng deploy
```

The command will produce an optimized build of your application (equivalent to the `ng deploy -prod` command), it will upload the production assets to Firebase hosting.

Conclusion

Throughout this book, we've built a complete working Angular application example using the latest Angular 8.3+ version.

You learned to mock a REST API backend for your Angular application with nearly zero-lines of code.

You learned to create a project using Angular CLI, add HttpClient and Angular Material for sending HTTP requests to your mocked REST API backend and styling the UI with Material Design components.

You have particularly seen how send HTTP GET requests with parameters using the `get()` method, how to handle HTTP errors using the RxJS `throwError()` and `catchError()` operators, unsubscribe from RxJS Observables for the cancelled HTTP requests using the `takeUntil()` operator and retry failed requests with the `retry()` operator.

Finally, you learned to deploy your Angular application to Firebase using the `ng deploy` command available starting from Angular 8.3+.