ANGULARDART

SUCCINCTLY

BY **ED FREITAS**

SUCCINCTLY EBOOK SERIES

Syncfusion®

AngularDart Succinctly

By

Ed Freitas

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA

ISBN: 978-1-64200-203-4

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, vice president of content, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Ed Freitas is a consultant on software development related to financial process automation, accounts payable processing, and data extraction.

He loves technology and enjoys playing soccer, running, traveling, life hacking, learning, and spending time with his family.

You can reach him at https://edfreitas.me.

Acknowledgments

Many thanks to all the people who contributed to this book, including the amazing Syncfusion team that helped this book become a reality—especially Jacqueline Bieringer, Tres Watkins, and Graham High.

The manuscript manager and technical editor thoroughly reviewed the book's organization, code quality, and overall accuracy—Jacqueline Bieringer from Syncfusion and James McCaffrey from Microsoft Research. Thank you.

This book is dedicated to *Puntico*—may your journey be blessed—and to my father. For everything you did, for everyone you loved—thank you.

AngularDart

AngularDart is an open-source, web app framework created by Google that combines the power of Angular with the Dart programming language, focusing on productivity, performance, and stability.

AngularDart has been battle-tested internally at Google, and it is used to build many mission-critical apps that generate much of Google's revenue, such as Google Ads.

AngularDart is a port of Angular to Dart. Angular uses TypeScript, whereas AngularDart uses Dart. TypeScript and Dart share common objectives—both make building large-scale web apps easier. However, their approaches are quite different.

TypeScript maintains backward compatibility with JavaScript, whereas Dart departs from certain parts of JavaScript's syntax and semantics to reduce issues and improve performance.

The creators of AngularDart believe that both Dart and TypeScript give web developers more programming language choices when it comes to building high-performing web apps, using a robust framework like Angular.

One of the great aspects of choosing AngularDart as a technology is that Dart was designed by Google engineers to be easy to write development tools for, well-suited for modern app development, and capable of delivering high-performance implementations.

Another very interesting aspect of Dart is that it feels very familiar to any developer coming from a Java, C#, or any C-family programming language background. Dart is also a very mature programming language that executes quickly during runtime, and it is used by Google's Flutter framework for creating cross-platform, high-performing mobile apps.

If you'd like to learn more about how you can write cross-platform and high-performing mobile apps using Dart and Flutter, Syncfusion's *Succinctly* series has you covered with *Flutter Succinctly*.

Throughout this book, we will learn how to build a web app (**DocExpire**) that keeps track of important documents that have an expiration date, such as passports, driver licenses, and credit cards. It's a nice and convenient app that we can use to remind us when we need to renew important documents. We'll create this app with AngularDart and Firebase.

AngularDart is a great choice for developing web apps with a stable, performing, productive, and state-of-the-art web framework using Angular with Dart as a programming language.

Without any further ado, let's dive right into AngularDart development!

Chapter 1 Setup

Before starting

This book is going to follow a very practical and hands-on approach to learning AngularDart by building the **DocExpire** web app into a fully functional application.

If you have not done any previous Angular development, I highly recommended that you first get acquainted with AngularDart's architecture and the basic concepts and terminology that are going to be covered. You can download the project's full source code here.

Setup prerequisites

To set up a new AngularDart project, we first need to install the Dart SDK. My computer uses the Windows operating system, so the steps that follow are specific to setting up an AngularDart project on Windows. However, you should be able to easily install the Dart SDK on any other operating system by following the official Dart SDK documentation.

To install the Dart SDK on Windows, you can download the SDK as a zip file, or you can use the Chocolatey package manager (my preferred and recommended option) and execute the following command, preferably as an administrator. You'll need to have Chocolatey installed.

Code Listing 1-a: Command to Install the Dart SDK Using Chocolatey

```
choco install dart-sdk
```

If you have the Dart SDK already installed like I do, you can upgrade to the latest version by running the following command, preferably as an administrator.

Code Listing 1-b: Command to Upgrade the Dart SDK Using Chocolatey

```
choco upgrade dart-sdk
```

Before you execute this command, make sure you have no processes running that use the Dart SDK. When you execute this command, you will see an output similar to the following one.

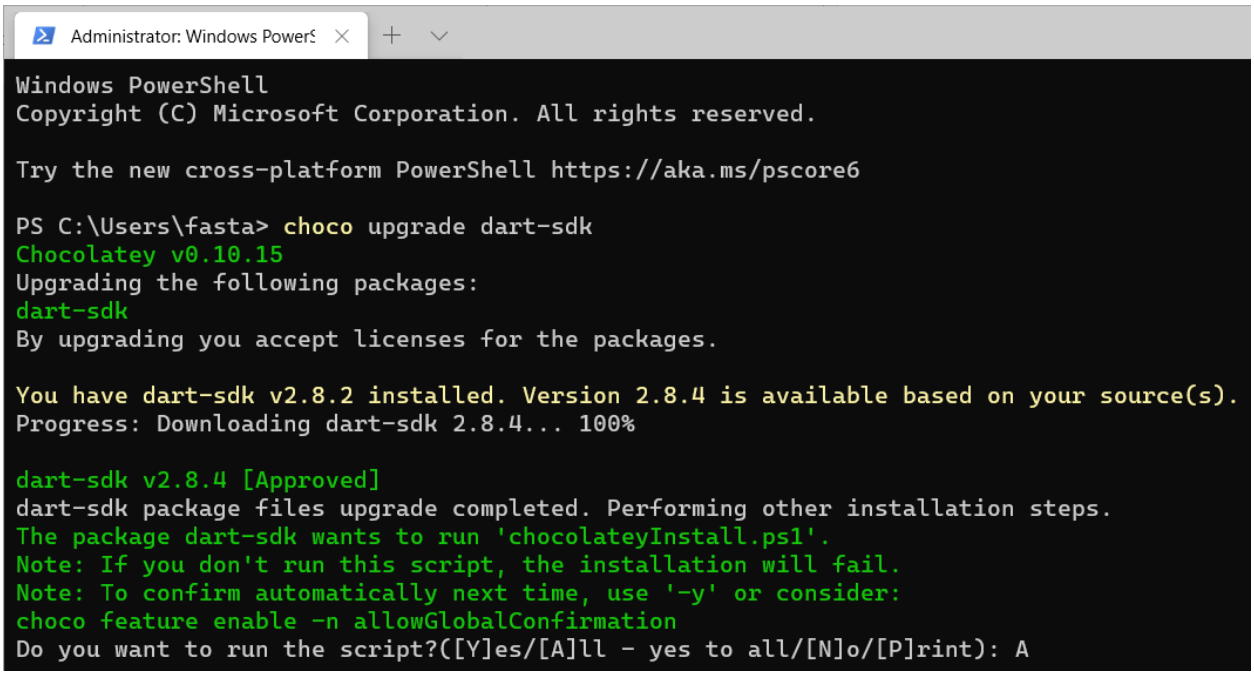

Figure 1-a: Upgrading the Dart SDK

When prompted to run the script, type **Y** for yes or **A** for all. Once the Dart SDK has been upgraded or installed, the next thing to do is to set up your editor of choice. In my case, I'll be using Visual Studio Code (VS Code).

To set up VS Code for working with Dart, it is necessary to install the Dart extension, which provides tools for editing, running, and working with AngularDart web apps.

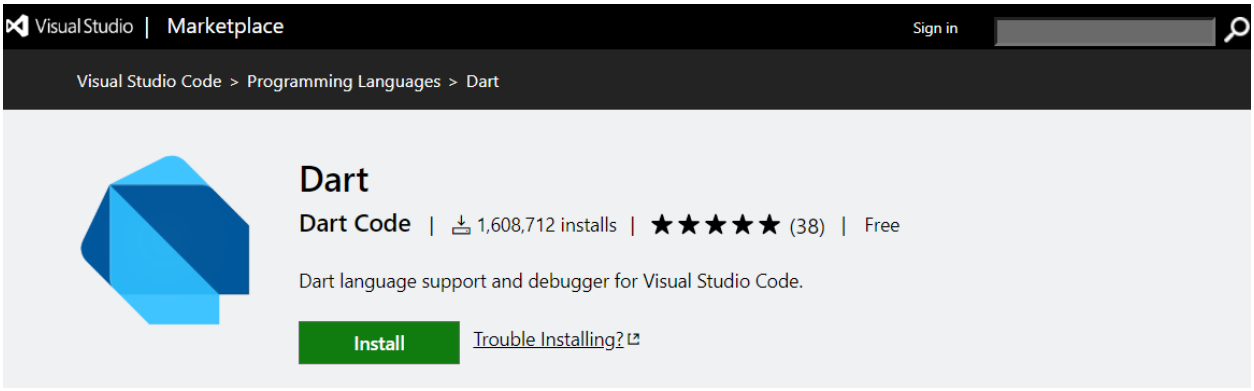

Figure 1-b: The Dart VS Code Extension

With the Dart extension installed, the next prerequisite is to install webdev, which will be used to compile AngularDart code into JavaScript and serve our web app.

To install webdev globally on your machine, you'll need to run the following Dart utility command, preferably as an administrator.

Code Listing 1-c: Command to Install webdev

```
pub global activate webdev
```

With webdev installed, we are ready to set up our AngularDart project.

AngularDart project setup

We are now ready to create our AngularDart project, which we will use as a basis to create the **DocExpire** web application we will be building throughout this book.

First, create a folder on your machine where the project will reside. Then, with VS Code open, click the **View** menu, and then the **Command Palette** option.

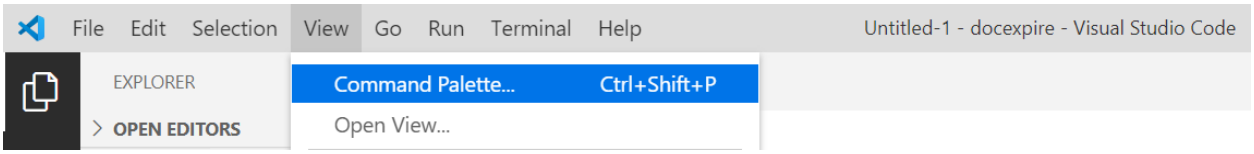

Figure 1-c: Command Palette Option - VS Code

You'll see the following options. Type in the word **Dart**.

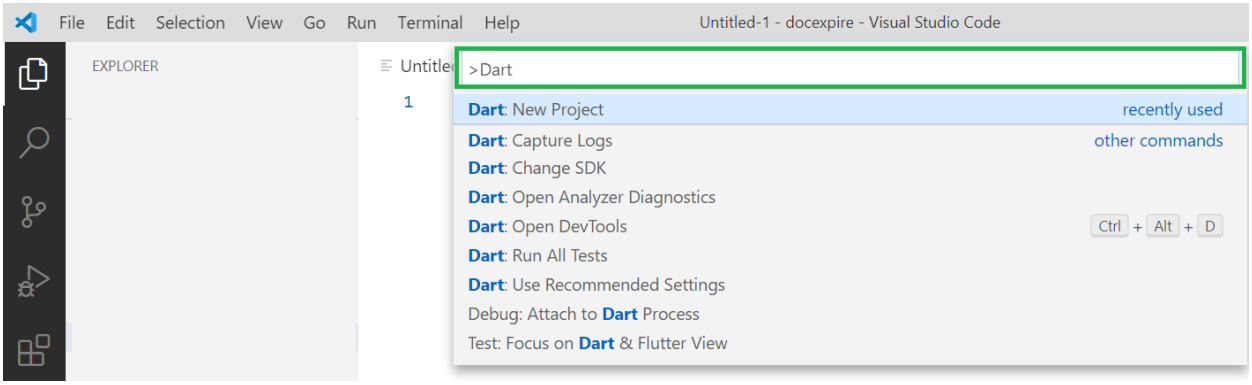

Figure 1-d: Command Palette Option (using Dart) - VS Code

Click the **Dart: New Project** option, which will then present the following project templates.

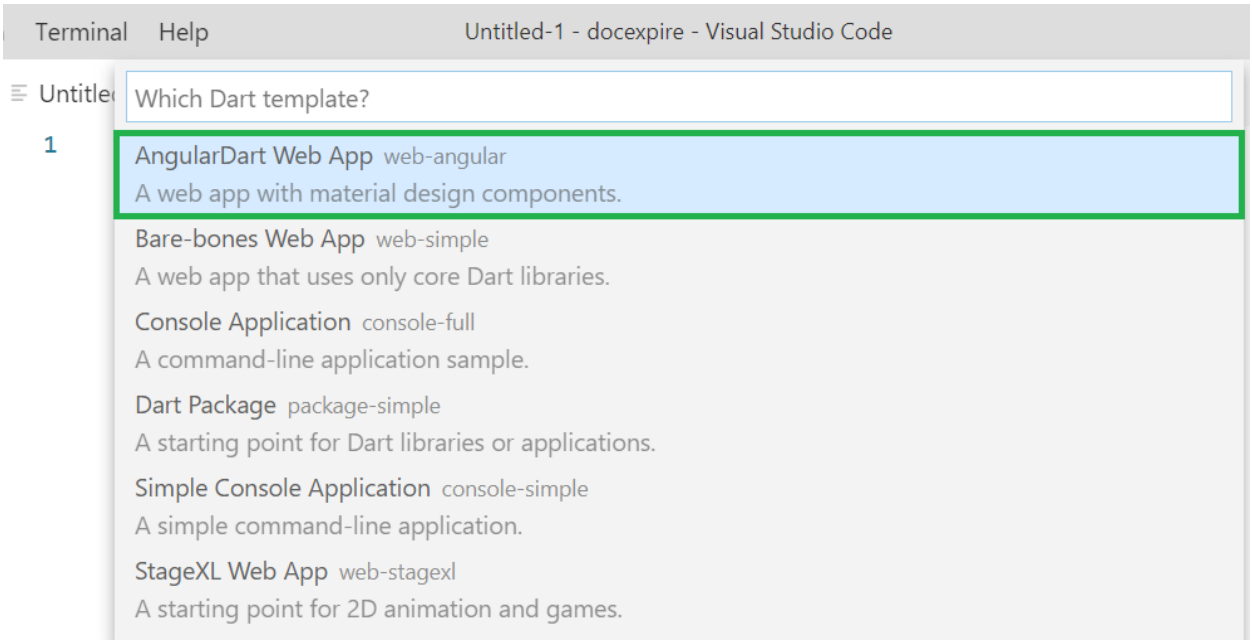

Figure 1-e: Command Palette Option (Dart Templates) - VS Code

Select the **AngularDart Web App** project template. You'll be asked to enter a name for your project. Dart project names should all be in lowercase. Let's name it **docexpire**.

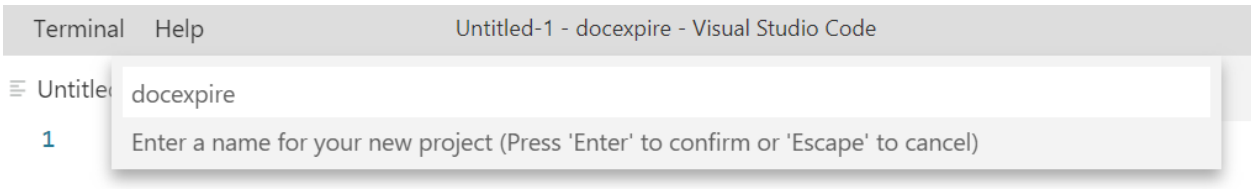

Figure 1-f: Command Palette Option (Dart Project Name) - VS Code

Once you have entered the name of the project, select the folder that was previously created, where the project will reside.

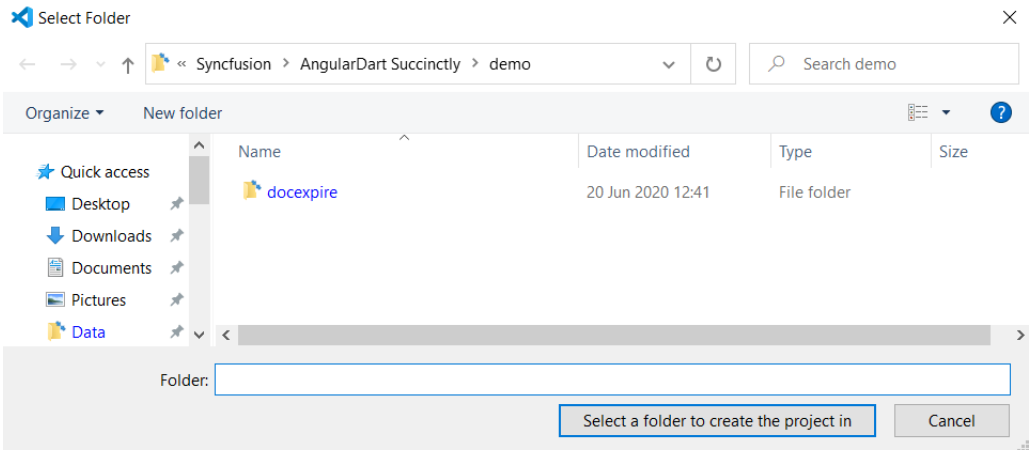

Figure 1-g: Selecting the AngularDart Project Folder - VS Code

The project will be scaffolded and created. Once the project creation steps have finished, you will see your project created within VS Code as follows.

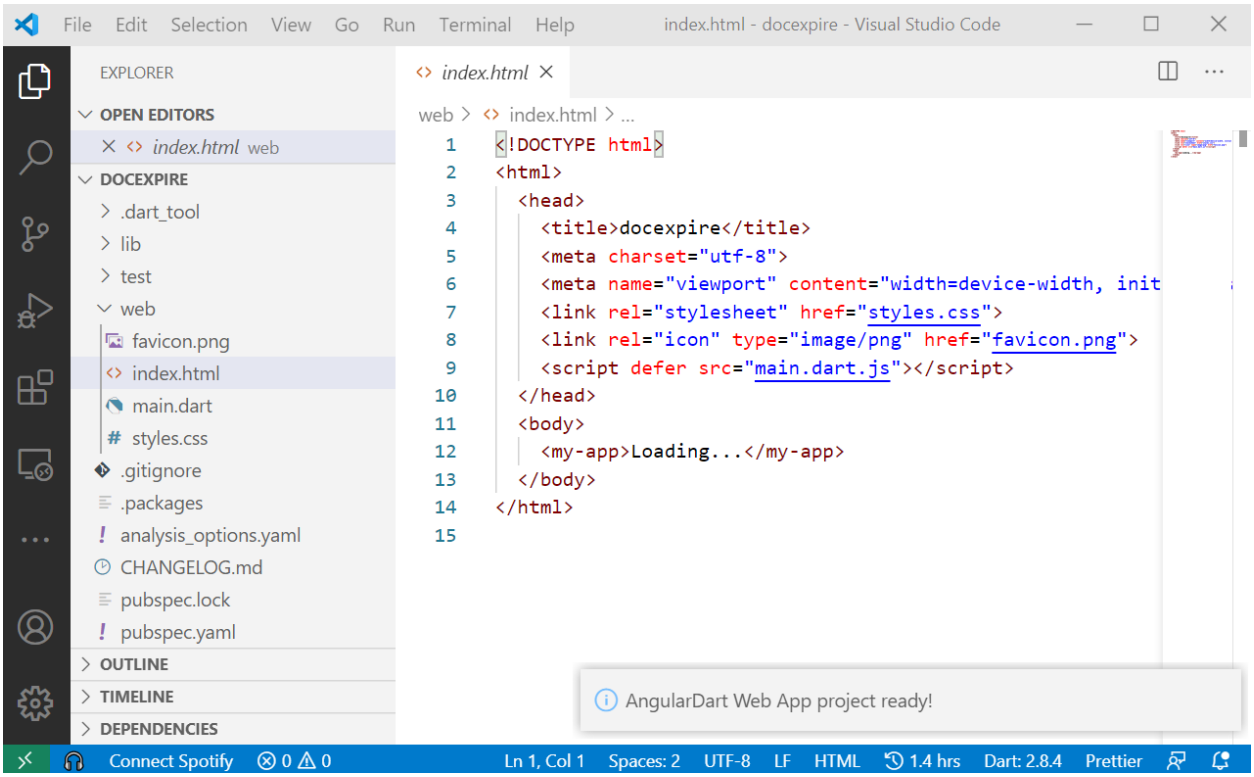

Figure 1-h: AngularDart Project Created - VS Code

Project verification

The project has been scaffolded—this means our app has been created using the default AngularDart web application template.

It's now time to verify that the project creation was successful. To do that, we need to run the application. We can do this by using the built-in terminal within VS Code. Select the **New Terminal** option from the **Terminal** menu.

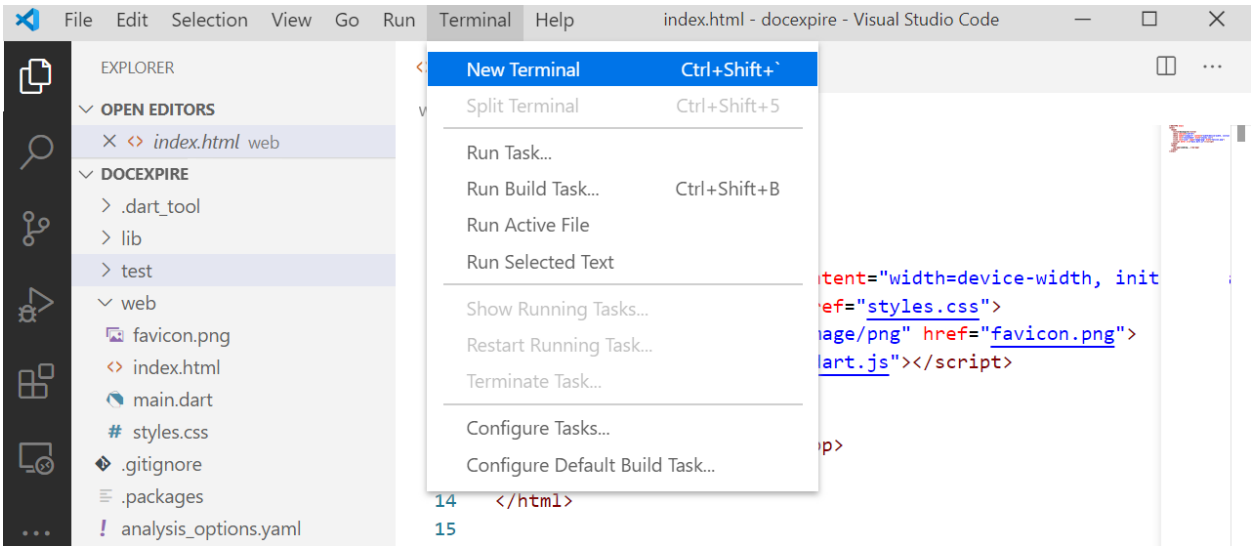

Figure 1-i: Opening the Built-In VS Code Terminal

The built-in terminal will open and become visible within VS Code, which can be seen as follows.

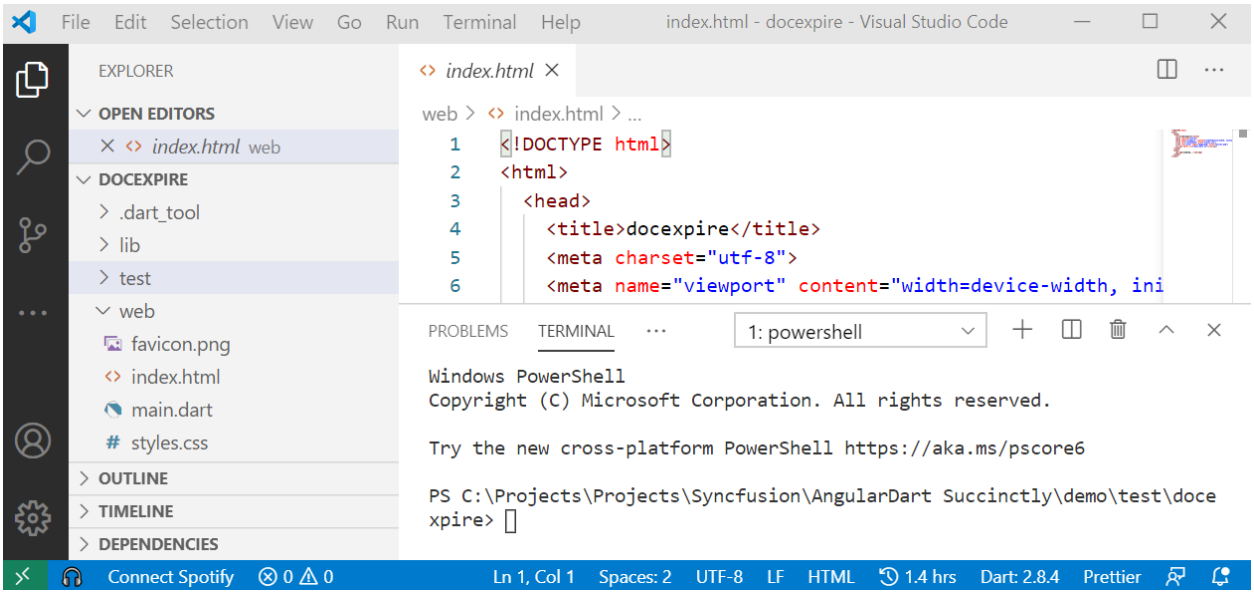

Figure 1-j: VS Code Terminal

Now we can execute the application and verify that it has been created successfully by running the following command.

Code Listing 1-d: Command to Run and Serve the Web Application

```
webdev serve
```

After executing this command, VS Code will connect to the Dart compiler to compile and build the application. Notice that a command line pop-up window might appear, as shown in Figure 1-k. This indicates that the Dart compiler (**dart.exe**) is running.

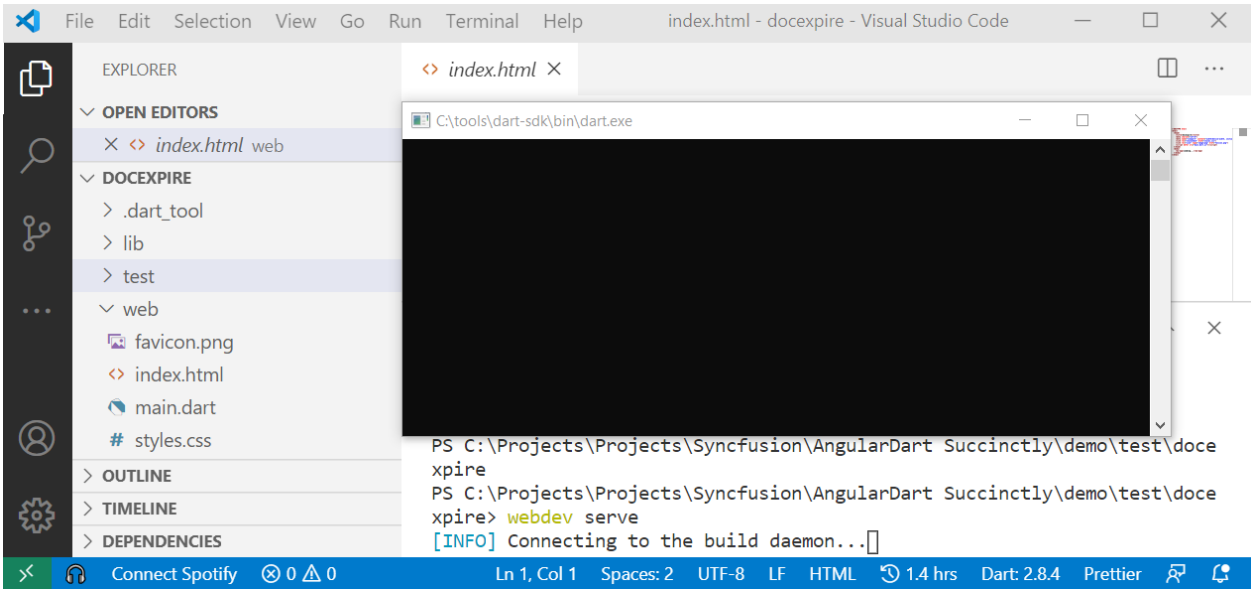

Figure 1-k: Connecting to the Dart SDK Runtime

Once the connection to the Dart SDK has been initialized, the app's prebuilt code is compiled from Dart into JavaScript and the application is served on the URL, as highlighted in the following figure.

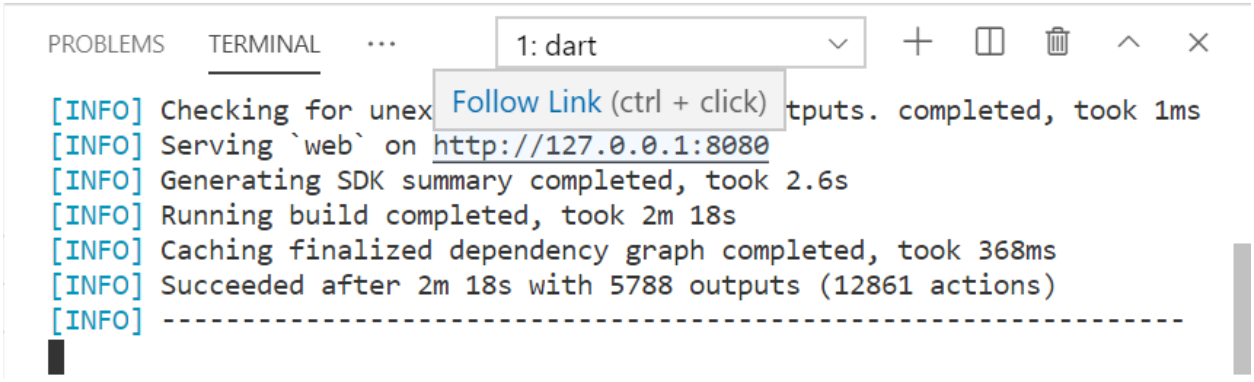

Figure 1-l: The Web App Running (VS Code Terminal)

To open the application, we can press the **Ctrl** key and click the URL link as indicated within the terminal. A browser window will open, and we'll be able to see the application running as follows.

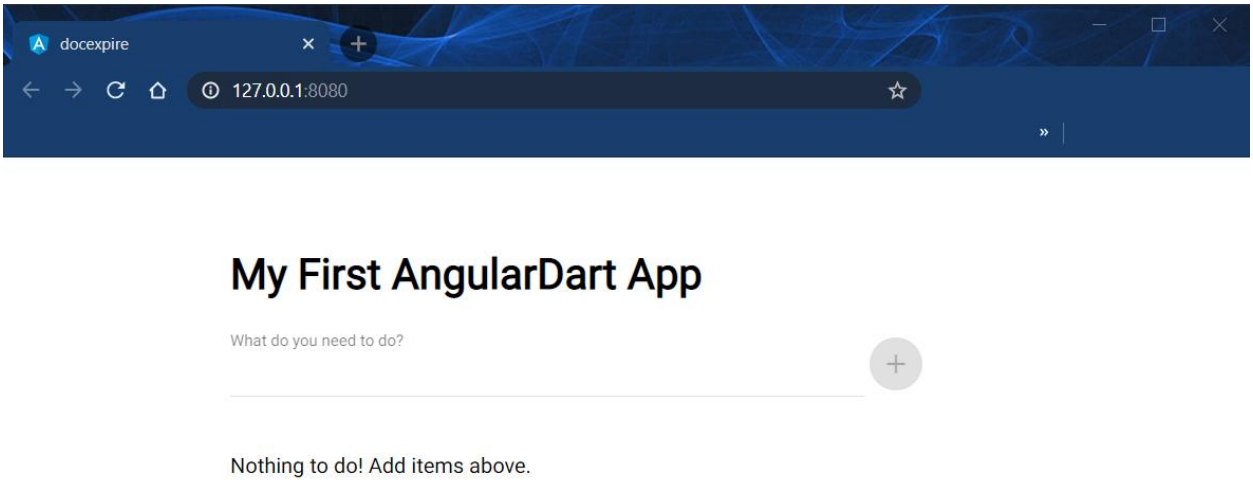

Figure 1-m: The Web App Running (Web Browser)

Awesome! We have now checked that the project has been successfully created and scaffolded.

Summary

Throughout this chapter, we have explored what AngularDart is and how to get started. We were able to do this by installing the Dart SDK and Dart extension for VS Code.

By using the VS Code Dart extension, we were able to set up an AngularDart project using a predefined template. We were able to verify it by running it through the built-in VS Code command line.

In the next chapter, we will slightly modify the structure of the project created by adding some additional subfolders and files, and also make some changes to the **pubspec.yaml** and **index.html** files, so our web app can start taking shape.

Chapter 2 Project Structure

Overview

To develop our application, we'll need to make some changes to the project structure, as we'll need to add a CSS framework. We'll also need to add some additional Dart packages and files that will be used within our application.

Dart packages

Let's start by making some changes to the default **pubspec.yaml** file that was automatically created when the project was scaffolded. Within VS Code, locate the **pubspec.yaml** file using the project explorer. Click on the file name to open it.

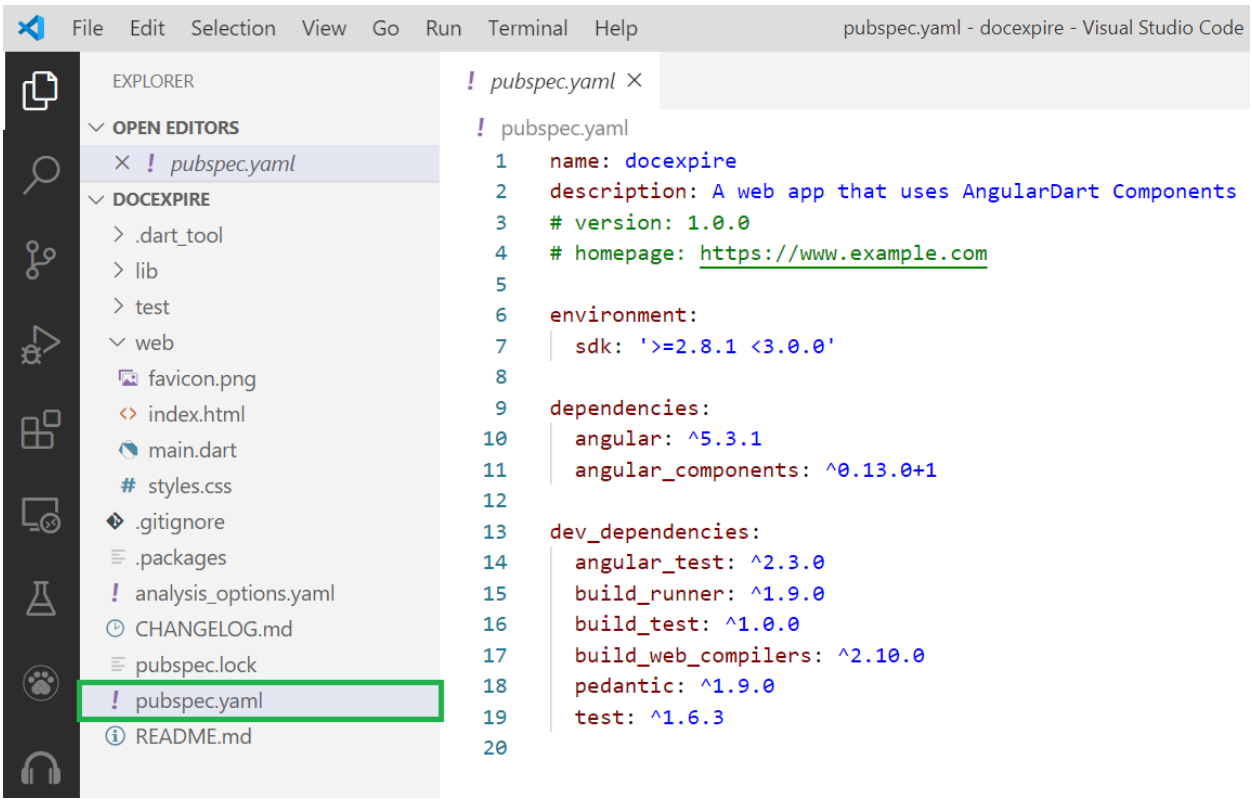

Figure 2-a: The Default pubspec.yaml File

The **pubspec.yaml** file contains the definitions of which Dart SDK version the project uses, as well as packages and development dependencies required to compile and build the project. Let's modify this file and paste the content outlined in Code Listing 2-a.

Code Listing 2-a: New Content of the pubspec.yaml File

```
name: docexpire
description: A web app that uses AngularDart
# version: 1.0.0
# homepage: https://www.example.com

environment:
  sdk: '>=2.8.1 <3.0.0'

dependencies:
  angular: ^5.3.1
  angular_forms: any
  firebase: ^7.3.0
  #angular_components: ^0.13.0+1

dev_dependencies:
  angular_test: ^2.3.0
  build_runner: ^1.9.0
  build_test: ^1.0.0
  build_web_compilers: ^2.10.0
  pedantic: ^1.9.0
  test: ^1.6.3
```

As you've noticed, we will be using the **angular_forms** and **firebase** packages as dependencies. Please note that prefixing a version number with the carat (**^**) character will accept versions even if the minor release number isn't an exact match.

We don't need **angular_components**; we will replace it with a lighter and easier-to-use Material Design for Bootstrap library.

Once you have pasted the code into **pubspec.yaml**, save the file within VS Code. With the file saved, we need to update the required dependencies. We can do this by clicking on the button highlighted in Figure 2-b, or by running the **`pub get`** command within the built-in VS Code terminal.

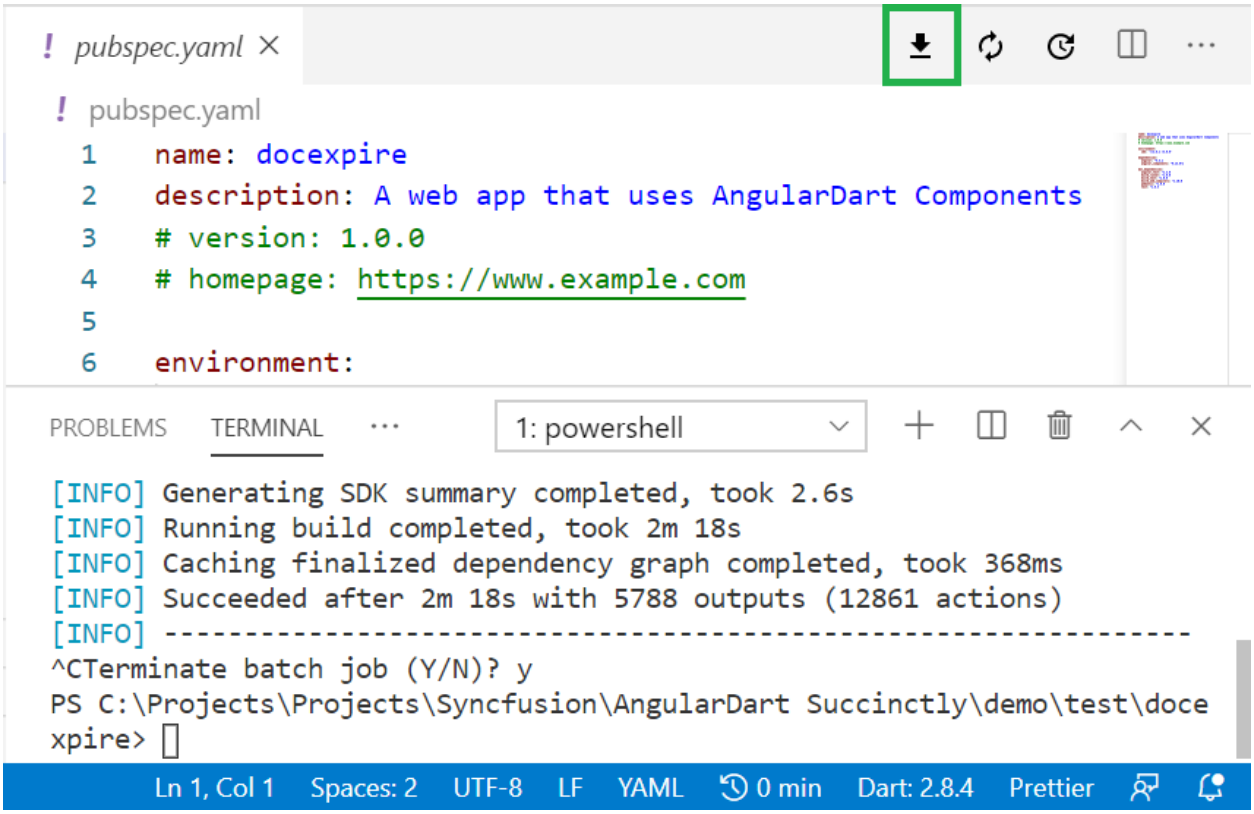

Figure 2-b: The Dependencies Update Button (pubspec.yaml – VS Code)

With the Dart packages updated and ready, let's make the necessary adjustments to our project files and folder structure.

Project structure

The project that was scaffolded is well organized and has the base structure that the application we will be building requires. Nevertheless, there are some changes we need to make. First, let's have a look at the out-of-the-box scaffolded project structure.

The project structure has three main parts. The first part is the **root** folder, which is where the **pubspec.yaml** file resides. The second part is the **web** folder, where the **index.html**, **main.dart**, and **styles.css** files reside. We'll need to modify the **index.html** file.

The third part is the **lib** folder, which contains the AngularDart components our application will use. This is where we will do most of our work. Within the **lib** folder, we have the application's main component files, these are **app_component.css**, **app_component.html**, and **app_component.dart**.

The **lib** folder also contains an **src** subfolder, which we will use for adding additional components that our application will use. The **src** folder is where, along with the **lib** folder, we will spend most of our time.

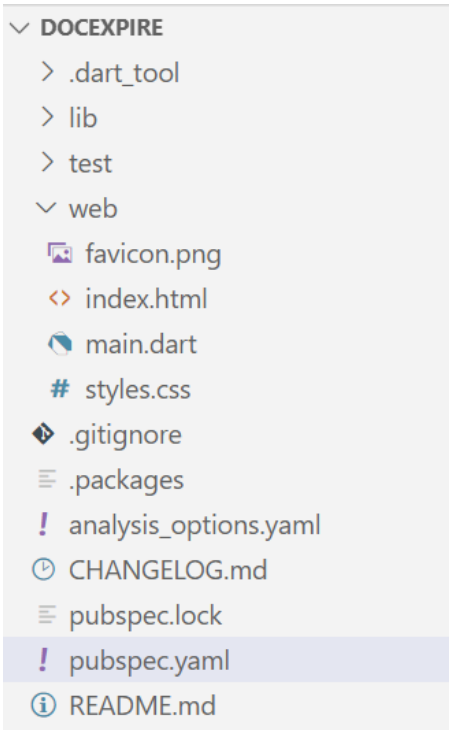

Figure 2-c: The Scaffolded Project Structure

With our project structure defined and understood, we can now make the necessary adjustments to the scaffolded application.

Updating index.html

The first major change we need to do is to update the **index.html** file. We are going to include references to Font Awesome, Material Design for Bootstrap, and Firebase.

The following listing contains the changes required for **index.html**. Let's have a look.

Code Listing 2-b: Updated index.html File

```
<!DOCTYPE html>
<html>
  <head>
    <title>DocExpire</title>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet"
     href="https://use.fontawesome.com/releases/v5.11.2/css/all.css">
    <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Ro
boto:300,400,500,700&display=swap">
```

```
    <link href="https://cdnjs.cloudflare.com/ajax/libs/twitter-
bootstrap/4.5.0/css/bootstrap.min.css" rel="stylesheet">
    <link href="https://cdnjs.cloudflare.com/ajax/libs/mdbootstrap/4.19.0/c
ss/mdb.min.css" rel="stylesheet">
    <link rel="stylesheet" href="styles.css">
    <link rel="icon" type="image/png" href="favicon.png">
    <script src="https://www.gstatic.com/firebasejs/7.13.1/firebase.js">
    </script>
    <script defer src="main.dart.js"></script>
    <body>
      <my-app>Loading...</my-app>
    </body>
  </head>
</html>
```

As you have seen, it's nothing too overwhelming—just references to the external libraries previously mentioned, and to **main.dart.js**, which will contain the app's Dart code compiled to JavaScript.

Notice that within the **body** tag, there is a reference to the **my-app** tag, which corresponds to the app's main component, **app_component**. The **my-app** tag is where the app's remaining HTML markup that renders the user interface will be injected.

There are no other changes required in the app's **web** folder.

Updating app_component

The next set of changes we need to do is going to be within the app's **lib** folder. We'll start by making adjustments to all **app_component**-related files.

So, open the **app_component.html** file and replace the existing markup with the one contained within the listing that follows.

Code Listing 2-c: Updated app_component.html File

```
<de-base></de-base>
```

DocExpire Base (**de-base**) is a custom component that we yet have to create, and it will be the base component for the application's user interface. We will later create the corresponding files for the **de-base** component.

The next thing we need to do is update the **app_component.dart** file. Let's do that by replacing the existing code with the following.

Code Listing 2-d: Updated app_component.dart File

```
import 'package:angular/angular.dart';
import 'src/de_base/de_base_component.dart';

@Component(
  selector: 'my-app',
  templateUrl: 'app_component.html',
  directives: [DocExpireBase],
  styleUrls: ['app_component.css']
)
class AppComponent {
}
```

We start by referencing the AngularDart package (**`package:angular/angular.dart`**) and our user interface base component (**`de_base_component.dart`**), which we yet have to create.

We then define the **`AppComponent`** class and annotate it using the **`@Component`** attribute. By using this annotation on the **`AppComponent`** class, we are indicating that the **`AppComponent`** class is no longer a regular Dart class, but instead an AngularDart component class.

The **`selector`** annotation property within the class annotation specifies the name of the tag that will be used within the markup (**`my-app`**), which is referenced within the **`body`** section of **index.html**.

The **`templateUrl`** annotation property indicates the name of the file that will contain the markup for **app_component**—which, as we previously saw, is found within **app_component.html**.

The **`directives`** annotation property is an array that is passed to the **`AppComponent`** class, which indicates the child components that are used and referenced within the code or markup of **app_component.dart** or **app_component.html**, respectively.

The **`styleUrls`** annotation property indicates the name of the stylesheet (CSS file) that will be referenced and used by **app_component.html**, which in this case is **app_component.css**.

With regards to the content of the **app_component.css** file, open the file and remove the predefined CSS styling contained within it.

Updating the lib and src folders

Go to the **src** folder (which resides within the **lib** folder) and delete any subfolders and files contained within, as shown in Figure 2-d.

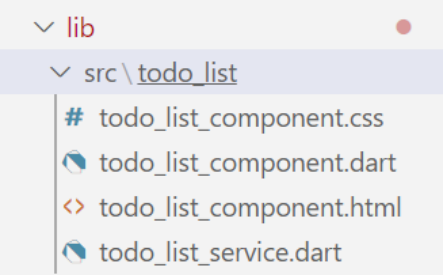

Figure 2-d: The lib\src Folder Content

With the **src** folder empty, create the following subfolders within it: **de_base**, **de_form**, **model** and **services**. The **de_base** and **de_form** folders will contain the two AngularDart subcomponents that our application will use.

The **de_base** component will render the app's main screen, which includes the list of documents that will expire. The following figure shows how the **de_base** component will look when the app is finished.

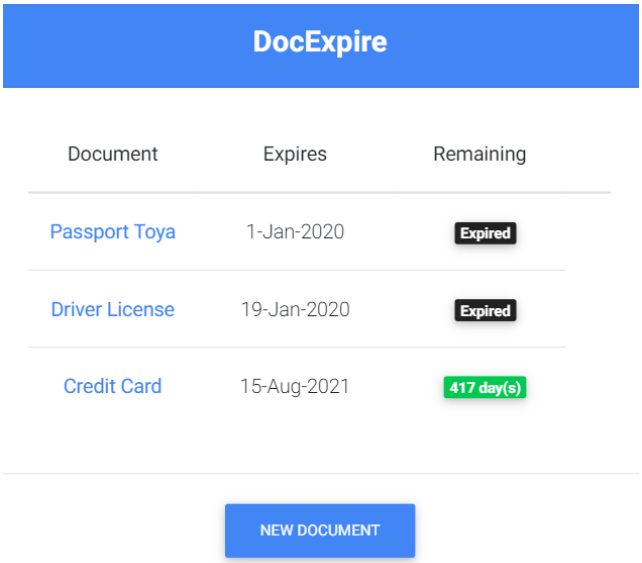

Figure 2-e: The de_base Component (Finalized App)

Please note that for the **de_base** component, the list of documents seen is variable and not fixed to three items, as shown on the figure. On the other hand, the **de_form** component will display the properties of each document. The following figure shows how the **de_form** component will look when the app is finished.

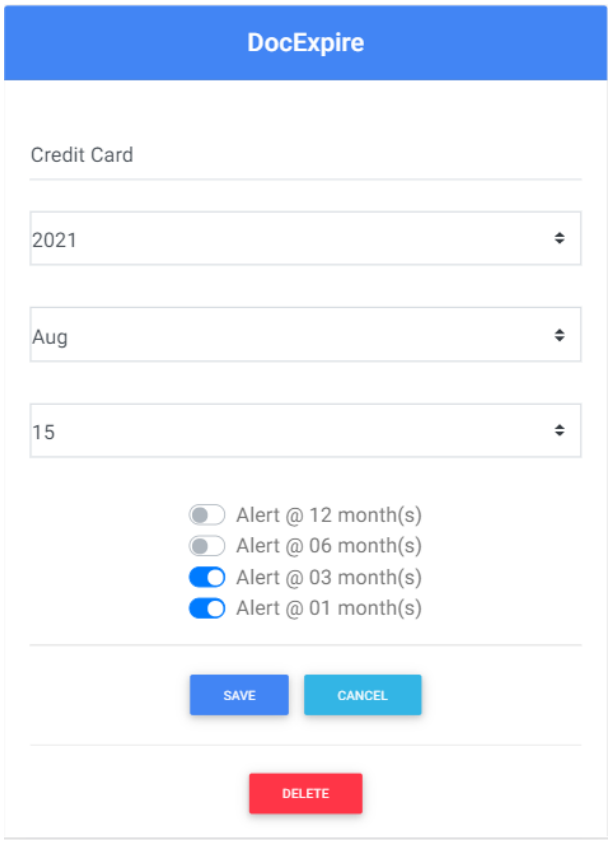

Figure 2-f: The de_form Component (Finalized App)

The **model** folder will contain the application's data model, which will be used for hosting the app's classes that will be responsible for manipulating document data. We'll use Firebase as our backend.

The **services** folder will contain the application's classes that will interact with Firebase, using the app's data model.

The de_base folder

It's now time to focus our attention on the **de_base** folder. Within the **de_base** folder we just created, let's create these files: **de_base_component.dart** and **de_base_component.html**.

For now, we are just going to create the files and leave them empty. Once you have created these files, you should have the following **de_base** folder structure.

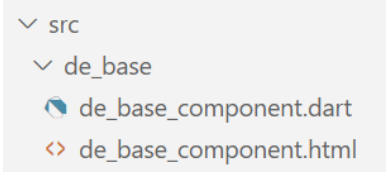

Figure 2-g: The de_base Folder Structure

Now, let's do the same for the **de_form** component.

The de_form folder

Within the **de_form** folder we recently created, let's create these files: **de_form_component.dart** and **de_form_component.html**. For now, we are just going to create the files and leave them empty. Once you have created these files, you should have the following **de_form** folder structure.

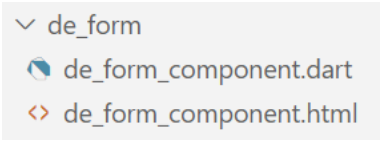

Figure 2-h: The de_form Folder Structure

The model folder

Within the **model** folder, let's create a file called **docs.dart**, which will contain the classes that will be used to work and manipulate the app's data model. For now, just create the file and leave it empty. Once you have created the file, you should have the following **model** folder structure.

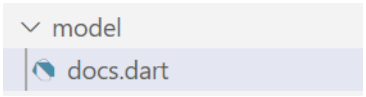

Figure 2-i: The model Folder Structure

The services folder

Within the **services** folder, let's create a file called **docs_service.dart**, which will contain the classes that will be used to interact with Firebase, using the app's data model. For now, just create the file and leave it empty. Once you have created the file, you should have the following **services** folder structure.

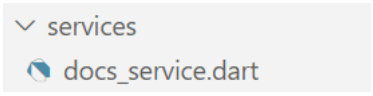

Figure 2-j: The services Folder Structure

Summary

We've now modified the app's original scaffolded project folder and file structure, and we have adapted it to our app's requirements and functionality. This is a great starting point that cements the foundations for what will become our application. In the next chapters, we will focus on writing the app's logic—which is going to be awesome!

Chapter 3 Base UI Component

Overview

Throughout this chapter, we will focus on how to build the base component, which will host our application's main screen. By building this component, we will not only see how to create the main UI, but also dive deep into AngularDart. Let's dive right in!

Component structure

My approach when developing applications is to first create the UI and then add all the necessary logic to make it work. This is exactly how we are going to proceed.

Based on my experience, this approach can save valuable development time, as it allows us to define how the application will look and feel, driving how the logic around it will behave.

But before we dive into the HTML markup for the base component, let's have a look at the following diagrams, which illustrate how the finished UI relates to the markup we will shortly explore.

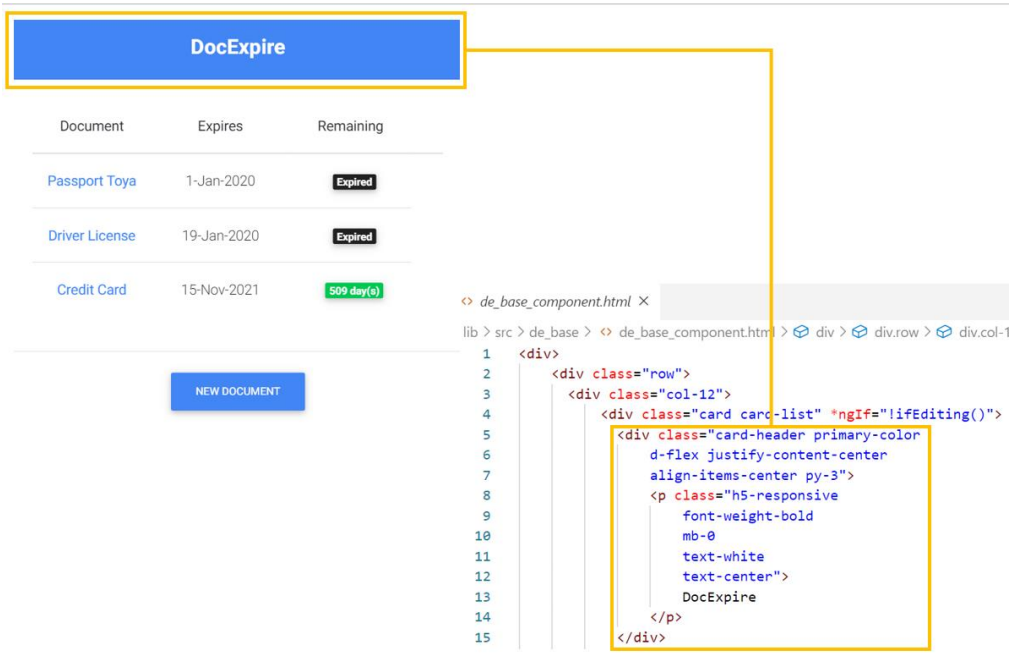

Figure 3-a: App Title—Base Component

We can see in Figure 3-a that the **`div`** element, highlighted in yellow, corresponds to the header of the app's title. Let's keep exploring the finished UI.

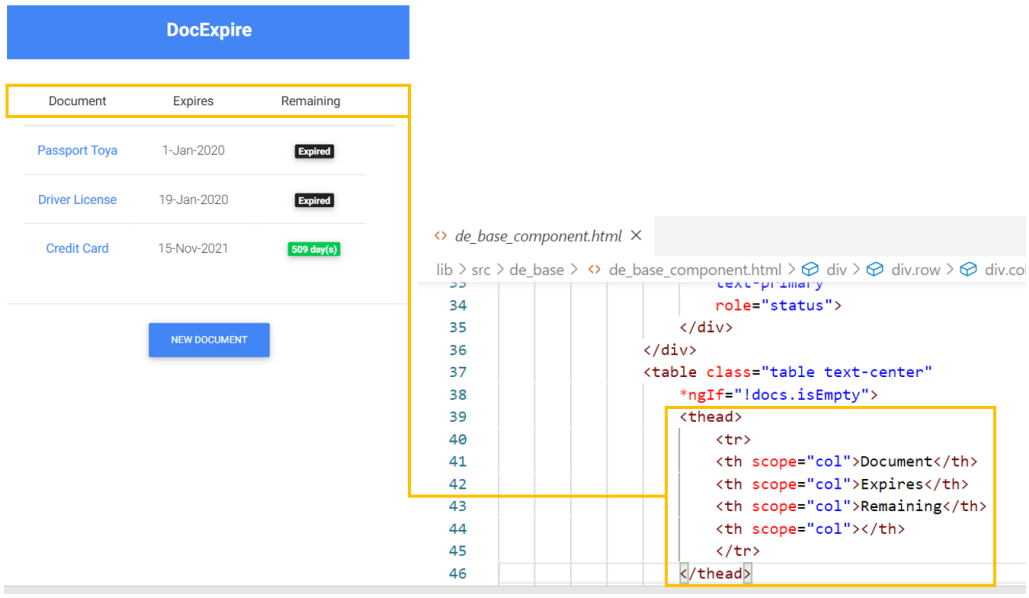

Figure 3-b: Table Header—Base Component

In Figure 3-b, we can see how the table header element **`thead`** is displayed within the finished UI. Let's continue to explore.

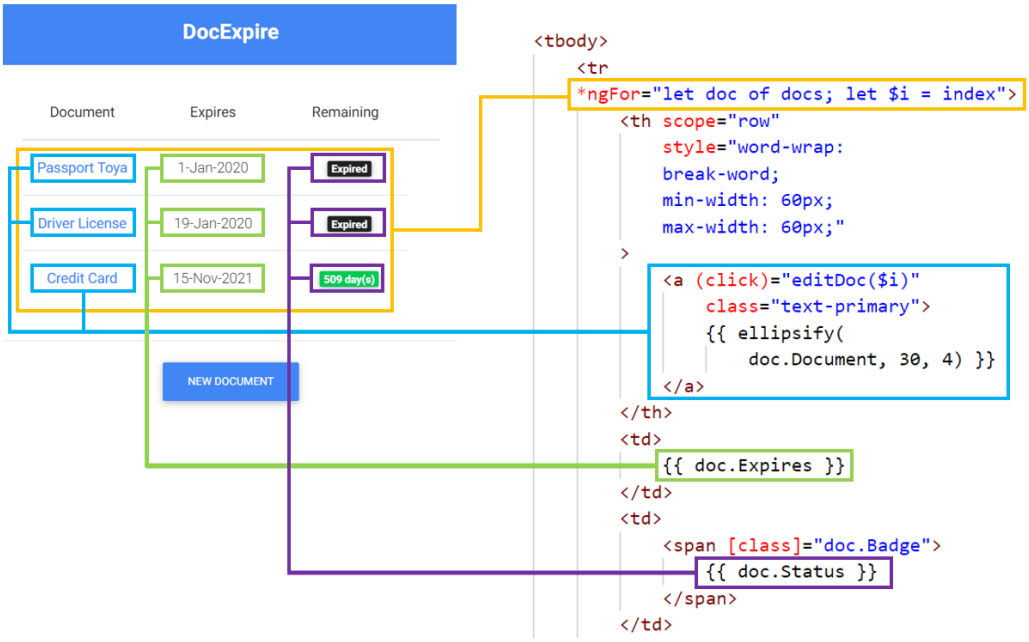

Figure 3-c: Table Items—Base Component

Figure 3-c is slightly more challenging to understand. We can see four distinctive areas, each highlighted with a different color.

The area highlighted in yellow corresponds to a dynamic list of documents, which in AngularDart is achieved by looping through the **`docs`** list using the ***`ngFor`** directive.

Each table row corresponds to a **doc** object, which has three distinctive properties: **doc.Document**, **doc.Expires**, and **doc.Status**.

The **doc.Document** property, highlighted in light blue, indicates the name of the document. By clicking on it, the user will be able to open and view details and edit or delete a document.

The **doc.Expires** property, highlighted in light green, represents the document's expiration date.

The **doc.Status** property, highlighted in purple, indicates how many days remaining the document has before it expires.

Let's continue to explore the remaining parts of the base component's UI.

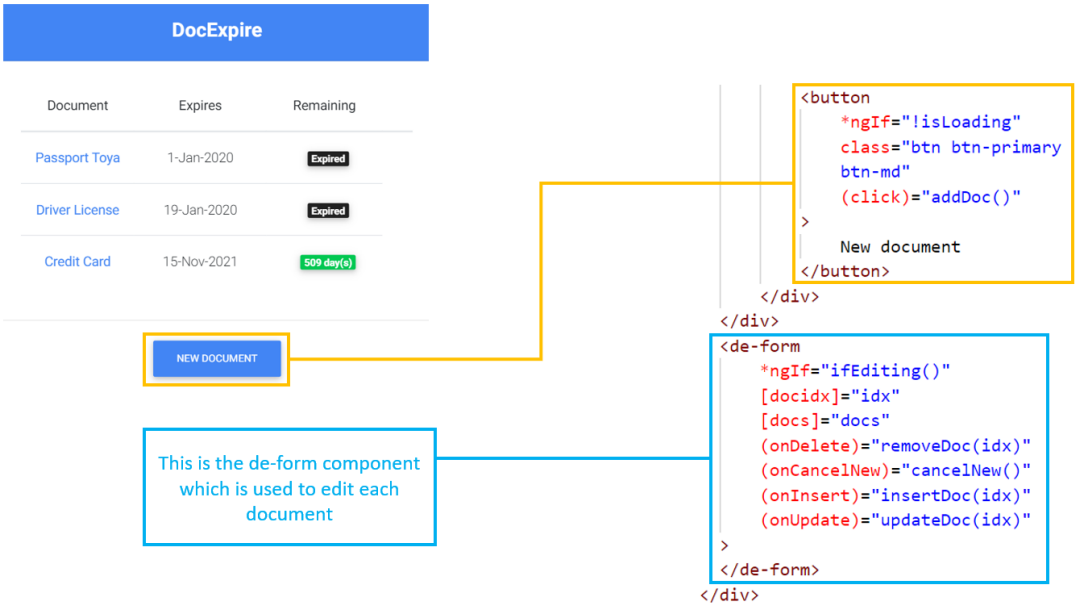

Figure 3-d: Footer—Base Component

The last part of the base component's UI is the footer section, which contains the button that can be used to create a new document, and is highlighted in yellow in Figure 3-d.

It also includes the **de-form** component markup, which is highlighted in light blue. The **de-form** component is going to be used for editing any of the documents in the list or adding a new one. The **de-form** component is not visible by default and will only be shown when the **ifEditing** function returns **true**. We will check what this function does later.

Notice as well that two values are passed to the **de-form** component, as well as four events that are triggered. We'll explore these later.

Component markup

Now that we have explored how the **de-base** component is structured and how the UI elements relate to the markup, let's have a look at the full HTML markup of the component.

Within VS Code, open the **de_base_component.html** file and add the following code from Code Listing 3-a.

This markup uses regular Bootstrap classes, which have a Material Design look and feel. As you will see, it's very easy to read, even though you might not have had any exposure to Boostrap before.

Code Listing 3-a: de_base_component.html (Full Code)

```
<div>
    <div class="row">
      <div class="col-12">
          <div class="card card-list" *ngIf="!ifEditing()">
            <div class="card-header primary-color
                d-flex justify-content-center
                align-items-center py-3">
                <p class="h5-responsive
                    font-weight-bold
                    mb-0
                    text-white
                    text-center">
                    DocExpire
                </p>
            </div>
            <div class="card-body">
                <div class="text-center">
                    <h5 *ngIf="docs.isEmpty && !isLoading"
                        class="card-title">
                        No documents...
                    </h5>
                    <p *ngIf="docs.isEmpty"
                        class="card-text">
                        <template [ngIf]="!isLoading">
                            Feel free to add a new one :)
                        </template>
                        <template [ngIf]="isLoading">
                            Loading documents...
                        </template>
                    </p>
                    <div *ngIf="isLoading"
                        class="spinner-border
```

```
                    text-primary"
                    role="status">
                </div>
            </div>
            <table class="table text-center"
                *ngIf="!docs.isEmpty">
                <thead>
                    <tr>
                    <th scope="col">Document</th>
                    <th scope="col">Expires</th>
                    <th scope="col">Remaining</th>
                    <th scope="col"></th>
                    </tr>
                </thead>
                <tbody>
                    <tr
                    *ngFor="let doc of docs; let $i = index">
                        <th scope="row"
                            style="word-wrap:
                            break-word;
                            min-width: 60px;
                            max-width: 60px;"
                        >
                            <a (click)="editDoc($i)"
                                class="text-primary">
                                {{ ellipsify(
                                    doc.Document, 30, 4) }}
                            </a>
                        </th>
                        <td>
                            {{ doc.Expires }}
                        </td>
                        <td>
                            <span [class]="doc.Badge">
                                {{ doc.Status }}
                            </span>
                        </td>
                    </tr>
                </tbody>
            </table>
        </div>
        <div class="card-footer
            white py-3
            d-flex
```

```
                justify-content-center">
                <button
                    *ngIf="!isLoading"
                    class="btn btn-primary
                    btn-md"
                    (click)="addDoc()"
                >
                    New document
                </button>
            </div>
        </div>
        <de-form
            *ngIf="ifEditing()"
            [docidx]="idx"
            [docs]="docs"
            (onDelete)="removeDoc(idx)"
            (onCancelNew)="cancelNew()"
            (onInsert)="insertDoc(idx)"
            (onUpdate)="updateDoc(idx)"
        >
        </de-form>
      </div>
    </div>
</div>
```

As you might have noticed, the markup is annotated with some AngularDart directives and specific syntax.

The **`*ngIf`** directive is used within AngularDart to display (or not) a specific section of the markup, given a condition.

The **`template`** tag is used to avoid code repetition. Within it, the **`*ngIf`** directive is written as **`[ngIf]`** instead.

The **`*ngFor`** directive is used within AngularDart to loop through a list of objects—in this case, the list of documents—**`docs`**.

The double curly braces are used for displaying the value of AngularDart objects fields and properties by injecting them into the markup's Document Object Model (DOM). So **`{{ doc.Expires }}`** shows the document's expiry date, and **`{{ doc.Status }}`** shows the document's status (the number of days remaining before expiration).

On the other hand, the **`{{ ellipsify(doc.Document, 30, 4) }}`** instruction passes the name of the document (**`doc.Document`**) to the **`ellipsify`** method, which shortens the document's description by adding an ellipsis (…) if the document's description is too long.

Events in AngularDart are annotated with parentheses. This means that **(click)**, **(onDelete)**, **(onCancelNew)**, **(onInsert)**, and **(onUpdate)** all represent events that get triggered.

Values get passed as properties (using two-way binding) to components by using square brackets. Therefore, **[docidx]** and **[docs]** are values that get passed to the **de-form** component.

Two-way data binding means that if those values are changed within the **de-form** component, those changes will get passed back to the parent component—**de-base**.

The rest of the markup syntax is regular HTML decorated with Material Design for Bootstrap CSS classes, which give the UI a nice and responsive look and feel. From this code, there were some functionalities we didn't mention when we were reviewing the component's structure.

First, we have **docs.isEmpty**, which indicates that the document list is empty (that there are no documents to expire) if its value is **true**. When **docs.isEmpty** is **false**, this indicates that the list of documents to expire is not empty.

If the value of **isLoading** is **true**, it indicates if the application is busy loading the list of documents from Firebase. If the document data has already been fetched from Firebase, then the value of **isLoading** will be **false**.

Therefore, based on the values of **docs.isEmpty** and **isLoading**, two UI sections can be shown or hidden, depending on the values of these properties, which are highlighted in bold in Code Listing 3-e.

When document data is being retrieved from Firebase—independently of whether or not there are documents—the following markup is shown.

```
<p *ngIf="docs.isEmpty"
  class="card-text">
  <template [ngIf]="isLoading">
    Loading documents...
  </template>
</p>
```

On the other hand, when document data has been retrieved from Firebase and there are no documents to display, the following markup is shown.

```
<h5 *ngIf="docs.isEmpty && !isLoading"
  class="card-title">
  No documents...
</h5>

<p *ngIf="docs.isEmpty"
  class="card-text">
  <template [ngIf]="!isLoading">
    Feel free to add a new one :)
  </template>
</p>
```

That's all the HTML markup required for the base component. Let's now explore the component's logic.

Component logic

To bring to life our base component UI, we need to add some logic to it. We'll add this logic to the **de_base_component.dart** file. Within VS Code, open the **de_base_component.dart** file and add the following code.

Code Listing 3-b: de_base_component.dart (Full Code)

```
import 'dart:async';
import 'package:angular/angular.dart';

import '../de_form/de_form_component.dart';
import '../model/docs.dart';
import '../services/docs_service.dart';

@Component(
  selector: 'de-base',
  templateUrl: 'de_base_component.html',
  directives: [coreDirectives, DocForm],
  providers: [ClassProvider(DocsService)]
)
class DocExpireBase implements OnInit {
  int idx;
  Doc cDoc;
  bool isLoading = true;

  List<Doc> docs = [];

  final DocsService _docsService;
  DocExpireBase(this._docsService);

  void sort() async {
    await Future.delayed(
      const Duration(seconds : 1));
    Doc.sortDocs(docs);
  }

  @override
  Future<Null> ngOnInit() async {
    docs = await _docsService.getDocs();
    isLoading = false;
```

```
  sort();
}

bool ifEditing() {
  var r = false;
  if (docs != null) {
    for (var i = 0; i < docs.length; i++) {
      if (docs[i].Edit) {
        r = true;
        break;
      }
    }
  }
  return r;
}

void addDoc() {
  docs.add(Doc(
    ID: '-1',
    Document: '',
    Year: '',
    Month: '',
    Day: '',
    Expires: '',
    Status: 'No rush',
    Badge: 'badge badge-success',
    Edit: true,
    Alert12: false,
    Alert6: false,
    Alert3: false,
    Alert1: false
  ));

  idx = docs.length - 1;
}

void editDoc(int index) {
  idx = index;
  cDoc = Doc(
    ID: docs[index].ID,
    Document: docs[index].Document,
    Year: docs[index].Year,
    Month: docs[index].Month,
```

```
      Day: docs[index].Day,
      Expires: docs[index].Expires,
      Status: docs[index].Status,
      Badge: docs[index].Badge,
      Edit: docs[index].Edit,
      Alert12: docs[index].Alert12,
      Alert6: docs[index].Alert6,
      Alert3: docs[index].Alert3,
      Alert1: docs[index].Alert1
    );
    docs[index].Edit = true;
  }

  void removeDoc(int index) async {
    // Firebase
    await _docsService.removeDoc(docs[index]);

    docs.removeAt(index);
    sort();
  }

  void insertDoc(int index) async {
    // Firebase
    var fbId = await
      _docsService.addDoc(docs[index]);

    docs[index].ID = fbId;
    await _docsService.updateDoc(docs[index]);

    sort();
  }

  void updateDoc(int index) async {
    // Firebase
    await _docsService.updateDoc(docs[index]);

    sort();
  }

  void cancelNew() {
    if (docs != null) {
      if (docs[idx].ID != '-1') {
        docs[idx] = cDoc;
      }
```

```
      for (var i = 0; i < docs.length; i++) {
        if (docs[i].ID == '-1') {
          docs.removeAt(i);
          break;
        }
      }
    }
  }

  String ellipsify(String t, int max, int before) {
    var r = '';

    if (t.length >= max) {
      for (var i = 0; i < t.length; i++) {
        if (i < max - before) {
          r += t[i];
        }
        else {
          r += '...';
          break;
        }
      }
    }
    else {
      r = t;
    }

    return r;
  }
}
```

To understand what this code does, let's break it into smaller pieces and analyze each one individually.

Component initialization

First, we have the **`import`** statements, where we reference the modules that our component will be using. The following two **`import`** statements make use of Dart's asynchronous library and the AngularDart core library, respectively.

```
import 'dart:async';
import 'package:angular/angular.dart';
```

Then, we import the **de-form** component, the application's data model (**docs.dart**), and the service responsible for interacting with Firebase (**docs_service.dart**).

```
import '../de_form/de_form_component.dart';
import '../model/docs.dart';

import '../services/docs_service.dart';
```

Then we have the **DocExpireBase** class **@Component** annotation, which makes the class an AngularDart component, rather than a regular Dart class.

```
@Component(
  selector: 'de-base',
  templateUrl: 'de_base_component.html',
  directives: [coreDirectives, DocForm],
  providers: [ClassProvider(DocsService)]
)
```

The **selector** property of the annotation tells AngularDart that the **DocExpireBase** class will be described within the markup as the **de-base** component.

The **templateUrl** property tells AngularDart that the HTML markup for the **de-base** component is described within the **de_base_component.html** file.

The **directives** property indicates that the **DocExpireBase** class will use the AngularDart **coreDirectives**, which include ***ngIf** and ***ngFor**. It also indicates that the **DocExpireBase** class will also use the **DocForm** class—which, as we will see later, corresponds to the **de-form** component.

The **providers** property indicates that the **DocsService** class will be used as the service **ClassProvider** of the **DocExpireBase** class. A service **ClassProvider** in AngularDart is mostly responsible for making the connection to the backend database, fetching data, and performing inserts and updates on the database.

Next, we have the declaration of the **DocExpireBase** class. It implements the **ngOnInit** method, which will be used to perform a few initializations, as we'll see shortly.

```
class DocExpireBase implements OnInit
```

Within the **DocExpireBase** class itself, we find a few variable declarations and initializations—let's explore them.

```
int idx;
Doc cDoc;
bool isLoading = true;
```

First, we have the **idx** variable, which we will use to know the document number (the index within the list) when a document is edited. The value of **idx** will be passed to the **de-form** component.

Next, we find **cDoc**, an instance of the **Doc** class that is contained within the app's data model and represents a document object. The variable **cDoc** stands for the current document.

The **isLoading** Boolean variable, when set to **true**, will be used to indicate if the app is busy fetching documents from Firebase.

A very important variable is **docs**, which is a **List** of **Doc** objects that will contain all the documents retrieved from Firebase by the **ClassProvider** service. This list is declared as follows:

```
List<Doc> docs = [];
```

The **_docsService** object is an instance of the **ClassProvider** service (**DocsService**), which is used for interacting with Firebase. This is declared with the **final** keyword, meaning that it won't change.

```
final DocsService _docsService;
```

Finally, we have the **DocExpireBase** class constructor, to which we inject the **ClassProvider** service instance (**_docsService**).

```
DocExpireBase(this._docsService);
```

With all the variables and initialization out of the way, we can focus on the class methods and functionality.

Component methods

Let's start by exploring the **sort** method. As its name implies, this method is used to sort the list of documents by the number of days remaining until expiration.

Code Listing 3-c: de_base_component.dart (The sort Method)

```
void sort() async {
  await Future.delayed(
    const Duration(seconds : 1));
  Doc.sortDocs(docs);
}
```

The documents expiring soon will appear at the top of the list, whereas the documents expiring later will appear at the bottom of the list. Essentially, documents are sorted in ascending order based on the number of days remaining until expiration.

Figure 3-e: Document List—Ordered by Days Remaining

The **sort** method can perform document sorting by invoking the **sortDocs** method from the **Doc** instance, which belongs to the data model.

To give enough time for the documents to be retrieved successfully from Firebase by the **ClassProvider** service instance (**_docsService**), we pause the execution of the code for one second before invoking **sortDocs**. This is achieved by executing the following instruction:

```
await Future.delayed(const Duration(seconds : 1));
```

Next, we have the **ngOnInit** method, which the **DocExpireBase** class explicitly implements. This method, which returns a potential value or promise (known in Dart as a **Future**), is used for retrieving the list of documents from Firebase.

Code Listing 3-d: de_base_component.dart (The ngOnInit Method)

```
@override
Future<Null> ngOnInit() async {
  docs = await _docsService.getDocs();
  isLoading = false;

  sort();
}
```

The documents are fetched from Firebase using the **ClassProvider** service instance (**_docsService**) by invoking its **getDocs** method. As this is a Dart asynchronous operation, the **await** keyword is used; therefore, the **ngOnInit** method is marked as **async**.

Once the documents have been retrieved, the **isLoading** variable is set to **false**, and after, the **sort** method is invoked.

Adding and editing

Next, we have the **ifEditing** method, which determines whether one of the documents from the list is being edited.

Code Listing 3-e: de_base_component.dart (The ifEditing Method)

```
bool ifEditing() {
  var r = false;
  if (docs != null) {
    for (var i = 0; i < docs.length; i++) {
      if (docs[i].Edit) {
        r = true;
        break;
      }
    }
  }
  return r;
}
```

To determine if one of the documents is being edited, the method loops through each of the documents and checks the value of the **Edit** property of each one. If one of the documents has its **Edit** property set to **true**, it would indicate that the document is being edited. In that case, the method breaks the loop and return **true**; otherwise, **false** is returned.

The **ifEditing** method is key to determine if the **de-form** component needs to be displayed or hidden.

Next, let's have a look at the **addDoc** method, which will be responsible for adding a new and empty **Doc** instance to the list of documents (**docs**).

Code Listing 3-f: de_base_component.dart (The addDoc Method)

```
void addDoc() {
  docs.add(Doc(
    ID: '-1',
    Document: '',
    Year: '',
    Month: '',
    Day: '',
    Expires: '',
    Status: 'No rush',
    Badge: 'badge badge-success',
    Edit: true,
    Alert12: false,
    Alert6: false,
```

```
    Alert3: false,
    Alert1: false
  ));

  idx = docs.length - 1;
}
```

As you can see, the new **Doc** instance is initialized with default values, and its **ID** property is set to **-1**. Only when the **Doc** instance is saved—which occurs within the **de-form** component—does the **ID** property value get updated, as well as the rest of the **Doc** instance properties.

The value of **idx** is updated to reflect the new **Doc** instance added to the **docs** list.

When we need to edit a specific document (instead of creating one) we need to create a **Doc** instance by assigning to its properties the values of the current document. Let's have a look at how this is done.

Code Listing 3-g: de_base_component.dart (The editDoc Method)

```
void editDoc(int index) {
  idx = index;
  cDoc = Doc(
    ID: docs[index].ID,
    Document: docs[index].Document,
    Year: docs[index].Year,
    Month: docs[index].Month,
    Day: docs[index].Day,
    Expires: docs[index].Expires,
    Status: docs[index].Status,
    Badge: docs[index].Badge,
    Edit: docs[index].Edit,
    Alert12: docs[index].Alert12,
    Alert6: docs[index].Alert6,
    Alert3: docs[index].Alert3,
    Alert1: docs[index].Alert1
  );
  docs[index].Edit = true;
}
```

The first thing this method does is assign the value of the current document **index**, to the **idx** variable, which is internally responsible for keeping track of the document that is being edited.

Next, to the **cDoc** object, we assign a **Doc** instance—which has been assigned the values of the selected document, from the **docs** list—**docs[index]**. The selected document is the one that is clicked using the UI. This is achieved by using the following markup.

```
<a (click)="editDoc($i)"
  class="text-primary">
  {{ ellipsify(doc.Document, 30, 4) }}
</a>
```

So, **$i** represents the index of the document from the list that is clicked. The value of **$i** is derived from the ***ngFor** directive's condition, as follows.

```
*ngFor="let doc of docs; let $i = index"
```

Therefore, the value of **$i** is passed as a parameter to the **editDoc** method, which becomes the method's **index** parameter, which then gets assigned to the variable **idx**.

Finally, for the editing process to work, and the **de-form** component to be displayed, the **Edit** property of the selected document is set to **true**.

```
docs[index].Edit = true;
```

Remove, cancel, insert, and update

Now that we know how to add a new document and assign the instance properties when editing a document, let's explore how we can remove, cancel, insert, and update a document.

If you recall, within the **de-base** component's markup, the **de-form** component is referenced as follows.

```
<de-form
  *ngIf="ifEditing()"
   [docidx]="idx"
   [docs]="docs"
   (onDelete)="removeDoc(idx)"
   (onCancelNew)="cancelNew()"
   (onInsert)="insertDoc(idx)"
   (onUpdate)="updateDoc(idx)"
>
</de-form>
```

The **de-form** component triggers the execution of four events: **onDelete**, **onCancelNew**, **onInsert**, and **onUpdate**.

As their names imply, these events execute the actions when a document is deleted, when adding or editing a new document, when the operation is canceled, when a document is inserted, and when a document is updated, respectively.

Each event executes a **DocExpireBase** method when triggered. The **onDelete** event executes the **removeDoc** method, the **onCancelNew** event executes the **cancelNew** method, the **onInsert** event executes the **insertDoc** method, and the **onUpdate** event executes the **updateDoc** method.

Let's start by looking at the **removeDoc** method to understand what it does.

Code Listing 3-h: de_base_component.dart (The removeDoc Method)

```
void removeDoc(int index) async {
  // Firebase
  await _docsService.removeDoc(docs[index]);

  docs.removeAt(index);
  sort();
}
```

The **removeDoc** method invokes the method with the same name from the **ClassProvider** service instance (**_docsService**), which deletes the document from Firebase. We'll explore those details later.

All Firebase operations are asynchronous; this is why the **await** keyword is used when calling the **removeDoc** method from the service instance. Because the **await** keyword is used, the method is marked as **async**.

After the document has been removed from Firebase, the document itself is removed from the list of documents. This is what **docs.removeAt(index)** does.

Once the document has been removed from the list of documents, we call the **sort** method, so all the existing documents can be sorted by the number of days remaining before expiration.

Next, we have the **insertDoc** method, which is responsible for adding a new document into Firebase.

Code Listing 3-i: de_base_component.dart (The insertDoc Method)

```
void insertDoc(int index) async {
  // Firebase
  var fbId = await
    _docsService.addDoc(docs[index]);

  docs[index].ID = fbId;
  await _docsService.updateDoc(docs[index]);

  sort();
}
```

The **insertDoc** method is used for adding a document to Firebase. This is done by invoking the **addDoc** method from the **ClassProvider** service instance (**_docsService**), to which the document being added (**docs[index]**) is passed.

As previously mentioned, all Firebase operations are asynchronous. This is why the **await** keyword is used when calling the **addDoc** method from the service instance, and the method is marked as **async**.

Once the document has been added to Firebase, the document's Firebase ID (**fbId**) is returned and assigned to the **ID** property of the document within the **docs** list. This is what **docs[index].ID = fbId** does.

Then, that same value is updated to the document's **ID** property stored in Firebase—which is what **_docsService.updateDoc(docs[index])** does.

Once the document has been added from the list of documents, we call the **sort** method.

Next, we have the **updateDoc** method, which is responsible for updating a document that has been edited to Firebase.

Code Listing 3-j: de_base_component.dart (The updateDoc Method)

```
void updateDoc(int index) async {
  // Firebase
  await _docsService.updateDoc(docs[index]);

  sort();
}
```

The **updateDoc** method is very simple. All it does is to invoke the method with the same name from the **ClassProvider** service instance (**_docsService**).

Once the document has been updated in Firebase, then the **sort** method is invoked. Here, the **await** keyword is used when calling the **updateDoc** method from the service instance, and the method is marked as **async**.

Now we know how the **de-base** component can remove, insert, and update documents. Let's have a look at how we can perform a cancelation operation on a document being inserted or edited.

Code Listing 3-k: de_base_component.dart (The cancelNew Method)

```
void cancelNew() {
  if (docs != null) {
    if (docs[idx].ID != '-1') {
      docs[idx] = cDoc;
    }
    for (var i = 0; i < docs.length; i++) {
```

```
      if (docs[i].ID == '-1') {
        docs.removeAt(i);
        break;
      }
    }
  }
}
```

A cancelation operation can be performed when the list of documents (**`docs`**) is not null.

If the **`ID`** of the document being canceled is different than **`-1`**, it means the document already exists within the document list, so its reference is updated on the list of documents. This is what **`docs[idx] = cDoc`** does.

For a new (unsaved) document that has been added to the document list, and whose operation has been canceled (its **`ID`** still has a value of **`-1`**), the cancelation process occurs, and the unsaved document is removed from the list. This is what **`docs.removeAt(i)`** does.

Awesome—now we know how the cancelation of an unsaved (new or existing) document works.

Adding ellipsis

Given that a document description is a string, and some descriptions might be lengthy, there is a possibility that such descriptions might not fit on the screen. This is why adding an ellipsis is a nice workaround to display the document's description without breaking the UI.

Let's suppose we have a document with the following description.

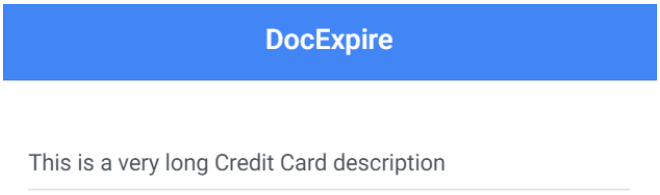

Figure 3-f: A Document with a Lengthy Description

On a smaller screen, such as a mobile device, this description could break the UI. Let's have a look.

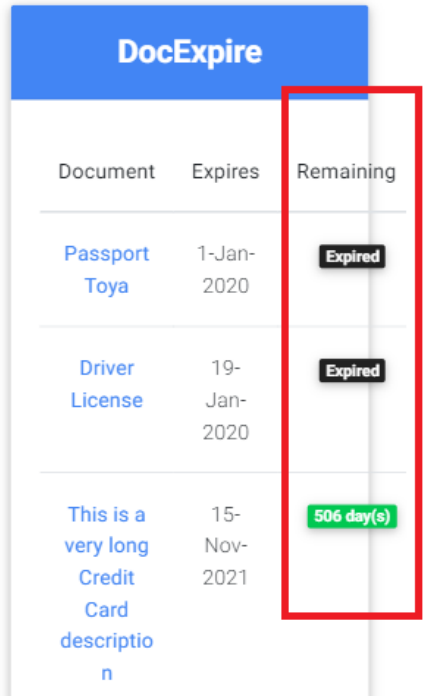

Figure 3-g: Broken UI

As you can see, the lengthy description overruns the app's UI. To prevent this, we can use the **`ellipsify`** method to shorten the description on the UI only by adding an ellipsis.

This is how the same screen would look when using the **`ellipsify`** method.

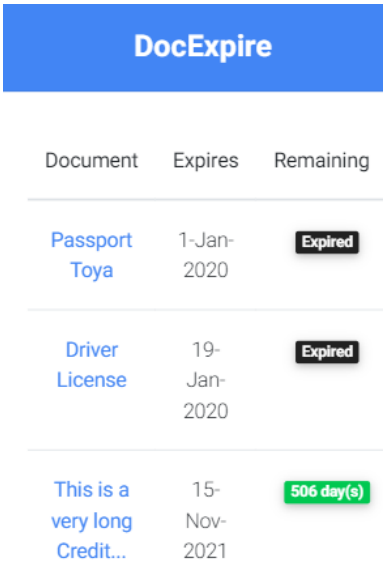

Figure 3-h: Using the ellipsify Method

As you can see, the description no longer overflows the screen and fits nicely within three lines on the main screen.

Notice that at the end of the description, an ellipsis has been added to avoid displaying the complete description. This is what the **ellipsify** method does. Let's have a look at how this achieved.

Code Listing 3-l: de_base_component.dart (The ellipsify Method)

```
String ellipsify(String t, int max, int before) {
  var r = '';

  if (t.length >= max) {
    for (var i = 0; i < t.length; i++) {
      if (i < max - before) {
        r += t[i];
      }
      else {
        r += '...';
        break;
      }
    }
  }
  else {
    r = t;
  }

  return r;
}
```

This method loops through the string and counts a number (**max**) of characters that it will return (which will be displayed on the screen).

An ellipsis is added when the character count reaches **max - before**. At that point, the method finishes looping through the characters of the string and returns the string with an ellipsis that will be displayed on the screen.

Besides the **ellipsify** method, the document's description adjustment to the UI (screen resolution) is enhanced by using the CSS classes and attributes (highlighted in bold in the following code) within the **de-base** markup.

```
<th scope="row"
  style="word-wrap:
  break-word;
  min-width: 60px;
  max-width: 60px;"
>
```

Summary

We now have ready the app’s main component, which is responsible for rendering the application's main screen, and the logic that governs its behavior.

We’ve seen that using some HTML with Material Design for Bootstrap, along with a few AngularDart directives and Dart logic, wasn’t as intimidating as originally thought—as combining all these technologies seemed quite a task.

In the next chapter, we’ll explore in depth the markup and logic behind the **`de-form`** component, which is used for editing or deleting existing documents, and adding new ones.

Chapter 4 Form UI Component

Overview

At this stage, we have the app's main screen, which displays the list of documents, ready to go—which is awesome! Now we need to focus our attention on the **de-form** component, which will be used by the application for editing and deleting existing documents or adding new ones. This is what we will build throughout this chapter.

Component structure

Just like we did with **de-base**, we'll explore the structure of the **de-form** component, so we can understand the relationship between the markup and finished UI. Let's have a look at the following diagrams.

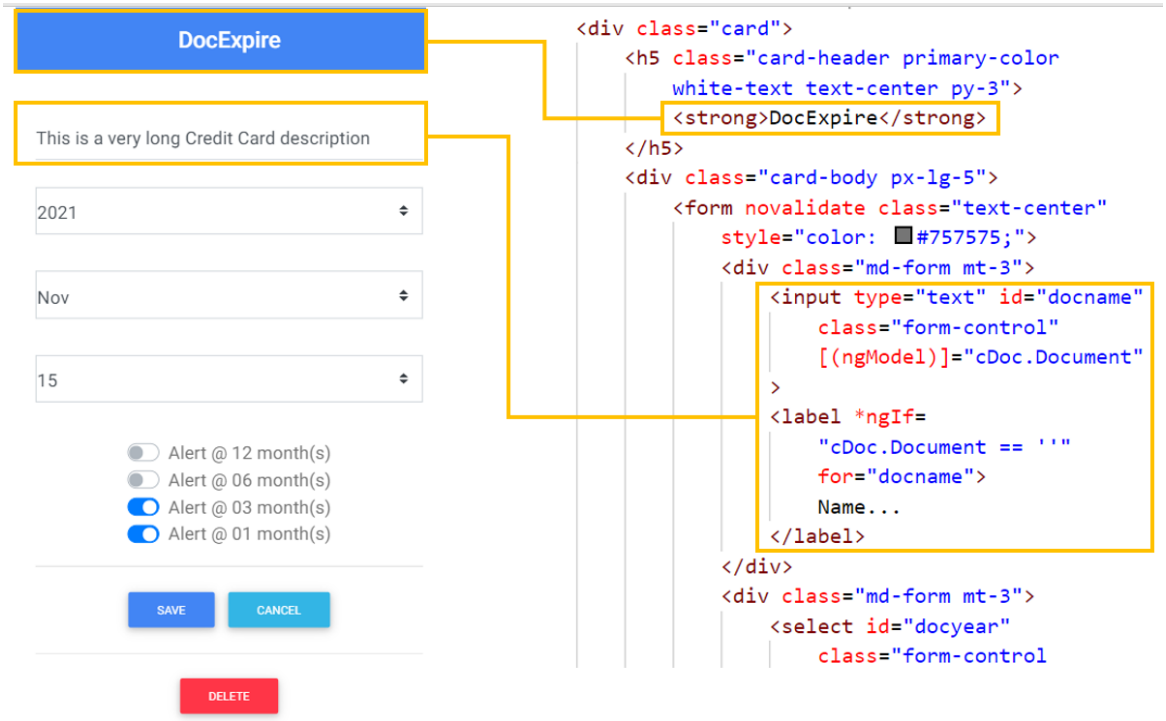

Figure 4-a: Header and Name—Form Component

We can see how the form's header is contained within a **strong** tag—so this relationship between the markup and UI is quite straightforward.

As for the document's description, the relationship between the markup and UI is slightly more complex.

We can see that the description relates to an **input** tag, which has a two-way binding with the **cDoc.Document** property. This is achieved using the **ngModel** directive.

If the document's description is empty, then the **label** with the value **Name…** will be shown on the **input** tag, which is achieved by using the ***ngIf** directive, when the value of **cDoc.Document** is empty.

Next, we have the **year** and **month** fields. Let's have a look at the following diagram to see how the markup of both relates to the UI.

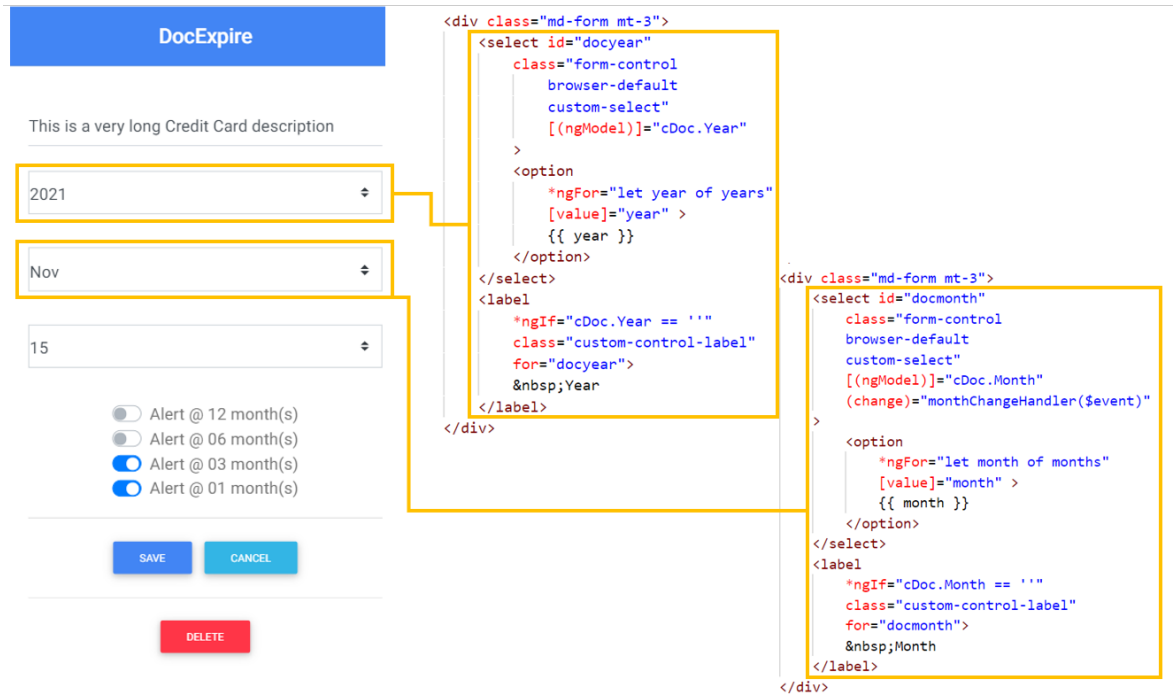

Figure 4-b: Year and Month—Form Component

We can see that the **year** is a **select** tag that has a two-way binding with **cDoc.Year**, which is achieved using the **ngModel** directive.

The available **years** range from **2030** to **2020**, each of which is added as an **option** to the **select** tag. This is achieved by using the ***ngFor** directive, by looping through the **years** array (which has values **2030** to **2020**). The **value** of the chosen **year** is shown by using the double curly braces syntax **{{ year }}**.

With regards to the **month**, we can see that the **select** tag for it has also a two-way binding with **cDoc.Month**, which is achieved using the **ngModel** directive.

The available **months** range from **Jan** to **Dec**, each of which is added as an **option** to the **select** tag. This is achieved by using the ***ngFor** directive, by looping through the **months** array.

The value of the chosen **month** is shown by using the double curly braces syntax **{{ month }}**.

Following that, we have the **day** and **alertyear** fields. Let's have a look at the following diagram to see how the markup of both relates to the UI.

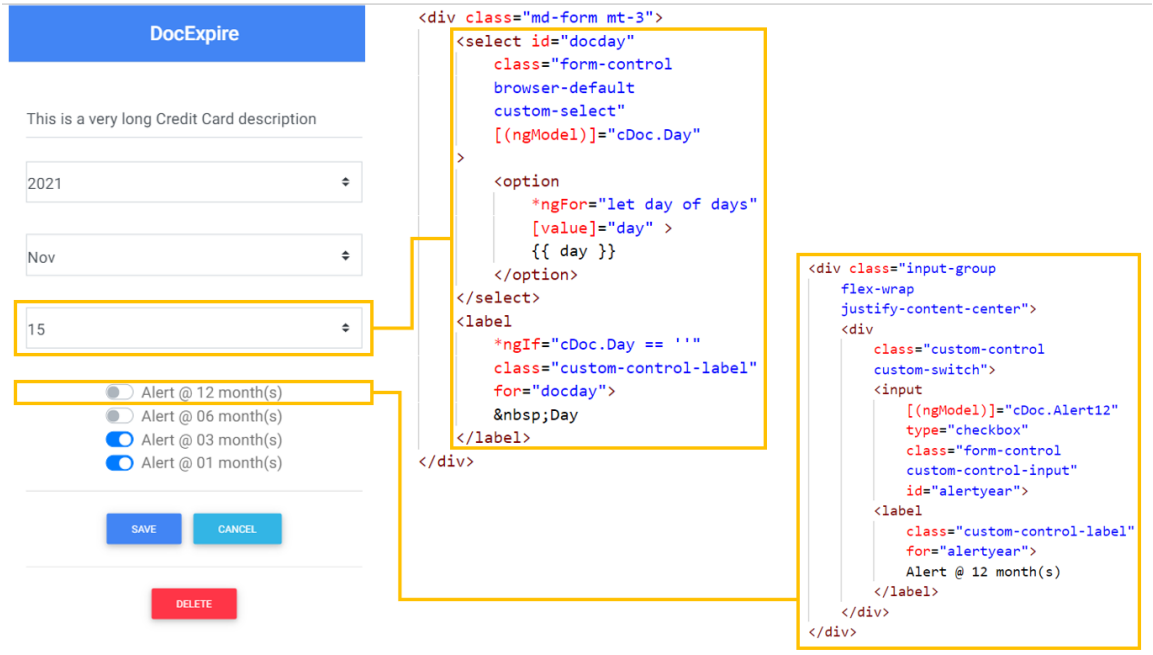

Figure 4-c: Day and 12 Months Alert—Form Component

We can see that the **day** is also a **select** tag that has a two-way binding with **cDoc.Day**, which is achieved using the **ngModel** directive.

The available **days** range mostly from **01** to **31**, but depending on the value of the chosen **month**, the range could be until **28** or **30**, each of which is added as an **option** to the **select** tag. This is achieved by using the ***ngFor** directive, by looping through the **days** array.

The **value** of the chosen **day** is shown with the double curly braces syntax **{{ day }}**.

With regards to the **alertyear** field, we can see it relates to an **input** tag of type **checkbox** (with its corresponding **label**), with a two-way binding with the **cDoc.Alert12** property, which is achieved using the **ngModel** directive.

Next, let's have a look at the **alert6months** and **alert3months** fields, and how their markup relates to the UI. You'll find this almost identical to the **alertyear** field.

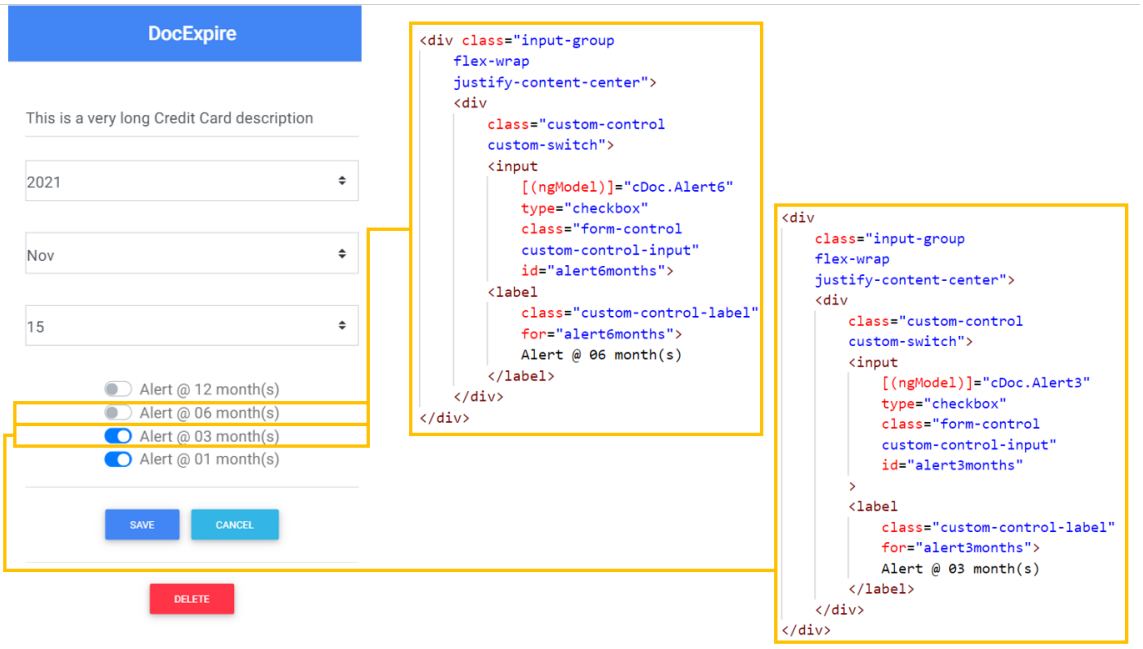

Figure 4-d: 6 and 3 Months Alerts—Form Component

Regarding the **`alert6months`** field, we can see it relates to an **`input`** tag of type **`checkbox`** (with its corresponding **`label`**) with a two-way binding with the **`cDoc.Alert6`** property, which is achieved using the **`ngModel`** directive.

As you can see, the **`alert3months`** field is the same as **`alert6months`**, with the difference that **`alert3months`** binds to **`cDoc.Alert3`**.

Next, let's have a look at the **`alert1month`** field and Save button.

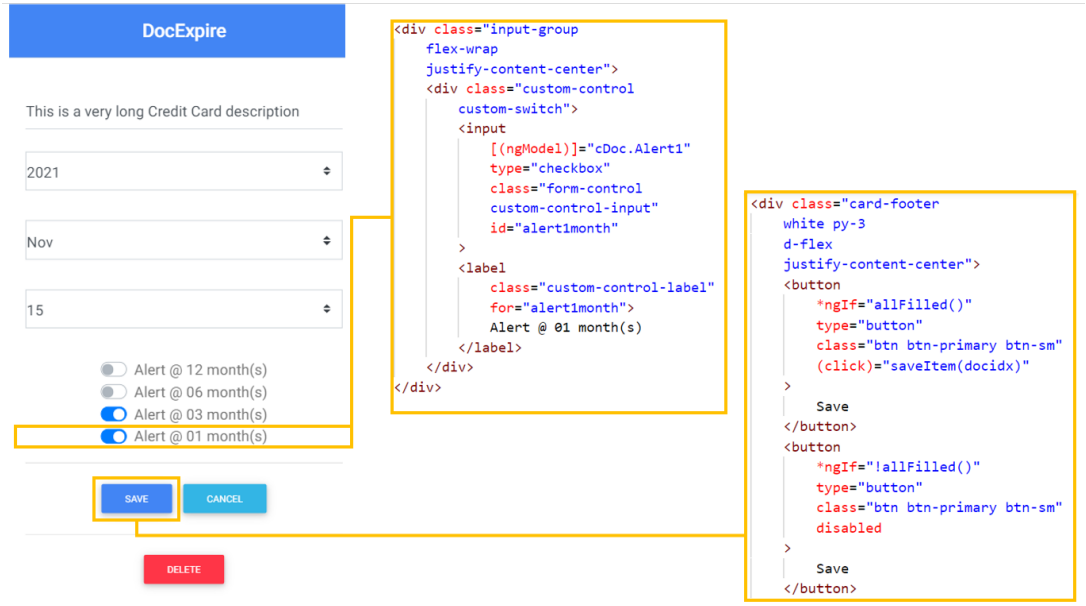

Figure 4-e: 1 Month Alert and Save Button—Form Component

Similar to other alert fields, the **alert1month** field relates to an **input** tag of type **checkbox** (with its corresponding **label**) with a two-way binding with the **cDoc.Alert1** property, which is achieved using the **ngModel** directive.

With regards to the Save button, we have two versions of the same button. The first one is displayed when all the fields in the form have been filled in (all fields have a value)—this shows a Save button that is enabled and clickable. This is visible when ***ngIf="allFilled()"** evaluates to **true**.

Notice that the Save button has a **saveItem** method that gets triggered when the **click** event executes.

The **saveItem** method will raise an event on the parent component, **de-base**, which will insert (by executing **insertDoc**) or update (by executing **updateDoc**) the document on Firebase and in the list of documents.

The second version is shown when not all the fields in the form have been filled in, displaying a Save button that is disabled and not clickable. This is visible when ***ngIf="!allFilled()"** evaluates to **true**. This button has no event attached to it.

Now, let's have a look at the Cancel and Delete buttons.

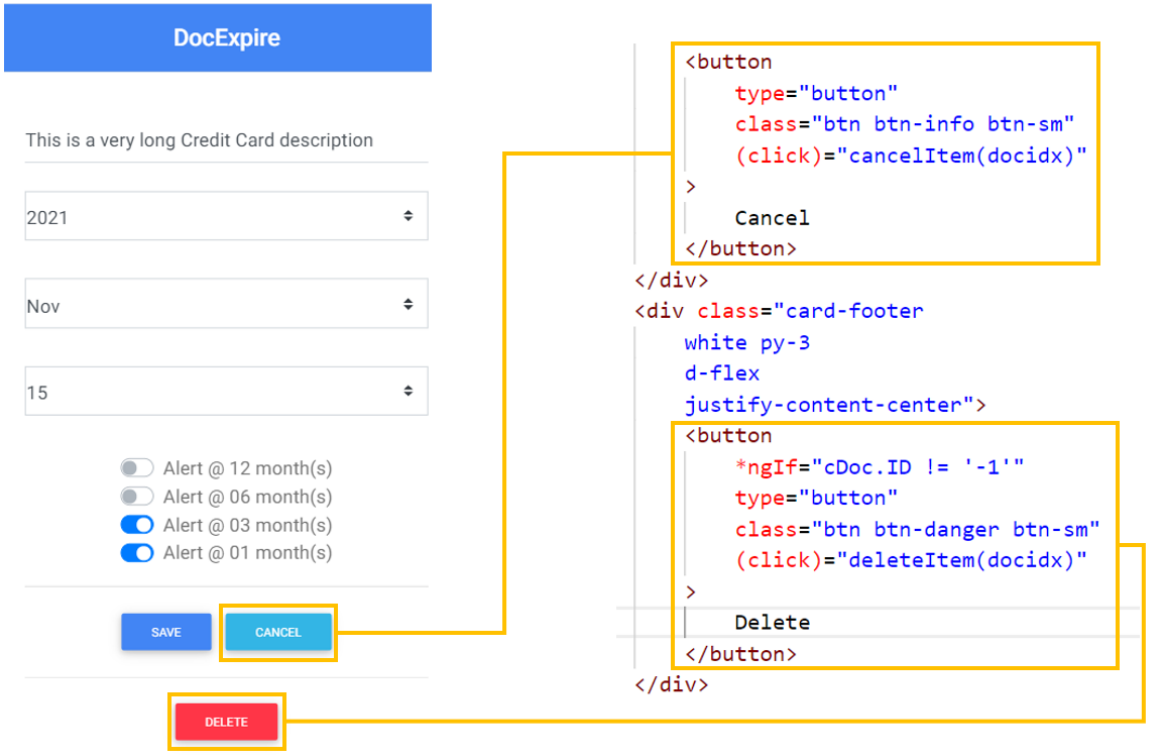

Figure 4-f: The Cancel and Delete Buttons—Form Component

The Cancel and Delete buttons are very similar to the Save button in terms of functionality, but with some slight differences.

The Cancel button is always displayed, and it has a **cancelItem** method that gets triggered when the **click** event executes.

The **cancelItem** method will raise an event on the parent component, **de-base**, which will cancel the insertion or update of the document (by executing **cancelNew**) on the document list.

The Delete button, on the other hand, is not always displayed. It is only displayed when ***ngIf="cDoc.ID != '-1'"** evaluates to **true**—when the value of the current document's ID (**cDoc.ID**) is different than the string value of **-1**. In other words, the Delete button is shown when the document is not new (when an existing document is being edited).

The Delete button has a **deleteItem** method that gets triggered when the **click** event executes. The **deleteItem** method will raise an event on the parent component, **de-base**, which will remove the document (by executing **removeDoc**) from Firebase and the document list.

Awesome! We now have a good understanding of how the markup of the **de-form** component relates to the UI.

Component markup

Now that we have explored how the **de-form** component is structured and how the UI elements relate to the markup, let's have a look at the full HTML markup of the component.

Within VS Code, open the **de_form_component.html** file and add the code from the following listing. This markup uses regular Bootstrap classes that have a Material Design look and feel, just like with the **de-base** component.

Code Listing 4-a: de_form_component.html (Full Code)

```
<div class="card">
    <h5 class="card-header primary-color
        white-text text-center py-3">
        <strong>DocExpire</strong>
    </h5>
    <div class="card-body px-lg-5">
        <form novalidate class="text-center"
            style="color: #757575;">
            <div class="md-form mt-3">
                <input type="text" id="docname"
                    class="form-control"
                    [(ngModel)]="cDoc.Document"
                >
                <label *ngIf=
                    "cDoc.Document == ''"
                    for="docname">
                    Name...
```

```
            </label>
        </div>
        <div class="md-form mt-3">
            <select id="docyear"
                class="form-control
                    browser-default
                    custom-select"
                    [(ngModel)]="cDoc.Year"
                >
                <option
                    *ngFor="let year of years"
                    [value]="year" >
                    {{ year }}
                </option>
            </select>
            <label
                *ngIf="cDoc.Year == ''"
                class="custom-control-label"
                for="docyear">
                 Year
            </label>
        </div>
        <div class="md-form mt-3">
            <select id="docmonth"
                class="form-control
                browser-default
                custom-select"
                [(ngModel)]="cDoc.Month"
                (change)="monthChangeHandler($event)"
            >
                <option
                    *ngFor="let month of months"
                    [value]="month" >
                    {{ month }}
                </option>
            </select>
            <label
                *ngIf="cDoc.Month == ''"
                class="custom-control-label"
                for="docmonth">
                 Month
            </label>
        </div>
        <div class="md-form mt-3">
```

```
            <select id="docday"
                class="form-control
                browser-default
                custom-select"
                [(ngModel)]="cDoc.Day"
            >
                <option
                    *ngFor="let day of days"
                    [value]="day" >
                    {{ day }}
                </option>
            </select>
            <label
                *ngIf="cDoc.Day == ''"
                class="custom-control-label"
                for="docday">
                 Day
            </label>
        </div>
        <div class="input-group
            flex-wrap
            justify-content-center">
            <div
                class="custom-control
                custom-switch">
                <input
                    [(ngModel)]="cDoc.Alert12"
                    type="checkbox"
                    class="form-control
                    custom-control-input"
                    id="alertyear">
                <label
                    class="custom-control-label"
                    for="alertyear">
                    Alert @ 12 month(s)
                </label>
            </div>
        </div>
        <div class="input-group
            flex-wrap
            justify-content-center">
            <div
                class="custom-control
                custom-switch">
```

```
                <input
                    [(ngModel)]="cDoc.Alert6"
                    type="checkbox"
                    class="form-control
                    custom-control-input"
                    id="alert6months">
                <label
                    class="custom-control-label"
                    for="alert6months">
                    Alert @ 06 month(s)
                </label>
            </div>
        </div>
        <div
            class="input-group
            flex-wrap
            justify-content-center">
            <div
                class="custom-control
                custom-switch">
                <input
                    [(ngModel)]="cDoc.Alert3"
                    type="checkbox"
                    class="form-control
                    custom-control-input"
                    id="alert3months"
                >
                <label
                    class="custom-control-label"
                    for="alert3months">
                    Alert @ 03 month(s)
                </label>
            </div>
        </div>
        <div class="input-group
            flex-wrap
            justify-content-center">
            <div class="custom-control
                custom-switch">
                <input
                    [(ngModel)]="cDoc.Alert1"
                    type="checkbox"
                    class="form-control
                    custom-control-input"
```

```
                    id="alert1month"
                >
                <label
                    class="custom-control-label"
                    for="alert1month">
                    Alert @ 01 month(s)
                </label>
            </div>
        </div>
        <div class="row"><p> </p></div>
        <div class="card-footer
            white py-3
            d-flex
            justify-content-center">
            <button
                *ngIf="allFilled()"
                type="button"
                class="btn btn-primary btn-sm"
                (click)="saveItem(docidx)"
            >
                Save
            </button>
            <button
                *ngIf="!allFilled()"
                type="button"
                class="btn btn-primary btn-sm"
                disabled
            >
                Save
            </button>
            <button
                type="button"
                class="btn btn-info btn-sm"
                (click)="cancelItem(docidx)"
            >
                Cancel
            </button>
        </div>
        <div class="card-footer
            white py-3
            d-flex
            justify-content-center">
            <button
                *ngIf="cDoc.ID != '-1'"
```

```
                    type="button"
                    class="btn btn-danger btn-sm"
                    (click)="deleteItem(docidx)"
                >
                    Delete
                </button>
            </div>
        </form>
    </div>
</div>
```

By reviewing the **de-form** component's structure in the previous section, we covered most of the functionality behind this markup. As you might have noticed, the markup is annotated with some AngularDart directives and specific syntax, which we covered when we reviewed the **de-base** markup. Nevertheless, let's go over each of the parts of the **de-form** markup to make sure we have everything covered.

The **de-form** component is contained within a Bootstrap for Material Design card class, which is described as follows.

```
<div class="card">

</div>
```

After the component's header, we find the **form** itself, which is described as follows.

```
<form novalidate class="text-center" style="color: #757575;">
```

Notice that the **form** is marked with the **novalidate** attribute, which means that the input should not be validated when submitted.

The document description field contains an **input** tag and a **label**, as follows.

```
<input type="text" id="docname" class="form-control"
  [(ngModel)]="cDoc.Document">
<label *ngIf="cDoc.Document == ''" for="docname">Name...</label>
```

By using the **[(ngModel)]** directive, it is possible to perform two-way binding between the **input** tag's value and the value of the **cDoc.Document** property. When the value of the **input** tag is empty—when ***ngIf="cDoc.Document == ''"** evaluates to **true**—then the **Name…** value is displayed on the **input** tag.

Figure 4-g shows how the **input** tag's value looks when the document's description (**cDoc.Document**) is empty. As you can see, the **Name…** value is displayed.

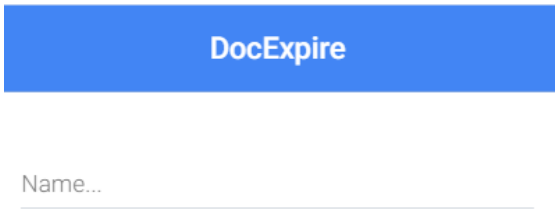

Figure 4-g: Empty Document Description—Form Component

Now, let's explore the how the year drop-down (**docyear**) works.

```
<select id="docyear" class="form-control browser-default
  custom-select" [(ngModel)]="cDoc.Year">
  <option
    *ngFor="let year of years" [value]="year" >
    {{ year }}
  </option>
</select>
<label
  *ngIf="cDoc.Year == ''" class="custom-control-label" for="docyear">
   Yea

</label>
```

Just like with the document's description (**cDoc.Document**), the year drop-down (**docyear**) has a label associated. This label will display a default value when the **year** value has not been chosen—when ***ngIf="cDoc.Year == ''"** evaluates to **true**. Here's what it looks like.

Figure 4-h: Year Not Selected—Form Component

The year drop-down (**docyear**) has a two-way binding relationship with **cDoc.Year**, which is achieved by using the **[(ngModel)]** directive. The available options for the year drop-down (**docyear**) are added as **[value]="year"** by using the ***ngFor** directive. This is done by iterating through the **years** array, which is what ***ngFor="let year of years"** does. The values of the **years** array range from **2030** to **2020**.

Now, let's have a look at the Month drop-down (**docmonth**).

```
<select id="docmonth" class="form-control browser-default custom-select"
  [(ngModel)]="cDoc.Month"(change)="monthChangeHandler($event)"
>
  <option
    *ngFor="let month of months" [value]="month" >
    {{ month }}
```

```
  </option>
</select>
<label
  *ngIf="cDoc.Month == ''" class="custom-control-label" for="docmonth">
   Month
</label>
```

Just like with the year drop-down (**docyear**), the month drop-down (**docmonth**) has a label associated, which displays a default value when the **month** value has not been chosen. We can see this as follows.

Figure 4-i: Month Not Selected—Form Component

The month drop-down (**docmonth**) works in almost the same way as the year drop-down (**docyear**). The main difference is that **docmonth** has a two-way binding relationship with the **cDoc.Month** property, and the values of the **months** array range from **Jan** to **Dec**.

Another notable difference is that the month drop-down (**docmonth**) has a **change** event listener attached, which triggers the **monthChangeHandler** method.

The **monthChangeHandler** method is used for dynamically adding the values available for the day drop-down (**docday**). This means that, depending on the month selected, the number of days on the day drop-down will vary. For example, if **Jun** is chosen as a month, then the maximum number of days for that month will be **30**. If **Dec** is chosen as a month, then the maximum number of days for that month will be **31**.

Now, let's have a look at the day drop-down (**docday**).

```
<select id="docday" class="form-control browser-default custom-select"
   [(ngModel)]="cDoc.Day">
  <option *ngFor="let day of days" [value]="day" >
    {{ day }}
  </option>
</select>
<label
  *ngIf="cDoc.Day == ''" class="custom-control-label" for="docday">
   Day
</label>
```

As you can see, the markup for the day drop-down (**docday**) is almost identical for the year dropdown. The main differences are that **docday** has a two-way binding relationship with the **cDoc.Day** property, and the values of the **days** array varies dynamically, depending on the month selected.

Concerning the four alerts, they are pretty much identical. Let's have a look at the following one.

```
<input [(ngModel)]="cDoc.Alert12" type="checkbox" class="form-control
  custom-control-input" id="alertyear">
<label class="custom-control-label" for="alertyear">
  Alert @ 12 month(s)
</label>
```

We can see that the **alertyear** input tag is of type **checkbox** and has a two-way data binding relationship with the **cDoc.Alert12** property. The other alert tags are almost identical to this one. The only difference is that each will bind to a different property. This means that the **alert6months** input tag will bind to **cDoc.Alert6**, **alert3months** to **cDoc.Alert3**, and **alert1month** to **cDoc.Alert1**.

Regarding the Save, Cancel, and Delete buttons, the most important characteristic is that all of them have an event attached that triggers a specific method. The Save button's event triggers the **saveItem** method, the Cancel button's event triggers the **cancelItem** event, and the Delete button's event triggers the **deleteItem** event.

These methods cascade specific actions that get executed within the parent component, **de-base**, which ultimately updates the document list and interacts with Firebase.

Component logic

Now that we've explored how the markup of the **de-form** component is organized, let's see how the logic behind it works. Within VS Code, open the **de_form_component.dart** file and copy and paste the code from the following listing.

Code Listing 4-b: de_form_component.dart (Full Code)

```
import 'dart:async';

import 'package:angular/angular.dart';
import 'package:angular_forms/angular_forms.dart';

import '../model/docs.dart';

@Component(
  selector: 'de-form',
  templateUrl: 'de_form_component.html',
  directives: [coreDirectives, formDirectives]
)
class DocForm implements OnInit {
  @Input() int docidx;
  @Input() List docs;
```

```
Doc cDoc;

List<String> years = [
  '2030',
  '2029',
  '2028',
  '2027',
  '2026',
  '2025',
  '2024',
  '2023',
  '2022',
  '2021',
  '2020'
];

List<String> months = [
  'Jan',
  'Feb',
  'Mar',
  'Apr',
  'May',
  'Jun',
  'Jul',
  'Aug',
  'Sep',
  'Oct',
  'Nov',
  'Dec'
];

List<int> daysMonth = [
  31,
  28,
  31,
  30,
  31,
  30,
  31,
  31,
  30,
  31,
  30,
  31
```

```
];

List<String> days = [];

final _removeCtrl = StreamController();
@Output('onDelete')
Stream get removeItem => _removeCtrl.stream;

final _cancelNewCtrl = StreamController();
@Output('onCancelNew')
Stream get cancelNewItem => _cancelNewCtrl.stream;

final _insertCtrl = StreamController();
@Output('onInsert')
Stream get insertItem => _insertCtrl.stream;

final _updateCtrl = StreamController();
@Output('onUpdate')
Stream get updateItem => _updateCtrl.stream;

@override
Future<Null> ngOnInit() async {
  cDoc = docs[docidx];
  updateDays(months.indexOf(cDoc.Month));
}

void monthChangeHandler(event) {
  var midx = months.indexOf(event.target.value);
  updateDays(midx);
}

void updateDays(int m) {
    if (m > -1) {
      days = List<String>.
        generate(daysMonth[m],
        (i) => (i + 1).toString());
    }
}

void saveItem(int index) {
  // insert item
  if (cDoc.ID == '-1') {
    _insertCtrl.add(null);
  }
```

```
    // update item
    else {
      _updateCtrl.add(null);
    }
    docs[index].Edit = false;
    docs[index]..update(cDoc);
  }

  bool allFilled() {
    var r = false;
    if (cDoc.Document != '' &&
        cDoc.Year != '' &&
        cDoc.Month != '' &&
        cDoc.Day != '') {
        r = true;
    }
    return r;
  }

  void cancelItem(int index) {
    docs[index].Edit = false;
    _cancelNewCtrl.add(null);
  }

  void deleteItem(int index) {
    cancelItem(index);
    _removeCtrl.add(null);
  }
}
```

To get a full understanding of this logic, let's break it down into smaller parts to see what each one does.

Component initialization

Just like we did with the **de-base** component, we start by importing the required modules that the **de-form** component will need.

```
import 'dart:async';

import 'package:angular/angular.dart';
import 'package:angular_forms/angular_forms.dart';

import '../model/docs.dart';
```

We reference Dart's asynchronous library, the AngularDart and AngularDart forms, and finally, the application's data model (**docs.dart**)—which we will explore later.

Next, we find the **@Component** annotation, which makes **DocForm** an AngularDart component, rather than a standard Dart class.

The **selector** property of the **@Component** annotation specifies how the component will be referenced within the markup—which, in this case, is **de-form**.

The **templateUrl** property of the **@Component** annotation indicates the name of the file that contains the source code for the markup—which, in this case, is **de_form_component.html**.

The **directives** property of the **@Component** annotation indicates the name of the directives that will be used by the component—**coreDirectives** and **formDirectives**.

Just like the **de-base** component, the **de-form** component implements the **ngOnInit** method, which is used for calling the initialization code. This is described by the **implements OnInit** instruction.

Component inputs

As we have seen, the **de-form** component is embedded within the **de-base** component's markup—as you can see in the following code. The variables **docidx** and **docs** are passed to **de-form**.

```
<de-form *ngIf="ifEditing()"[docidx]="idx" [docs]="docs"
 (onDelete)="removeDoc(idx)" (onCancelNew)="cancelNew()"
 (onInsert)="insertDoc(idx)" (onUpdate)="updateDoc(idx)">
</de-form>
```

For the **de-form** component to be able to receive these variables from the parent component (**de-base**), we need to declare them within the **DocForm** class and annotate each with the **@Input()** attribute. We can see this as follows.

```
@Input() int docidx;
@Input() List docs;
```

By annotating these variables with the **@Input()** annotation, we are indicating to the **DocForm** class that these are variables that are passed from **de-base** to **de-form**. This is the way to pass data from a parent component to a child component in AngularDart.

Other component variables

The **cDoc** variable is an instance of type **Doc**, and it contains the data of the document being edited or created through the **de-form** UI.

Moving on, we find the **years**, **months**, **daysMonth**, and **days** lists, which will be used to fill in the values of the drop-`downs.

Component events

The **de-form** component can raise events by using Dart's StreamController class, which is designed to send data by creating a stream that other components can listen on. This is how data is passed from the child component to the parent component—in our case, from **de-form** to **de-base**.

The **de-form** component will raise an event for every action that may occur when a document is being edited or created. Two events might occur when a document is being edited: **onDelete** or **onUpdate**. Let's have a look at the following code to see how this works.

```
final _removeCtrl = StreamController();
@Output('onDelete')
Stream get removeItem => _removeCtrl.stream;
```

The **onDelete** event gets triggered when the Delete button is clicked. For the **onDelete** event, we create an immutable (**final**) instance of a **StreamController** class. The **stream** property of that **StreamController** instance is assigned to the **Stream** that will be invoked by the **onDelete** event, which in this case is **removeItem**.

For this event to be intercepted by the parent component (**de-base**) so that its corresponding parent method (**removeDoc**) is invoked, the **onDelete** event is marked with the **@Output** annotation attribute.

The way the **onUpdate** event works is almost identical to the way the **onDelete** event works. Let's have a look.

```
final _updateCtrl = StreamController();
@Output('onUpdate')

Stream get updateItem => _updateCtrl.stream;
```

The **onUpdate** event gets triggered when the Save button is clicked (for an existing document). So, for the **onUpdate** event, we create an immutable (**final**) instance of a **StreamController** class.

The **stream** property of that **StreamController** instance is assigned to the **Stream** that will be invoked by the **onUpdate** event, which in this case is **updateItem**.

For this event to be intercepted by the parent component (**de-base**) so that its corresponding parent method (**updateDoc**) is invoked, the **onUpdate** event is marked with the **@Output** annotation attribute.

The triggering of the **onCancelNew** event follows the same logic and process.

```
final _cancelNewCtrl = StreamController();
@Output('onCancelNew')
Stream get cancelNewItem => _cancelNewCtrl.stream;
```

The **onCancelNew** event gets triggered when the Cancel button is clicked. So, for the **onCancelNew** event, we create an immutable (**final**) instance of a **StreamController** class.

The **stream** property of that **StreamController** instance is assigned to the **Stream** that will be invoked by the **onCancelNew** event, which in this case is **cancelNewItem**.

For this event to be intercepted by the parent component **de-base**—so it's corresponding parent method (**cancelNew**) is invoked—the **onCancelNew** event is marked with the **@Output** annotation attribute.

On the other hand, only one event occurs when a new document is created and saved: **onInsert**. The way this event gets triggered follows the same logical process.

```
final _insertCtrl = StreamController();
@Output('onInsert')
Stream get insertItem => _insertCtrl.stream;
```

The **onInsert** event gets triggered when the Save button is clicked (for a new document). So, for the **onInsert** event, we create an immutable (**final**) instance of a **StreamController** class. The **stream** property of that **StreamController** instance is assigned to the **Stream** that will be invoked by the **onInsert** event, which in this case is **insertItem**.

For this event to be intercepted by the parent component (**de-base**) so its corresponding parent method (**insertDoc**) is invoked, the **onInsert** event is marked with the **@Output** annotation attribute.

Now that we know how the events within the **de-form** component get triggered, and how those events are intercepted by the parent component (**de-base**) to carry out the corresponding operations with Firebase and the list of documents, let's focus on the **de-form** component methods.

Component methods

With the component's initialization and events covered, it's now time to dive into the component's methods and what they do. We'll start with the **ngOnInit** method, which the **DocForm** class implements, as you can see in the following listing.

Code Listing 4-c: de_form_component.dart (ngOnInit Method)

```
@override
Future<Null> ngOnInit() async {
```

```
  cDoc = docs[docidx];
  updateDays(months.indexOf(cDoc.Month));
}
```

This method—which is asynchronous (**async**) and returns a **Future** (a promise)—gets executed when the **de-form** component loads (thus the **OnInit** suffix). This method is used for assigning to the **cDoc** instance the reference of the document being edited or inserted (**docs[docidx]**).

Once that has been done, the day drop-down is updated with the corresponding number of days available for that chosen month (**cDoc.Month**). This is only applicable when an existing document is being edited, which is what the **updateDays** method does.

Let's explore in detail how the days are updated, depending on the month selected.

Updating days

The number of days available, which depends on the month selected, gets updated on two occasions: when the component loads (as we just saw), and when the month selected changes. Let's have a look.

Code Listing 4-d: de_form_component.dart (Updating Days Methods)

```
void monthChangeHandler(event) {
  var midx = months.indexOf(event.target.value);
  updateDays(midx);
}

void updateDays(int m) {
  if (m > -1) {
    days = List<String>.
      generate(daysMonth[m], (i) => (i + 1).toString());
  }
}
```

The **monthChangeHandler** method gets executed when the month changes, for which the **change** event is responsible. This method obtains the selected month by inspecting the **event.target.value** property. Once the selected month is known, then the **updateDays** method is invoked.

The **updateDays** method dynamically generates the **days** list values, based on the value of **daysMonth** for the selected month (**m**).

Checking filled fields

As you might recall, the enabled Save button is displayed (and becomes clickable) when all the fields have been filled in; this is achieved with the **allFilled** method. Let's have a look.

Code Listing 4-e: de_form_component.dart (allFilled Method)

```
bool allFilled() {
  var r = false;
  if (cDoc.Document != '' && cDoc.Year != '' && cDoc.Month != '' &&
      cDoc.Day != '') {
    r = true;
  }
  return r;
}
```

This method checks the values of the **Document**, **Year**, **Month**, and **Day** properties, and if all are not empty, then **true** is returned—which indicates that all fields have been filled in.

If this method returns **false**, it means at least one field has not been filled in. In that case, the disabled Save button (which is not clickable) is displayed instead.

Saving a document

One of the key aspects of the **de-form** component is its ability to save a new or existing document, which triggers either an insert or update operation on the parent component, **de-base**. Let's explore how saving a document works.

Code Listing 4-f: de_form_component.dart (saveItem Method)

```
void saveItem(int index) {
  // insert item
  if (cDoc.ID == '-1') {
    _insertCtrl.add(null);
  }
  // update item
  else {
    _updateCtrl.add(null);
  }
  docs[index].Edit = false;
  docs[index]..update(cDoc);
}
```

As we can see, the **saveItem** method has two distinct parts, which are visible by the comments in the code.

When an insert operation is taking place—when a new document is being added—we can see that **_insertCtrl** (the **StreamController** instance) is invoked. This results in the **insertDoc** method being invoked on the parent component, **de-base**.

On the other hand, when an update operation is taking place—when an existing document is being edited—we can see that **_updateCtrl** (the **StreamController** instance) is invoked. This results in the **updateDoc** method being invoked on the parent component, **de-base**.

The **saveItem** method sets the **Edit** property of the current document being worked on (**docs[index]**) to **false**, indicating that the document doesn't need to be further changed.

Then, the **saveItem** method invokes the **update** method from the model—passing the current document instance (**cDoc**) as a parameter to the **update** method.

Notice that the **update** method is called with what is known in Dart as the cascade notation (..), which means that we can execute operations in sequence on the same object.

Canceling a document

Another key aspect of the **de-form** component is its ability to cancel a document from being inserted or updated. Let's have a look at how this is done.

Code Listing 4-g: de_form_component.dart (cancelItem Method)

```
void cancelItem(int index) {
  docs[index].Edit = false;
  _cancelNewCtrl.add(null);
}
```

All that happens is the **Edit** property of the current document being worked on (**docs[index]**) is set to **false**, and the **_cancelNewCtrl** (the **StreamController** instance) is invoked—which results in the **cancelNew** method being invoked on the parent component.

Deleting a document

Another important aspect of the **de-form** component is its ability to delete a document. Let's see how that works.

Code Listing 4-h: de_form_component.dart (deleteItem Method)

```
void deleteItem(int index) {
  cancelItem(index);
  _removeCtrl.add(null);
}
```

As you can see, deleting a document calls the **`cancelItem`** method, and then the **`_removeCtrl`** (the **`StreamController`** instance) is invoked—which results in the **`removeDoc`** method being invoked on the parent component.

Cool—that's all the code required for the **`de-form`** component to work.

Summary

At this stage, we've written the markup and logic of the two components our application uses, **`de-base`** and **`de-form`**, which, as you have seen, wasn't that difficult. In the chapters that follow, we will finish the application's logic by creating the app's model and service provider, which will be responsible for interacting with Firebase. Once we've finished creating the application's model, we will set up a Firebase real-time database, and then create the service provider to finalize the app.

Chapter 5 Model

Overview

We have reached an important milestone—our app's components are ready. However, there are two important parts of our app's logic we are missing: the model and service provider. In this chapter, we are going to create our application's model.

Model

The application's model is going to contain the object definition of the data that will be stored in Firebase (along with some utility methods). The model is going to be used by the service provider that will be responsible for fetching, inserting, updating, and deleting documents.

In VS Code, go to the **model** folder within your app's project and open the **docs.dart** file previously created. Once this file is open, copy and paste the code from the following listing.

Code Listing 5-a: docs.dart (Full Code)

```
import 'dart:convert';
import 'dart:math';

class utils {
    static final Random _random = Random.secure();

    static String CreateRandomString([int length = 32]) {
        var values = List<int>.generate(
            length, (i) => _random.nextInt(256));

        return base64Url.encode(values);
    }
}

class Doc {
  List<String> months = [
    'Jan|01',
    'Feb|02',
    'Mar|03',
    'Apr|04',
    'May|05',
    'Jun|06',
    'Jul|07',
```

```
    'Aug|08',
    'Sep|09',
    'Oct|10',
    'Nov|11',
    'Dec|12'
  ];

  String ID;
  String Document;
  String Year;
  String Month;
  String Day;
  String Expires;
  String Status;
  String Badge;
  bool Edit;
  bool Alert12;
  bool Alert6;
  bool Alert3;
  bool Alert1;
  int Days;

  Doc({
    this.ID,
    this.Document,
    this.Year,
    this.Month,
    this.Day,
    this.Expires,
    this.Status,
    this.Badge,
    this.Edit = true,
    this.Alert12,
    this.Alert6,
    this.Alert3,
    this.Alert1,
    this.Days
  });

  String getMonthNum(String m) {
    var n = '';

    for (var i = 0; i < months.length; i++) {
      if (months[i].contains(m)) {
```

```
      var p = months[i].split('|');
      n = p[1];
      break;
    }
  }

  return n;
}

List determineStatus(String y, String m, String d) {
  var r = '';
  var badge = '';

  var exp = DateTime(int.parse(y),
    int.parse(getMonthNum(m)), int.parse(d));
  var diffDays = exp.difference(DateTime.now()).inDays;

  if (diffDays <= 0) {
    r = 'Expired';
    r = r.replaceAll('-', '');
    badge = 'badge badge-dark ml-2';
  }
  else if (diffDays > 0 && diffDays <= 30) {
    r = diffDays.toString() + ' day(s)';
    badge = 'badge badge-danger ml-2';
  }
  else if (diffDays > 30 && diffDays <= 60) {
    r = diffDays.toString() + ' day(s)';
    badge = 'badge badge-warning ml-2';
  }
  else if (diffDays > 60 && diffDays <= 90) {
    r = diffDays.toString() + ' day(s)';
    badge = 'badge badge-secondary ml-2';
  }
  else if (diffDays > 90 && diffDays <= 120) {
    r = diffDays.toString() + ' day(s)';
    badge = 'badge badge-info ml-2';
  }
  else if (diffDays > 120 && diffDays <= 240) {
    r = diffDays.toString() + ' day(s)';
    badge = 'badge badge-primary ml-2';
  }
  else if (diffDays > 240 && diffDays <= 365) {
    r = diffDays.toString() + ' day(s)';
```

```
    badge = 'badge badge-default ml-2';
  }
  else if (diffDays > 365) {
    r = diffDays.toString() + ' day(s)';
    badge = 'badge badge-success ml-2';
  }

  return [r, diffDays, badge];
}

void update(Doc doc) {
  if (doc.ID == '-1') {
    ID = utils.CreateRandomString();
  }

  Document = doc.Document;
  Year = doc.Year;
  Month = doc.Month;
  Day = doc.Day;

  Expires = doc.Day + '-' +
    doc.Month + '-' + doc.Year;

  var st = determineStatus(Year, Month, Day);
  Status = st[0];
  Badge = st[2];

  Alert12 = doc.Alert12;
  Alert6 = doc.Alert6;
  Alert3 = doc.Alert3;
  Alert1 = doc.Alert1;
  Days = st[1];
}

static Function(Doc, Doc) sorter(int sortOrder, String property) {
   int handleSortOrder(int sortOrder, int sort) {
     if (sortOrder == 1) {
       // a is before b
       if (sort == -1) {
         return -1;
       } else if (sort > 0) {
         // a is after b
         return 1;
       } else {
```

```
          // a is same as b
          return 0;
        }
      }
    }

    return (Doc a, Doc b) {
     if (a != null && b != null) {
       switch (property) {
         case 'Days':
             var sort = a.Days.compareTo(b.Days);
             return handleSortOrder(sortOrder, sort);
         default:
             break;
       }
     }
   };
}

// sortOrder = 1 ascending | 0 descending
static void sortDocs(List<Doc> docs,
  {int sortOrder = 1, String property = 'Days'}) {
  if (docs.isNotEmpty) {
    switch (property) {
      case 'Days':
        docs.sort(sorter(sortOrder, property));
        break;
      default:
        print('Unrecognized property $property');
    }
  }
}

// Firebase
Doc.fromMap(Map mp) : this (
  ID: mp['ID'].toString(),
  Document: mp['Document'].toString(),
  Year: mp['Year'].toString(),
  Month: mp['Month'].toString(),
  Day: mp['Day'].toString(),
  Expires: mp['Expires'].toString(),
  Status: mp['Status'].toString(),
  Badge: mp['Badge'].toString(),
  Edit:
```

```
      mp['Edit'].
        toString().
        toLowerCase() == 'true' ?? false,
    Alert12:
      mp['Alert12'].
        toString().
        toLowerCase() == 'true' ?? false,
    Alert6:
      mp['Alert6'].
        toString().
        toLowerCase() == 'true' ?? false,
    Alert3:
      mp['Alert3'].
        toString().
        toLowerCase() == 'true' ?? false,
    Alert1:
      mp['Alert1'].
        toString().
        toLowerCase() == 'true' ?? false,
    Days: int.parse(
      mp['Days'].toString())
  );

  // Firebase
  Map asMap() => {
    'ID': ID,
    'Document': Document,
    'Year': Year,
    'Month': Month,
    'Day': Day,
    'Expires': Expires,
    'Status': Status,
    'Badge': Badge,
    'Edit': Edit,
    'Alert12': Alert12,
    'Alert6': Alert6,
    'Alert3': Alert3,
    'Alert1': Alert1,
    'Days': Days
  };
}
```

To understand what is happening here, let's dissect the code into smaller parts. We start by importing the modules we need.

```
import 'dart:convert';

import 'dart:math';
```

We'll be using Dart's convert and math standard libraries, which are key to converting variables from one object type to another and performing mathematical operations, respectively.

Unique document ID

The **utils** class contains a method called **CreateRandomString**, which is used for generating a unique ID string (randomly generated) for each document.

Code Listing 5-b: docs.dart (The utils Class)

```
class utils {
    static final Random _random = Random.secure();

    static String CreateRandomString([int length = 32]) {
        var values = List<int>.generate(
            length, (i) => _random.nextInt(256));

        return base64Url.encode(values);
    }
}
```

The generated string returned by the **CreateRandomString** method is Base64 encoded.

Doc class initialization

The **Doc** class is used for representing how documents can be stored as objects, not only during runtime, but also in Firebase. Let's explore its details.

Code Listing 5-c: docs.dart (The Doc Class)

```
class Doc {
  List<String> months = [
    'Jan|01',
    'Feb|02',
    'Mar|03',
    'Apr|04',
    'May|05',
    'Jun|06',
    'Jul|07',
    'Aug|08',
```

```
    'Sep|09',
    'Oct|10',
    'Nov|11',
    'Dec|12'
  ];

  String ID;
  String Document;
  String Year;
  String Month;
  String Day;
  String Expires;
  String Status;
  String Badge;
  bool Edit;
  bool Alert12;
  bool Alert6;
  bool Alert3;
  bool Alert1;
  int Days;

  Doc({
    this.ID,
    this.Document,
    this.Year,
    this.Month,
    this.Day,
    this.Expires,
    this.Status,
    this.Badge,
    this.Edit = true,
    this.Alert12,
    this.Alert6,
    this.Alert3,
    this.Alert1,
    this.Days
  });

 // More code to follow

}
```

We start by looking at the **months** list, which is used as a placeholder that contains the names of each month, along with their corresponding month number.

```
List<String> months = ['Jan|01', 'Feb|02', 'Mar|03', 'Apr|04', 'May|05',
'Jun|06', 'Jul|07', 'Aug|08', 'Sep|09', 'Oct|10', 'Nov|11', 'Dec|12'];
```

This is used within the **determineStatus** method to calculate how many days remain until a document expires.

Following that, we find the properties that our **Doc** class uses declared as follows.

```
String ID;
…

int Days;
```

Finally, we have the **Doc** class constructor, which is used to initialize the properties declared.

```
Doc({this.ID, this.Document, this.Year, this.Month, this.Day, this.Expires,
     this.Status, this.Badge, this.Edit = true, this.Alert12, this.Alert6,
     this.Alert3, this.Alert1, this.Days});
```

Notice that by default, the **Edit** property is set to **true** within the constructor's initialization. This means that when we create an instance of the **Doc** class, we are indicating our intention of using it to edit an existing document or create a new one.

Days remaining

One of the application's coolest features—which is visible on the app's main screen (**de-base** component)—is that, for each document, there is a badge that indicates how many days remain before the document expires.

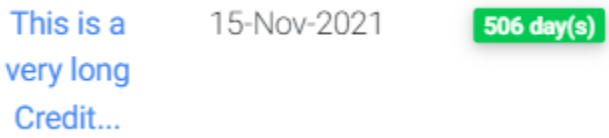

Figure 5-a: Days Remaining Badge (Right-Hand)

The creation of this badge and calculation of the number of days remaining is an integral part of every document, which is why this functionality is included within the model, and specifically within the **Doc** class.

Let's have a look at how this calculation is done by exploring the following two methods.

Code Listing 5-d: docs.dart (Days Remaining Methods)

```
String getMonthNum(String m) {
  var n = '';

  for (var i = 0; i < months.length; i++) {
    if (months[i].contains(m)) {
      var p = months[i].split('|');
      n = p[1];
      break;
    }
  }

  return n;
}

List determineStatus(String y, String m, String d) {
  var r = '';
  var badge = '';

  var exp = DateTime(int.parse(y),
    int.parse(getMonthNum(m)), int.parse(d));
  var diffDays = exp.difference(DateTime.now()).inDays;

  if (diffDays <= 0) {
    r = 'Expired';
    r = r.replaceAll('-', '');
    badge = 'badge badge-dark ml-2';
  }
  else if (diffDays > 0 && diffDays <= 30) {
    r = diffDays.toString() + ' day(s)';
    badge = 'badge badge-danger ml-2';
  }
  else if (diffDays > 30 && diffDays <= 60) {
    r = diffDays.toString() + ' day(s)';
    badge = 'badge badge-warning ml-2';
  }
  else if (diffDays > 60 && diffDays <= 90) {
    r = diffDays.toString() + ' day(s)';
    badge = 'badge badge-secondary ml-2';
  }
  else if (diffDays > 90 && diffDays <= 120) {
    r = diffDays.toString() + ' day(s)';
```

```
    badge = 'badge badge-info ml-2';
  }
  else if (diffDays > 120 && diffDays <= 240) {
    r = diffDays.toString() + ' day(s)';
    badge = 'badge badge-primary ml-2';
  }
  else if (diffDays > 240 && diffDays <= 365) {
    r = diffDays.toString() + ' day(s)';
    badge = 'badge badge-default ml-2';
  }
  else if (diffDays > 365) {
    r = diffDays.toString() + ' day(s)';
    badge = 'badge badge-success ml-2';
  }

  return [r, diffDays, badge];
}
```

From the preceding code, we can see that the main method used for calculating the number of days remaining is **`determineStatus`**. This method receives the expiration year (**`y`**), month (**`m`**), and day (**`d`**) of the current document as parameters.

These values are then converted into the **`DateTime`** representation of the document's expiration date by the following instruction, for which the **`getMonthNum`** method is also used.

```
var exp = DateTime(int.parse(y), int.parse(getMonthNum(m)), int.parse(d));
```

The number of days (**`diffDays`**) remaining is calculated by the difference between the document's expiration date (**`exp`**) and the current date and time.

```
var diffDays = exp.difference(DateTime.now()).inDays;
```

Depending on the number of days remaining (**`diffDays`**), the correct badge value (**`badge`**) and response (**`r`**) are assigned. The badge value represents the CSS class and color scheme, which vary depending on the number of days left.

The **`determineStatus`** method returns the **`badge`**, text response (**`r`**), and the number of days remaining (**`diffDays`**). As you have seen, calculating the number of days is not difficult.

Doc update

As you might recall, in the previous chapter we explored what happens when saving a document (within the **`de-form`** component)—specifically on the **`saveItem`** method—where we invoke the **`update`** method of a **`Doc`** instance using the Dart cascade notation.

```
docs[index]..update(cDoc);
```

Let's dive into the details of what the **update** method does.

Code Listing 5-e: docs.dart (update Method)

```
void update(Doc doc) {
  if (doc.ID == '-1') {
    ID = utils.CreateRandomString();
  }

  Document = doc.Document;
  Year = doc.Year;
  Month = doc.Month;
  Day = doc.Day;

  Expires = doc.Day + '-' +
    doc.Month + '-' + doc.Year;

  var st = determineStatus(Year, Month, Day);
  Status = st[0];
  Badge = st[2];

  Alert12 = doc.Alert12;
  Alert6 = doc.Alert6;
  Alert3 = doc.Alert3;
  Alert1 = doc.Alert1;
  Days = st[1];
}
```

The **update** method receives a **Doc** object (**doc**)—the current document being edited or added.

If the **ID** property has a value of **'-1'** (which indicates a new document is being added), then the **CreateRandomString** method is called to assign a unique **ID** value to the document (**doc**).

The rest of the **update** method assigns to the instance properties the respective values of the **doc** object properties, which we can see as follows.

```
Document = doc.Document;
Year = doc.Year;
Month = doc.Month;
Day = doc.Day;
```

```
Alert12 = doc.Alert12;
Alert6 = doc.Alert6;
Alert3 = doc.Alert3;
Alert1 = doc.Alert1;
```

The values of the **Status**, **Badge**, and **Days** properties are assigned from the values returned by the **determineStatus** method, which can be seen as follows.

```
var st = determineStatus(Year, Month, Day);
Status = st[0];

Badge = st[2];

Days = st[1];
```

Sorting methods

One of the most important characteristics of the application is its ability to sort documents by the number of expiration days. This means the documents that have expired (or with the lowest number of days remaining) are shown first, and documents with the highest number of days before expiration are displayed last.

To achieve this, we rely on the **sorter** and **sortDocs** methods. Let's explore both.

Code Listing 5-f: docs.dart (Sorting Methods)

```
static Function(Doc, Doc) sorter(int sortOrder, String property) {
   int handleSortOrder(int sortOrder, int sort) {
     if (sortOrder == 1) {
       // a is before b
       if (sort == -1) {
         return -1;
       } else if (sort > 0) {
         // a is after b
         return 1;
       } else {
         // a is the same as b
         return 0;
       }
     }
   }

   return (Doc a, Doc b) {
     if (a != null && b != null) {
       switch (property) {
         case 'Days':
           var sort = a.Days.compareTo(b.Days);
           return handleSortOrder(sortOrder, sort);
         default:
           break;
```

```
      }
    }
  };
}

// sortOrder = 1 ascending | 0 descending
static void sortDocs(List<Doc> docs,
  {int sortOrder = 1, String property = 'Days'}) {
  if (docs.isNotEmpty) {
    switch (property) {
      case 'Days':
        docs.sort(sorter(sortOrder, property));
        break;
      default:
        print('Unrecognized property $property');
    }
  }
}
```

The **sortDocs** method is responsible for sorting the list of documents using the **Days** property. The **sortDocs** method invokes the list's (**docs**) **sort** method, to which the **sorter** method is passed, the method that does the actual sorting.

The **sorter** method returns a function (**Function(Doc, Doc)**) that iteratively invokes the **handleSortOrder** function—also contained within the **sorter** method—that indicates which of the two elements of the list being compared has precedence (goes first). The process repeats itself for every document on the list of documents.

The function being returned by the **sorter** method, **(Doc a, Doc b)**, performs the comparison of each pair of elements within the list (**a** and **b**), by checking the value of the **Days** property of each element. This is what **a.Days.compareTo(b.Days)** does.

As you can see, the comparison for the sorting mechanism logic is very simple and straightforward to understand. The only tricky part is how the code for the **sorter** method is structured, as it requires the logic to be split into an internal **handleSortOrder** function and another part that returns the **(Doc a, Doc b)** function, which gets passed to the list's **sort** method.

```
docs.sort(sorter(sortOrder, property));
```

fromMap and asMap

To be able to read **Doc** objects from Firebase, and display the list of documents within the application, we need to be able to read each document stored in Firebase as JSON.

To achieve that, we have to convert the document from JSON to its **Doc** object equivalent, which we can do with **fromMap**. Let's have a look.

Code Listing 5-g: docs.dart (fromMap Method)

```
Doc.fromMap(Map mp) : this (
  ID: mp['ID'].toString(),
  Document: mp['Document'].toString(),
  Year: mp['Year'].toString(),
  Month: mp['Month'].toString(),
  Day: mp['Day'].toString(),
  Expires: mp['Expires'].toString(),
  Status: mp['Status'].toString(),
  Badge: mp['Badge'].toString(),
  Edit:
    mp['Edit'].
      toString().
      toLowerCase() == 'true' ?? false,
  Alert12:
    mp['Alert12'].
      toString().
      toLowerCase() == 'true' ?? false,
  Alert6:
    mp['Alert6'].
      toString().
      toLowerCase() == 'true' ?? false,
  Alert3:
    mp['Alert3'].
      toString().
      toLowerCase() == 'true' ?? false,
  Alert1:
    mp['Alert1'].
      toString().
      toLowerCase() == 'true' ?? false,
  Days: int.parse(
    mp['Days'].toString())
);
```

What is going on here? The **fromMap** method takes a **Map** object (**mp**) as a parameter—which in Dart represents a key-value pair.

The **Doc** constructor is invoked, for which the values contained within the **mp** object are assigned to their respective properties within the **Doc** instance.

Object values stored in Firebase's real-time database, when assigned to their corresponding properties within the **Doc** instance, must be converted to the right data type. This is why the **Edit**, **Alert1**, **Alert3**, **Alert6**, and **Alert12** properties are all converted within the **fromMap** method to Boolean.

On the other hand, the **Days** property, which indicates the number of days remaining before expiration, is converted to an integer.

So, when our application retrieves data from Firebase, it uses the **fromMap** method to convert the document from JSON to its **Doc** object equivalent. But what happens when we need to take document data stored as **Doc** objects and write that to Firebase? That's when we need to use the **asMap** method. Let's have a look.

Code Listing 5-h: docs.dart (asMap Method)

```
Map asMap() => {
  'ID': ID,
  'Document': Document,
  'Year': Year,
  'Month': Month,
  'Day': Day,
  'Expires': Expires,
  'Status': Status,
  'Badge': Badge,
  'Edit': Edit,
  'Alert12': Alert12,
  'Alert6': Alert6,
  'Alert3': Alert3,
  'Alert1': Alert1,
  'Days': Days
};
```

As we can see, the **asMap** method (written as a lambda expression) creates a JSON object by returning a **Map** containing each of the values of the **Doc** object properties. Each key (property name) is assigned its corresponding value.

Summary

Awesome—we now have explored how our application's model works, which is essential to being able to read and write data into Firebase.

Next, we are going to see how to set up and work with Firebase, and then create a service provider that is responsible for handling all the communication from the application to the real-time database.

Once we have done that, we'll have our AngularDart application ready. We're close to completing the project—which is exciting!

Chapter 6 Firebase

Overview

There's one important aspect of our application's logic that is missing, and that's the service provider that is going to be responsible for interacting with Firebase. But before we write the code for the service provider, let's get started with Firebase and set it up.

Quick intro

Firebase is an amazing service from Google that provides a comprehensive app development platform encompassing authentication, hosting, storage, cloud functions, and a real-time database. For this application, we'll be using Firebase's real-time database.

Project setup

Let's get started with Firebase. Using your Google or Gmail account, navigate to the Firebase Console. Once you have logged in, you will see a screen similar to the following one.

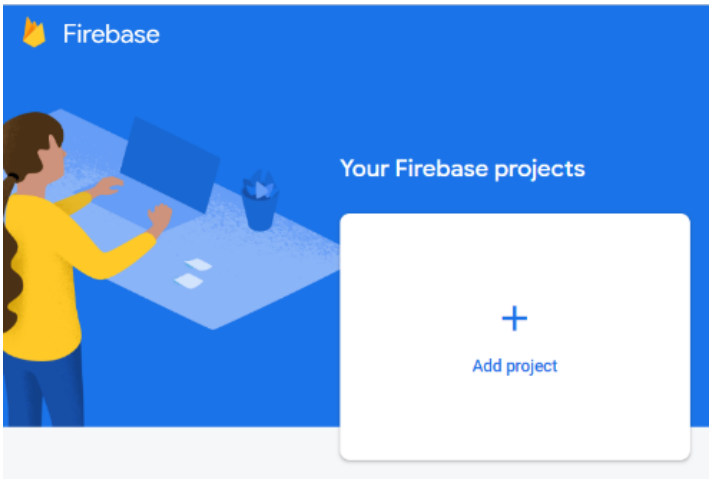

Figure 6-a: The Firebase Console

Click **Add project**, and we'll give our project a name. You should see a screen similar to the following one.

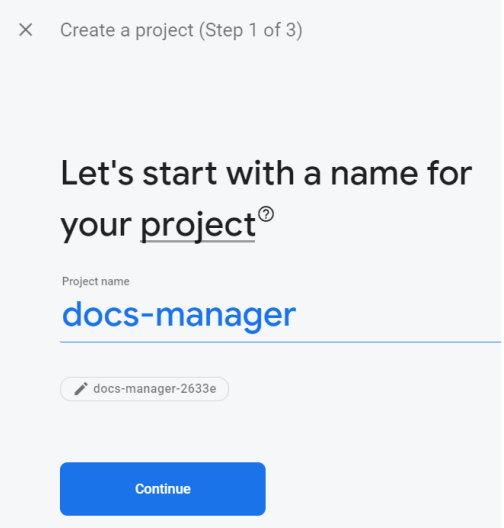

Figure 6-b: Creating a Firebase Project (Step 1 of 3)

Once you have given the project a name, click **Continue**. In the next step, you'll choose whether to enable Google Analytics for this project.

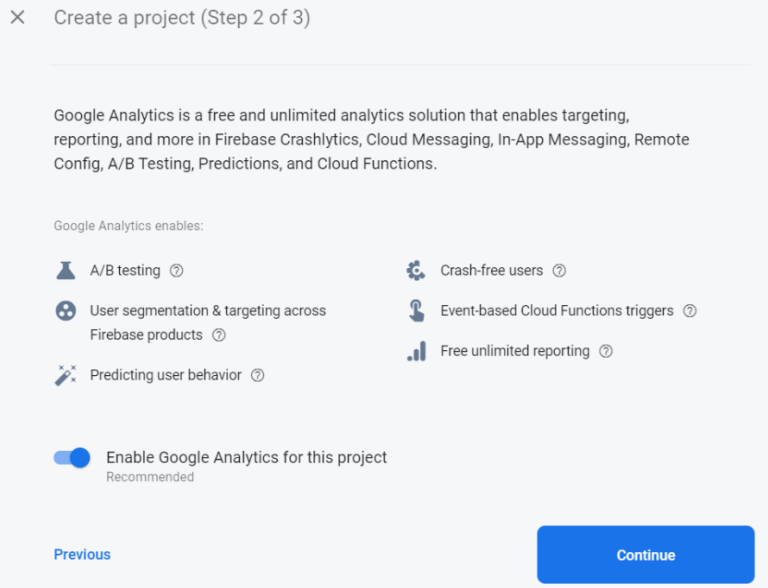

Figure 6-c: Creating a Firebase Project (Step 2 of 3)

For this project, it's not necessary to have Google Analytics enabled, as we will only be using the real-time database feature of Firebase. However, you are free to enable it if you would like to later deploy and host the application to Firebase hosting and be able to gather usage metrics.

Since it's always a good idea to be prepared for the future, let's click the **Enable Google Analytics for this project** option, and then click **Continue**.

Finally, let's select the **Analytics location** (I've set mine to the United States) and then click on **Create project**.

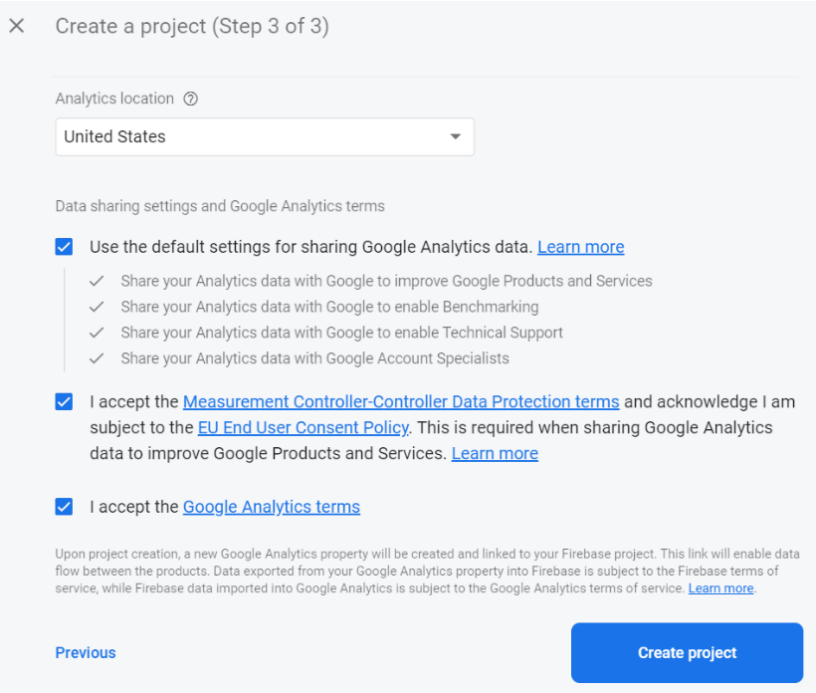

Figure 6-d: Creating a Firebase Project (Step 3 of 3)

Finally, the project will be ready to be used. We'll see the following confirmation.

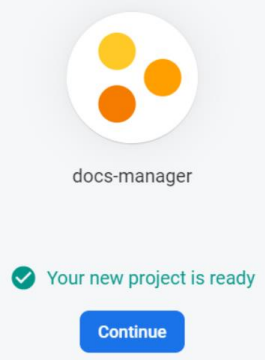

Figure 6-e: Firebase Project Created

Once you click **Continue**, you can start to work on our project. Let's do it.

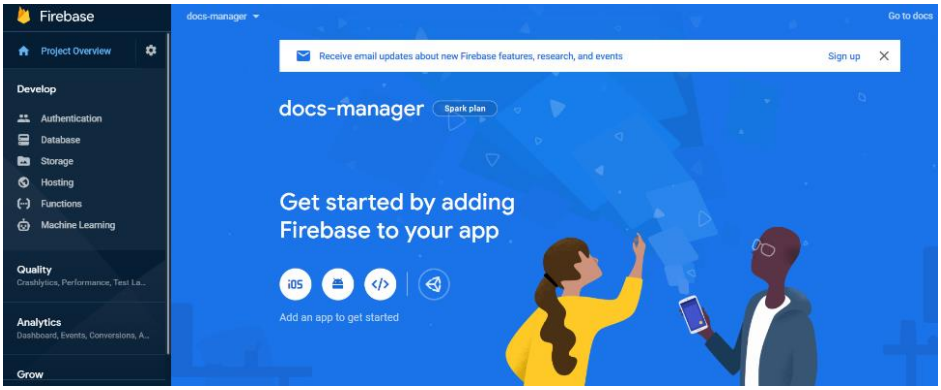

Figure 6-f: Firebase Project Starter Page

We have our Firebase project created, but to be able to use it, we need to add hosting—which we can later use if we want to deploy the application. To do that, click the **Project Overview** configuration icon, and then the **Project settings** option.

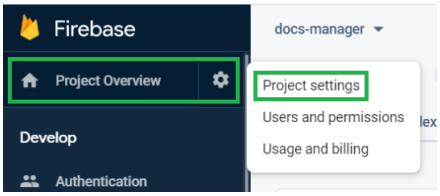

Figure 6-g: Project Overview Options

Once we have done that, a project settings page similar to the following one will be shown.

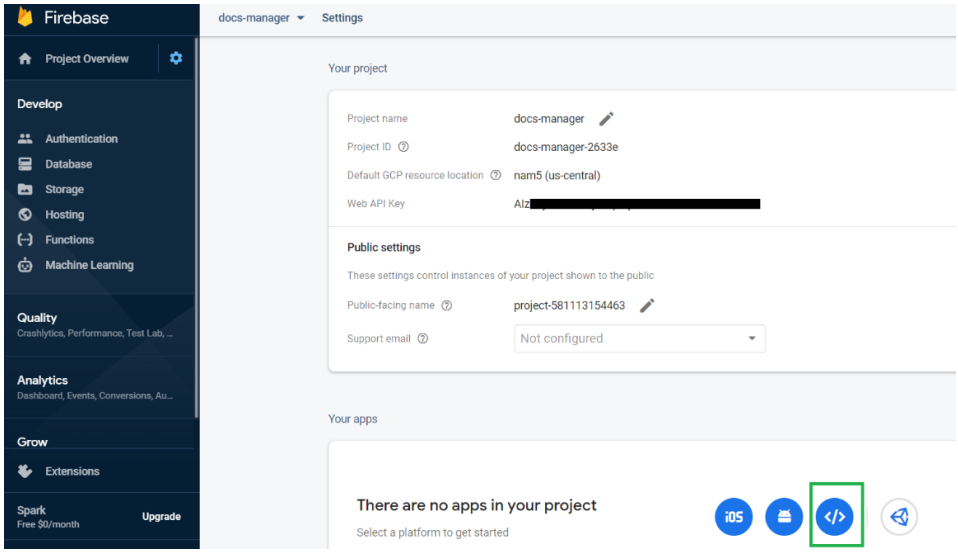

Figure 6-h: Firebase Project Overview

Web app

On this project overview page, in the bottom-right corner, click the icon highlighted in green (as seen in the preceding figure). By doing this, we'll add database capabilities to our Firebase web app project.

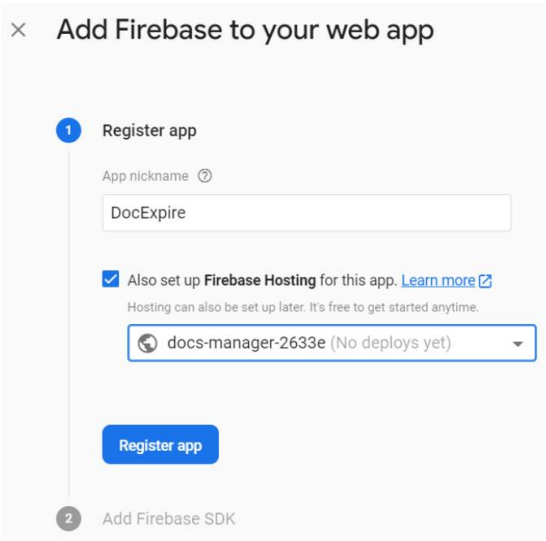

Figure 6-i: Add Firebase to your web app (Register app)

First, let's give our web app a name—**DocExpire**. Let's also set up **Firebase Hosting** and select our Firebase project. Then, click **Register app**.

Once we have done that, we'll be shown the scripts that our application will need. The required script(s) described in this step has already been included in the **index.html** file of our project—so we can skip this step. Let's click **Next**.

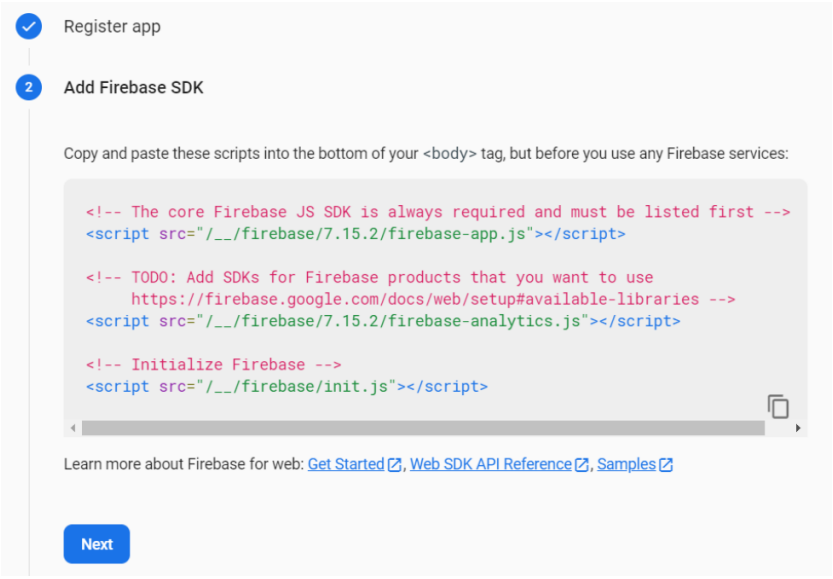

Figure 6-j: Add Firebase to your web app (Add Firebase SDK)

We'll be presented with a screen similar to the following one.

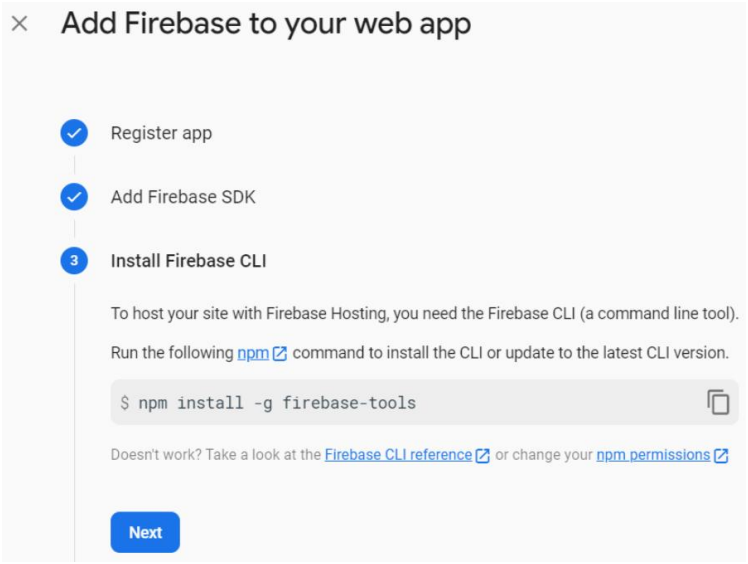

Figure 6-k: Add Firebase to your web app (Install Firebase CLI)

We can choose to install the Firebase CLI, which is necessary if we want to deploy the application later to Firebase hosting.

Installing the Firebase CLI is very easy to do. To do it, open the terminal or command line as an administrator and type in the **npm install -g firebase-tools** command.

To do that, you'll need to have the long-term support (LTS) version of Node.js installed. Here's how it looks when I run this command on my machine.

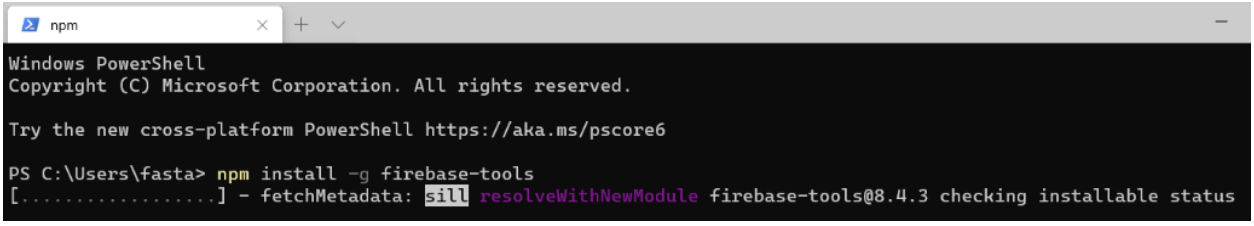

Figure 6-l: Add Firebase to your web app (Install Firebase CLI)

Once we have these tools installed, we can click **Next** on the **Add Firebase to your web app** screen, where we left it. This will bring us to the following screen.

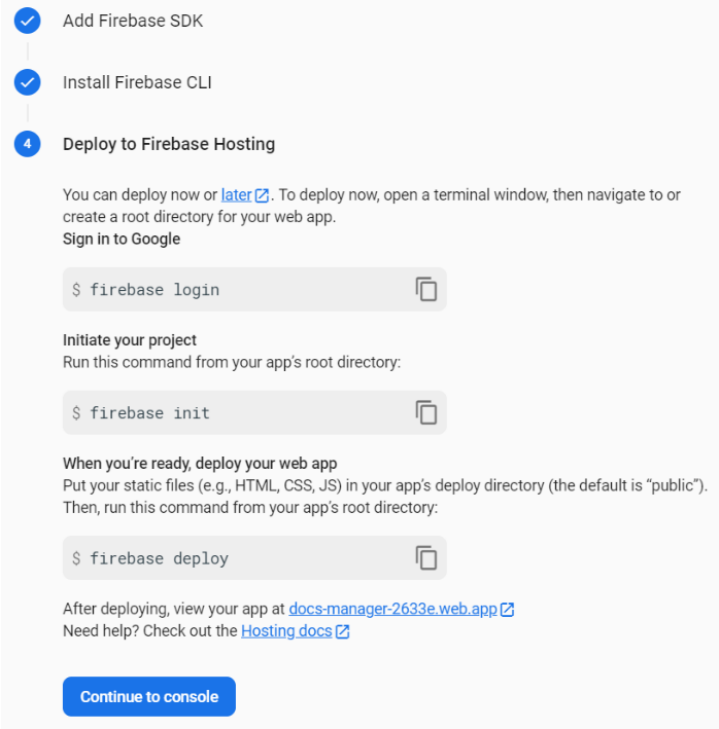

Figure 6-m: Add Firebase to your web app (Deploy to Firebase Hosting)

Here we can see the commands that are available for us to use if we want to deploy our app to Firebase Hosting. Let's click **Continue to console** to carry on.

Back at the Firebase console, let's click on the **Project Overview** icon, and then the **Project settings** option. Once there, scroll down to **Your apps**, highlighted with the green arrow in the following figure.

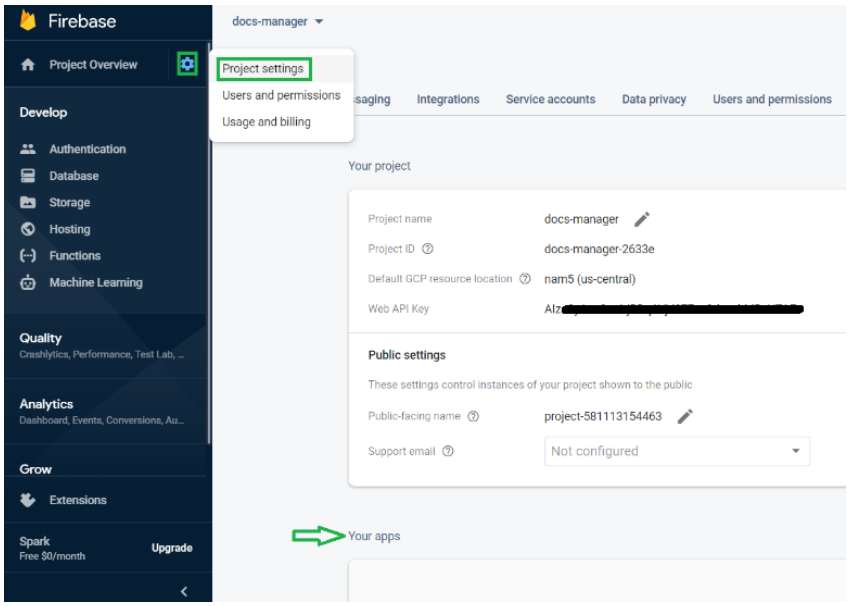

Figure 6-n: Firebase Project Settings (Your apps)

There, you should be able to find the application's configuration settings we will use within the service provider later to connect to the real-time database within Firebase. Copy those configuration settings to a text file, as we will need them later.

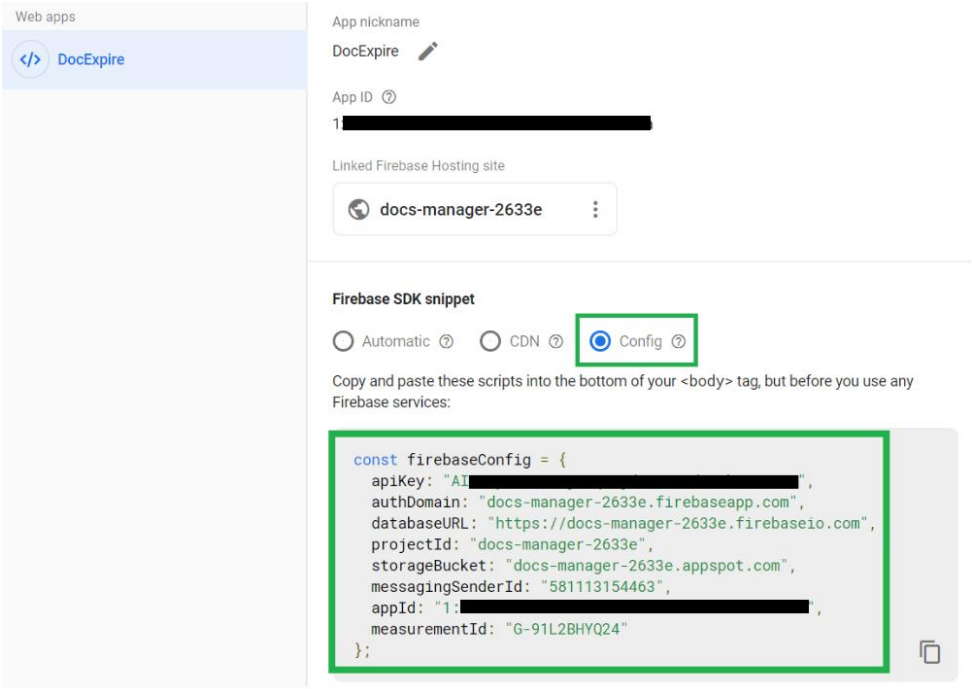

Figure 6-o: DocExpire Config Values in Firebase

Realtime database

Once you have copied these values, scroll up to the top of the page and click the **Database** option on the left-hand side of the screen. We are going to set up the real-time database.

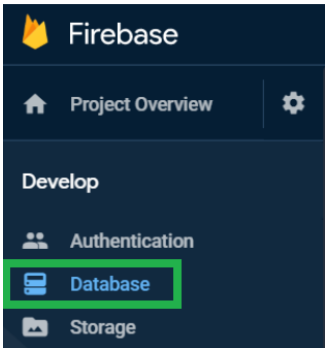

Figure 6-p: Firebase Console Menu (Database Option)

Once you have clicked this option, you will see the **Database** screen. There, select the **Realtime Database** option from the drop-down, and then click **Rules**.

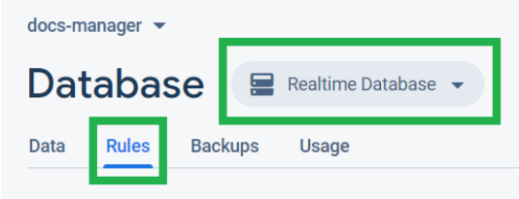

Figure 6-q: Database Screen

You should see a screen that looks similar to the following one.

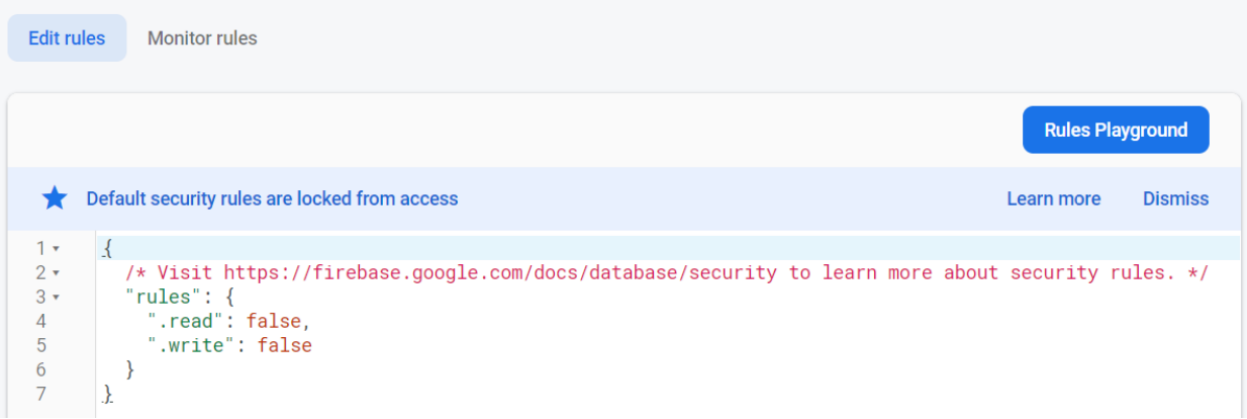

Figure 6-r: Database Rules (Default Values)

Here, click **Edit rules**. Let's change the value of the **read** and **write** properties from **false** to **true**, as shown in the following figure.

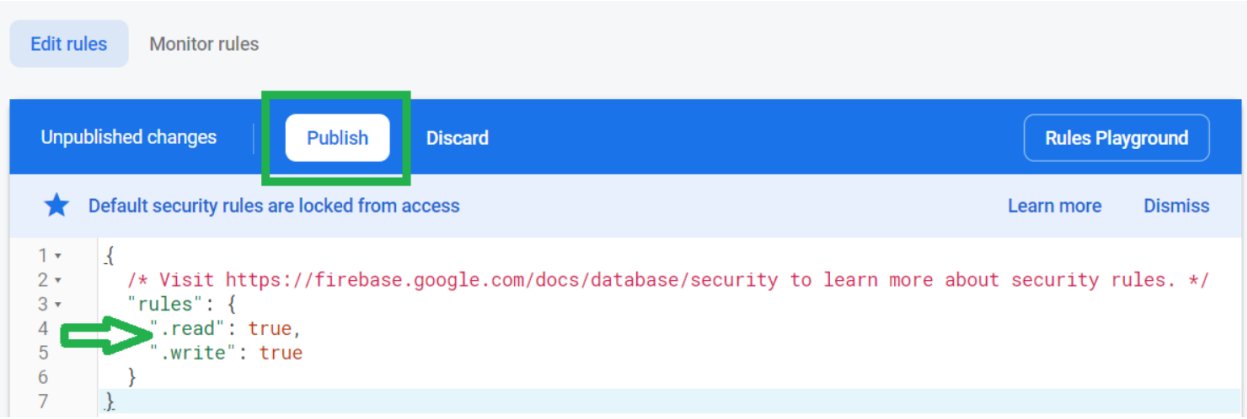

Figure 6-s: Database Rules (Edited Values)

Once you have edited the values, click **Publish**. Awesome—we have the Firebase real-time database ready.

Authorized domains

There's one last thing we need to set up with Firebase, and that's adding our local machine domain. As we are going to be running our application from our local machine, we need to tell Firebase to accept incoming requests from our local machine. This is an essential step for testing the application.

To do that, on the Firebase menu (left-hand side of the Firebase console screen), click **Authentication**.

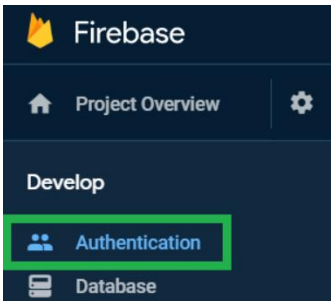

Figure 6-t: Firebase Console Menu (Authentication Option)

Once you're on the **Authentication** screen, click the **Sign-in method** tab.

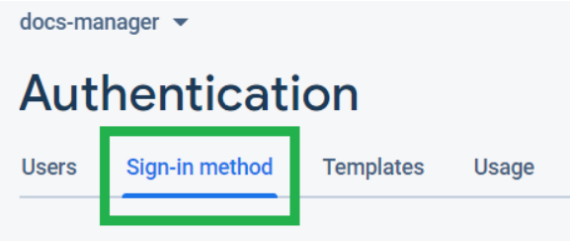

Figure 6-u: Authentication Screen

You should then see a screen similar to the following one. Under **Authorized domains**, click **Add domain**.

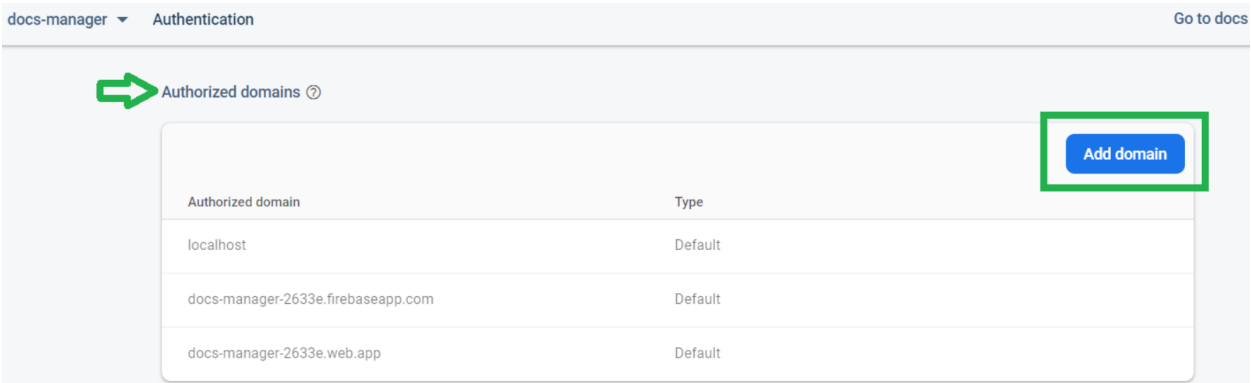

Figure 6-v: Authorized Domains

You will be asked to enter the domain for your local machine—which normally is **http://127.0.0.1**.

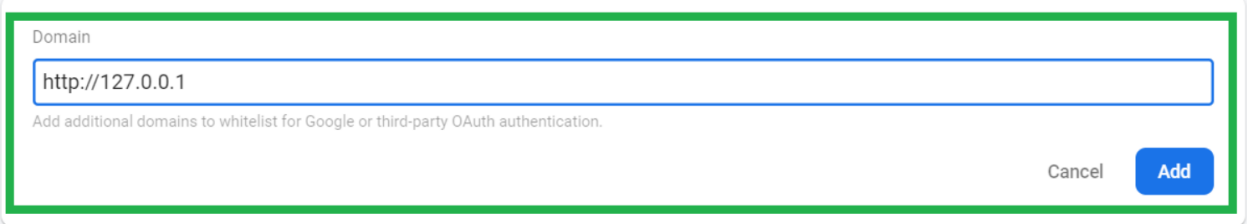

Figure 6-w: Entering the Local Machine Domain

Once you have entered the local machine domain, click **Add**.

That's it—we are good to go, and we can proceed by adding the app's service provider that will work with the Firebase real-time database.

Summary

Throughout this chapter, we have seen how to set up a Firebase project, a web app that can be hosted on Firebase, and a real-time database within Firebase. We also added our local machine domain as an authorized domain, so we can test our application locally.

We have one last step remaining—which is to write the logic for our service provider that is going to allow our application to interact with Firebase. This is what we are going to do next.

Chapter 7 Service Provider

Overview

We've reached the most exciting part of our journey—which is to bring our application to life. That's what the service provider is all about. The service provider is a layer of AngularDart logic that is going to use the model and allow the **de-base** component to interact with Firebase.

Service logic

Within VS Code, go to the **services** folder and open the **docs_service.dart** file we previously created. With the file open, copy and paste the code from the listing that follows.

Code Listing 7-a: docs_service.dart (Full Code)

```
import 'dart:async';
import 'package:angular/angular.dart';

import '../model/docs.dart';

// Firebase
import 'package:firebase/firebase.dart';

@Injectable()
class DocsService {
  DocsService() {
    // Firebase
    initializeApp(
      apiKey: '<< Your API key goes here >>',
      authDomain: 'docs-manager-2633e.firebaseapp.com',
      databaseURL: 'https://docs-manager-2633e.firebaseio.com',
      projectId: 'docs-manager-2633e',
      storageBucket: 'docs-manager-2633e.appspot.com');

    // Firebase
    _db = database();
    _ref = _db.ref('docs').ref;
  }

  // Firebase
  Database _db;
  DatabaseReference _ref;
```

```
  // Firebase
  Future<List<Doc>> getDocs() async {
    List<Doc> docs = [];
    final QueryEvent queryEvent = await _ref.once('value');
    final DataSnapshot snapshot = queryEvent.snapshot;
    final dataset = snapshot.val();

    if (dataset != null) {
      dataset.forEach((key, val) {
        var rec = val as Map<String, dynamic>;
        docs.add(Doc.fromMap(rec));
      });
    }

    return docs;
  }

  // Firebase
  Future addDoc (Doc dc) async {
    var r = await _ref.push(dc.asMap());
    return r.key;
  }

  // Firebase
  Future updateDoc (Doc dc) async {
    return await _ref.child(dc.ID).set(dc.asMap());
  }

  Future removeDoc (Doc dc) async {
    return await _ref.child(dc.ID).remove();
  }
}
```

The code for the service provider is very simple, but there are some details we need to be aware of.

Initialization

Remember those Firebase config settings that we previously copied? We need those now. So, before we do anything, let's make sure we set the Firebase config settings correctly. Let's have a look.

```
initializeApp(
```

```
  apiKey: '<< Your API key goes here >>',
  authDomain: 'docs-manager-2633e.firebaseapp.com',
  databaseURL: 'https://docs-manager-2633e.firebaseio.com',
  projectId: 'docs-manager-2633e',
```

```
 storageBucket: 'docs-manager-2633e.appspot.com');
```

From the config details that you copied, paste the **apiKey** value where it says **<< Your API key goes here >>**. In my case, the **authDomain** is **docs-manager-2633e.firebaseapp.com**, but in your case, it might be slightly different, so make sure you use your value (and not mine).

The same applies to the **databaseUrl**, **projectId**, and **storageBucket** properties—make sure you use your values, and not mine.

With the correct values configured, let's explore the rest of the code to understand what it does. Like with every other code, we start by importing the modules we need.

```
import 'dart:async';
import 'package:angular/angular.dart';
import '../model/docs.dart';
// Firebase
import 'package:firebase/firebase.dart';
```

We import the Dart asynchronous module because all Firebase operations are asynchronous, and we also import AngularDart.

We then import the app's model and the Firebase module—which was installed at the beginning of the project when the **pubspec.yaml** file was defined—containing the `firebase: ^7.3.0` dependency definition.

As you might recall, the Firebase dependency was installed using the **pub get** command—which is something we did in the first chapter.

Dependency injection

Then, we define the **DocsService** class and annotate it with the **@Injectable** attribute, which we can see as follows.

```
@Injectable()
class DocsService
```

The reason we add the **@Injectable** attribute to the **DocsService** class is that the service provider is "injected" into the **DocExpireBase** class (found within **de_base_component.dart**). This is known as dependency injection in AngularDart.

The code highlighted in bold in the following code shows how the service provider (**DocsService** class) gets injected into the **DocExpireBase** class (found within **de_base_component.dart**).

```
@Component(
  selector: 'de-base',
  templateUrl: 'de_base_component.html',
  directives: [coreDirectives, DocForm],
  providers: [ClassProvider(DocsService)]
)
class DocExpireBase implements OnInit
```

Constructor

Next, we have the constructor of the **DocsService** class, which, as you can see, invokes the **initializeApp** method that sets the Firebase config values.

Code Listing 7-b: docs_service.dart (DocsService Constructor)

```
DocsService() {
  // Firebase
  initializeApp(
    apiKey: '<< Your API key goes here >>',
    authDomain: 'docs-manager-2633e.firebaseapp.com',
    databaseURL: 'https://docs-manager-2633e.firebaseio.com',
    projectId: 'docs-manager-2633e',
    storageBucket: 'docs-manager-2633e.appspot.com');

  // Firebase
  _db = database();
  _ref = _db.ref('docs').ref;
}
```

The constructor of the **DocsService** class also creates the Firebase instance, which is done by calling the **database** method.

With the Firebase database instance created, we create and get a reference to the real-time database, which we call **docs**. We do this by calling **_db.ref('docs').ref**.

The Firebase instance and reference to the real-time database are declared as follows.

```
Database _db;
DatabaseReference _ref;
```

Retrieving documents

Next, we have the **getDocs** method, which is used for fetching the list of documents from the real-time database in Firebase. Let's have a look.

Code Listing 7-c: docs_service.dart (getDocs Method)

```
Future<List<Doc>> getDocs() async {
  List<Doc> docs = [];
  final QueryEvent queryEvent = await _ref.once('value');
  final DataSnapshot snapshot = queryEvent.snapshot;
  final dataset = snapshot.val();

  if (dataset != null) {
    dataset.forEach((key, val) {
      var rec = val as Map<String, dynamic>;
      docs.add(Doc.fromMap(rec));
    });
  }

  return docs;
}
```

First, we declare an empty list of documents (**`List<Doc> docs`**), which is going to contain the documents retrieved from Firebase.

Then, we declare a **`QueryEvent`** instance that returns the value of the reference to the real-time database (**`_ref.once('value')`**).

As this operation is asynchronous, the **`await`** keyword is used—this is why the **`getDocs`** method is marked with the **`async`** keyword.

Then, we define a **`DataSnapshot`** instance of the real-time database, which is retrieved by invoking **`queryEvent.snapshot`**.

Now that we have the snapshot of the real-time database, we need to get its full data content. To do that, we assign to the **`dataset`** variable the value returned by invoking the **`val`** method from the **`DataSnapshot`** instance.

If the **`dataset`** retrieved is not **`null`**, then for each key-value pair (for each JSON object—each representing a document), we create a record by casting the key-value pair value (**`val`**) as a **`Map<String, dynamic>`** object.

This **`Map<String, dynamic>`** object (**`rec`**) is then converted to a **`Doc`** instance, by invoking the **`Doc.fromMap`** method.

The resultant **`Doc`** instance is then added to the list of documents (**`docs`**). After all the JSON documents retrieved have been inspected, the list of documents (**`docs`**) is returned.

As you can see, the **`getDocs`** method returns a **`Future`**, which is a promise that the list of documents will be returned, once the asynchronous operation has finalized.

Adding, updating, and removing docs

Now that we know how to retrieve documents from Firebase and have the Firebase instance and reference to the real-time database ready, adding, updating, and removing documents is quite simple. Let's have a look.

Code Listing 7-d: docs_service.dart (Add, Update, and Remove Method)

```
Future addDoc (Doc dc) async {
  var r = await _ref.push(dc.asMap());
  return r.key;
}

Future updateDoc (Doc dc) async {
  return await _ref.child(dc.ID).set(dc.asMap());
}

Future removeDoc (Doc dc) async {
  return await _ref.child(dc.ID).remove();
}
```

Just like the **getDocs** method, the **addDoc**, **updateDoc**, and **removeDoc** methods also return a **Future** (promise) and are marked with the **async** keyword—as their corresponding Firebase operations are all asynchronous.

The **addDoc** method can insert a new document into the real-time database by invoking the **_ref.push** method. The **Doc** instance (**dc** object) is converted to JSON by using the **asMap** method.

Once the document has been added, the response's **key** (which represents the document's Firebase ID) is returned.

Updating a document is also very straightforward—this is what the **updateDoc** method does. This occurs when calling **_ref.child**, passing the ID of the document to update (**dc.ID**), and invoking the **set** method.

This is done by passing the updated **Doc** instance (**dc** object) as JSON using the **asMap** method.

Removing a document is almost identical to updating one. The removal of the document is done by the **removeDoc** method. The only difference with the update process is that the **remove** method is used, instead of **set**.

Alright, that's all there is to it—we've finalized the app's service provider and we are now ready to test our application and see how it works.

Running the App

Before executing the application, there's one final check we need to do. Within VS Code, open the **app_component.html** file and make sure it contains the following markup.

Code Listing 7-e: app_component.html

```
<de-base></de-base>
```

Executing the application locally on our machine is quite simple. All we need to do is open the built-in terminal within VS Code and run the **webdev serve** command—as we can see below.

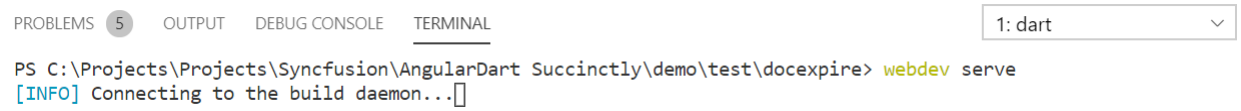

Figure 7-a: Executing the App (webdev serve)

This command will compile and build the application. Once the app has been built and is ready, we will see the following output.

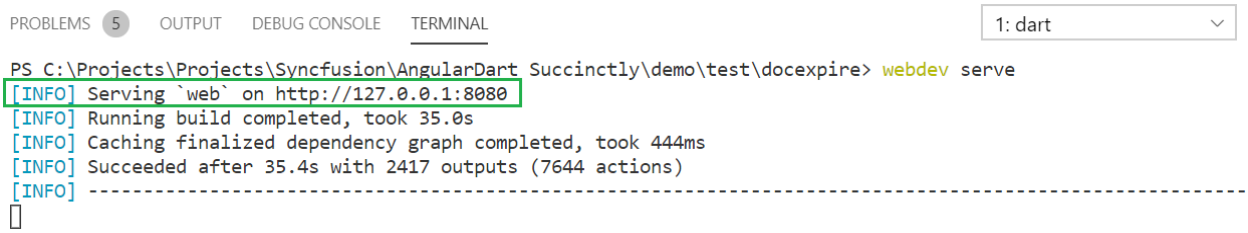

Figure 7-b: The App Running

Notice how on the built-in terminal output, the URL where the app is running locally is mentioned. So, open your browser and enter that URL.

You can also press **Ctrl** (on Windows) and click on the link—this will also open that URL on the browser. In my case, I already have data within the Firebase real-time database, so when I open the browser, I see the following.

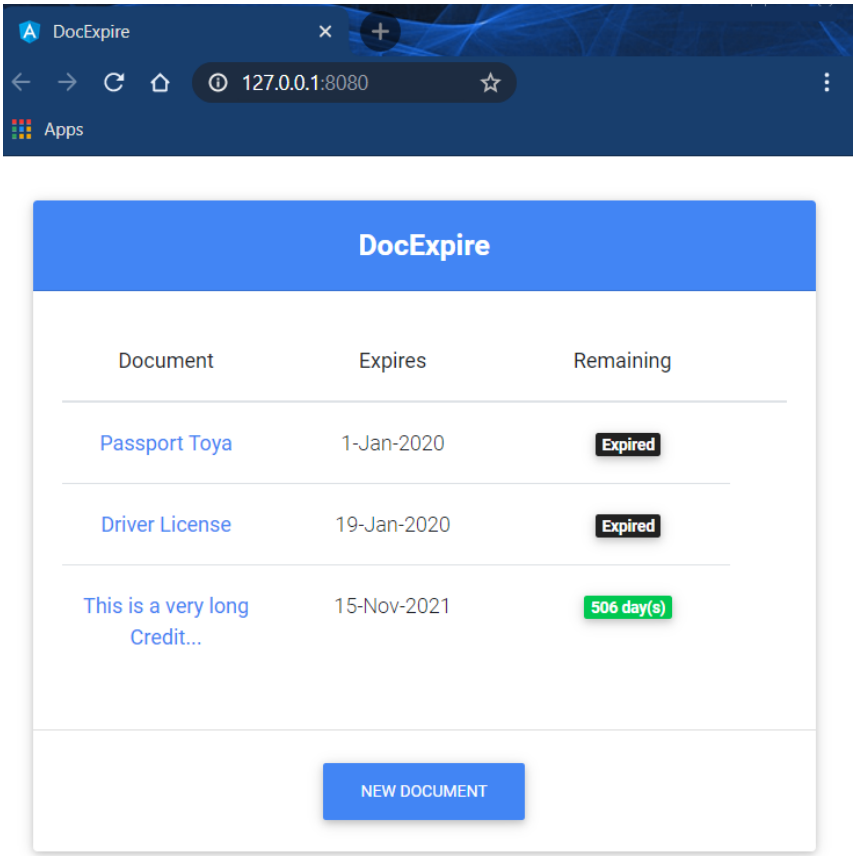

Figure 7-c: The App in the Browser (with Documents)

In your case, if you haven't added any data manually to the Firebase real-time database, you will see that there are no documents, so you can go ahead and add a new one.

So, in that case, you should see the following screen.

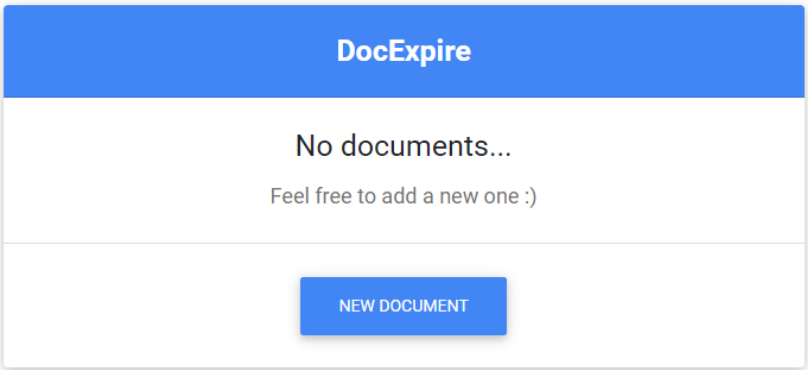

Figure 7-d: The App in the Browser (with No Documents)

Go ahead and add some documents. Once you have added some docs, go to the Firebase console and check out how the data looks within the real-time database. The following is an example of my data.

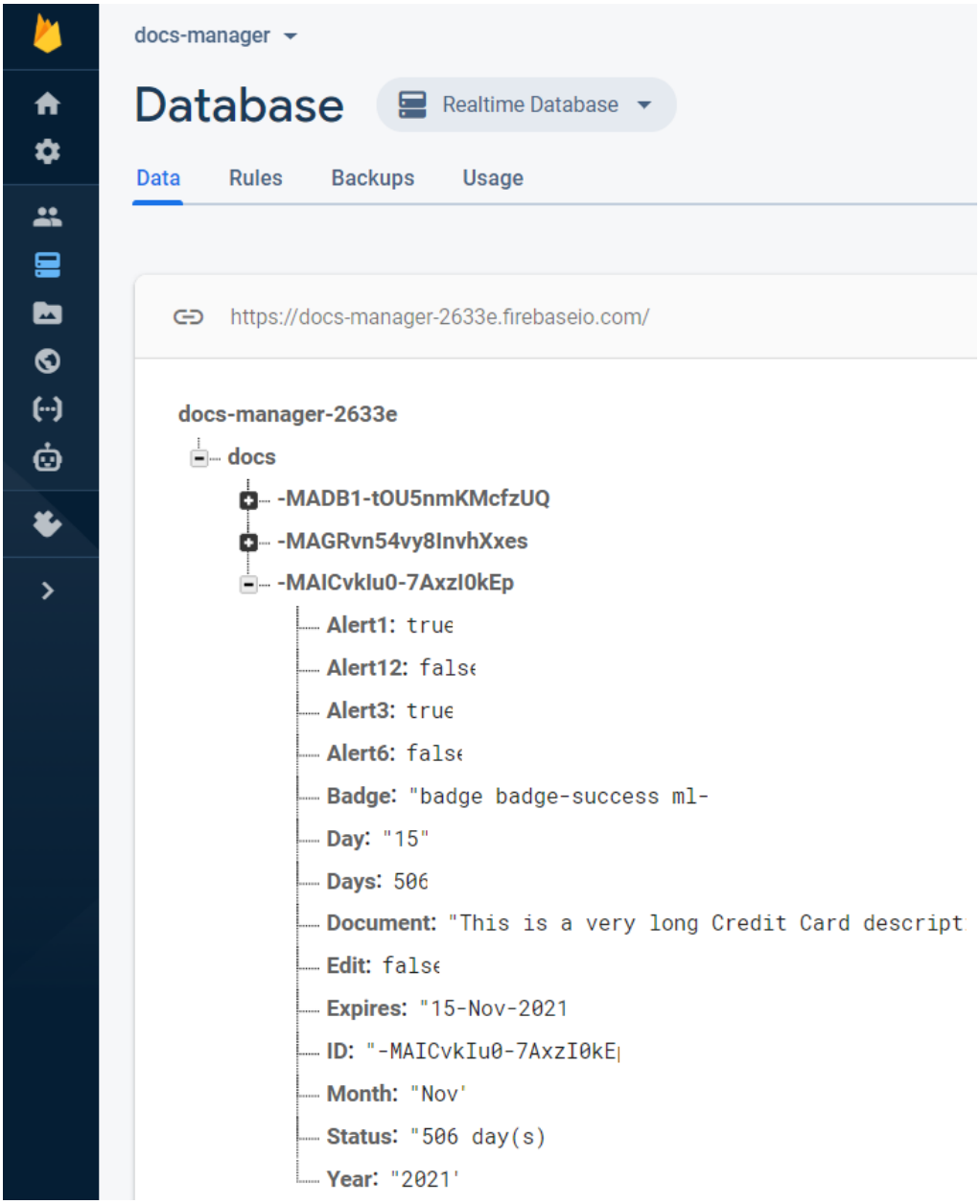

Figure 7-e: Realtime Database Explorer (Firebase)

As you can see from the preceding figure, there are three documents stored in the real-time database within Firebase, and the one expanded corresponds to the document that expires in 506 days (the last one on the app's document list).

Awesome—we now have a fully working AngularDart app. Well done!

Final thoughts

We've covered quite a lot of ground in this book and created a fully functional application using AngularDart. It was quite a journey. Nevertheless, there were a couple of things we didn't cover, which you can take on board and continue your journey by exploring this technology even further.

One point to look at would be deploying your application to Firebase Hosting, using the **firebase-tools** commands that were previously installed. The foundations are there, so this is relatively easy and straightforward to do.

Another potential challenge to take on is implementing some server-side code in Firebase that can query the real-time database and send emails or push notifications based on the alerts set for each document by checking its corresponding expiry date. This could be a cool feature to add.

So, I leave you with these two open items, for you to take further and see what your ingenuity and imagination can achieve.

Although AngularDart is not widely used outside of Google, it is a rock-solid web framework built on the pillars of giants, such as the Dart programming language and Angular. Google has used it to build and run many of its mission-critical web apps that generate much of its revenue—and I can see why.

I hope this book has given you some good insights into this amazing technology, and hopefully, the journey has been a fun one.

Thank you for reading and until next time, continue to explore and build amazing things with AngularDart. All the best.

Full source code

You can download the project's full source code in .rar format (which you can extract using the free 7-Zip utility program) from this URL.