

Uttam Agarwal

05-Sep-2018

Hands-On Full Stack Development with Angular 5 and Firebase

Build real-time, serverless, and progressive web
applications with Angular and Firebase



Packt

Hands-On Full Stack Development with Angular 5 and Firebase

Build real-time, serverless, and progressive web applications with Angular and Firebase

Uttam Agarwal

Packt

BIRMINGHAM - MUMBAI

Hands-On Full Stack Development with Angular 5 and Firebase

Copyright © 2018 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Kunal Chaudhari

Acquisition Editor: Nigel Fernandes

Content Development Editor: Arun Nadar

Technical Editors: Prashant Mishra and Prajakta Mhatre

Copy Editor: Safis Editing

Project Coordinator: Sheejal Shah

Proofreader: Safis Editing

Indexer: Aishwarya Gangawane

Production Coordinator: Deepika Naik

First published: **February 2018**

Production reference: 1210218

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78829-873-5

www.packtpub.com



mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Uttam Agarwal at present works at VMware. He has more than 10 years of experience and has a wide range of knowledge of various technologies. He runs his own website, which is called CodingChum. He is an active open source contributor and regularly maintains his GitHub repository. He loves blogging and creating new products. You can check out his latest application—Heartbeat—on Google Play.

I would like to acknowledge my parents and family, whose love and support have always encouraged me to try harder in my life. Special thanks to my wife, Shilpa Agarwal. Her constant love and support in my tough times has helped me to ultimately complete this book.

Finally, I want to thank all the reviewers of this book, they helped me to improve its content.

About the reviewers

Jonathan Irvin, as a full-stack developer, creates complex and responsive websites, web apps, and mobile apps in order to keep businesses continuously thriving. He analyzes the pain points of businesses, and alleviates them using the latest and greatest technologies. Jonathan is a freelancer who has worked with a wide range of technologies, such as React, Angular, Firebase, Node.js, PHP, Laravel, Ionic, MySQL, and MongoDB. You can contact him on Twitter at [Jonathan_Irvin](#).

Nishant Shreshth is a dual degree (B. Tech. and M. Tech.) graduate in Electrical Engineering from IIT Bombay. He joined Amadeus Software Labs after college as a backend developer and worked on airline fares and pricing modules. He then moved to VMware, where he is currently employed, and works as a full-stack developer.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

[Title Page](#)

[Copyright and Credits](#)

[Hands-On Full Stack Development with Angular 5 and Firebase](#)

[Packt Upsell](#)

[Why subscribe?](#)

[PacktPub.com](#)

[Contributors](#)

[About the author](#)

[About the reviewers](#)

[Packt is searching for authors like you](#)

[Preface](#)

[Who this book is for](#)

[What this book covers](#)

[To get the most out of this book](#)

[Download the example code files](#)

[Conventions used](#)

[Get in touch](#)

[Reviews](#)

[1. Organizing Your Angular Project Structure](#)

[Creating a project outline](#)

[Project structure](#)

[Overview of package.json](#)

[Setting up Firebase](#)

[Setting up an account for Firebase](#)

[Angular terminologies](#)

[The application's project structure](#)

[App module](#)

[App routing](#)

[Authentication module](#)

[Authentication routing](#)

[Login component](#)

[Login component HTML](#)

[Login component scss](#)

[Login component spec](#)

[Authentication service](#)

[Date pipe](#)

[Common principles](#)

[Coding standard guidelines](#)

[Summary](#)

2. Creating a Signup Component

[Enabling authentication in Firebase](#)

[Introduction to angularfire2](#)

[Creating the authentication module](#)

[Creating services](#)

[Authentication service](#)

[User service](#)

[Defining domain model](#)

[Creating signup templates](#)

[FormModule](#)

[Bootstrap elements](#)

[Angular data binding](#)

[Error handling](#)

[Firebase errors](#)

[Creating a customized alert dialog](#)

[Creating a signup component](#)

[Summary](#)

3. Creating a Login Component

Adding login functionalities to existing services

Authentication service

User service

Reusing the domain model

Creating a login template

Error handling for login

Firebase error

Creating a login component

Resetting the password

Adding modal template

Adding the onReset() functionality

Editing the password-reset template in Firebase

Summary

4. Routing and Navigation between Components

Enabling routes in an app component

Creating a routing module for authentication

Exploring more routing techniques

Adding authentication guards

Firebase session lifecycle

Our project structure as of now

Summary

5. Creating a User Profile Page

Introduction to RxJS

Passing data between module components

Introduction to SASS

Creating a user profile component

Enhancing services for update operation

Creating an edit dialog component

Firebase session for the update operation

Summary

6. Creating a User's Friend List

Creating user's friend template

Creating the friend's service

Creating a Firebase node in our database

Implementing the Friend model class

Implementing the friend's service

Creating a friend's component

Creating our first date pipe

Summary

7. Exploring Firebase Storage

Introducing Firebase storage

Configuring Firebase storage

Uploading the profile picture

Downloading friends images

Deleting the profile images

Handling errors in Firebase storage

Summary

8. Creating a Chat Component

[Creating a chat module](#)

[Creating a color variable](#)

[Creating a chat component](#)

[Creating a chat message list component](#)

[Creating a mixin for the message view](#)

[Creating a chat message component](#)

[Creating a chat message form component](#)

[Summary](#)

9. Connecting Chat Components with Firebase Database

Passing data using route parameters

Adding a route parameter ID

Linking a route to the parameter

Reading the parameter

Passing friend data to different chat components

Designing a Firebase database for chat

Creating a messaging service

Integrating our service to chat components

Summary

10. Unit Testing Our Application

[Introduction to Angular testing](#)

[Unit testing an Angular component](#)

[Unit testing an Angular service](#)

[Unit testing Angular pipe](#)

[Code coverage](#)

[Summary](#)

11. Debugging Techniques

Debugging Angular

Installing [Augury](#)

Using Augury's features

Component tree

Router tree

NgModules

Debugging a web application

HTML DOM

Layout preview

Debugging TypeScript

Viewing and searching a source file

Putting in breakpoints and watching live values

Adding code in the console window

Debugging CSS

Exploring the styles panel

Discovering and modifying styles

Network debugging

Summary

12. Firebase Security and Hosting

[Introducing Firebase security](#)

[Adding security rules for users](#)

[Adding security rules for chat messages](#)

[Indexing users and friends](#)

[Setting up multiple environments](#)

[Hosting the friends app in Firebase](#)

[Summary](#)

13. Growing Our Application Using Firebase

Introduction to Firebase cloud messaging

Adding FCM to our application

Google data analytics

Learning about Google adsense

Summary

14. Transforming Our App into a PWA

[Introduction to PWA](#)

[Introduction to service worker](#)

[Adding our application to phone home screens](#)

[Enabling offline mode](#)

[Compliance testing using Lighthouse](#)

[Summary](#)

Other Books You May Enjoy

[Leave a review - let other readers know what you think](#)

Preface

Hands-On Full Stack Development with Angular 5 and Firebase will provide you with the practical knowledge you need to develop web applications. In this book, we have introduced all the major tools and technologies required to write a complete social media application. The book is written in such a fashion that you will build the application from scratch, and as you progress through the book, you will learn about the major concepts in web development. As part of our development process, this book religiously follows common software principles and coding standards. I have also introduced unit testing our Angular component, service, and pipe. These practices in software development make us better developers.

As a part of this book, we will build web application from development to production with Angular as the frontend and Firebase as the backend. Firebase is completely scalable and real time, providing all the tools needed to develop rich, collaborative applications. With Firebase, it's easy to build and prototype any business application without creating complex backend services.

You will use Angular framework to build client-side applications using HTML and CSS. Angular provides advanced features that modularize the client-side code into HTML, CSS, components, and services. As a part of our development, we will also integrate our application with other commonly used libraries, such as RxJS.

So, stay tuned for a wonderful journey.

Who this book is for

This book is for JavaScript developers who have some previous knowledge of the Angular framework and want to start developing real-time applications with Angular and Firebase. This book is quite handy for any small scale start-up who want their business or idea to be online, as this book provide a practical tips to develop applications faster. This book is also very well suited for college students who want to build a complete web application without much investment. If you are looking for a more practical and less theory-based approach to learn major web application concepts, then this book is for you.

What this book covers

[Chapter 1](#), *Organizing Your Angular Project Structure*, exposes you to the Angular project structure. You will create an Angular project using the Angular CLI command and go through all the important libraries in Angular projects.

[Chapter 2](#), *Creating a Signup Component*, covers how to enable Firebase authentication and create a signup component, template, and service. You also learn about the AngularFire2 library.

[Chapter 3](#), *Creating a Login Component*, teaches you how to create a login component and template. You will also perform a reset password operation.

[Chapter 4](#), *Routing and Navigation between Components*, helps you to enable routes for modules and create a navigation bar to navigate between components.

[Chapter 5](#), *Creating a User Profile Page*, focuses on the RxJs library and pass data between component modules. You will also create user profile page with edit operation.

[Chapter 6](#), *Creating a User's Friend List*, teaches you how to create a friend list page with pagination. You will also create the friends service and custom Angular date pipe.

[Chapter 7](#), *Exploring Firebase Storage*, discusses Firebase storage and configure storage for your application. You will also upload and download images from Firebase storage into your application.

[Chapter 8](#), *Creating a Chat Component*, helps you create a chat module with sub-components. We also learn more about SCSS variables and mixin concepts.

[Chapter 9](#), *Connecting Chat Components with Firebase Database*, covers how to integrate your chat component with Firebase database. You will also learn about passing data using route parameters.

[Chapter 10](#), *Unit Testing Our Application*, teaches you about Angular testing. You

will write unit test cases for our component, service, and pipe, and learn about code coverage.

[Chapter 11](#), *Debugging Techniques*, covers different aspects of debugging techniques. As a part of this chapter, we will cover debugging for Angular, the web, TypeScript, CSS, and networks.

[Chapter 12](#), *Firebase Security and Hosting*, teaches you about Firebase security and explains how to add security rules for users and chat messages. You also learn how to create multiple environments and host our friends application.

[Chapter 13](#), *Growing Our Application Using Firebase*, covers Firebase cloud messaging. You will also learn about Google Analytics and Ads.

[Chapter 14](#), *Transforming Our App into a PWA*, covers PWA features and shows how to make your application PWA-compliant. You will also learn about service workers.

To get the most out of this book

To get the most out of this book, you should have some experience of JavaScript development with HTML and CSS. As you progress through the chapters, there are links to all the important tools, editors, or frameworks required to develop Angular application.

Download the example code files

You can download the example code files for this book from your account at www.packtpub.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Hands-On-Full-Stack-Development-with-Angular-5-and-Firebase>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`codeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
| public addUser(user: User): void {  
|     this.firebaseio.database().ref(`$${USERS_CHILD}/${user.uid}`).set(user);  
| }
```

Any command-line input or output is written as follows:

```
| $ cd <your directory>\friends\src\app
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Click on Authentication."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

Organizing Your Angular Project Structure

In this chapter, we will create project structure using the `@angular/cli` command. We will download all the necessary libraries using **npm (node package manager)**. Then, we will start the project to take a look at our first Angular application running on the browser. We will go through all the important files that are required to develop a high-quality Angular application.



npm is a package manager for JavaScript. It also helps in installing packages required to develop an application.

We will also run through setting up a testing framework setup and write test cases for some of the important components of our social application. Our aim is to develop a production-ready application through testing, development, and analytics.

At the end, we will go through the recommended guidelines for coding styles, as this is the most ignored part in development. We feel this is required when we develop any application in the team so that common terms and terminologies are followed across the development team.

The following topics will be covered in this chapter:

- Creating a project outline
- Project structure
- Setting up Firebase
- Angular terminologies
- The application project structure
- Common principles
- Coding standard guidelines

Creating a project outline

Angular CLI (command-line interface) makes it easy to create a project. It also helps in creating component, routes, services, and pipes with a simple-to-use command. We will use Angular CLI to create a sample project outline. This provides all the necessary files to start building your application.

We require the following four steps to run our first Angular application without any coding and get our first welcome page. We also need npm to install important libraries; you can download npm from <https://nodejs.org/en/download/>:

1. Let's install Angular CLI using npm:

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



Please note that the `-g` flag installs the angular CLI globally so that it can be accessed in any project.

2. Create the friends project structure using the `ng new` command. Since we are using SASS to create our style sheets, we will provide the `--style=sass` option as well, and this configures SASS in our application. Take a look at the following command:

```
| $ng new friends --style=sass
```



Don't forget to give your project name at the end, otherwise it will create a default project name.

3. Go to the newly created `friends` folder and execute `npm install`. This installs all the packages required to build our application, and it also creates the `node_modules` folder. Take a look at the following command:

```
| $npm install
```

4. Finally, deploy the newly created friends project using `npm start` and take a look at your first Angular application running in a browser. Refer to the

following command. The default port is 4200, and you can type `http://localhost:4200` to see your sample application in a browser:

```
| $npm start
```

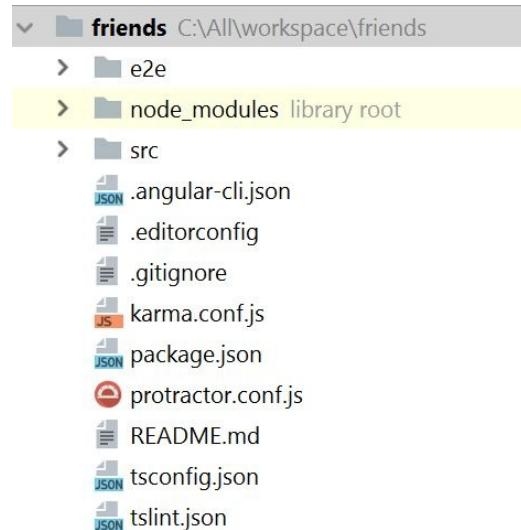
Congratulations on your first Angular application!

Project structure

The next step is to map the newly created project to an editor. We use WebStorm as our editor, which is a paid version. You can use Visual Studio Code or Sublime as your editor, which are free versions, and you can download them from the following URLs:

- Visual Studio Code: <https://code.visualstudio.com/download>
- Sublime Text: <https://www.sublimetext.com/download>

We mapped the project to our WebStorm editor and installed the dependent libraries using `npm install`, and it creates a `node_modules` folder. This folder contains all the dependent libraries. Our project structures in the editor looks like the following screenshot:



Overview of package.json

The `package.json` file specifies the starter packages for running the application. We can also add packages in this file as our application evolves.

The `package.json` file contains the following two sets of dependencies:

- `dependencies`: The packages in "dependencies" contain all essential libraries for running the Angular application:

```
"dependencies": {  
    "@angular/animations": "^5.2.0",  
    "@angular/common": "^5.2.0",  
    "@angular/compiler": "^5.2.0",  
    "@angular/core": "^5.2.0",  
    "@angular/forms": "^5.2.0",  
    "@angular/http": "^5.2.0",  
    "@angular/platform-browser": "^5.2.0",  
    "@angular/platform-browser-dynamic": "^5.2.0",  
    "@angular/router": "^5.2.0",  
    "core-js": "^2.4.1",  
    "rxjs": "^5.5.6",  
    "zone.js": "^0.8.19"  
},
```

Dependencies consist of the following libraries. We have explained only the important libraries in the following list:

- `@angular/common`: It provides commonly used functionalities, such as pipes, services, and directives.
- `@angular/compiler`: It understands templates and converts them into a format that the browser understands so that our application can run and render. We don't interact directly with this library.
- `@angular/core`: It provides all common metadata, such as component, directive, dependency injection, and component life cycle hooks.
- `@angular/forms`: It provides a basic layout for inputs. This is used in login, signup, or feedback.
- `@angular/http`: It is an Angular service that provides a utility function for HTTP **rest** calls.

- `@angular/platform-browser`: It provides the `bootstrapStatic()` method for bootstrapping applications for production builds.
 - `@angular/platform-browser-dynamic`: It is mainly used in bootstrapping during development.
 - `@angular/router`: It provides a component router for navigation between components.
 - `core-js`: It patches the global context window with essential features of ES2015 (ES6).
 - `rxjs`: It is a library for reactive programming using observables that help in writing asynchronous code for HTTP.
 - `zone.js`: It provides an execution context that persists across asynchronous tasks.
- `devDependencies`: The packages in `devDependencies` contain libraries that are mostly required during the development process. You can exclude `devDependencies` during production builds using the following `npm` command:

```
| $npm install my-application --production
```

Setting up Firebase

In our project, we use Firebase as a backend service, **a mobile and web application platform**. It provides complete suites of products integrated into one platform and makes the development process much faster with the major part handled by the platform. Its products are segregated around two major themes:

- **Develop and test your application:** These suites provide all the services required to develop a scalable application
- **Grow and engage your audience:** This is mostly required to grow our application

You can refer to the following link for more information on Firebase: <https://firebase.google.com/>.

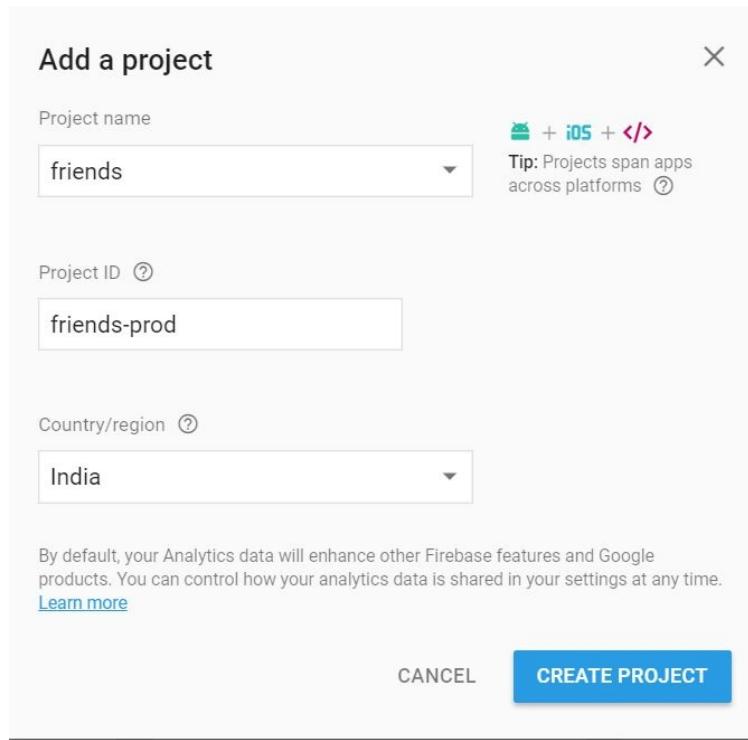
Setting up an account for Firebase

The first step in creating a [new Firebase project](#) is to create your new Google account or work with the current account.

Now, open the Firebase portal and follow four steps to start your Firebase project:

1. Click on GO TO CONSOLE on the top-right corner.
2. Click on Add Project with plus (+) sign.
3. In the pop-up window, enter Project name and Country/region. Project ID is an optional field, and this will take a default value.
4. Click on CREATE PROJECT.

The following screenshot shows the friends project:



Finally, you get your configuration details on the welcome page of Firebase and then click on Add Firebase to your web app. You can copy this configuration into `environments.ts`, as follows:

```
export const environment = {
  production: false,
  firebase: {
    apiKey: 'XXXX',
    authDomain: 'friends-4d4fa.firebaseio.com',
    databaseURL: 'https://friends-4d4fa.firebaseio.com',
    projectId: 'friends-4d4fa',
    storageBucket: '',
    messagingSenderId: '321535044959'
  }
};
```

Angular terminologies

In this section, we will discuss important terminologies in Angular. You may be familiar with most of them, but this will refresh your knowledge:

- **Module:** Angular supports modularity using modules. All Angular projects have at least one module known as `AppModule`. When we build large applications, we can divide our application into multiple feature modules with common related capabilities. We can create module using the `@NgModule` annotation.
- **Component:** Component is a controller, which has views and logic to manage view events and navigation between components. It interacts with the view through various data binding techniques. You can generate component using the following CLI command:

```
| $ng g component <component-name>
```

- **Templates:** A template represents the view of the web page and it is created using HTML tags. It also has many custom tags such as angular directives along with native HTML tags.
- **Metadata:** It assigns a behavior to any class. It is the metadata on top of the class that tells Angular the behavior of the class. For example, the component is created using the `@Component` annotation.
- **Data Binding:** It is a process by which a template interacts with the component. The data is passed to and fro using various data binding techniques. Angular supports the following three types of data binding:

- **Interpolation:** In this binding, we use two curly braces to access the property value of component members. For example, if we have `name` as a class member property in a component, then we can define `{{name}}` in the template to access the name value.
- **Property Binding:** This data binding technique helps in passing a value from a parent component to a child component. For example, if we have `name` as a class member property in a parent component and

`userName` in a child component, then we can assign a value from parent to child using `[userName] = "name"`.

- **Event Binding:** This event-driven data binding helps to pass a value from template to component. For example, we display a name from the list; when the user clicks on the list item, we pass a click value using event binding `(click)="clickName(name)"`.
- **Directives:** It is a behavior injected into the templates, which modifies the way the **DOM (document object model)** is rendered. There are basically two kinds of directive:
 - **Structural:** It alters the DOM layout by adding, removing, or replacing DOM elements. Examples for this kind of directive are `ngFor` and `ngIf`.
 - **Attribute:** It alters the appearance and behavior of the existing DOM element. The `ngModel` directive is an example of attributes directives, which change the behavior of the existing element by responding to `change` event.
- **Service:** It is a usable entity, that consumed by any Angular components and helps to separate the view logic from the business logic. We normally write HTTP-specific call for a particular module in a service, as it helps readability and maintainability of the code. Popular examples of services are logging service or data service. You can create `service` using the following CLI command:

```
| $ng g service <service-name>
```

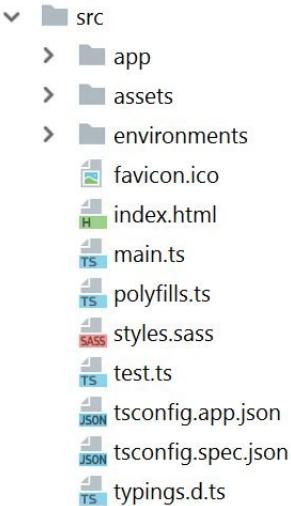
- **Pipe:** It is one of the simplest but most useful features in Angular. It provides a way to write a utility functionality, which can be reused across applications. Angular provides built-in pipe, such as date and currency. You can create `pipe` using the following CLI command:

```
| $ng g pipe <pipe-name>
```

The application's project structure

When we browse our sample friends application, we will come across the `src` folder, which contains all the application core files that have view, business logic, and navigation functionalities. A developer spends most of their time in

this folder:



```
src
├── app
├── assets
└── environments
  ├── favicon.ico
  ├── index.html
  ├── main.ts
  ├── polyfills.ts
  ├── styles.sass
  ├── test.ts
  ├── tsconfig.app.json
  ├── tsconfig.spec.json
  └── typings.d.ts
```

The main idea we followed in organizing our folder is feature module. Each similar functionality is organized into features module. In order to understand it better, we will take a look at an example of an authentication feature in our friends application and a few of the file references comes in later chapters of this book.

In our application, all the authentication-related functionalities, such as login and signup, are grouped into a module named `authentication`. This same pattern will be applied to all the features in our application. This makes our application more modular and testable. We have taken an example of an authentication feature module and explained it in more detail in the following sections.

Our authentication feature module looks the following screenshot; authentication features have login and signup functionalities, and all the components are declared in modules—it also has its own routes module for internal navigation:

```
▼ src
  ▼ app
    ▶ about
    ▼ authentication
      ▼ login
        login.component.html
        login.component.scss
        login.component.spec.ts
        login.component.ts
      ▼ signup
        signup.component.html
        signup.component.scss
        signup.component.ts
        authentication.module.ts
        authentication.routing.ts
```

App module

App module is the root module of our entire project. All Angular projects have at least one app module, which is mandatory, and it is used for bootstrapping our project to launch the application. The app module is declared with the `@NgModule` decorator. The metadata in the `NgModule` annotation are as follows:

- `imports`: In the `imports` tag, we declare all dependent feature modules. In the following code example, we declare `BrowserModule`, `AuthenticationModule`, and `AppRoutingModule` modules.
- `declarations`: In the `declarations` tag, we declare all the components for this root module. In the following example, we declare `AppComponent` in `AppModule`.
- `providers`: In the `providers` tag, we declare all the services or pipes. In the following example, we declare `AngularFireAuth` and `AngularFireDatabase`.
- `bootstrap`: In the `bootstrap` tag, we declare the root component, which is required in `index.html`.

This sample `app.module.ts` is created using the `ng new` command, as follows:

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';
import {AppComponent} from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {
```

App routing

The root routing module is used to navigate between different feature modules. We will create a route containing the navigation flow for the main application. The following example contains the sample of routes, which shows the navigation between common components in applications, such as `AboutComponent` and `PageNotFoundComponent`. The wildcard `'**'` in the path specifies that any wrong URL will be redirected to the page not found component. App routing is created with the `@NgModule` decorator and the routes are configured using `RouterModule`.

The sample `app.routing.ts` as follows:

```
import {RouterModule, Routes} from '@angular/router';
import {NgModule} from '@angular/core';
import {PageNotFoundComponent} from './notfound/not-found.component';
import {AboutComponent} from './about/about.component';

export const ROUTES: Routes = [
  {path: 'app-about', component: AboutComponent, pathMatch: 'full'},
  {path: '**', component: PageNotFoundComponent},
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      ROUTES
    )
  ],
  exports: [
    RouterModule
  ]
})
export class AppRoutingModule { }
```

Authentication module

The `authentication` module is a feature module, which contains login and signup feature set. All the features related to authentication are contained in this folder. We included `LoginComponent` and `SignupComponent` in the declarations metadata:

```
import {NgModule} from '@angular/core';
import {AuthenticationRouting} from './authentication.routing';
import {LoginComponent} from './login/login.component';
import {SignupComponent} from './signup/signup.component';
import {FormsModule} from '@angular/forms';
import {AuthenticationService} from '../services/authentication.service';

/**
 * Authentication Module
 */
@NgModule({
  imports: [
    FormsModule,
    AuthenticationRouting
  ],
  declarations: [
    LoginComponent,
    SignupComponent
  ],
  providers: [
    AuthenticationService
  ]
})
export class AuthenticationModule { }
```

Authentication routing

This routing helps in the navigation within the child component, so we create routes for login and signup components. This will evolve as we start implementing our login and a signup component in upcoming chapters.

A sample authentication.routing.ts is as follows:

```
import {NgModule} from '@angular/core';
import {RouterModule, Routes} from '@angular/router';
import {LoginComponent} from './login/login.component';
import {SignupComponent} from './signup/signup.component';

export const ROUTES: Routes = [
  {path: 'app-friends-login', component: LoginComponent},
  {path: 'app-friends-signup', component: SignupComponent}
];

/**
 * Authentication Routing Module
 */
@NgModule({
  imports: [
    RouterModule.forChild(ROUTES)
  ],
  exports: [
    RouterModule
  ]
})
export class AuthenticationRouting {
```

Login component

A component is the basic building block of an UI (user interface) in an Angular application. The `@Component` decorator is used to mark a class as a Angular component. The following example shows the Login component, which is defined using `@Component` and contains **SCSS (Sassy CSS)** and template files:

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

@Component({
  selector: 'friends-login',
  styleUrls: ['./login.component.scss'],
  templateUrl: './login.component.html'
})
export class LoginComponent {
```

Login component HTML

Login component HTML is a view and contains the form data for login. The following example shows a simple HTML with two tags; a template in Angular follows the same syntax as HTML:

```
| <h3>Login Component</h3>
| <p>This is the Login component!</p>
```

Login component scss

In our project, we use **SASS (syntactically awesome stylesheet)**, which is a preprocessor, to convert our SCSS file to a CSS file. This helps organize our CSS element in a simple and readable format. It provides many features, such as variables, nesting, mixins, and inheritance, which help in writing complex CSS in a reusable way. We will cover these features when we create our feature module. The following SCSS file is a simple example where a black variable is externalized to a variable: `$black`. This variable can be used with other CSS elements:

```
$black: #000;  
| h1.title {  
|   color: $black;  
| }
```

Login component spec

We use the **Jasmine** test framework to write our unit test cases. It is a very good habit to write test cases from the start of our development. Test cases should be part of our development process. Unfortunately, we normally write test cases after our application is in production or when we see lot of issues during production. We know writing test cases takes time during the initial phase of development, but it saves a lot of time later on. Earlier, the software industries basis for delivering a new product was *time to market*, but now the focus has shifted to *time to market with a quality product*. So, when writing test cases becomes a part of development, then we write test cases faster and deliver quality products on time. We hope this explains the importance of this process during development.

We will create a `spec` file to write unit test cases. The following is a sample code snippet, which will be explained in more detail in [Chapter 10, Unit Testing Our Application:](#)

```
| describe('LoginComponent tests', () => {  
});
```

Once our test cases are ready, we can execute the test case using `npm`:

```
| $npm test
```

Authentication service

The best way to separate UI logic from business logic in Angular is service. It provides a well-defined class to perform specific logic. In our project, we interact with a Firebase database using HTTP in service so that heavy load goes to service. In Authentication service, we perform actions such as login, signup, and sign-out.

When we clearly separate responsibility in a well-defined class, we adhere to the **SRP (single responsibility principle)**. SRP is explained in more detail in the following section. We use the `Injectable` decorator on top of service so that this service is injected in any component or service by Angular framework, and we don't have to create the object. This pattern is known as dependency injection.

A sample `authentication.service.ts` is as follows:

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

/**
 * Authentication service
 *
 */
@Injectable()
export class AuthenticationService {
```

Date pipe

Pipes help to write utility functions in an Angular application.

We will create a pipe by extending `PipeTransform` and override the `transform` method from the super class to provide our functionality. Our application's date pipe is decorated with pipe metadata. We will provide a name for our pipe using a `name` tag, which is used to transform data in a template.

A sample `friendsdate.pipe.ts` is as follows:

```
import {Pipe, PipeTransform} from '@angular/core';

/**
 * It is used to format the date
 */
@Pipe({
  name: 'friendsdate'
})
export class FriendsDatePipe implements PipeTransform {

  transform(dateInMillis: string) {
  }
}
```

The following is how you will use the pipe in a template:

```
| <div>{{<timeInMillis>: friendsdate}} </div>
```

Common principles

~~Principles help us to design our application in a better way, and these principles are not specific to any programming languages or platform. In this section, we will cover some of the most used principles in an Angular application:~~

- **Single responsibility principle:** This principle states that a class should have only one reason to change and that the responsibility should be encapsulated in the class. We apply this principle to all our component, services, or routers. This helps code maintainability and testability. For example, a component is responsible for managing the logic related to view. Other complex server logic is delegated to services. We mostly implement the utility functions in pipe.
- **Single file:** It is a good practice to define components or services per file. This makes our code easier to read, maintain, and export across module.
- **Small functions:** It is always good to write small functions with a well-defined purpose. This really makes the life of other developers—who use your code—easier, as the new developer can read and easily understand the code.
- **LIFT:** Always follow the *lift* principle so that you can *locate* code quickly, *identify* the code, keep the structure *flat*, and *try* to be dry. This ensures consistent structures across your application. This *tremendously increases* developer efficiency.
- **Folders-by-feature:** During the development of our application, we followed this principle and all the features are grouped into feature modules.
- **Shared feature:** We create a shared folder to declare all reusable components, directives, and pipes, and these are shared among other folders. SharedModules are created so that they are referenced in the entire application.

Coding standard guidelines

Coding standard are common patterns we follow while writing our application. This makes our code consistent across application. This is quite useful when multiple developers are working on the same application. A few coding standards are as follows:

- **Naming Conventions:** It is extremely important to have consistent naming conventions for our filename, class name, and so on. This is essential from the maintainability and readability point of view. This also helps in understanding the code faster, and debugging becomes easier.
 - **Filename:** The recommended pattern for a filename in Angular is `feature.type.ts`. For example, the name of the authentication component file is `authentication.component.ts`.
 - **Class name:** The class name follows the camel case and takes the name from the filename. For example, the class name for the authentication component file is `AuthenticationComponent`.
 - **Method and properties name:** We follow camel case to name our class method and properties. We don't prefix private variables with an underscore.
 - **Selector name:** The selector name in a component is hyphenated, and all characters are in lowercase. The name of our login component is shown here:

```
| selector: 'app-friends-login'
```

Pipe name: Pipes are named after their features. For example, the currency pipe name is 'currency', as follows:

```
| name: 'currency'
```

- **Constant:** The constant is the final variable that is declared using the `const` keyword, and the constant value is assigned once. Constant variables are declared in uppercase, as follows:

```
| export const USER_DETAILS_CHILD = 'user-details';
```

- **Members sequence:** All member variables are declared first, followed by the method. Private members are named after public members.

Summary

In this chapter, we created our first Angular application using the Angular CLI command. We went through all the important files in our Angular application. We learned common Angular terminologies and then went through our application project structure and internals. Finally, we covered common principles and coding guidelines, which we will follow as part of our development process.

In the next chapter, we will create our first signup page. We will apply many concepts from this chapter to create our signup component.

7-Sep-2018

Creating a Signup Component

In this chapter, we will start our journey in application development. We will build a signup page. During the process, we will also explore different features of Firebase. We will take a look at how to enable Firebase authentication in the console, which is required to interact with the authentication feature module. Firebase supports many authentication mechanisms but, in this project, we will enable Email/Password authentication. Then, we will go ahead and build a signup form template. We will add a functionality in the signup component. In this process, we will also handle error scenarios. This will make our application more robust and less error-prone.

In this chapter, we will cover the following topics:

- Enabling authentication in Firebase
- Introduction to AngularFire2
- Creating an authentication module
- Creating services
- Defining a domain model
- Creating a signup template
- Error handling
- Creating a customized alert dialog
- Creating a signup component

Enabling authentication in Firebase

The first step in our application development is to enable authentication in our Firebase portal. We have already created our Firebase project in the preceding chapter. Perform the following steps to enable Firebase authentication:

1. Open your friends project in the Firebase console.
2. Expand DEVELOP on left panel.
3. Click on Authentication.
4. Go to the SIGN-IN METHOD tab on the right panel.
5. Click on the Email/Password item and enable it.

The enabled Email/Password authentication in Firebase will look as the following screenshot:

Sign-in providers	
Provider	Status
✉ Email/Password	Enabled
📞 Phone	Disabled
GOOGLE Google	Disabled
FACEBOOK Facebook	Disabled
TWITTER Twitter	Disabled
GITHUB GitHub	Disabled
ANONYMOUS Anonymous	Disabled

This is all we need to do to enable the Email/Password mode of authentication. Firebase also supports other modes of authentication. In this book, we will implement Email/Password authentication only. You can work with other authentications as an exercise.

~~Different modes of authentications are required to target different sets of users. It's good to understand other authentication modes. Other supported modes are as follows:~~

- **Phone:** This is a simple form of authentication and doesn't require much information from the user. Just the user's mobile number is enough to authenticate the user. This mode has become popular in mobile applications.
- **Google:** It is good to enable Google authentication, as most of the targeted users have a Google account. In this mode, authentication happens via Google credentials.
- **Facebook:** This is similar to Google authentication. The only difference is that authentication happens via Facebook credentials instead of Google credentials.
- **Twitter:** This is similar to the preceding authentication, but the authentication happens via your Twitter account.
- **GitHub:** This is quite helpful for the developer community. They have a GitHub account, and you don't need to give any personal account information.
- **Anonymous:** This authentication is useful for an application where its user doesn't want to sign up. This is mainly used on an e-commerce application, where the user can browse the products using this authentication.

In all the preceding authentications, Firebase generates a unique ID called a user UID and the user is registered in a Firebase authentication with a UID, identifier, and so on. The same UID is also used to register the user in Firebase database with more information, such as email, UID, name, and number.

Introduction to angularfire2

~~angularfire2 is the official library for Angular and Firebase. We will use this library in our application. This library provides the following features:~~

- It uses the power of RxJS, Angular, and Firebase
- It provides APIs to interact with a Firebase database and authentication
- It synchronizes data in real time
- It provides APIs to interact with `AngularFirestore`

For more details, refer <https://github.com/angular/angularfire2>.

Creating the authentication module

In this section, we will create our first module using the Angular CLI command. Perform the following steps to create the authentication module:

1. Navigate to the `src` folder of your project and execute the module CLI command, as follows:

```
$ cd <your directory>\friends\src\app  
$ ng g module authentication --routing`
```

The preceding command creates the file skeleton of `authentication.module.ts` and `authentication.routing.ts`. You can follow the Angular CLI command to create other components.

2. Add an authentication module in the app module and also configure Angular- and Firebase-related components, as follows; take a look at the `app.module.ts` file:

```
import {NgModule} from '@angular/core';
import {AppComponent} from './app.component';
import {FormsModule} from '@angular/forms';
import {BrowserModule} from '@angular/platform-browser';

import {AuthenticationModule} from './authentication/authentication.module';
import {AngularFireModule} from 'angularfire2';
import {environment} from './environments/environment';
import {BrowserAnimationsModule} from '@angular/platform-browser/animations';
import {AngularFireAuth} from 'angularfire2/auth';
import {AngularFireDatabase} from 'angularfire2/database';
import {CommonModule} from '@angular/common';
import {RouterModule} from '@angular/router';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    CommonModule,
    BrowserModule,
    FormsModule,
    AngularFireModule.initializeApp(environment.firebaseio),
    BrowserAnimationsModule,
    RouterModule.forRoot([]),
    AuthenticationModule
  ]
})
```

```
  ],
  providers: [
    AngularFireAuth,
    AngularFireDatabase,
  ],
  bootstrap: [AppComponent]
})
export class AppModule {  
}
```

3. Add the placeholder in the app component template, as shown in the following `app.component.html` file:

```
|     <router-outlet></router-outlet>
```

Creating services

Services in an Angular application contain core business logic. As a part of the signup component, we created two services:

- Authentication service
- User service

Authentication service

We introduced authentication service in the preceding chapter, so we will add more methods in an authentication service. Remember the following steps while creating angular services.

1. **Writing the services:** AngularFire2 has an `AngularFireAuth` class. This class gives access to `firebase.auth.Auth`, which has the `createUserWithEmailAndPassword` API to sign up to Firebase.

```
import {Injectable} from '@angular/core';
import {AngularFireAuth} from 'angularfire2/auth';

/**
 * Authentication service
 */
@Injectable()
export class AuthenticationService {

    /**
     * Constructor
     *
     * @param {AngularFireAuth} angularFireAuth provides the
     * functionality related to authentication
     */
    constructor(private angularFireAuth: AngularFireAuth) {
    }

    public signup(email: string, password: string): Promise<any> {
        return
            this.angularFireAuth.auth.createUserWithEmailAndPassword(
                email,
                password);
    }
}
```

2. **Registering the service:** We will need to include this service in the authentication module before using the APIs. Take a look at the following details.

We will need to change `authentication.module.ts` to include authentication service. The `NgModule` metadata of the authentication module will look the following code snippet; we have added the `providers` tag to include the `AuthenticationService`.

```
| @NgModule({
|   imports: [
|     ...
|   ],
|   declarations: [
|     ...
|   ],
|   providers: [
|     AuthenticationService
|   ]
| })
| ...
```

3. **Injecting and using the service:** Once the service is registered, we will need to declare it in the constructor. The instance will be injected by the Angular framework. Finally, the signup component uses the `AuthenticationService signup()` API to authenticate a user to Firebase.

The following example shows the declaration of `AuthenticationService` in the `signup` component:

```
| constructor(private authService: AuthenticationService) {}
```

User service

In addition to registering a new user to Firebase authentication, we will also need to store more information about the user, such as the mobile, email, and so on, in our Firebase database.

The user service is used to enter user information in the Firebase database. We use `AngularFireDatabase` from `angularfire2` to set user information to the Firebase database. This class is injected in the constructor of the `UserService` class:

```
| constructor(private fireDb: AngularFireDatabase) {}
```

`AngularFireDatabase` provides an object method, which accepts the path of the data in the Firebase database; this returns `AngularFireObject` to set the data to Firebase:

```
| public addUser(user: User): void {
|   this.fireDb.object(`${USERS_CHILD}/${user.uid}`).set(user);
| }
```

The complete `user.service.ts` file as of now is as follows:

```
import {Injectable} from '@angular/core';
import {AngularFireDatabase} from 'angularfire2/database';
import {User} from './user';
import 'firebase/storage';
import {USERS_CHILD} from './database-constants';

/**
 * User service
 */
@Injectable()
export class UserService {

  /**
   * Constructor
   *
   * @param {AngularFireDatabase} fireDb provides the functionality for
   *         Firebase Database
   */
  constructor(private fireDb: AngularFireDatabase) {}

  public addUser(user: User): void {
    this.fireDb.object(`${USERS_CHILD}/${user.uid}`).set(user);
  }
}
```

As covered in the preceding section, we will need to register the service in `authentication.module.ts`:

```
| @NgModule({
|   imports: [
|     ...
|   ],
|   declarations: [
|     ...
|   ],
|   providers: [
|     AuthenticationService,
|     UserService
|   ]
| })
```

Defining domain model

The object model contains information about the key domain of our application. This helps to store our unstructured data in a more readable and structured way. In our application, we will introduce a lot of object models to store our domain information.

When we sign up a new user, we will store the user details in the Firebase database, and we created a user model with properties related to the user to store the user's data. All the properties are declared as member variables with type since TypeScript supports types for variables as shown in the following `user.ts`:

```
export class User {  
    email: string;  
    name: string;  
    mobile: string;  
    uid: string;  
    friendcount: number;  
    image: string;  
}
```



TypeScript is the type version of JavaScript, and it compiles into JavaScript. Programming in Typescript is much easier and faster. You can learn more about this at <https://www.typescriptlang.org/docs/home.html>.

Creating signup templates

A signup template represents the view of web page. It provides the form elements to type user inputs. It also handles user errors in the template. We will go through all tags used in the template in this section. Error handling used in the template will be covered in the next section.

FormModule

We use `<form>` from `FormsModule` to create the template for signup. `ngForm` contains the signup form data. This is required to retrieve the user-filled data. We pass this form data to the `onSignup()` method in the signup component. This user-filled data is retrieved using, for example, `signupFormData.value.email`, as follows:

```
| <form name="form" (ngSubmit)="onSignup(signupFormData)" #signupFormData='ngForm'>
| </form>
```

Bootstrap elements

We used bootstrap elements to design our signup form. We included bootstrap and other dependent libraries such as tether and jQuery in our index.html file, as follows:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset=UTF-8>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <script src="https://nmpcdn.com/tether@1.2.4/dist/js/tether.min.js"></script>
  <link rel="stylesheet"
    href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/
    css/bootstrap.min.css">
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/
    jquery.min.js"></script>
  <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/
    js/bootstrap.min.js"></script>
  <title>Friends - A Social App</title>
  <base href="/">
</head>
<body>
<app-friends>
  Loading...
</app-friends>
</body>
</html>
```



Bootstrap is an open source toolkit for developing HTML, CSS, and JavaScript. For more details, refer to <https://getbootstrap.com/docs/4.0/getting-started/introduction/>.

The elements used are as follows:

- **Grid styles:** We use bootstrap grid styles to align our form to the middle as follows:

```
|   <div class="col-md-6 col-md-offset-3"></div>
```

- **Alert:** This element is used to provide context messages when a user performs any action on the element as follows:

```
|   <div class="alert alert-danger"></div>
```



For more details about bootstrap, refer to <https://getbootstrap.com/docs/4.0/components/alerts/>.



Angular data binding

As discussed in the preceding chapter, Angular supports various ways to bind data. In this particular case, we will support one-way binding using `(ngModel)='name'.`

Take a look at the complete `signup.component.html` file as of now:

```
<div class="col-md-6 col-md-offset-3">
  <h2>Signup</h2>
  <app-error-alert *ngIf="showError" [errorMessage]="errorMessage">
  </app-error-alert>
  <form name="form" (ngSubmit)="onSignup(signupFormData)"
    #signupFormData='ngForm'>
    <div class="form-group">
      <label for="name">Name</label>
      <input type="text" class="form-control" name="name"
        (ngModel)="name" #name="ngModel" required id="name"/>
      <div [hidden]="name.valid || name.pristine"
        class="alert alert-danger">
        Name is required
      </div>
    </div>
    <div class="form-group">
      <label for="email">Email</label>
      <input type="text" class="form-control" name="email"
        (ngModel)="email" #email="ngModel"
        required
        pattern="^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*
        (\.\w{2,3})+$"
        id="email"/>
      <div [hidden]="email.valid || email.pristine"
        class="alert alert-danger">
        <div [hidden]="!email.hasError('required')">Email is
          required</div>
        <div [hidden]="!email.hasError('pattern')">Email format
          should be
          <small><b>codingchum@gmail.com</b></small>
        </div>
      </div>
    </div>
    <div class="form-group">
      <label for="password">Password</label>
      <input type="password" class="form-control" name="password"
        (ngModel)="password" #password="ngModel" required
        id="password"/>
      <div [hidden]="password.valid || password.pristine"
        class="alert alert-danger">
        Password is required
      </div>
    </div>
  </form>
```

```
<div class="form-group">
    <label for="name">Retype Password</label>
    <input type="password" class="form-control"
        id="confirmPassword"
        required
            passwordEqual="password"
        (ngModel)="confirmPassword" name="confirmPassword"
#confirmPassword="ngModel">
        <div [hidden]="confirmPassword.valid || confirmPassword.pristine"
            class="alert alert-danger">
            Passwords did not match
        </div>
    </div>
    <div class="form-group">
        <label for="mobile">Mobile</label>
        <input type="text" class="form-control" name="mobile"
            (ngModel)="mobile" #mobile="ngModel"
        required
            pattern="[0-9]*
minlength="10"
maxlength="10"
id="mobile"/>
        <div [hidden]="mobile.valid || mobile.pristine"
            class="alert alert-danger">
            <div [hidden]!="mobile.hasError('minlength')">Mobile
                should be 10 digit</div>
            <div [hidden]!="mobile.hasError('required')">Mobile is
                required</div>
            <div [hidden]!="mobile.hasError('pattern')">Mobile
                number should be only numbers</div>
        </div>
    </div>
    <div class="form-group">
        <button type="submit" class="btn btn-success"
            [disabled]!="signupFormData.form.valid">SIGNUP</button>
        <a [routerLink]=["/app-friends-login"] class="btn btn-link">CANCEL</a>
    </div>
</form>
</div>
```

Error handling

Error handling is an important step to create a good application. It makes our product more robust and error-resistant.

We use angular validators to validate user inputs for accuracy and completeness. For user input, we can use a common inbuilt validator or create our own custom validator.



Please note that these validators are built into HTML, not angular itself.

The following are a few of the **inbuilt validators** that are used:

- **Required:** This makes the input field mandatory.
- **Min length:** This defines a lower limit to the user input. For example, we can limit the minimum length of mobile numbers to 10 digits.
- **Max length:** This defines a higher limit to the user input. For example, we can limit the maximum length of mobile number to 10 digits.
- **Pattern:** We can create a pattern for user input. For example, the mobile number takes only number as its input.

Inbuilt validators can be used in a signup template, as follows:

```
<div class="form-group">
    <label for="mobile">Mobile</label>
    <input type="text" class="form-control" name="mobile"
        [(ngModel)]="model.mobile" #mobile="ngModel"
        required
        pattern="[0-9]*
        minlength="10"
        maxlength="10"
        id="mobile"/>
    <div [hidden]="mobile.valid || mobile.pristine"
        class="alert alert-danger">
        <div [hidden]="!mobile.hasError('minlength')">Mobile should be
            10 digit</div>
        <div [hidden]="!mobile.hasError('required')">Mobile is
            required</div>
        <div [hidden]="!mobile.hasError('pattern')">Mobile number
            should be only numbers</div>
    </div>
</div>
```

A custom validator is the customized validator for our application's use case. In our application, we will take a password and make the user retype their password for confirmation, to validate those two passwords, we will create a custom password validator.

To create our validator, we will extend `Validator` from our form module. This provides a `validate` method to write our custom implementation:

```
import {Directive, forwardRef, Attribute} from '@angular/core';
import {Validator, AbstractControl, NG_VALIDATORS} from '@angular/forms';

@Directive({
  selector: '[passwordEqual][formControlName],[passwordEqual]
[FormControl],[passwordEqual][ngModel]',
  providers: [
    {provide: NG_VALIDATORS, useExisting: forwardRef(() =>
      PasswordEqualValidator), multi: true}
  ]
})
export class PasswordEqualValidator implements Validator {
  constructor(@Attribute('passwordEqual') public passwordEqual: string) {}

  validate(control: AbstractControl): { [key: string]: any } {
    let retypePassword = control.value;

    let originalPassword = control.root.get(this.passwordEqual);

    // original & retype password is equal
    return (originalPassword && retypePassword !==
      originalPassword.value)
      ? {passwordEqual: false} : null;
  }
}
```

We include the validator in our module, and our `authentication.module.ts` looks as follows:

```
@NgModule({
  imports: [
    ...
  ],
  declarations: [
    PasswordEqualValidator
  ],
  providers: [
    ...
  ]
})...
```

Finally, we use `PasswordEqualValidator` using the `passwordEqual` selector in our signup template to confirm the password, as shown in the following code:

```
| <input type="password" class="form-control" id="confirmPassword"
|   required
|   passwordEqual="password"
| [(ngModel)]="model.confirmPassword" name="confirmPassword"
| #confirmPassword="ngModel">
```

Firebase errors

Once a user provides correct input and clicks on the SIGNUP button, we can call the `signup()` method and the user is directed to their profile page. If any user input is incorrect, Firebase APIs throw an error to the application, and we will need to handle it in our application.

`createUserWithEmailAndPassword` of `AngularFireAuth` throws the following errors:

- **auth/email-already-in-use:** This error is thrown when a user provides an already-used email address for signing up.
- **auth/invalid-email:** This error is thrown when the email address is not valid.
- **auth/operation-not-allowed:** This error is thrown when Firebase authentication is not enabled when we create signup for the user. Most of the time, this occurs during development.
- **auth/weak-password:** This is thrown when the password provided is weak.

When we call the `signup` API, `auth` returns `Promise<any>`. This class has `then` and `catch` methods for success and failure scenarios. In the `catch` block, we read the error message and show error in `alert` dialog. Firebase error message has a human-readable error message, so we don't map the error code with message. We can also a create custom alert dialog and show the error message. In the next section, we will create a custom error alert and integrate it with the `signup` template.

```
onSignup(signupFormData): void {    this.authService.signup(signupFormData.value.email,
  signupFormData.value.password).then((userInfo) => {
    ...
  }).catch((error) => {
    this.showError = true;
    this.errorMessage = error.message;
  });
}
```


Creating a customized alert dialog

In this section, we will create an alert dialog component. This is used to display an error message. It is a reusable component, which means it can be used across the application. We will need to follow the following steps to create and configure our independent alert dialog:

- **Creating the component:** This is same as creating any other component. We have provided the `@Input errorMessage: any` binding to accept an error message from an other integrated component. This message will be displayed on the signup page.

Here's the complete `error-alert.component.ts`:

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

@Component({
  selector: 'app-error-alert',
  templateUrl: './error-alert.component.html',
  styleUrls: ['./error-alert.component.scss']
})
export class ErrorAlertComponent {
  @Input() errorMessage: any;
}
```

- **Creating the template:** We have used alert from bootstrap components. The `errorMessage` variable in curly braces accepts the text and displays the error message on the signup page.

Here's the complete `error-alert.component.html`:

```
<div class="error-alert-container">
  <div class="alert alert-danger fade show" role="alert">
    {{errorMessage}}
  </div>
</div>
```

- **Creating the style sheets:** We have margin `top` and `bottom` to align the alert properly.

Here's the complete `error-alert.component.scss`:

```
.error-alert-container {  
    margin-top: 24px;  
    margin-bottom: 8px;  
}
```

- **Configuring in the signup template:** Once our error alert is ready, we can integrate it with other components. In this chapter, we integrated an error alert with our signup component. This contains the `*ngIf` directive to enable the error alert and error message binding to display the text from the signup component.

Here's the modified `signup.component.html` file with an error alert, as follows:

```
<div class="col-md-6 col-md-offset-3">  
    <h2>Signup</h2>  
    <app-error-alert *ngIf="showError" [errorMessage]="errorMessage">  
    </app-error-alert>  
    ...  
</div>
```

- **Assigning the error message:** We have created two member variables: `errorMessage` and `showError`. These variables are enabled in the `onSignup()` method when an error occurs. `errorMessage` is assigned from `error.message`.

Here's the complete `signup.component.ts` with an error message:

```
export class SignupComponent {  
  
    errorMessage: string;  
  
    showError: boolean;  
  
    onSignup(signupFormData): void {  
        this.authService.signup(signupFormData.value.email,  
        signupFormData.value.password).then((userInfo) => {  
            // Register the new user  
            ...  
        }).catch((error) => {  
            this.showError = true;  
            this.errorMessage = error.message;  
        });  
    }  
}
```

During signup of a new user, if you type a password with just one character, the error in the Signup page looks as follows:

Signup

Password should be at least 6 characters

Creating a signup component

A signup component is a controller and it is used to react to a user's action, such as sign up or cancel. It is injected with the following two services:

- **Authentication service:** It provides authentication-related functionalities, such as login, signup, and logout
- **User service:** It interacts with the Firebase database to store extra user information, such as the mobile number and name.

The constructor of the signup component accepts authentication and user services, as follows:

```
constructor(private authService: AuthenticationService, private userService: UserService) { }
```

When the user clicks on the SIGNUP button, the `onSignup` method is called. It accepts form data as a parameter, which has user-typed information. The email and password are retrieved and passed to the authentication service's `signup` method. On successful signup, we retrieve other information from the form data and store it in the user domain model. Finally, we will pass this newly created `user` object to user service and register it in the Firebase database:



On successful signup, the user info contains the UID. This is the unique identifier for a particular user. It is used as an indicator to store data in the Firebase database. This is also used to retrieve user information from the database.

```
onSignup(signupFormData): void {
    this.authService.signup(signupFormData.value.email,
        signupFormData.value.password).then((userInfo) => {
            // Register the new user
            const user: User = new User(signupFormData.value.email,
                signupFormData.value.name, signupFormData.value.mobile,
                userInfo.uid, 0, '');
            this.writeNewUser(user);
        }).catch((error) => {
            this.showError = true;
            this.errorMessage = error.message;
        });
}

private writeNewUser(user: User): void {
```

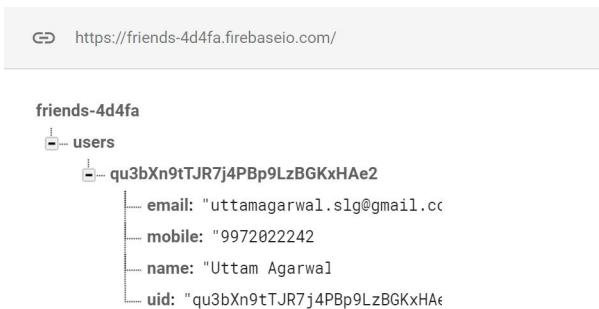
```
|     this.userService.addUser(user);  
| }  
| }
```

After a successful signup, Firebase authentication will have the following entry:

The screenshot shows the Firebase console interface. At the top, there is a search bar labeled "Search by email address, phone number, or user UID" and a blue "ADD USER" button. Below the search bar is a table header with columns: Identifier, Providers, Created, Signed In, and User UID ↑. A single user entry is listed: "uttamagarwal.slg@gmail.com" with an envelope icon under Providers, "Aug 3, 2019" under Created, "Aug 3, 2019" under Signed In, and "qu3bXn9tTJR7j4PBp9LzB..." under User UID. At the bottom of the table, there are pagination controls: "Rows per page: 50", "1-1 of 1", and navigation arrows.

Identifier	Providers	Created	Signed In	User UID ↑
uttamagarwal.slg@gmail.com	✉	Aug 3, 2019	Aug 3, 2019	qu3bXn9tTJR7j4PBp9LzB...

The Firebase database will have the following entry:



Here's the complete `signup.component.ts` file as of now:

```
import {Component} from '@angular/core';  
import {User} from '../../../../../services/user';  
import {AuthenticationService} from '../../../../../services/authentication.service';  
import {UserService} from '../../../../../services/user.service';  
  
@Component({  
  selector: 'app-friends-signup',  
  styleUrls: ['signup.component.scss'],  
  templateUrl: 'signup.component.html'  
})  
export class SignupComponent {  
  
  errorMessage: string;  
  
  showError: boolean;  
  
  constructor(private authService: AuthenticationService,  
              private userService: UserService) {}  
  
  onSignup(signupFormData): void {  
    this.authService.signup(signupFormData.value.email,  
                           signupFormData.value.password).then((userInfo) => {  
      // Register the new user  
      const user: User = new User(signupFormData.value.email,
```

```

        signupFormData.value.name, signupFormData.value.mobile,
        userInfo.uid, 0, '');
    this.writeNewUser(user);
}).catch((error) => {
    this.showError = true;
    this.errorMessage = error.message;
});
}

private writeNewUser(user: User): void {
    this.userService.addUser(user);
}

```

Finally, register `signupComponent` in the authentication routing module, as follows:

```

import {NgModule} from '@angular/core';
import {RouterModule, Routes} from '@angular/router';
import {SignupComponent} from './signup/signup.component';

export const ROUTES: Routes = [
  {path: 'app-friends-signup', component: SignupComponent}
];

/**
 * Authentication Routing Module
 */
@NgModule({
  imports: [RouterModule.forChild(ROUTES)],
  exports: [RouterModule]
})
export class AuthenticationRouting {
}

```

Now, take a look at your component in the browser using `http://localhost:4200/app-friends-signup`, the component looks as follows:

localhost:4200/app-friends-signup

Apps Bookmarks Java Designs Others My Projects Android Big Data Vm

Signup

Name

Email

Password

Retype Password

Mobile

SIGNUP CANCEL

Summary

Finally, we build our signup component. The key learning in this chapter is injectable services. We created authentication and user services. These services used the Angular fire library to interact with Firebase authentication and database services. These new services were added to the signup component using dependency injection. This component used the services to sign up new users and add them to Firebase database. We inbuilt and customized message alerts.

In the next chapter, we will create a login component and log-in the newly registered user account. After successful login, the user will be directed to the user profile page.

8 - Sept-2018

Creating a Login Component

In this chapter, we will build our second component. We will create a login page, which will be similar to the signup page. We will also add more functionalities to services. We will use Email/Password authentication to log in the user in. User details are already added to the Firebase database during the signup process. We will retrieve the user details from Firebase and pass them to the user profile page. We will also handle common error scenarios, as this will reinforce the concept.

The following topics will be covered in this chapter:

- Adding login functionalities to existing services
- Reusing the domain model
- Creating a login template
- Error handling for login
- Creating login components
- Resetting password

Adding login functionalities to existing services

In the preceding chapter, we used Email/Password authentication and also added our user to the Firebase database. We acquired basic knowledge about pushing data to Firebase. In this section, we will log the user and retrieve the user details from Firebase. We will add login functionalities in the authentication and user service.

Authentication service

During signup, the user is registered to Firebase. `AngularFireAuth` has the `signInWithEmailAndPassword` method to login the user. This method returns `firebase.Promise<any>`. This class has `then` and `catch` methods to handle success and failure scenarios. On success, we redirect the user to the user-profile case, and we show the error message on failure.

Considering the following methods:

- **Login:** This method validates the user and passes user information on successful login, as follows:

```
public login(email: string, password: string): Promise<any> {
    return this.angularFireAuth.auth.signInWithEmailAndPassword(email, password);
}
```

- **Reset:** `AngularFireAuth` provides an API to reset the password. Firebase provides the infrastructure for password resets, such as password email notification. We just need to call the `resetPassword` API in the authentication service, as follows:

```
public resetPassword(email: string): Promise<any> {
    return this.angularFireAuth.auth.sendPasswordResetEmail(email);
}
```

Here's the complete `authentication.service.ts` as of now:

```
import {Injectable} from '@angular/core';
import {AngularFireAuth} from 'angularfire2/auth';

/**
 * Authentication service
 */
@Injectable()
export class AuthenticationService {

    /**
     * Constructor
     *
     * @param {AngularFireAuth} angularFireAuth provides the
     * functionality related to authentication
    
```

```
 */
constructor(private angularFireAuth: AngularFireAuth) {
}

public signup(email: string, password: string): Promise<any> {
  return
    this.angularFireAuth.auth.createUserWithEmailAndPassword(email,
      password);
}

public login(email: string, password: string): Promise<any> {
  return this.angularFireAuth.auth.signInWithEmailAndPassword(email,
    password);
}

public resetPassword(email: string): Promise<any> {
  return this.angularFireAuth.auth.sendPasswordResetEmail(email);
}

}
```

User service

Here, we cover how to retrieve value from the Firebase by performing a read operation. `AngularFire2` has the `AngularFireDatabase` class, which provides the following two methods to read the data from Firebase:

- `object`: This retrieves the JSON object. It returns `AngularFireObject<T>`, which provides the `valueChanges` method to return the Observable. For example, if we want to get a user object from Firebase, then we use this method.
- `list`: This retrieves an array of JSON objects. It returns `AngularFireList<T>`, which provides the `valueChanges` method to return the Observable with an array of objects. For example, if we want to get all the users registered in our application, then this method is handful.

Once the user types their correct credentials, then we retrieve the user details using the `object` method of `AngularFireDatabase`.

The list of methods in the `AngularFireDatabase` class, as follows:

```
export declare class AngularFireDatabase {  
    app: FirebaseApp;  
    database: database.Database;  
    constructor(app: FirebaseApp);  
    list<T>(pathOrRef: PathReference, queryFn?: QueryFn):  
        AngularFireList<T>;  
    object<T>(pathOrRef: PathReference): AngularFireObject<T>;  
    createPushId(): string | null;  
}
```

Mostly in our application we use the `list` and `object` methods of `AngularFireDatabase`. These methods accept the `pathOrRef` parameters. The `list` method accepts an additional `queryFn` as a parameter. The objective of these parameters is as follows:

- `pathOrRef`: This parameter accepts the path of data in the Firebase database. As shown in the following example, to access user data we provide the path till the user `uid`.

```
public getUser(uid: string): Observable<User> {  
    return this.firebaseio.database.ref(`$USERS_CHILD` + uid).valueChanges();  
}
```

For example, suppose we want to retrieve user information for user id qu3bXn9tTJR7j4PBp9LzBGKxHAe2, then the path in this case is /users/qu3bXn9tTJR7j4PBp9LzBGKxHAe2:



- `queryFn`: This optional parameter in the `list` method, filters the list based on the filter criteria. For example, suppose we want the first three registered users, then we use `limitToFirst` query.

Here's the complete `user.service.ts` as of now:

```
import {Injectable} from '@angular/core';
import {AngularFireDatabase} from 'angularfire2/database';
import {User} from './user';
import 'firebase/storage';
import {USERS_CHILD} from './database-constants';
import {Observable} from 'rxjs/Observable';

/**
 * User service
 */
@Injectable()
export class UserService {

    /**
     * Constructor
     *
     * @param {AngularFireDatabase} fireDb provides the functionality for
     *         Firebase Database
     */
    constructor(private fireDb: AngularFireDatabase) {
    }

    public addUser(user: User): void {
        this.fireDb.object(`/${USERS_CHILD}/${user.uid}`).set(user);
    }

    public getUser(uid: string): Observable<User> {
        return this.fireDb.object<User>
            (`/${USERS_CHILD}/${uid}`).valueChanges();
    }
}
```

Reusing the domain model

Once the user is successfully logged in, we will retrieve the user object from our Firebase database. On successful log in, we get the string uid of the user and use this uid to retrieve the user details from the user's node in the Firebase database. As discussed in the preceding section, we get the data in the JSON format as follows:

```
{  
  "email": "uttamagarwal.slg@gmail.com",  
  "mobile": "9972022242",  
  "name": "Uttam Agarwal",  
  "uid": "qu3bXn9tTJR7j4PBp9LzBGKxHAe2"  
}
```

This JSON object needs to be mapped to the user object. When we retrieve the JSON object from Firebase using `AngularFireDatabase`, we provide the type object in angle brackets, `<User>` and this maps the JSON to the user object:

```
public getUser(uid: string): Observable<User> {  
  return this.firebaseio.object<User>(`${USERS_CHILD}/${uid}`).valueChanges();  
}
```

The constructor accepts all the parameters assigned to its member variable as shown in the following code; here's the complete `user.ts`:

```
export class User {  
  
  email: string;  
  
  name: string;  
  
  mobile: string;  
  
  uid: string;  
  
  constructor(email: string,  
             name: string,  
             mobile: string,  
             uid: string) {  
    this.email = email;  
    this.name = name;  
    this.mobile = mobile;  
    this.uid = uid;  
  }  
}
```

Creating a login template

A login template is the view and this part is similar to the signup template. We have reused the email and password elements of signup. It has the following three parts:

1. **Input form:** This is a textbox and takes a user-typed value
2. **Submit action:** It triggers the `onLogin()` method with login form data as its parameter to the component
3. **Navigation:** This will be covered in detail in detail in [Chapter 4, Routing and Navigation between Components](#), so I will not touch on this part here

The following is the complete `login.component.html`:

```
<div class="col-md-6 col-md-offset-3 container">
    <h2>Login</h2>
    <app-error-alert *ngIf="showError" [errorMessage]="errorMessage"></app-error-alert>
    <form name="form" (ngSubmit)="onLogin(loginFormData)"
        #loginFormData='ngForm'>
        <div class="form-group">
            <label for="email">Email</label>
            <input type="text" class="form-control" name="email"
                (ngModel)="email" #email="ngModel"
                required
                pattern="^\\w+([\\.-]?\\w+)*@\\w+([\\.-]?\\w+)*
                (\\.\\w{2,3})+$"
                id="email"/>
            <div [hidden]="email.valid || email.pristine"
                class="alert alert-danger">
                <div [hidden]="!email.hasError('required')">Email is
                    required</div>
                <div [hidden]="!email.hasError('pattern')">Email format
                    should be
                    <small><b>codingchum@gmail.com</b></small>
                </div>
            </div>
        </div>
        <div class="form-group">
            <label for="password">Password</label>
            <input type="password" class="form-control" name="password"
                (ngModel)="password" #password="ngModel"
                required id="password"/>
        </div>
    </form>
</div>
```

```
        <div [hidden]="password.valid || password.pristine"
      class="alert alert-danger">
          Password is required
        </div>
    </div>
<div class="form-group">
    <button type="submit" id="login-btn" class="btn btn-
success" [disabled]="!loginFormData.form.valid">
    LOGIN</button>
    <button routerLink="/app-friends-signup" data-tag="signup-
tag" routerLinkActive="active"
    class="btn btn-success">
        SIGNUP
    </button>
</div>
</form>
</div>
```

Error handling for login

As discussed in the preceding chapter, we will handle user input and server errors. In the login component, the user input error remains the same as with the signup component. In this component, we used the same in-built Angular error. This error message helps the user enter correct information.

Firebase error

Login Firebase APIs throw an error when a user logs in to a friend's application. We use the `signInWithEmailAndPassword()` method of `AngularFireAuth` to sign up the user. This API throws the following errors:

- `auth/invalid-email`: As the name implies, this error occurs when a user provides an invalid email address.
- `auth/user-disabled`: This error occurs when the registered user account is disabled in Firebase. This feature is required when the registered user is not complying with the terms and conditions of the application. Then, we can show a meaningful message to the user.

You can disable the user account in the following three steps:

1. Go to Firebase.
2. Go to DEVELOP|Authentication on the left panel.
3. Highlight the user on the right panel. Click on the overflow icon and then on the Disable account option.

Take a look at the following Disable account option in Firebase:

The screenshot shows the Firebase Authentication console. At the top, there is a search bar labeled "Search by email address, phone number, or user UID" and a blue "ADD USER" button. Below the search bar is a table header with columns: Identifier, Providers, Created, Signed In, and User UID. A user account for "uttamagarwal.slg@gmail.com" is listed, showing "Aug 3, 2023" under Created and "Aug 11, ..." under Signed In. The User UID is "qu3bXn9tTJR7j4F". To the right of the table, a context menu is open over the User UID, listing "Reset password", "Disable account", and "Delete account". At the bottom of the screen, there is a pagination control with "Rows per page: 50" and a page indicator showing "1".

- `auth/user-not-found`: This error occurs when the user has not signed up in our application. In this case, we can direct the user to the signup page.
- `auth/wrong-password`: This error occurs when the password is not correct. In this case, the user has two options: either provide the correct password or reset the password.

The login method in the authentication service returns `Promise<any>`. We handle the error in the `catch` block.



A promise is a result of any asynchronous operation. After a success or fail operation, we use the promise object to retrieve the stored data.

We will then reuse the error alert created in [Chapter 2, Creating a Signup Component](#) and show the error :

```
onLogin(loginFormData): void {
  this.authService.login(loginFormData.value.email,
  loginFormData.value.password).then((userInfo) => {
    ...
  }).catch((error) => {
    this.errorMessage = error.message;
    this.showError = true;
  });
}
```

When a user provides a wrong password, the customized error message will be shown as follows:

Login

The password is invalid or the user does not have a password.

Creating a login component

The login component handles the user interaction with the UI. It is injected with three services to perform various actions:

- **Authentication service:** It provides the login API for signing in following the registered user.
- **User service:** It provides a method to retrieve the user details from the Firebase database.
- **Router:** This service is required to route to different pages in the application. In the login component, we will use this service to route the user to the signup page. This will be covered in more detail in the next chapter.

Services are injected in the `constructor` of the login component, as follows:

```
constructor(  
    private userService: UserService,  
    private router: Router,  
    private authService: AuthenticationService  
) {}
```

The login component also handles user click events. When the user clicks on the LOGIN button, the `onLogin` method is called. This method accepts login form data as a parameter, that has user-typed data. We retrieve the email and password as `loginFormData.value.email` and `loginFormData.value.password`. This data is passed to the authentication service for login. On successful log in, we get the `uid` of the user and then use this `uid` to retrieve the user details from our Firebase database. We also cache these user details in User Service for future reference in other pages:

```
onLogin(loginFormData): void {  
    this.authService.login(loginFormData.value.email,  
        loginFormData.value.password).then((userInfo) => {  
            // Login user  
            const uid: string = userInfo.uid;  
            this.getUserInfo(uid);  
        }).catch((error) => {  
            ...  
        });  
}  
  
private getUserInfo(uid: string) {  
    this.userService.getUser(uid).subscribe(snapshot => {  
        this.user = snapshot;
```

```
|   });
| }
```

Finally, we saw how to log in as a registered user. Now, the only part missing is password recovery, which we will cover in the next section.

Here's the complete `login.component.ts` as of now:

```
import {Component} from '@angular/core';
import {User} from '../../services/user';
import {Router} from '@angular/router';
import {AuthenticationService} from '../../../../../services/authentication.service';
import {UserService} from '../../../../../services/user.service';
import {AngularFireAuth} from 'angularfire2/auth';

@Component({
  selector: 'app-friends-login',
  styleUrls: ['login.component.scss'],
  templateUrl: 'login.component.html',
})
export class LoginComponent {

  errorMessage: string;
  showError: boolean;

  private user: User;

  constructor(private userService: UserService,
              private router: Router,
              private authService: AuthenticationService,
              private angularFireAuth: AngularFireAuth) {
    this.angularFireAuth.auth.onAuthStateChanged(user => {
      if (user) {
        this.getUserInfo(user.uid);
      }
    });
  }

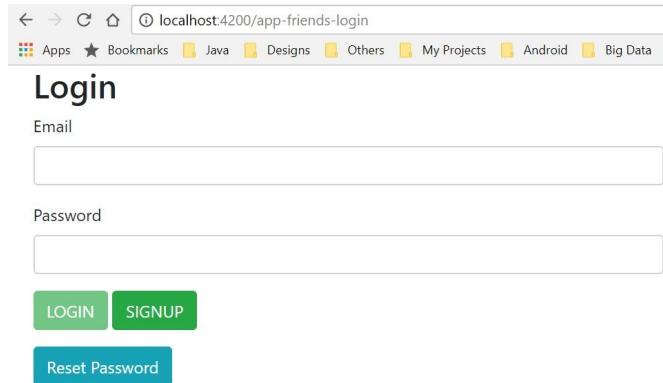
  onLogin(loginFormData): void {
    this.authService.login(loginFormData.value.email,
    loginFormData.value.password).then((user) => {
      // Login user
      const uid: string = user.uid;
      this.getUserInfo(uid);
    }).catch((error) => {
      this.errorMessage = error.message;
      this.showError = true;
    });
  }

  private getUserInfo(uid: string) {
    this.userService.getUser(uid).subscribe(snapshot => {
      this.user = snapshot;
    });
  }
}
```

Finally we register `LoginComponent` in authentication routing modules:

```
| {path: 'app-friends-login', component: LoginComponent}
```

So paste the URL, `http://localhost:4200/app-friends-login` in the browser and our login component looks like the following:



Resetting the password

It is good to provide a password recovery option in our application and this process increases the usability of our application. The excellent thing is that Firebase provides all the infrastructure required to perform this action. We will add this feature in our application step by step.

Adding modal template

The first step in the password a reset action is to take a user email address, and we will use a modal in this scenario. A modal is a popup/dialog that appears on top of the current page view. We will use a modal to display the popup to take the user email address.

Adding the modal template in the login html: We have modified the `login.component.html` file to add the Reset password button, as shown in the following code:

```
<div id="password_reset" class="modal fade" role="dialog">
  <div class="modal-dialog modal-sm">
    <form name="form" (ngSubmit)="onReset(resetFormData)"
      #resetFormData='ngForm'>
      <div class="modal-content">
        <div class="modal-header">
          <h4 class="modal-title">Forgot Password?</h4>
          <button type="button" class="close" data-
            dismiss="modal">&times;</button>
        </div>
        <div class="modal-body">
          <p>Please enter your registered email to sent you
            the password reset instructions.</p>

          <div class="form-group">
            <label for="reset_email">Email</label>
            <input type="text" class="form-control"
              name="email" (ngModel)="email"
              #email="ngModel"
              required
              pattern="^\w+([\.-]?\w+)*@\w+([\.-]?
                \w+)*(\.\w{2,3})+$"
              id="reset_email"/>
            </div>
          </div>
          <div class="modal-footer form-group">
            <button type="submit" class="btn btn-default"
              [disabled]="!resetFormData.form.valid"
              >Reset
            </button>
            <button type="button" class="btn btn-default" data-
              dismiss="modal">Close</button>
          </div>
        </div>
      </form>
    </div>
  </div>
</div>
```

When the user clicks on Reset Password, the following modal appears:

Forgot Password? ×

Please enter your registered email to sent you the password reset instructions.

Email

[Reset](#)

[Close](#)

Here's the complete `login.component.html` as of now:

```
<div class="col-md-6 col-md-offset-3 container">
  <h2>Login</h2>
  <app-error-alert *ngIf="showError" [errorMessage]="errorMessage"></app-error-alert>
  <form name="form" (ngSubmit)="onLogin(loginFormData)"
    #loginFormData='ngForm'>
    <div class="form-group">
      <label for="email">Email</label>
      <input type="text" class="form-control" name="email"
        (ngModel)="email" #email="ngModel"
        required
        pattern="^\\w+([\\.-]?\\w+)*@\\w+([\\.-]?\\w+)*
        (\\.\\w{2,3})+$"
        id="email"/>
      <div [hidden]="email.valid || email.pristine"
        class="alert alert-danger">
        <div [hidden]="!email.hasError('required')">Email is
          required</div>
        <div [hidden]="!email.hasError('pattern')">Email format
          should be
          <small><b>codingchum@gmail.com</b></small>
        </div>
      </div>
    </div>
    <div class="form-group">
      <label for="password">Password</label>
      <input type="password" class="form-control" name="password"
        (ngModel)="password" #password="ngModel"
        required id="password"/>
      <div [hidden]="password.valid || password.pristine"
        class="alert alert-danger">
        Password is required
      </div>
    </div>
    <div class="form-group">
      <button type="submit" id="login-btn" class="btn btn-
        success" [disabled]="!loginFormData.form.valid">
        LOGIN</button>
      <button routerLink="/app-friends-signup" data-tag="signup-
        tag" routerLinkActive="active" class="btn btn-success">
        SIGNUP
      </button>
    </div>
  </form>
```

```

<button type="button" data-tag="password-reset-tag" class="btn btn-
info" data-toggle="modal" data-target="#password_reset">
    Reset Password
</button>
</div>
<div id="password_reset" class="modal fade" role="dialog">
    <div class="modal-dialog modal-sm">
        <form name="form" (ngSubmit)="onReset(resetFormData)"
            #resetFormData='ngForm'>
            <div class="modal-content">
                <div class="modal-header">
                    <h4 class="modal-title">Forgot Password?</h4>
                    <button type="button" class="close" data-
                        dismiss="modal">&times;</button>
                </div>
                <div class="modal-body">
                    <p>Please enter your registered email to sent you
                        the password reset instructions.</p>
                    <div class="form-group">
                        <label for="reset_email">Email</label>
                        <input type="text" class="form-control"
                            name="email" (ngModel)="email"
                            #email="ngModel"
                            required
                            pattern="^[\w+([\.-]?\w+)*@\w+([\.-]?
                            \w+)*(\.\w{2,3})+$"
                            id="reset_email"/>
                    </div>
                </div>
                <div class="modal-footer form-group">
                    <button type="submit" class="btn btn-default"
                        [disabled]="!resetFormData.form.valid"
                    >Reset
                    </button>
                    <button type="button" class="btn btn-default" data-
                        dismiss="modal">Close</button>
                </div>
            </div>
        </form>
    </div>
</div>

```

Adding the onReset() functionality

The next step is to add reset method in the login component. The `onReset()` method in the login component resets the password and sends the reset instruction to the registered email. This email contains the reset link; when we click on the link, it opens another browser tab with an alert dialog to provide the new password.

We add the `onReset()` method in the login component as shown here:

```
onReset(resetFormData): void {
  this.authService.resetPassword(resetFormData.value.email).then(() =>
  {
    alert('Reset instruction sent to your mail');
  }).catch((error) => {
    this.errorMessage = error.message;
    this.showError = true;
  });
}
```

Editing the password-reset template in Firebase

Firebase provides the option to change the template for the email. We can customize the email body. In this application, we are using the default template, even though you can change it through this process:

1. Go to Firebase authentication
2. On the right panel, click on the TEMPLATES tab
3. Click on the Password reset option on the left panel
4. Click on the pencil icon to edit it

Consider the following password reset template in Firebase:

The screenshot shows the Firebase Authentication interface. The top navigation bar has tabs for USERS, SIGN-IN METHOD, TEMPLATES (which is selected), and USAGE. The main content area is titled 'Templates'. On the left, there's a sidebar with 'Email Types' and 'SMS Types'. Under 'Email Types', 'Password reset' is highlighted with a blue background. The main panel displays the 'Password reset' template details. It includes a description: 'When a user forgets their password, a password reset email is sent to help them set up a new one.' with a 'Learn more' link. Below this, there's a note: 'Update the [project-level setting](#) below to continue' with a gear icon. It shows the 'Project public-facing name' set to 'project-807434545532'. There are also fields for 'Sender name' and 'From'.

Summary

In this chapter, we implemented the login component and enhanced the authentication and user services. The login template and component look similar to the sign-in template and component. We then implemented password reset features in our application. Firebase provides the ingredient to implement reset password functionality. We also implemented our first modal in our application.

In the next chapter, we will be covering navigation flow between different components. We will also add Angular guard to restrict or enable navigation to a component based on a guard condition.

9-Sep-2018

Routing and Navigation between Components

In this chapter, we will cover navigation in an Angular application. We will implement routers for our app component and authentication. We will also cover router outlets for our component view and create a router module for authentication. This will make our authentication feature completely independent of other modules. Component navigation in an authentication module will be taken care of by a child router module. We will also discuss Angular guard, which will restrict the navigation based on the condition to the component. This enhances the security of our application. Finally, we will cover Firebase session life cycles in depth and redirect our user to the Login or My Profile page.

In this chapter, we will cover the following topics:

- Enabling routes in an app component
- Creating a routing module for authentication
- Adding a router link
- Using authentication guard
- Firebase session life cycles
- Our project structure as of now

Enabling routes in an app component

In this section, we enable routing and create the main navigation bar for our application. The steps to enable routes are as follows:

- **Adding the base reference URL:** We will need to add a base element in `index.html` to tell the Angular router how to compose the navigation URL.

We added `href` in the head tag of `index.html`:

```
| <base href="/">
```

- **Creating the main navigation bar:** Most web applications have a navigation bar at the top of the page for navigating to different pages in the application. We have added a main navigation bar for our friends application using the bootstrap `nav bar` component.

1. The first step is to include the `nav` tag in `app.component.html`, as follows:

```
| <nav class="navbar navbar-expand-lg navbar-light bg-color"></nav>
```

2. The second step is to create a list of items using the `ul` tag:

```
| <ul class="navbar-nav"></ul>
```

3. The third step is to create the items using the `li` tag. In our application, we have to activate tabs based on the user login condition, for example, a user profile tab will appear when the user logs in. We followed these conditions to activate tabs in our navigation bar:

- **User not signed:** We activated only the about and login tabs. A user can navigate to the signup page by clicking on the SIGNUP button.
- **User signed:** We deactivated the login page and activated the user profile page.

In order to achieve these scenarios, we will access the authentication service object, check the user logged-in status, and activate the tabs. We use the `ngIf` directive for condition check:

```
<li class="nav-item" *ngIf="authenticationService?.isAuthenticated()"><a class="nav-link" routerLink="/app-friends-userprofile" routerLinkActive="active">My Profile</a></li>
```

The question mark in `authenticationService?` in `AuthenticationService` in the template ensures that the object is not null. This object is defined in `app.component.ts`, as shown in the following code:

```
authenticationService: AuthenticationService;  
  
constructor(private authService: AuthenticationService) {  
    this.authenticationService = authService;  
}
```

Here's the complete `app.component.html` as of now:

```
<h1 class="title">Friends - A Social App</h1>  
<div class="nav-container">  
<nav class="navbar navbar-expand-lg navbar-light bg-color">  
    <div class="collapse navbar-collapse" id="navbarNav">  
        <ul class="navbar-nav">  
            <li class="nav-item"  
                *ngIf="authenticationService?.isAuthenticated()"><a class="nav-link" routerLink="/app-friends-userprofile" routerLinkActive="active">My Profile</a></li>  
            <li class="nav-item"  
                *ngIf="authenticationService?.isAuthenticated()">  
                <a class="nav-link" routerLink="/app-friends-userfriends" routerLinkActive="active">Friends</a></li>  
            <li class="nav-item" ><a class="nav-link" routerLink="/app-friends-about" routerLinkActive="active">About</a></li>  
            <li class="nav-item" active  
                *ngIf="!authenticationService?.isAuthenticated()">  
                <a class="nav-link" routerLink="/app-friends-login" routerLinkActive="active">Login</a></li>  
        </ul>  
        <div class="form-container">  
            <form class="form-inline my-2 my-lg-0">  
                <input class="form-control mr-sm-2" type="text" placeholder="Search friends..." aria-label="Search">  
                <button class="btn btn-success my-2 my-sm-0" type="submit">Search</button>  
            </form>  
        </div>  
    </div>  
</nav>  
</div>  
<router-outlet></router-outlet>
```

- **Creating page not found and about component:** The `PageNotFoundComponent` component is used to display the view for an incorrect URL and `AboutComponent` is used to display information about the site.

Both these components look similar and have a message header with the `h2` tag with text. In these components, we defined a template and style

sheets in component annotation. This is one simple way to create the component.

The following is the complete `page-not-found.component.ts`:

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

@Component({
  selector: 'app-friends-page-not-found',
  template: '<h2>Page not found</h2>'
})
export class PageNotFoundComponent {}
```

The following is the complete `about.component.ts`:

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

@Component({
  selector: 'app-friends-about',
  template: '<h2>Friends is a social app</h2>'
})
export class AboutComponent {}
```

- **Creating routes for an app component:** We create routes for our main app module for the about and page not found components.

We create the routes for two components with the following path as; the double star (**) is a wildcard so that when the user provides any incorrect URL, the router navigates to the page not found component:

```
export const ROUTES: Routes = [
  {path: 'app-friends-about', component: AboutComponent,
   pathMatch: 'full'},
  {path: '**', redirectTo: 'app-friends-page-not-found'},
];
```

We inject routes for the app module in Angular `RouterModule`; the complete `app.routing.ts` as of now is as follows:

```
import {RouterModule, Routes} from '@angular/router';
import {NgModule} from '@angular/core';
import {PageNotFoundComponent} from './notfound/page-not-found.component';
import {AboutComponent} from './about/about.component';

export const ROUTES: Routes = [
  {path: 'app-friends-about', component: AboutComponent,
   pathMatch: 'full'},
  {path: '**', redirectTo: 'app-friends-page-not-found'},
];

@NgModule({
  imports: [
```

```
    RouterModule.forRoot(
      ROUTES
    ]),
    exports: [
      RouterModule
    ]
})
export class AppRoutingModule { }
```

Finally, we integrate the app routing module in our main app module. We add `AppRoutingModule` in the import tag as follows:

```
@NgModule({
  declarations: [
    AppComponent,
    PageNotFoundComponent,
    AboutComponent
  ],
  imports: [
    ...
    AuthenticationModule,
    AppRoutingModule
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

When the user is not logged in, the navigation bar will show the About and Login tabs:

Friends - A Social App

About Login

When the user is logged in, the navigation bar will show the My Profile, Friends, and About tabs:

Friends - A Social App

My Profile Friends About

Creating a routing module for authentication

As discussed earlier, we will build the **individual routes for each feature module**. We have two components in authentication:

- Login Component
- Signup Component

In order to define the routing module, we will need to create routes constant for navigation:

```
| export const ROUTES: Routes = [
|   {path: 'app-friends-login', component: LoginComponent},
|   {path: 'app-friends-signup', component: SignupComponent}
| ];
```

Since these routes are child components of the main app component, we inject the routes into the child router module:

```
| RouterModule.forChild( ROUTES )
```

Here's the complete `authentication.routing.ts` as of now:

```
import {NgModule} from '@angular/core';
import {RouterModule, Routes} from '@angular/router';
import {LoginComponent} from './login/login.component';
import {SignupComponent} from './signup/signup.component';

export const ROUTES: Routes = [
  {path: 'app-friends-login', component: LoginComponent},
  {path: 'app-friends-signup', component: SignupComponent}
];

/**
 * Authentication Routing Module
 */
@NgModule({
  imports: [
    RouterModule.forChild(ROUTES)
  ],
  exports: [
    RouterModule
  ]
})
export class AuthenticationRouting {}
```

After creating the routing module, we include the `AuthenticationRouting` module in the main authentication module; this makes our authentication module independent of the main app module.

Here is a sample `authentication.module.ts`:

```
import { NgModule } from '@angular/core';
import { AuthenticationRoutingModule } from './authentication.routing';

/**
 * Authentication Module
 */
@NgModule({
  imports: [
    ...
    AuthenticationRoutingModule
  ],
  declarations: [
    ...
  ],
  providers: [
    ...
  ]
})
export class AuthenticationModule { }
```

Exploring more routing techniques

In this section, we will explore two navigation methods in our application:

- **Static Routing:** In static routing, we will provide the navigation in the HTML template. Angular router provides a directive to perform the navigation action. We will include the `routerLink` directive with we inject the routes into the child router module: path for navigation. As seen in the following code, when you click on the `SIGNUP` button, the Angular framework navigates to the `signup` component using `Router`:

```
<button routerLink="/app-friends-signup" data-tag="signup-tag"
        routerLinkActive="active" class="btn btn-success">
    SIGNUP
</button>
```

- **Dynamic Routing:** In dynamic routing, we use the `Router` component of the Angular framework. The instance is injected using dependency injection in the constructor:

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

Router provides the `navigateByUrl()` method to route to different components; in the following scenario, after successful login, we navigate to the user profile page:

```
private navigateToUserProfile() {
    this.router.navigateByUrl('/app-friends-userprofile');
}
```

Now we have added navigation to our application; thus, in the next section we add conditional-based navigation guards.

Adding authentication guards

Guard are a really useful feature in Angular for protecting routes. They provide the security feature used to restrict resources in our application so that the user is not able to consume a resource without proper permission.

There are different guard types in Angular:

- `CanActivate`: This is used to decide whether the route can be activated
- `CanActivateChild`: This is used to decide whether the child route can be activated
- `CanDeactivate`: This is used to decide whether the route can be deactivated
- `CanLoad`: This is used to decide whether the module can be loaded lazily

We will take a look at an example of the `CanActivate` guard in authentication. We will allow the user to access the user profile and friends pages only after their authentication is successful. This means that the user will not be allowed to access `http://localhost:4200/app-friends-userprofile` without authentication and will be redirected to the login page. The steps involved in activating the guard are as follows:

1. **Guard condition:** We need to provide conditions for activating the guard. In this scenario, we are checking the current user status. This condition was also used in the app component to show various tabs based on the condition:

```
public isAuthenticated(): boolean {
    let user = this.angularFireAuth.auth.currentUser;
    return user ? true : false;
}
```

Here's the complete `authentication.service.ts` as of now:

```
import {Injectable} from '@angular/core';
import {AngularFireAuth} from 'angularfire2/auth';

/**
 * Authentication service
 */
@Injectable()
```

```

export class AuthenticationService {

    /**
     * Constructor
     *
     * @param {AngularFireAuth} angularFireAuth provides the
     * functionality related to authentication
     */
    constructor(private angularFireAuth: AngularFireAuth) {
    }

    public signup(email: string, password: string): Promise<any> {
        return
            this.angularFireAuth.auth.createUserWithEmailAndPassword(email,
            password);
    }

    public login(email: string, password: string): Promise<any> {
        return
            this.angularFireAuth.auth.signInWithEmailAndPassword(email,
            password);
    }

    public resetPassword(email: string): Promise<any> {
        return this.angularFireAuth.auth.sendPasswordResetEmail(email);
    }

    public isAuthenticated(): boolean {
        const user = this.angularFireAuth.auth.currentUser;
        return user ? true : false;
    }

    public signout() {
        return this.angularFireAuth.auth.signOut();
    }
}

```

2. Guard implementation: We implement the guard by extending the `CanActivate` interface and overriding the `canActivate` method. In this method, we navigate to the login page when the user authentication is not valid and this helps route to the login page based on the guard condition.

Here's the complete `authentication.guard.ts` as of now:

```

import {Injectable} from '@angular/core';
import {ActivatedRouteSnapshot, CanActivate, Router, RouterStateSnapshot} from
'@angular/router';
import {AuthenticationService} from './authentication.service';

@Injectable()
export class AuthenticationGuard implements CanActivate {

    constructor(private authService: AuthenticationService,
                private router: Router) {
    }

    canActivate(route: ActivatedRouteSnapshot, state:
RouterStateSnapshot): boolean {
        const isLoggedIn: boolean = this.authService.isAuthenticated();
        if (!isLoggedIn) {
            this.router.navigate(['/login']);
            return false;
        }
        return true;
    }
}

```

```

        if (!isLoggedIn) {
            this.router.navigateByUrl('/app-friends-login');
        }
        return isLoggedIn;
    }
}

```

3. **Adding a guard to the user module:** This guard is added in the user module. The user module is covered in the next chapter in more detail. Here, we configure a guard in the user routing module to restrict the user to accessing the user profile and friends pages:

```

import {AuthenticationGuard} from '../services/authentication.guard';

/**
 * User Module
 */
@NgModule({
    imports: [
        ...
    ],
    declarations: [
        ...
    ],
    providers: [
        AuthenticationGuard
    ]
})
export class UserModule {
}

```

4. **Adding a guard to protect the component:** As shown in the following code, we can add this guard to any component that requires this condition check. We add the guard to the user profile and user friend list component. This means that these pages are protected from illegal access.

Here is the complete `user-routing.module.ts`:

```

import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import {UserProfileComponent} from './user-profile/user-profile.component';
import {AuthenticationGuard} from '../services/authentication.guard';

const ROUTES: Routes = [
    {path: '', redirectTo: '/app-friends-userprofile', pathMatch:
    'full' , canActivate: [AuthenticationGuard]},
    {path: 'app-friends-userprofile', component: UserProfileComponent
     , canActivate: [AuthenticationGuard]}
];

@NgModule({
    imports: [
        RouterModule.forChild(ROUTES)
    ],

```

```
    exports: [
      RouterModule
    ],
    providers: [
      AuthenticationGuard
    ]
  })
export class UserRoutingModule { }
```

5. **Testing the guard:** You can test this guard by pasting the user profile URL (<http://localhost:4200/app-friends-userprofile>) into the browser, and the user is redirected to the login page for authentication.

Firebase session lifecycle

Firebase persists the user state, hence the user is always logged in even if they refresh or restart the page again, and the page is redirected to the main page instead of the login page. We will cover two navigation scenarios for a Firebase session life cycle:

- **User token exists:** In this scenario, the user token is still valid and the user is redirected to the user profile page.

`AngularFireAuth.auth` provides the `onAuthStateChanged` method to know user state information. We subscribe to this method, check for our user, and redirect them to the profile page.

Here's is a sample of `login.component.ts`:

```
import {Component} from '@angular/core';
import {User} from '../../services/user';
import {Router} from '@angular/router';
import {AuthenticationService} from '../../services/authentication.service';
import {UserService} from '../../services/user.service';
import {AngularFireAuth} from 'angularfire2/auth';

@Component({
  selector: 'app-friends-login',
  styleUrls: ['login.component.scss'],
  templateUrl: 'login.component.html',
})
export class LoginComponent {

  ...

  private user: User;

  constructor(private userService: UserService,
              private router: Router,
              private authService: AuthenticationService,
              private angularFireAuth: AngularFireAuth) {
    this.angularFireAuth.auth.onAuthStateChanged(user => {
      if (user) {
        this.getUserInfo(user.uid);
      }
    });
  }

  private navigateToUserProfile() {
    this.router.navigateByUrl('/app-friends-userprofile');
  }

  private getUserInfo(uid: string) {
```

```

        this.userService.getUser(uid).subscribe(snapshot => {
            this.user = snapshot;
            this.navigateToUserProfile();
        });
    }
}

```

- **User token expires:** In this scenario, the user token expires and the user is redirected to the login page. Normally, the user token expires on the following conditions:
 - **User clears browsing history:** The user token can expire by clearing the browser history. This will clear the token and the user is redirected to the login page.
 - **User changes the password:** When a user changes the password, the user token expires and they are directed to the login page. This scenario is covered in the next chapter.
 - **User sign out:** When a user signs out, the user token expires and they are directed to the login page. We will cover this scenario in this section.

We have implemented the sign-out functionality in the user profile page. The user profile component is covered in more detail in the next chapter. In this section, we will add only a sign-out feature.

We create a button in the user profile template. When the user clicks on the LOGOUT button, the user session is cleared and the users are redirected to the login page.

Here's is sample `user-profile.component.html`:

```

<div class="user-profile">
  <div class="user-profile-btn">
    <button type="button" (click)='onLogout()' class="btn btn-
      info">LOGOUT</button>
  </div>
</div>

```

When the user clicks on the LOGOUT button, then `onLogout()` method of `UserProfileComponent` is called and we call `signout()` in the authentication service.

Here's the sample `user-profile.component.ts` as of now:

```

import {Component, OnInit} from '@angular/core';
import {AuthenticationService} from '../../../../../services/authentication.service';
import {Router} from '@angular/router';

```

```
@Component({
  selector: 'app-friends-userprofile',
  styleUrls: ['user-profile.component.scss'],
  templateUrl: 'user-profile.component.html'
})
export class UserProfileComponent {

  constructor(private authService: AuthenticationService,
              private router: Router) {
  }

  onLogout(): void {
    this.authService.signout().then(() => {
      this.navigateToLogin();
    });
  }

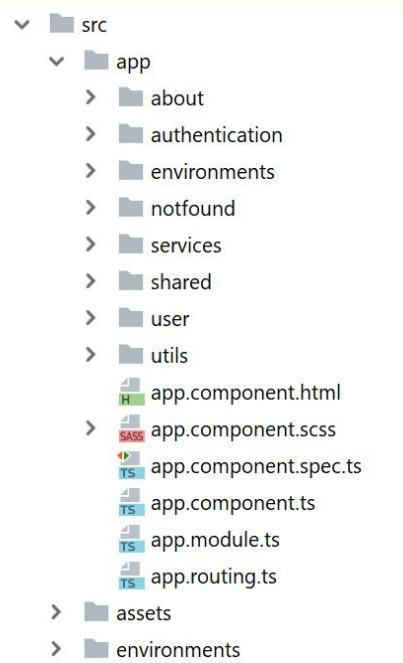
  navigateToLogin() {
    this.router.navigateByUrl('/app-friends-login');
  }
}
```

The authentication service has a sign-out functionality in `authentication.service.ts`:

```
public signout(): Promise<any> {
  return this.angularFireAuth.auth.signOut();
}
```

Our project structure as of now

We thus come to the end of our first authentication module. Our authentication module will be as follows:



Summary

In this chapter, we learned about navigation between page views. We enabled navigation in the base module. We used the router link directive to create our main navigation bar. We saw how to access component variables in the template and enabled navigation items based on the condition. We built our first guard to restrict our user to navigating to our user profile page so that only authenticated users are allowed to view the page. Finally, we covered the Firebase session life cycle and implemented navigation based on a user token. Finally, we explored the project structure for the authentication module.

In the next chapter, we will build more complex modules and explore more features of Firebase and Angular.

9-Sep-2018

Creating a User Profile Page

In this chapter, we will write a user profile component. We will cover **RxJS (ReactiveX)**, which is a popular library for asynchronous programming. In this section, we will use the `observable` of RxJS to pass the domain model from the component of the authentication module to the component of the user module. We will use this passed user model to populate the user profile component. We will edit the user data and update the Firebase authentication and database. As a part of edit, we will implement a reusable edit component, which uses bootstrap modal for taking user input. Finally, we will see the Firebase session token's life cycle when the password is changed.

In this chapter, we will cover the following topics:

- Introduction to RxJS
- Passing data between module components
- Introduction to SASS
- Creating a user profile component
- Enhancing services for the update operation
- Creating an edit dialog component
- Firebase session for the update operation

Introduction to RxJS

RxJS is a popular library for asynchronous and event-based programming. In this section, we will cover only the basics about this library so that you understand the real reason for using this library. For more details, you can refer to the official site of RxJS at <http://reactivex.io/rxjs/>.

Here are a few of this library's key terms:

- **observable**: This is a collection of values or events that can be consumed. For example, `observable` could be a collection of an array of numbers:

```
| let observable = Rx.Observable.from([1, 2, 3]);
```

- **subscription**: To read data from the `observable`, we subscribe and then the event or values are delivered using observer:

```
| let subscription = observable.subscribe(x => console.log(x));
```

- **subject**: This is an extension of the observable and is used to broadcast events or values to multiple observers. This example will be covered in our application's use case. This is a very basic understanding of the RxJS library.

Passing data between module components

We have completed our authentication module in the preceding chapter. In this section, we will cover passing data between module components, which is an important part of our application development. While implementing a web application, we always face the problem of how to pass data from one component module to another component module. Our initial thought will be to store the model in a common application class, such as a singleton class, and then retrieve it in other components. Angular provides many ways to pass the data, and we have already touched upon Angular binding. This method is useful when we have components with a parent-child relationship in the same module.



Singleton class is a software design pattern that restricts the instantiation of class to one object. This single object is available to all the components of the application.

We use `subject` of the RxJS library in the `service` class to pass the data to the component of the different module. This design helps to create an independent module.

Perform the following steps to pass the data from authentication module to user module:

1. **Store user model:** The first step is to store the data using `subject` in the `service` class. We store the user model in the user service. We use `BehaviorSubject`, which is an extension of the `subject` class to store the user model.



The behavior subject of RxJS emits the latest data to the subscriber.

We declare the `BehaviorSubject` variable in our user service, as shown in the following code snippet; the initial value is declared as `null`, and this will

be populated with the latest user model:

```
|     private subject: BehaviorSubject<User> = new BehaviorSubject(null);
```

We save the user model in `subject`. We call this method from the Login and Sign up component:

```
|     public saveUser(user: User){  
|         this.subject.next(user);  
|     }
```

2. **Create method in service class:** We can create `getSavedUser()` in the `userService` class. This method returns the `subject`, and the caller needs to subscribe or use the `getValue()` method to retrieve the saved `user` object:

```
|     public getSavedUser(): BehaviorSubject<User>{  
|         return this.subject;  
|     }
```

3. **Retrieve user model in component class:** We can retrieve the value from the `subject` using the `getValue()` method. You can also subscribe and retrieve the user model:

```
|     ngOnInit() {  
|         this.user = this.userService.getSavedUser().getValue();  
|     }
```

Here's the complete `user.service.ts` as of now:

```
import {Injectable} from '@angular/core';  
import {AngularFireDatabase} from 'angularfire2/database';  
import {User} from './user';  
import {USERS_CHILD} from './database-constants';  
import {Observable} from 'rxjs/Observable';  
import {BehaviorSubject} from 'rxjs/BehaviorSubject';  
  
/**  
 * User service  
 *  
 */  
@Injectable()  
export class UserService {  
  
    private subject: BehaviorSubject<User> = new  
    BehaviorSubject(null);  
  
    /**  
     * Constructor  
     *  
     * @param {AngularFireDatabase} fireDb provides the functionality  
     * for Firebase Database  
     */  
    constructor(private fireDb: AngularFireDatabase) {
```

```
}

public addUser(user: User): void {
    this.firebaseio.object(` ${USERS_CHILD}/${user.uid}`).set(user);
    this.saveUser(user);
}

public getUser(uid: string): Observable<User> {
    return this.firebaseio.object<User>
(` ${USERS_CHILD}/${uid}`).valueChanges();
}

public saveUser(user: User) {
    this.subject.next(user);
}

public getSavedUser(): BehaviorSubject<User> {
    return this.subject;
}

public updateEmail(user: User, newEmail: string): void {
    this.firebaseio.object(` ${USERS_CHILD}/' ${user.uid}`).update({email:
newEmail});
    this.saveUser(user);
}

public updateMobile(user: User, mobile: string): void {
    this.firebaseio.object(` ${USERS_CHILD}/' ${user.uid}`).update({mobile:
mobile});
    this.saveUser(user);
}

public updateName(user: User, name: string): void {
    this.firebaseio.object(` ${USERS_CHILD}/' ${user.uid}`).update({name:
name});
    this.saveUser(user);
}

}
```

Introduction to SASS

SASS (Systematically Awesome Style Sheets) is a CSS preprocessor, which adds more functionality to the existing CSS. It helps to add variables, nested rules, mixins, inheritance, and many more. This feature helps to organize our style sheets in a more systematic way.

SASS provides two flavors:

- SASS
- SCSS

SASS is the older of the two syntaxes, and SCSS is the more commonly used one. In this book, we have used the SCSS format. A few of the supported features are as follows:

- **Partial:** Partial is a reusable CSS element, which can be included in other SCSS files. This helps to modularize our CSS into smaller reusable elements across SCSS files. A partial filename contains a leading underscore so that the compiler knows that this is a partial file and does not convert to a CSS file, for example, `_shared.scss`. We can import a partial file into other components using the `@import` directive.

The following is an example to include a partial in other SCSS files, as follows:

```
| @import "/shared/shared";  
| .chat-message-main-container {  
| }
```

- **Extend:** This is similar to inheritance in high-level programming languages. We will write CSS properties in the common class selector and then extend this in other class selectors.

The following is an example of `@extend`:

```
| .user-profile{  
|   margin-top: 10px;  
| }
```

```
.user-profile-name{  
  @extend .user-profile;  
  border-color: green;  
}
```

- **Mixin:** This is used to group declarations that can be used throughout our application. This is similar to method signature in a class:

```
@mixin message-pointer($rotate , $skew) {  
  transform: rotate($rotate) skew($skew);  
  -moz-transform: rotate($rotate) skew($skew);  
  -ms-transform: rotate($rotate) skew($skew);  
  -o-transform: rotate($rotate) skew($skew);  
  -webkit-transform: rotate($rotate) skew($skew);  
}
```

Creating a user profile component

In this section, we will create a user profile component. After a successful login or signup, the user is directed to their profile page. This component displays the user information, such as name and email, and provides edit functionalities to change user information. The following are the steps to create a user profile component:

1. **Creating a user profile template:** The first step is to create the template for the user profile page. In this template, we will display the name, email, mobile, and password. Each of these pieces of information has an Edit button.

First, we create a `div` container that contains all the user information elements. We use `*ngIf` to check the user data:

```
<div class="user-profile" *ngIf="user">  
</div>
```

Second, we create `div` for each of the user information in `div` with `label`, `user.name`, and `Edit` buttons:

```
<div class="user-profile-name">  
  <label>Name: </label>  
  <div class="user-profile-name-value">{{user?.name}}</div>  
  <button (click)="onNameChange()" type="button" class="btn btn-default btn-sm user-profile-name-btn">  
    Edit  
  </button>  
</div>
```

Here's the complete `user-profile.component.html`:

```
<div class="user-profile" *ngIf="user">  
  <div class="person-icon">  
    <img [src]="profileImage" style="max-width: 100%; max-height: 100%;">  
  </div>  
  <div class="user-profile-name">  
    <label>Name: </label>  
    <div class="user-profile-name-value">{{user?.name}}</div>  
    <button (click)="onNameChange()" data-toggle="modal" data-target="#editModal" type="button"  
      class="btn btn-default btn-sm user-profile-name-btn">  
      Edit  
    </button>
```

```

    </div>
    <div class="user-profile-email">
        <label>Email: </label>
        <div class="user-profile-email-value">{{user?.email}}</div>
        <button (click)="onEmailChange()" data-toggle="modal" data-target="#editModal" type="button"
class="btn btn-default btn-sm">
            Edit
        </button>
    </div>
    <div class="user-profile-mobile">
        <label>Mobile: </label>
        <div class="user-profile-mobile-value">{{user?.mobile}}</div>
        <button (click)="onMobileChange()" data-toggle="modal" data-target="#editModal" type="button"
class="btn btn-default btn-sm user-profile-mobile-btn">
            Edit
        </button>
    </div>

    <div class="user-profile-password">
        <label>Password: </label>
        <div class="user-profile-password-value">****</div>
        <button (click)="onPasswordChange()" data-toggle="modal" data-target="#editModal" type="button"
class="btn btn-default btn-sm user-profile-password-btn">
            Edit
        </button>
    </div>
    <div class="user-profile-btn">
        <button type="button" (click)='onLogout()' class="btn btn-info">LOGOUT</button>
    </div>
</div>

```

User information is aligned using the style sheets. We use nesting of SCSS for class selector from the container to the child selectors:

```

.user-profile{
    width: 50%;
    margin-left: 24px;
    margin-top: 10px;
    .user-profile-name{
        text-align: left;
        margin-top: 10px;
        .user-profile-name-value{
            display: inline-block;
            margin-left: 10px;
        }
        .user-profile-name-btn{
            margin-left: 100px;
        }
    }
}

```

The following is the complete `user-profile.component.scss` code:

```

.user-profile{
    width: 50%;
    margin-left: 24px;
}

```

```

margin-top: 10px;
.user-profile-name{
  text-align: left;
  margin-top: 10px;
  .user-profile-name-value{
    display: inline-block;
    margin-left: 10px;
  }
  .user-profile-name-btn{
    margin-left: 100px;
  }
}
.user-profile-email{
  text-align: left;
  margin-top: 20px;
  .user-profile-email-value{
    display: inline-block;
    margin-left: 10px;
  }
}
.user-profile-mobile{
  text-align: left;
  margin-top: 20px;
  .user-profile-mobile-value{
    display: inline-block;
    margin-left: 10px;
  }
  .user-profile-mobile-btn{
    margin-left: 110px;
  }
}
.user-profile-password{
  text-align: left;
  margin-top: 20px;
  .user-profile-password-value{
    display: inline-block;
    margin-left: 10px;
  }
  .user-profile-password-btn{
    margin-left: 154px;
  }
}
.user-profile-btn{
  margin-top: 20px;
}

```

2. **Creating the user profile component:** We will define our logic to retrieve the user model and handle the events in the component.

The first step is to retrieve the user model from the `service` class. We implement an `OnInit` interface and override `ngOnInit` to retrieve the user model, as follows:

```

export class UserProfileComponent implements OnInit {
  private user: User;
  constructor(private authService: AuthenticationService,
             private userService: UserService,
             private router: Router) {

```

```

        }
        ngOnInit() {
            this.user = this.userService.getSavedUser().getValue();
        }
    }
}

```



onInit is a life cycle hook interface, which is managed by Angular framework. It has a *ngOnInit()* method, which is called when the component and the directive are fully initialized.

The following is the complete `user-profile.component.ts` as of now:

```

import {Component, OnInit, ViewChild} from '@angular/core';
import {AuthenticationService} from '../../../../../services/authentication.service';
import {Router} from '@angular/router';
import {User} from '../../../../../services/user';
import {UserService} from '../../../../../services/user.service';
import {EditDialogComponent} from '../../../../../edit-dialog/edit-dialog.component';
import {EditType} from '../../../../../edit-dialog/edit-details';

@Component({
    selector: 'app-friends-userprofile',
    styleUrls: ['user-profile.component.scss'],
    templateUrl: 'user-profile.component.html'
})
export class UserProfileComponent implements OnInit {

    profileImage: any = '../../../../../assets/images/person_edit.png';

    user: User;

    @ViewChild(EditDialogComponent) editDialog: EditDialogComponent;

    constructor(private authService: AuthenticationService,
                private userService: UserService,
                private router: Router) {
    }

    ngOnInit() {
        this.user = this.userService.getSavedUser().getValue();
    }

    onLogout(): void {
        this.authService.signout().then(() => {
            this.navigateToLogin();
        });
    }

    navigateToLogin() {
        this.router.navigateByUrl('/app-friends-login');
    }
}

```

Our user profile page view should be as follows:

Name: Uttam Agarwal [Edit](#)

Email: uttamagarwal.slg@gmail.com [Edit](#)

Mobile: ██████████ [Edit](#)

Password: **** [Edit](#)

[LOGOUT](#)

Enhancing services for update operation

In this section, we will enhance our existing services to provide updates for user information. As a part of this exercise, we will discuss how to update the Firebase authentication and database.

We will update the following user information:

- **Username:** This data is stored in the Firebase database, so we add new `update` API to perform this operation. We add the `updateName()` method in our user service and update the stored user data in Firebase:

```
public updateName(user: User, name: string): void {
    this.firebaseio.object(`$USERS_CHILD`/`${user.uid}`).update({name: name});
    this.saveUser(user);
}
```

- **User email:** This data is stored in the Firebase authentication and database, so we will need to update it in both places.

We will need to add a `changeEmail()` method in our authentication service:

```
public changeEmail(email: string): Promise<any> {
    return this.angularFireAuth.auth.currentUser.updateEmail(email);
}
```

Once this has been done in the authentication service, we can update a new email in our Firebase database using user services:

```
public updateEmail(user: User, newEmail: string): void {
    this.firebaseio.object(`$USERS_CHILD`/`${user.uid}`).update({email: newEmail});
    this.saveUser(user);
}
```

Now editing the mobile password is the same as the preceding code, and you can follow the following code. The updated version of `authentication.service.ts` and `user.service.ts` is as follows:

```

import {Injectable} from '@angular/core';
import {AngularFireAuth} from 'angularfire2/auth';

/**
 * Authentication service
 *
 */
@Injectable()
export class AuthenticationService {

    /**
     * Constructor
     *
     * @param {AngularFireAuth} angularFireAuth provides the
     * functionality related to authentication
     */
    constructor(private angularFireAuth: AngularFireAuth) {
    }

    public signup(email: string, password: string): Promise<any> {
        return
            this.angularFireAuth.auth.createUserWithEmailAndPassword(
                email, password);
    }

    public login(email: string, password: string): Promise<any> {
        return this.angularFireAuth.auth.signInWithEmailAndPassword(
            email, password);
    }

    public resetPassword(email: string): Promise<any> {
        return
            this.angularFireAuth.auth.sendPasswordResetEmail(email);
    }

    public isAuthenticated(): boolean {
        const user = this.angularFireAuth.auth.currentUser;
        return user ? true : false;
    }

    public signout(): Promise<any>{
        return this.angularFireAuth.auth.signOut();
    }

    public changeEmail(email: string): Promise<any> {
        return
            this.angularFireAuth.auth.currentUser.updateEmail(email);
    }

    public changePassword(password: string): Promise<any> {
        return
            this.angularFireAuth.auth.currentUser.updatePassword
            (password);
    }
}

```

The following is the updated `user.service.ts` file:

```

import {Injectable} from '@angular/core';
import {AngularFireDatabase} from 'angularfire2/database';
import {User} from './user';
import {USERS_CHILD} from './database-constants';

```

```
import {Observable} from 'rxjs/Observable';
import {BehaviorSubject} from 'rxjs/BehaviorSubject';

/**
 * User service
 *
 */
@Injectable()
export class UserService {

    private subject: BehaviorSubject<User> = new BehaviorSubject(null);

    /**
     * Constructor
     *
     * @param {AngularFireDatabase} fireDb provides the functionality
     * for Firebase Database
     */
    constructor(private fireDb: AngularFireDatabase) {
    }

    public addUser(user: User): void {
        this.fireDb.object(` ${USERS_CHILD}/ ${user.uid}`).set(user);
        this.saveUser(user);
    }

    public getUser(uid: string): Observable<User> {
        return this.fireDb.object<User>
        (` ${USERS_CHILD}/ ${uid}`).valueChanges();
    }

    public saveUser(user: User) {
        this.subject.next(user);
    }

    public getSavedUser(): BehaviorSubject<User> {
        return this.subject;
    }

    public updateEmail(user: User, newEmail: string): void {
        this.fireDb.object(` ${USERS_CHILD}/ '$ {user.uid}`).update({email:
        newEmail});
        this.saveUser(user);
    }

    public updateMobile(user: User, mobile: string): void {
        this.fireDb.object(` ${USERS_CHILD}/ '$ {user.uid}`).update({mobile:
        mobile});
        this.saveUser(user);
    }

    public updateName(user: User, name: string): void {
        this.fireDb.object(` ${USERS_CHILD}/ '$ {user.uid}`).update({name:
        name});
        this.saveUser(user);
    }

}
```

Creating an edit dialog component

The edit dialog component is used to take user input to update user information in Firebase. The same component is reused to take the information of all other user details, such as name, email, mobile, and password. This component is included in the user profile component, and the edit dialog box appears when a user clicks on the Edit button.

The following are the steps involved in creating an edit `dialog` component:

1. **Creating the edit dialog template:** The first step is to create the template for the edit dialog. This template contains a header, title for the text, and an input box.

We use bootstrap modal to create an edit dialog. It has an input box to receive the user input.

The first step is to create the `div` container with an `isVisible` condition, and this variable changes dynamically when the user clicks on the Edit button:

```
| <div *ngIf="isVisible" class="modal fade show in danger" id="editModal"  
|   role="dialog" />
```

We use `form` element to take the user input, which has a `submit` button, as follows:

```
| <div class="modal-dialog">  
|   <form name="form" (ngSubmit)="onSubmit(editFormData)"  
|     #editFormData='ngForm'>  
|   </form>  
| </div>
```

Because the preceding template is also used for different editing purposes, we will need to dynamically change the text of the header, title, and so on. We can use a one-way Angular binding to assign a variable:

```
| <p>This will change your {{bodyTitle}}</p>
```

The following is the complete `edit-dialog.component.html` file:

```

<div *ngIf="isVisible" class="modal fade in" id="editModal" role="dialog">
  <div class="modal-dialog">
    <form name="form" (ngSubmit)="onSubmit(editFormData)"
      #editFormData='ngForm'>
      <div class="modal-content">
        <div class="modal-header">
          <button type="button" class="close" data-
            dismiss="modal">&times;</button>
          <h4 class="modal-title">{{titleMessage}}</h4>
        </div>
        <div class="modal-body">
          <p>This will change your {{bodyTitle}}</p>

          <div class="form-group">
            <label for="editDetail">{{bodyLabel}}</label>
            <input type="text" class="form-control"
              name="editValue" (ngModel)="editValue"
              id="editDetail"/>
          </div>
        </div>
        <div class="modal-footer form-group">
          <button type="submit" class="btn btn-default"
            [disabled]="!editFormData.form.valid">
            Edit
          </button>
          <button type="button" class="btn btn-default"
            data-dismiss="modal"
            (click)="hide()">Close</button>
        </div>
      </form>
    </div>
  </div>

```

2. **Creating an edit dialog component:** This component takes a variable input from the user profile component when the user clicks on the Edit button. It takes a user input in the dialog box and passes it to the `EditDetails` class to update the information.

We will use builder pattern to take input variables from `UserProfileComponent`.



Builder pattern is a creational pattern that is used to create complex objects. This is basically used when a constructor in a class accepts many parameters. This reduces the complexity of the constructor.

In our case, we will need parameters to dynamically change the title, `label`. We will create multiple methods for each variable input—for example, for title, we will create a `setTitle()` method and return `this`, which is the instance of the class. This helps in chaining the method calls

in a single line:

```
public setTitle(title: string): EditDialogComponent {
    this.titleMessage = title;
    return this;
}
```

We will then need to toggle the `isVisible` variables using the `show` and `hide` methods, as follows:

```
public show() {
    this.isVisible = true;
}

public hide() {
    this.isVisible = false;
}
```

Here's the complete `edit-dialog.component.ts` file as of now:

```
import {Component, ViewChild} from '@angular/core';
import {AuthenticationService} from '../services/authentication.service';
import {UserService} from '../services/user.service';
import {User} from '../services/user';
import {EditDetails, EditType} from './edit-details';

@Component({
    selector: 'app-edit-dialog',
    templateUrl: './edit-dialog.component.html',
})
export class EditDialogComponent {
    isVisible: boolean;

    titleMessage: string;

    bodyTitle: string;

    bodyLabel: string;

    editType: EditType;

    editDetails: EditDetails;

    constructor(private authService: AuthenticationService,
                private userService: UserService) {
        this.editDetails = new EditDetails(authService, userService);
    }

    public setTitle(title: string): EditDialogComponent {
        this.titleMessage = title;
        return this;
    }

    public setBodyTitle(bodyTitle: string): EditDialogComponent {
        this.bodyTitle = bodyTitle;
        return this;
    }

    public setBodyLabel(bodyLabel: string): EditDialogComponent {
```

```

        this.bodyLabel = bodyLabel;
        return this;
    }

    public setEditType(editType: EditType): EditDialogComponent {
        this.editType = editType;
        return this;
    }

    public show() {
        this.isVisible = true;
    }

    public hide() {
        this.isVisible = false;
    }

    private onSubmit(editFormData): void {
        this.editDetails.edit(this.editType,
            editFormData.value.editValue);
    }
}

```

3. **Creating an update operation:** When a user provides the new data for updating and clicks on `submit`, the `onSubmit()` method is called. For each of the update operations, we will call `edit()` of the `EditDetails` class.

Here's the complete `edit-details.ts` file as of now:

```

import {AuthenticationService} from '../services/authentication.service';
import {UserService} from '../services/user.service';
import {User} from '../services/user';

export enum EditType {
    NAME,
    EMAIL,
    MOBILE,
    PASSWORD
}

export class EditDetails {

    constructor(private authService: AuthenticationService,
                private userService: UserService) {
    }

    public edit(editType: EditType, value: string) {
        switch (editType) {
            case EditType.NAME:
                this.editName(value);
                break;

            case EditType.EMAIL:
                this.editEmail(value);
                break;

            case EditType.MOBILE:
                this.editMobile(value);
                break;
        }
    }
}

```

```

        case EditType.PASSWORD:
            this.editPassword(value);
            break;
    }
}

private editName(name: string) {
    const user: User = this.userService.getSavedUser().getValue();
    user.name = name;
    this.userService.updateName(user, name);
    alert('Name changed successfully');
}

private editEmail(newEmail: string) {
    this.authService.changeEmail(newEmail).then(() => {
        const user =
            this.userService.getSavedUser().getValue();
        user.email = newEmail;
        this.userService.updateEmail(user, newEmail);
        alert('Email changed successfully');
    }).catch(function (error) {
        const errorMessage = error.message;
        alert(errorMessage);
    });
}

private editMobile(mobile: string) {
    const user: User = this.userService.getSavedUser().getValue();
    user.mobile = mobile;
    this.userService.updateMobile(user, mobile);
    alert('Mobile changed successfully');
}

private editPassword(value: string) {
    const newPassword: string = value;
    this.authService.changePassword(newPassword).then(() => {
        alert('Password changed successfully');
    }).catch(function (error) {
        const errorMessage = error.message;
        alert(errorMessage);
    });
}
}

```

4. **Using the edit dialog component:** Finally, we will use the edit dialog component in our user profile component. The first step will be to include this component in the `user-profile.component.html`, as follows:

```

<div class="user-profile" *ngIf="user">
    ...
</div>
<app-edit-dialog></app-edit-dialog>

```

The second step will be to initialize the edit dialog component in the `user-profile.component.ts`, as follows:

```

export class UserProfileComponent implements OnInit {
    @ViewChild(EditDialogComponent) editDialog: EditDialogComponent;
}

```

```
|     }  
| }
```

When the user clicks on any of the Edit buttons, we will need to initialize the variables and call the `show()` method:

```
| onNameChange() {  
|   this.editDialog.setTitle('Do you want to edit name?')  
|     .setBodyTitle('name')  
|     .setBodyLabel('Enter new name')  
|     .setEditType(EditType.NAME)  
|     .show();  
| }
```

The following are other methods for other update operations in the `user-profile.component.ts`:

```
| onEmailChange() {  
|   this.editDialog.setTitle('Do you want to edit email?')  
|     .setBodyTitle('email')  
|     .setBodyLabel('Enter new email')  
|     .setEditType(EditType.EMAIL)  
|     .show();  
| }  
  
| onMobileChange() {  
|   this.editDialog.setTitle('Do you want to edit mobile?')  
|     .setBodyTitle('mobile')  
|     .setBodyLabel('Enter new mobile')  
|     .setEditType(EditType.MOBILE)  
|     .show();  
| }  
  
| onPasswordChange() {  
|   this.editDialog.setTitle('Do you want to edit password?')  
|     .setBodyTitle('password')  
|     .setBodyLabel('Enter new password')  
|     .setEditType(EditType.PASSWORD)  
|     .show();  
| }
```

Finally, we configure the edit component in our user module, as follows:

```
| import {NgModule} from '@angular/core';  
| import {EditDialogComponent} from '../edit-dialog/edit-dialog.component';  
  
| /**  
|  * User Module  
|  */  
| @NgModule({  
|   imports: [  
|     ...  
|   ],  
|   declarations: [  
|     ...  
|     EditDialogComponent  
|   ]  
| })
```

```
|   export class UserModule {  
|     }  
|   }
```

Now, when the user clicks on the Edit button, the following edit dialog will appear:

Do you want to edit name? ×

This will change your name

Enter new name

Submit

Close

Firebase session for the update operation

When a user edits their email and password, Firebase asks for the user to log in again. Firebase throws an error when we call the `updatePassword()` method in `AngularFireAuth`, and this error is added for security reasons.

This operation is sensitive and requires recent authentication. Log in again before retrying this request.

The preceding message is shown in the alert dialog in our application, and to edit the email or password, we need to log out and refresh the session immediately and then perform the action.

The best way to enhance the user experience is to ask the user to log out using the popup and refresh the token. We are not implementing this behavior as a part of this book, so you can take this as an exercise.

Summary

Congratulations on completing this chapter! This is one of the most advanced chapters. We covered many important concepts related to programming paradigm. We discussed passing the data from one module component to another module component. As a part of this, we developed two independent modules using the least dependencies. We covered RxJS library. We developed an edit component and included it in the user profile component. Finally, we covered a Firebase security feature, which expires the session while editing sensitive information, such as an email or a password.

In the next chapter, we will enhance our friends application to add the user's friends features. We will also retrieve the list of friends and display them in the list. We will add pagination to navigate to the friend list.

17-Sep-2018

Creating a User's Friend List

In this chapter, we will move toward the more advanced features in Angular and Firebase. We will retrieve our user's friend list using a Firebase list. We will display the friend list in a card component provided by Bootstrap. We will implement the pagination concept using a Firebase filter. Finally, we will discuss Angular pipes.

In this chapter, we will cover the following topics:

- Creating user's friend template
- Creating the friend's service
- Creating the friends component
- Creating our first date pipe

Creating user's friend template

In this section, we will cover a slightly more complex template using a Bootstrap card component. We will retrieve the friend's list of a defined size and display the user's friend list in a card item. We will call the Firebase API to get three items and loop the friend's list using an `*ngFor` directive.

Card is a flexible and extensible container. It has an option to display the header, footer, title, and so on. We will use the following properties of the card component:

- `card-img-top`: This is used to display the friend's image on the top.
- `card-title`: This is used to display the friend's name.
- `card-text`: This is used to display their email and phone number.
- `card-footer`: This is used to display the date using a custom pipe. We will implement custom pipe in a later part of this chapter.

```
<div *ngFor="let friend of friends" class="card">
  
  <div class="card-block">
    <h4 class="card-title">{{friend.name}}</h4>
    <p class="card-text">{{friend.email}} | {{friend.mobile}}</p>
  </div>
  <div class="card-footer">
    <small class="text-muted">Friends from {{friend.time | friendsdate}}</small>
  </div>
</div>
```

After we display the first page, we need the left and right icon to scroll to the next and previous page. These icons will be visible based on the total items in the list, and `isLeftVisible` will be set from the component class:

```
| <div *ngIf="isLeftVisible" (click)="onLeft()" class="left"></div>
```

Here's the complete `user-friends.component.html` file:

```
<div class="main_container">
  <div *ngIf="friends" class="content_container">
    <div *ngIf="isLeftVisible" (click)="onLeft()" class="left">
      
    </div>
    <div class="card-deck list">
```

```

<div *ngFor="let friend of friends" class="card">
  
  <div class="card-block">
    <h4 class="card-title">{{friend.name}}</h4>
    <p class="card-text">{{friend.email}} | {{friend.mobile}}</p>
  </div>
  <div class="card-footer">
    <small class="text-muted">Friends from
      {{friend.time | friendsdate}}</small>
  </div>
</div>
<div *ngIf="isRightVisible" (click)="onRight()" class="right">
  
</div>
</div>
<div *ngIf="!friends || friends.length === 0"
  class="no_info_container">
  <h1>No friends in your list</h1>
</div>
</div>

```

We assign class selectors to apply styles to the elements. In the friends list page, we align the elements horizontally using `display:inline`. Also, the left icon, card list, and right icon are displayed one after another, so we use `float:left`.

Here's the complete `user-friends.component.scss` as of now:

```

.main_container {
  margin-top: 10px;
  margin-left: 80px;
  .content_container {
    display: inline;
    .list {
      float: left;
      .card-img-top {
        height: 180px;
        width: 260px;
        background-image:
          url(' ../../../../assets/images/person.png');
      }
    }
    .left {
      float: left;
      margin-top: 140px;
    }
    .right {
      float: left;
      margin-top: 140px;
    }
  }
}

```

Creating the friend's service

We will introduce one more service as part of our friend's component. This will fetch friend's details from the Firebase. In this section, we will cover the following topics:

- Creating a Firebase node in our database
- Implementing the `Friend` class
- Implementing the friend's service

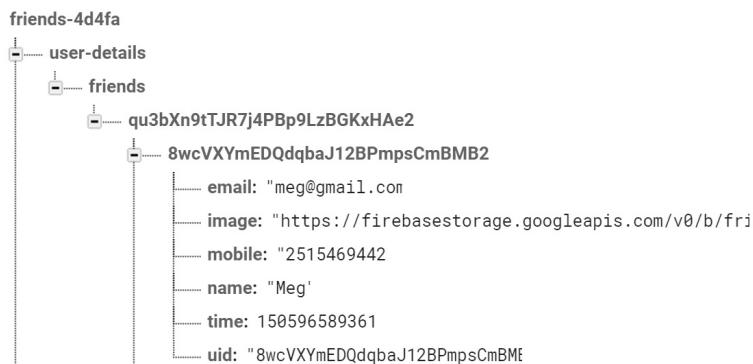
Creating a Firebase node in our database

Now, we have prefilled the friend's details in the Firebase as shown in the next image. We introduced a separate node called `user-details`. This will store all the user's information, and we don't need to query the user node for more information, as this increases the query performance.

The following are a few key observations for this instance of Firebase:

- We have not implemented the add friend feature; hence we will add the friend's information manually.
- We use the UID relationship to list the friends of a user. In this case, the UID `qu3bXn9tTJR7j4PBp9LzBGKxHAe2` is the user ID and the other UID `—8wcVXYmEDQdqbaJ12BPmpsCmBMB2`—is the friend's ID that is generated when the friend signs up to the application.
- In Firebase, we repeat a lot of data. This is a common pattern while organizing data in a NoSQL database, as this avoids multiple hits to the database. Although it increases the write time, it helps our app to scale when reading data. It prevents large queries that slow down our database and nested nodes that take longer to read.

The friends node in the Firebase database is as follows:



Implementing the Friend model class

We will implement the `Friend` model class to map an array of friend's JSON object from Firebase. This class is similar to the `User` model class, and it is a good practice to segregate the responsibility in a separate class.

This class has `name`, `email`, `mobile`, `uid`, `time`, and `image` properties. The `time` property is used to show the duration of friendship and is stored in milliseconds. We will need to convert the time in milliseconds to a human-readable date format using an Angular pipe.

The following is the complete `friend.ts` file:

```
export class Friend {  
  name: string;  
  mobile: string;  
  email: string;  
  uid: string;  
  time: string;  
  image: string;  
  
  constructor(name: string,  
             mobile: string,  
             email: string,  
             uid: string,  
             time: string,  
             image: string) {  
    this.name = name;  
    this.mobile = mobile;  
    this.email = email;  
    this.uid = uid;  
    this.time = time;  
    this.image = image;  
  }  
}
```

Implementing the friend's service

As a part of this service, we will need to retrieve a list of friends.

AngularFireDatabase provides a list API to retrieve the list of friends. This service consists of the following three methods to give us complete pagination functionalities:

- **Retrieving the first page:** The `getFirstPage()` method accepts `uid` and `pageSize` as parameters. These are used to retrieve the first `pageSize` data from Firebase. We pass the `pageSize` in the second argument of the query function:

```
getFirstPage(uid: string, pageSize: number): Observable<Friend[]>
{
  return this.firebaseio.list<Friend>
(`${USER_DETAILS_CHILD}/${FRIENDS_CHILD}/${uid}`,
  ref => ref.limitToFirst(pageSize)
).valueChanges();
}
```

- **Retrieving the next page:** `loadNextPage()` accepts `uid`, `friendKey`, and `pageSize`. `uid` and `friendKey` are used to set the queries. This means that they retrieve the next `pageSize` data from the last retrieved `friendKey` data:

```
loadNextPage(uid: string, friendKey: string, pageSize: number):
Observable<Friend[]> {
  return this.firebaseio.list<Friend>
(`${USER_DETAILS_CHILD}/${FRIENDS_CHILD}/${uid}`,
  ref => ref.orderByKey().startAt(friendKey)
    .limitToFirst(pageSize + 1)
).valueChanges();
}
```

- **Retrieving the previous page:** `loadPreviousPage()` method accepts `uid`, `friendKey`, and `pageSize`. The last two parameters are used to retrieve the previous `pageSize` data from the starting `friendKey` element:

```
loadPreviousPage(uid: string, friendKey: string, pageSize: number):
Observable<Friend[]> {
  return this.firebaseio.list<Friend>
(`${USER_DETAILS_CHILD}/${FRIENDS_CHILD}/${uid}`,
  ref => ref.orderByKey().startAt(friendKey)
    .limitToLast(pageSize + 1)
).valueChanges();
```

```
|     }
```

Here's the complete `friends.service.ts`:

```
import {Injectable} from '@angular/core';
import {AngularFireDatabase} from 'angularfire2/database';
import {Observable} from 'rxjs/Observable';
import 'rxjs/add/operator/map';
import {Friend} from './friend';
import {FRIENDS_CHILD, USER_DETAILS_CHILD} from './database-constants';

/**
 * Friends service
 *
 */
@Injectable()
export class FriendsService {

    /**
     * Constructor
     *
     * @param {AngularFireDatabase} fireDb provides
     *         the functionality related to authentication
     */
    constructor(private fireDb: AngularFireDatabase) {
    }

    getFirstPage(uid: string, pageSize: number): Observable<Friend[]>
    {
        return this.fireDb.list<Friend>
            (`${USER_DETAILS_CHILD}/${FRIENDS_CHILD}/${uid}`,
                ref => ref.limitToFirst(pageSize)
            ).valueChanges();
    }

    loadNextPage(uid: string, friendKey: string, pageSize: number):
    Observable<Friend[]> {
        return this.fireDb.list<Friend>
            (`${USER_DETAILS_CHILD}/${FRIENDS_CHILD}/${uid}`,
                ref =>
                    ref.orderByKey().startAt(friendKey)
                    .limitToFirst(pageSize + 1)
            ).valueChanges();
    }

    loadPreviousPage(uid: string, friendKey: string, pageSize: number):
    Observable<Friend[]> {
        return this.fireDb.list<Friend>
            (`${USER_DETAILS_CHILD}/${FRIENDS_CHILD}/${uid}`,
                ref =>
                    ref.orderByKey().startAt(friendKey)
                    .limitToLast(pageSize + 1)
            ).valueChanges();
    }
}
```

Creating a friend's component

This is the main controller of our friend page. In this component, we will need to manage the navigation and visibility of our next and previous icon. We will cover the following two main things in this section:

- Displaying the next and previous page
- Visibility of icons

In order to display the next and previous page, we have already created the API required to display the friend's information. We have extended the `onInit` interface and called `getFirstPage()` on `ngOnInit` with `uid` and `pageSize` as filter parameters, as follows:

```
ngOnInit() {
    this.user = this.userService.getSavedUser().getValue();
    this.friendService.getFirstPage(this.user.getUid(), this.pageSize)
        .subscribe(friends => {
            this.friends = friends;
            ...
        });
}
```

The `ngOnInit()` method runs when the page is loaded.



Consequently, we will retrieve the next and previous page using the API in the friend's service, as follows; the only difference is that we will also pass the friend `uid` so that we can retrieve the next page size data, starting from the last retrieved item:

```
next() {
    this.friendService.loadNextPage(this.user.getUid(),
        this.friends[this.friends.length - 1].getUid(),
        this.pageSize
    ).subscribe(friends => {
        this.friends = friends;
        ...
    });
}
```

Now, we will move on to our next part. We will need to take care of the next and previous icons for which we will need the total number of friends. In our previous discussion, we obtained the size as `pageSize`. In order to solve this problem, we have to create `friendcount` in our Firebase user node. Whenever we add a friend, we increase the count. We have added this property in our `User` class; all the other parts remain the same:

```
| private friendcount: number
```

We will then retrieve the total item count in `ngOnInit`, as follows:

```
ngOnInit() {
    this.user = this.userService.getSavedUser().getValue();
    this.totalCount = this.user.getFriendcount();
    this.friendService.getFirstPage(this.user.getUid(), this.pageSize)
        .subscribe(friends => {
            ...
            let count: number = this.friends.length;
            this.currentCount = count;
            this.leftArrowVisible();
            this.rightArrowVisible();
        });
}
```

Next, we initialize the current count to the item retrieved and then call the visibility based on the total and current count:

```
leftArrowVisible(): void{
    this.isLeftVisible = this.currentCount > this.pageSize;
}

rightArrowVisible(): void{
    this.isRightVisible = this.totalCount > this.currentCount;
}
```

Here's the complete `user-friends.component.ts` file:

```
import {Component, OnInit} from '@angular/core';
import {FriendsService} from '../../../../../services/friends.service';
import {Friend} from '../../../../../services/friend';
import {UserService} from '../../../../../services/user.service';
import {User} from '../../../../../services/user';
import 'firebase/storage';
import {Router} from '@angular/router';

@Component({
    selector: 'app-friends-userfriends',
    styleUrls: ['user-friends.component.scss'],
    templateUrl: 'user-friends.component.html'
})
export class UserFriendsComponent implements OnInit {

    friends: Friend[];
```

```
totalCount: number;

pageSize = 3;

currentCount = 0;

previousCount = 0;

isLeftVisible = false;

isRightVisible = true;

user: User;

constructor(private friendService: FriendsService,
            private userService: UserService) {
}

ngOnInit() {
  this.user = this.userService.getSavedUser().getValue();
  this.totalCount = this.user.friendcount;
  this.friendService.getFirstPage(this.user.uid, this.pageSize)
    .subscribe(friends => {
      this.friends = friends;
      const count: number = this.friends.length;
      this.currentCount = count;
      this.leftArrowVisible();
      this.rightArrowVisible();
    });
}

onLeft(): void {
  this.previous();
}

onRight(): void {
  this.next();
}

next() {
  this.friendService.loadNextPage(this.user.uid,
    this.friends[this.friends.length - 1].uid,
    this.pageSize
  ).subscribe(friends => {
    this.friends = friends;
    const count: number = this.friends.length;
    this.previousCount = count - 1;
    this.currentCount += this.previousCount;
    this.leftArrowVisible();
    this.rightArrowVisible();
  });
}

previous() {
  this.friendService.loadPreviousPage(this.user.uid,
    this.friends[0].uid,
    this.pageSize
  ).subscribe(friends => {
    this.friends = friends;
    const count: number = this.friends.length;
  });
}
```

```

        this.currentCount -= this.previousCount;
        this.leftArrowVisible();
        this.rightArrowVisible();
    });

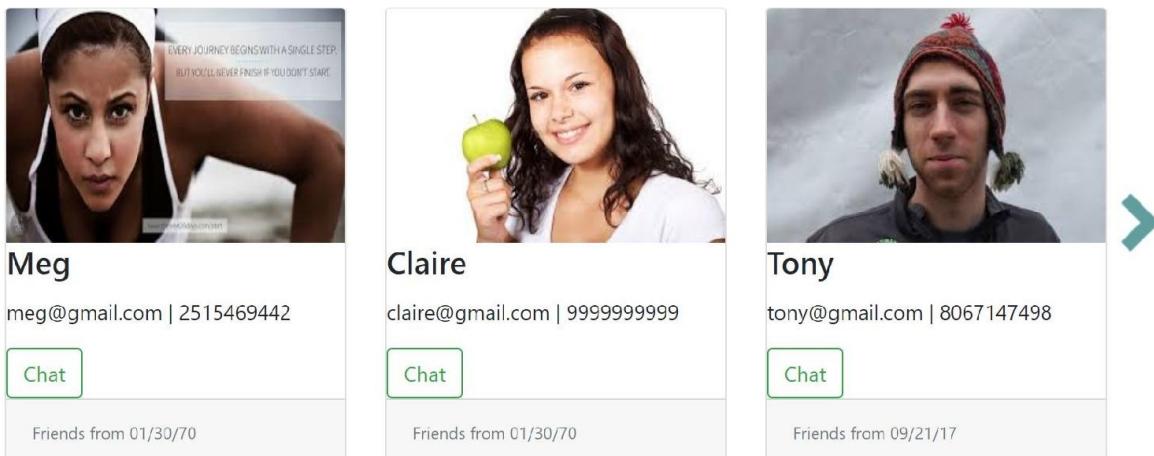
}

leftArrowVisible(): void {
    this.isLeftVisible = this.currentCount > this.pageSize;
}

rightArrowVisible(): void {
    this.isRightVisible = this.totalCount > this.currentCount;
}
}

```

The user's friends page displays three friends with navigation enabled:



Creating our first date pipe

A Pipe takes an input as its data and transforms it into the desired output. It is used to transform data into a usable form.

We use pipe to transform time into a human-friendly date format. To create the pipe, we implement the `PipeTransform` interface and override the `transform` method. In this method, we take the date in milliseconds and use [the moment library](#) to transform the time to a particular date format. We provide the selector name, which is used in the HTML tag with an input data:

```
import * as moment from 'moment';
import {Pipe, PipeTransform} from '@angular/core';

/**
 * It is used to format the date
 */
@Pipe({
  name: 'friendsdate'
})
export class FriendsDatePipe implements PipeTransform {
  transform(dateInMillis: string) {
    if (dateInMillis === '0' || dateInMillis === '-1') {
      return 'Invalid Date';
    }
    return moment(dateInMillis, 'x').format('MM/DD/YY');
  }
}
```



Moment is a JavaScript library used to format, manipulate, or parse dates.

After creating the pipe, we add it in our `user` module:

```
@NgModule({
  imports: [
    ...
  ],
  declarations: [
    ...
    FriendsDatePipe
  ]
})
export class UserModule { }
```

Finally, we provide the `friendsdate` pipe to our `time` value from the `friend` object in

a template, as follows:

```
| <div class="card-footer">
|   <small class="text-muted">Friends from {{friend.getTime() | friendsdate}}</small>
| </div>
```

Summary

In this chapter, we covered a lot of important concepts. We covered the card component, which is used in most applications nowadays. We decorated our view with styles and also created a new service. We discussed the Firebase list and then provided the filter options. This implemented pagination for our friend's list. Finally, we discussed the Angular pipe, which we used to convert time into a human-friendly date format.

In the next chapter, we will cover Firebase storage and learn how to store a profile image and then retrieve it.

19-Sep-2018

Exploring Firebase Storage

In this chapter, we will move forward to explore the other features of Firebase. Nowadays, images, audios, and videos have become an integral part of any website development. Keeping this in mind, Firebase has introduced **storage**.

We will take a look at how to upload a profile picture using the Firebase storage API. We will upload some random pictures using the Firebase portal and download the uploaded images using API to be shown in our friends list. We will then take a look at how to delete the files in Firebase storage. Finally, we will cover error handling.

In this chapter, we will cover the following topics:

- Introducing Firebase storage
- Configuring Firebase storage
- Uploading the profile picture
- Downloading friends images
- Deleting the profile image
- Handling errors in Firebase storage

Introducing Firebase storage

Firebase storage provides the flexibility for application developers to store various content types in the storage. It stores images, videos, and audios. The contents are stored in the Google Cloud storage bucket, and they can be accessed from both Firebase and Google Cloud.

Firebase storage integrates with Firebase authentication and allows strong security. We can also apply a declarative security model to control access to the content. We will study this in more detail in a later section.

Firebase storage provides the following key features:

- **Scale:** It is backed by Google Cloud Storage and can scale to petabytes of size. You can read more about Google Cloud Storage at <https://cloud.google.com/storage/>.
- **Security:** Each of the uploaded files can be secured using storage security rules.

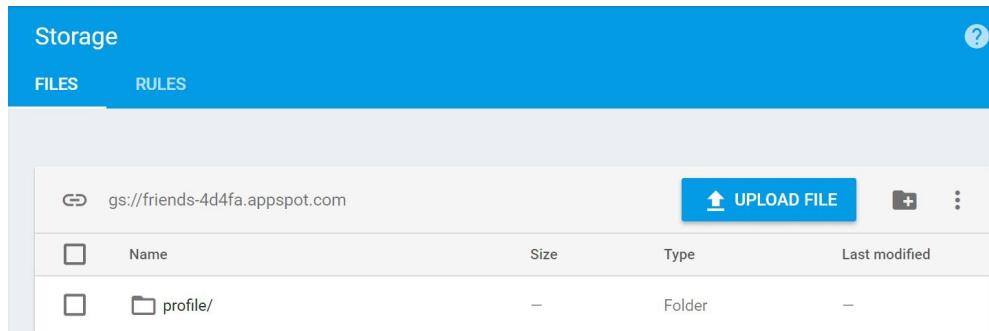
The default security rules for Firebase storage are as follows:

```
service
  firebase.storage {
    match /b/{bucket}/o {
      match /{allPaths=**} {
        allow read, write: if request.auth!=null;
      }
    }
  }
```

- **Network:** Firebase storage automatically takes care of the network problem during uploads and downloads of files.

Configuring Firebase storage

To configure Firebase storage in our application, we need the storage bucket URL. This can be found within the Files tab under Storage in the header, as shown in the following screenshot; in our case, the storage bucket URL of our application friends is `gs://friends-4d4fa.appspot.com`:



The screenshot shows the Firebase Storage interface. At the top, there's a blue header bar with the word "Storage". Below it, a navigation bar has "FILES" selected. In the main area, there's a table with one row. The first column contains a link icon and the URL "gs://friends-4d4fa.appspot.com". To the right of the URL are three buttons: "UPLOAD FILE" (with an upward arrow icon), a folder icon, and a more options icon. The table has four columns: "Name", "Size", "Type", and "Last modified". The single row shows a folder named "profile/" with a size of "-", a type of "Folder", and last modified date "-".

Here's the modified `environment.ts` file with the storage bucket URL:

```
export const environment = {
  production: false,
  firebase: {
    apiKey: 'XXXX',
    authDomain: 'friends-4d4fa.firebaseio.com',
    databaseURL: 'https://friends-4d4fa.firebaseio.com',
    projectId: 'friends-4d4fa',
    storageBucket: 'friends-4d4fa.appspot.com',
    messagingSenderId: '321535044959'
  }
};
```

Uploading the profile picture

In this section, we will cover how to upload a file to Firebase storage. In the user profile page, we will add the user profile picture on the top of the page.

The steps involved in uploading and showing the user profile image are as follows:

1. **Adding HTML properties in a user profile template:** We provide an `input` tag to take a user-selected image from the file chooser. Normally, in the `input` tag with a file type, we have a button to choose the file; however, in this case, we require the user to click on the default image. We have hidden the button in the `input` tag using the built in styles, as follows:

```
<div class="person-icon">
  <img [src]="profileImage" style="max-width: 100%; max-height: 100%;">
  <input (change)="onPersonEdit($event)" required accept=".jpg" type="file" style="opacity: 0.0; position: absolute; top:120px; left: 30px; bottom: 0; right:0; width: 200px; height:200px;" />
</div>
```

2. **Adding a default image in style sheets:** Initially, we show the default image by declaring the default image path in the `user-profile.component.ts` file, as follows:

```
export class UserProfileComponent implements OnInit {
  profileImage: any = '../../../../../assets/images/person_edit.png';
  ...
}
```

The following is the modified `user-profile.component.html` file:

```
<div class="user-profile" *ngIf="user">
  <div class="person-icon">
    <img [src]="profileImage" style="max-width: 100%; max-height: 100%;">
    <input (change)="onPersonEdit($event)" required accept=".jpg" type="file" style="opacity: 0.0; position: absolute; top:120px; left: 30px; bottom: 0; right:0; width: 200px; height:200px;" />
  </div>
  ...
</div>
<app-edit-dialog></app-edit-dialog>
```

Here's the modified `user-profile.component.scss` file:

```
.user-profile{
  width: 50%;
  margin-left: 24px;
  margin-top: 10px;
  .person-icon{
    width: 200px;
    height: 200px;
  }
  ...
}
```

3. **Handling a click event in the user profile component:** We will implement the `onPersonEdit()` method, which accepts `event` as its parameter. As you can see from the following code, we need to retrieve the selected files from the `event` object and pass them to `userService`:

```
onPersonEdit(event) {
  const selectedFiles: FileList = event.target.files;
  const file = selectedFiles.item(0);
  this.userService.addProfileImage(this.user, file);
}
```

4. **Adding a method in user service:** In `user.service.ts`, we initialize the instance of Firebase storage in the constructor, as follows:

```
@Injectable()
export class UserService {

  private fbStorage: any;

  private basePath = '/profile';

  /**
   * Constructor
   *
   * @param {AngularFireDatabase} fireDb provides the functionality
   * related to authentication
   */
  constructor(private fireDb: AngularFireDatabase) {
    this.fbStorage = fireDb.app.storage();
  }
  ...
}
```

Now we implement the `addProfileImage()` method in our user service.

1. First, we create the path for storing the image in Firebase storage:

```
`${this.basePath}/${file.name}`
```

2. Second, we call the `put()` method from the Firebase storage

reference, as follows:

```
|     this.fbStorage.ref(` ${this.basePath}/${file.name}`).put(file)
```

After a successful upload, we save the download URL in the user node of Firebase and refresh the cache user object:

```
public addProfileImage(user: User, file: File) {
    this.fbStorage.ref(` ${this.basePath}/${file.name}`).put(file).then(
        snapshot => {
            const imageUrl: string = snapshot.downloadURL;
            this.firebaseio.object(` ${USERS_CHILD}/ ${user.uid}`).update({image: imageUrl});
            user.image = imageUrl;
            this.saveUser(user);

        }).catch((error) => {
            ...
        });
}
```

5. **Refreshing the user profile image:** After a successful upload, we need to update the image in our user profile page. We subscribe to `user` observable in the user profile component and update the profile image, as follows:

```
ngOnInit() {
    this.user = this.userService.getSavedUser().getValue();
    this.userService.getSavedUser().subscribe(
        (user) => {
            if (this.user.image) {
                this.profileImage = this.user.image;
            }
        }
    );
}
```

After a successful refresh, the user profile page will look as follows:



Name: Uttam Agarwal

[Edit](#)

Email: uttamagarwal.slg@gmail.com

[Edit](#)

Mobile: 9999999999

[Edit](#)

Password: ****

[Edit](#)

[LOGOUT](#)

Downloading friends images

In the user profile page, we have uploaded the profile image in Firebase storage. We have also stored a downloadable URL in the user node in Firebase, which can be accessed by friends using the UID. After fetching the friends list, we have to call another Firebase API to get the downloadable URL from our user node.

The following is the downloadable URL:

```
| https://firebasestorage.googleapis.com/v0/b/friends-  
| 4d4fa.appspot.com/o/profile%2Fclaire.jpg?alt=media&token=e00012af-c71c-48eb-92bc-  
| 4a0c9f989cbd
```

In HTML, images are defined with an `` tag. The `src` attribute specifies the URL address of the image, as follows:

```
| 
```

In `user-friends.component.html`, we add the default image with a downloadable URL:

```
| 
```

In `user-friends.component.scss`, we use `background-image` and also add `width` and `height` so that the image fits into the card layout, like this:

```
.main_container {  
    margin-top: 10px;  
    margin-left: 80px;  
    .content_container {  
        display: inline;  
        .list {  
            float: left;  
            .card-img-top {  
                height: 180px;  
                width: 260px;  
                background-image:  
                    url(' ../../assets/images/person.png');  
            }  
        }  
        .left {  
            float: left;  
            margin-top: 140px;  
        }  
        .right {  
            float: left;  
            margin-top: 140px;  
        }  
    }  
}
```

```
| } }
```

Firebase provides an API to get the downloadable URL using Firebase storage:

```
public getDownloadURL(user: User, file: File) {
  this.fbStorage.child('images/claire.jpg').getDownloadURL().then((url) => {
    // assign to the img src
  }).catch((error) => {
    // Handle any errors
  });
}
```

Deleting the profile images

Firebase storage provides an API to delete the file from Firebase. The `delete` operation is similar to other Firebase storage APIs. We have not implemented this use case in our application; however, you may require this concept in your application:

```
public deleteFile() {
    this.fbStorage.child('images/claire.jpg').delete().then(function()
{
    // File deleted successfully
}).catch((error) => {
    // Handle any errors
});
}
```

Handling errors in Firebase storage

Firebase storage throws an error based on different conditions, as follows:

- **storage/unknown:** This may occur because of any unknown error. This is similar to the default condition in the `switch...case` statement.
- **storage/object_not_found:** This occurs when the file/image reference is not available in the Firebase storage location.
- **storage/bucket_not_found:** This error occurs when the Firebase bucket is not configured.
- **storage/project_not_found:** This error occurs when the Firebase project has not configured Firebase storage.
- **storage/quota_exceeded:** This error occurs when the free tier plan expires and is asked to be upgraded to the paid plan.
- **storage/unauthenticated:** This error occurs when the user is not authenticated but is still able to access files and images in Firebase storage.
- **storage/unauthorized:** This error occurs when an unauthorized user accesses the files/images in Firebase storage.
- **storage/retry_limit_exceeded:** This error occurs when a user exceeds the retry limit because of a slow network, or because of no network.
- **storage/invalid_checksum:** This error occurs when the checksum on the client does not match that of the server's.
- **storage/canceled:** This error occurs when a user intervenes an upload or a download operation.
- **storage/invalid_event_name:** This error occurs when an invalid event name is provided to the Firebase storage API. The correct events are *running*, *progress*, and *pause*.
- **storage/invalid-argument:** The Firebase storage `put()` method accepts file, `Blob`, and `Uint8` as parameters. This error occurs when we pass a wrong argument.

When we encounter an error within our application, we implement the `then()` method of Promise and retrieve the error message and show it in alert dialog.

Here is a modified `addProfileImage()` method in the `user.service.ts` class:

```
public addProfileImage(user: User, file: File) {
  this.fbStorage.ref(`.${this.basePath}/${file.name}`).put(file).then(
    snapshot => {
      ...
    })
  .catch((error) => {
    const errorMessage = error.message;
    alert(errorMessage);
  });
}
```

Summary

In this chapter, we discussed Firebase storage. We uploaded the profile picture to Firebase storage and stored the downloadable URL in the user node in our database. We showed the image in the `img` tag of HTML, as this helps in downloading the images from Firebase storage. We covered Firebase security where a user needs to be properly authenticated to access the images/files in Firebase storage. Finally, we discussed error handling in Firebase storage.

In the next chapter, we will cover a more interesting and exciting part of our application. We will create a chat application and we will also cover how Firebase supports real-time updates.

21-Sep-2018

Creating a Chat Component

In this chapter, we will create our chat application in the existing application and take a look at real-time message updates using the Firebase database. We will explain the chat feature in this and the next chapter.

Since we have already created components in the preceding chapter, we will design a more complex component involving multiple components, in this chapter. As per the general rule, we will create this as a module, so we will have the main component as a chat component, and this will contain a message list component, form component, and message component. We will explore more ways of data binding while implementing the chat feature. We will write more complex SCSS in this chapter. We believe that, if you follow this chapter properly, most of the Angular stuff will be clearer to you, and you'll be able to build a more complex component yourself.

In this chapter, we will cover the following topics:

- Creating a chat module
- Creating a color variable
- Creating a chat component
- Creating a chat message list component
- Creating a mixin for the message view
- Creating a chat message component
- Creating a chat message form component

Creating a chat module

The first step in creating a module is to define the routes and include them in the chat module. In the chat routing module, we create chat routes and configure them in RouterModules.

The following is the complete `chat-routing.module.ts` file:

```
import {NgModule} from '@angular/core';
import {RouterModule, Routes} from '@angular/router';
import {ChatComponent} from './chat.component';

export const ROUTES: Routes = [
  {path: 'app-friends-chat/:id', component: ChatComponent}
];

/**
 * Chat Routing Module
 */
@NgModule({
  imports: [
    RouterModule.forChild(ROUTES)
  ],
  exports: [
    RouterModule
  ]
})
export class ChatRoutingModule { }
```

The chat module contains a declaration of all the components, modules, and services. In chat features, we have the following four components:

- **Chat component:** This is the main component, and it encapsulates the message list and message form components.
- **Chat message list component:** This is a message list, which displays the messages in the list. It calls the message component for populating the message in the textbox.
- **Chat message form component:** This is a form that takes a message as user input and adds the message in the Firebase database.
- **Chat message component:** This component displays the user-typed message and the date when the message was posted.

The following is the complete `chat.module.ts` as of now:

```
| import {NgModule} from '@angular/core';
```

```

import {CommonModule} from '@angular/common';
import {FormsModule} from '@angular/forms';
import {BrowserModule} from '@angular/platform-browser';
import {UserService} from '../services/user.service';
import {ChatMessageComponent} from './chat-message/chat-message.component';
import {ChatMessageListComponent} from './chat-message-list/chat-message-
list.component';
import {ChatMessageFormComponent} from './chat-message-form/chat-message-
form.component';
import {ChatComponent} from './chat.component';
import {ChatRoutingModule} from './chat-routing.module';

/**
 * Chat Module
 */
@NgModule({
  imports: [
    CommonModule,
    BrowserModule,
    FormsModule,
    ChatRoutingModule
  ],
  declarations: [
    ChatMessageComponent,
    ChatMessageListComponent,
    ChatMessageFormComponent,
    ChatComponent
  ],
  providers: [
    UserService
  ]
})
export class ChatModule {
}

```

Finally, we will include the chat module in the app module, as shown in the following code; here's the modified `app.module.ts` as of now:

```

...
@NgModule({
  declarations: [
    ...
  ],
  imports: [
    ...
    ChatModule,
  ],
  providers: [
    ...
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}

```

Presently, our chat module is part of the main app module. Now, we will implement the components.

Creating a color variable

In this section, we discuss variable support in SCSS. In CSS, we need to declare the color code for each property, and we don't have any mechanism to reuse the same color code in another CSS property:

```
#messages {  
    background-color: #F2F2F2 !important;  
}
```

In our application, we use variables and partials to reuse the same color across the whole application. We declare all colors in a color file using variables as shown in the subsequent code. This file is known as a partial in SCSS and normally declared with an underscore.

The following is the complete `_colors.scss` as of now:

```
$mercury_solid: #e5e5e5;  
$sushi: #8BC34A;  
$concrete_solid: #F2F2F2;  
$iron: #E1E2E3;  
$pickled_bluewood: #2d384a;
```

First, we import partials into another SCSS file and then use the variables to access the color. In the following example, we use the `$concrete_solid` variable to reuse the color:

```
@import "..../shared/colors";  
  
.chat-message-list-main-container {  
  
    #messages {  
        background-color: $concrete_solid !important;  
    }  
}
```

SCSS variables help us to centralize all the colors in a single file so that, when we change the color combination in one file, this reflects in our entire application.

Creating a chat component

Chat component is the main container, and it contains a message list component and message form component.

It uses Bootstrap component to create a message list column view.

```
div class="chat-main-container">
  <div class="main_container">
    <div class="col-md-8 col-md-offset-2">
      ...
    </div>
  </div>
```

Chat template encapsulates the chat message list and chat message form references.

The following is the complete `chat.component.html` as of now:

```
div class="chat-main-container">
  <div class="main_container">
    <div class="col-md-8 col-md-offset-2">
      <app-chat-message-list [friendUid]="uid">
      </app-chat-message-list>
      <app-chat-message-form [friendUid]="uid">
      </app-chat-message-form>
    </div>
  </div>
</div>
```

We align the main container to the middle of the page using `margin-top` and `margin-left`, as shown in the following complete `chat.component.scss` as of now:

```
.chat-main-container {
  margin-top: 10px;
  margin-left: 80px;

  p {
    font-size: 10px;
  }
}
```

Chat component declares the template, style sheet, and selector. Here's the complete `chat.component.ts` as of now:

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

```
| @Component({  
|   selector: 'app-friends-chat',  
|   styleUrls: ['chat.component.scss'],  
|   templateUrl: 'chat.component.html',  
| })  
| export class ChatComponent {  
| }
```

Chat component provides the layout for other child components.

Creating a chat message list component

The chat message list component displays the message text in a list layout. It calls the message component to populate the message text data and time.

First, we create the list in container `div` and tag the message list `div` with `#scrollContainer`, as this helps scroll the list to the bottom of the chat window when a new message arrives. We read this tag in the component using the `@ViewChild` annotation:

```
<div class="chat-message-list-main-container">
  <div #scrollContainer class="message-list-container"
    id="messages">
    ...
  </div>
</div>
```

Finally, we include the chat message selector and loop the messages. The following is the complete `chat-message-list.component.html` as of now:

```
<div class="chat-message-list-main-container">
  <div #scrollContainer class="message-list-container" id="messages">
    <app-chat-message *ngFor="let message of messages;">
      [message]="message">
    </app-chat-message>
  </div>
</div>
```

In the following HTML `div` tag, we include two selectors in our template—we add a class selector and an ID selector:

```
| <div #scrollContainer class="message-list-container" id="messages"></div>
```

We read the ID selector using a hash followed by name of the selector in the SCSS file to style `background-color`:

```
| #messages {
|   background-color: $concrete_solid !important;
| }
```

We use the `box-shadow` and `border-radius` properties to give elevated looks for our

list container:

```
.message-list-container {  
    ...  
    box-shadow: inset 0 3px 6px rgba(0, 0, 0, .05);  
    border-radius: 8px;  
}
```

Here's the complete `chat-message-list.component.scss` as of now:

```
.chat-message-list-main-container {  
  
    #messages {  
        background-color: #F2F2F2 !important;  
    }  
  
    .message-list-container {  
        position: relative;  
        padding: 15px 15px 15px;  
        border-color: #e5e5e5 #eee #eee;  
        border-style: solid;  
        border-width: 1px 0;  
        background-color: #E1E2E3;  
        box-shadow: inset 0 3px 6px rgba(0, 0, 0, .05);  
        height: 60vh;  
        overflow-y: scroll;  
        background-color: #2d384a !important;  
        border-radius: 8px;  
    }  
  
    p {  
        font-size: 10px;  
    }  
}
```

In the chat message list component, we use `@ViewChild` to read the scroll container, as follows:

```
| @ViewChild('scrollContainer') private scrollContainer: ElementRef
```

We then implement `AfterViewChecked` in our component to handle the bottom scroll.



The life cycle method is called whenever the view of the component is checked during change detection.

```
interface AfterViewChecked{  
    ngAfterViewChecked: void  
}
```

We override the life cycle method and, after each new message, we scroll the message list to the bottom of the last message. Also, we detect component

changes using the `ChangeDetectorRef` class. This is required, as we need to force Angular to check for changes to the component because the scroll event runs outside Angular's zone:

```
ngAfterViewChecked() {
  this.scrollToBottom();
  this.cdRef.detectChanges();
}

scrollToBottom(): void {
  try {
    this.scrollContainer.nativeElement.scrollTop =
      this.scrollContainer.nativeElement.scrollHeight;
  } catch(err) {
    console.log("Error");
  }
}
```

Here's the complete `chat-message-list.component.ts` as of now:

```
import {AfterViewChecked, ChangeDetectorRef, Component, ElementRef, Input, OnInit, ViewChild} from '@angular/core';

@Component({
  selector: 'app-chat-message-list',
  styleUrls: ['chat-message-list.component.scss'],
  templateUrl: 'chat-message-list.component.html'
})
export class ChatMessageListComponent implements OnInit , AfterViewChecked{
  @ViewChild('scrollContainer') private scrollContainer: ElementRef;

  constructor(private messageService: MessagingService,
              private userService: UserService,
              private cdRef: ChangeDetectorRef) {}

  ngAfterViewChecked() {
    this.scrollToBottom();
    this.cdRef.detectChanges();
  }

  scrollToBottom(): void {
    try {
      this.scrollContainer.nativeElement.scrollTop =
        this.scrollContainer.nativeElement.scrollHeight;
    } catch(err) {
      console.log("Error");
    }
  }
}
```

Creating a mixin for the message view

In this section, we will cover SCSS mixins. This feature provides the flexibility to group the CSS properties, and we can reuse this mixin across our application. Just like the class method, we can also provide parameters to make the mixin more flexible.

We will use this mixin in our application to add message pointer to our chat features. We will declare mixin by prefixing the method name with the `@mixin` keyword and add parameters such as `$rotate` and `$skew`.

We created mixins for our chat messages in `_shared.scss`:

```
@mixin message-pointer($rotate, $skew) {
  transform: rotate($rotate) skew($skew);
  -moz-transform: rotate($rotate) skew($skew);
  -ms-transform: rotate($rotate) skew($skew);
  -o-transform: rotate($rotate) skew($skew);
  -webkit-transform: rotate($rotate) skew($skew);
}
```

We use this mixin in our message SCSS. First, we need to import the shared SCSS file in our message file and then we use `@include` to call the mixin with the parameters.

The following is the sample `chat-message.component.scss`:

```
@import "../../shared/shared";
.chat-message-main-container {
  .message-bubble::before {
    ...
    @include message-pointer(29deg, -35deg);
    ...
  }
}
```

Creating a chat message component

The message component is the message text container. It displays the message and time. A typical chat has a bubble view layout. We design this view for our chat features. We declare the following three class variables, which we use in the SCSS file:

- `message-bubble`: This selector gives the message bubble view layout
- `class.sender`: This aligns all the messages from the sender on the left side of the container
- `class.receiver`: This aligns all the messages from the receiver on the right side of the container

Here's the complete `chat-message.component.html` as of now:

```
<div class="chat-message-main-container">
  <div class="message-bubble" [class.receiver]="isReceiver(message)"
       [class.sender]="isSender(message)">
    <p>{{ message.message }}</p>
    <div class="timestamp">
      {{ message.timestamp | date:"MM/dd/yy hh:mm a" }}
    </div>
  </div>
</div>
```

We use a class selector and provide a style to the message box. This consists of the following two major parts:

- **Message box**: This gives the shadow effect to the view
- **Message pointer**: This gives a pointer to the message box

Here's the complete `chat-message.component.scss` as of now:

```
@import "../../shared/shared";
.chat-message-main-container {
  .message-bubble {
    background-color: #ffffff;
    border-radius: 5px;
    box-shadow: 0 0 6px #B2B2B2;
    display: inline-block;
    padding: 10px 18px;
    position: relative;
    vertical-align: top;
```

```

        width: 400px;
    }

.message-bubble::before {
    background-color: #ffffff;
    content: "\00a0";
    display: block;
    height: 16px;
    position: absolute;
    top: 11px;
    @include message-pointer(29deg , -35deg);
    width: 20px;
}

.sender {
    display: inherit;
    margin: 5px 45px 5px 20px;
}

.sender::before {
    box-shadow: -2px 2px 2px 0 rgba(178, 178, 178, .4);
    left: -9px;
}

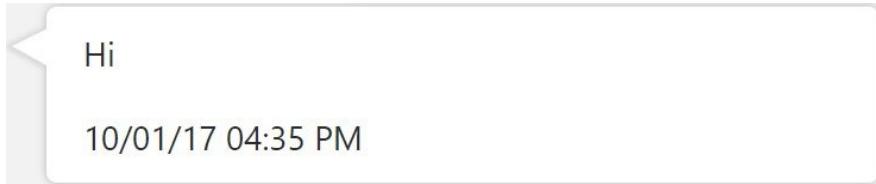
.receiver {
    display: inherit;
    margin: 5px 20px 5px 170px;
}

.receiver::before {
    box-shadow: 2px -2px 2px 0 rgba(178, 178, 178, .4);
    right: -9px;
}

}

```

The message box looks as follows:



Finally, we write the event method in our component. We retrieve the UID from the saved user in our service and write the logic to recognize the receiver and sender using the UID.

Here's the complete `chat-message.component.ts` as of now:

```

import {ChangeDetectionStrategy, Component, Input, OnInit} from '@angular/core';
import {UserService} from '../../../../../services/user.service';
import {Message} from '../../../../../services/message';

@Component({
    selector: 'app-chat-message',

```

```
    styleUrls: ['chat-message.component.scss'],
    templateUrl: 'chat-message.component.html',
    changeDetection: ChangeDetectionStrategy.OnPush
})
export class ChatMessageComponent implements OnInit {

  @Input() message: Message;
  uid: string;

  constructor(private userService: UserService) {
  }

  ngOnInit() {
    this.uid = this.userService.getSavedUser().getValue().uid;
  }

  isReceiver(message: Message) {
    return this.uid === message.receiverUid;
  }

  isSender(message: Message) {
    return this.uid === message.senderUid;
  }
}
```

Creating a chat message form component

In the chat message form component, we implement the message form to send the message to Firebase and update the list with a new message.

For these actions, we will require the following two elements:

- **Input with a text area:** The input text allows a user to type their message. We use `(key.enter)` in the input text to handle the keyboard's *Enter* key, and this calls the `sendMessage()` method.
- **Send button:** This calls the `sendMessage()` method and updates the Firebase database.

Here's the complete `chat-message-form.component.html` as of now:

```
<div class="chat-message-form-main-container">
  <div class="chat-message-form-container">
    <input type="textarea" placeholder="Type a message"
      class="message-text" [(ngModel)]="newMessage"
      (keyup.enter)="sendMessage()">
    <button (click)="sendMessage()"
      class="btn btn-outline-success my-2 my-sm-0"
      type="submit">SEND</button>
  </div>
</div>
```

We use the `chat-message-form-container` class selector to style the border with `border-radius` and the `message-text` to style input text-related properties.

Here's the complete `chat-message-form.component.scss` file as of now:

```
@import "../../shared/colors";
.chat-message-form-main-container {
  .chat-message-form-container {
    padding: 9px 50px;
    margin-bottom: 14px;
    background-color: #f7f7f9;
    border: 1px solid #e1e1e8;
    border-radius: 4px;

    .message-text {
      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: #ffffff;  
border: 1px solid $sushi;  
border-radius: 4px;  
width: 100%;  
}  
}
```

In the chat message form component, we retrieve the UID from the user object saved in the user service.

The following is the complete `chat-message-form.component.ts` file as of now:

```
import { Component, OnInit, Input } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import {MessagingService} from '../../../../../services/messaging.service';
import {Message} from '../../../../../services/message';
import {UserService} from '../../../../../services/user.service';

@Component({
  selector: 'chat-message-form',
  styleUrls: ['chat-message-form.component.scss'],
  templateUrl: 'chat-message-form.component.html'
})
export class ChatMessageFormComponent implements OnInit {

  uid: string;

  newMessage: string;

  constructor(private messageService: MessagingService,
              private userService: UserService) { }

  ngOnInit() {
    this.uid = this.userService.getSavedUser().getValue().getUid();
  }

  sendMessage() {
  }
}
```

Finally, our chat feature will look as follows:

Hi

10/01/17 04:35 PM

Hello

10/01/17 04:56 PM

Are you developing friend application

10/05/17 04:11 PM

Type a message

SEND

Summary

In this chapter, we designed a more complex UI component using multiple components. We implemented a chat module and integrated it with the main application. We covered new SCSS features such as variables, partials, and mixins. This really helped us to modularize our code and showed how reusability can be achieved in SCSS. We divided a big chat component into smaller components and then integrated these small components.

In the next chapter, we will integrate our component with the services. We will design our Firebase database for our chat application. Then, we will subscribe the real-time database and get instant updates.

Connecting Chat Components with Firebase Database

In this chapter, we will integrate our chat components with the new messaging service. We will discuss a new method to pass our data to our chat module using route parameters. Once a friend's UID is passed from the user friend list component, then we will pass this friend's UID to a different chat component, as this data is required in the message list and message form components. We will also design a database for our chat application, as a proper design avoids data duplication. Once the database is ready, we will query the data from the message service and integrate the message service with the chat components.

In this chapter, we will cover the following topics:

- Passing data using route parameters
- Passing friend data to different chat components
- Designing a Firebase database for chat
- Creating the messaging service
- Integrating service to chat components

Passing data using route parameters

When a user clicks on the Chat button, we pass the friend UID to the chat component using route parameters.

In the [Chapter 5, Creating a User Profile Page](#), we passed the user data using `BehaviorSubject` in the RxJS library. In this section, we will use the route parameter of the router link to pass the friend's UID. We perform the following three steps to pass the data using route parameters:

1. Adding a route parameter ID
2. Linking a route to the parameter
3. Reading the parameter

Adding a route parameter ID

In the first step mentioned in the preceding list, we need to add the friend's UID parameter to the route link. We add an ID parameter to the path element as shown here:

```
| export const ROUTES: Routes = [  
|   { path: 'app-friends-chat/:id', component: ChatComponent }  
| ];
```

When a user clicks on the Chat button, this ID is added to URL as

<http://localhost:4200/friends-chat/8wcVXYmEDQdqbaJ12BPmpsCmBMB2>.

Linking a route to the parameter

In the second step mentioned in the preceding list, we need to link the friend's UID to the router link. The following are the two methods to achieve it:

- **Router link directive:** We can directly link the parameter ID using the `routerLink` directive, as follows:

```
| <div *ngFor="let friend of friends" class="card" [routerLink]=["/app-friends-  
|   chat' , friend.getUid()]"></div>
```

- **Programmatically using router:** When a user clicks on the Chat button, we pass the UID to the method parameter and use a router to pass the data.

We add a Chat button to the friends list of our user. We modify the template of our user's friends, as follows:

```
| <div *ngFor="let friend of friends" class="card">  
|   ...  
|     <button (click)="onChat(friend.uid)" class="btn btn-outline-  
|       success my-2 my-sm-0" type="submit">Chat</button>  
|   </div>  
|   ...  
</div>
```

When a user clicks on the Chat button, the `onChat()` method is called with a UID as its parameter. Finally, we call the `router` method to pass the `id` as the route parameter:

```
| onChat(id: string): void {  
|   this.router.navigate(['/app-friends-chat' , id]);  
| }
```

Reading the parameter

We use `ActivatedRoute` to read the parameter ID. This component provides a collection of parameters to read the ID. We subscribe to route parameters and store the subscription object in member variables and unsubscribe it in the Angular life cycle `ngOnDestroy()` method.



OnDestroy is an Angular life cycle hook interface. It has the `ngOnDestroy()` method and is called for cleanup logic when the components are destroyed.

The following is the complete `chat.component.ts` file as of now:

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

@Component({
  selector: 'friends-chat',
  styleUrls: ['chat.component.scss'],
  templateUrl: 'chat.component.html',
})
export class ChatComponent {

  uid: string;
  private sub: any;

  constructor(private route: ActivatedRoute) {
  }

  ngOnInit() {
    this.sub = this.route.params.subscribe(params => {
      this.uid = params['id'];
    });
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }
}
```

We will cover how to pass UID data to other chat components in the next section.

Passing friend data to different chat components

Once we have the friend's UID in our chat module, we can pass the friend's UID to our message list and the message form component using Angular data binding. We perform the following two steps to pass the data to two chat components:

1. **Declaring the input variables:** We declare the input variables using the Angular `@Input` annotation in the two chat components. The following snippet shows the changes for the message list and message form components.

The following snippet shows changes for `chat-message-list.component.ts`:

```
export class ChatMessageListComponent implements OnInit , AfterViewChecked{  
    @Input() friendUid: string;  
}  
|
```

The following snippet shows changes for `chat-message-form.component.ts` file:

```
export class ChatMessageFormComponent implements OnInit {  
    @Input() friendUid: string;  
}  
|
```

2. **Binding the data to input:** We can bind the input variable `friendUid` with `uid`, which was passed from the user module:

```
<div class="col-md-8 col-md-offset-2">  
    <app-chat-message-list [friendUid]="uid">  
    </app-chat-message-list>  
    <app-chat-message-form [friendUid]="uid">  
    </app-chat-message-form>  
</div>  
|
```

We will use this friend's UID to read or update the Firebase database.

Designing a Firebase database for chat

Designing the Firebase database is the most crucial part in writing the chat feature.

Chat involves communication between two persons. It has a sender and receiver, and both can see the same text messages. We have to associate two users with the same messages, and this involves mainly two steps:

1. **Creating a new chat ID:** The first step is to associate the unique message ID for both persons. We create a unique message ID and associate the two users using their UID. As shown in the following screenshot, we associate the "-KvMW57CNfA40GJNDF-F" key with both UIDs.



When the user clicks on the Chat button in their friend's page, we will check for the key and freshly create the ID, as follows:

```
|   const key = this.firebaseio.createPushId();
```

AngularFireDatabase has a `createPushId()` method to create the unique ID.

Here is the sample code from `messaging.service.ts`:

```
|   freshlyCreateChatIDEntry(uid: string, friendUid: string): string {  
|     const key = this.firebaseio.createPushId();
```

```

    this.firebaseio.object(`${USER_DETAILS_CHILD}/${CHAT_MESSAGES_CHILD}/${uid}/${friendKey}`);
    this.firebaseio.object(`${USER_DETAILS_CHILD}/${CHAT_MESSAGES_CHILD}/${friendUid}/${key}`);
    return key;
}

```

2. **Associating messages with the key:** Once we create the key for the users, we create the new node in our Firebase database, associate this key as a node, and push the messages in this node so that both the users have access. In the database schema, we will create receiver and sender nodes with UIDs so that we know who has sent the message and align the messages in the chat window based on this condition



We store the key in the member variable of the service so that we use this key to push a new message to our Firebase database.

The following is the sample code from `messaging.service.ts`:

```

createNewMessage(newMessage: Message) {
  const messageKey = this.firebaseio.createPushId();
  this.firebaseio.object(`${MESSAGE_DETAILS_CHILD}/${this.key}/
  ${messageKey}`).set(newMessage).catch(error => {
    console.log(error);
  });
}

```

Creating a messaging service

The first step in creating the messaging service is to define a data model. We create a message model with properties; the message model consists of the following four properties:

- **Message:** This contains the message in a string type.
- **Sender UID:** The sender UID is used to know the identity of the sender of the particular message and write logic to display the text message on the left panel of the chat window.
- **Receiver UID:** The receiver UID is used to know the identity of the receiver of the particular message and write logic to display the text message on the right panel of the chat window.
- **Timestamp:** This property is used to display the date and time of the message sent.

Here's the `message.ts` as of now:

```
export class Message {  
  message: string;  
  senderUid: string;  
  receiverUid: string;  
  timestamp: number;  
  
  constructor(message: string,  
             senderUid: string,  
             receiverUid: string,  
             timestamp: number) {  
    this.message = message;  
    this.senderUid = senderUid;  
    this.receiverUid = receiverUid;  
    this.timestamp = timestamp;  
  }  
}
```

As a part of the chat feature, we declare a constant to know the node of the Firebase database, as shown in the next code.

Here's the `database-constants.ts` file as of now:

```

| export const USER_DETAILS_CHILD = 'user-details';
| export const CHAT_MESSAGES_CHILD = "chat_messages";
| export const MESSAGE_DETAILS_CHILD = "message_details";

```

The core part of our messaging feature is the service. This service is responsible for creating the chat ID, pushing the new message, and getting messages. As part of this application, we have introduced the following four APIs:

- `isMessagePresent()`: When a user clicks on the Chat button, we check whether a message key is present and store the key in the member variable in this service. We use this key to push any new messages in the Firebase database.
- `freshlyCreateChatIdEntry()`: We call this API when the user starts a fresh communication with a friend. Then, we create the key and store it in the Firebase database.
- `getMessages()`: This API is used to subscribe to all the messages in the conversation, and we also get updates when a new message is received.
- `createNewMessage()`: When the user clicks on the SEND button, we call this API to store the new message in the Firebase database. The new message consists of message text, sender UID, receiver UID, and timestamp.

Here's the `messaging.service.ts` file as of now:

```

import {Injectable} from '@angular/core';
import {AngularFireDatabase} from 'angularfire2/database';
import {CHAT_MESSAGES_CHILD, MESSAGE_DETAILS_CHILD, USER_DETAILS_CHILD} from
'./database-constants';
import {FirebaseApp} from 'angularfire2';
import 'firebase/storage';
import {Observable} from 'rxjs/Observable';
import {Message} from './message';

/**
 * Messaging service
 */
@Injectable()
export class MessagingService {

  key: string;

  /**
   * Constructor
   *
   * @param {AngularFireDatabase} fireDb provides the functionality
   * related to authentication
   */
  constructor(private fireDb: AngularFireDatabase) {
  }

  isMessagePresent(uid: string, friendUid: string): Observable<any> {
    return
  }
}

```

```

    this.firebaseio.object(`${USER_DETAILS_CHILD}/${CHAT_MESSAGES_CHILD}/
    ${uid}/${friendUid}`).valueChanges();
}

createNewMessage(newMessage: Message) {
  const messageKey = this.firebaseio.createPushId();
  this.firebaseio.object(`${MESSAGE_DETAILS_CHILD}/${this.key}/
  ${messageKey}`).set(newMessage).catch(error => {
    console.log(error);
  });
}

freshlyCreateChatIDEntry(uid: string, friendUid: string): string {
  const key = this.firebaseio.createPushId();
  this.firebaseio.object(`${USER_DETAILS_CHILD}/
  ${CHAT_MESSAGES_CHILD}/${uid}/${friendUid}`).set({key: key});
  this.firebaseio.object(`${USER_DETAILS_CHILD}/
  ${CHAT_MESSAGES_CHILD}/${friendUid}/${uid}`).set({key: key});
  return key;
}

getMessages(key: string): Observable<Message[]> {
  return this.firebaseio.list<Message>
  (`${MESSAGE_DETAILS_CHILD}/${key}`).valueChanges();
}

setKey(key: string) {
  this.key = key;
}
}

```

In the next section, we will integrate this service to our chat components.

Integrating our service to chat components

Finally, we will integrate the messaging service to the components. We will cover the following three use cases:

- **Checking the message key for a new chat:** When the user initiates a conversation with their friend, we call the `isMessagePresent()` API of the messaging service. For a fresh conversation, this chat key will not be present, and we need to create the new key and use the same key for subsequent communication:

```
ngOnInit() {
    this.user = this.userService.getSavedUser().getValue();
    this.messageService.isMessagePresent(this.user.getUid(),
    this.friendUid).subscribe(snapshot => {
        let snapshotValue = snapshot.val();
        let friend: Friend;
        if (snapshotValue == null) {
            console.log("Message is empty");
            this.key =
                this.messageService.freshlyCreateChatIDEntry
                (this.user.getUid(), this.friendUid);
        } else {
            this.key = snapshotValue.key;
        }
        this.messageService.setKey(this.key);
        this.subscribeMessages();
    });
}
```

- **Subscribing to the message list:** We will call the `getMessages()` method to get a message list observable. We will then subscribe to this observable to get a message list and then assign it to `messages` member variables:

```
subscribeMessages() {
    this.messageService.getMessages(this.key)
    .subscribe(
        messages => {
            this.messages = messages;
        });
}
```

Here's the `chat-message-list.component.ts` file as of now:

```
import {AfterViewChecked, ChangeDetectorRef, Component, ElementRef, Input, OnInit, ViewChild} from '@angular/core';
import {MessagingService} from '../../../../../services/messaging.service';
import {Message} from '../../../../../services/message';
import {UserService} from '../../../../../services/user.service';
import {User} from '../../../../../services/user';

@Component({
  selector: 'app-chat-message-list',
  styleUrls: ['chat-message-list.component.scss'],
  templateUrl: 'chat-message-list.component.html'
})
export class ChatMessageListComponent implements OnInit, AfterViewChecked {
  @Input() friendUid: string;
  private user: User;
  messages: Message[];
  key: string;
  @ViewChild('scrollContainer') private scrollContainer: ElementRef;

  constructor(private messageService: MessagingService,
              private userService: UserService,
              private cdRef: ChangeDetectorRef) {}

  ngOnInit() {
    this.user = this.userService.getSavedUser().getValue();
    this.messageService.isMessagePresent(this.user.uid, this.friendUid).subscribe(snapshot => {
      if (snapshot == null) {
        console.log('Message is empty');
        this.key =
          this.messageService.freshlyCreateChatIDEntry
          (this.user.uid, this.friendUid);
      } else {
        this.key = snapshot.key;
      }
      this.messageService.setKey(this.key);
      this.subscribeMessages();
    });
  }

  ngAfterViewChecked() {
    this.scrollToBottom();
    this.cdRef.detectChanges();
  }

  scrollToBottom(): void {
    try {
      this.scrollContainer.nativeElement.scrollTop =
        this.scrollContainer.nativeElement.scrollHeight;
    } catch (err) {
      console.log('Error');
    }
  }

  subscribeMessages() {
    this.messageService.getMessages(this.key)
      .subscribe(
        messages => {
          this.messages = messages;
        }
      );
  }
}
```

```

        });
    }
}

```

- **Sending messages to Firebase database:** When a user types the message and clicks on the SEND button, we create the new message object and call the `createNewMessage()` method in the messaging service and this takes care of sending message to Firebase database.

```

sendMessage() {
  const message: Message = new Message(this.newMessage,
  this.uid, this.friendUid, Date.now());
  this.messageService.createNewMessage(message);
}

```

Here's the `chat-message-form.component.ts` file as of now:

```

import {Component, Input, OnInit} from '@angular/core';
import {MessagingService} from '../../services/messaging.service';
import {Message} from '../../../../../services/message';
import {UserService} from '../../../../../services/user.service';

@Component({
  selector: 'app-chat-message-form',
  styleUrls: ['chat-message-form.component.scss'],
  templateUrl: 'chat-message-form.component.html'
})
export class ChatMessageFormComponent implements OnInit {
  @Input() friendUid: string;

  uid: string;
  newMessage: string;

  constructor(private messageService: MessagingService,
              private userService: UserService) {}

  ngOnInit() {
    this.uid = this.userService.getSavedUser().getValue().uid;
  }

  sendMessage() {
    const message: Message = new Message(this.newMessage,
    this.uid, this.friendUid, Date.now());
    this.messageService.createNewMessage(message);
  }
}

```

Finally, we have created a fully functional chat feature for our friends application.

Summary

We have now come to the end of the chat feature in our friends application. In this chapter, we covered the route parameter and used this to pass the friend's UID to our chat module. We passed this UID to different chat components using the `@Input` binding. We then discussed how to design Firebase database for our chat feature. We used Firebase database API in the service and created four APIs to perform different operations in our chat feature. Finally, we integrated this service with the chat components.

In the next chapter, we will discuss unit testing in Angular. We will also discuss the Jasmine framework and use this framework to unit-test our application.

Unit Testing Our Application

In this chapter, we will cover Angular testing in detail. We will start with a basic introduction to Angular testing and learn about tools and technologies used for unit testing. We will write unit tests for our login component using Angular testing framework and configure a dependent module. We will also unit test our user service. As part of our testing, we will create stubs for dependent services or components so that we can focus only on the class unit under test. We will unit test our component and service using Angular framework so that dependent modules are initialized. We will test an Angular pipe in isolation so that the pipe is initialized directly using a `new` keyword. Finally, we will take a look at the code coverage for our code.

In this chapter, we will cover the following topics:

- Introduction to Angular testing
- Unit testing an Angular component
- Unit testing Angular services
- Unit testing Angular pipe
- Introduction to code coverage

Introduction to Angular testing

Unit testing is an important part of a software development life cycle. The benefits of unit testing are as follows:

- It helps to align our implementation with the design
- It helps in protecting our application from regression
- Refactoring becomes easier if we have good test cases

Angular provides various tools and technologies to test our application. As a part of our unit testing, we will use the following technologies:

- **Jasmine**: This provides the basic skeleton to write our unit test. It comes with a HTML test runner and runs on a browser.
- **Angular testing framework**: This comes along with Angular framework and helps to create a test environment for Angular code under test. It also gives us access to DOM elements.
- **Karma**: We use the Karma tool to run our application. We run our unit tests using the following command:

```
| $ng test
```

We can write two types of Angular test:

- **Unit testing with Angular test framework**: We will use Angular test framework to write unit tests for our component and services. This will create a test environment and provide us access to various elements of the Angular framework.
- **Isolated unit test**: We can write independent unit tests without Angular dependency. This kind of unit test is really useful for testing services and pipes. In this chapter, we will test our date pipe this way.

It is good to note that it is always better to stick to isolated unit tests when you can, and to try to write as few integrated and end-





to-end tests whenever possible, because isolated unit tests are the easiest to maintain.

Unit testing an Angular component

In this section, we will write our first Angular test for the login component. The steps involved in writing test cases are as follows:

1. **Identifying the class under test:** The first step in writing unit test cases is to identify the dependencies. The login component constructor shows all the dependencies and is dependent on `UserService`, `Router`, `AuthenticationService`, and `AngularFireAuth`:

```
constructor(private userService: UserService,
            private router: Router,
            private authService: AuthenticationService,
            private angularFireAuth: AngularFireAuth)
```

2. **Creating all the mock or stubs classes:** Once we identify all the dependencies, we need to eliminate these external dependencies and concentrate only on the component class under test. So we create mock or stub classes to eliminate these dependencies.

In the login component, we use user service to retrieve user information using the `getUser()` method so we create `UserServiceStub` with the `getUser()` method, which returns mock user encapsulates in an `observable` object; we create a test data class for the mock user, that contains user details, as follows:

```
class UserServiceStub {  
  getUser(): Observable<User> {  
    return Observable.of(mockUserJSON);  
  }  
}
```

Here's a sample `user-test-data.ts` file:

```
export const mockUserJSON = {  
  email: 'user@gmail.com',  
  friendcount: 0,  
  image: '',  
  mobile: '99999999999',  
  name: 'User',  
  uid: 'xxxx'  
};
```

We use an authentication service in the login component to log in and reset the password so we create an `AuthenticationServiceStub` class with an empty `login()` and `resetPassword()` methods:

```
class AuthenticationServiceStub {  
    login(email: string, password: string) {}  
    resetPassword(email: string) {}  
}
```

The `AngularFireAuth` class is part of Angular's Fire library. This class is responsible for authentication in our application and contains an `auth` object, so we create a stub for the `auth` class, too:

```
class AngularFireAuthStub {  
    readonly auth: AuthStub = new AuthStub();  
}
```

Here's the `AuthStub` class:

```
class AuthStub {  
    onAuthStateChanged() {  
        return Observable.of({uid: '1234'});  
    }  
}
```

Finally, we use a router to navigate to pages in our application. This provider is part of Angular framework. We create a stub for this class as well:

```
class RouterStub {  
    navigateByUrl(url: string) {  
        return url;  
    }  
}
```

3. **Creating the test suites:** Once we eliminate the external dependencies, we can create test suites within the `describe()` method in Jasmine framework. This method accepts `description` for test suites and the `specDefinitions` function for Jasmine framework to invoke inner suites of specs:

```
|     describe(description: string, specDefinitions: () => void)
```

4. **Creating the test environment:** We create the Angular test environment for components under test. Angular provides a `TestBed` class to create the test

environment; it initializes the dependent modules, providers, services, and components. We call a `TestBed.configureTestingModule()` method within `beforeEach()` so that this configures modules before each test case execution:

```
beforeEach(async(() => {
  TestBed.configureTestingModule({
    declarations: [
      LoginComponent,
      ErrorAlertComponent
    ],
    imports: [
      CommonModule,
      BrowserModule,
      FormsModule
    ],
    providers: [
      {provide: UserService, useClass: UserServiceStub},
      {provide: Router, useClass: RouterStub},
      {provide: AuthenticationService, useValue: mockAuthService},
      {provide: AngularFireAuth, useClass: AngularFireAuthStub}
    ]
  }).compileComponents();
}));
```

5. **Initializing the testing objects:** Once we configure modules, we can create a login component fixture using `TestBed.createComponent()` and initialize the login component and debug element:

```
beforeEach(() => {
  fixture = TestBed.createComponent(LoginComponent);
  component = fixture.componentInstance;
  de = fixture.debugElement;
  fixture.detectChanges();
});
```

6. **Writing our first test:** The final step is to write the test cases. Our first test case is to check whether the login component is instantiated or not. We will write the test case within the `it()` method and use `expect()` to validate the instance:

```
it('Should instantiate LoginComponent', async(() => {
  expect(component instanceof LoginComponent).toBe(true,
    'LoginComponent not created');
}));
```

7. **Destroying the created instance:** After each test case, we clear the instance in the `afterEach()` method, as follows:

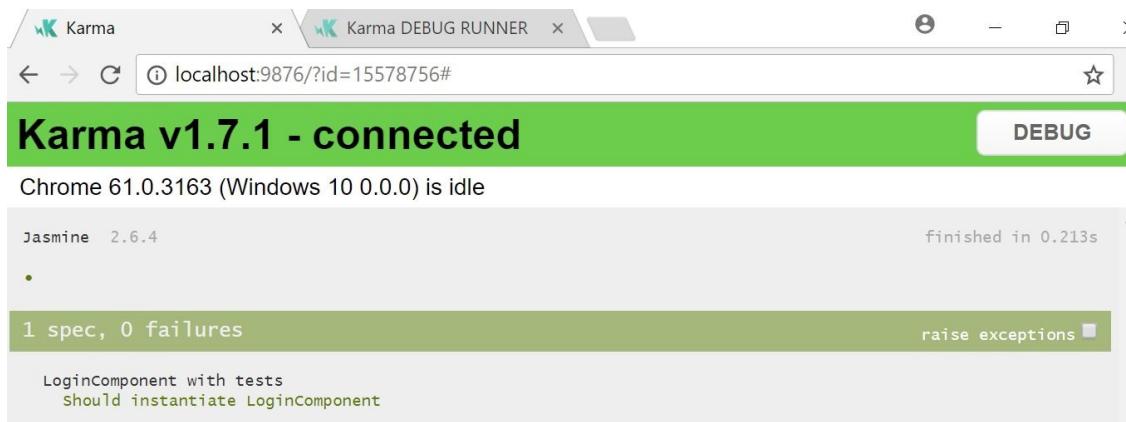
```
afterEach(async(() => {
  fixture.detectChanges();
  fixture.whenStable().then(() => fixture.destroy());
```

```
|     }));
```

8. **Running the test:** Finally, we run our first test case using the following command:

```
| $ng test
```

After its successful run, this opens in a browser and shows the status as 1 spec successful with 0 failures:



9. **Adding more unit tests:** In the next test case, we check whether the `login` method of our service is called when a user enters their email and password and clicks on the LOGIN button. First, we initialize the DOM elements, such as email input text, password input text, and login button. Next, we initialize the default values for email and password and `spyOn` the service `login` method so that this mock method is called when the user clicks on the LOGIN button. After the LOGIN button is clicked, we then call `detectChanges()` to notify the DOM to refresh the elements. Finally, we validate that the `login()` method should be called:

```
it('Should call login', async(() => {
  const loginButton = de.query(By.css('#login-btn'));
  expect(loginButton).not.toBeNull('Login button not found');

  spyOn(mockAuthService, 'login').and.callThrough();
  de.query(By.css('#email')).nativeElement.value =
    'user@gmail.com';
  de.query(By.css('#password')).nativeElement.value = 'password';
  fixture.detectChanges();

  // Click on Login button
  loginButton.nativeElement.click();
  fixture.detectChanges();
  expect(mockAuthService.login).toHaveBeenCalled();
}));
```

Here's the `login.component.spec.ts` file as of now:

```
import {async, ComponentFixture, TestBed} from '@angular/core/testing';
import { LoginComponent } from './login.component';
import { Router } from '@angular/router';
import { UserService } from '../../../../../services/user.service';
import { Observable } from 'rxjs/Observable';
import { User } from '../../../../../services/user';
import { mockUserJSON } from '../../../../../test-data/user-test-data';
import { AuthenticationService } from '../../../../../services/authentication.service';
import { AngularFireAuth } from 'angularfire2/auth';
import { CommonModule } from '@angular/common';
import { BrowserModule, By } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { DebugElement } from '@angular/core';
import { ErrorAlertComponent } from '../../../../../shared/error-alert/error-alert.component';

class RouterStub {
  navigateByUrl(url: string) {
    return url;
  }
}

class UserServiceStub {

  getUser(): Observable<User> {
    return Observable.of(mockUserJSON);
  }
}

class AuthenticationServiceStub {

  login(email: string, password: string) {
  }

  resetPassword(email: string) {
  }
}

class AngularFireAuthStub {
  readonly auth: AuthStub = new AuthStub();
}

class AuthStub {

  onAuthStateChanged() {
    return Observable.of({uid: '1234'});
  }
}

describe('LoginComponent with tests', () => {

  let fixture: ComponentFixture<LoginComponent>;
  let component: LoginComponent;
  let de: DebugElement;
  const mockAuthService: AuthenticationServiceStub = new
  AuthenticationServiceStub();

  beforeEach(async(() => {
    TestBed.configureTestingModule({
      declarations: [LoginComponent]
    })
    .compileComponents();
  }));

  it('should create the component', () => {
    expect(component).toBeTruthy();
  });

  it('should call auth service on login', () => {
    component.login('test@example.com', 'password');
    expect(mockAuthService.auth.onAuthStateChanged).toHaveBeenCalled();
  });

  it('should show error if login fails', () => {
    const error = new Error('Login failed');
    mockAuthService.auth.onAuthStateChanged = () => Observable.of(error);
    component.login('test@example.com', 'password');
    expect(de.nativeElement.innerHTML).toContain('Login failed');
  });

  it('should show success message if login succeeds', () => {
    component.login('test@example.com', 'password');
    expect(de.nativeElement.innerHTML).toContain('Success');
  });
})
```

```

declarations: [
  LoginComponent,
  ErrorAlertComponent
],
imports: [
  CommonModule,
  BrowserModule,
  FormsModule
],
providers: [
  {provide: UserService, useClass: UserServiceStub},
  {provide: Router, useClass: RouterStub},
  {provide: AuthenticationService, useValue: mockAuthService},
  {provide: AngularFireAuth, useClass: AngularFireAuthStub}
]
}).compileComponents();
}));

beforeEach(() => {
  fixture = TestBed.createComponent(LoginComponent);
  component = fixture.componentInstance;
  de = fixture.debugElement;
  fixture.detectChanges();
});

afterEach(async(() => {
  fixture.detectChanges();
  fixture.whenStable().then(() => fixture.destroy());
}));

it('Should instantiate LoginComponent', async(() => {
  expect(component instanceof LoginComponent).toBe(true,
    'LoginComponent not created');
}));

it('Should call login', async(() => {
  const loginButton = de.query(By.css('#login-btn'));
  expect(loginButton).not.toBeNull('Login button not found');

  spyOn(mockAuthService, 'login').and.callThrough();
  de.query(By.css('#email')).nativeElement.value = 'user@gmail.com';
  de.query(By.css('#password')).nativeElement.value = 'password';
  fixture.detectChanges();

  // Login button is enabled
  expect(loginButton.nativeElement.disabled).toBeFalsy();
  loginButton.nativeElement.click();
  fixture.detectChanges();
  expect(mockAuthService.login).toHaveBeenCalled();
}));
});

```

Unit testing an Angular service

In this section, we unit test an Angular service and we write test cases for our user service. The steps for unit testing a service are the same as for our component.

The first step in writing unit test cases is to analyze the dependent components, as we see user service is dependent on `AngularFireDatabase` and initializes the Firebase storage object:

```
| constructor(private fireDb: AngularFireDatabase)
```

So we create a stub for this dependent object such as `AngularFireDatabaseStub`, which contains other dependent stubs such as `AngularFireAppStub` and `AngularFireObjectStub` object references and the `object()` method:

```
class AngularFireDatabaseStub {  
  app: AngularFireAppStub = new AngularFireAppStub;  
  angularFireObject: AngularFireObjectStub;  
  constructor(angularFireObject: AngularFireObjectStub) {  
    this.angularFireObject = angularFireObject;  
  }  
  object(pathOrRef: PathReference): AngularFireObjectStub {  
    return this.angularFireObject;  
  }  
}
```

The following is the `AngularFireAppStub` stub class with an empty mock method:

```
class AngularFireAppStub {  
  storage() {}  
}
```

The following is the `AngularFireObjectStub` stub class with empty mock methods:

```
class AngularFireObjectStub {  
  set() {}  
  valueChanges() {}  
  update() {}  
}
```

```
| }
```

The next step is to initialize the test environment using `TestBed` and retrieve the user service object using `TestBed.get()`:

```
const angularFireObject: AngularFireObjectStub = new AngularFireObjectStub();
const mockAngularFireDatabase: AngularFireDatabaseStub = new
AngularFireDatabaseStub(angularFireObject);
let userService: UserService;

beforeEach(() => {
  TestBed.configureTestingModule({
    providers: [
      {provide: AngularFireDatabase, useValue:
        mockAngularFireDatabase},
      {provide: UserService, useClass: UserService}
    ]
  });
  userService = TestBed.get(UserService);
});
```

Now, we will start writing the test cases for our user service. We will cover the following test cases for the user service:

- The first test case is to add a user to Firebase database. We will add `spyOn` to the `set` method of Angular's fire object and call the add user method using mock user; then we expect the `set` method of Angular fire object to be called:

```
it('Add user', () => {
  spyOn(angularFireObject, 'set');
  userService.addUser(mockUserJSON);
  expect(angularFireObject.set).toHaveBeenCalledWith();
});
```

The next test case is to receive our user from the Firebase database. We add `spyOn`, the value change method of Angular's fire object, which returns a mock user. We then call a `getUser` method, subscribe to the `observable` object, and then we validate the method call and also test the content of our mock user with the expected values:

```
it('getUser return valid user', () => {
  spyOn(angularFireObject,
  'valueChanges').and.returnValue(Observable.of(mockUserJSON));
  userService.getUser(mockUserJSON.uid).subscribe((user) => {
    expect(angularFireObject.valueChanges).toHaveBeenCalledWith();
    expect(user.uid).toBe(mockUserJSON.uid);
    expect(user.name).toBe(mockUserJSON.name);
    expect(user.mobile).toBe(mockUserJSON.mobile);
    expect(user.email).toBe(mockUserJSON.email);
  });
});
```

```
});
```

3. The next test case is to save the user in `member` variables. In this test case, we will save a mock user in an `observable`, then retrieve the user using a `get` method, and validate all properties of the mock user:

```
it('saveUser saves user in Subject', () => {
  userService.saveUser(mockUserJSON);
  userService.getSavedUser().subscribe((user) => {
    expect(user.uid).toBe(mockUserJSON.uid);
    expect(user.name).toBe(mockUserJSON.name);
    expect(user.mobile).toBe(mockUserJSON.mobile);
    expect(user.email).toBe(mockUserJSON.email);
  });
});
```

4. The next test case is to update the email in the Firebase database and also update the cache user object in the user service class; we spy on the `update` method of Angular's fire object, pass a new email to update the `email` method, which updates the Firebase database and the cache user object, test the Firebase database call, retrieve the user from the `get` method, and validate all the properties of the mock user:

```
it('updateEmail update the email', () => {
  spyOn(angularFireObject, 'update');
  userService.saveUser(mockUserJSON);
  mockUserJSON.email = 'user1@gmail.com';
  userService.updateEmail(mockUserJSON, mockUserJSON.email);
  userService.getSavedUser().subscribe((user) => {
    expect(angularFireObject.update).toHaveBeenCalled();
    expect(user.email).toBe(mockUserJSON.email);
  });
});
```

Here's the `user.service.spec.ts` file as of now:

```
import {UserService} from './user.service';
import {AngularFireDatabase, PathReference} from 'angularfire2/database';
import {FirebaseApp} from 'angularfire2';
import {mockUserJSON} from '../test-data/user-test-data';
import {AngularFireAuth} from 'angularfire2/auth';
import {TestBed} from '@angular/core/testing';
import {Observable} from 'rxjs/Observable';
import {User} from './user';

class AngularFireDatabaseStub {
  app: AngularFireAppStub = new AngularFireAppStub();
  angularFireObject: AngularFireObjectStub;
```

```

    constructor(angularFireObject: AngularFireObjectStub) {
      this.angularFireObject = angularFireObject;
    }

    object(pathOrRef: PathReference): AngularFireObjectStub {
      return this.angularFireObject;
    }
  }

  class AngularFireAppStub {
    storage() {}
  }

  class AngularFireObjectStub {
    set() {}
    valueChanges() {}
    update() {}
  }

  describe('User service test suites', () => {

    const angularFireObject: AngularFireObjectStub = new
      AngularFireObjectStub();
    const mockAngularFireDatabase: AngularFireDatabaseStub = new
      AngularFireDatabaseStub(angularFireObject);
    let userService: UserService;

    beforeEach(() => {
      TestBed.configureTestingModule({
        providers: [
          {provide: AngularFireDatabase, useValue:
            mockAngularFireDatabase},
          {provide: UserService, useClass: UserService}
        ]
      });
      userService = TestBed.get(UserService);
    });

    it('Add user', () => {
      spyOn(angularFireObject, 'set');
      userService.addUser(mockUserJSON);
      expect(angularFireObject.set).toHaveBeenCalled();
    });

    it('getUser return valid user', () => {
      spyOn(angularFireObject,
        'valueChanges').and.returnValue(Observable.of(mockUserJSON));
      userService.getUser(mockUserJSON.uid).subscribe((user) => {
        expect(angularFireObject.valueChanges).toHaveBeenCalled();
        expect(user.uid).toBe(mockUserJSON.uid);
        expect(user.name).toBe(mockUserJSON.name);
        expect(user.mobile).toBe(mockUserJSON.mobile);
        expect(user.email).toBe(mockUserJSON.email);
      });
    });

    it('saveUser saves user in Subject', () => {
      userService.saveUser(mockUserJSON);
    });
  });

```

```
    userService.getSavedUser().subscribe((user) => {
      expect(user.uid).toBe(mockUserJSON.uid);
      expect(user.name).toBe(mockUserJSON.name);
      expect(user.mobile).toBe(mockUserJSON.mobile);
      expect(user.email).toBe(mockUserJSON.email);
    });
  });

it('updateEmail update the email', () => {
  spyOn.angularFireObject, 'update');
  userService.saveUser(mockUserJSON);
  mockUserJSON.email = 'user1@gmail.com';
  userService.updateEmail(mockUserJSON , mockUserJSON.email);
  userService.getSavedUser().subscribe((user) => {
    expect.angularFireObject.update).toHaveBeenCalled();
    expect(user.email).toBe(mockUserJSON.email);
  });
});

it('updateMobile update the mobile', () => {
  spyOn.angularFireObject, 'update');
  userService.saveUser(mockUserJSON);
  mockUserJSON.mobile = '88888888';
  userService.updateMobile(mockUserJSON , mockUserJSON.mobile);
  userService.getSavedUser().subscribe((user) => {
    expect.angularFireObject.update).toHaveBeenCalled();
    expect(user.mobile).toBe(mockUserJSON.mobile);
  });
});
```

Unit testing Angular pipe

Angular pipe unit testing is an example of testing a class independently of the Angular test environment. In this example, we test our friend's date pipe class and create the object in the test class:

```
| const pipe = new FriendsDatePipe();
```

On this object, we will write the following two test cases:

1. First, test the green field scenario, where we pass a valid date in milliseconds and test the transformed human-readable date format
2. Second, test the edge-case scenario where we pass an invalid date as -1, and we expect a string return value as "Invalid Date"

```
import {FriendsDatePipe} from './friendsdate.pipe';

describe('friendsdatepipe', () => {
  const pipe = new FriendsDatePipe();

  it('Transform dateInMillis to MM/DD/YY', () => {
    expect(pipe.transform('1506854340801')).toBe('10/01/17');
  });

  it('Transform invalid date', () => {
    expect(pipe.transform('-1')).toBe('Invalid Date');
  });
});
```

Code coverage

Code coverage of the application reflects the overall coverage of our code. This gives an overview of the line and function coverage of the code so that we can write more test cases to cover other parts of the code.

We can enable code coverage in `package.json`, as follows:

```
"scripts": {  
  "ng": "ng",  
  "start": "ng serve",  
  "build": "ng build",  
  "test": "ng test --sourcemaps false",  
  "coverage": "ng test --sourcemaps false --watch=false  
    --code-coverage",  
  "lint": "ng lint",  
  "e2e": "ng e2e"  
}
```

We will execute the following command, which runs the test cases, and creates a `coverage` folder, which has `index.html` to show coverage statistics:

```
| $ng test --codeCoverage
```

When we open `index.html`, it shows a beautiful table with an overview of the coverage:

All files	Statements	Branches	Functions	Lines
70.77% Statements 92/130 50% Branches 7/14 38.89% Functions 11/36 69.57% Lines 88/115				
File	Statements	Branches	Functions	Lines
src	100%	8/8	100%	0/0
src/app/authentication/login	55.26%	21/38	0%	0/2
src/app/services	68.66%	46/67	0%	0/2
src/app/shared/error-alert	100%	6/6	100%	0/0
src/app/test-data	100%	1/1	100%	0/0
src/app/utils	100%	10/10	100%	4/4

Summary

In this chapter, we covered unit testing and discussed various terms and terminologies in Angular unit testing. We implemented unit test for our login component, covered `TestBed`, and configured our modules. We wrote a unit test for our service, created stubs for external dependent classes, and injected these stubbed classes in our module. We also wrote isolated test cases for Angular pipe. Finally, we discussed code coverage and ran code coverage for our specs.

In the next chapter, we will discuss debugging techniques. This will help us to solve and debug our issues faster.

Debugging Techniques

Debugging software is considered the most important part of software development. This process increases the efficiency and decreases the development time. In this chapter, we will discuss the different methodologies and cover all the aspects required to debug an application. We will start with Angular and HTML debugging using Chrome's developer tools and also touch on Augury for debugging an Angular component. We will move forward with TypeScript debugging, as this will help to check for errors within our TypeScript code. We will cover CSS so that we can design many style elements from the tools themselves. Finally, we will take a look at network API calls using **Postman** and developer tools. As part of this chapter, we will cover Chrome's browser tools.

In this chapter, we will cover the following topics:

- Debugging Angular
- Debugging a web application
- Debugging TypeScript
- Debugging CSS
- Debugging a network

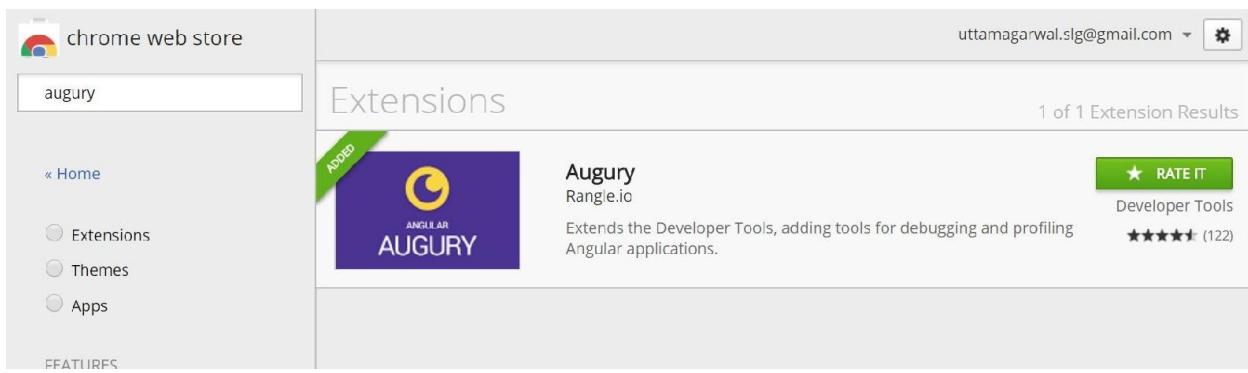
Debugging Angular

In this section, we will start with Augury for Chrome. This tool gives a very nice representation of various Angular components with dependencies. It provides an insight into the application and relationships between different building blocks.

Installing Augury

The best way to install Augury is to use the web store in Chrome. Alternatively, you can also install it from the Augury site. Perform the following steps to install Augury:

1. Open `chrome://apps/` on your Chrome browser.
2. Click on Web Store on the bottom-right corner.
3. On the web store page, type `augury` into the search box and press *Enter*, and Augury appears on the right panel, as follows; in my case, this appears as ADDED, but for a new installation an ADD TO CHROME button appears, which you need to click:



Finally, when Augury is installed, the black moon with the yellow background will appear on the top-right corner of the Chrome browser, as follows:



Using Augury's features

Augury provides good features, which helps to preview our data and components at a glance. In this chapter, we cover the following features:

- Component tree
- Router tree
- NgModules

Component tree

When you launch the Augury console, the first view that you can see is the component tree. The following screenshot shows the user profile view of our friends application:

The screenshot shows the Augury interface with two main panels. On the left is a hierarchical component tree, and on the right is a properties panel.

Component Tree (Left Panel):

- AppComponent
 - a (text="My Profile", hash="")
 - a (text="Friends", hash="")
 - a (text="About", hash="")
 - form (method="get")
 - router-outlet (name="")
- UserProfileComponent
 - EditDialogComponent

Properties Panel (Right Panel):

- Properties** tab selected.
- UserProfileComponent (View Source)** (with a link to `($$el in Console)`)
- Change Detection:** Default
- State**:
 - profileImage: `https://firebasestorage.googleapis.com/v0/b/1`
 - ▶ authService: AuthenticationService
 - ▶ userService: UserService
 - ▶ router: Router
 - ▶ @ViewChild(EditDialogComponent) editDialog: EditDialogCom
 - ▶ user: User
- Dependencies**:
 - AuthenticationService
 - UserService

The preceding component tree shows a hierarchical view of `AppComponent` and `UserProfileComponent`. On the right panel, we have the Properties tab, which contains the following:

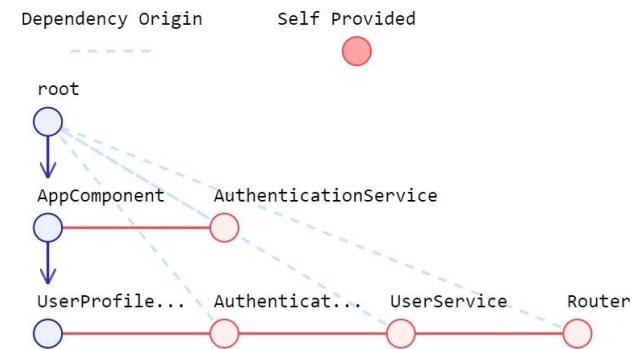
- **View Source:** This shows the actual source code of the component
- **Change Detection:** This shows whether we have used a change detection in our component or not
- **State:** This shows all the instance member variables of the class
- **Dependencies:** This shows the dependencies of the component with other components or providers

Another view is the Injector Graph, which shows where the particular component or provider is injected:

Component Hierarchy

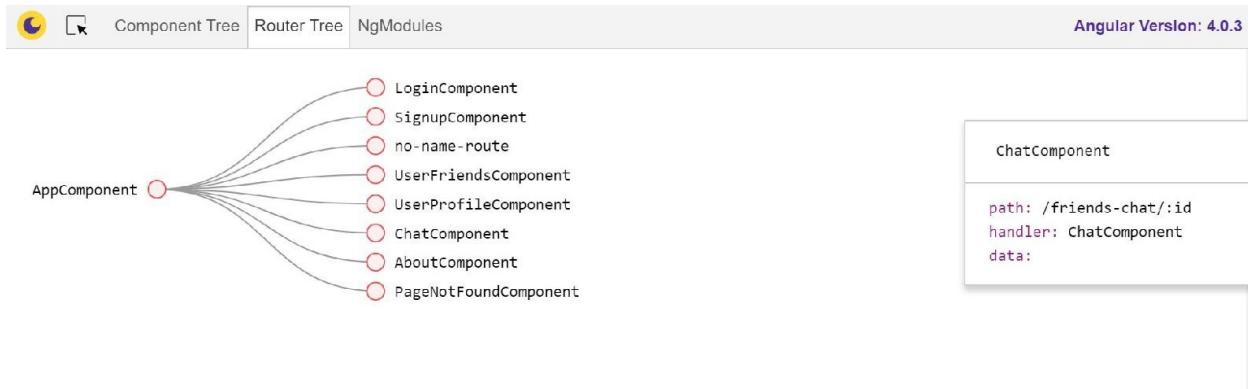
AppComponent » UserProfileComponent

Injector Graph



Router tree

Augury provides the routing information for an application. This helps in viewing all the routes used in the application. This option is available next to the Component Tree option, as shown in the following screenshot:



NgModules

NgModules is another useful feature added in Augury. It is available next to Router Tree. It provides information about all the Imports, Exports, Providers, Declarations, and providersInDeclarations configured in particular application modules, as follows:

BrowserAnimationsModule				
Imports	Exports	Providers	Declarations	ProvidersInDeclarations
BrowserModule	None	AnimationDriver AnimationStyleNormalizer AnimationEngine RendererFactory2	None	None
AuthenticationModule				
Imports	Exports	Providers	Declarations	ProvidersInDeclarations
CommonModule BrowserModule FormsModule AuthenticationRoutingModule	None	AuthenticationService AuthenticationGuard UserService	LoginComponent SignupComponent PasswordEqualValidator	None

Debugging a web application

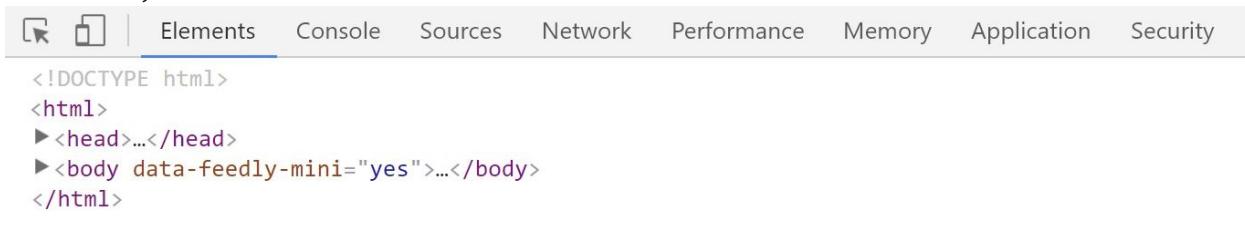
In this section, we will cover Chrome developer tools. These tools provide many features for analyzing our application. We cover the following topics:

- HTML DOM
- Layout Preview

HTML DOM

You can open your Chrome developer tools by right-clicking on the mouse on the browser page and Inspect or press *F12*. The Chrome developer tool opens with multiple tabs.

You can click on the Elements tab, which shows the DOM elements with an `<html>` tag as the root. You can further expand it by a right arrow icon on each elements, as follows:



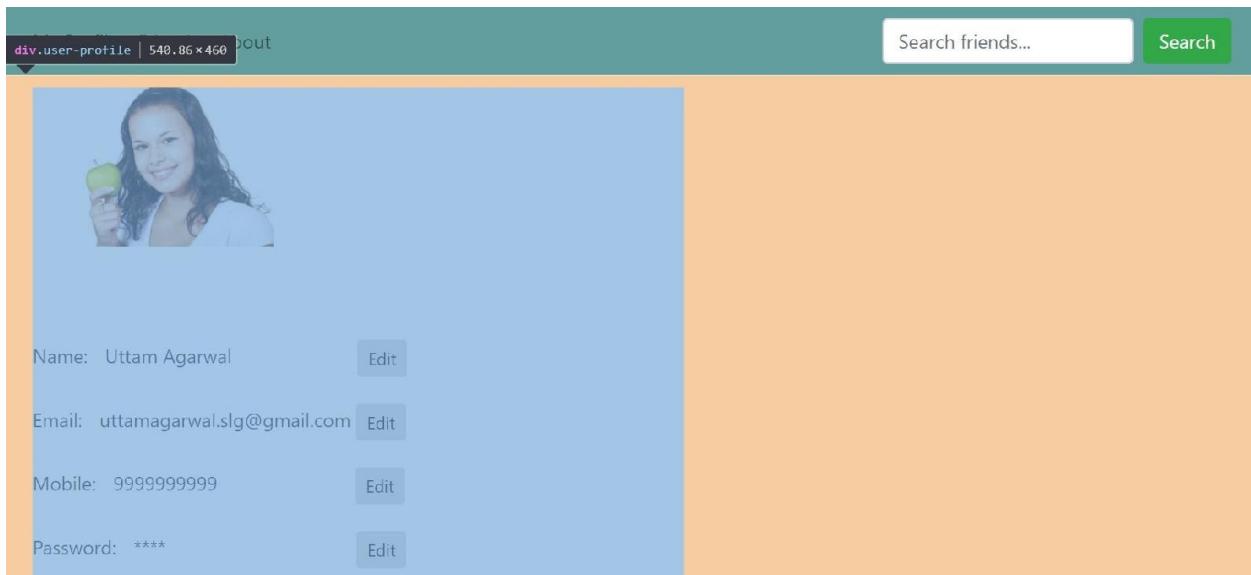
The screenshot shows the Chrome Developer Tools interface with the 'Elements' tab selected. The top navigation bar includes icons for refresh, zoom, and file operations, followed by tabs for Elements, Console, Sources, Network, Performance, Memory, Application, and Security. Below the tabs, the main area displays the HTML document structure:

```
<!DOCTYPE html>
<html>
  ><head>...</head>
  ><body data-feedly-mini="yes">...</body>
</html>
```

The preceding preview helps to debug HTML elements.

Layout preview

Layout preview is a nice feature to preview the layout with an actual browser view. This tool provides a two-way view, as you can hover over the web view on the browser using the mouse's arrow key, and it shows actual HTML elements on your developer tool. You can also hover over the HTML elements in the developer tool and take a look at the highlight view in the browser, as shown in the subsequent screenshot. As you hover over the browser, you will see the actual `<div>` tag with the style used as `.user-profile`, and when you click on the element, the elements in the developer tool open.



This layout preview helps to debug our live application.

Debugging TypeScript

This is an another important aspect of debugging the live TypeScript code. The Chrome tool provides a good mechanism to debug the code. In this section, we cover the following features of the Sources tab in the Chrome developer tool:

- Viewing and searching the source file
- Putting breakpoints and watching live values
- Adding code in the console window

Viewing and searching a source file

We can see our TypeScript files on the Sources tab in the developer tool. All the files appear on the left panel under the `webpack://` folder. The folder will appear as



follows:

You can also search the files using the `Ctrl + P` command. The source code appears in the middle panel:

The screenshot shows the Chrome DevTools Sources tab. On the left, a file tree displays the project structure under the `webpack://` root. The `user-profile.component.ts` file is selected and shown in the main pane. The code content is as follows:

```
import {Component, OnInit, ViewChild} from '@angular/core';
import {AuthenticationService} from '../../../../../services/authentication';
import {Router} from '@angular/router';
import {User} from '../../../../../services/user';
import {UserService} from '../../../../../services/user.service';
import {EditDialogComponent} from '../../../../../edit-dialog/edit-dialog';
import {EditType} from '../../../../../edit-dialog/edit-type';

@Component({
  selector: 'friends-userprofile',
  styles: [require('./user-profile.component.scss').css],
  template: require('./user-profile.component.html').toString()
})
export class UserProfileComponent implements OnInit {
  private user: User;
  private profileImage: any;
}
```

The right side of the interface contains several developer tools panels:

- Watch: Not paused.
- Call Stack: Not paused.
- Scope: Not paused.
- Breakpoints: No breakpoints.
- XHR Breakpoints
- DOM Breakpoints
- Global Listeners
- Event Listener Breakpoints

Putting in breakpoints and watching live values

Live debugging is the most interesting part of debugging and really helps in debugging the legacy code, as this helps to know the code flow by putting breakpoints in the code.

You can enable the breakpoint by just clicking on the number in the code line, as



A screenshot of a code editor showing a file named 'user-profile.component.ts?38e8'. The code is written in TypeScript. A blue rectangle highlights line 28, which contains a breakpoint. The code is as follows:

```
12
13     template: require('./user-profile.component.html')
14 }
15 export class UserProfileComponent implements OnInit {
16
17     private user: User;
18
19     private profileImage: any;
20
21     @ViewChild(EditDialogComponent) editDialog: EditDialogComponent;
22
23     constructor(private authService: AuthenticationService,
24                 private userService: UserService,
25                 private router: Router) {
26
27     }
28     ngOnInit() {
29         this.user = this.userService.getSavedUser().getValue();
30         this.userService.getSavedUser().subscribe(
31             (user) => {
32                 this.profileImage = this.user.getImage();
```

follows:

When you refresh the page, the application stops at the breakpoint line. You can traverse the code using the panel on the top-right corner, which has a command to step over, step into, step out, and resume the application. You can also add watch values using the plus icon on the right corner of the Watch panel; as shown in the following screenshot, we added a user object on the Watch panel on the right:

The screenshot shows an IDE interface with a code editor on the left and a debugger sidebar on the right.

Code Editor:

```
11     styles: [require('./user-profile.component.scss')],  
12     template: require('./user-profile.component.html')  
13 }  
14 export class UserProfileComponent implements OnInit {  
15  
16     private user: User;  
17  
18     private profileImage: any;  
19  
20     @ViewChild(EditDialogComponent) editDialog: EditDialogComponent;  
21  
22     constructor(private authService: AuthenticationService,  
23                  private userService: UserService,  
24                  private router: Router) {  
25     }  
26  
27     ngOnInit() {  
28         this.user = this.userService.getSavedUser().getValue();  
29         this.userService.getSavedUser().subscribe(  
30             (user) => {  
31                 this.fillForm(this.user);  
32             }  
33         );  
34     }  
35 }
```

Debugger Sidebar:

- Debugger paused
- Watch
 - body: <not available>
 - fetchParams: <not available>
 - reqsFetchParams: <not available>
 - requests: <not available>
- this.user: User
- Call Stack
 - UserProfileComponent.ngOnInit
 - user-profile.co...ent.ts?38e8:29
 - checkAndUpdateDirectiveInline
 - core.es5.js?0445:10704
 - checkAndUpdateNodeInline
 - core.es5.js?0445:12083
 - checkAndUpdateNode
 - core.es5.js?0445:12051

This is a quick way to debug an issue or to know the flow of the application.

Adding code in the console window

The developer tool provides a feature to add code on to the deployed code and do real-time coding. We can open a file and just click on the line to add implementation on to the existing file.

In the following screenshot, we added `console.log` to note the log on the console

```
26
27
28 ngOnInit() {
29     this.user = this.userService.getSavedUser().getValue();
30     this.userService.getSavedUser().subscribe(
31         (user) => {
32             this.profileImage = this.user.getImage();
33         }
34     );
35 }
```

window:

This editor also provides content assist, which is context-sensitive content completion upon user request.

Content assist appears when we enter the following command into any line in the code; a window appears with an all reference option, and you can filter the option further by typing a specific alphabet key: **\$Ctrl + Space bar**

Take a look at the content assist for the HTML page:

This feature helps to add code to the existing file and also debugs the code faster.

Debugging CSS

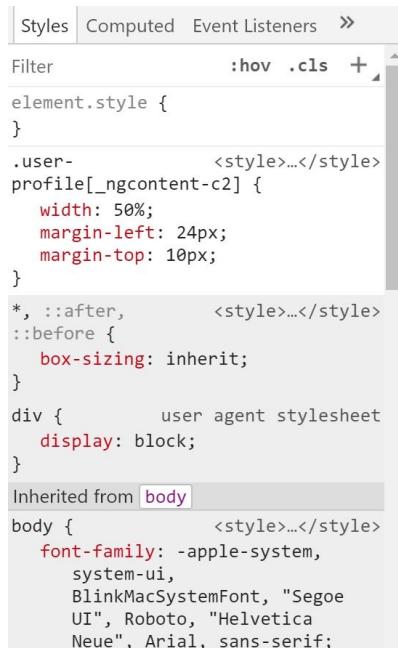
Debugging CSS is another aspect of web development. In this section, we cover debugging and designing live CSS elements. The Chrome developer tool provides an option to change the style elements and add a new element. We will cover the following features:

- Exploring the styles panel
- Discovering and modifying styles

Exploring the styles panel

Once you have opened Chrome's developer tools, open the Elements tab and click on any HTML `<div>` tag; you should see a Styles panel appear on the right side. This panel consists of three tabs:

- **Styles:** This shows all the styles applied to a particular HTML element:



The screenshot shows the 'Styles' tab of the Chrome DevTools Elements panel. At the top, there are three tabs: 'Styles' (which is active), 'Computed', and 'Event Listeners'. Below the tabs is a 'Filter' input field containing ':hov .cls +' and a '+' button. The main area displays a list of CSS rules. The first rule is 'element.style { }'. The second rule is '.user-profile[_ngcontent-c2] { width: 50%; margin-left: 24px; margin-top: 10px; }'. The third rule is '*,:after,:before { box-sizing: inherit; }'. The fourth rule is 'div { user agent stylesheet display: block; }'. Below these, a grey box labeled 'Inherited from body' contains the rule 'body { font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", Roboto, "Helvetica Neue", Arial, sans-serif; }'. The entire screenshot is framed by a light grey border.

- **Computed:** This shows all the computed values applied to a particular HTML element; this also shows the box model, which contains content, padding, border, and margin information about the selected HTML element:

Styles Computed Event Listeners >

The screenshot shows the Chrome DevTools Computed tab. At the top, there are three tabs: Styles, Computed (which is selected), and Event Listeners. Below the tabs is a box model diagram. The outermost box is orange and represents the margin, with a value of 10px. Inside it is a yellow box representing the border, with a value of - (none). Inside that is a green box representing the padding, with a value of - (none). The innermost box is blue and represents the content area, with a value of 540.857 x 456.286 pixels. To the left of the box model diagram, there is some text: "margin 10", "border -", "padding -", "24", and "-". Below the box model diagram is a list of computed styles:

Style	Value
box-sizing	border-box
color	rgb(41, 128, 185)
display	block
font-family	-apple-system, system-ui, sans-serif
font-size	16px
font-weight	normal
height	456.286px
line-height	24px
margin-left	24px
margin-top	10px

- **Event Listeners:** This shows all the click events available for a particular HTML element:

Styles Computed Event Listeners >

Filter: All

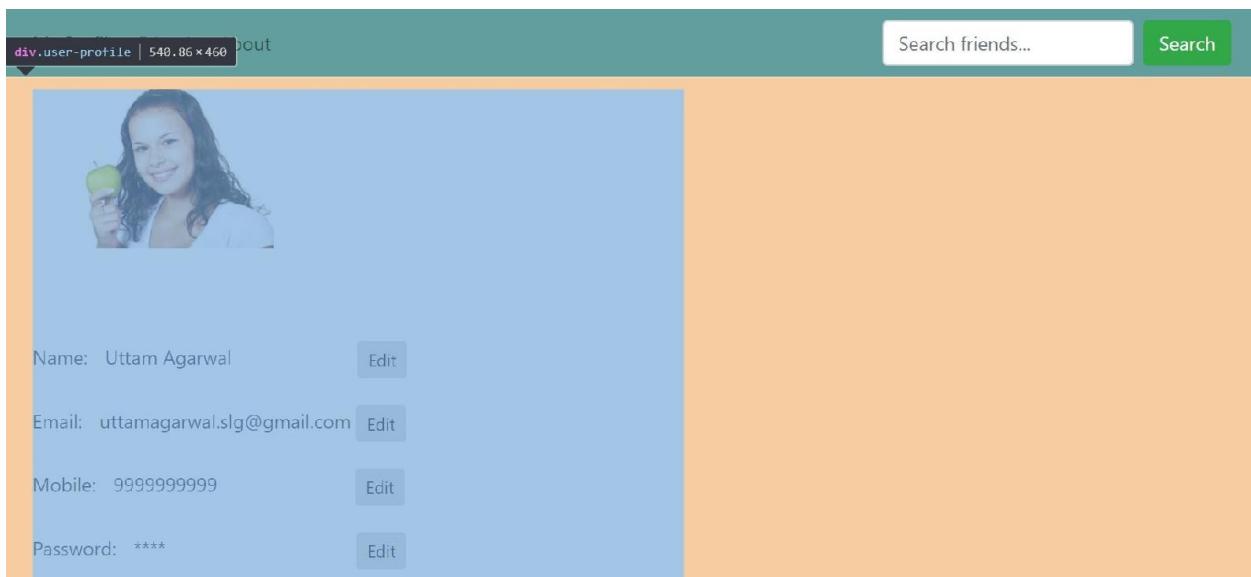
Event Listeners:

- DOMContentLoaded
- blur
- click
- focus
- focusin
- focusout
- hashchange
- keydown
- load
- message
- offline
- online
- popstate
- storage
- unload
- visibilitychange

Discovering and modifying styles

In this section, we will modify the styles of an HTML element, and, as part of an exercise, we modify the existing user profile page. We perform the following steps:

1. Open the friends application.
2. Go to the user profile page.
3. Hover over the user profile page, as follows:



4. Click on the user profile page on the browser page; it opens the Chrome developer tool with highlight on the HTML element and on the right panel you can see the `.user-profile` style.
5. When you hover over the style element, check boxes appear for all the style rules, and we can uncheck a check box to disable the particular style and see the effect on the user profile page. In the following example, we disabled the width style rule:

```
.user-          <style>...</style>
profile[_ngcontent-c2] {
 width: 50%;  

 margin-left: 24px;  

 margin-top: 10px;  

}  
:
```

6. We can add a new style to our existing styles rules. When we hover around the overflow icon, a tool tip appears with an option to add new styles, and, as we add styles, it also supports content assist:

```
.user-          <style>...</style>
profile[_ngcontent-c3] {
    width: 60%;  

    margin-left: 24px;  

    margin-top: 10px;  

}
```

7. Finally, we can edit the existing styles by clicking on the existing item. In the following example we changed the width from 50% to 60%:

```
.user-          <style>...</style>
profile[_ngcontent-c3] {
    width: 60%;  

    margin-left: 24px;  

    margin-top: 10px;  

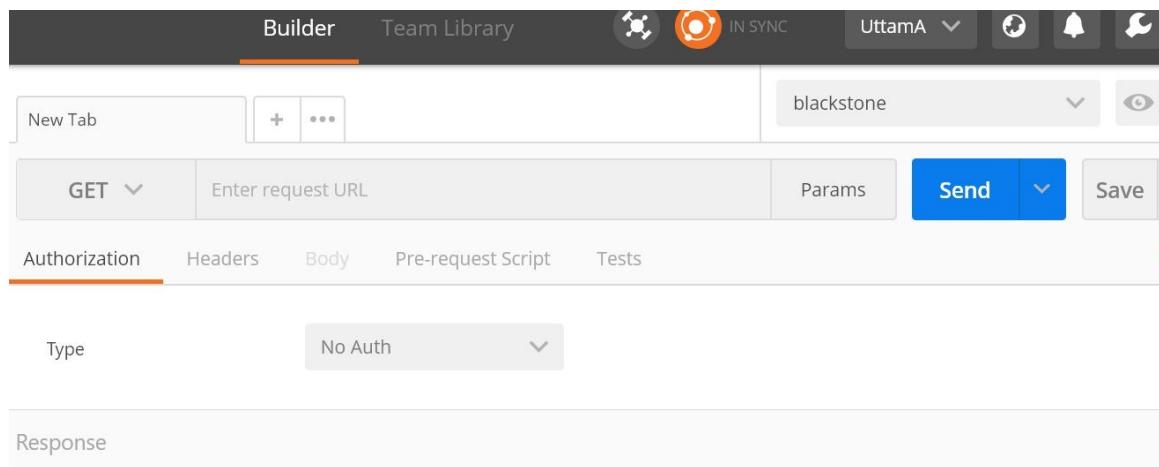
}
```

This option in the tool helps to debug the styles within our web pages.

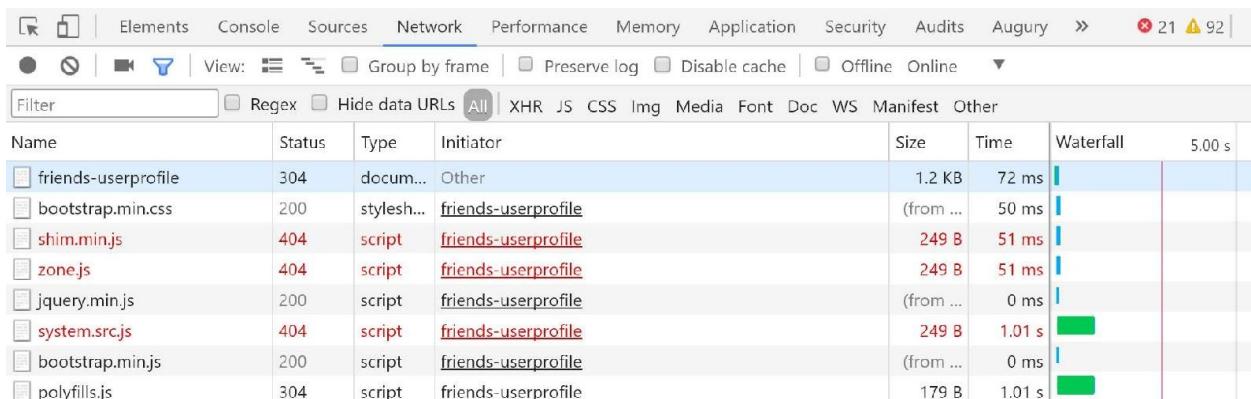
Network debugging

Network debugging is quite useful in understanding the API call and its response. In the Firebase API call, we won't find this of much use, as the Firebase database portal provides a view of the JSON response. Network debugging tools are quite handy when we explore live debugging of network calls. In this section, we will discuss the following two tools:

- **Postman:** This is similar to Augury extension; you can install the Postman from the Chrome extension, or you can download the OS-specific installer from <https://www.getpostman.com/>. This tool is really useful in the initial phase of development, as this helps to understand the APIs and response and to integrate the APIs in the application accordingly. You can create HTTP methods, such as GET, POST, PUT, or DELETE, using Authorization, Headers, and Body:



- **The network tab in the Chrome developer tool:** This is really useful in live debugging of network calls in the Chrome developer tool. This shows all the network calls when the page is loaded. You can also apply the filter to see a particular network call type:



Network debugging helps confirm the expected response from the server.

Summary

In this chapter, we covered the different aspects of debugging techniques. We started with browser-related debugging techniques, which helped in analyzing and previewing HTML elements. We covered TypeScript debugging and placed a breakpoint on the deployed application file. We also covered CSS debugging. We disabled and added the style on the CSS panel. Finally, we covered networking debugging, where we discussed the Postman tool and the Chrome developer network tab. Debugging techniques help a lot in the process of becoming an efficient web developer.

In the next chapter, we will deploy our application to the Firebase server. We will also enable Firebase security, as this makes our application more secure.

Firebase Security and Hosting

Firebase provides flexible security rules with a JavaScript-like syntax, as this helps to structure our data and index the frequently used data. Security rules are integrated with Firebase authentication, which helps to define read and write access based on the user. In this chapter, we will add security rules for users and chat nodes to our Firebase database. Firebase security rules provide a nice simulator to check new rules before releasing them into production. We will also index our user's and their friends data for faster queries. Finally, we will deploy our application to the Firebase server. We will set up a different deployment environment so that we can test our application in the staging server and then deploy the application to the production server.

In this chapter, we will cover the following topics:

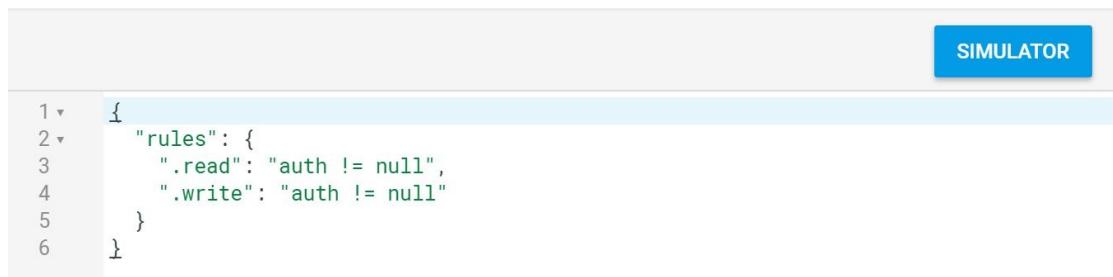
- Introducing Firebase security
- Adding security rules for users
- Adding security rules for chat messages
- Indexing users and their friends
- Setting up multiple deployment environments
- Hosting the friends app in Firebase

Introducing Firebase security

Firebase security provides tools to manage the security of our application, as we can add rules and validate inputs for our data in the Firebase database. Firebase provides the following security for our application:

- **Authentication:** The first step to secure our application is to identify the user. Firebase authentication supports multiple authentication mechanisms, such as Google, Facebook, email, and password authentication.
- **Authorization:** Once the user is authenticated, we will need to control the access to data in our database. Firebase security rules has built-in variables and functions, such as an `auth` object, as it helps to control read and write operations for users.

Go to the Firebase portal and navigate to Database|RULES tab. The default Firebase security rule is as follows; the `auth != null` condition means that only authenticated users have access to data in the Firebase database.



```
1 v   {
2 v     "rules": {
3 v       ".read": "auth != null",
4 v       ".write": "auth != null"
5 v     }
6 }
```

Firebase security rules provides the following four types of functions:

- `.read`: We can define this function for the data and control the user's read operation.

The following example shows that only a logged-in user can read their own user data:

```
|   {
|     "rules": {
```

```

    "users": {
        "$uid": {
            ".read": "auth != null && $uid === auth.uid"
        }
    }
}

```

- `.write`: We define this for the data and control the user's write operation.

The following example shows that only a logged-in user can write on their own user data node:

```

{
    "rules": {
        "users": {
            "$uid": {
                ".write": "auth != null && $uid === auth.uid"
            }
        }
    }
}

```

- `.validate`: This function maintains the integrity of the data, and this variable provides data validation.

The following example validates the `name` field to be a string:

```

{
    "rules": {
        "users": {
            "$uid": {
                "name": {
                    ".validate": "newData.isString()"
                }
            }
        }
    }
}

```

- `.indexOn`: This provides a child index for the querying and ordering of data.

In the following example, we index the `name` field of user data.

```

{
    "rules": {
        "users": {
            ".indexOn": ["name"],
            "$uid": {
                "name": {
                    ".validate": "newData.isString()"
                }
            }
        }
    }
}

```

```
|     }
```

Firebase security rules also provide the following predefined variables, which are used to define the security rules:

- `root`: This variable gives a `RuleDataSnapshot` instance to access data from the root of the Firebase database.

The following root variables are used to traverse the Firebase database path from the root user node:

```
{  
  "rules":{  
    "users":{  
      "$uid":{  
        "image": {  
          ".read":"root.child('users').  
            child(auth.uid).child('image').val() === ''"  
        }  
      }  
    }  
  }  
}
```

- `newData`: This variable provides a `RuleDataSnapshot` instance that represents the new data, which exists after the insert operation.

The following example validates the new data to be a string:

```
"name": {  
  ".validate":"newData.isString()"  
}
```

- `data`: This variable gives a `RuleDataSnapshot` instance that represents the data that exists before the insert operation.

The following example shows that the current data in the `name` field is not null:

```
"user": {  
  "$uid":{  
    ".read":"data.child('name').val() != null"  
  }  
}
```

- `$variables`: This variable represents the dynamic IDs and keys.

In the following example, the unique ID is assigned to the `$uid` variable:

```
"user": {  
    "$uid": {  
        }  
    }
```

- `auth`: This represents the `auth` object, which provides the UID of the user.

In the following example, we access the `auth` object to get a UID of a user:

```
"$uid": {  
    ".write": "auth != null && $uid === auth.uid"  
}
```

- `now`: This variable provides the current time in milliseconds and helps to validate the time stamp.

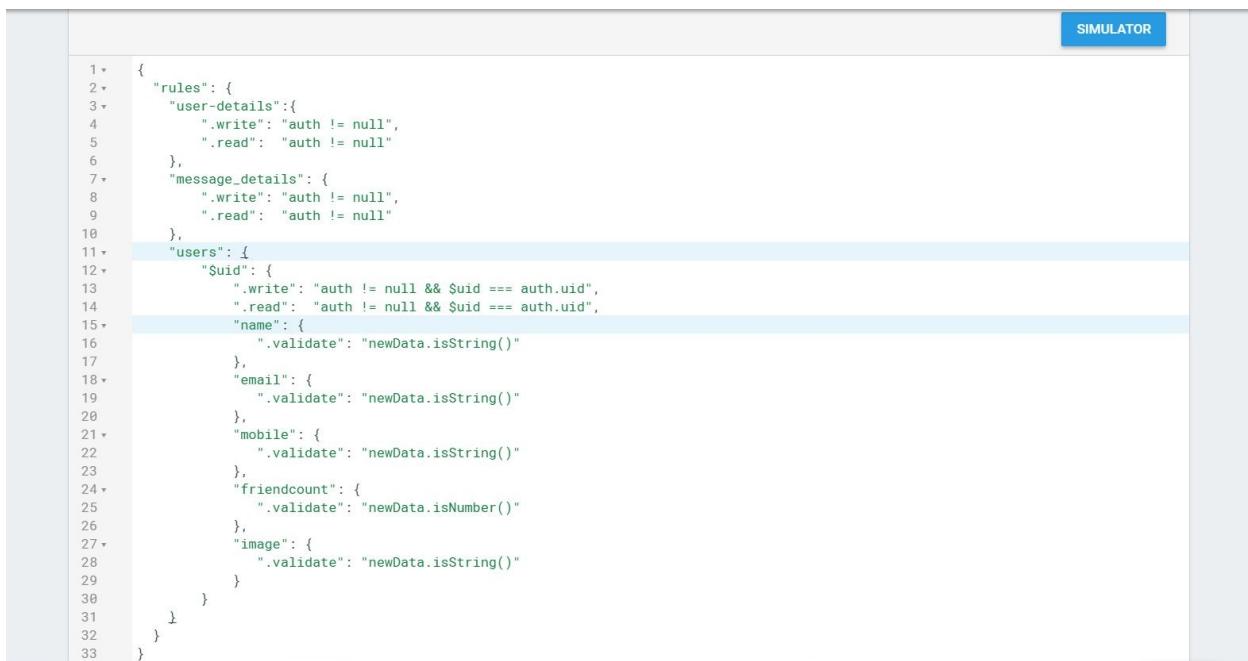
In the following example, we conclude that the time stamp is greater than the current time:

```
"$uid": {  
    ".write": "newData.child('timestamp').val() > now"  
}
```

Adding security rules for users

In our application, users' details play a critical part, so we need to provide security rules for our users' details. We have already seen the default settings for security. By default, only an authenticated user can access any part of our Firebase database. We will modify the security rules for the user node and retain the default security rules for the other nodes for now.

As you can see from the following screenshot, for `users` node, `read` and `write` operations are allowed for an authentic user with the same unique user ID; we will also need to validate the type of data in our user node to maintain the data integrity:



The screenshot shows the Firebase Rules tab. At the top right, there is a blue button labeled "SIMULATOR". The code area contains the following security rules:

```
1  {
2    "rules": {
3      "user-details": {
4        ".write": "auth != null",
5        ".read": "auth != null"
6      },
7      "message_details": {
8        ".write": "auth != null",
9        ".read": "auth != null"
10     },
11    "users": {
12      "$uid": {
13        ".write": "auth != null && $uid === auth.uid",
14        ".read": "auth != null && $uid === auth.uid",
15        "name": {
16          ".validate": "newData.isString()"
17        },
18        "email": {
19          ".validate": "newData.isString()"
20        },
21        "mobile": {
22          ".validate": "newData.isString()"
23        },
24        "friendcount": {
25          ".validate": "newData.isNumber()"
26        },
27        "image": {
28          ".validate": "newData.isString()"
29        }
30      }
31    }
32  }
33 }
```

To validate the changes in our security rules, Firebase provides a simulator to test our changes before deploying it into production. You will see a SIMULATOR option on the top-right corner of the RULES tab. This tool provides the mock operation without actually performing any CRUD operation within the database. We will test the following scenarios on the simulator:

- **Successful read operation for an authentic user:** Open the simulator and

enable the Authenticated switch button; it provides a mock uid in the Auth token payload text box. In the Location text box, we enter the path as `/users/6e115890-7802-4f56-87ed-4e6ac359c2e0` and click on the RUN button. This operation will be successful when the Simulated read allowed message appears, as shown in the following screenshot:

The screenshot shows the Firebase Emulator Suite interface. On the left, there is a code editor window titled "Simulated read allowed" containing a portion of a Firebase rules file. The code defines rules for "rules", "user-details", "message_details", and "users" nodes. The "users" node has a rule for ".write" that checks if auth is not null and \$uid is equal to auth.uid. It also has a rule for ".read" that checks if auth is not null and \$uid is equal to auth.uid. The "name" child node under "users" has validation rules for "validate": "newData.isString()", "email": { ".validate": "newData.isString()" }, "mobile": { ".validate": "newData.isString()" }, "friendcount": { ".validate": "newData.isNumber()" }, and "image": { ".validate": "newData.isString()" }. The code editor has line numbers from 1 to 33.

On the right, there is a "Simulator" panel with the following settings:

- Simulation type:** Read (radio button selected)
- Location:** `https://friends-4d4fa.firebaseio.com`
`/users/6e115890-7802-4f56-87ed-`
- Authenticated:**
- Provider:** Custom
- Permissions:** Admin
- Auth token payload:** A JSON object with "provider": "anonymous" and "uid": "6e115890-7802-4f56-87ed-4e6ac359c2e0". The "uid" field is highlighted in red.

- **Successful write operation with an authentic user and proper data:** In this scenario, we will provide a location path with the user UID with the string name data as JSON payload in the simulator. When we click on RUN, this operation is considered successful when the Simulated write allowed message appears, as shown in the following screenshot. Also, the succeeding screenshots show two ticks, which indicates that our authorization and data validation have been successful.

The screenshot shows the Firebase Simulator interface. On the left, there is a code editor window titled "Simulated write allowed" containing a JSON rules file. The file defines rules for "user-details", "message_details", and "users" nodes. The "users" node has a rule for ".write" that checks if the user's UID matches the path's UID. The code editor has syntax highlighting and line numbers from 1 to 33.

On the right, the "Simulator" panel is open. It includes sections for "Simulation type" (Read is off, Write is on), "Location" (set to https://friends-4d4fa.firebaseio.com /users/6e115890-7802-4f56-87ed), "Data (JSON)" (containing a simple object with a name field), "Authenticated" (switch is on), "Provider" (set to Custom), and "Permissions" (Admin checkbox is off). At the bottom right of the simulator panel, there is a link "Auth token payload" with a help icon.

```

1  {
2    "rules": {
3      "user-details": {
4        ".write": "auth != null",
5        ".read": "auth != null"
6      },
7      "message_details": {
8        ".write": "auth != null",
9        ".read": "auth != null"
10     },
11     "users": {
12       "$uid": {
13         ".write": "auth != null && $uid === auth.uid",
14         ".read": "auth != null && $uid === auth.uid",
15         "name": {
16           ".validate": "newData.isString()"
17         },
18         "email": {
19           ".validate": "newData.isString()"
20         },
21         "mobile": {
22           ".validate": "newData.isString()"
23         },
24         "friendcount": {
25           ".validate": "newData.isNumber()"
26         },
27         "image": {
28           ".validate": "newData.isString()"
29         }
30       }
31     }
32   }
33 }
```

- **Failed write operation with a different UID:** In this scenario, we provide a wrong UID in the user path location and then perform the same write operation. This operation fails, resulting in a Simulated write denied message and a cross on the write tag, as follows; you can take a look at more error details by clicking on the DETAILS button:

The screenshot shows the Firebase Simulator interface. On the left, there is a code editor window titled "Simulated write denied" containing a portion of a Firebase rules file. The code is as follows:

```

1  {
2   "rules": {
3     "user-details": {
4       ".write": "auth != null",
5       ".read": "auth != null"
6     },
7     "message_details": {
8       ".write": "auth != null",
9       ".read": "auth != null"
10    },
11    "users": {
12      "$uid": {
13        ".write": "auth != null & $uid === auth.uid",
14        ".read": "auth != null & $uid === auth.uid",
15        "name": {
16          ".validate": "newData.isString()"
17        },
18        "email": {
19          ".validate": "newData.isString()"
20        },
21        "mobile": {
22          ".validate": "newData.isString()"
23        },
24        "friendcount": {
25          ".validate": "newData.isNumber()"
26        },
27        "image": {
28          ".validate": "newData.isString()"
29        }
30      }
31    }
32  }
33

```

The line at index 13, ".write": "auth != null & \$uid === auth.uid", is highlighted with a red background and a red X icon, indicating a validation error.

On the right side of the interface, there are several configuration panels:

- Simulation type:** A radio button group where "Write" is selected.
- Location:** A text input field showing the path "/users/90b4c556-dda-4b78-9539-516b5".
- Data (JSON):** A JSON input field containing:


```
{
        "name": "value"
      }
```
- Authenticated:** A toggle switch that is turned on (blue).
- Provider:** A dropdown menu set to "Anonymous".
- UID:** A text input field showing the value "90b4c556-dda-4b78-9539-516b5".
- Auth token payload:** A JSON input field containing:


```
{
        "provider": "anonymous",
        "uid": "90b4c556-dda-4b78-9539-516b52a96e0e"
      }
```

- **Failed write operation with data validation:** In this scenario, we provide the correct user UID path in Location but the wrong data type in payload. For example, we will give number data type for a string name data. This operation fails in data validation tag, as follows:

Simulated write denied

DETAILS CLEAR

```
1  "rules": {
2    "user-details": {
3      ".write": "auth != null",
4      ".read": "auth != null"
5    },
6    "message_details": {
7      ".write": "auth != null",
8      ".read": "auth != null"
9    },
10   "users": {
11     "$uid": {
12       ".write": "auth != null && $uid === auth.uid",
13       ".read": "auth != null && $uid === auth.uid",
14       "name": {
15         ".validate": "newData.isString()"
16       },
17       "email": {
18         ".validate": "newData.isString()"
19       },
20       "mobile": {
21         ".validate": "newData.isString()"
22       },
23       "friendcount": {
24         ".validate": "newData.isNumber()"
25       },
26       "image": {
27         ".validate": "newData.isString()"
28       }
29     }
30   }
31 }
32 }
```

Simulator

Simulation type
 Read Write

Location
https://friends-4d4fa.firebaseio.com
/users/90b4c556-dda-4b78-9539-{:}

Data (JSON)
{
 "name": "value"
}

Authenticated

Provider
Anonymous

UID [?](#)
90b4c556-dda-4b78-9539-516b5:

Auth token payload [?](#)
{
 "provider": "anonymous",
 "uid": "90b4c556-dda-4b78-9539-516b52a96e0e"
}

Adding security rules for chat messages

In this section, we will enable the security rules for chat messages. The message details node contains two identifiers, as follows:

- **\$identifierKey:** The first is the identifier key, which is used for users under conversation, and this key is also stored in the user details node. In the following example, "-L-0uxNuc6gC95iQytu9" is the identifier key.
- **\$messageKey:** The second is the message key, which is generated when we push a new message in the node. In the following example, "-L-125Am3LVQQQiN_x1G" is the message key:

```
"message_details" : {  
    "-L-0uxNuc6gC95iQytu9" : {  
        "-L-125Am3LVQQQiN_x1G" : {  
            "message" : "Hello",  
            "receiverUid" : "2HIVnEJvN0003PtByU2ACBhSMDe2",  
            "senderUid" : "YnmOB5rTAwVERXcmMuJkHDEb4i92",  
            "timestamp" : 1511862854520  
        }  
    }  
}
```

We will define the following security rules for a message details node:

- **Read permission:** We give read permission only to the authentic users
- **Write permission:** We give write permission to authentic users and also check whether any new data exists before the data push happens
- **Validation:** We validate all the fields in the message so that data integrity is maintained for any new data insertion, as shown here:

```

"message_details": {
    ".read": "auth != null",
    "$identifierKey": {
        "$messageKey": {
            ".write": "auth != null && newData.exists()",
            ".validate": "newData.hasChildren(['message', 'receiverUid', 'senderUid', 'timestamp'])",
            "message": {
                ".validate": "newData.isString()"
            },
            "receiverUid": {
                ".validate": "newData.isString()"
            },
            "senderUid": {
                ".validate": "newData.isString()"
            },
            "timestamp": {
                ".validate": "newData.isNumber()"
            }
        }
    }
}

```

Finally, we will validate the new rules in the simulator to check whether they work:

The screenshot shows the Firebase Simulator interface. On the left, there is a code editor window titled "Simulated write allowed" containing the provided Firebase rules. The code editor has a green header bar with "Simulated write allowed", "DETAILS", and "CLEAR" buttons. The code itself is a JSON object with nested rules for "rules" and "message_details". Lines 11, 12, 14, 17, 20, 23, and 24 have green checkmarks next to them, indicating successful validation. On the right, there is a "Simulator" panel with tabs for "Simulator", "Data (JSON)", "Authenticated", and "Provider". The "Simulator" tab is active, showing "Simulation type" set to "Write", "Location" set to "https://friends-4d4fa.firebaseio.com", and "Data (JSON)" set to a JSON object with fields: "message": "value", "receiverUid": "dfsfs", "senderUid": "dfsts", and "timestamp": 664646. The "Authenticated" tab has a blue toggle switch turned on, and the "Provider" dropdown is set to "Anonymous".

Indexing users and friends

Firebase provides the querying and ordering of data upon collection of the nodes using any common child key. This query becomes slow when the data grows. To increase the performance, the Firebase recommends that you index within a particular child field. Firebase indexes the key to the server to increase the performance of the query.

As part of this section, we will implement an index on our user data to search or find friends, which is quite common in any social application. To achieve this, we will perform the following tasks:

- **Creating an index within the name field of the user data:** We provide an index within the name field of our user data. We will use the `.indexOn` tag for the name field, as follows:

```
"users": {
  ".read": "auth != null",
  ".indexOn": ["name"],
  "$uid": {
    ".write": "auth != null && $uid === auth.uid",
    "name": {
      ".validate": "newData.isString()"
    },
    "email": {
      ".validate": "newData.isString()"
    },
    "mobile": {
      ".validate": "newData.isString()"
    },
    "friendcount": {
      ".validate": "newData.isNumber()"
    },
    "image": {
      ".validate": "newData.isString()"
    }
  }
},
```

- **Creating a service for querying the data based on the text:** In this task, we will query the user data based on the search text. We will provide `orderByChild` as the name field of the user.

Here's the `friends-search.service.ts`:

```
import {Injectable} from '@angular/core';
import {AngularFireDatabase} from 'angularfire2/database';
import {Observable} from 'rxjs/Observable';
import {User} from './user';
```

```

import {FRIENDS_CHILD, USER_DETAILS_CHILD} from './database-constants';

/**
 * Friends search service
 */
@Injectable()
export class FriendsSearchService {

  constructor(private db: AngularFireDatabase) {}

  getSearchFriends(start, end): Observable<User[]> {
    return this.db.list<User>('/users',
      ref => ref.orderByChild('name').limitToFirst(10).
      startAt(start).endAt(end)
    ).valueChanges();
  }
}

```

- **Modifying the template:** We modify the app template to show the search result in the drop-down menu below the search text.

Here's the modified `app.component.html` file:

```

<h1 class="title">Friends - A Social App</h1>
<div class="nav-container">
  <nav class="navbar navbar-expand-lg navbar-light bg-color">
    <div class="collapse navbar-collapse" id="navbarNav">
      ...
      <div class="form-container">
        <form class="form-inline my-2 my-lg-0">
          <div class="dropdown">
            <input class="form-control mr-sm-2" type="text"
              (keyup)="onSearch($event)" name="searchText"
              data-toggle="dropdown" placeholder="Search friends..." aria-label="Search">
            <div class="dropdown-menu" aria-labelledby="dropdownMenuButton">
              <div class="list-group" *ngFor="let user of users">
                <div class="list-group-item list-group-item-action flex-column align-items-start">
                  <div class="d-flex w-100 justify-content-between">
                    <label>{{user?.name}}</label>
                    <button type="button" class="btn btn-light"
                      (click)="onAddFriend(user)">ADD</button>
                  </div>
                </div>
              </div>
            </div>
            <button class="btn btn-success my-2 my-sm-0"
              type="submit">Search</button>
          </form>
        </div>
      </div>
    </div>
  </nav>
</div>
<router-outlet></router-outlet>

```

- **Modifying the component:** When an app component is loaded we query for all the user in `ngOnInit()` method and when user click in search text box then user list appears with all the names. We also filter the list with the user type on the text box and then the `onSearch()` method is called and we query the Firebase database with the query range.

Here's the complete `app.component.ts` file as of now:

```

import {Component, OnInit} from '@angular/core';
import {AuthenticationService} from './services/authentication.service';
import {User} from './services/user';
import {FriendsSearchService} from './services/friends-search.service';

@Component({
  selector: 'app-friends',
  styleUrls: ['app.component.scss'],
  templateUrl: './app.component.html',
})
export class AppComponent implements OnInit {

  startAt: string;
  endAt: string;
  users: User[];
  searchText: string;
  authenticationService: AuthenticationService;

  constructor(private authService: AuthenticationService,
              private friendsSearchService: FriendsSearchService) {
    this.authenticationService = authService;
  }

  ngOnInit() {
    console.log(this.currentLoginUser);
    this.searchText = '';
    this.onSearchFriends(this.searchText);
  }

  onSearch(event) {
    const text = event.target.value;
    this.onSearchFriends(text);
  }

  onSearchFriends(searchText) {
    const text = searchText;
    this.startAt = text;
    this.endAt = text + '\uf8ff';
    this.friendsSearchService.getSearchFriends(this.startAt,
                                              this.endAt)
      .subscribe(users => this.users = users);
  }
}

```

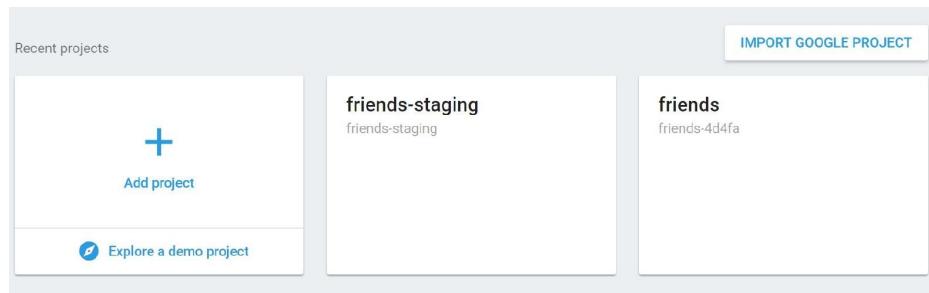


The \uffff character is a very high code point in the unicode range, which allows you to match all values that start with your search text.

Setting up multiple environments

When our application is ready to deploy, we will need to separate the Firebase projects for the development and production environments so that we can test our code changes in a development environment before deploying them into production. We will follow these steps to set up a separate environment:

- **Creating a new staging project in Firebase:** Because we cannot use the same Firebase features, such as database and storage, we will need to separate the production and staging environments. We will then create a new project with the `friends-staging` name; this project has the new Firebase environment variables:



- **Creating a new environment variable:** The new Firebase project has a new environment variable, and you can get configuration from the Firebase project. So, navigate to Project Overview | Project settings | Add Firebase to your web app.

Copy the content into the new environment file, as shown in the following code; we have two environment files for staging and production:

- The `environment.prod.ts` file with production is set as `true`, and this is used for the production environment:

```
export const environment = {
  production: true,
  firebase: {
    apiKey: 'XXXX',
    authDomain: 'friends-4d4fa.firebaseio.com',
    databaseURL: 'https://friends-4d4fa.firebaseio.com',
```

```
        projectId: 'friends-4d4fa',
        storageBucket: 'friends-4d4fa.appspot.com',
        messagingSenderId: '321535044959'
    };
};
```

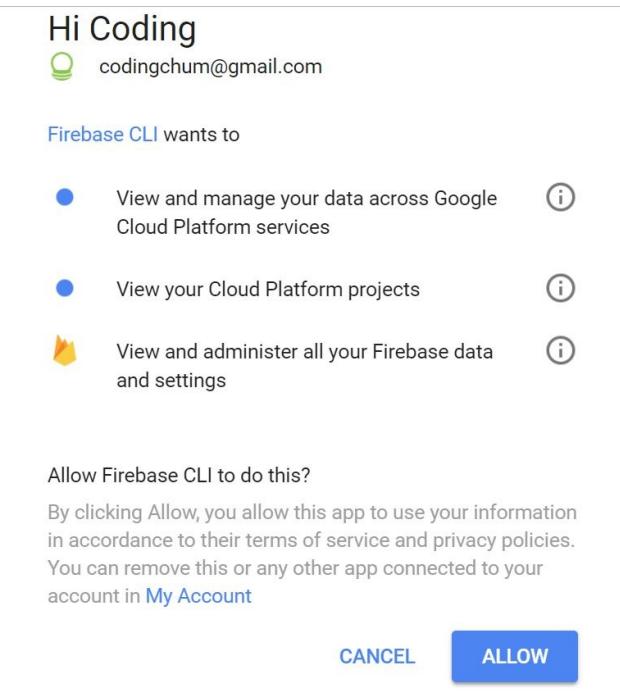
- The `environment.ts` file with production is set as `false`; this will be used for the staging environment:

```
export const environment = {
  production: false,
  firebase: {
    apiKey: 'XXXX',
    authDomain: 'friends-4d4fa.firebaseio.com',
    databaseURL: 'https://friends-4d4fa.firebaseio.com',
    projectId: 'friends-4d4fa',
    storageBucket: 'friends-4d4fa.appspot.com',
    messagingSenderId: '321535044959'
  }
};
```

- **Installing Firebase tools:** Once you create a new Firebase project, you will need to install the Firebase tools and log into the Firebase portal with the following command:

```
$ npm install -g firebase-tools
$ firebase login
```

The preceding command will open your Gmail permission page; clicking on ALLOW will give permission to list all the available projects:



- **Using the new Firebase project:** Once we give permission, we will need to add the available project based on the current environment in use. Suppose that we need to test a new feature under development, we can select a staging environment and give the alias name for the staging environment for future use. We can switch the environment using the alias name, as follows:

```
| $ firebase use --add  
| $ firebase use staging
```

This creates the `.firebaserc` file in the base project directory, which looks like this:

```
{  
  "projects": {  
    "default": "friends-4d4fa",  
    "staging": "friends-staging"  
  }  
}
```

Hosting the friends app in Firebase

Firebase supports hosting as a service, and application deployment is easy in Firebase. Most applications adopt a two-stage deployment, that is, first staging and then production. Once we test the application in the staging environment, we can deploy it into the production environment. The steps to deploy an application are as follows:

1. The first is to build the application, which creates a `dist` folder that has `index.html` and other required files. You can build for production just by adding a `--prod` option:

```
| $ ng build
```

2. The next step is to initialize the application project. We will execute the `init` command and select the Firebase features using the space key in the command prompt; for our friends application, we will use the database, storage, and hosting features of Firebase, and when we select the respective features, it creates a default database and storage rules:

```
| $ firebase init
```

It also creates the `firebase.json` file, which looks as follows:

```
{
  "hosting": {
    "public": "src",
    "ignore": [
      "firebase.json",
      "**/.*",
      "**/node_modules/**"
    ]
  },
  "database": {
    "rules": "database.rules.json"
  },
  "storage": {
    "rules": "storage.rules"
  }
}
```

- Finally, we deploy our application using the following command; once our application is deployed, we can take a look at the deployed application in the Firebase hosting our project:

```
| $ firebase deploy
```

The deployed application appears in the Firebase portal. You can see the deployed application in the Firebase portal by navigating to DEVELOP | Hosting, and on the right panel Domain and Deployment history appear as follows:

The screenshot shows the Firebase portal interface. At the top, there is a blue button labeled "CONNECT DOMAIN". Below it, the "Domain" section lists a single entry: "friends-staging.firebaseio.com" with a "Type" of "Default". In the bottom half of the screen, the "Deployment history" section is visible. It contains a table with columns: Status, Time, Deploy, and Files. One deployment is listed: "Current" (star icon), Oct 26, 2017, 11:07 AM, by "codingchum@gmail.com" (with ID "kxDkYB"), and 13 files. At the bottom of the deployment history table, there are pagination controls: "Rows per page: 10", "1-1 of 1", and navigation arrows.

Domain	Type	Status
friends-staging.firebaseio.com	Default	

Status	Time	Deploy	Files
★ Current	Oct 26, 2017 11:07 AM	 codingchum@gmail.com kxDkYB	13

- Finally, you can open your live application using the following command or by pasting the URL as <https://friends-staging.firebaseio.com>:

```
| $ firebase open
```

Summary

In this chapter, we covered the Firebase security mechanism. We added security rules for our friends application database to make our application more secure. We indexed the `name` field of our user node for our database so that the search query became faster. We then used a search API in our friends application. Finally, we created multiple environments for our application so that we were able to separate staging from production. We then deployed our application on to Firebase.

In the next chapter, we will learn about Firebase cloud messaging, Google analytics, and ads.

Growing Our Application Using Firebase

In this chapter, we will take a look at how Firebase provides cloud messaging to engage our users. We will add a Firebase cloud messaging feature to our application. We will cover Google analytics, which provides a good dashboard to analyze our application and take action accordingly, as this helps to improve our application further. Finally, we will discuss Google ads, which helps to monetize our application.

In this chapter, we will cover the following topics:

- Introduction to Firebase cloud messaging
- Adding FCM to our application
- Google data analytics
- Learning about Google Ads

Introduction to Firebase cloud messaging

Firebase cloud messaging (FCM) is a cross-platform service used to deliver messages reliably across different platforms. It uses the push methodology to send messages, and we can send up to 4 KB of data to the client. It supports many use cases, such as to engage users with promotional messages and send a message when the user is in the background.

It has the following two main modules:

- **Trusted server:** This server is used to send messages to clients and can be a Firebase console or a server SDK implementation.
- **Client application:** This includes client applications in the web, Android, or iOS. This receives messages from the trusted server.

Adding FCM to our application

In this section, we will configure FCM in our application. We will perform the following steps to configure FCM:

1. **Creating a manifest.json file:** Our first step will be to create a `manifest.json` file within the `src` folder. This file contains `gcm_sender_id`, which authorizes a client to a trusted FCM server and enables FCM to send messages to our application. For a desktop browser, the client ID—`103953800507`—remains fixed for a web application, so you don't need to change it.



The web manifest file is a simple JSON file, in which we can specify the configuration for our application, such as its name, display, and orientation.

Here's the code for `manifest.json`:

```
{  
  "name": "Friends",  
  "short_name": "Friends",  
  "start_url": "/index.html",  
  "display": "standalone",  
  "orientation": "portrait",  
  "gcm_sender_id": "103953800507"  
}
```

2. **Configuring manifest.json in index.html:** Once we create the manifest file, we include the file reference in `index.html`.

Then we include the modified `index.html`:

```
<!DOCTYPE html>  
<html>  
  <head>  
    <link rel="manifest" href="/manifest.json">  
  </head>  
  <body>  
    <app-friends>  
      Loading...  
    </app-friends>  
  </body>  
</html>
```

3. **Creating a service worker:** After we create a manifest file, we create a Firebase service worker to process incoming push messages from the

trusted server and to register our Firebase app with a messaging sender ID, which we can get from the Firebase portal by navigating to Project Overview > Project settings > CLOUD MESSAGING.



A service worker is a type of web worker that runs in the background and helps in push in notifications.

Here's the `firebase-messaging-sw.js` file as of now:

```
importScripts('https://www.gstatic.com/firebasejs/3.9.0.firebaseio-app.js');
importScripts('https://www.gstatic.com/firebasejs/3.9.0/firebase-
messaging.js');

firebase.initializeApp({
  'messagingSenderId': '807434545532'
});

const messaging = firebase.messaging();
```

4. **Referring to the manifest and service worker in angular-cli.json:** Next, we mention the service worker and manifest file's reference in `angular-cli.json`; the following is the modified `angular-cli.json`:

```
...
"apps": [
  {
    "assets": [
      "assets",
      "favicon.ico",
      "firebase-messaging-sw.js",
      "manifest.json"
    ],
    ...
  }
]
```

5. **Creating FCM Service:** This service class is used to receive the client token and insert the token to the Firebase database. It is also used to register for token refresh when a token expires. The method steps for creating this service class are as follows:

The first method step is to get user permission for notifications by an alert dialog, and once the user clicks on the Allow button, we call `getToken()` from the Firebase messaging object to get the token. We send this token to the Firebase database so that we can use this token in the future to send promotional messages to all of our users. We also create an

`onTokenRefresh()` method to refresh our token once it expires.

The second method step is to register for push notification messages by calling `onMessage()` when the application is in the foreground.

Here's the `fcm-messaging.service.ts` file as of now:

```
import {Injectable} from '@angular/core';
import {AngularFireDatabase} from 'angularfire2/database';
import {AngularFireAuth} from 'angularfire2/auth';
import 'firebase/messaging';

@Injectable()
export class FcmMessagingService {

  messaging = null;

  constructor(private angularFireDatabase: AngularFireDatabase,
  private afAuth: AngularFireAuth) {
    this.messaging = angularFireDatabase.app.messaging();
  }

  getPermission() {
    this.messaging.requestPermission()
      .then(() => {
        console.log('Permission granted.');
        this.getToken();
      })
      .catch((err) => {
        console.log('Permission denied', err);
      });
  }

  getToken() {
    this.messaging.getToken()
      .then((currentToken) => {
        if (currentToken) {
          console.log(currentToken);
          this.sendTokenToServer(currentToken);
        } else {
          console.log('No token available');
        }
      })
      .catch((err) => {
        console.log('An error occurred while retrieving token.', err);
      });
  }

  onMessage() {
    this.messaging.onMessage((payload) => {
      console.log('Message received. ', payload);
    });
  }

  onTokenRefresh() {
    this.messaging.onTokenRefresh(function () {
      this.messaging.getToken()
        .then(function (refreshedToken) {
          console.log('Token refreshed.');
        });
    });
  }
}
```

```

        this.sendTokenToServer(refreshedToken);
    })
    .catch(function (err) {
        console.log('Unable to retrieve refreshed token ',
        err);
    });
});
}

sendTokenToServer(token) {
    this.afAuth.authState.subscribe(user => {
        if (user) {
            const data = {[user.uid]: token};
            this.angularFireDatabase.object('fcm-
tokens/').update(data);
        }
    });
}
}

```

- 6. Registering Firebase messaging for update in app component:** Once we create service methods, we call user permission, token, and register for a message update in our app component as shown in the following

app.component.ts:

```

import {Component, OnInit} from '@angular/core';
import {AuthenticationService} from './services/authentication.service';
import {FcmMessagingService} from './services/fcm-messaging.service';

@Component({
    selector: 'app-friends',
    styleUrls: ['app.component.scss'],
    templateUrl: './app.component.html',
})
export class AppComponent implements OnInit {

    ...
    authenticationService: AuthenticationService;

    constructor(private authService: AuthenticationService,
                private friendsSearchService: FriendsSearchService,
                private fcmService: FcmMessagingService) {
        this.authenticationService = authService;
    }

    ngOnInit() {
        this.fcmService.getPermission();
        this.fcmService.onMessage();
        this.fcmService.onTokenRefresh();
    }
    ...
}

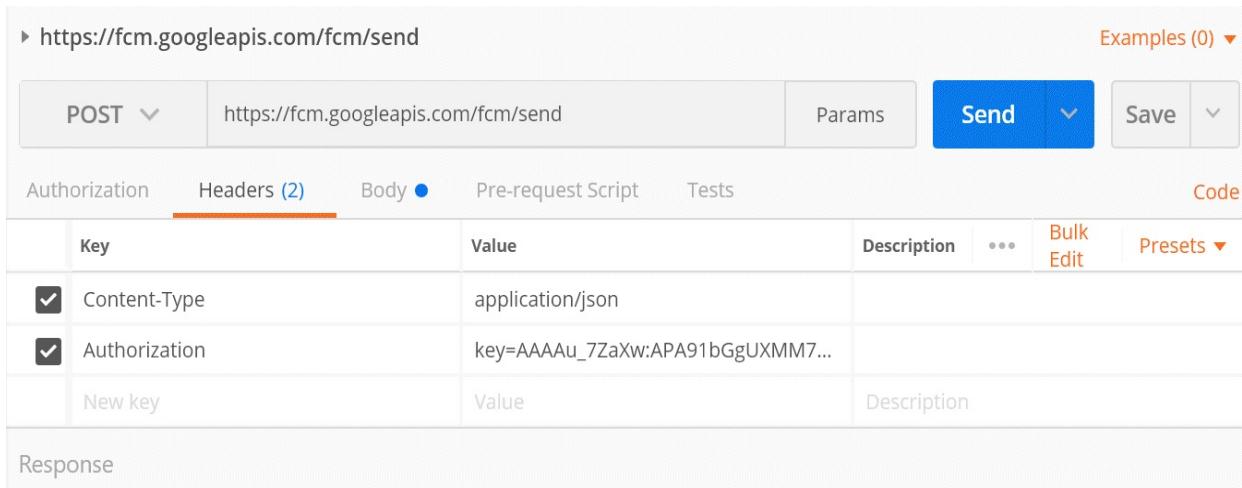
```

Finally, our application is now ready to send push notification messages. You can send messages either by a curl command or Postman request.

To send a push notification from Postman, we require the following details:

- **URL:** This is an FCM endpoint registered with our trusted server. Its unique URL is <https://fcm.googleapis.com/fcm/send>.
- **Content type:** This is a type of content sent to the server and in our case is JSON type as application/json.
- **Authorization:** This key is the server key for our Firebase project. You can find this key in our Firebase portal by navigating to Project Overview|Project settings|CLOUD MESSAGING|Server Key.
- **Body:** This contains the title, body, action, and the target sender ID. The sender token ID is saved in our Firebase database.

This is an example of a Postman request for a Headers tab:



A screenshot of the Postman interface showing a request configuration. The URL is https://fcm.googleapis.com/fcm/send. The method is set to POST. The Headers tab is selected, showing two entries: Content-Type: application/json and Authorization: key=AAAAu_7ZaXw:APA91bGgUXMM7... . There are also tabs for Body, Params, and Tests, along with a Send button and a Save dropdown.

Key	Value	Description	...	Bulk Edit	Presets
Content-Type	application/json				
Authorization	key=AAAAu_7ZaXw:APA91bGgUXMM7...				
New key	Value	Description			

The Postman for the Body tab will be as follows:

POST https://fcm.googleapis.com/fcm/send Params Save

Authorization Headers (2) **Body** ● Pre-request Script Tests

form-data x-www-form-urlencoded raw binary **JSON (application/json)**

```
1 {  
2   "notification": {  
3     "title": "Welcome to friends app!",  
4     "body": "Enjoy the cool new features in friends",  
5     "click_action": "http://localhost:4200"  
6   },  
7   "to": "fYa8N7mCZ6c:APA91bEEMREJZLRFwwNcle7Nb_Znd6gkPXpPVu_4mm00gwfV8_EDAyshxk59VMuV1bjHHssbZeubgv72  
-4sSq4VDjaGk-jvYw6f0Jq9HDArR3WKKDxoNYCsLHCx7v_YqcVB_TZYmuhs"  
8 }
```

The notification that appears on the bottom-right corner of your screen should look as follows:



Google data analytics

Google analytics is a free service offered by Google that provides statistics about the visitors and traffic to our website. It provides a more valuable input about the visitors and geographies. It also provides input about the behavior of the visitor when they use our website. The following are the steps for registering Google analytics to your application.

Creating a Google analytics account: We can create a Google analytics account with an existing Gmail account or a new Gmail account by performing the following steps:

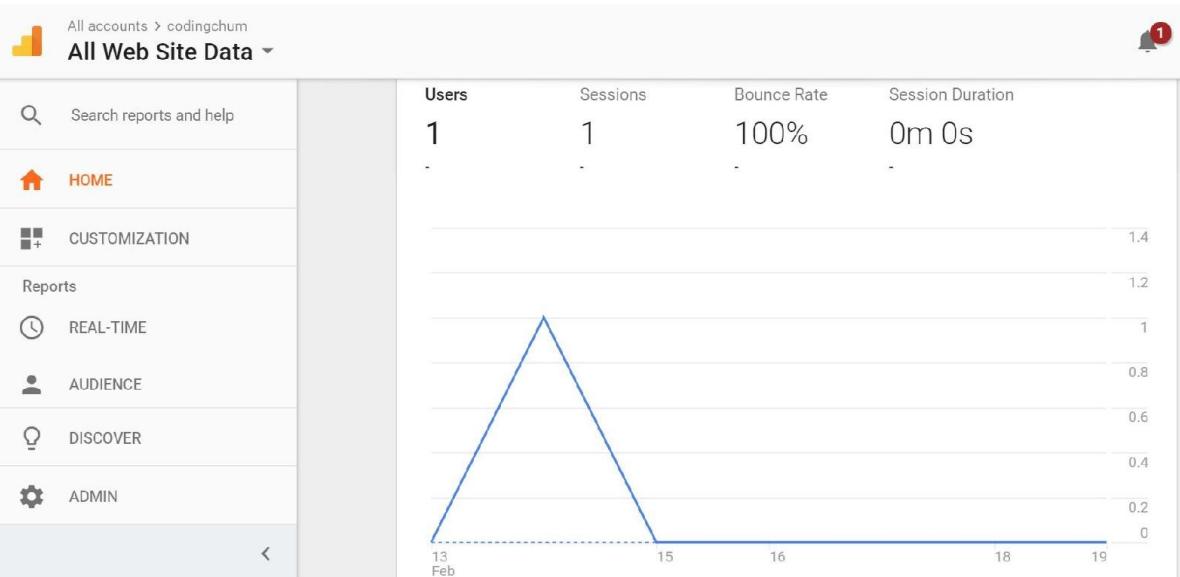
1. Open the browser and paste in the analytics URL (<https://analytics.google.com/analytics>)
2. Click on the SIGNUP button
3. Fill out your live application URL and form information
4. Click on the Get Tracking ID button

Integrating tracking code into our application: After a successful signup, we can integrate the generated global site tag into our application. Take a look at the following sample global site code in `index.html`:

```
...
<script async src="https://www.googletagmanager.com/gtag/js?id=UA-108905892-1"></script>
<script>
window.dataLayer = window.dataLayer || [];
function gtag(){dataLayer.push(arguments);}
gtag('js', new Date());

gtag('config', 'UA-108905892-1');
</script>
...
</head>
```

After a successful signup, our Google analytics dashboard should look like this:



Learning about Google adsense

Google adsense provides a platform to monetize our web application. Firebase ads are not supported for this web application, so we will work with Google ads to monetize our application. In this section, we will take a look at how to add ads to our application.

Creating an adsense account: We can create an adsense account with the existing Gmail account or create a new Gmail account by performing the following steps:

1. Open the browser and paste in the adsense URL (<https://www.google.com/adsense>)
2. Click on the SIGNUP button
3. Fill out your live application URL and address details
4. Click on the SUBMIT button

Adding the adsense script to our application: When you click on the Submit button, Google ads will provide steps to register your site with ads. It gives code to paste into the `index.html` of our application. Take a look at the following sample script to paste into the `index.html`: `<head>`

```
...
<script async src="//pagead2.googlesyndication.com/pagead/js/adsbygoogle.js">
</script>
<script>
(adsbygoogle = window.adsbygoogle || []).push({
  google_ad_client: "ca-pub-6342144115183345",
  enable_page_level_ads: true
});
</script>
...
</head>
```

After a successful signup, our Google adsense dashboard look as follows:

Estimated earnings

Today so far

\$0.00

Yesterday

\$0.00

Last 7 days

\$0.00

This month

\$0.00+\$0.00
vs same day last week+\$0.00
vs previous 7 days**Balance****\$0.00**Last payment:
\$0.00**Performance**

Last 7 days vs previous 7 days ▾

Page views

0+0

Page RPM

\$0.00+\$0.00

Impressions

0+0**Ad units**

Last 7 days vs previous 7 days ▾

There is no data available at the moment.



Summary

In this chapter, we covered Firebase cloud messaging. We integrated FCM into our application, which helps to engage our users. We also covered Google analytics and saw how to enable analytics in our application. This gives a good perspective about the application's usage. Finally, we discussed Google adsense, which helps to monetize our application.

In the next chapter, we will discuss the **Progressive Web App (PWA)** and add a few features to make our application PWA-compliant.

Transforming Our App into a PWA

Progressive Web App (PWA) is a new way to develop a web application. As a part of this chapter, you will learn about PWA and explore features that make an application PWA-compliant. As a part of this, we will add our friends application to mobile home screens, and this will make our friends application a part of other mobile-native apps. We will also cover offline mode for our application so that we can show data for our users to browse. Finally, we will audit our application in the **Lighthouse** tool, which provides a good insight into our Progressive Web Application.

In this chapter, we will cover the following topics:

- Introduction to PWA
- Introduction to service worker
- Adding our application to phone home screens
- Enabling offline mode
- Compliance testing using Lighthouse

Introduction to PWA

PWA is a web app that uses enhanced features to deliver a mobile app-like experience to the users. This web application meets certain requirements and is deployed to the Firebase server, which is accessible through URL and indexed by the Google search engine. Progressive web application development is a paradigm shift in recent years to make your web application universally available.

The following are a few of the features that make an application PWA-compliant:

- **Power of website and app:** The application is optimized to work perfectly well on mobiles and in browsers. It has all the capabilities of a mobile app, such as offline mode and push notifications.
- **App store not required:** Similar to a website, we don't require an app store and this is readily available for use with the latest software update.
- **App-Like:** The web application looks like a mobile application. It appears along with other mobile apps and occupies the full screen as a normal app.
- **Connectivity-Independent:** These apps are independent of network types. They work quite well on weak networks, which gives users a seamless experience.
- **Additions to the home screen:** This allows a user to add our website to their home screen so that they become a part of the app family. Users can frequently launch the application without opening the browser.
- **Safe:** This app works on HTTPS, so they are safe from attacks or hacks.
- **Push Notification:** With the advent of service worker, it is possible to send push notifications to a web application. This is really helpful in engaging the user with our application.
- **Searchable:** Similar to a website, this app is searchable using Google Search. We can optimize our website with keywords so that PWA apps are recognizable and easily searchable by users.
- **Link able:** This kind of app is easily shareable via links just like a normal

web application.

Introduction to service worker

Service worker is a script that runs in background. This background script does not interact with DOM elements. This helps support features such as push notification and offline mode, and service worker will be greatly enhanced in future to support other new features.

The following are the prerequisites for service worker:

- **Browser support:** Service worker is supported in the Chrome, Firefox, and Opera browsers, and the support will be extended for other browsers soon.
- **HTTPS support:** **Hyper text transfer protocol secure (HTTPS)** is a secure version of HTTP and is one of the prerequisites for PWA. This ensures that all communication between browser and server is encrypted.

Adding our application to phone home screens

This is one of the most important features of Progressive Web Apps and provides many advantages, as follows:

- **Easier access:** Users normally put their most-used app on the home screen, as this provides easier access to the application
- **Engagement:** A user can engage with our application more frequently

The steps involved in making our web application appear on the home screen are as follows:

1. In order to give our web application a mobile app appearance, we will modify the `manifest.json` file as shown in the following code.
2. Open the friends application in your phone's Chrome browser using the deployed application URL available from your Firebase portal. The page prompts you with an option to add the application to the home screen.

Here's the `manifest.json` file:

```
{
  "name": "Friends",
  "short_name": "Friends",
  "icons": [
    {
      "src": "/assets/images/android-chrome-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    }
  ],
  "theme_color": "#689f38",
  "background_color": "#689f38",
  "start_url": "/index.html",
  "display": "standalone",
  "orientation": "portrait",
  "gcm_sender_id": "103953800507"
}
```

Take a look at the following detailed description of properties:

- `name`: This name appears when the add to home screen banner appears and

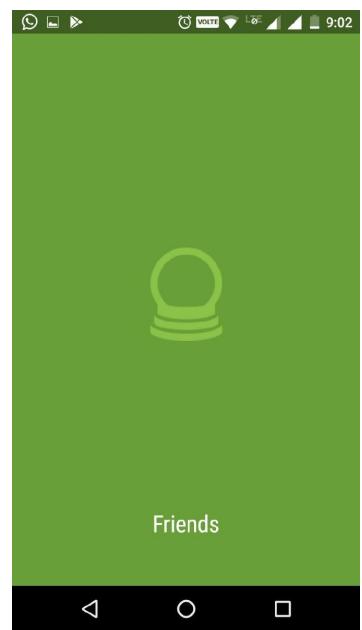
Chrome gives the option to modify the name dynamically.

- `short_name`: This appears below the application icon in the phone home screen. In our application, `name` and `short_name` are the same.
- `icons`: According to the PWA standard, the recommended icon size is 192 x 192, and this icon appears on phone home screens.
- `background_color`: The specified background color appears as the icon's background color.
- `theme_color`: When a user clicks on the friends application on the home screen, this color appears in the splash screen of your mobile application.
- `display`: Android Chrome gives native style when a page is opened, as it removes the navigation bar and give up tab to task switcher.
- `start_url`: This page is `index.html` in web applications, and typically this is our home page.
- `orientation`: This enforces either portrait or landscape orientation.

Our application on a phone home screen looks like this:



The splash screen for our app screen looks as follows:



Enabling offline mode

In this section, we will cover how to enable offline mode for our application, which helps the user open our web application without an internet connection.

In order to support offline mode, we have to cache resources in the client browser, and for that, we use the precache plugin to cache our resources using service worker. It creates the service worker file using **sw-precache**. The steps involved are as follows:

1. **Installing the plugin:** The first step is to install the precache plugin in our current project using the following command:

```
| $npm install --save-dev sw-precache-webpack-plugin
```

2. **Creating precache JavaScript:** The precache plugin uses the precache configuration file to define resources to cache in the client browser. For more details about the precache plugin, refer to <https://github.com/goldhand/sw-precache-webpack-plugin>.

Here's the complete `precache.config.js`:

```
var SWPrecacheWebpackPlugin = require('sw-precache-webpack-plugin');
module.exports = {
  navigateFallback: '/index.html',
  navigateFallbackWhitelist: [/^(!\//)],
  stripPrefix: 'dist',
  root: 'dist/',
  plugins: [
    new SWPrecacheWebpackPlugin({
      cacheId: 'friend-cache',
      filename: 'service-worker.js',
      staticFileGlobs: [
        'dist/index.html',
        'dist/**.js',
        'dist/**.css'
      ],
      stripPrefix: 'dist/assets/',
      mergeStaticsConfig: true
    })
  ]
};
```

3. **Configuring package.json:** Once we create the configuration file, we need to create a new build tag called `pwa` and refer the cache file in `package.json`.

Here's the modified `package.json`:

```
...
"scripts": {
  "ng": "ng",
  "start": "ng serve",
  "build": "ng build",
  "test": "ng test --sourcemaps false",
  "coverage": "ng test --sourcemaps false --watch=false --code-coverage",
  "lint": "ng lint",
  "e2e": "ng e2e",
  "pwa": "ng build --prod && sw-precache --root=dist --config=precache-config.js"
}
...
```

4. **Registering service worker:** Once we create the new build, we need to register the service worker that is created by the precache plugin in `index.html`, as follows. Here's the modified `index.html`:

```
...
<body>
  <app-root></app-root>

  <script>
    if ('serviceWorker' in navigator) {
      console.log("Will the service worker register?");
      navigator.serviceWorker.register('/service-worker.js')
        .then(function(reg){
          console.log("Service Worker Registered");
        }).catch(function(err) {
          console.log("Service Worker Not Registered: ", err)
        });
    }
  </script>
</body>
...
```

5. **Running the new build script:** Once we configure the service worker, we can run the production build using the following command; this will create all the files with service worker in their distribution folder:

```
| $ng pwa
```

5. **Deploying:** Finally, we deploy the newly created files to Firebase. Once they are deployed, we can open the application on our phone's home screen, and it will cache all the required resources in the client browser.

Compliance testing using Lighthouse

Lighthouse is an open source automated tool. It audits the application on performance, accessibility, progressive web apps, and so on. This is available with Chrome's developer tools in the Audits tab. So, go to the Chrome developer tool, then open the Audits tab, and click on the Perform an audit... button

In order to see improvements in our application, we can use this tool in two phases:

- **Without any PWA changes:** We can run this tool without any of the preceding changes in our application and take a look at the performance. Since our application does not comply with the PWA criteria, our score will not be good.

Take a look at the following screenshot showing the score when we run Lighthouse—it fails in five audits and the score is shown in red:

Progressive Web App

These audits validate the aspects of a Progressive Web App, as specified by the baseline [PWA Checklist](#).

5 failed audits

▶ Does not register a Service Worker	✗
▶ Does not respond with a 200 when offline	✗
▶ User will not be prompted to Install the Web App	✗
Failures: No manifest was fetched, Site does not register a Service Worker, Manifest start_url is not cached by a Service Worker.	
▶ Is not configured for a custom splash screen	✗
Failures: No manifest was fetched.	
▶ Address bar does not match brand colors	✗
Failures: No manifest was fetched, No '<meta name="theme-color">' tag found.	

- **With PWA changes:** Now, apply all the PWA changes as discussed in this chapter in our friends application, then run this tool and take a look at our audit performance. As shown in the following screenshot, our PWA score is 82 and is shown in green:

Progressive Web App

These audits validate the aspects of a Progressive Web App, as specified by the baseline [PWA Checklist](#).

82

2 failed audits

- ▶ Does not provide fallback content when JavaScript is not available ✗
The page body should render some content if its scripts are not available.
- ▶ Is not configured for a custom splash screen ✗
Failures: Manifest does not have icons at least 512px.

▶ 9 Passed Audits

▶ Manual checks to verify

Summary

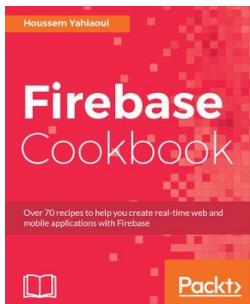
In this chapter, we discussed progressive web apps. We covered PWAs and all their key features. We discussed service worker, which supports push notification, offline mode, and so on. We enhanced `manifest.json` for our web application and added our application to a phone home screen. We enabled the offline cache using the `sw-precache` plugin. Finally, we used the Lighthouse tool to evaluate our application's PWA compliance.

Finally, we come to the end of our book, but this is not the end of web application development. This book introduced you to a practical approach to Angular and Firebase. You need to carry forward this knowledge and develop another live application; this will give you a lot of confidence.

All the best for your next application using Angular and Firebase!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

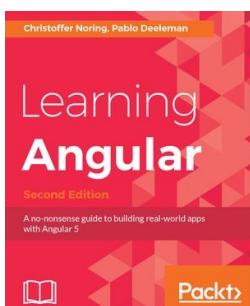


Firebase Cookbook

Houssem Yahiaoui

ISBN: 978-1-78829-633-5

- Use Firebase Diverse Authentication systems
- Integrate easy, secure File Hosting using Firebase Storage services
- Make your application serverless using Firebase Cloud Functions
- Use the powerful Firebase Admin SDK for privilege management
- Use Firebase within NativeScript apps for cross-platform applications
- Modify, structure, save and serve data in and from Realtime Database
- Get acquainted with the newly introduce Cloud Firestore, a scalable database for your web and mobile applications



Learning Angular - Second Edition

Christoffer Noring, Pablo Deelman

ISBN: 978-1-78712-492-9

- Set up the workspace and the project using webpack and Angular-Cli
- Explore the features of TypeScript and organize the code in ES6 modules
- Work with HTTP and Data Services and understand how data can flow in the app
- Create multiple views and learn how to navigate between them
- Make the app beautiful by adding Material Design
- Implement two different types of form handling and its validation
- Add animation to some standard events such as route change, initialization, data load, and so on
- Discover how to bulletproof your applications by introducing smart unit testing techniques and debugging tools

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!