

Build A Full-stack Web Application Using



Angular And Firebase

Author

Ankit Sharma



C#Corner

Preface

Angular is an open-source framework that allows us to create applications for multiple platforms such as web, mobile web, native mobile, and native desktop. It is one of the most popular Single Page Application frameworks.

In this book, we are going to create a blogging application using Angular on the frontend and Google cloud Firestore as our database. We will also learn how to deploy the application on Firebase.

Our application will have the following features:

- Material design
- Add a new blog post
- Edit an existing blog
- Delete an existing blog
- Authorization with Google account
- Role-based authentication
- Pagination for the blogs
- Post comment on each blog
- Option to share the blog on social channels

We will learn about the following Angular concepts in this book:

- Using Cloud Firestore with an Angular application
- Angular Material and Bootstrap
- Template-driven forms
- Form validation
- Custom pipes
- Auth-guards in Angular
- Authentication and Authorization
- Login with Google using Firebase
- Social share option using ngx-share
- Client-Side pagination using ngx-pagination

A working sample of the app is deployed at <https://blogsite-30c69.firebaseio.com/>.

Navigate to the URL and see what we are going to build in this book.

By the end of this book, you will have mastered the advanced concepts of the Angular framework. You will be able to create a rich and interactive web application using Angular and Google's Firebase.

About The Author



Ankit Sharma is a Software Engineer currently working as Senior Member Technical with ADP in Hyderabad, India. He has over 6 years of extensive experience in Microsoft technologies including C#, ASP.NET, SQL Server and UI technologies such as JQuery, Angular and Blazor.

Ankit is a technical author and speaker and loves to contribute to the open-source community. He writes articles for multiple platforms, which include C-sharpcorner, Dzone, Medium and TechNet Wiki. For his dedicated contribution to the developer's community, he has been recognized as the Google Developer Expert for Angular, a C-sharpcorner MVP and a Dzone MVB. He is also the author of the first-ever book on Blazor – Blazor Quick Start Guide. You can tweet him @ankitsharma_007.

Acknowledgment

I would like to thank my mother for her continuous support throughout the process of writing this book.

I would like to thank Prakash Tripathi (@Prakash_MANIT) and Santosh Yadav (@SantoshYadavDev) for the valuable time on reviewing the book. Their constructive feedback was crucial in enhancing the quality of the content.

Finally, I would like to thank Mahesh Chand and the C# Corner team for their support in reviewing and publishing the book.

Prerequisites for the reader

The readers are expected to have a basic understanding of web development in general. They are also required to have a basic knowledge of HTML, CSS, and JavaScript. This book will not teach you about the basics of Angular. A fundamental understanding of the Angular framework is required before you proceed. Please learn and understand the following basic concepts of Angular.

- What is Angular?
- How Angular works in general?
- Angular component
- Angular Module
- Angular Services

If you have a basic idea of these concepts, then let's dive deep and start with our application.

Contents

Preface	1
About The Author	2
Acknowledgment	2
Prerequisites for the reader	3
What is a Single Page Application (SPA)?	7
What is Typescript?	7
What is Angular?	7
Features of Angular	8
Angular lifecycle hook	8
What is Firebase?	9
Why should you use Firebase?	10
What is Angular Material?	11
Setting up the Angular development environment	11
Source Code	14
Prerequisites for the application	14
Create a new Angular app	14
Open Visual Studio Code	14
Configuring firebase	15
Creating a project on firebase	15
Add Firebase configuration to your application	16
Create the Cloud Firestore database	19
Install @angular/fire and firebase	20
Install Angular Material packages	20
Add a material theme	21
Add the module for Angular material	22
What is Bootstrap?	23
Add Bootstrap CSS package	24
Serve the application	24
Add the navigation bar	25
Create the Home Page	26
Add Router module	26
Update the AppComponent	27
Add Forms module	28

Creating the data model	28
Create the blog service	29
Install CKEditor package	30
Add the blog editor	30
Add a route for the addpost page	31
Add CKEditor to BlogEditorComponent	31
Update the BlogEditorComponent template	32
Add a new blog	34
Add buttons in Navbar	35
Add styles to BlogEditorComponent	35
Checkpoint 1	36
Create custom pipes	38
Get the blogs from database	39
Add a BlogCardComponent	39
Add the BlogCardComponent to the home page	43
Checkpoint 2	43
Add Font Awesome library	44
Read a blog post	44
Checkpoint 3	47
Create the Snackbar service	48
Delete a blog post	49
Checkpoint 3	50
Edit an existing blog post	51
Checkpoint 4	54
Pagination on the home page	55
Create the PaginatorComponent	56
Add the PaginatorComponent to the BlogCard	58
Update the BlogCardComponent template	59
Checkpoint 5	60
Add the Google authentication	61
Create the AppUser model	61
Create the Authentication service	62
Update the AppComponent	64
Add Login button in the Navigation bar	65

Update the App module	67
Authenticated access for Edit and Delete	67
Capture the Author name	68
Secure the routes	70
Add route guards in App module	71
Checkpoint 6	71
Update Firestore database security rules	73
Implementing Authorization	74
Configure the Firestore database for admin role	74
Create the admin-auth guard	75
Update the BlogCardComponent	77
Add route guards in App module	77
Checkpoint 7	78
Adding the author profile	79
Checkpoint 8	82
Add the scroller to the blog page	82
Post comment on the blog	85
Create the comment model	85
Create the comment service	85
Create the Comment Component	87
Update the BlogCardComponent	92
Creating an index on the Firestore database	93
Checkpoint 9	94
Adding the share option for the blog	96
Install ngx-sharebuttons	96
Create the social-share component	98
Configure the icon pack	99
Checkpoint 10	100
Deploy the app on Firebase	102
References and Useful Links	105
Personal blog	105
Connect with me	105

What is a Single Page Application (SPA)?

A Single Page Application is a Web app having only one HTML page. The initial HTML page is downloaded from the server when we launch the application. Thereafter everything else is handled in the browser. After the initial page load, there is no HTML sent over the network by the server. The browser requests only data from the server and the server responds with the requested data. The data is re-written on the current page without reloading the page. In an SPA, the web page only refreshes (not reloaded) whenever a new set of data is requested from the server. This results in smooth user experience.

What is Typescript?

According to the official documents “TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.” It is an object-oriented, strongly-typed, compiled language. It was created by Anders Hejlsberg at Microsoft. He is also the creator of the C# language. Typescript is open source and is supported by all the browsers and operating systems.

What is Angular?

Angular is a JavaScript framework that allows us to create a client-side web application, using Typescript as the language. Angular is a Single Page Application (SPA) framework. It allows us to build apps for multiple platforms such as web, mobile web, native mobile, and native desktop. Angular is an open-source framework that is developed and maintained by Google.

Features of Angular

Some of the features of Angular are mentioned below:

- Single Page Application framework
- Highly scalable and performant
- Component-based architecture
- Inbuilt support for forms and form validation
- Cross platforms support for the development
- Open source and maintained by Google

A fundamental understanding of the Angular framework is required to get the most out of this book. If you have no prior knowledge of Angular, I recommend you familiarize yourself with the Angular framework before proceeding further.

Angular lifecycle hook

The Angular framework has a component-based architecture. Every component will go through a set of events from creation to destruction. These key events in the life of a component are defined by lifecycle hooks. The Angular core library provides a set of lifecycle hook interfaces that allows us to tap into those key moments in the lifecycle of the components. Each interface has a single hook method whose name is the interface name prefixed with ng. Angular provided Eight lifecycle hook methods as shown in the table below. This table is referenced from <https://angular.io/guide/lifecycle-hooks>.

Hook	Purpose and Timing
<code>ngOnChanges()</code>	Respond when Angular (re)sets data-bound input properties. The method receives a <code>SimpleChanges</code> object of current and previous property values. Called before <code>ngOnInit()</code> and whenever one or more data-bound input properties change.
<code>ngOnInit()</code>	Initialize the directive/component after Angular first displays the data-bound properties and sets the component's input properties. Called once, after the first <code>ngOnChanges()</code> .
<code>ngDoCheck()</code>	Detect and act upon changes that Angular can't or won't detect on its own. Called during every change detection run, immediately after <code>ngOnChanges()</code> and <code>ngOnInit()</code> .

<code>ngAfterContentInit()</code>	Respond after Angular projects external content into the component's view / the view that a directive is in. Called once after the first <code>ngDoCheck()</code> .
<code>ngAfterContentChecked()</code>	Respond after Angular checks the content projected into the directive/component. Called after the <code>ngAfterContentInit()</code> and every subsequent <code>ngDoCheck()</code> .
<code>ngAfterViewInit()</code>	Respond after Angular initializes the component's views and child views / the view that a directive is in. Called once after the first <code>ngAfterContentChecked()</code> .
<code>ngAfterViewChecked()</code>	Respond after Angular checks the component's views and child views / the view that a directive is in. Called after the <code>ngAfterViewInit()</code> and every subsequent <code>ngAfterContentChecked()</code> .
<code>ngOnDestroy()</code>	Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks. Called just before Angular destroys the directive/component.

The directive has the same set of lifecycle hooks as that of a component. A component or a directive will not implement all of the lifecycle hooks. A hook method is only invoked if it is defined.

What is Firebase?

Firebase is a mobile and web application development platform. It was developed by Firebase, Inc. in 2011, then acquired by Google in 2014. Firebase is considered as **Backend-as-a-Service – BaaS**.

Why should you use Firebase?

Firebase provides us with the following advantages:

1. **Build apps fast, without managing infrastructure:** Firebase allows us to build our apps faster by providing us features like analytics, databases, messaging and crash reporting.
2. **Supported by Google:** Firebase is built on Google infrastructure and scales automatically, for even the largest apps.

3. **One platform, with products that work better together:** Firebase products work great individually but share data and insights, so they work even better together.

Firebase supports a variety of platforms such as IOS, Android, Web, Unity, and C++.

Firebase is a comprehensive app development platform that provides 18 products. These products are divided into the following three categories:

- Build better apps
- Improve app quality
- Grow your business

A few of the products from each category are mentioned in the table shown below. You can refer to <https://firebase.google.com/products> to get the complete details on each product.

Build better apps	Improve app quality	Grow your business
Cloud Firestore	Crashlytics	Google Analytics
Cloud Storage	Performance Monitoring	Predictions
Authentication	Test Lab	Cloud Messaging
Hosting	App Distribution	Remote Config

Important Note:

Firebase supports the following two pricing plans:

- **Spark Plan:** This is a Free plan. This is suitable for small business and demo apps.
- **Blaze Plan:** This is a Pay as you go plan. This is suitable for large enterprises.

You can get the complete pricing information at <https://firebase.google.com/pricing>

What is Angular Material?

Angular Material is a UI component library for Angular, based on Google's material design specifications. It provides us with modern UI components that work across multiple platforms such as web, mobile, and desktop. It is optimized for Angular and can be integrated with an Angular application easily. The Angular Material is also performant and supports all modern browsers. It also provides inbuilt as well as custom themes to enhance the look and feel of our application.

Setting up the Angular development environment

We need to install the following software to start with Angular development.

- **Node.js**

Node.js will act as our development server. It will allow our Angular application to run in the local machine.

Navigate to <https://nodejs.org/en/download/> and install the LTS version of Node.js as per your operating system. Installing Node.JS will also install the NPM package manager in your machine.

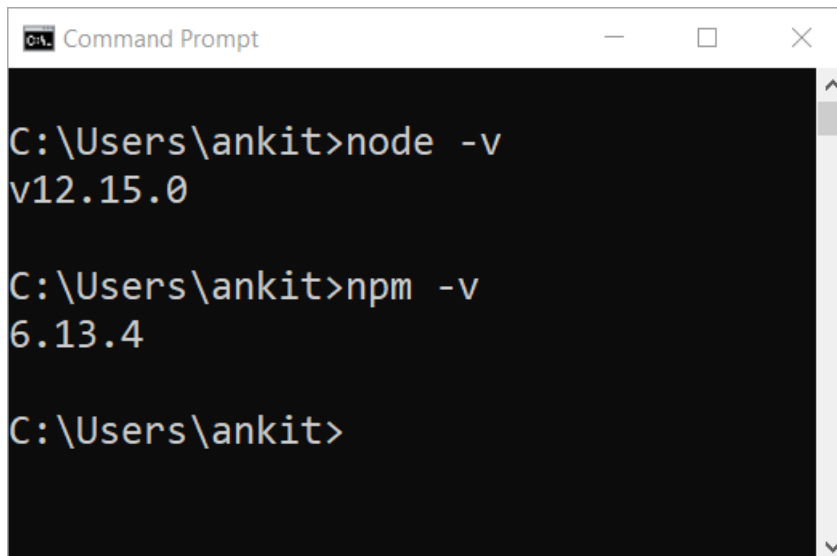
At the time of writing this book, the version of Node.JS is 12.15. Open a command window and run the command shown below to find the node version.

```
node -v
```

The version of NPM is 6.13.4. Run the command shown below to find the NPM version.

```
npm -v
```

Refer to the screenshot shown below.

A screenshot of a Windows Command Prompt window. The title bar says "Command Prompt". The prompt is "C:\Users\ankit>". The user has entered "node -v" and the output is "v12.15.0". The user has entered "npm -v" and the output is "6.13.4". The prompt is now "C:\Users\ankit>".

```
C:\Users\ankit>node -v
v12.15.0

C:\Users\ankit>npm -v
6.13.4

C:\Users\ankit>
```

- **Angular CLI**

Angular CLI is a command-line interface tool that allows us to develop, scaffold and initialize an Angular application. Angular CLI provides us the tools and commands out of the box to facilitate Angular app development.

Open a command window and run the following command to install the Angular CLI.

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

At the time of writing this book, the version of Angular CLI is 9.0.1. Open a command window and run the command shown below to find the Angular CLI version.

```
ng version
```

Refer to the screenshot shown below.

```
Command Prompt
C:\Users\ankit>ng version

Angular CLI
Angular CLI: 9.0.1
Node: 12.15.0
OS: win32 x64

Angular:
...
Ivy Workspace:

Package                                  Version
-----
@angular-devkit/architect               0.900.1
@angular-devkit/core                    9.0.1
@angular-devkit/schematics              9.0.1
@schematics/angular                    9.0.1
@schematics/update                      0.900.1
rxjs                                    6.5.3

C:\Users\ankit>
```

- **Visual Studio Code**

Visual Studio Code is a free and open-source IDE (Integrated Development Environment) developed by Microsoft. It is a lightweight code editor and supports development in languages such as C++, C#, Java, PHP, Python, Typescript, etc. It is available for Windows, macOS, and Linux. Visual Studio Code is one of the most widely-used IDEs for Angular development. Angular is supported by other IDEs also such as WebStorm, Sublime Text, Atom, etc. However, we will use the Visual Studio Code as our preferred IDE in this book.

Navigate to <https://code.visualstudio.com/> and install the latest version of Visual Studio code for your operating system.

Source Code

The source code for this application is available at <https://github.com/AnkitSharma-007/blogging-app-with-Angular-CloudFirestore>. You can clone the repository and refer to the code as we proceed with the app creation.

Prerequisites for the application

Set up the Angular development environment as described earlier. Apart from this, you will also need a Gmail account, which is required to login to Firebase. If you do not have a Gmail account, please create one before proceeding further.

Create a new Angular app

Navigate to the directory where you want to create the new project. Open the command prompt. Run the command shown below to create a new Angular app named as blogsite.

```
ng new blogsite --routing=false --style=scss
```

Open Visual Studio Code

Change the directory to the root project folder i.e. blog site, and open the project in VS Code by running the following set of commands.

```
cd blogsite  
code .
```

Configuring Firebase

We will create a project on firebase and configure the Google cloud Firestore database for it. We will use this database for our Angular application. The steps are shown below.

Creating a project on Firebase

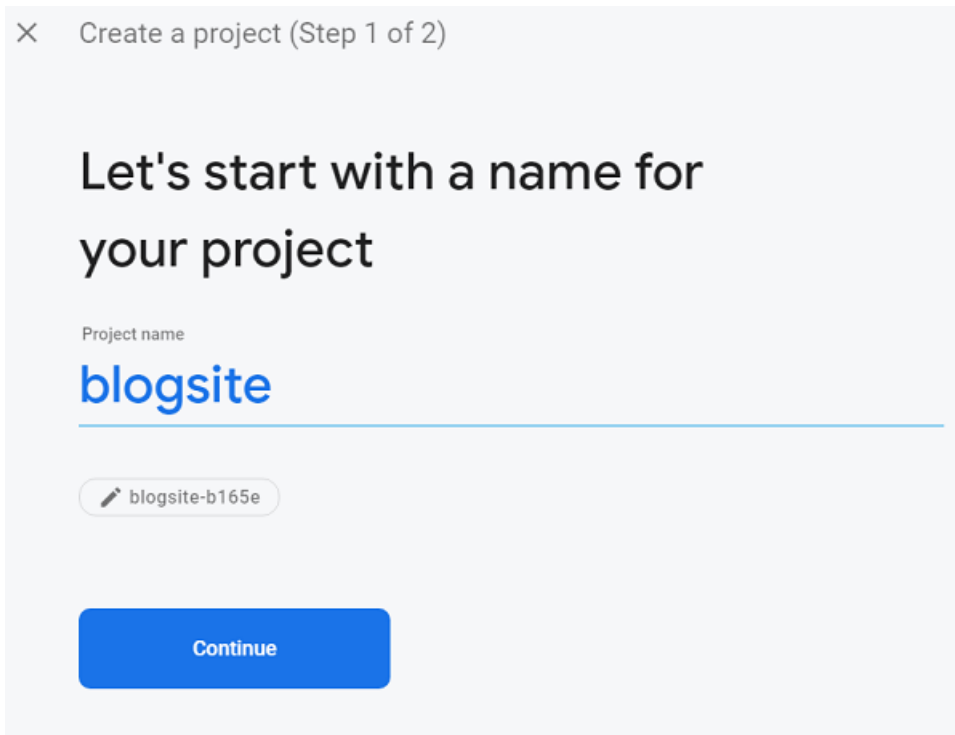
To create a new project on Firebase, follow the steps mentioned below.

Step 1: Navigate to <https://console.firebase.google.com/> and log in using your Gmail account.

Step 2: Click on the “Create a Project” button.

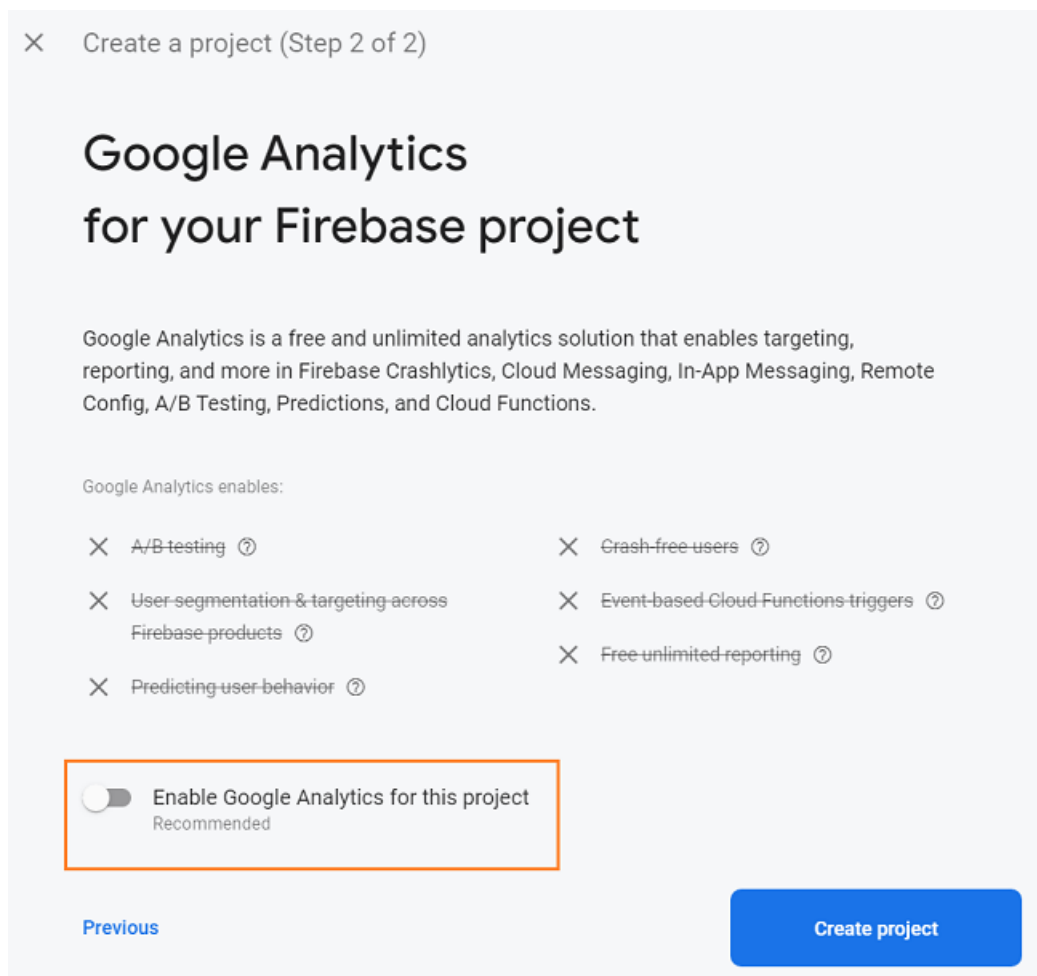
Step 3: Enter your project name. You can give any name of your choice. Here we will use the name blogsite. Click on Continue.

Refer to the image shown below for reference.



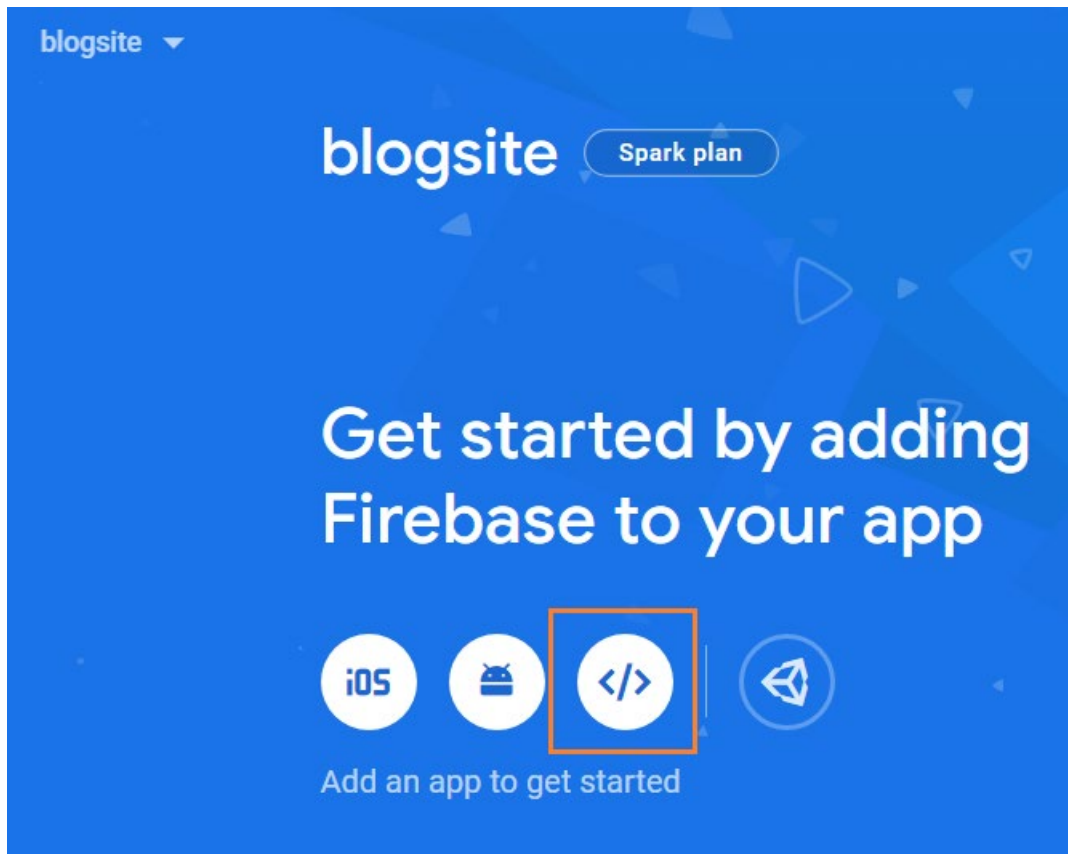
Step 4: On the next screen, disable the “Enable Google Analytics for this project” button. Click on “Create project”.

Refer to the image shown below for reference.



Add Firebase configuration to your application

We will create a web app for the Firebase project. We will add the configuration data of the Firebase web app to the Angular application. This will allow the Angular app to connect to the Firebase web app. On the project overview page, click on the “Web” icon as shown in the image below.



On the next page, provide a nickname for your app. You can use any name of your choice. Here we will use the name blogsite, the same name as our project. Click on the “Register app” button. Refer to the image shown below.

× Add Firebase to your web app

- ✓ Register app
- 2 Add Firebase SDK

Copy and paste these scripts into the bottom of your <body> tag, but before you use any Firebase services:

```
<!-- The core Firebase JS SDK is always required and must be listed first -->
<script src="https://www.gstatic.com/firebasejs/7.5.2/firebase-app.js"></script>

<!-- TODO: Add SDKs for Firebase products that you want to use
https://firebase.google.com/docs/web/setup#available-libraries -->

<script>
  // Your web app's Firebase configuration
  var firebaseConfig = {
    apiKey: "AIzaSyCxqWK4SVAAJkozEkURuteREIW9197z6-s",
    authDomain: "blogsite-b165e.firebaseio.com",
    databaseURL: "https://blogsite-b165e.firebaseio.com",
    projectId: "blogsite-b165e",
    storageBucket: "blogsite-b165e.appspot.com",
    messagingSenderId: "1057108181105",
    appId: "1:1057108181105:web:ac5bbb18e5f34c7e575bd0"
  };
  // Initialize Firebase
  firebase.initializeApp(firebaseConfig);
</script>
```

Copy the `firebaseConfig` object from the `<script>` tag. Paste the copied code into `src/environments/environment.ts` as shown in the code snippet below.

```
firebaseConfig: {
  apiKey: "AIzaSyCxqWK4SVAAJkozEkURuteREIW9197z6-s",
  authDomain: "blogsite-b165e.firebaseio.com",
  databaseURL: "https://blogsite-b165e.firebaseio.com",
  projectId: "blogsite-b165e",
  storageBucket: "blogsite-b165e.appspot.com",
  messagingSenderId: "1057108181105",
  appId: "1:1057108181105:web:ac5bbb18e5f34c7e575bd0"
}
```

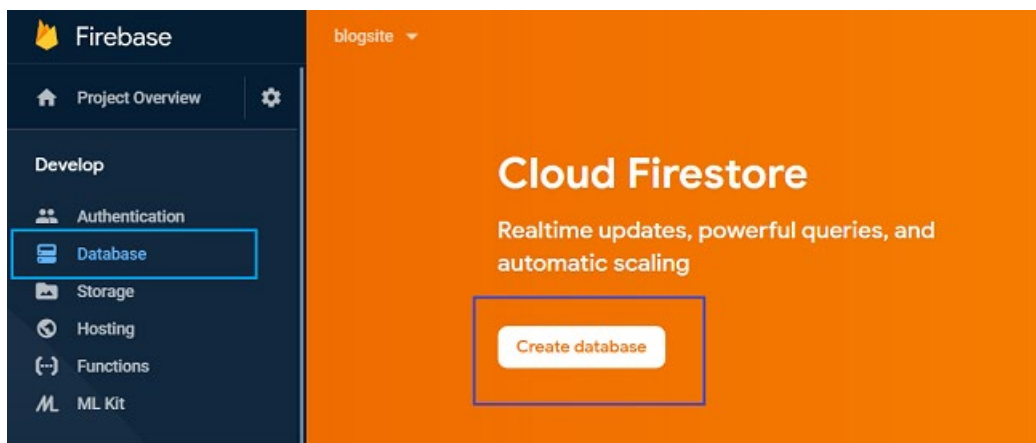
Similarly, paste the code into `src/environments/environment.prod.ts`. Import the environment constant, into `src/app/app.module.ts` as shown in the code snippet below.

```
...  
import { AppComponent } from './app.component';  
import { environment } from 'src/environments/environment';  
...
```

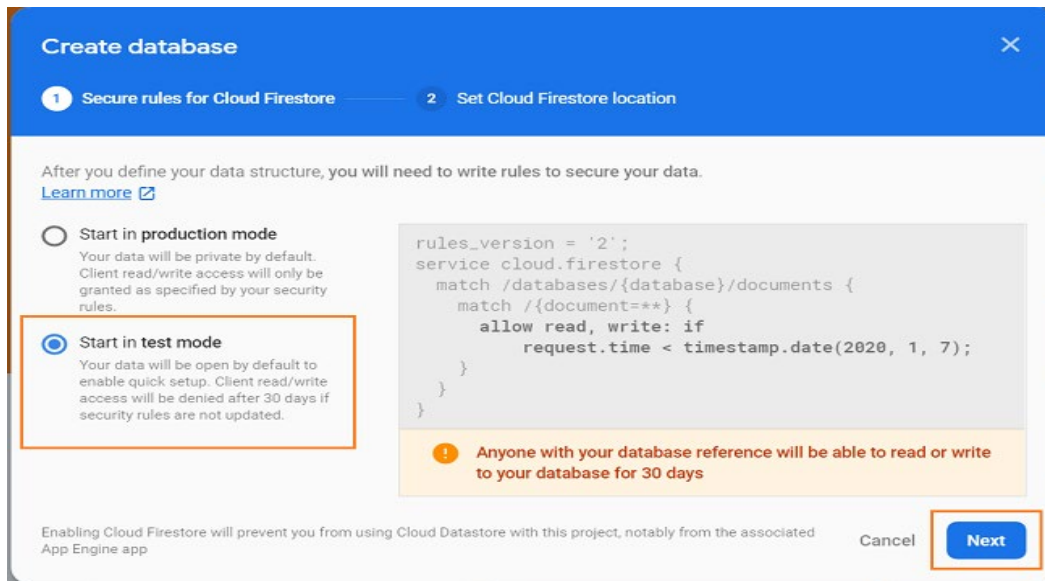
Click "Continue to the console" on the Firebase web page.

Create the Cloud Firestore database

Navigate to the "Project Overview" page of your Firebase project. Select "Database" under the "Develop" section from the menu on the left. Click on the "Create database" button.



In the "Create database" popup, select "Start in test mode". Click "Next". On the next screen, keep the default value for "Cloud Firestore location" and click on Done. The Cloud Firestore database is now configured for your Firebase project. Refer to the image shown below.



Setting the database in test mode is recommended for a sample application. However, this is not correct from a security point of view. As you can see a warning message is being displayed: “Anyone with your database reference will be able to read or write to your database for 30 days”. We will update this rule to allow only the authenticated users to write to our database in the latter part of this book.

Install @angular/fire and firebase

Open a new command window in the root project folder. We will execute all our Angular CLI commands in this window. To install the Firebase packages for Angular, run the following command.

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

Import the libraries in [src/app/app.module.ts](#) as shown below.

```
import { AngularFireModule } from '@angular/fire';
import { AngularFireStoreModule } from '@angular/fire/firestore';

@NgModule({
  ...
```

```
imports: [  
  // other imports  
  AngularFireModule.initializeApp(environment.firebaseConfig),  
  AngularFireFirestoreModule,  
],  
...  
})
```

Install Angular Material packages

Execute the following command in the console. This command will install the Angular material, the Component Dev Kit (CDK) and the Angular animations libraries.

```
npm install --save @angular/material @angular/cdk @angular/animations
```

After the successful installation of the Angular material packages, we will import the libraries into the `src/app/app.module.ts` file as shown below.

```
import {BrowserAnimationsModule} from '@angular/platform-  
browser/animations';  
  
@NgModule({  
  ...  
  imports: [  
    ...  
    BrowserAnimationsModule,  
  ],  
})
```

Add a material theme

Angular Material provided four pre-built themes as mentioned below.

- `deeppurple-amber.css`
- `indigo-pink.css`
- `pink-bluegrey.css`
- `purple-green.css`

To learn more about material themes, refer to <https://material.angular.io/guide/theming>.

To include a theme in an Angular application, we need to add the reference in `styles.scss` file. We will add the `indigo-pink.css` material theme globally by including the following line in [src/styles.scss](#) file.

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

Add the module for Angular material

We will create a new module to include all the Angular material related components. Run the following command in the console to create a new module.

```
ng g m ng-material
```

Open [src/app/ng-material/ng-material.module.ts](#) and replace the existing code with the code shown below.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { MatButtonModule } from '@angular/material/button';
import { MatCardModule } from '@angular/material/card';
import { MatDividerModule } from '@angular/material/divider';
import { MatIconModule } from '@angular/material/icon';
import { MatInputModule } from '@angular/material/input';
import { MatMenuModule } from '@angular/material/menu';
import { MatProgressSpinnerModule } from '@angular/material/progress-
spinner';
import { MatSelectModule } from '@angular/material/select';
import { MatSnackBarModule } from '@angular/material/snack-bar';
import { MatToolbarModule } from '@angular/material/toolbar';
import { MatTooltipModule } from '@angular/material/tooltip';

@NgModule({
  declarations: [],
  imports: [
```

```
CommonModule,  
MatToolbarModule,  
MatButtonModule,  
MatCardModule,  
MatInputModule,  
MatIconModule,  
MatDividerModule,  
MatMenuModule,  
MatSelectModule,  
MatSnackBarModule,  
MatProgressSpinnerModule,  
MatTooltipModule,  
],  
exports: [  
  CommonModule,  
  MatToolbarModule,  
  MatButtonModule,  
  MatCardModule,  
  MatInputModule,  
  MatIconModule,  
  MatDividerModule,  
  MatMenuModule,  
  MatSelectModule,  
  MatSnackBarModule,  
  MatProgressSpinnerModule,  
  MatTooltipModule,  
]  
})  
export class NgMaterialModule { }
```

We are importing all the required modules for Angular material components which we are going to use in this application. A separate module for Angular material will make the application easy to maintain.

Import the NgMaterialModule in `src/app/app.module.ts` file as shown below.

```
import { NgMaterialModule } from './ng-material/ng-material.module';  
  
@NgModule({  
  ...  
  imports: [  
    ...  
    NgMaterialModule,  
  ],  
})
```


What is Bootstrap?

Bootstrap is a CSS framework for building responsive, mobile-first web apps. It is an open-source framework and is widely used by web developers across the globe. It is considered as the world's most popular CSS framework for web development. Bootstrap 4 is the latest version of Bootstrap. Bootstrap 4 supports all major browsers except IE9.

Add Bootstrap CSS package

Run the following command to install Bootstrap in your app.

```
npm install bootstrap --save
```

Include the reference of Bootstrap globally in `src/styles.scss` file as shown below.

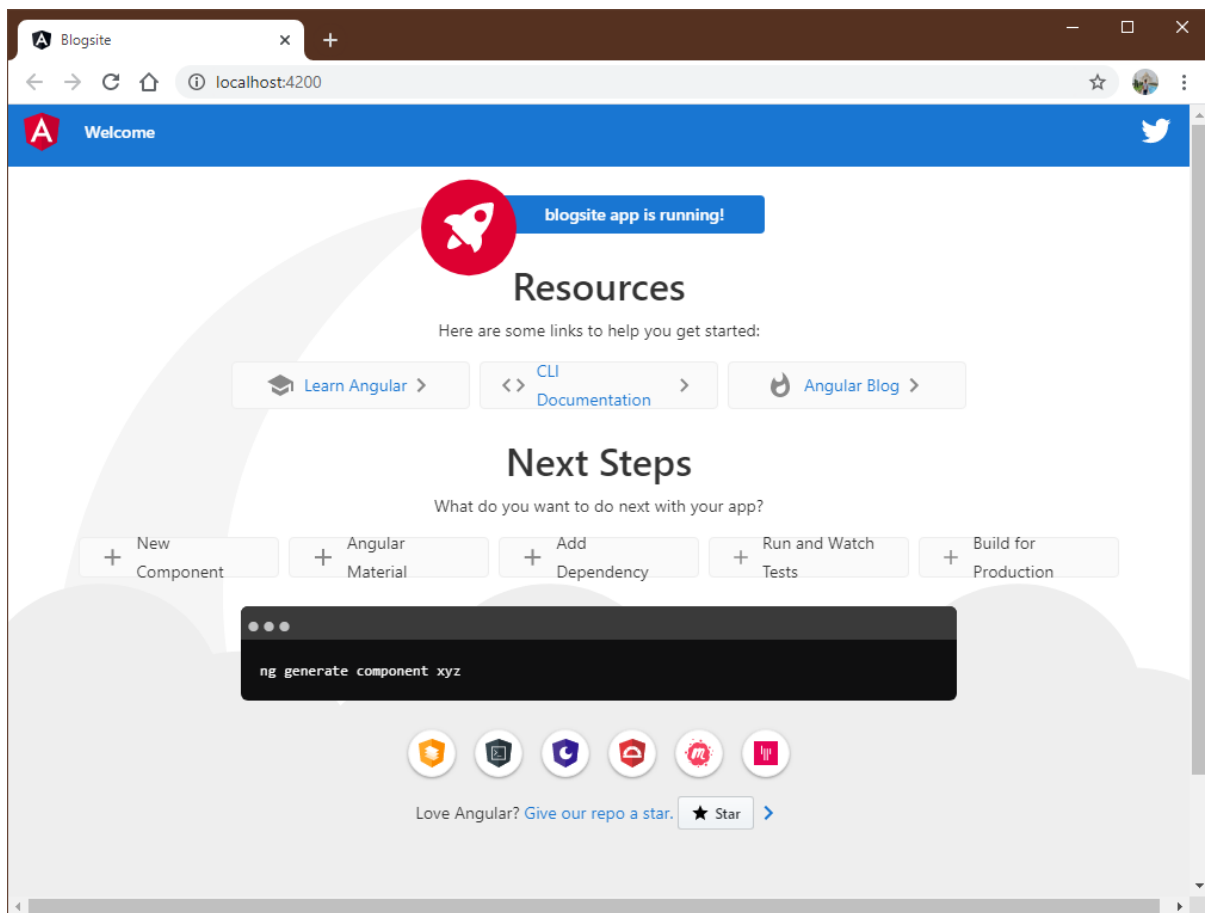
```
@import "~bootstrap/dist/css/bootstrap.css";
```

Serve the application

Open a new terminal window and run the following command.

```
ng serve -o
```

The Angular CLI will now serve the application at `localhost:4200`. The default browser will open this URL. You can see the window as shown in the image below.



The application will recompile and reload whenever a file changes. We will keep the server running and proceed with creating our components.

Add the navigation bar

Run the following command, in the original terminal, to generate a navigation bar component.

```
ng g c components/nav-bar
```

Open `src/app/components/nav-bar/nav-bar.component.html` and replace what is there with the following code.

```
<mat-toolbar class="nav-bar mat-elevation-z2"></mat-toolbar>
```

We will add the styling for nav bar in `src/app/components/nav-bar/nav-bar.component.scss` as shown below.

```
.nav-bar {  
  background-color: #1565C0;  
  color: #FFFFFF;  
  position: fixed;  
  top: 0;  
  z-index: 99;  
}  
  
button:focus {  
  outline: none;  
  border: 0;  
}
```

Create the Home Page

Run the following command to create the HomeComponent.

```
ng g c components/home
```

At this point, we will not add any code to HomeComponent. We will revisit it in the latter part of this book.

Add Router module

We will add the RouterModule into `src/app/app.module.ts` as shown below.

```
import { RouterModule } from '@angular/router';  
  
@NgModule({  
  ...  
  imports: [  
    ...
```

```
RouterModule.forRoot([
  { path: '', component: HomeComponent, pathMatch: 'full' },
  { path: '**', component: HomeComponent }
]),
],
}))
```

The empty path represents the default path for the application. If the path in the URL is empty, the app will render the HomeComponent. The `**` path is a wildcard. The router will select this route if the requested URL doesn't match any paths for routes defined earlier in the configuration.

The order in which we define the routes for the application is very important. The router uses a first-match wins strategy when matching routes. Therefore, more specific routes should be placed above less-specific routes. The wildcard route should be placed at the end because it matches every URL and should be selected only if no other routes are matched first.

Update the AppComponent

Open [src/app/app.component.html](#) and replace the content of the file with the following code.

```
<app-nav-bar></app-nav-bar>
<div class="container">
  <router-outlet></router-outlet>
</div>
```

The `<router-outlet>` is a placeholder that Angular dynamically fills with the component based on the current state of the router.

Add the following styles to `src/styles.scss`.

```
body {
  background-color: #fafafa;
}
```

```
.container {  
  padding-top: 60px;  
}
```

Add Forms Module

We will import the FormsModule in `src/app/app.module.ts` as shown below. This will allow us to use template-driven forms in our application.

```
import { FormsModule } from '@angular/forms';  
  
@NgModule({  
  ...  
  imports: [  
    ...  
    FormsModule,  
  ],  
})
```

Creating the data model

Create a new folder called `models` inside the `src/app` folder. Now we will create a file `post.ts` in the `models` folder. Open `src/app/models/post.ts` and put the following code.

```
export class Post {  
  postId: string;  
  title: string;  
  content: string;  
  author: string;  
  createdAt: any;  
  
  constructor() {  
    this.content = '';  
  }  
}
```

We have initialized the content property to an empty string in the constructor of the class. This is to ensure that the content is not shown as “undefined” while binding it to a form-field.

Create the blog service

We will create a service to handle database operations. Create a new service using the command shown below.

```
ng g s services/blog
```

Open the `src/app/services/blog.service.ts` file and add the following import definitions at the top.

```
import { Post } from '../models/post';
import { AngularFireStore } from '@angular/fire/firestore';
import { map } from 'rxjs/operators';
import { Observable } from 'rxjs';
```

We will inject the AngularFireStore in the constructor as shown below.

```
constructor(private db: AngularFireStore) { }
```

Now we will add the method to create a new post. Put the following method definition in the `blog.service.ts` file.

```
createPost(post: Post) {
  const postData = JSON.parse(JSON.stringify(post));
  return this.db.collection('blogs').add(postData);
}
```

This method will accept a parameter of type `Post`. We will parse the parameter to a JSON object and add it to the ‘blogs’ collection in our database. If the collection already exists then

the JSON object will get added to it. However, if the collection does not exist in the database, the add method will create the collection and then add the new object to it.

Install CKEditor package

We will use the CKEditor for adding and editing our blog post. CKEditor is a modern JavaScript rich text editor with a modular architecture. Its clean UI and features provide the perfect WYSIWYG user experience for creating semantic content. It is compatible with all modern web browsers.

Execute the command shown below to install the CKEditor editor component for Angular.

```
npm install --save @ckeditor/ckeditor5-angular
```

Run the command shown below to install one of the official editors builds which is the classic editor.

```
npm install --save @ckeditor/ckeditor5-build-classic
```

Import the CKEditorModule in `src/app/app.module.ts` as shown below.

```
import { CKEditorModule } from '@ckeditor/ckeditor5-angular';

@NgModule( {
  imports: [
    ...
    CKEditorModule,
  ],
})
```

Add the blog editor

We will create a new component for adding and editing the blog. Execute the command shown below.

```
ng g c components/blog-editor
```

Add a route for the addpost page

Add the route for the BlogEditor component in the `app.module.ts` file as shown below.

```
RouterModule.forRoot([  
  ...  
  { path: 'addpost', component: BlogEditorComponent },  
  ...  
])
```

Add CKEditor to BlogEditorComponent

Open `blog-editor.component.ts` and add the import definitions as shown below.

```
import * as ClassicEditor from '@ckeditor/ckeditor5-build-classic';  
import { Post } from 'src/app/models/post';  
import { DatePipe } from '@angular/common';  
import { BlogService } from 'src/app/services/blog.service';  
import { Router, ActivatedRoute } from '@angular/router';
```

We will also initialize some properties for this component. Add the following lines of code in the `BlogEditorComponent` class.

```
public Editor = ClassicEditor;  
ckeConfig: any;  
postData = new Post();  
formTitle = 'Add';  
postId = '';
```


We will create a method to define the configuration for the blog editor. Add the `setEditorConfig` method definition in the `BlogEditorComponent` class as shown below.

```
setEditorConfig() {
    this.ckeConfig = {
        removePlugins: ['ImageUpload', 'MediaEmbed'],
        heading: {
            options: [
                { model: 'paragraph', title: 'Paragraph', class: 'ck-
heading_paragraph' },
                { model: 'heading1', view: 'h1', title: 'Heading 1', class: 'ck-
heading_heading1' },
                { model: 'heading2', view: 'h2', title: 'Heading 2', class: 'ck-
heading_heading2' },
                { model: 'heading3', view: 'h3', title: 'Heading 3', class: 'ck-
heading_heading3' },
                { model: 'heading4', view: 'h4', title: 'Heading 4', class: 'ck-
heading_heading4' },
                { model: 'heading5', view: 'h5', title: 'Heading 5', class: 'ck-
heading_heading5' },
                { model: 'heading6', view: 'h6', title: 'Heading 6', class: 'ck-
heading_heading6' },
                { model: 'Formatted', view: 'pre', title: 'Formatted' },
            ]
        }
    };
}
```

This method is used to set the configuration of the classic editor. We will remove the plugins ‘ImageUpload’ and ‘MediaEmbed’ from the editor as we won’t be using these functionalities in our application. We will set the formatting options for the editor to include headings, paragraphs, and formatted text.

We will invoke this method inside the `ngOnInit` method as shown below.

```
ngOnInit() {
    this.setEditorConfig();
}
```

Update the BlogEditorComponent template

Open `blog-editor.component.html` and put the following code into it.

```
<h1>{{formTitle}} Post</h1>
<hr />
<form #myForm="ngForm" (ngSubmit)="myForm.form.valid && saveBlogPost()"
accept-charset="UTF-8" novalidate>
  <input type="text" class="blogHeader" placeholder="Add title..."
class="form-control" name="postTitle"
  [(ngModel)]="postData.title" #postTitle="ngModel" required />
  <span class="text-danger" *ngIf="myForm.submitted &&
postData.errors?.required">
    Title is required
  </span>
  <br />
  <div class="form-group">
    <ckeditor name="myckeditor" [config]="ckeConfig"
[(ngModel)]="postData.content" #myckeditor="ngModel"
    debounce="300" [editor]="Editor"></ckeditor>
  </div>
  <div class="form-group">
    <button type="submit" mat-raised-button
color="primary">Save</button>
    <button type="button" mat-raised-button color="warn"
(click)="cancel()">CANCEL</button>
  </div>
</form>
```

We have defined a template-driven form which will invoke the `saveBlogPost` method on successful submission. The form has two fields, one will accept the title of the blog whereas the other field will accept the content of the blog. The title field is a required field. The content field contains the CKEditor editor component. The `editor` property is used to set the type of editor. Here we are using the classic editor version of CKEditor. We will bind the `config` property of the `ckeditor` component to the `ckeConfig` variable which we have configured in the previous section. We will bind the content of the `ckeditor` component to the `content` property of the `postData` object which is of type `Post`.

The ckeditor component will return the content in the HTML format, not in plain text format. We will save each blog post as HTML in the database. This is to ensure that the formatting of the content is not lost while saving and retrieving the data from the database.

Add a new blog

We will now implement the feature of adding a new blog to our application. Open `blog-editor.component.ts` and inject the following service definitions in the constructor.

```
constructor(private route: ActivatedRoute,  
  private datePipe: DatePipe,  
  private blogService: BlogService,  
  private router: Router) { }
```

Add the provider for the DatePipe under in the `@Component` decorator section as shown below.

```
@Component({  
  ...  
  providers: [DatePipe]  
})
```

We will add the definition for the `saveBlogPost` method as shown below.

```
saveBlogPost() {  
  this.postData.createdDate = this.datePipe.transform(Date.now(), 'MM-dd-yyyy HH:mm');  
  this.blogService.createPost(this.postData).then(  
    () => {  
      this.router.navigate(['/']);  
    }  
  );  
}
```

We will add the current date in the postData object as the creation date for the blog post. We will then call the createPost method from the BlogService to add a new blog post to our database. This method will be invoked on clicking the Save button.

We will add the following method definition for the cancel method which will be invoked on click of the Cancel button.

```
cancel() {  
    this.router.navigate(['/']);  
}
```

Add buttons in Navbar

We will add the navigation button to the blog editor and home page in the navbar. Add the following code inside the <mat-toolbar> element in src/app/components/navbar/navbar.component.html.

```
<button mat-button [routerLink]='[""]'> My blog </button>  
<button mat-button [routerLinkActive]='["link-active"]'  
[routerLink]='["/addpost"]'>  
    Add Post  
</button>  
<span class="spacer"></span>
```

Add styles to BlogEditorComponent

We will add styling for blog editor in styles.scss file as shown below.

```
.ck-editor__editable {  
    max-height: 350px;  
    min-height: 350px;  
}  
  
pre {
```

```
display: block;
padding: 9.5px;
margin: 0 0 10px;
font-size: 13px;
line-height: 1.42857143;
color: #333;
word-break: break-all;
word-wrap: break-word;
background-color: #f5f5f5;
border: 1px solid #ccc;
border-radius: 4px;
}

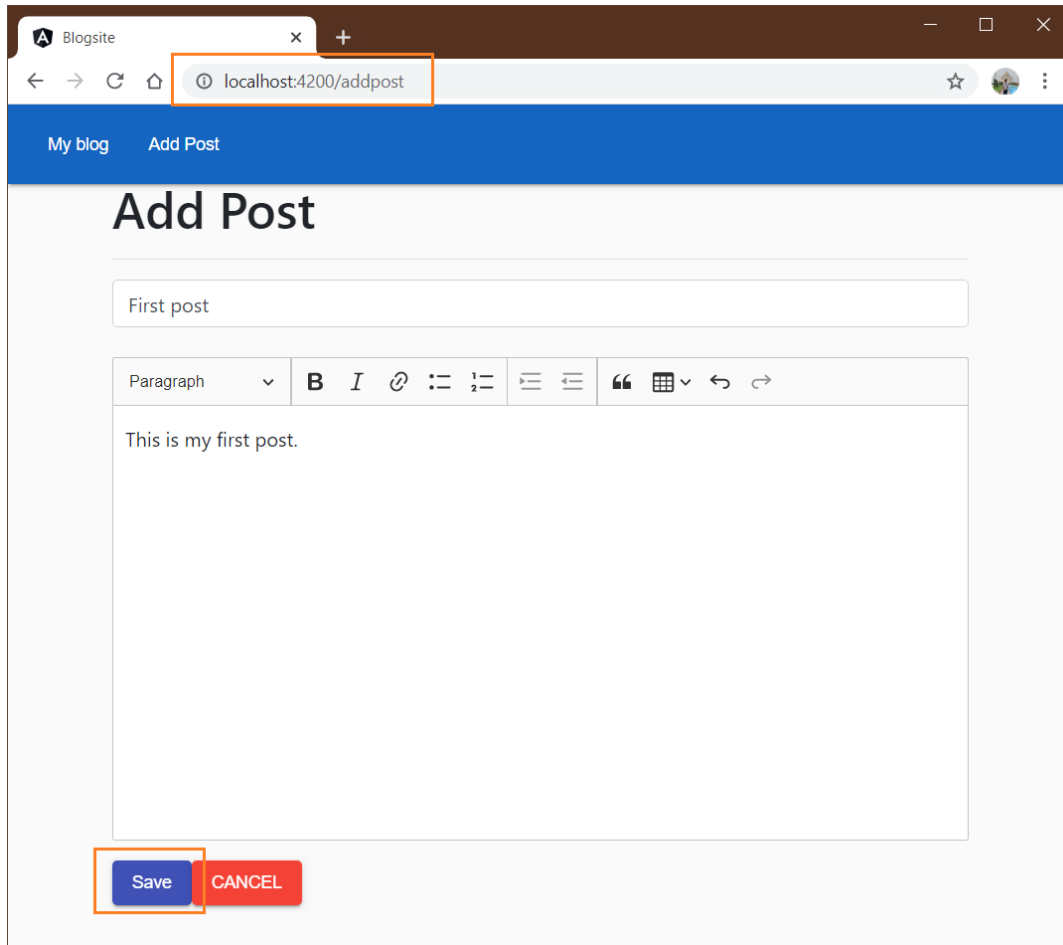
blockquote {
display: block;
padding: 10px 20px;
margin: 0 0 20px;
font-size: 17.5px;
border-left: 5px solid #eee;
}

.spacer {
flex: 1 1 auto;
}

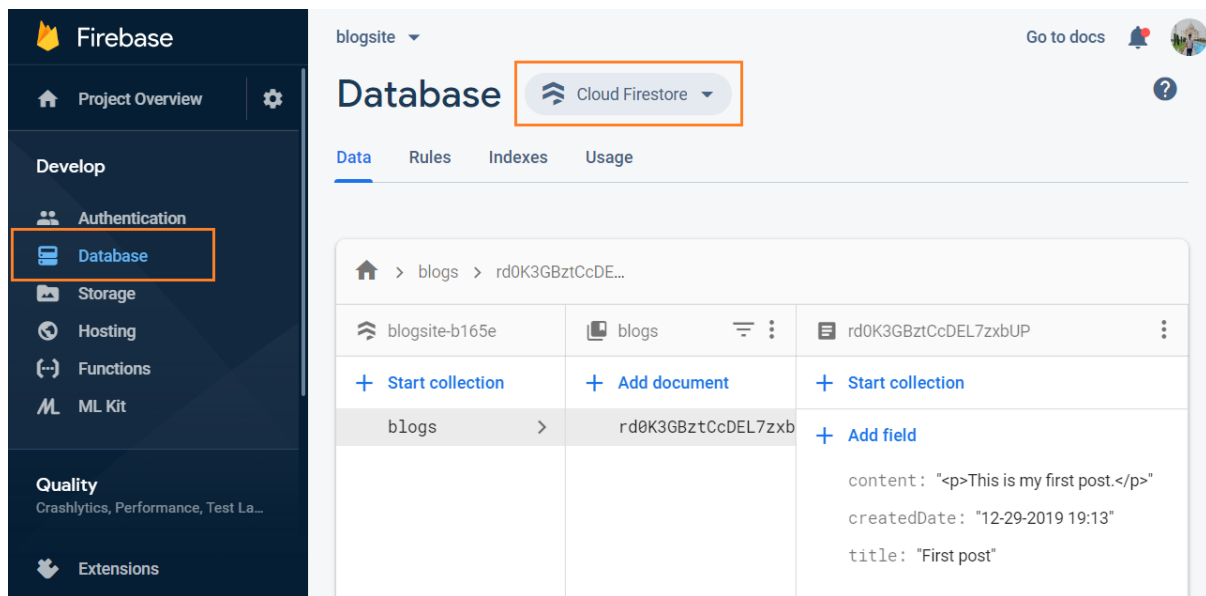
img{
max-width: 100%;
}
```

Checkpoint 1

Since the server is running, the web page will refresh to reflect the new changes. Open the browser and click on the “AddPost” button on the nav-bar. You will be navigated to the addpost page. Add a new blog and click on the save button to save the blog in the database. Refer to the image shown below.



Once we click on the Save button, the blog post will be added to our cloud Firestore database and we will be navigated to the home page of the application. To verify if the data has been added successfully or not, open the Firebase console. Navigate to your project overview page and click on the “Database” link in the menu on the left. Here you can see the record for your newly added blog. The “content” field contains the content of the blog in HTML format.



Create custom pipes

We will add two custom pipes in our app

- **Excerpt:** This will show a summary of the post on the blog card.
- **Slug:** This will show the URL slug for a post.

Run the following command to generate the excerpt pipe.

```
ng g p customPipes/excerpt
```

Open the <src/app/customPipes/excerpt.ts> file and replace the transform method with the code shown below.

```
transform(content: string) {  
  const postSummary = content.replace(/(<([>]+)>)/ig, '');  
  if (postSummary.length > 300) {  
    return postSummary.substr(0, 300) + ' [...]';  
  } else {  
    return postSummary;  
  }  
}
```

This pipe will accept the contents of the blog as a string and return the first 300 characters as the summary of the blog. Since the content of the blog is in HTML format, we will remove all the HTML tags before extracting the summary.

Run the following command to generate the slug pipe.

```
ng g p customPipes/slug
```

Open the <src/app/customPipes/slug.ts> file and replace the transform method with the code shown below.

```
transform(title: string) {  
  const urlSlug = title.trim().toLowerCase().replace(/ /g, '-');  
  return urlSlug;  
}
```

This pipe will accept the title of the blog and return the title as a URL slug. We will replace the white space character between the words in the title with a '-' character to create the URL slug.

Get the blogs from database

We will add the `getAllPosts` method in `src/app/services/blog.service.ts` file.

This method is used to fetch all the blog posts from the database. The definition of this method is shown below.

```
getAllPosts(): Observable<Post[]> {  
  const blogs = this.db.collection<Post>('blogs', ref =>  
    ref.orderBy('createdAt', 'desc'))  
    .snapshotChanges().pipe(  
      map(actions => {  
        return actions.map(  
          c => ({  
            postId: c.payload.doc.id,  
            ...c.payload.doc.data()  
          }));  
      });  
}
```



```
    }));  
    return blogs;  
}
```

We will fetch the blog in the descending order of the created date to ensure that the latest blog is always on the top.

Add a BlogCardComponent

Run the command shown below to create the blog card component.

```
ng g c components/blog-card
```

Open `src/app/components/blog-card.component.ts` and add the following import definitions.

```
import { OnDestroy } from '@angular/core';  
import { BlogService } from 'src/app/services/blog.service';  
import { Post } from 'src/app/models/post';  
import { Subject } from 'rxjs';  
import { takeUntil } from 'rxjs/operators';
```

Inject the Blog Service in the constructor of `BlogCardComponent` class as shown below.

```
constructor(private blogService: BlogService) { }
```

Add a property to hold the current blog post.

```
blogPost: Post[] = [];  
private unsubscribe$ = new Subject<void>();
```

Now we will create a method to get the blog post and invoke it inside `ngOnInit` in the `src/app/components/blog-card.component.ts` file.

```
ngOnInit() {  
  this.getBlogPosts();  
}  
  
getBlogPosts() {  
  this.blogService.getAllPosts()  
    .pipe(takeUntil(this.unsubscribe$))  
    .subscribe(result => {  
      this.blogPost = result;  
    });  
}
```

We will implement the `OnDestroy` interface on the `BlogCardComponent` class. Inside the `ngOnDestroy` method, we will complete the `unsubscribe$` subscription. Refer to the code snippet shown below.

```
export class BlogCardComponent implements OnInit, OnDestroy {  
  
  ...  
  
  ngOnDestroy() {  
    this.unsubscribe$.next();  
    this.unsubscribe$.complete();  
  }  
}
```

Open `src/app/components/blog-card.component.html` and replace the existing code with the code snippet shown below.

```
<ng-template #emptyblog>  
  <div class="spinner-container">  
    <mat-spinner></mat-spinner>  
  </div>  
</ng-template>  
<ng-container *ngIf="blogPost && blogPost.length>0; else emptyblog">  
  <div *ngFor="let post of blogPost">  
    <mat-card class="blog-card mat-elevation-z2">  
      <mat-card-content>
```

```
        <a class="blog-title" [routerLink]="['/blog/',
post.postId, post.title | slug]">
            <h2>{{ post.title}} </h2>
        </a>
    </mat-card-content>
    <mat-card-content>
        <div [innerHTML]="post.content | excerpt"></div>
    </mat-card-content>
    <mat-divider></mat-divider>
    <mat-card-actions align="end">
        <ng-container>
            <button mat-raised-button color="accent"
[routerLink]="['/editpost',post.postId]">Edit</button>
            <button mat-raised-button color="warn"
(click)="delete(post.postId)">Delete</button>
        </ng-container>
        <span class="spacer"></span>
        <button mat-raised-button color="primary"
[routerLink]="['/blog/', post.postId, post.title | slug]">Read
More</button>
    </mat-card-actions>
    </mat-card>
</div>
<mat-divider></mat-divider>
</ng-container>
```

We will display the blog title and summary in a mat-card. Mat-card is an Angular material component that can be used to display the content in a card layout. We have used the slug pipe to create the URL for a blog. We have used the excerpt pipe to get the summary of each blog. Each blog card will have buttons to Edit and Delete the blog post. At this point, any user can access the Edit and Delete buttons. In the future, we will add authorization to our application so that only the Admin user will have access to these functionalities.

At this point, you might get an error in this file. We have added a Delete button and defined a delete method on the click event of the button. However, we have not added any delete method yet. Therefore, we will get a compile-time error. To resolve this issue, add the following code in the `src/app/components/blog-card.component.ts` file.

```
delete(postId: string) {
    // Method definition to be added later
}
```

Here, we have defined an empty method. We will add delete logic to this method in the latter part of the book.

Open `src/app/components/blog-card/blog-card.component.scss` and replace the existing code with the code snippet shown below.

```
.blog-card {  
  margin-bottom: 15px;  
}  
  
.blog-title {  
  text-decoration: none;  
}  
  
a:hover {  
  color: indianred;  
}  
  
.spinner-container{  
  display: flex;  
  align-items: center;  
  justify-content: center;  
  height: 100%;  
}
```

Add the BlogCardComponent to the home page

Since we will display the blog cards on the home page, therefore, we need to add the blog card component to the home component.

Open `src/app/components/home.component.html` and replace the existing code with the following HTML code.

```
<div class="row left-panel">  
  <div class="col-md-9">  
    <app-blog-card></app-blog-card>  
  </div>
```

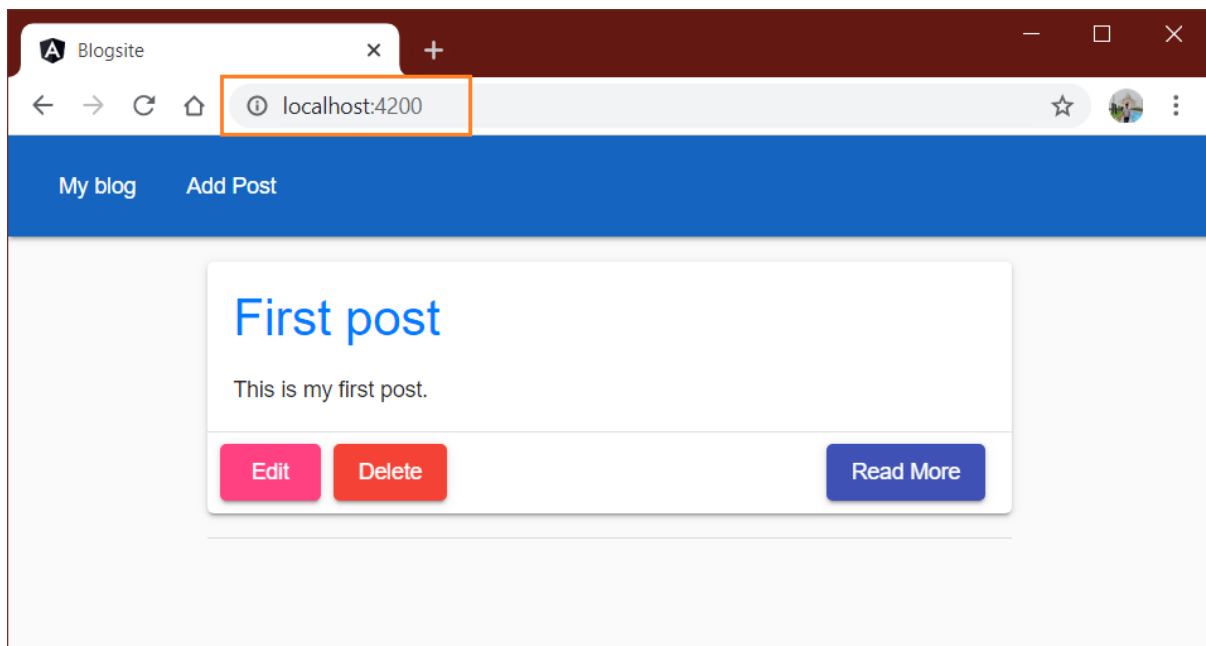
```
</div>
```

Open `src/app/components/home/home.component.scss` and add the following style definition inside it.

```
.left-panel {  
  margin-top: 15px;  
}
```

Checkpoint 2

At this point, we have successfully added the blog card to the home page. Open the browser and you can see the blog displayed in a card on the home page. This is the same blog which we have added in checkpoint 1. The card will also have Edit, Delete and Read More buttons. But at this point, they are non-functional. We will add the logic for these buttons in the latter part of this book. Refer to the image shown below.



Add Font Awesome library

We will add Font Awesome and Material Icons libraries in our application. We will use icon sets provided by this library for styling our app. Add the following lines in the `index.html` file.

```
<link href="https://fonts.googleapis.com/icon?family=Material+Icons"
rel="stylesheet">
<link rel="stylesheet" type="text/css"
href="https://maxcdn.bootstrapcdn.com/font-awesome/4.7.0/css/font-
awesome.min.css" />
```

Read a blog post

We will add the feature of reading a blog. Run the following command to create the blog component

```
ng g c components/blog
```

Add the router link for this component in `app.module.ts` as shown below.

```
{ path: 'blog/:id/:slug', component: BlogComponent },
```

We have defined two parameters in this route. The id is the unique id of each blog, whereas the slug is the URL slug which is created from the title of the blog.

Add the following method definition in the `blog.service.ts` file. This method will fetch the blog details based on the id from the 'blogs' collections.

```
getPostById(id: string): Observable<Post> {
  const blogDetails = this.db.doc<Post>('blogs/' + id).valueChanges();
  return blogDetails;
}
```

Open `src/app/components/blog/blog.component.ts` and add import definitions as shown below.

```
import { OnDestroy } from '@angular/core';
import { Post } from 'src/app/models/post';
import { ActivatedRoute } from '@angular/router';
import { BlogService } from 'src/app/services/blog.service';
import { Subject } from 'rxjs';
import { takeUntil } from 'rxjs/operators';
```

We will update the `BlogComponent` class as shown below.

```
export class BlogComponent implements OnInit, OnDestroy {

  postData: Post = new Post();
  postId;
  private unsubscribe$ = new Subject<void>();

  constructor(private route: ActivatedRoute,
               private blogService: BlogService) {
    if (this.route.snapshot.params['id']) {
      this.postId = this.route.snapshot.paramMap.get('id');
    }
  }

  ngOnInit() {
    this.blogService.getPostbyId(this.postId)
      .pipe(takeUntil(this.unsubscribe$))
      .subscribe(
        (result: Post) => {
          this.postData = result;
        }
      );
  }

  ngOnDestroy() {
    this.unsubscribe$.next();
    this.unsubscribe$.complete();
  }
}
```

In the constructor of the class, we will get the id from the URL. In the `ngOnInit`, we will call the `getPostbyID` method of the `BlogService` and pass the `postId` as the parameter to fetch the details of a particular blog. Inside the `ngOnDestroy` method, we will complete the `unsubscribe$` subscription.

Open `src/app/components/blog/blog.component.html` and replace the existing code with the code shown below.

```
<div class="docs-example-viewer-wrapper">
  <h1 class="entry-title">{{postData.title}}</h1>
  <mat-card-subtitle class="blog-info">
    <i class="fa fa-calendar" aria-hidden="true"></i>
    {{postData.createdDate | date:'longDate'}}
  </mat-card-subtitle>
  <mat-divider></mat-divider>
  <div class="docs-example-viewer-body">
    <div [innerHTML]="postData.content">
    </div>
  </div>
</div>
```

This component is used to display the full blog post. We will show the blog created date and also display a calendar icon using font-awesome. Since the content of the blog is stored and fetched as HTML, we will bind the `innerHTML` property of `<div>` element to the content of the blog. This will ensure that the blog is displayed as plain text on the screen.

Finally, we will add styling for `BlogComponent`. Open `src/app/components/blog/blog.component.scss` and put the following style definitions into it.

```
.docs-example-viewer-body {
  padding: 20px;
  align-content: center;
  align-items: center;
  font-size: 14px;
}

.docs-example-viewer-wrapper {
  border: 1px solid rgba(0, 0, 0, .03);
  box-shadow: 0 1px 1px rgba(0, 0, 0, .24), 0 0 2px rgba(0, 0, 0, .12);
  border-radius: 4px;
}
```



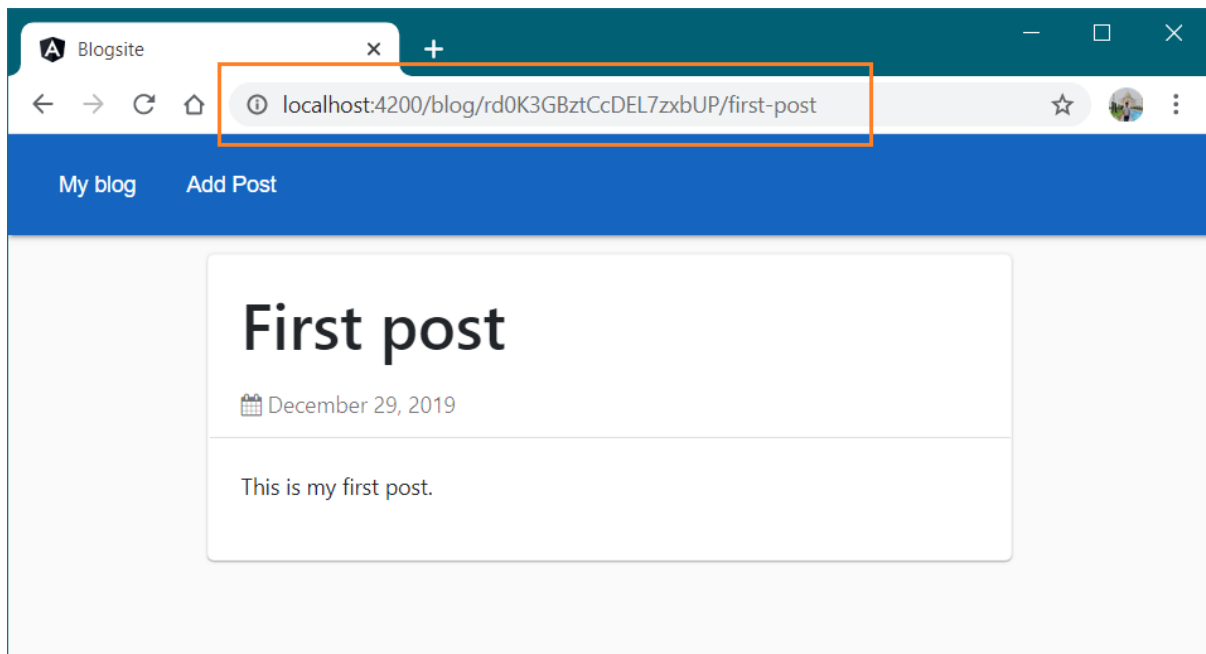
```
margin: 1em auto;
background-color: #FFFFFF;
}

.entry-title {
margin: 20px;
}

.blog-info {
margin: 15px 20px 10px;
align-content: center;
align-items: center;
.fa-user {
margin-left: 10px;
}
}
```

Checkpoint 3

Open the browser and click on the “Read More” button on the blog card. You will be navigated to a new page that will display the full blog. The blog creation date is displayed just below the title of the blog. Also, notice the URL for this page. The URL contains the id and the title of the blog as a slug. Refer to the screenshot below.



Create the Snackbar service

We will create a Snackbar service which will help us to show the user-friendly messages on the screen as snack-bar notifications. Create the service using the command shown below.

```
ng g s services/Snackbar
```

open `src/app/services/snackbar.service.ts` and put the following code inside it.

```
import { Injectable } from '@angular/core';
import { MatSnackBar } from '@angular/material/snack-bar';

@Injectable({
  providedIn: 'root'
})
export class SnackbarService {

  constructor(private snackBar: MatSnackBar) { }

  showSnackBar(message: string) {
    this.snackBar.open(message, 'Close', {
      duration: 2000,
      panelClass: 'snackbar-ribon',
      verticalPosition: 'top',
      horizontalPosition: 'center'
    });
  }
}
```

We have created the `showSnackBar` method which will accept the message to display as a parameter. The display duration is set to 2000 milliseconds. We have defined the position in such a way that the snack-bar will be displayed in the top-center of the page.

Add the following styling for the snack-bar in `styles.scss` file.

```
.snackbar-ribon {
  color: #FFFFFF;
  background: #17a2b8;
}
```

Delete a blog post

We will add the feature of deleting a blog. We will delete the blog from the 'blogs' collection based on the `postId`. Add the following lines of code in the `src/app/services/blog.service.ts` file.

```
deletePost(postId: string) {  
  return this.db.doc('blogs/' + postId).delete();  
}
```

Open `src/app/components/blog-card/blog-card.component.ts` and import the `SnackbarService` as shown below.

```
import { SnackbarService } from 'src/app/services/snackbar.service';
```

Inject the `SnackbarService` in the constructor as shown below.

```
constructor(  
  // other service injection  
  private snackBarService: SnackbarService  
) { }
```

We will also update the delete method in the `blog-card.component.ts` file. The method definition is shown below.

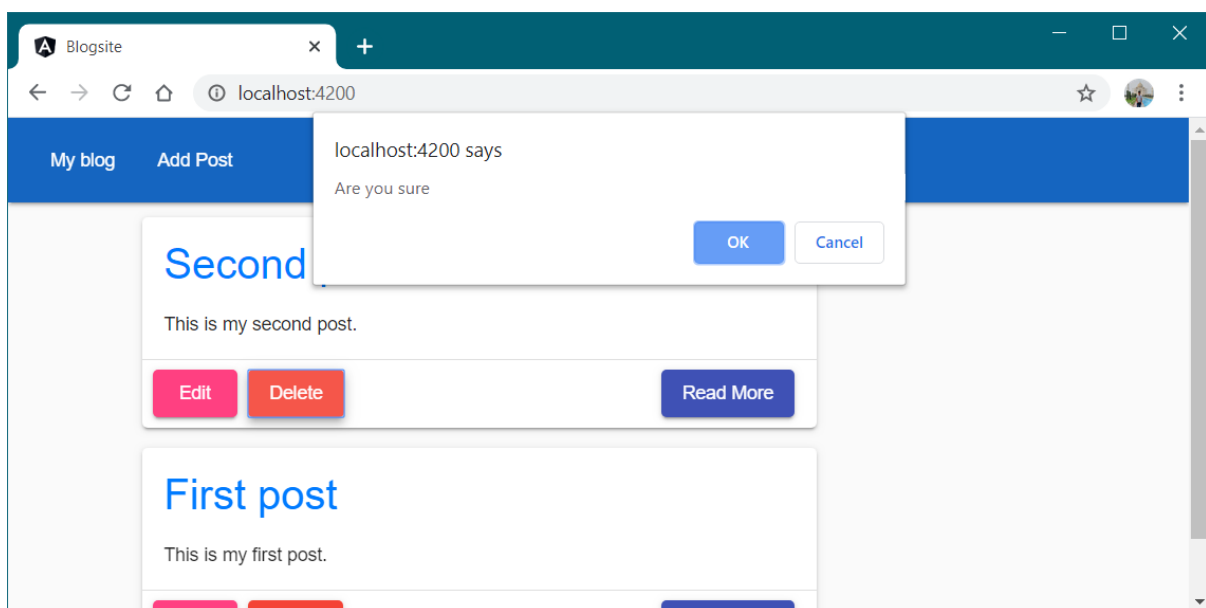
```
delete(postId: string) {  
  if (confirm('Are you sure')) {  
    this.blogService.deletePost(postId).then(  
      () => {  
        this.snackBarService.showSnackBar('Blog post deleted  
successfully');  
      }  
    );  
  }  
}
```

```
}  
}
```

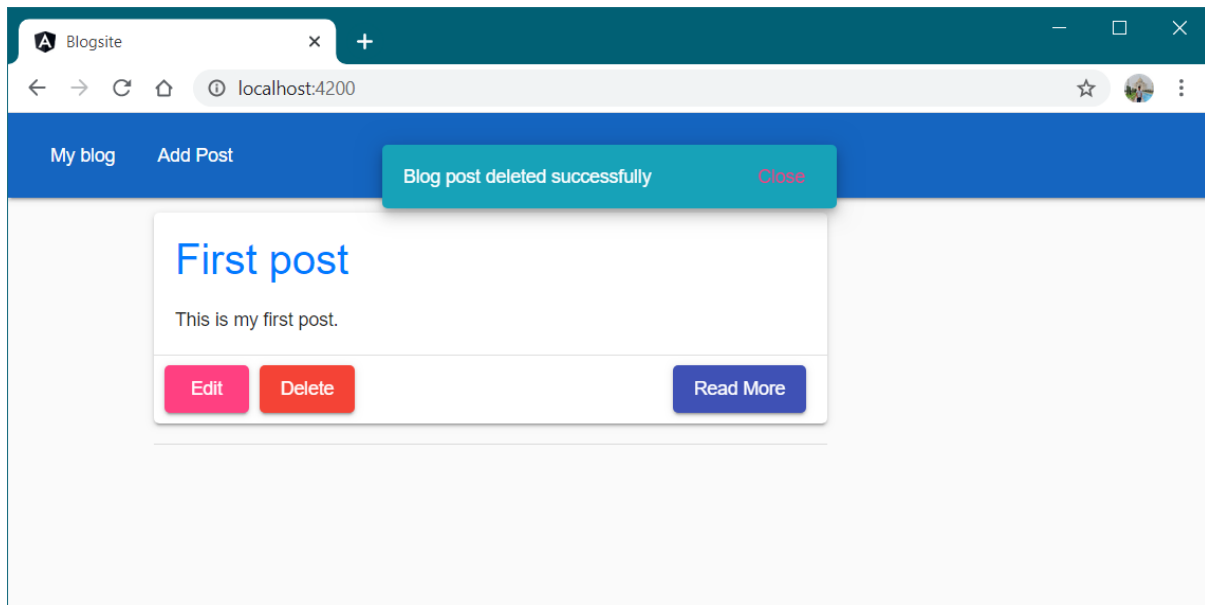
The delete method will accept the `postId` as a parameter. It will display an alert box asking for the confirmation to delete. If the user clicks on “OK”, then we will invoke the `deletePost` method of the `BlogService`. Upon the successful deletion of the blog, we will show the success message using a snack-bar.

Checkpoint 3

Open the browser and click on the “Delete” button on the blog card. A JavaScript confirmation box will open asking for the confirmation to delete the blog. Refer to the image shown below.



If you click on the OK button, the blog will be deleted and you will get a confirmation message in a snackbar. Refer to the image shown below.



Edit an existing blog post

We will now implement the functionality to edit an existing blog. Add the following code definition in `blog.service.ts`.

```
updatePost(postId: string, post: Post) {  
  const putData = JSON.parse(JSON.stringify(post));  
  return this.db.doc('blogs/' + postId).update(putData);  
}
```

The `updatePost` method will accept the `postId` and an object of type `Post` as the parameter. We will parse the `Post` object to a JSON object and then update the object in the 'blogs' collection.

Add the routing for edit functionality in `app.module.ts` as shown below.

```
RouterModule.forRoot([  
  ...  
  { path: 'editpost/:id', component: BlogEditorComponent },  
])
```

```
    ...  
  })
```

Add the following import definition in the `blog-editor.component.ts` file.

```
import { Subject } from 'rxjs';  
import { takeUntil } from 'rxjs/operators';
```

Declare a new Subject variable. Refer to the code snippet shown below.

```
private unsubscribe$ = new Subject<void>();
```

Open `blog-editor.component.ts` and add the following code in the constructor. Here, we are fetching the id of the blog from the URL with the help of the `ActivatedRoute` class.

```
if (this.route.snapshot.params['id']) {  
  this.postId = this.route.snapshot.paramMap.get('id');  
}
```

We will add the method to set the edit form when we click on the “Edit” button on the blog card on the home page. The method definition is shown below.

```
setPostFormData(postFormData) {  
  this.postData.title = postFormData.title;  
  this.postData.content = postFormData.content;  
}
```

Update the `ngOnInit` method inside the `BlogEditorComponent` class as shown below.

```
ngOnInit() {  
  this.setEditorConfig();  
  if (this.postId) {  
    this.formTitle = 'Edit';  
  }  
}
```

```
        this.blogService.getPostbyId(this.postId)
            .pipe(takeUntil(this.unsubscribe$))
            .subscribe(
                result => {
                    this.setPostFormData(result);
                }
            );
    }
}
```

If the postId is set then it means that this is an Edit request. We will set the title of the form to “Edit”. We will call the `getPostbyId` method from `BlogService` to fetch the details of the blog corresponding to the postId.

Upon clicking Save we need to handle the case of both adding a new blog as well as editing an existing blog. Hence we will update the `saveBlogPost` as shown below.

```
saveBlogPost() {
    if (this.postId) {
        this.blogService.updatePost(this.postId, this.postData).then(
            () => {
                this.router.navigate(['/']);
            }
        );
    } else {
        this.postData.createdDate = this.datePipe.transform(Date.now(), 'MM-dd-yyyy HH:mm');
        this.blogService.createPost(this.postData).then(
            () => {
                this.router.navigate(['/']);
            }
        );
    }
}
```

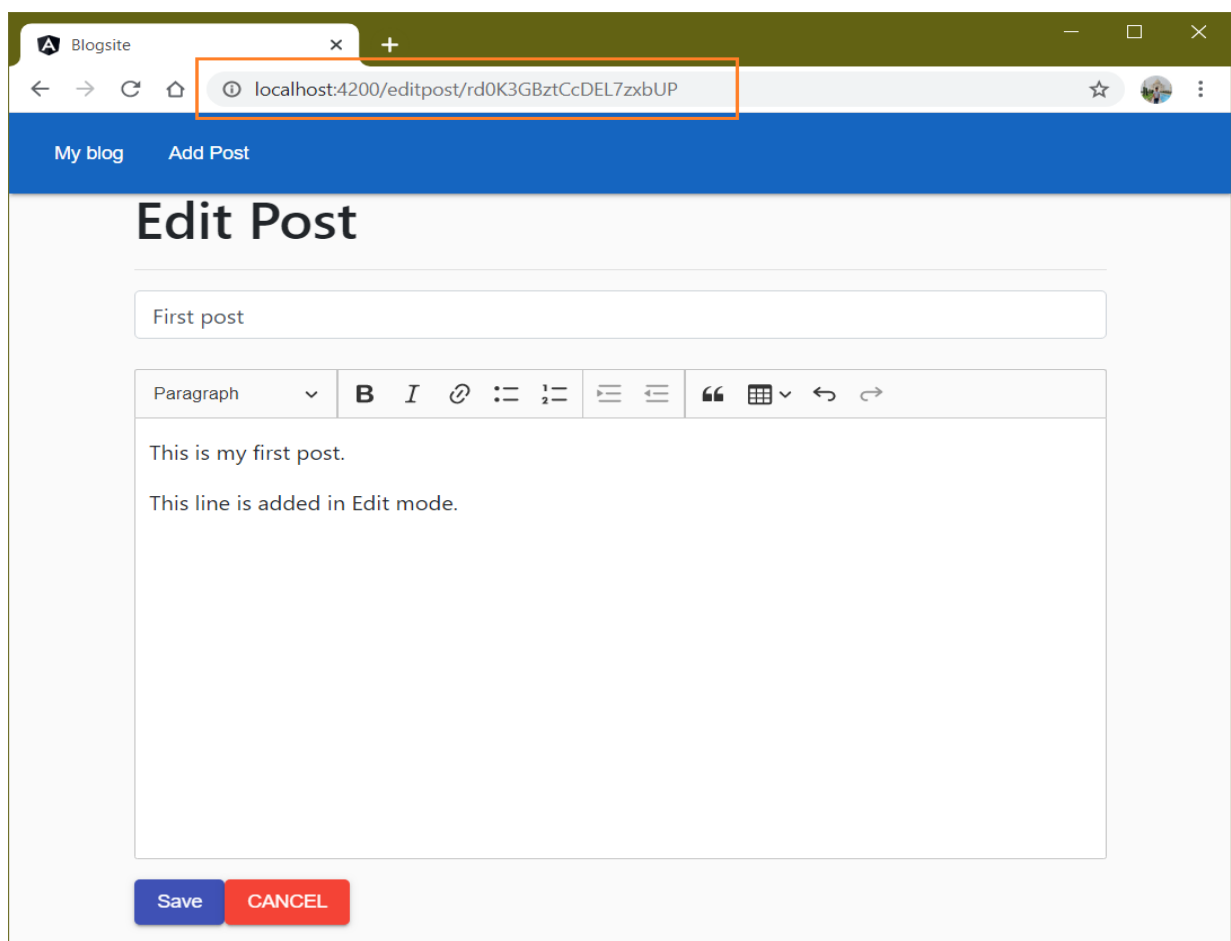
We will implement the `OnDestroy` interface on the `BlogEditorComponent` class. Inside the `ngOnDestroy` method, we will complete the `unsubscribe$` subscription. Refer to the code snippet shown below.

```
ngOnDestroy() {
    this.unsubscribe$.next();
}
```

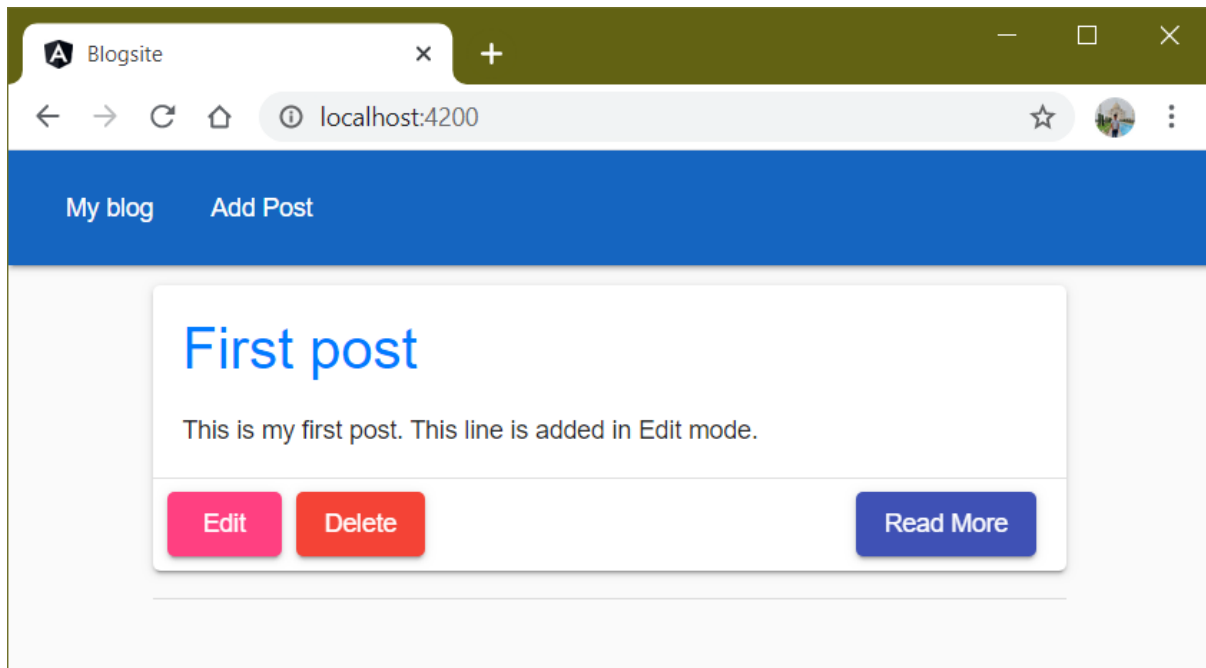
```
this.unsubscribe$.complete();  
}
```

Checkpoint 4

Open the browser and click on the “Edit” button in the blog card on the Home page. You will be navigated to the “Edit Post” page. You can see the blog editor with the content of the post already populated inside it. The URL of the page will contain the postId for the particular blog. Refer to the image shown below.



You can edit the content of the blog inside the blog editor. Click on save to update the blog content and return to the home page. You can see the updated content in the blog excerpt. Refer to the image shown below.



Pagination on the home page

We will add the feature of pagination on the home page. We will use the ngx-pagination for this purpose. It is an open-source component which provides a simple and easy to use pagination feature for Angular apps.

Execute the command shown below to install the ngx-pagination component for Angular.

```
npm i ngx-pagination --save
```

Import the NgxPaginationModule in src/app/app.module.ts as shown below.

```
import { NgxPaginationModule } from 'ngx-pagination';

@NgModule( {
  imports: [
    ...
    NgxPaginationModule,
  ],
})
```

Create the PaginatorComponent

Run the following command, in the original terminal, to generate a Paginator component.

```
ng g c components/paginator
```

Open `src/app/components/paginator/paginator.component.ts` and add the following import definitions at the top.

```
import { Input } from '@angular/core';  
import { Router } from '@angular/router';
```

We will add two Input properties for this component as shown below.

```
@Input()  
pageSizeOptions: [];  
  
@Input()  
config: any;
```

Inject the Router class in the constructor as shown below.

```
constructor(private router: Router) { }
```

We will add the method to handle the `pageChange` event for our paginator. The definition of this method is shown below.

```
pageChange(newPage: number) {  
  this.router.navigate(['/page/', newPage]);  
}
```

We will add the `changePageItemCount` method in the `PaginatorComponent` class. This method is used to set up a dynamic page size for the paginator. It will set the number of items to show on each page based on a selection from the drop-down list.

```
changePageItemCount(selectedItem) {  
    localStorage.setItem('pageSize', selectedItem.value);  
    this.config.itemsPerPage = selectedItem.value;  
}
```

We are storing the value selected by the user in the local storage. This is to ensure that the value won't be lost on page refresh and the application has a smooth user experience.

Open `src/app/components/paginator/paginator.component.html` and replace what is there with the code shown below.

```
<div class="paginator-controls">  
  <div>  
    <pagination-controls (pageChange)="pageChange($event)" class="my-  
pagination"></pagination-controls>  
  </div>  
  <div>  
    <mat-form-field>  
      <mat-label>Items per page: </mat-label>  
      <mat-select [(ngModel)]="config.itemsPerPage"  
(selectionChange)="changePageItemCount($event)">  
        <mat-option *ngFor="let page of pageSizeOptions" [value]="page">  
          {{ page }}  
        </mat-option>  
      </mat-select>  
    </mat-form-field>  
  </div>  
</div>
```

Finally, we will add styling for `PaginatorComponent`. Open `src/app/components/paginator/paginator.component.scss` and replace what is there with the style definitions shown below.

```
.my-pagination ::ng-deep .ngx-pagination {  
  margin: 10px 0px 10px 0px;  
  padding-inline-start: 0px;
```

```
}  
.paginator-controls {  
  display: flex;  
  justify-content: space-between;  
  padding-top: 10px;  
}  
@media screen and (min-width: 320px) and (max-width: 420px) {  
  .paginator-controls {  
    flex-direction: column-reverse;  
  }  
}
```

Now we will add a new router link in `app.module.ts` to support the pagination as shown below.

```
{ path: 'page/:pagenum', component: HomeComponent },
```

Add the PaginatorComponent to the BlogCard

To enable pagination, we need to add the `PaginatorComponent` to the `BlogCardComponent`. Open `src/app/components/blog-card.component.ts` and declare two properties as shown below.

```
config: any;  
pageSizeOptions = [];
```

Add the import for `ActivatedRoute` in the component as shown below.

```
import { ActivatedRoute } from '@angular/router';
```

Inject the `ActivatedRoute` class in the constructor as shown below.

```
constructor(  
    // other services  
    private route: ActivatedRoute) { }
```

Add the following codes snippet inside the constructor of BlogCardComponent class to initialize the newly declared properties.

```
this.pageSizeOptions = [2, 4, 6];  
const pageSize = localStorage.getItem('pageSize');  
this.config = {  
    currentPage: 1,  
    itemsPerPage: pageSize ? +pageSize : this.pageSizeOptions[0]  
};
```

Now update the ngOnInit method as shown below.

```
ngOnInit() {  
    this.route.params.subscribe(  
        params => {  
            this.config.currentPage = +params['pagenum'];  
            this.getBlogPosts();  
        }  
    );  
}
```

Update the BlogCardComponent template

Open src/app/components/blog-card.component.html and add the paginator component as shown below.

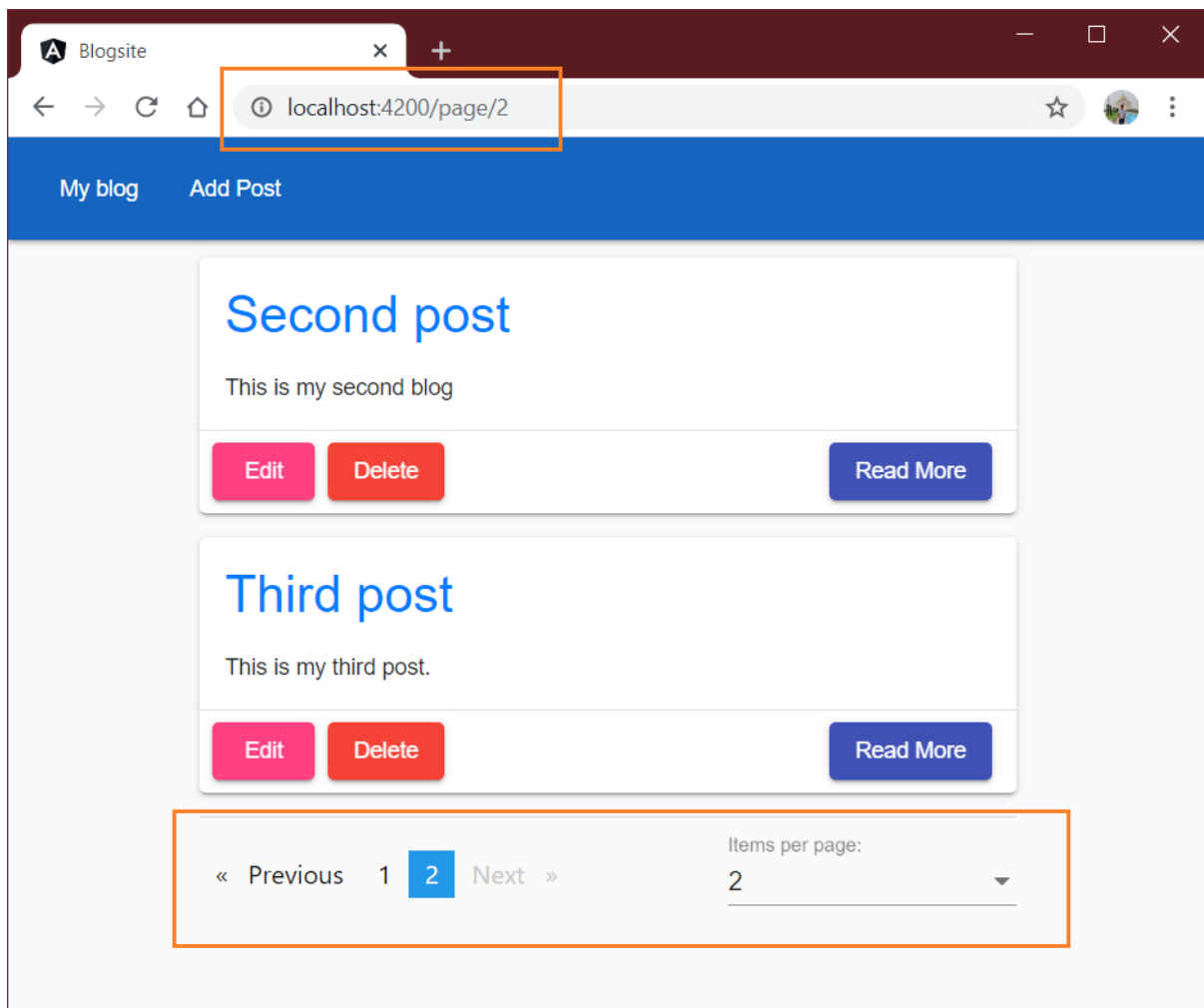
```
<app-paginator [pageSizeOptions]="pageSizeOptions"  
[config]="config"></app-paginator>
```

We will also add a paginate pipe in the ngFor directive while iterating through the list of blog posts as shown below.

```
<div *ngFor="let post of blogPost | paginate: config">
```

Checkpoint 5

Open the browser and you can see a paginator on the home page. You can jump through the pages and the URL will change with the updated page number. You can also see a drop-down list besides the paginator which will allow you to select the number of items to be shown on each page. Refer to the image shown below.

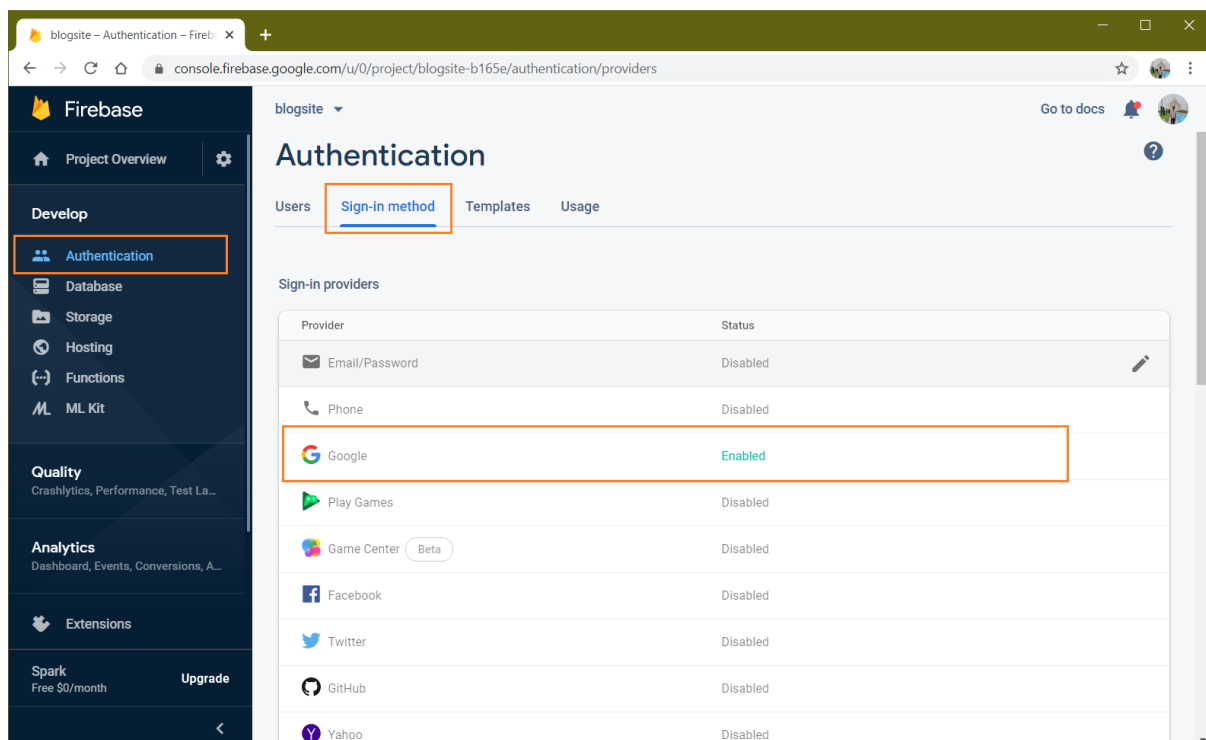


Add the Google authentication

We will add the feature of “login with Google account” into our application. We need to enable the authentication feature from the Firebase console. Follow the steps mentioned below to enable Google authentication on Firebase.

- **Step 1:** Navigate to the "Project Overview" page of your Firebase project.
- **Step 2:** Select “Authentication” under the “Develop” menu from the list on the left.
- **Step 3:** Navigate to the "Sign-in method" tab.
- **Step 4:** Click on "Google" from the available list of "Sign-in providers" on the page.
- **Step 5:** Click on Enable toggle button then click on "Save"

Refer to the image shown below.



Now we will configure our application to use the Google authentication provided by Firebase.

Create the AppUser model

Create a new file `src/app/models/appuser.ts` and put the following code.

```
export class AppUser {  
    name: string;  
    email: string;  
    isAdmin: boolean;  
    photoURL: string;  
}
```

Create the Authentication service

We will create a service to handle the authentication. Create a new service using the command shown below.

```
ng g s services/auth
```

Open the <src/app/services/auth.service.ts> file and add the following import definitions.

```
import { AppUser } from '../models/appuser';  
import { Observable, of } from 'rxjs';  
import { AngularFireAuth } from '@angular/fire/auth';  
import { ActivatedRoute, Router } from '@angular/router';  
import { AngularFireStore } from '@angular/fire/firestore';  
import { switchMap } from 'rxjs/operators';  
import * as firebase from 'firebase/app';
```

Declare an observable of type `AppUser` in the `AuthService` class as shown below.

```
appUser$: Observable<AppUser>;
```

Inject the services in the constructor as shown below.

```
constructor(  
    public afAuth: AngularFireAuth,
```



```
private route: ActivatedRoute,  
private router: Router,  
private db: AngularFirestore  
)
```

The observable `appUser$` will get the auth state of the user. If the user is logged in, it will fetch the user details from the Firestore database, else it will return null. Put the following code in the constructor of the `AuthService` class.

```
this.appUser$ = this.afAuth.authState.pipe(  
  switchMap(user => {  
    if (user) {  
      return this.db.doc<AppUser>(`appusers/${user.uid}`).valueChanges();  
    } else {  
      return of(null);  
    }  
  })  
);
```

We will also add a method `updateUserData` to save the user data into our database upon successful login. We will store the name, email address and photo URL of the photo in Google account for each user in our database. The method definition is shown below.

```
private updateUserData(user) {  
  const userRef = this.db.doc(`appusers/${user.uid}`);  
  const data = {  
    name: user.displayName,  
    email: user.email,  
    photoURL: user.photoURL  
  };  
  return userRef.set(data, { merge: true });  
}
```

We are storing this data for the following two reasons:

- Track all the users who logged into our application.
- Display user name and photo on the nav-bar after successful login.

Add the method definition for login and logout in the `AuthService` class as shown below.

```
async login() {
    const returnUrl = this.route.snapshot.queryParamMap.get('returnUrl')
    || this.router.url;
    localStorage.setItem('returnUrl', returnUrl);

    const credential = await this.afAuth.auth.signInWithPopup(new
firebase.auth.GoogleAuthProvider());
    return this.updateUserData(credential.user);
}

async logout() {
    await this.afAuth.auth.signOut().then(() => {
        this.router.navigate(['/']);
    });
}
```

We will get the `returnUrl` from the route and store its value in the local storage. We are storing the value in the local storage to ensure that the value won't be lost during page redirect or page refresh.

The `signInWithPopup` method will authenticate a Firebase client using a popup-based OAuth authentication flow. If the login is successful, it will return the signed-in user along with the provider's credentials. If sign-in is unsuccessful, it will return an error object containing additional information about the error. The `logout` method will sign out the current user and navigate the user to the home page.

Update the AppComponent

Open `src/app/app.component.ts` and add the following import definitions at the top.

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

Inject the services in the constructor as shown below.

```
constructor(
    private authService: AuthService,
    private router: Router
```

```
) { }
```

We will subscribe to the observable `appUser$` inside the `ngOnInit` method of the `AppComponent` class as shown below.

```
this.authService.appUser$.subscribe(user => {  
  if (!user) {  
    return;  
  } else {  
    const returnUrl = localStorage.getItem('returnUrl');  
    if (!returnUrl) {  
      return;  
    }  
    localStorage.removeItem('returnUrl');  
    this.router.navigateByUrl(returnUrl);  
  }  
});
```

If the user is logged in, we will fetch the value of the return URL from the local storage. If the return URL is available, the user will be navigated to it. We will then remove the return URL from the local storage.

Add Login button in the Navigation bar

Open `src/app/components/nav-bar/nav-bar.component.ts` and add the following import definitions at the top.

```
import { AuthService } from 'src/app/services/auth.service';  
import { AppUser } from 'src/app/models/appuser';
```

We will declare a property to hold the user data. We will also Inject the `AuthService` in the constructor. Refer to the code snippet shown below.

```
appUser: AppUser;  
  
constructor(private authService: AuthService) {}
```

We will subscribe to the observable `appUser$` from `AuthService` and set the `appUser` property. Implement the `OnInit` interface on the `NavBarComponent` class. Add the following line of code in the `ngOnInit` method.

```
this.authService.appUser$.subscribe(appUser => this.appUser = appUser);
```

We will also add the method to handle login and logout from our application inside the `NavBarComponent` class as shown below.

```
login() {  
    this.authService.login();  
}  
  
logout() {  
    this.authService.logout();  
}
```

We will update the template for the navigation bar. Open the `nav-bar.component.html` file and replace the existing code with the code shown below.

```
<mat-toolbar class="nav-bar mat-elevation-z2">  
  <button mat-button [routerLink]='["/"]'> My blog </button>  
  <ng-container *ngIf="appUser">  
    <button mat-button [routerLinkActive]='["link-active"]'  
[routerLink]='["/addpost"]'>  
      Add Post  
    </button>  
  </ng-container>  
  <span class="spacer"></span>  
  
  <ng-template #anonymousUser>  
    <button mat-button (click)="login()">Login with Google</button>  
  </ng-template>  
  <ng-container *ngIf="appUser; else anonymousUser">  
    <img mat-card-avatar class="user-avatar" src={{appUser.photoURL}}>  
    <button mat-button [matMenuTriggerFor]="menu">  
      {{appUser.name}}<mat-icon>arrow_drop_down</mat-icon>  
    </button>  
    <mat-menu #menu="matMenu">  
      <button mat-menu-item (click)="logout()">Logout</button>
```

```
</mat-menu>
</ng-container>
</mat-toolbar>
```

If the user is logged in, we will display the name and the photo as per the user's Google account. If the user is not logged in, we will display a "Login with Google" button on the nav-bar.

We will add the following style definition in `src/app/components/nav-bar/nav-bar.component.scss` file.

```
.user-avatar {
  height: 40px;
  width: 40px;
  border-radius: 50%;
  flex-shrink: 0;
}
```

Update the App module

Import the `AngularFireAuthModule` into `src/app/app.module.ts` file as shown below.

```
import { AngularFireAuthModule } from '@angular/fire/auth';

@NgModule({
  ...
  imports: [
    ...
    AngularFireAuthModule,
  ],
})
```

Authenticated access for Edit and Delete

Open `src/app/components/blog-card.component.ts` and add the following two import definitions at the top.

```
import { AppUser } from 'src/app/models/appuser';  
import { AuthService } from 'src/app/services/auth.service';
```

Similar to the Nav-bar component, we will declare a property in `BlogCardComponent` class and Inject the `AuthService` in the constructor. Refer to the code snippet shown below.

```
appUser: AppUser;  
  
constructor(  
    // other services  
    private authService: AuthService) {}
```

We will subscribe to the observable `appUser$` from `AuthService` and set the `appUser` property. Add the following line of code in the `ngOnInit` method of the `BlogCardComponent` class.

```
this.authService.appUser$.subscribe(appUser => this.appUser = appUser);
```

Open <src/app/components/blog-card.component.html> and add a `ngIf` directive to the `<ng-container>` tag which contains the Edit and Delete buttons. This will restrict the Edit and Delete functionality for logged in users only.

```
<ng-container *ngIf="appUser">
```

At this point in time, we are allowing the Edit and Delete feature for any logged-in user. In the next section, we will implement the feature of authorization. We will then update this code to allow these features for admin users only.

Capture the Author name

Since we are allowing only the logged-in user to add a new blog, we can capture the author's name. The author's name will be the same name as his Google account name. We will display the author's name on the blog page alongside the blog creation date.

Add the following import definitions in the `src/app/components/blog-editor/blog-editor.component.ts` file.

```
import { AuthService } from 'src/app/services/auth.service';
import { AppUser } from 'src/app/models/appuser';
```

We will declare a property to hold the user data. We will Inject the `AuthService` in the constructor of `BlogEditorComponent`. Inside the `ngOnInit` method, we will subscribe to the observable `appUser$` from `AuthService` and set the `appUser` property. Refer to the code snippet shown below.

```
appUser: AppUser;

constructor(
  // other service injection
  private authService: AuthService) { }

ngOnInit() {
  ...
  this.authService.appUser$.subscribe(appUser => this.appUser =
appUser);
}
```

We will set the author's property of the `postData` object while saving a new blog. Add the following line of code inside the else section of the `saveBlogPost` method, just before invoking the `createPost` method of `BlogService`.

```
this.postData.author = this.appUser.name;
```

Add the following line in the `src/app/components/blog/blog.component.html` file. Add this code after the line where we are binding the `createdDate` inside the `<mat-card-subtitle>` tag. This is used to display the author's name just beside the blog creation date.

```
<i class="fa fa-user" aria-hidden="true"></i> {{postData.author}}
```

Secure the routes

We will add an auth guard to our application to restrict unauthorized access to certain routes. To create a new guard, run the command as shown below.

```
ng g g guards/auth --implements CanActivate
```

Open the `src/app/guards/auth.guard.ts` file and add the following import definitions.

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

Add a constructor in the `AuthGuard` class. Inject `Router` and `AuthService` in the constructor as shown in the below code snippet.

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


We will update the `canActivate` method to handle the unauthenticated access to the routes. Add the following code in this method.

```
return this.authService.appUser$.pipe(map(user => {  
  if (user) {  
    return true;  
  }  
  this.router.navigate(['/'], { queryParams: { returnUrl: state.url }  
});  
  return false;  
}));
```

We will subscribe to the observable `appUser$` to fetch the authentication state of the user. If the user is logged in, we will return true. If the user is not logged in, the following three things will happen: -

- Set the query parameter called `returnUrl` to the value of the current URL.
- Navigate the user to the home page.
- Return false from the method.

Add route guards in App module

To activate route guard for a particular route, we need to add `canActivate` property to the route in the `app.module.ts`. We will secure the routes for adding a new post by adding the `canActivate` property.

Add the following import definition in the `app.module.ts` file.

```
import { AuthGuard } from './guards/auth.guard';
```

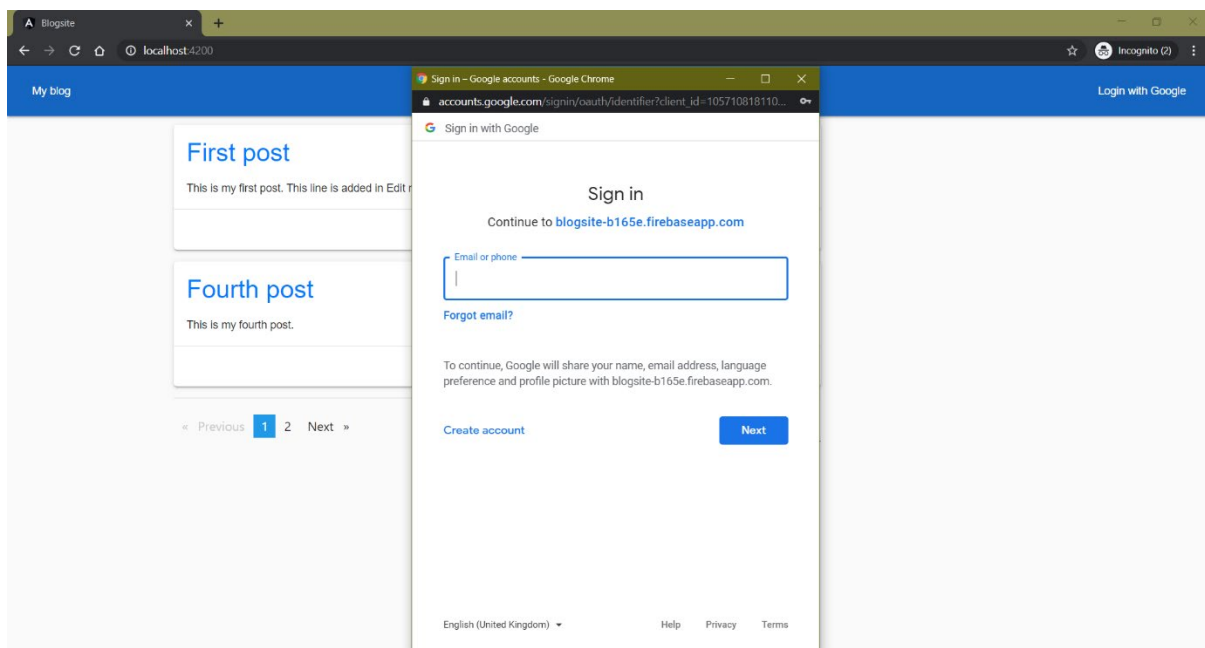
Update the route for 'addpost' as shown below.

```
{ path: 'addpost', component: BlogEditorComponent, canActivate:  
[AuthGuard] },
```

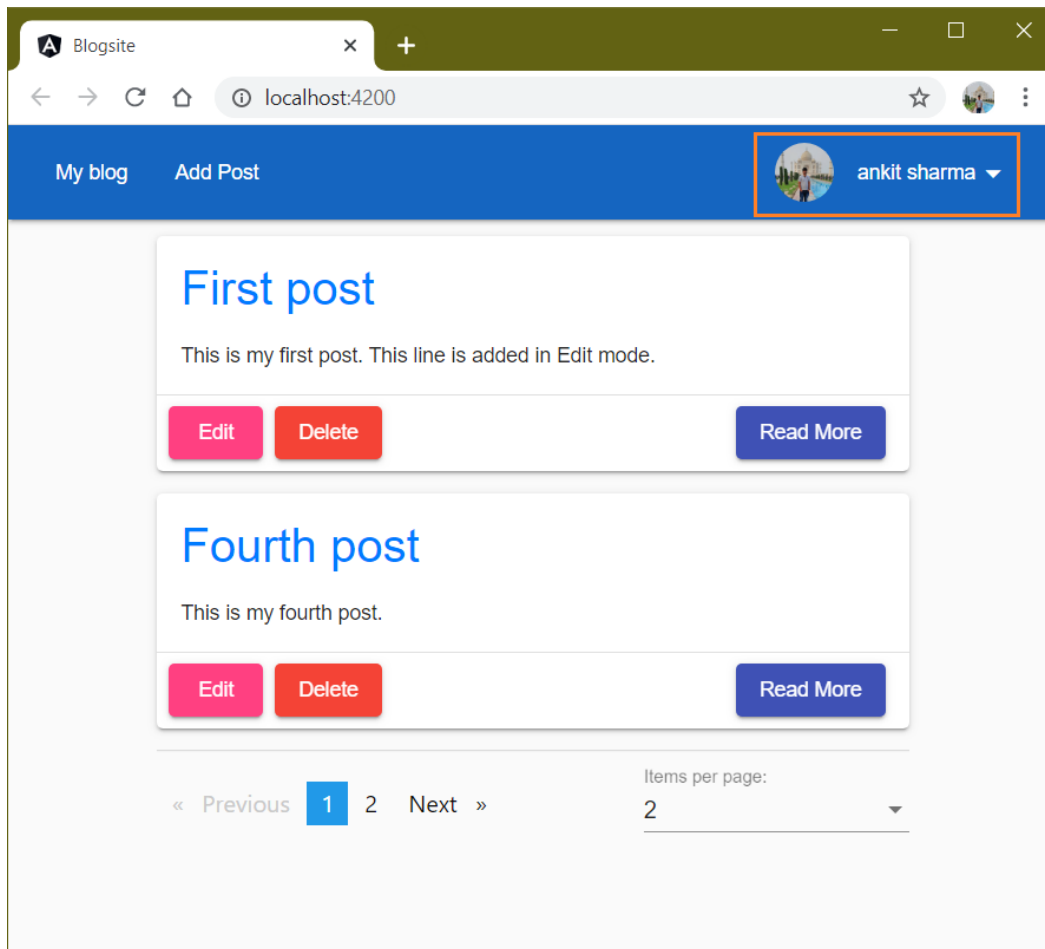
When the user tries to access the ‘addpost’ route, the AuthGuard class will be invoked. If the AuthGuard class returns true, the user will be allowed to access the ‘addpost’ route, else the access to this route will be denied for the user.

Checkpoint 6

Open the browser and you can observe that the blog card doesn’t have the Edit and Delete button. Also, the nav-bar does not show the “Add post” button. This is because the user has not logged in. You can also see a “Login with Google” button on the top right corner of the nav-bar. Click on this button, a pop-up modal will open asking you to login with your Google account. Refer to the image shown below for the reference.



The page will refresh once Google authentication is successful. You can see that the route for the “Add post” appears on the nav-bar. The top right corner of the nav-bar also displays the user name of the logged-in user. The blog card will show the Edit and Delete button now. Refer to the image shown below.



Update Firestore database security rules

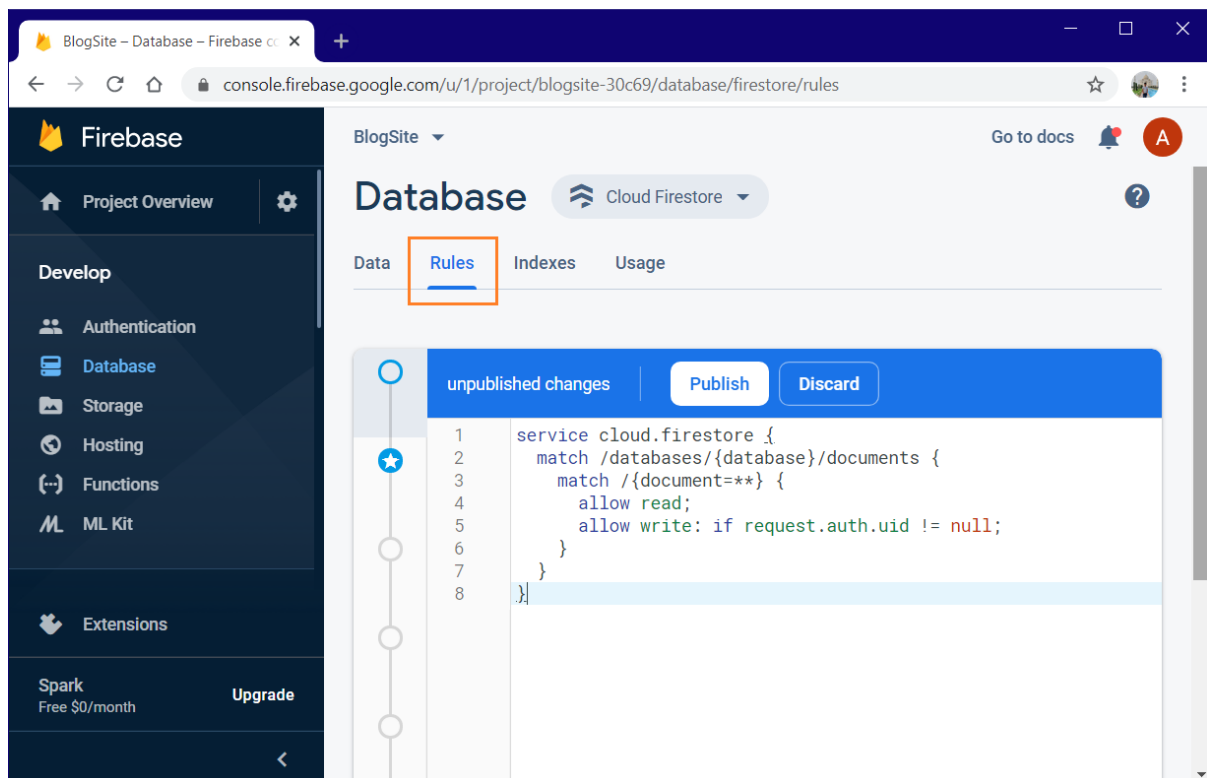
While creating the database, we created it in the “test” mode. We will now update the database security rule to allow the write operation only for the authenticated user.

Select “Database” under the “Develop” section from the menu on the left. Click on the “Rules” tab. Update the rules as shown below.

```
service cloud.firestore {  
  match /databases/{database}/documents {  
    match /{document=**} {
```

```
    allow read;
    allow write: if request.auth.uid != null;
  }
}
```

Click on the “Publish” button to publish the rule. Refer to the image shown below.



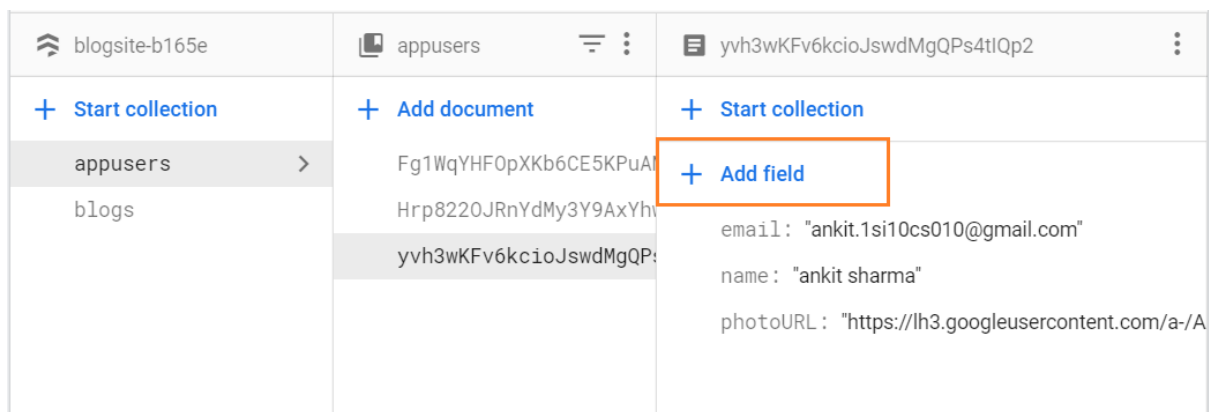
Implementing Authorization

We will implement role-based authorization for our website. We will define a role – admin and only the users with the admin role have the permissions to edit and delete a blog. In a future section, we will add the feature of posting comments on the blog. The admin user will have permission to delete the comments also.

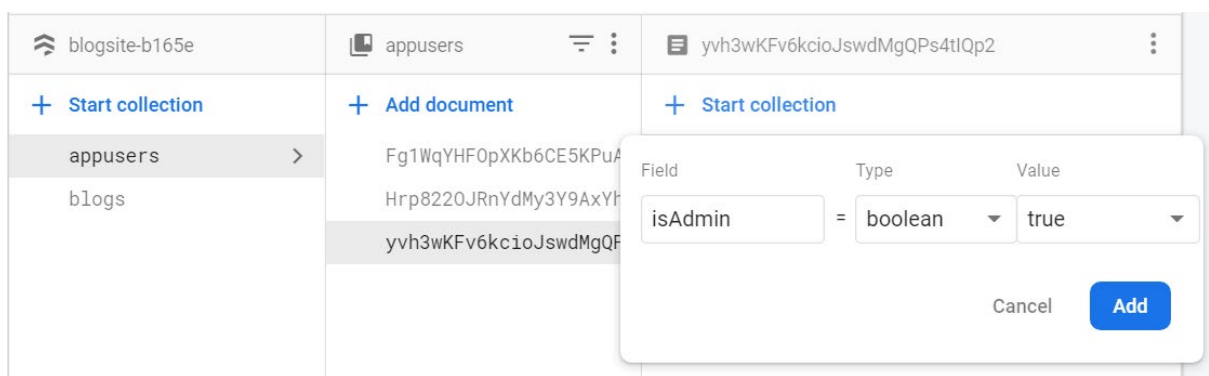
Configure the Firestore database for admin role

We have defined a Boolean property `isAdmin` in the `AppUser` class. This property is used to define the admin role for a user. We will manually add a new field called `isAdmin` for the users we want to provide admin access. We will make the changes in the `appusers` collection inside the Firestore database.

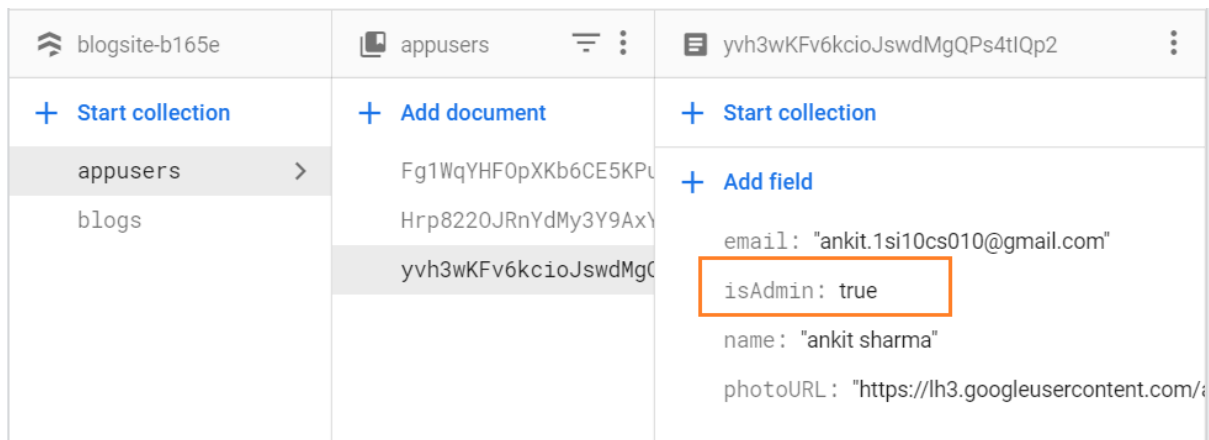
Navigate to the Firestore database and open the `appusers` collection. This collection stores the record of all the users who have logged in to our application. Refer to the image shown below.



Click on the “Add field” button. A popup box will open asking you to define a new field. Set the field name as `isAdmin`, Type as boolean and Value as true. Click on the Add button to add the new field. Refer to the image shown below.



The new field will be added to the collection. Refer to the image shown below.



Upon successful login, we will fetch the user details from the `appusers` collection and bind it to the object of type `AppUser` class. We will then use the `isAdmin` property to restrict the access for the user.

Since we are creating a demo application having only one role, therefore we are setting the `isAdmin` property manually. This is not recommended for an enterprise-level application where we might have to manage multiple roles. Ideally, we should create a separate application to manage roles for all the users.

Create the admin-auth guard

Execute the command shown below to create a new guard named `AdminAuthGuard`.

```
ng g g guards/admin-auth --implements CanActivate
```

Open `src/app/guards/admin-auth.guard.ts` and add the following import statements at the top.

```
import { AuthService } from '../services/auth.service';
import { map } from 'rxjs/operators';
import { AppUser } from '../models/appuser';
```

Add a constructor and inject Router and AuthService in it as shown in the below code snippet.

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

We will update the `canActivate` method to handle unauthorized access to the routes.

Update the `canActivate` method by adding the following code.

```
return this.authService.appUser$.pipe(map((user: AppUser) => {  
    if (user && user.isAdmin) {  
        return true;  
    }  
    this.router.navigate(['/'], { queryParams: { returnUrl: state.url }  
}));  
    return false;  
}));
```

We will subscribe to the observable `appUser$` to fetch the authentication state of the user. If the user is logged in and the value of the `isAdmin` property for the user is set to true, we will return true. Similar to the behavior of the `AuthGuard` class, if the user is not logged in, the following three things will happen: -

- Set the query parameter called `returnUrl` to the value of the current URL.
- Navigate the user to the home page.
- Return false from the method.

Update the BlogCardComponent

In the previous section, we have updated the `BlogCardComponent` to restrict access to the Edit and Delete buttons to logged-in users. Now we will update the code again to allow access to admin users only. Open `src/app/components/blog-card.component.html` and update the `ngIf` directive in the `<ng-container>` tag which contains the Edit and Delete

buttons. Refer to the code snippet shown below. This will restrict the Edit and Delete functionality for logged in users only.

```
<ng-container *ngIf="appUser?.isAdmin">
```

Add route guards in App module

Add the following import statement in the `app.module.ts` file.

```
import { AdminAuthGuard } from './guards/admin-auth.guard';
```

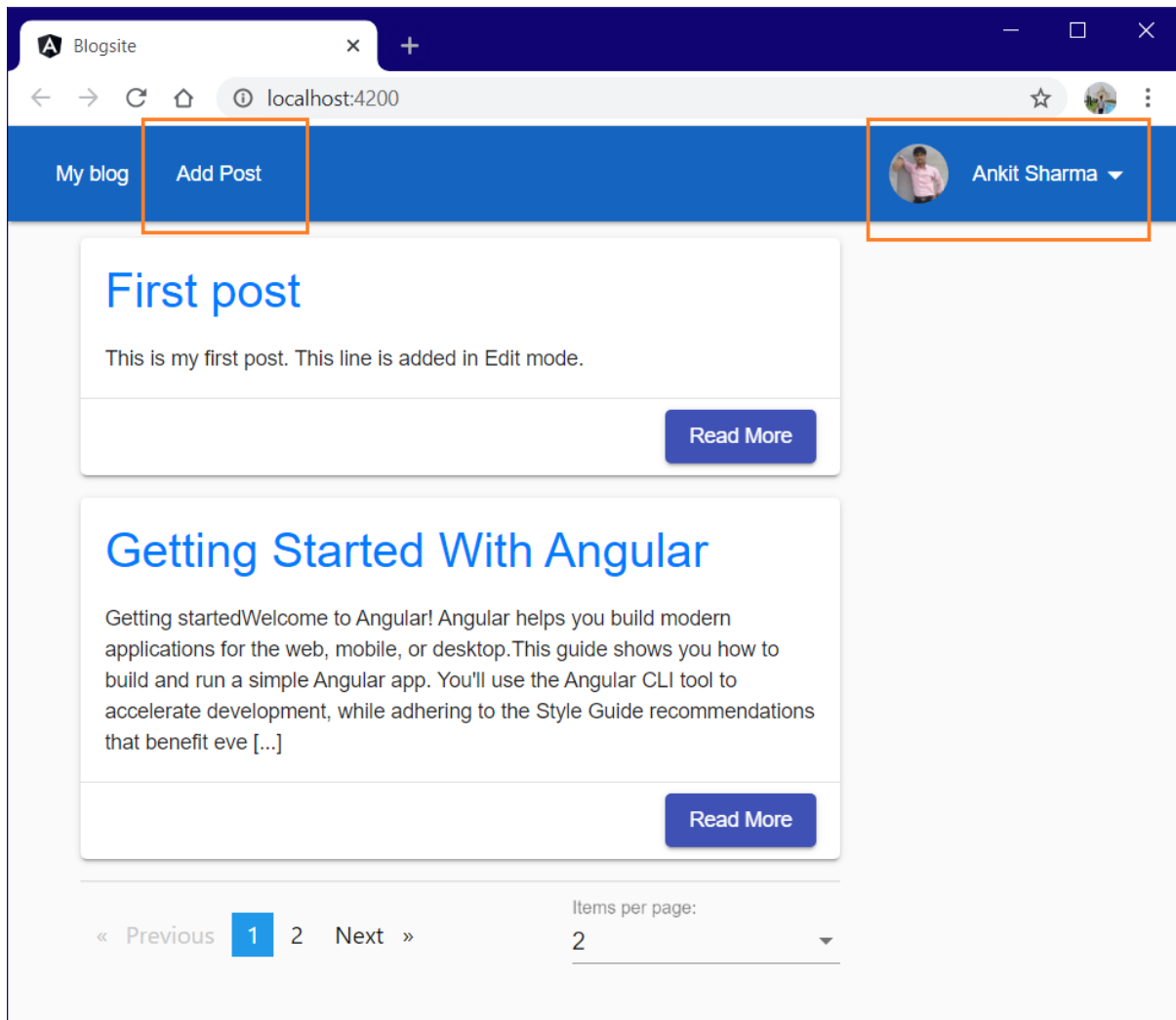
Update the route for “editpost” as shown below.

```
{ path: 'editpost/:id', component: BlogEditorComponent, canActivate: [AdminAuthGuard] },
```

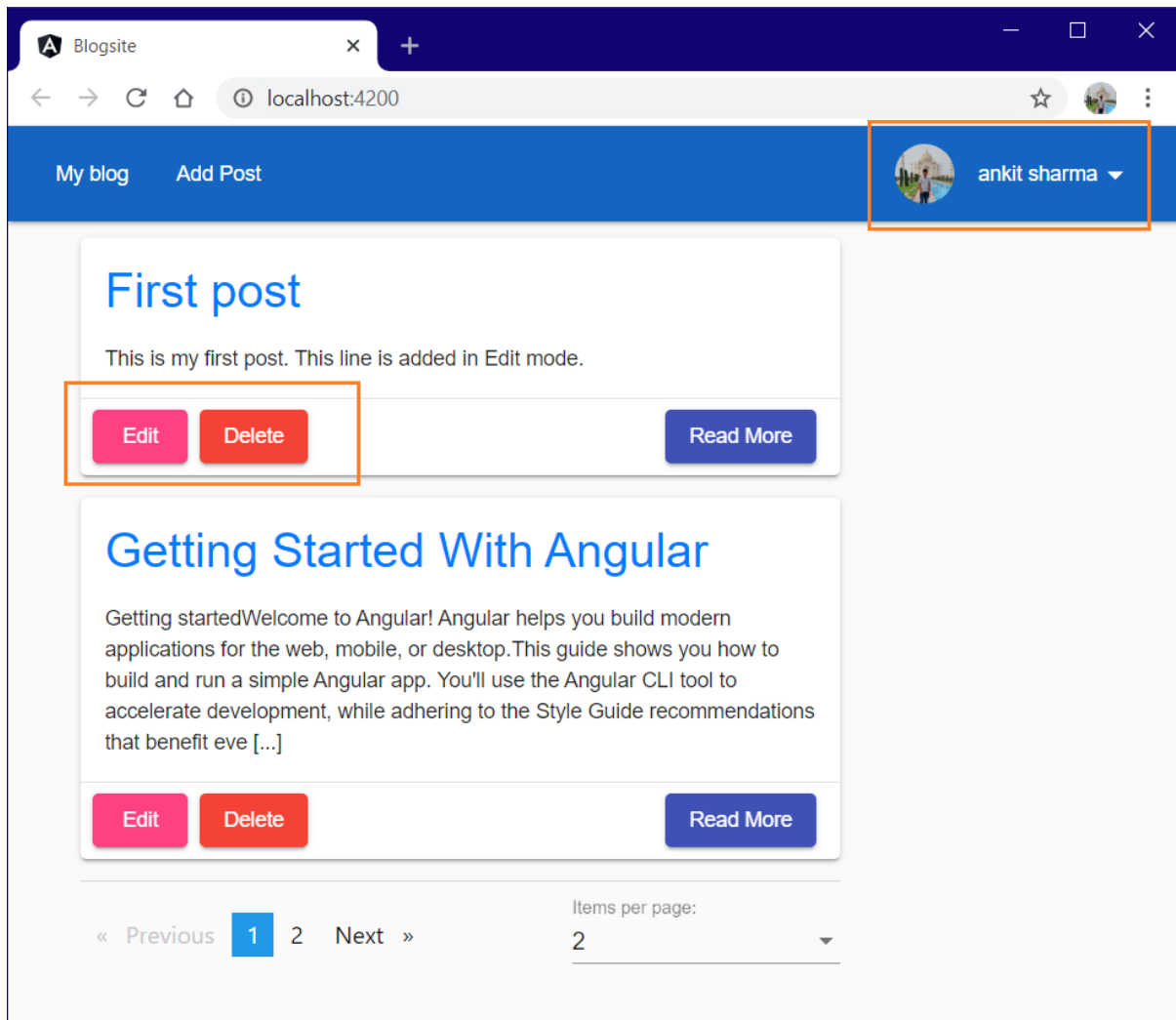
Here we are restricting the ability to edit the post to the admin users only. When the user tries to access the “editpost” route, the `AdminAuthGuard` class will be invoked. If the `AdminAuthGuard` class returns true, the user will be allowed to access to the “editpost” route, otherwise the access to this route will be denied for the user.

Checkpoint 7

Open the browser and log in to the application with a non-admin user. After login, you can see the “AddPost” link on the nav-bar, but the Edit and Delete buttons won’t be visible on the blog card. This means the user can add a new blog, but he does not have the permission to edit or delete any existing blog. Refer to the image shown below.



Navigate to the Firestore database and add a new field `isAdmin` for the user as we have discussed earlier. Alternatively, you can also log out and log in again with another user for which the `isAdmin` field is set to `true`. This time you can see the Edit and Delete buttons on the blog card. Refer to the image shown below.



Adding the author profile

We will show the author's profile on the home page. Execute the command shown below to create the author-profile component.

```
ng g c components/author-profile
```

We are going to show an image of the author along with his social media links. The image will be served from within the app itself. We will place the author's image inside the `src/assets` folder.

Open `src/app/components/author-profile/author-profile.component.html` and put the following code inside it.

```
<mat-card class="rightpanel-card mat-elevation-z2">
  <mat-card-content>
    <h4 class="rightdivtext">
      Author
    </h4>
  </mat-card-content>
  <mat-card-content>
    <div class="authorimagecontainer">
      
      <h5>Ankit Sharma</h5>
    </div>
  </mat-card-content>
  <mat-divider></mat-divider>
  <mat-card-content>
    <h4 class="rightdivtext">
      Follow Me
    </h4>
  </mat-card-content>
  <mat-card-content>
    <a href="https://www.facebook.com/Ankit.Sharma.0709"
target="_blank"><i class="fa fa-facebook-square"
      aria-hidden="true"></i></a>
    <a href="https://twitter.com/ankitsharma_007" target="_blank"><i
class="fa fa-twitter-square"
      aria-hidden="true"></i></a>
    <a href="https://www.linkedin.com/in/ankitsharma-007/"
target="_blank"><i class="fa fa-linkedin-square"
      aria-hidden="true"></i></a>
    <a href="https://github.com/AnkitSharma-007" target="_blank"><i
class="fa fa-github-square"
      aria-hidden="true"></i></a>
  </mat-card-content>
</mat-card>
```

Open `src/app/components/author-profile/author-profile.component.scss` and put the following style definitions in it.

```
.fa-twitter-square {
  color: #55acee;
}

.fa-facebook-square {
```

```
        color: #3b5998;
    }

    .fa-linkedin-square {
        color: #0976b4;
    }

    .fa-github-square {
        color: #333;
    }

    .fa {
        font-size: 3em;
        width: 1em;
        margin-top: 5px;
        cursor: pointer;
    }

    .mat-card-avatar {
        width: 100px;
        height: 100px;
        margin: auto;
        padding: 5px;
    }

    .authorimagecontainer {
        text-align: center;
    }

    .rightdivtext {
        color: #636467;
        text-transform: uppercase;
        padding: 2px;
    }

    .rightpanel-card {
        margin-bottom: 15px;
    }
}
```

To show the author profile on the home page, we need to add the `AuthorProfileComponent` to the `HomeComponent`.

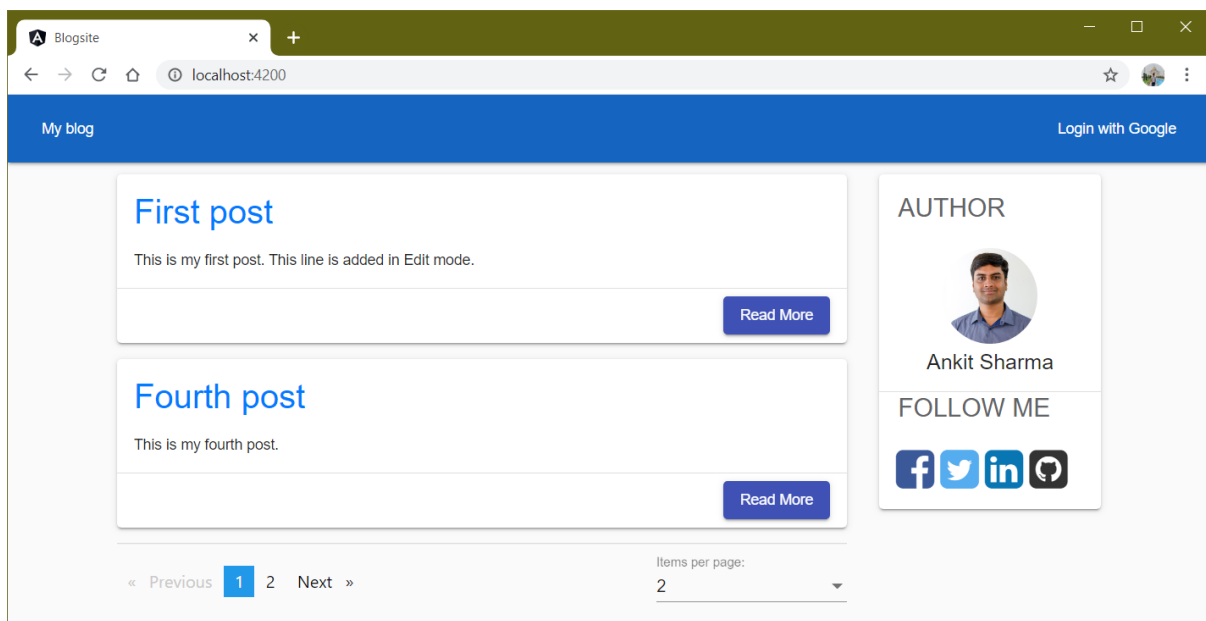
Open the `src/app/components/home/home.component.html` file and update the contents as shown below.

```
<div class="row left-panel">
```

```
<div class="col-md-9">
  <app-blog-card></app-blog-card>
</div>
<div class="col-md-3">
  <app-author-profile></app-author-profile>
</div>
</div>
```

Checkpoint 8

Open the browser and you can observe an author profile on the right-hand side of the home page. The profile will show the image of the author along with his social media links. Refer to the image shown below.



Add the scroller to the blog page

We will add a scroller to the blog page. It will display a “Scroll to top” button whenever we scroll down while reading the blog. Clicking on this button will move the user to the top of the page with a smooth transition.

Execute the command shown below to create the Scroller component.

```
ng g c components/Scroller
```

Open `src/app/components/scroller/scroller.component.ts` and replace the existing content with the following code.

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

@Component({
  selector: 'app-scroller',
  templateUrl: './scroller.component.html',
  styleUrls: ['./scroller.component.scss']
})
export class ScrollerComponent {

  showScroller: boolean;
  showScrollerPosition = 100;

  @HostListener('window:scroll')
  checkScroll() {
    const scrollPosition = window.pageYOffset ||
document.documentElement.scrollTop || document.body.scrollTop || 0;

    if (scrollPosition >= this.showScrollerPosition) {
      this.showScroller = true;
    } else {
      this.showScroller = false;
    }
  }

  gotoTop() {
    window.scroll({
      top: 0,
      left: 0,
      behavior: 'smooth'
    });
  }
}
```

We will set the value of the variable `showScrollerPosition` to 100. This is the value in pixels after which the scroller will be visible. The `checkScroll` method will listen for the scroll event using the `@HostListener` decorator. We will calculate the current scroll

position of the page. If the current scroll position is greater than the `showScrollerPosition`, we will display the scroll to top button on the page.

The `gotoTop` method will set the behavior of the scroller. This method will scroll the page to the top with a smooth transition.

Open `src/app/components/scroller/scroller.component.html` and replace the existing content with the following code.

```
<div *ngIf="showScroller" (click)="gotoTop()" class="scroll-to-top"><i class="fa fa-angle-up"></i></div>
```

Open `src/app/components/scroller/scroller.component.scss` and put the following code inside it.

```
.scroll-to-top {
  display: block;
  background: rgba(100, 100, 100, 0.4);
  color: #ffffff;
  bottom: 4%;
  cursor: pointer;
  position: fixed;
  right: 20px;
  z-index: 999;
  font-size: 24px;
  text-align: center;
  width: 45px;
  height: 45px;
  border-radius: 50%;

  .fa {
    font-weight: 900;
  }
}

.scroll-to-top:hover {
  background-color: #b2b2b2;
}
```

Now we will add the `ScrollerComponent` to the `BlogComponent`. Open `src/app/components/blog/blog.component.html` and add the following line at the end of the file.

```
<app-scroller></app-scroller>
```

We have successfully created a scroller for the blog page. We will see the output demo for the scroller along with the comment feature in the next section.

Post comment on the blog

We will add the feature of posting a comment on each blog. Any logged-in user can post a comment on the blog. The admin user has permission to delete individual comments on the blog. If we delete a blog, all the comments posted on that blog will also be deleted.

Create the comment model

Create a new file [src/app/models/comment.ts](#) and put the following code inside it.

```
export class Comments {  
  commentId: string;  
  blogId: string;  
  email: string;  
  commentedBy: string;  
  content: string;  
  commentDate: any;  
}
```

Create the comment service

We will create a service to handle the database related operations on comments. Execute the command shown below to create the Comment Service.

```
ng g s services/Comment
```


Open `src/app/services/comment.service.ts` and add the following import statements at the top.

```
import { AngularFirestore } from '@angular/fire/firestore';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';
import { Comments } from '../models/comment'
```

Now add the following method definitions inside the `CommentService` class.

```
export class CommentService {

  constructor(private db: AngularFirestore) { }

  saveComment(comment: Comments) {
    const commentData = JSON.parse(JSON.stringify(comment));
    return this.db.collection('comments').add(commentData);
  }

  getAllCommentsForBlog(blogId: string): Observable<Comments[]> {
    const comments = this.db.collection<Comments>('comments',
      ref => ref.where('blogId', '==', blogId).orderBy('commentDate',
        'desc')).snapshotChanges().pipe(
      map(actions => {
        return actions.map(
          c => ({
            commentId: c.payload.doc.id,
            ...c.payload.doc.data()
          }));
      }));
    return comments;
  }

  deleteAllCommentForBlog(blogId: string) {
    const commentsToDelete = this.db.collection('comments', ref =>
      ref.where('blogId', '==', blogId)).snapshotChanges();

    commentsToDelete.forEach(
      commentList => {
        commentList.forEach(comment => {
          this.db.doc('comments/' + comment.payload.doc.id).delete();
        });
      }
    );
  }
}
```

```
}

deleteSingleComment(commentId: string) {
    return this.db.doc('comments/' + commentId).delete();
}
}
```

The `saveComment` method will accept an object of type `Comments` as a parameter. We will parse the parameter to a JSON object and add it to the 'comments' collection in our database. If the collection already exists then the JSON object will get added to it. However, if the collection does not exist in the database, the add method will create the collection and then add the new object to it.

The `getAllCommentsForBlog` method will accept the blog Id as a parameter. This method will query the 'comments' collection and return the list of all the comments corresponding to the blog Id passed to it. The list of comments is sorted by `commentDate` in a descending fashion. This is to ensure that the latest comment stays on the top of the list.

The `deleteAllCommentForBlog` method will accept the blog Id as a parameter. This method will delete all the comments from the 'comments' collection based on the blog Id passed to it.

The `deleteSingleComment` method will accept the comment Id as a parameter. This method will delete a single comment from the 'comments' collection based on the comment Id passed to it.

Create the Comment Component

We will create a component that will handle the comments posted by the user. Execute the command shown below to create the Comments component.

```
ng g c components/Comments
```

Open the `src/app/components/comments/comments.component.ts` file and add the following import statements at the top.

```
import { Input, OnDestroy } from '@angular/core';
import { DatePipe } from '@angular/common';
import { AppUser } from 'src/app/models/appuser';
import { Comments } from 'src/app/models/comment';
import { CommentService } from 'src/app/services/comment.service';
import { AuthService } from 'src/app/services/auth.service';
import { SnackbarService } from 'src/app/services/snackbar.service';
import { Subject } from 'rxjs';
import { takeUntil } from 'rxjs/operators';
```

Add the provider for the DatePipe under in the @Component decorator section as shown below.

```
@Component({
  ...
  providers: [DatePipe]
})
```

We will update the CommentsComponent class as shown below.

```
export class CommentsComponent implements OnInit, OnDestroy {

  @Input()
  blogId;

  appUser: AppUser;
  public comments = new Comments();
  commentList: Comments[] = [];
  private unsubscribe$ = new Subject<void>();

  constructor(private datePipe: DatePipe,
    private commentService: CommentService,
    private authService: AuthService,
    private snackBarService: SnackbarService) { }
}
```

This component will accept blogId as an input. We will inject the services in the constructor of the class.

Now add the following method definitions inside the `CommentsComponent` class.

```
ngOnInit() {
    this.authService.appUser$.subscribe(appUser => this.appUser =
appUser);
    this.getAllComments();
}

onCommentPost(commentForm) {
    this.comments.commentDate = this.datePipe.transform(Date.now(), 'MM-
dd-yyyy HH:mm:ss');
    this.comments.blogId = this.blogId;
    this.commentService.saveComment(this.comments).then(
commentForm.resetForm()
    );
}

getAllComments() {
    this.commentService.getAllCommentsForBlog(this.blogId)
    .pipe(takeUntil(this.unsubscribe$))
    .subscribe(result => {
        this.commentList = result;
    });
}

deleteComment(commentId) {
    if (confirm('Do you want to delete this comment!!!')) {
        this.commentService.deleteSingleComment(commentId).then(
        () => {
            this.snackBarService.showSnackBar('Comment Deleted
successfully');
        }
    );
}

login() {
    this.authService.login();
}

ngOnDestroy() {
    this.unsubscribe$.next();
    this.unsubscribe$.complete();
}
```

Inside the `onCommentPost` method, we will set the `commentDate` to the current date. We will also set the `blogId` property of the comment object to the id of the blog for which the

comment is posted. We will invoke the `saveComment` method of the `CommentService` to store the comment in the database.

The `getAllComments` method will invoke the `getAllCommentsForBlog` from the `CommentService` and supply the `blogId` as the parameter. This method will fetch the list of all the comments posted on the blog.

The `deleteComment` method will allow us to delete a particular comment. It will display a confirmation box. If the user confirms the delete action, we will invoke the `deleteSingleComment` method from the `CommentService` to delete a comment.

The `login` method will allow the user to login to the application using the Google account.

Open `src/app/components/comments/comments.component.html` and replace the existing content with the code shown below.

```
<ng-template #anonymousUser>
  <mat-card class="comment-card mat-elevation-z2">
    <a (click)="login()">Login with Google</a> to post comments
  </mat-card>
</ng-template>
<mat-card *ngIf="appUser; else anonymousUser" class="comment-card mat-elevation-z2">
  <mat-card-title>
    LEAVE A REPLY
  </mat-card-title>
  <mat-card-subtitle>
    Your email address will not be published. Required fields are marked *
  </mat-card-subtitle>
  <mat-card-content>
    <form #commentForm="ngForm" (ngSubmit)="commentForm.form.valid && onCommentPost(commentForm)" novalidate>
      <mat-form-field class="full-width">
        <input matInput placeholder="Name" name="commentedBy" [(ngModel)]="comments.commentedBy" #commentedBy="ngModel" required>
        <mat-error *ngIf="commentForm.submitted && commentedBy.errors?.required">Name is required</mat-error>
      </mat-form-field>
      <mat-form-field class="full-width">
        <input matInput placeholder="Email" name="email" [(ngModel)]="comments.email" #email="ngModel" email
```

```

        required>
        <mat-error *ngIf="commentForm.submitted &&
email.errors?.required">Email is required</mat-error>
        <mat-error *ngIf="commentForm.submitted &&
email.errors?.email">Invalid email</mat-error>
        </mat-form-field>
        <mat-form-field class="full-width">
        <textarea matInput placeholder="Comment" name="content"
[(ngModel)]="comments.content"
        #content="ngModel" required></textarea>
        <mat-error *ngIf="commentForm.submitted &&
content.errors?.required">Comment is required</mat-error>
        </mat-form-field>
        <mat-card-actions>
        <button type="
submit" mat-raised-button color="primary">Post
Comment</button>
        </mat-card-actions>
    </form>
</mat-card-content>
</mat-card>
<mat-card *ngFor="let comment of commentList" class="comment-card mat-
elevation-z2">
    <mat-card-title>
        <div class="comment-card-title">
            <div>
                {{comment.commentedBy}}
            </div>
            <div *ngIf="appUser?.isAdmin">
                <button mat-icon-button matTooltip="Delete comment"
matTooltipPosition="before" color="accent"
                (click)="deleteComment(comment.commentId)">
                    <mat-icon>delete</mat-icon>
                </button>
            </div>
        </div>
    </mat-card-title>

    <mat-card-subtitle>{{comment.commentDate | date:'medium'}}</mat-
card-subtitle>
    <mat-card-content>
        <p>{{comment.content}}</p>
    </mat-card-content>
</mat-card>

```

The feature to post a comment on the blog is available to logged-in users only. If the user is not logged in, we will display a “Login with Google” link asking the user to login with the Google account to post comments.

We are using a template-driven form to capture the user comments. This form will invoke the `onCommentPost` method on successful submission. The form will have the following three fields

- Name – This is a required field and used to capture the name of the person posting the comment.
- Email – This is a required field and used to capture the name of the person posting the comment.
- Comment – This is a required field and used to capture the comment text.

We will also display the comments posted on the blog in a card layout using a `<mat-card>` element. We will display the name of the person who posted the comment, the date of the comment and the comment text. The list of comments will be displayed just below the comment form. If the user is an admin user, we will display a delete icon on each comment which will allow the admin to delete a comment.

Open the `src/app/components/comments/comments.component.scss` file and put the following code inside it.

```
a:not([href]):not([tabindex]) {
  text-decoration: underline;
  cursor: pointer;
  color: #1565C0;
}
.comment-card-title{
  display: flex;
  justify-content: space-between;
}

.comment-card {
  margin: 10px 0 15px 0;
}

.full-width {
  width: 100%;
}
```

Now we will add the `CommentsComponent` to the `BlogComponent`. Open `src/app/components/blog/blog.component.html` and add the following code at the

end of the file. This code snippet should be added just above the line where you have added the ScrollerComponent.

```
</mat-divider></mat-divider>  
<app-comments [blogId]="postId"></app-comments>
```

Update the BlogCardComponent

We will implement the feature of deleting all the comments related to a blog when a particular blog is deleted. Open `src/app/components/blog-card/blog-card.component.ts` and add the import definition for the `CommentService`. We will inject the service in the constructor. Refer to the code snippet shown below.

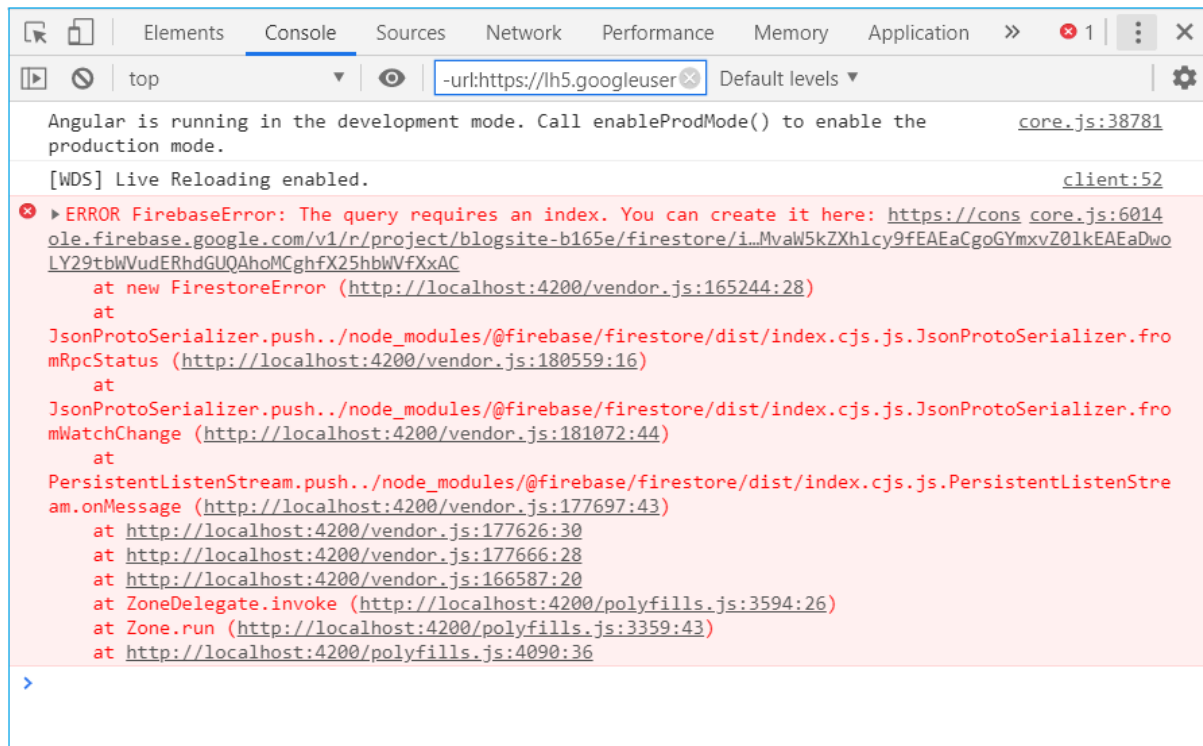
```
import { CommentService } from 'src/app/services/comment.service';  
...  
constructor(  
  // Other service injection  
  private commentService: CommentService) { }
```

We will update the delete method inside the `BlogCardComponent` class. We will call the `deleteAllCommentForBlog` method defined in the `CommentService`. Add the following line of code inside the then block of `delete` method just before invoking the `showSnackBar` method. This will ensure that while deleting a blog, all the comments associated with that blog will also be deleted.

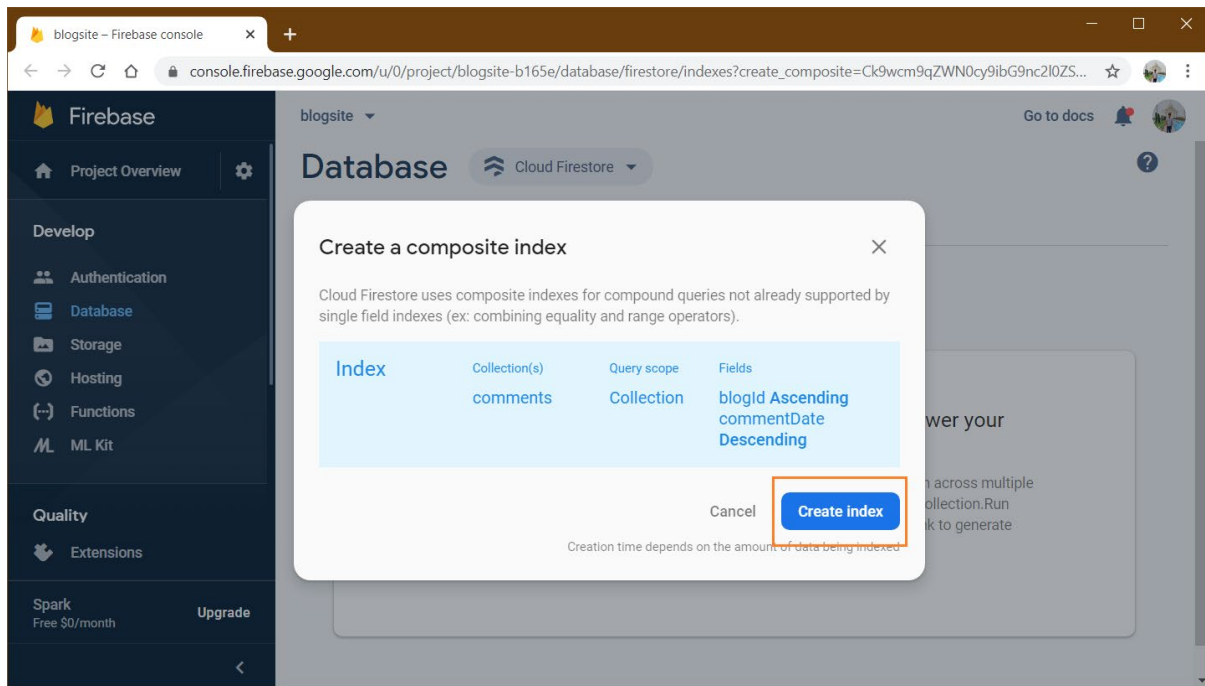
```
this.commentService.deleteAllCommentForBlog(postId);
```


Creating an index on the Firestore database

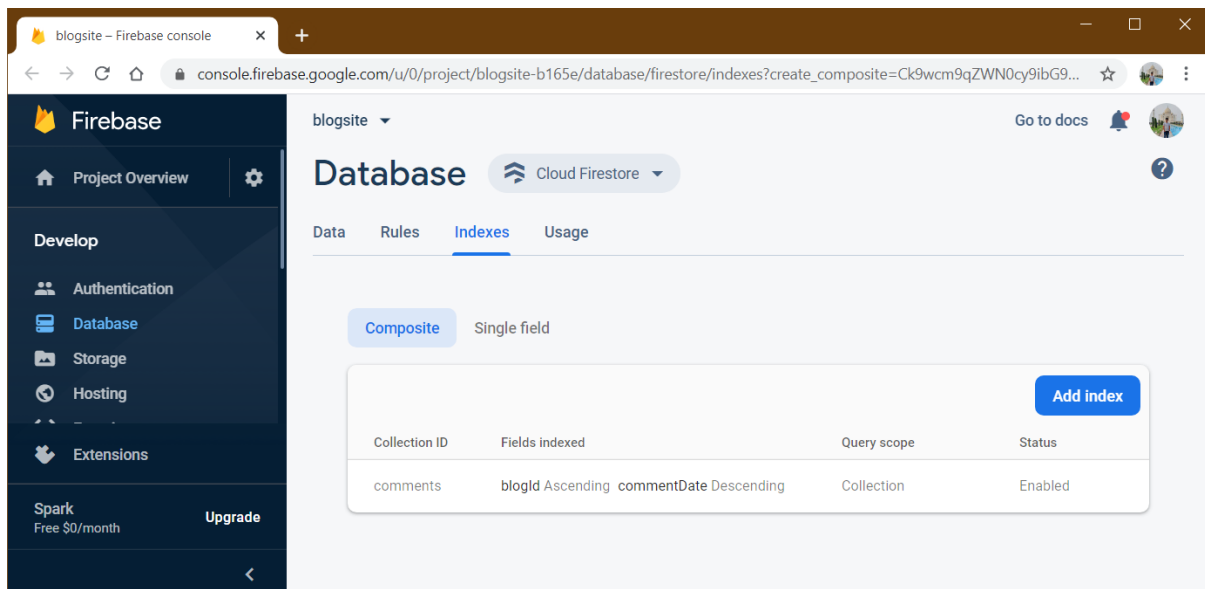
If you open the browser at this point and navigate to the blog details page, you will get an error in the browser console. The error says “ERROR FirebaseError: The query requires an index”. Refer to the image below for the reference.



Since we are using a where clause with the equality operator inside the `getAllCommentsForBlog` method, we need to create an index on our database. This index is required for our query to work. You can observe that the error message is also providing a URL to create the index. Click on the URL and you will be navigated to the firebase console. You can see the screen as shown below.

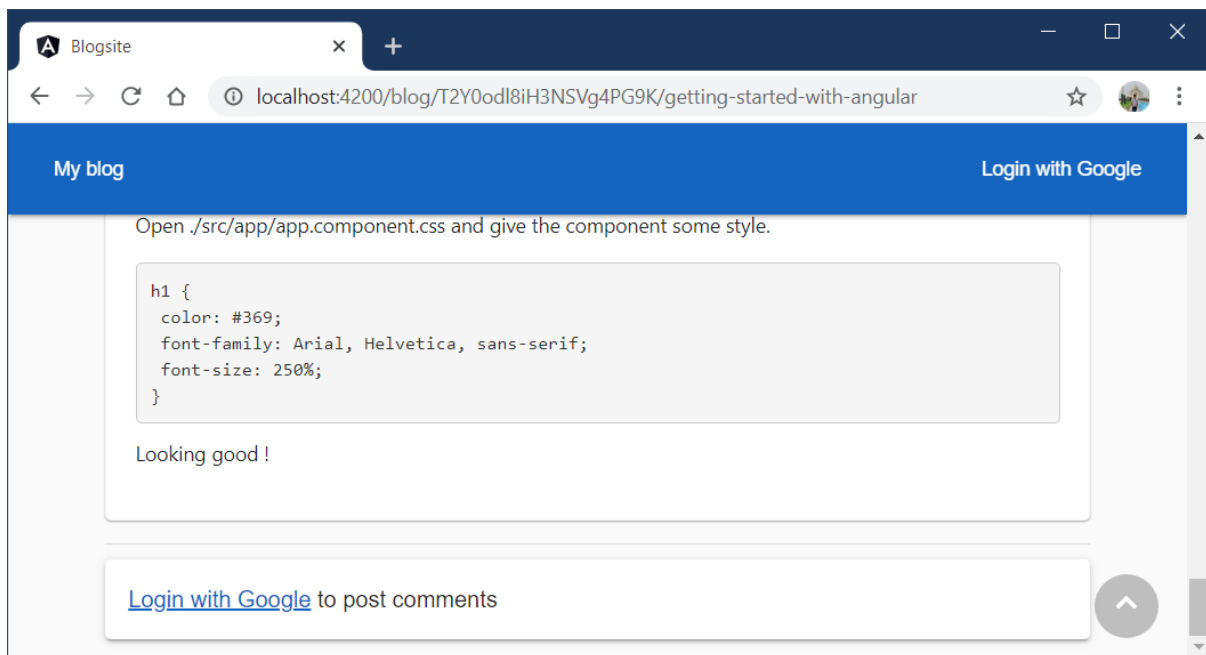


Here you can see a “Create a composite index” popup box on the screen. The required configuration for the index is already set up. Click on the “Create index” button to create the index. The index will take a few minutes to build. Once the index is created successfully, you can see the status of the index as “Enabled”. Refer to the image shown below.

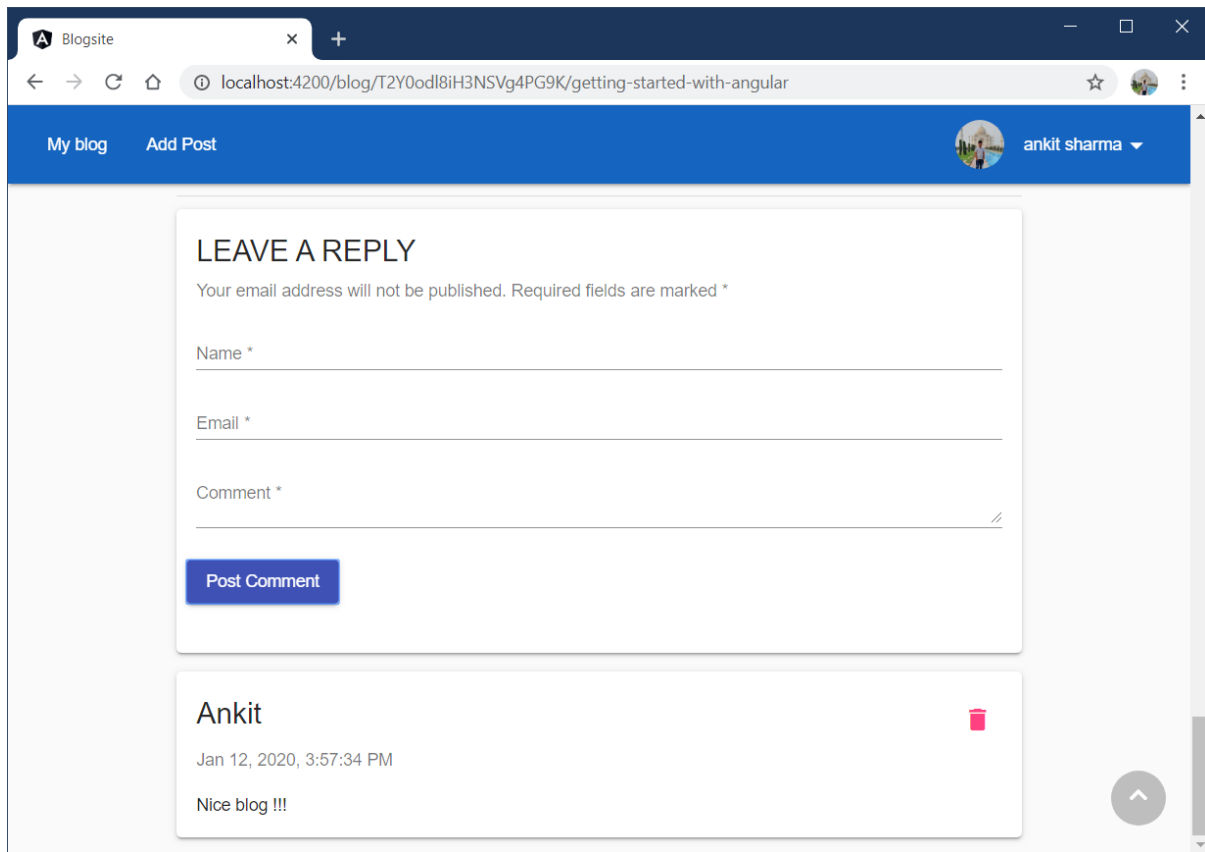


Checkpoint 9

Open the browser and logout from the application if you are already logged in. Navigate to any blog page. Scroll to the bottom of the page. You can see the “scroll to top” button on the right side of the page. If you click on this button, the page will scroll up to the top with a smooth transition effect. Refer to the image shown below.



You can also see a message which will ask you to log in with Google to start posting comments. Log in with your Google account. You will see a form for posting the comments. Fill out the details and click on the “Post Comment” button. The comment will be posted and displayed in a card just below the form. Refer to the image shown below.



The screenshot shows a web browser window with the address bar displaying 'localhost:4200/blog/T2Y0odl8iH3NSVg4PG9K/getting-started-with-angular'. The browser's address bar shows the URL 'localhost:4200/blog/T2Y0odl8iH3NSVg4PG9K/getting-started-with-angular'. The page has a blue header with 'My blog' and 'Add Post' links, and a user profile for 'ankit sharma'. Below the header is a 'LEAVE A REPLY' form with fields for Name, Email, and Comment, and a 'Post Comment' button. Below the form is a comment card for 'Ankit' posted on 'Jan 12, 2020, 3:57:34 PM' with the text 'Nice blog !!!'. A red trash icon is visible in the top-right corner of the comment card, indicating a delete option for admin users. A scroll-to-top button is located at the bottom right of the page.

The comment card will display the name of the person who posted the comment along with the date and time of posting the comment. If you are logged in as an admin user, then you can also see the “Delete comment” button on the top-right corner of the comment card.

Adding the share option for the blog

We will add the feature of sharing the blog. We will provide the option of sharing a blog via social media and email. We will be using the ngx-sharebuttons library to implement the sharing options. This is an open-source library which provides us an out of the box solution for adding share buttons and icons for various social media and messaging platforms.

Install ngx-sharebuttons

Execute the command shown below to install the ngx-share packages.

```
npm i -S @ngx-share/core @ngx-share/button @ngx-share/buttons  
@angular/cdk
```

We will also install the icon packs using the command shown below.

```
npm i -S @fortawesome/fontawesome-svg-core @fortawesome/angular-  
fontawesome @fortawesome/free-solid-svg-icons @fortawesome/free-brands-  
svg-icons
```

Import share buttons theme in the global style in the app/src/style.scss file.

```
@import '~@ngx-share/button/themes/circles/circles-dark-theme';
```

Import the ShareButtonsModule, ShareButtonsConfig, and HttpClientModule into src/app/app.module.ts file. We will create a custom configuration for the ShareButtonsModule. Refer to the code snippet shown below.

```
import { ShareButtonsConfig, ShareModule } from '@ngx-share/core';  
import { FontAwesomeModule } from '@fortawesome/angular-fontawesome';  
import { HttpClientModule } from '@angular/common/http';  
  
const customConfig: ShareButtonsConfig = {  
  twitterAccount: 'ankitsharma_007'  
};  
  
@NgModule({  
  ...  
  imports: [  
    ...  
    HttpClientModule,  
    FontAwesomeModule,  
    ShareModule.withConfig(customConfig),  
  ],  
})
```

```
})
```

In the custom config, we will set the `twitterAccount` name of the blog Author. This will ensure that, whenever we share the blog on Twitter, this account will be tagged in the tweet.

Create the social-share component

Execute the command shown below to create the social-share component.

```
ng g c components\social-share
```

Open the `src/app/components/social-share/social-share.component.html` file and put the following code into it.

```
<p><strong>Found this article helpful!!! Share this with your  
Friends</strong></p>  
  
<button mat-fab shareButton="facebook"  
[style.backgroundColor]="share.prop.facebook.color">  
  <fa-icon [icon]="share.prop.facebook.icon" size="lg"></fa-icon>  
</button>  
<button mat-fab shareButton="twitter"  
[style.backgroundColor]="share.prop.twitter.color">  
  <fa-icon [icon]="share.prop.twitter.icon" size="lg"></fa-icon>  
</button>  
<button mat-fab shareButton="linkedin"  
[style.backgroundColor]="share.prop.linkedin.color">  
  <fa-icon [icon]="share.prop.linkedin.icon" size="lg"></fa-icon>  
</button>  
<button mat-fab shareButton="reddit"  
[style.backgroundColor]="share.prop.reddit.color">  
  <fa-icon [icon]="share.prop.reddit.icon" size="lg"></fa-icon>  
</button>  
<button mat-fab shareButton="whatsapp"  
[style.backgroundColor]="share.prop.whatsapp.color">  
  <fa-icon [icon]="share.prop.whatsapp.icon" size="lg"></fa-icon>  
</button>  
<button mat-fab shareButton="telegram"  
[style.backgroundColor]="share.prop.telegram.color">  
  <fa-icon [icon]="share.prop.telegram.icon" size="lg"></fa-icon>  
</button>
```

```
<button mat-fab shareButton="print"
[style.backgroundColor]="share.prop.print.color">
  <fa-icon [icon]="share.prop.print.icon" size="lg"></fa-icon>
</button>
<button mat-fab shareButton="email"
[style.backgroundColor]="share.prop.email.color">
  <fa-icon [icon]="share.prop.email.icon" size="lg"></fa-icon>
</button>
```

We will add the `SocialShareComponent` in the `BlogComponent`. Open the `src/app/components/blog/blog.component.html` file and add the following line in it. This code snippet should be added just above the `<mat-divider>` tag.

```
<app-social-share></app-social-share>
```

Configure the icon pack

The `ngx-sharebuttons` library internally uses the `FontAwesome/angular-fontawesome` library for its icon packs. Therefore, to use the icons for our share buttons we need to configure the `fontawesome` icon packs in the `SocialShareComponent`.

Create a new file inside called `icons.ts` inside the `src` folder. Open `src/icons.ts` file and put the following code inside it.

```
import { faTelegramPlane } from '@fortawesome/free-brands-svg-
icons/faTelegramPlane';
import { faFacebookF } from '@fortawesome/free-brands-svg-
icons/faFacebookF';
import { faTwitter } from '@fortawesome/free-brands-svg-
icons/faTwitter';
import { faRedditAlien } from '@fortawesome/free-brands-svg-
icons/faRedditAlien';
import { faLinkedinIn } from '@fortawesome/free-brands-svg-
icons/faLinkedinIn';
import { faWhatsapp } from '@fortawesome/free-brands-svg-
icons/faWhatsapp';
import { faPrint } from '@fortawesome/free-solid-svg-icons/faPrint';
import { faEnvelope } from '@fortawesome/free-solid-svg-
icons/faEnvelope';

export const iconpack = [
```

```
faFacebookF, faTwitter, faLinkedinIn, faRedditAlien,  
faTelegramPlane, faWhatsapp, faEnvelope, faPrint  
];
```

Here we are importing all the icons which we are going to use in our application from the fortawesome library. We will then export the icon pack to make it available to be used in the component.

Open the `src/app/components/social-share/social-share.component.ts` file and add the following import statement at the top.

```
import { FaIconLibrary } from '@fortawesome/angular-fontawesome';  
import { iconpack } from 'src/icons';  
import { ShareService } from '@ngx-share/core';
```

Update the constructor of the SocialShareComponent class as shown below.

```
constructor(library: FaIconLibrary, public share: ShareService) {  
  library.addIcons(...iconpack);  
}
```

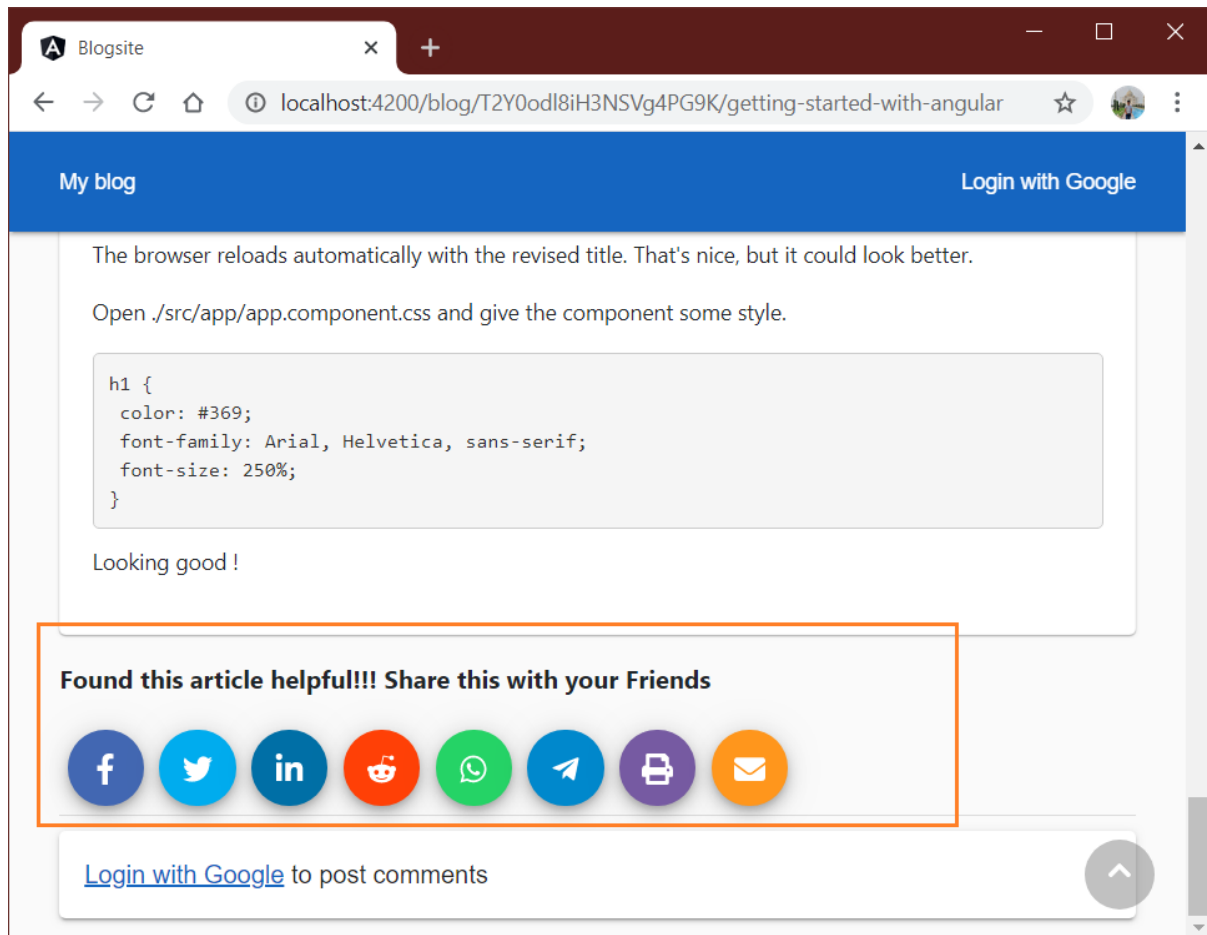
Open `src/app/components/social-share/social-share.component.scss` and add the following style definition.

```
button{  
  margin: 5px;  
}
```

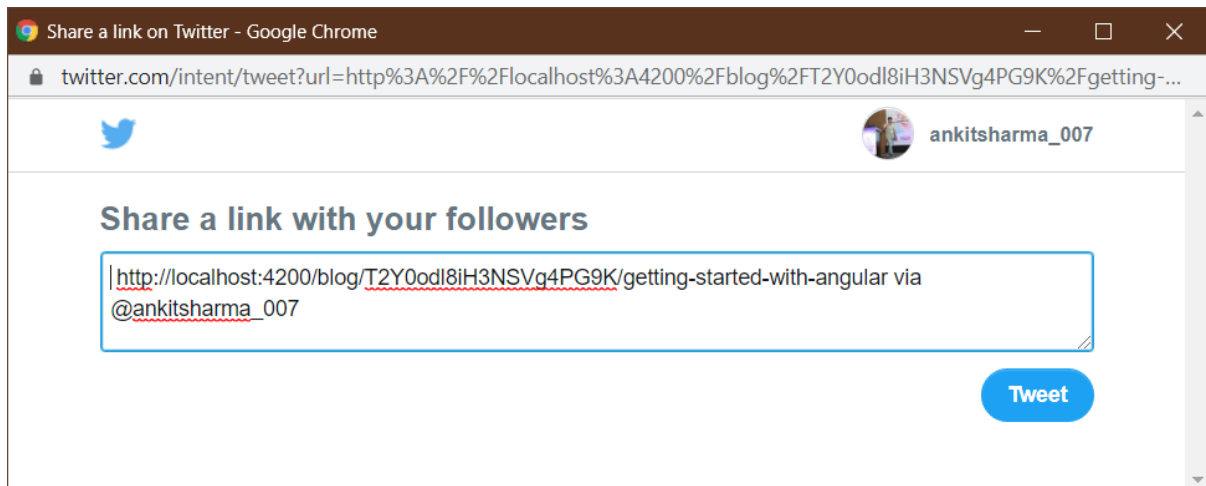

Checkpoint 10

Open the browser. Navigate to any blog page and scroll to the bottom. You will see a list of share buttons displayed there. You will get all the share options that we have configured.

Refer to the image shown below.



If you click on any of the share buttons, the corresponding application page will open asking you to log in. Upon successful login, you will get the share link of the blog. Refer to the image shown below to see the Twitter share option in action. You can observe that it is also tagging the author's twitter handle as "via @ankitsharma_007". If you click on the "Tweet" button, this tweet will be shared on your Twitter timeline. The tagging the author feature is available only for Twitter.



Deploy the app on Firebase

The final step is to deploy the app on Firebase. We will follow the steps mentioned below.

Step 1: Install firebase CLI tools via npm. Run the command as shown below.

```
npm install -g firebase-tools
```

Step 2: Run the following command to build the app in the production configuration.

```
ng build --prod
```

The 'prod' option will set the build configuration to the production target. The production target is set up in the workspace configuration such that all builds make use of bundling, limited tree-shaking, and also limited dead code elimination.

Step 3: Open a command prompt window inside the /blogsite/dist folder. And run the following command to login into the firebase.

```
firebase login
```

It will open a browser window and ask you to log in to Firebase. Login using your Google account. Upon successful login navigate back to your CLI.

Step 4: Execute the following command to initialize the app

```
firebase init
```

This command will initialize a firebase project. You will be asked a set of questions. Answer them as shown below: -

- Are you ready to proceed? – Y
- Which Firebase CLI features do you want to set up for this folder? – select Hosting
- Please select an option - use an existing project.
- Select a default Firebase project for this directory: Select your project name from the list.
- What do you want to use as your public directory? – blogsite
- Configure as a single-page app (rewrite all urls to /index.html)? – y
- File blogsite/index.html already exists. Overwrite? – N

You will get a “Firebase initialization complete!” message.

Step 4: Deploy on Firebase. Run the following command to deploy your application on Firebase.

```
firebase deploy
```

This command will deploy your angular application on Firebase and upon success, it will give you a hosting URL. Navigate to the hosting URL to see your deployed app in action. Refer to the image shown below for reference.

```

C:\ Select C:\Windows\System32\cmd.exe
E:\Projects\Ebook\blogsite

Before we get started, keep in mind:

  * You are initializing in an existing Firebase project directory

? Are you ready to proceed? Yes
? Which Firebase CLI features do you want to set up for this folder? Press Space to select features, then Enter to confirm your choices. Hosting: Configure and deploy Firebase Hosting sites

=== Project Setup

First, let's associate this project directory with a Firebase project.
You can create multiple project aliases by running firebase use --add,
but for now we'll just set up a default project.

? Please select an option: Use an existing project
? Select a default Firebase project for this directory: blogsite-b165e (blogsite)
i Using project blogsite-b165e (blogsite)

=== Hosting Setup

Your public directory is the folder (relative to your project directory) that
will contain Hosting assets to be uploaded with firebase deploy. If you
have a build process for your assets, use your build's output directory.

? What do you want to use as your public directory? blogsite
? Configure as a single-page app (rewrite all urls to /index.html)? Yes
+ Wrote blogsite/index.html

i Writing configuration info to firebase.json...
i Writing project information to .firebaserc...

+ Firebase initialization complete!

E:\Projects\Ebook\blogsite\dist>firebase deploy

=== Deploying to 'blogsite-b165e'...

i deploying hosting
i hosting[blogsite-b165e]: beginning deploy...
i hosting[blogsite-b165e]: found 1 files in blogsite
+ hosting[blogsite-b165e]: file upload complete
i hosting[blogsite-b165e]: finalizing version...
+ hosting[blogsite-b165e]: version finalized
i hosting[blogsite-b165e]: releasing new version...
+ hosting[blogsite-b165e]: release complete

+ Deploy complete!

Project Console: https://console.firebase.google.com/project/blogsite-b165e/overview
Hosting URL: https://blogsite-b165e.firebaseio.com

E:\Projects\Ebook\blogsite\dist>

```

You can also find the hosting URL on the firebase dashboard. Navigate to the "Project Overview" page of your Firebase project. Select "Hosting" under the "Develop" menu from the list on the left. You can see the domain names for your web app in the panel on the right.

This completes our application. We learned how to create a simple blogging application using Angular on the frontend and cloud Firestore as a database.

References and Useful Links

- <https://firebase.google.com/>
- <https://material.angular.io/>
- <https://cli.angular.io/>
- <https://ckeditor.com/>
- <https://fontawesome.com/>
- <https://getbootstrap.com/>
- <https://www.typescriptlang.org/docs/home.html>
- <https://angular.io/start>
- <https://blog.angular-university.io/tag/angular-for-beginners/>
- <https://www.c-sharpcorner.com/technologies/angularjs>
- <https://www.npmjs.com/package/ngx-pagination>
- <https://www.npmjs.com/package/@ngx-share/button>
- <https://github.com/AnkitSharma-007/blogging-app-with-Angular-CloudFirestore>
- <https://blogsite-30c69.firebaseio.com/>

Personal blog

You can read articles on Angular on my blog at <https://ankitsharmablogs.com/>

Connect with me

You can connect with me via social channels and GitHub

- LinkedIn – <https://www.linkedin.com/in/ankitsharma-007/>
- Twitter – https://twitter.com/ankitsharma_007
- GitHub - <https://github.com/AnkitSharma-007>



Download



Download



Download



Download