# LEARN COMPLETE ANGULAR IN 15 STEPS

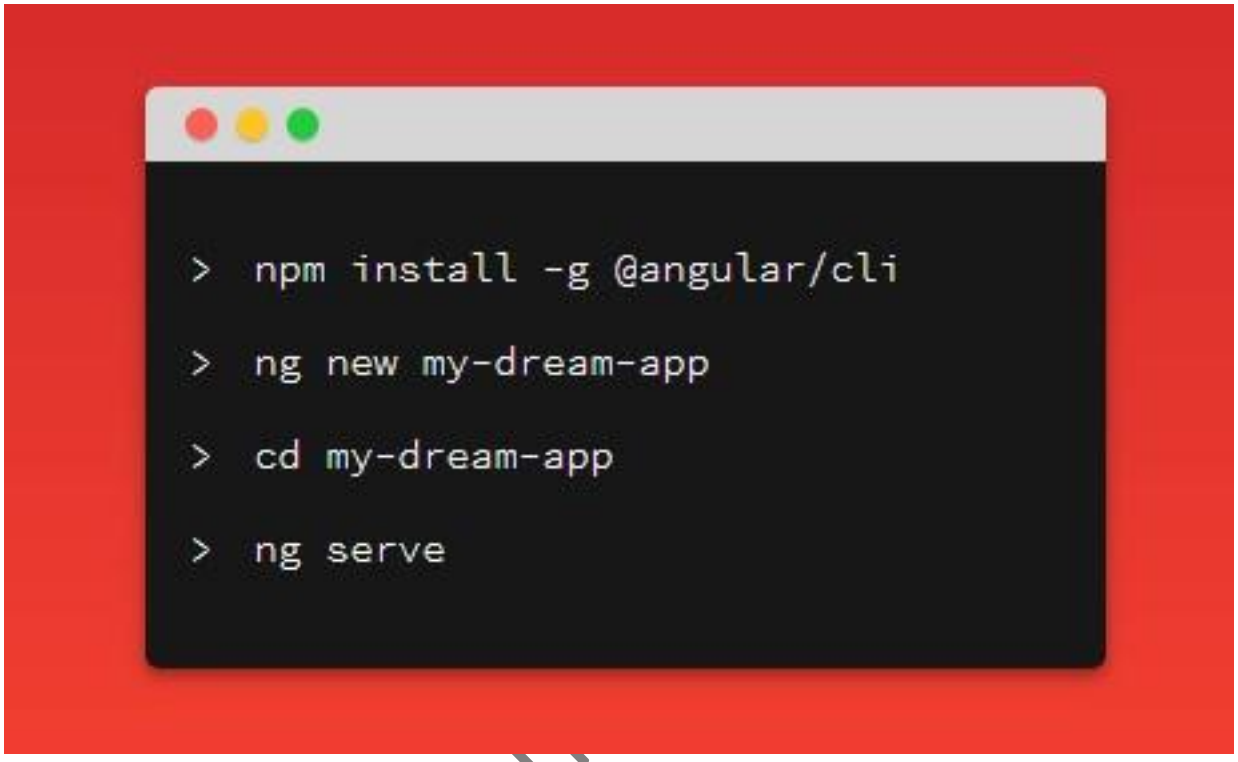
Curated By- Himanshu Kumar ; Download here -> https://t.me/the_rising_engineers

## Below are the 15 Easy Steps-

- 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.

# 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).




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:

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 command.

```
$ 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:

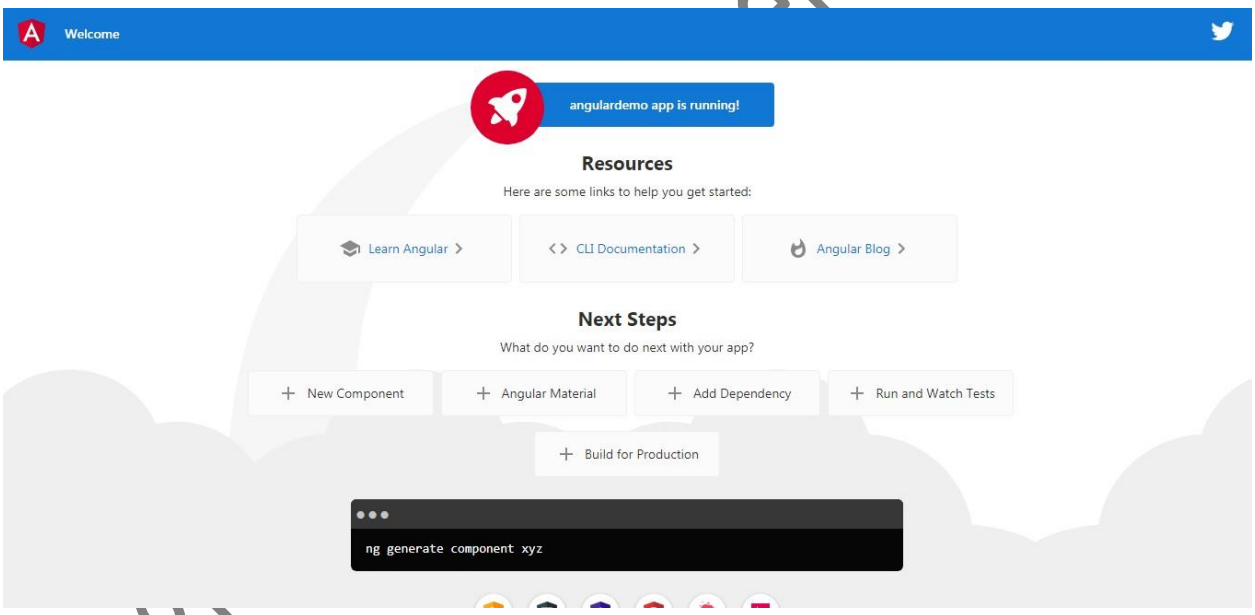
```
$ cd ngstore
$ ng serve
```

A local development server will start listeningon 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:




Angular 8 Project



**Download here -> https://t.me/the_rising_engineers**

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

In the next step, we'll learn how to create a fake JSON REST AP I 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"
  },
```

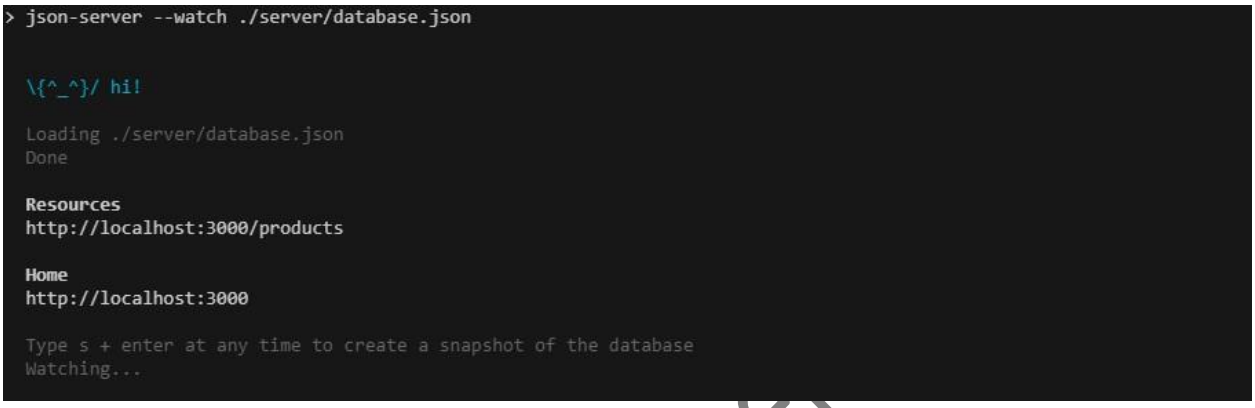
Next, head https://youtu.be/A6eHr0spgqs back to your command-line interface and run the generate script using the following command:

```
$ npm run generate
```

Finally,run the RESTAPIserver by executing the following command:

```
1 $ npm run server
```

You can now send H T T P requests to the server just like any typical REST AP I server. Your server will be available from the http://localhost:3000/ address.




REST API Server

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

GET the 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 AP I 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 AP I 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 thisstep,we'llproceedto setupthe 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.

Openyourexampleproject with a codeeditoror I DE.We'll be usingVisualStudioCode[10].

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

```
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 { HttpClientModule } from '@angular/common/http';
6    @NgModule({
7    declarations: [
8    AppComponent,
9    ],
10   imports: [
11   BrowserModule,
12   AppRoutingModule,
13   HttpClientModule
14   ],
15   providers: [],
16   bootstrap: [AppComponent]
17   })
18   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 U I. Head back to a new command-line interface and run the following command:

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

This is theoutputof thecommand:

CREAT E src/app/home/home.component.html(19bytes)CREAT E src/app/home/home.component.spec.ts

(614 bytes) CREAT E src/app/home/home.component.ts (261 bytes) CREAT E src/app/home/home.component.css (0 bytes) U P DAT E 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'screate the aboutcomponent using thefollowingcommand:

```
$ ng generate component about
```

Next,openthe 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 thisstep,we'llproceedto addroutingto ourexample.

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 nextstep of ourexample, we'llset upAngular Material in our projectfor 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 U I. Angular Material[1] provides Material Design components that allow developers to create professional U Is. 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,
```

```
4    MatButtonModule,
5    MatProgressSpinnerModule } from '@angular/material';
```



We imported the following modules:

- MatToolbar[2] 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[3456] that provides a circular indicator of progress and activity.

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

```
1    @NgModule({
2    declarations: [
3    AppComponent,
4    HomeComponent,
5    AboutComponent
6    ],
7    imports: [
8    BrowserModule,
9    AppRoutingModule,
10   HttpClientModule,
11   BrowserAnimationsModule,
12   MatToolbarModule,
13   MatIconModule,
14   MatButtonModule,
15   MatCardModule,
16   MatProgressSpinnerModule
17   ],
18   providers: [],
19   bootstrap: [AppComponent]
20   })
```

```
21        export class AppModule { }
```



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

```
1   <mat-toolbar color="primary">
2   <h1>
3   ngStore
4   </h1>
5   <button mat-button routerLink="/">Home</button>
6   <button mat-button routerLink="/about">About</button>
7   </mat-toolbar>
8   <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 AP I 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 AP Is (which are based on the H T T P protocol) using either the XMLHttpRequest interface or the fetch() AP I.

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 AP I interface and powerful features such as easy testability, typed request and response objects, request and response interceptors, reactive AP Is with RxJS Observables, and streamlined error handling.

## Why Angular HttpClient?

The Http Client built in service provides many advantages toAngular developers:

- HttpClient makes it easy to send and process H T T P 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 developmenttasks such as:

- The concelation of H T T P requests,
- Listenning for the progression of download and upload operations,
- Easyerror handling,
- Retrying failed H T T P 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 AP I. 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 {
```

```
8       private REST_API_SERVER = "http://localhost:3000";
9       constructor(private httpClient: HttpClient) { }
10      }
```

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 AP I server. Next, add a sendGetRequest() method that sends a GET request to the REST AP I endpoint to retrieve JSON data:

```
1           import { Injectable } from '@angular/core';
2           import { HttpClient } from '@angular/common/http';
3           @Injectable({
4           providedIn: 'root'
5           })
6           export class DataService {
7           private REST_API_SERVER = "http://localhost:3000";
8           constructor(private httpClient: HttpClient) { }
9           public sendGetRequest(){
10          return this.httpClient.get(this.REST_API_SERVER);
11          }
12          }
```

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:

```
1   import { Component, OnInit } from '@angular/core';
2   import { DataService } from '../data.service';
3   @Component({
4   selector: 'app-home',
5   templateUrl: './home.component.html',
6   styleUrls: ['./home.component.css']
7   })
8   export class HomeComponent implements OnInit {
9   products = [];
10  constructor(private dataService: DataService) { }
11  ngOnInit() {
12  this.dataService.sendGetRequest().subscribe((data: any[])=>{
13  console.log(data);
14  this.products = data;
15  })
16  }
17  }
```

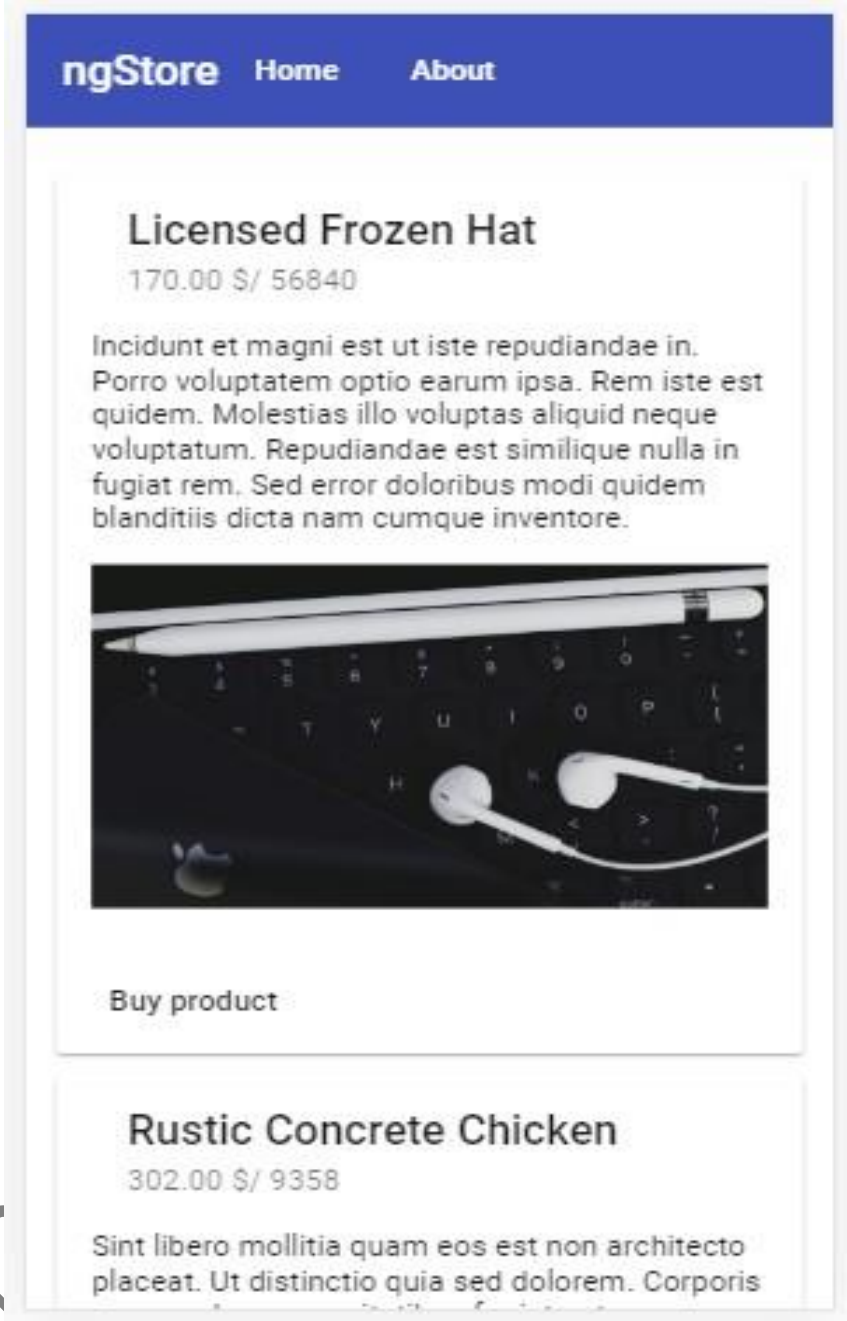
We imported and injected DataService as a private dataService instance via the component constructor.

Next, we defined a products variable and called the sendGetRequest() method of the service for fetching data from the JSON REST AP I 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 H T T P GET request and process the H T T P 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 AP I 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:




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 thisstep,we'llproceedto adderrorhandlingin ourexample 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.

Thereare twotypes oferrors infront-endapplications:

- 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 H T T P 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.

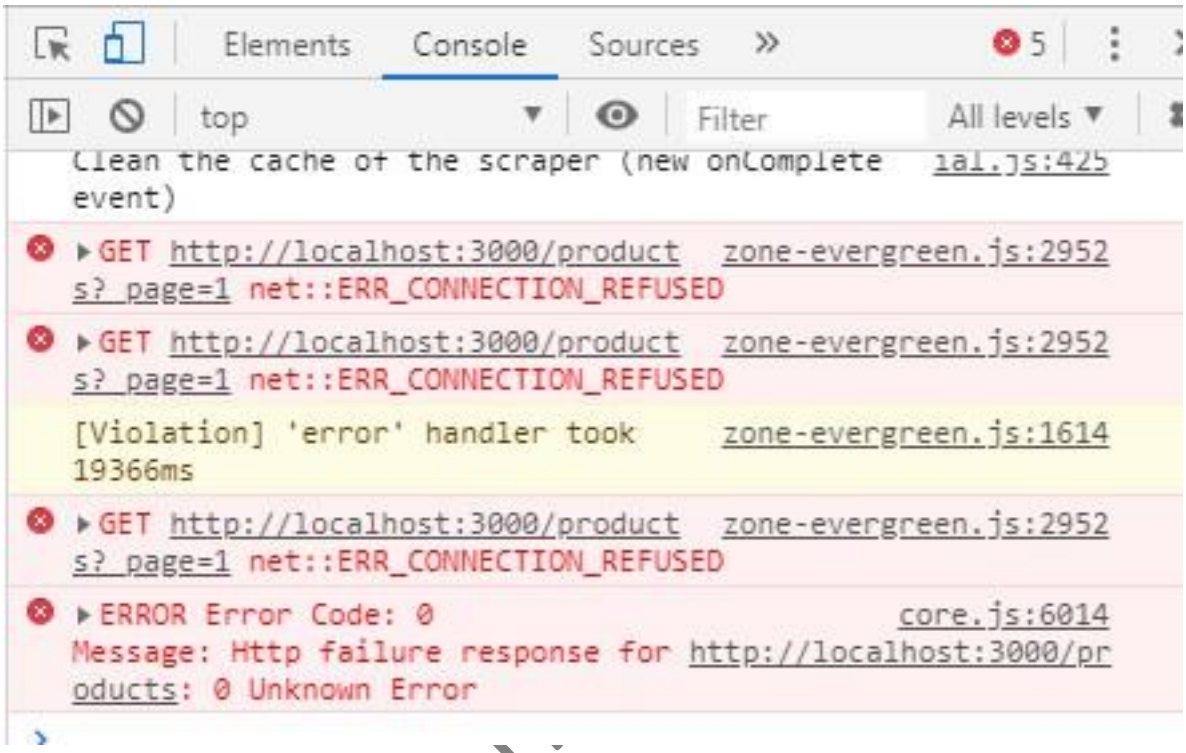
Now,let's see this byexample. Open the src/app/data.service.ts file and update it accordingly:

```
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 ascreenshot of an error on the console if the server is unreachable:




Angular HttpClient Error Example

In the next step, we'll see how to improve our data service by automatically retry sending the failed H T T P requests.



## 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 H T T P 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 H T T P

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:

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

Next update the sendGetRequest() method as follows:

```
1   public sendGetRequest(){
2   return this.httpClient.get(this.REST_API_SERVER).pipe(retry(3), catchError(this.han\
3 dleError));
4   }
```

This will retry sending the failed H T T P 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 H T T P 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();
} 26 }
```

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 H T T P 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 H T T P request by unsubscribing from the returned Observable in case the user descides to navigate away from the component before the H T T P 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:

```
1 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"})};returnthis.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.

---

## 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 H T T P allows the server to point an interested client to another resource containing metadata about the requested resource.Wikipedia[22]

Go to thesrc/app/data.service.tsfile 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:

```
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\
```

```
6     leError), tap(res => {
7     console.log(res.headers.get('Link'));
8     this.parseLinkHeader(res.headers.get('Link'));
9     }));
10    }
```



We added the observe option with the response value in the options parameter of the get() method so we can have the full H T T P 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 H T T P response, we need to update the home component so open the src/app/home/home.component.ts file and import HttpResponse as follows:

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

Next, update the subscribe() method as follows:

```
1   ngOnInit() {
2   this.dataService.sendGetRequest().pipe(takeUntil(this.destroy$)).subscribe((res: Htt\
3   pResponse<any>)=>{ 4 console.log(res);
5   this.products = res.body;
6   })
7   }
```

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

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

```
6
7   }));
8   }
```

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:

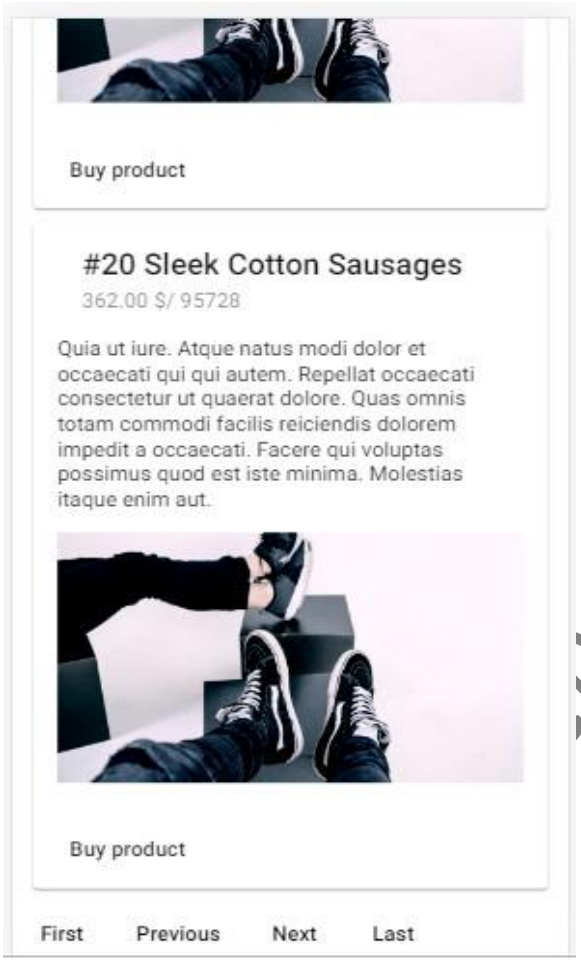
```
1   public firstPage() {
2   this.products = [];
3   this.dataService.sendGetRequestToUrl(this.dataService.first).pipe(takeUntil(this.de\
4   stroy$)).subscribe((res: HttpResponse<any>) => {
5   console.log(res);
6   this.products = res.body;
7   })
8   }
9   public previousPage() {
10  if (this.dataService.prev !== undefined && this.dataService.prev !== '') {
11  this.products = [];
12  this.dataService.sendGetRequestToUrl(this.dataService.prev).pipe(takeUntil(this.des\
13  troy$)).subscribe((res: HttpResponse<any>) => {
14  console.log(res);
15  this.products = res.body;
16  })
17  }
18  }
19  public nextPage() {
20  if (this.dataService.next !== undefined && this.dataService.next !== '') {
21  this.products = [];
22  this.dataService.sendGetRequestToUrl(this.dataService.next).pipe(takeUntil(this.des\
23  troy$)).subscribe((res: HttpResponse<any>) => {
24  console.log(res);
25  this.products = res.body;
26  })
27  }
28  }
29  public lastPage() {
30  this.products = [];
31  this.dataService.sendGetRequestToUrl(this.dataService.last).pipe(takeUntil(this.des\
32  troy$)).subscribe((res: HttpResponse<any>) => {
33  console.log(res);
34  this.products = res.body;
35  })
```

```
}
```

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:




# Step 14-Requesting a Typed HTTP Response with Angular HttpClient

In this step, we'll see how to use typedH T T Presponsesinour 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:

```
1 $ ng generate interface product
```

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

```
1  export interface Product {
2  id: number;
3  name: string;
4  description: string;
5  price: number;
6  quantity: number;
```

```
7    imageUrl: string;
8    }
```

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:

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

Next:

```
1    public sendGetRequest(){
2    return this.httpClient.get<Product[]>(this.REST_API_SERVER, { params: new HttpParams\
3    ({fromString: "_page=1&_limit=20"}), observe: "response"}).pipe(retry(3), catchError\
4    (this.handleError), tap(res => {
5    console.log(res.headers.get('Link'));
6    this.parseLinkHeader(res.headers.get('Link'));
7
8    }));
9    }
10   public sendGetRequestToUrl(url: string){
11   return this.httpClient.get<Product[]>(url, { observe: "response"}).pipe(retry(3), c\
12   atchError(this.handleError), tap(res => {
13   console.log(res.headers.get('Link'));
14   this.parseLinkHeader(res.headers.get('Link'));
15
16   }));
17   }
```

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

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

Next change the type of the products array as follows:

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

Next chnage the type of theH T T Presponsein the sendGetRequest() call:

```
1    ngOnInit() {
2    this.dataService.sendGetRequest().pipe(takeUntil(this.destroy$)).subscribe((res: Htt\
3    pResponse<Product[]>) => { 4       console.log(res);
5    this.products = res.body;
6    })
```

```
}
```

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.

AngularCLI

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 associated 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
```

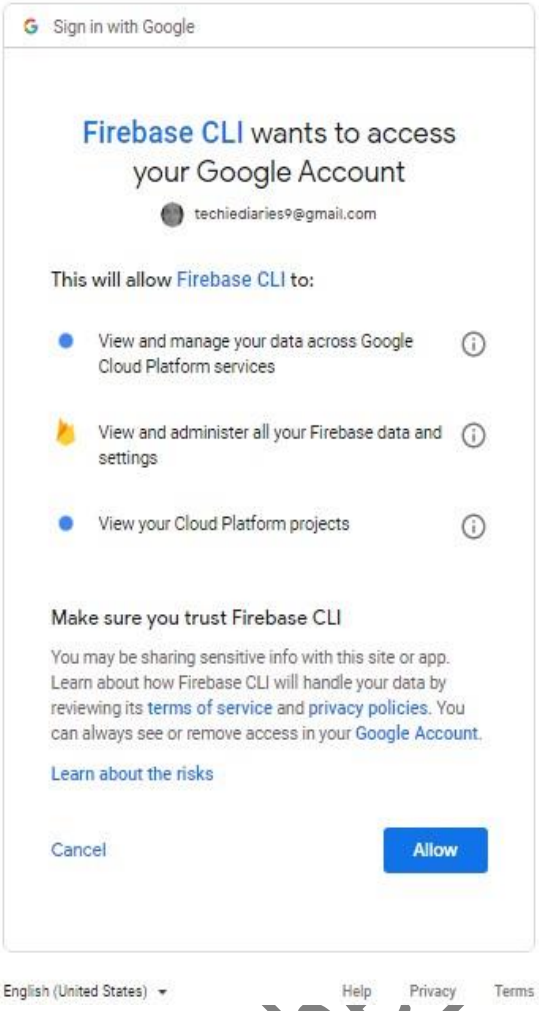
This will add the Firebase deployment capability to your project.

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

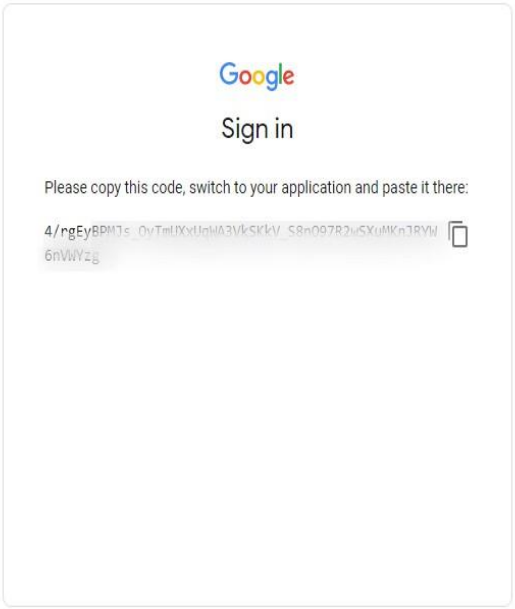
The command will also update the package.json of our project by adding this section:

```
"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:




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




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,deployyourapplicationto Firebase,using thefollowingcommand:

```
1 $ ng deploy
```

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