Image by Aphinya Dechalert. Angular + Firebase Firestore CRUD tutorial

How to CRUD in Angular + Firebase Firestore

A complete example walk through with code snippets and full code repo

Aphinya Dechalert Mar 8, 2019 · 14 min read ★

UPDATE: This post is now also available on DottedSquirrel.com — where all my code related stories are available for free. Click here to go to the article.

you can increase the complexity of the queries but at the end of the day, it all boils down the these four actions.

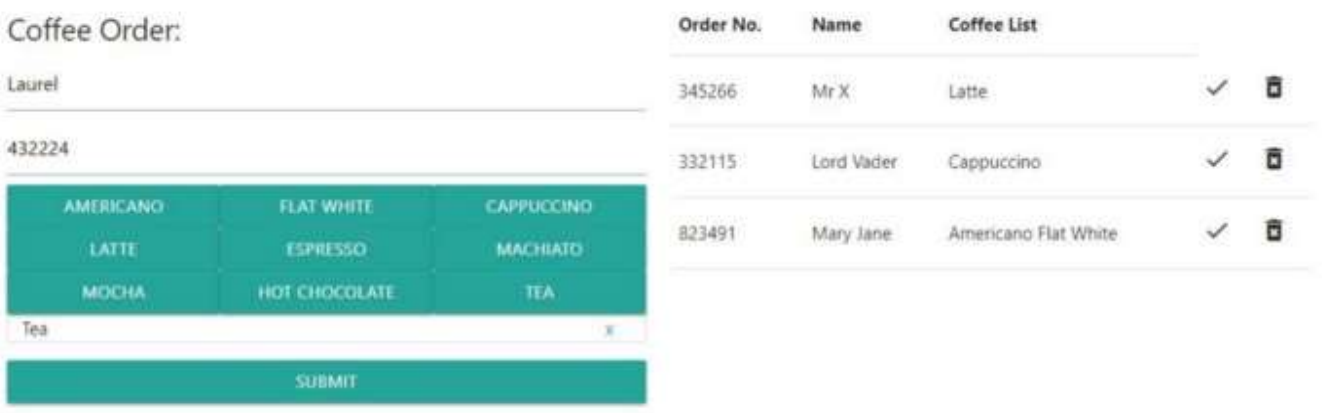
Firebase is a Google owned cloud based system that comes complete with API hooks, file storage space, auth system and hosting capabilities. It's a much underrated system that should be utilized more for prototyping and rapid application development.

If you want to boot up a progressive web app but don't have the backend experience of setting up servers, creating APIs and dealing with databases, then Firebase makes a fantastic option for front end developers who may feel isolated and bogged down by the massive hill of information they have to process to even get their app up and running.

Or if you're just short on time, Firebase can cut your development hours almost in half so you can focus on user experience and implementing those UI flows. It's also flexible enough to migrate out your front end application and use a different database if needed.

Here's a quick guide on how to implement CRUD actions with Angular and Firebase.

What we'll be making



(delete).

The purpose of this tutorial is to help you get started with Firebase Firestore and see how easy it is to connect to and get started on the Google owned service. This is not an advert for Google (I get no kickbacks from them for this) but merely an illustration of how Angular plays with the database.

The app we're going to make is not perfect. There are things missing like data validation and a million possible features that I can also add. But that's not the point. The point is to set up in Angular as quickly as possible and get it working with a live database.

So, enough of the intro — here's the code walk through.

The initial setup

Set up your Angular app via the Angular CLI. If you don't already have Angular CLI, you can find out more about it here.

In short, simply run these commands in your terminal in the directory where you want your Angular app to sit. Here are the commands and what they do.

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

Installs Angular CLI on your terminal if you don't already have it.

```
ng new name-of-app-here
```

This will create a new Angular project using the latest version of Angular available.

```
cd name-of-app-here
```

When you create a new project via Angular CLI, it will create a new project folder for you. `cd` will take you into that folder via the terminal.

```
ng serve
```

Setting up Angular

The structure

We're going to create 3 new parts in total for this Angular app — orders, order-list and shared/ordersService. The first two are components that will contain the interface for the app and the shared/orders service which will keep all the Firebase API calls together.

To create the necessary files, run the following commands:

```
ng g c orders
```

`g` stands for *generate* and `c` stands for *component*. The last part of the command is the name of your file, which for the case above is called *orders*. You can also use `ng generate component orders` to achieve the same effect.

Create another component for *order-list* using the following command.

```
ng g c order-list
```

And finally, for the service, use `s` instead of `c` like below:

```
ng g s shared/orders
```

This will create an `orders.service.ts` file in a folder named *shared*. Be sure to add `orders.service.ts` into `app.module.ts` because this is not done for you automatically like the components. You can do this via `import` and adding it to the `providers` list like so:

```
import { OrdersService } from "./shared/orders.service";
...
providers: [OrdersService]
...
```

than the default css. It's not necessary and you can use whatever CSS framework or your own custom CSS if you want. You can check out the details of Materialize CSS here.

We're also going to use Googles's material icons as buttons to mark the coffee order as completed or delete the order.

One way to implement this is to have the code below right above the `</head>` tag in your `index.html` file located in the `src` folder.

```
<!-- Compiled and minified Materialize CSS -->
<link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/css/ma
terialize.min.css">

<!-- Compiled and minified Materialize JavaScript -->
<script
src="https://cdnjs.cloudflare.com/ajax/libs/materialize/1.0.0/js/mate
rialize.min.js"></script>

<!-- Google Material Icons -->
<link href="https://fonts.googleapis.com/icon?family=Material+Icons"
rel="stylesheet" />
```

The initial Angular view code

In `app.component.html`, delete all the starter code. We'll be using our own content for this.

We'll hook into the component we've just created and display them on the screen by using the selectors `app-orders` and `app-order-list`. To do this, we write the code below:

```
<div class="container">
    <div class="row">
        <app-orders class="col s6"></app-orders>
        <app-order-list class="col s6"></app-order-list>
```

The `container`, `row`, `col` and `s6` classes are part of Materialize CSS grid system. All classes that you will see in the rest of this tutorial is from Materialize CSS, unless mentioned otherwise.

Setting up the form

In order to set up forms, import `ReactiveFormsModule` in `app.module.ts` and remember to import it in the `imports` array.

```
import { ReactiveFormsModule } from "@angular/forms";
```

Inside `orders.service.ts` import `FormControl` and `FormGroup` from `@angular/forms` and create a new form outside the `constructor` where you can set the properties of the form as below:

```
import { FormControl, FormGroup } from "@angular/forms";
...
...

export class OrdersService {
    constructor() {}
    form = new FormGroup({
        customerName: new FormControl(''),
        orderNumber: new FormControl(''),
        coffeeOrder: new FormControl(''),
        completed: new FormControl(false)
    })
}
```

We'll use these values to store in Firebase a little later in this tutorial.

Using the form inside a component

`import` the `OrdersService` class into your component so you can use the object inside your component and then create an object of the class inside the `constructor`.

```
constructor(private ordersService:OrdersService){}
```

Inside your `orders.component.html` use the `formGroup` directive to reference the form object created in `OrdersService`. Each `formControlName` references the names we used in the `formGroup` created inside the `OrdersService` class. This will allow the associated controller to use the variables typed into the form. See code below:

```
<form [formGroup]="this.ordersService.form">
    <input placeholder="Order Number"
           formControlName="orderNumber"
           type="text"
           class="input-field col s12">
    <input placeholder="Customer Name"
           formControlName="customerName"
           type="text"
           class="input-field col s12">
</form>
```

In the `orders.component.ts`, we're going to set up an array off coffee for looping through on our `orders.component.html`. In theory, you could also set this up inside your Firestore database, do a call to the collection and then use it. But for purposes of length, we're going to set it up as a local array. Inside your `OrdersComponent` class, set up the following array:

```
coffees = ["Americano", "Flat White", "Cappuccino", "Latte",
"Espresso", "Machiato", "Mocha", "Hot Chocolate", "Tea"];
```

In your `orders.component.html` and inside your `<form>` tags, loop through it using `*ngFor` with an `(click)` action handler to add new coffees to your order. We're going to display the order list right below with the ability to remove individual coffee as following:

```
        (click)="addCoffee(coffee)">
{{coffee}}
</button>

<ul class="collection">
    <li *ngFor="let coffee of coffeeOrder">
        <span class="col s11"> {{ coffee }} </span>
        <a class="col s1" (click)="removeCoffee(coffee)">x</a>
    </li>
</ul>
```

Inside `orders.component` you create an empty array to house the coffee order, use the function `addCoffee()` to add new coffees, and `removeCoffee()` to remove a beverage from your order list.

```
coffeeOrder = [];

addCoffee = coffee => this.coffeeOrder.push(coffee);

removeCoffee = coffee => {
    let index = this.coffeeOrder.indexOf(coffee);
    if (index > -1) this.coffeeOrder.splice(index, 1);
};
```

Handling form submission

Add a *Submit Order* input inside and at the bottom of the `<form>` tags and add the `onSubmit()` to a click handler like below:

```
<form [formGroup]="this.ordersService.form" (ngSubmit)="onSubmit()">
...
  <button
       class="waves-effect waves-light btn col s12"
       (click)="onSubmit()">
  Submit
  </button>
</form>
```

hooking up to your Firebase database.

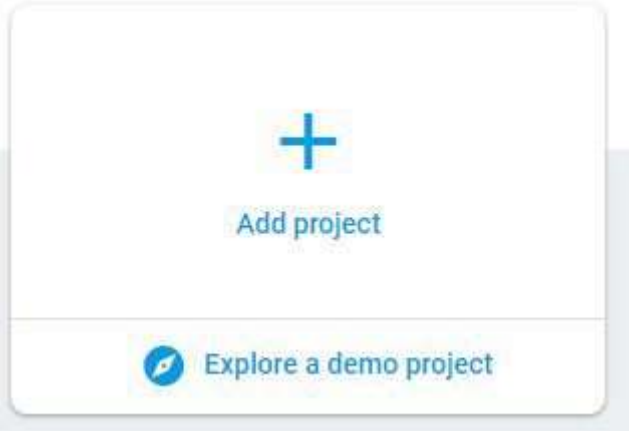
Setting up the listing component

Now we just need a space to display our coffee orders from the database. For now, we're just going to set up the scaffold html.

Navigate to your `order-list.component.html` and create a table with 3 headings and 5 data cells. The first 3 data cells will hold the values pulled from the database and final 2 will hold extra functionality that will allow you to mark the order as complete or delete the order.

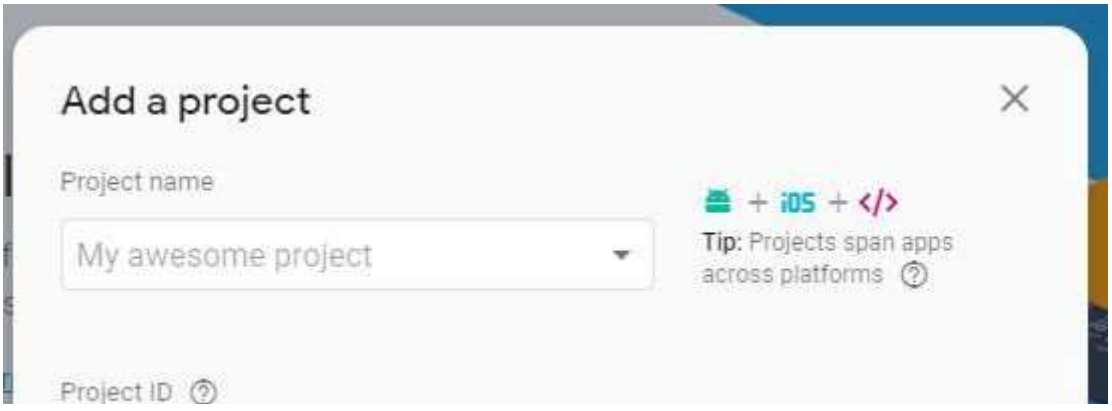
Setting up Firebase

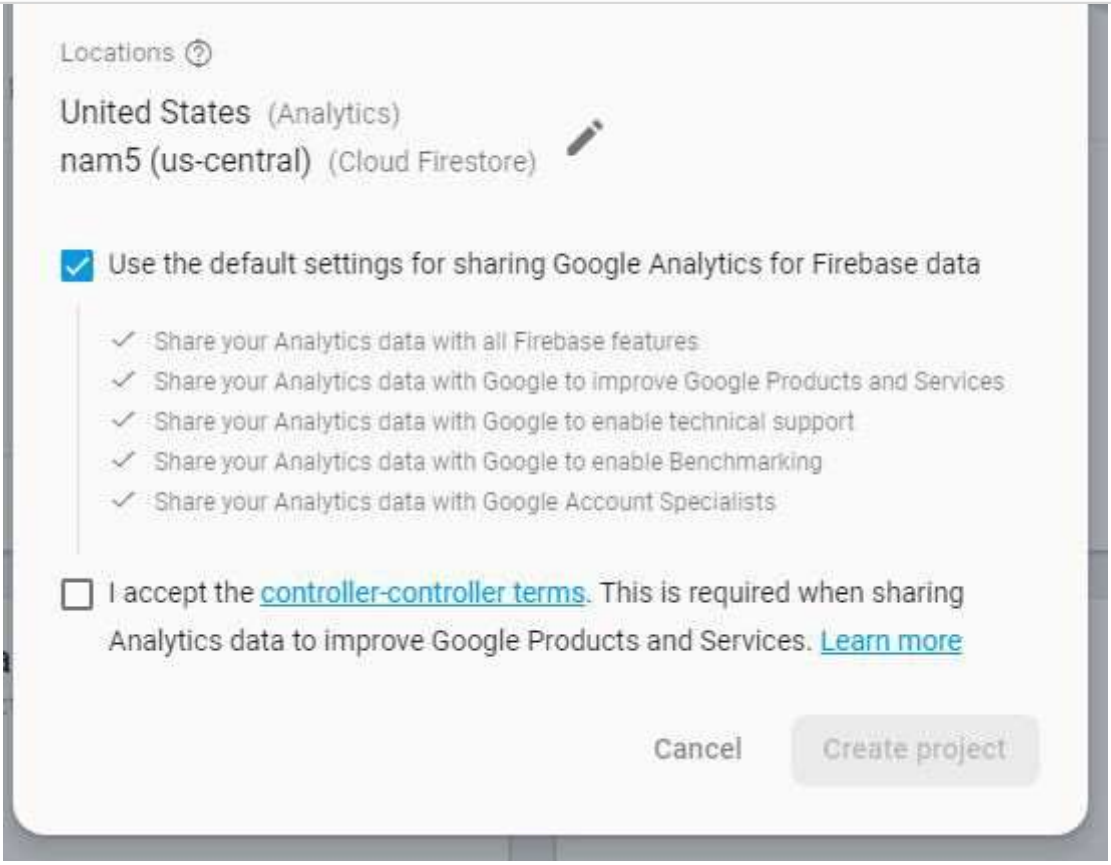
Go to your Firebase console and add a new project.



Click on 'Add Project' to create a new project.

Add a project name, accept the terms and conditions and click on create project.

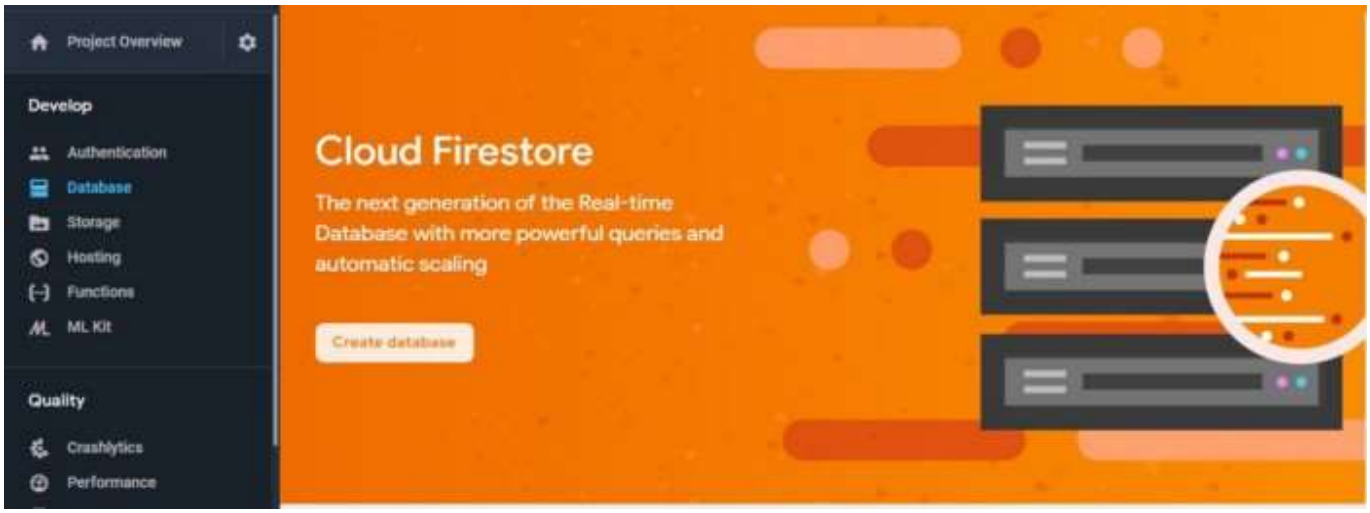




Give your project a name and accept the controller-controller terms in order to create a project.

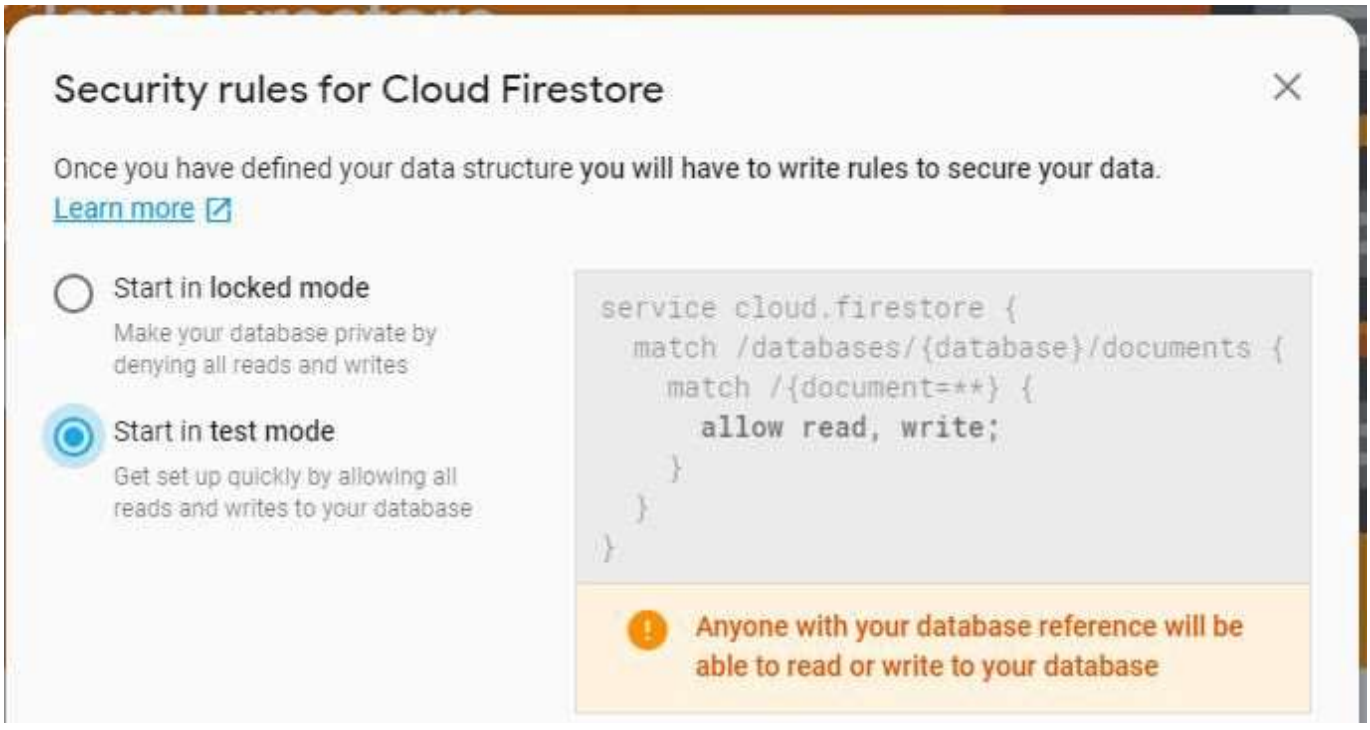
When your Firebase project has been set up, you will see something like this.

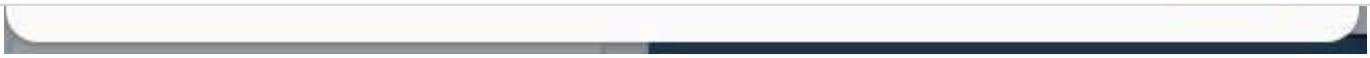
Create a database by selecting *Database* on the side panel (located under *Develop*) and then click on *Create Database* under the *Cloud Firestore* banner.



Click on 'Create database'

Select *start in* ***test mode*** for security rules. You can modify it later.





Remember to keep your credentials on your local machine only for security purposes.

You will get an empty Firestore database like this.

Connecting Firebase Firestore to Angular

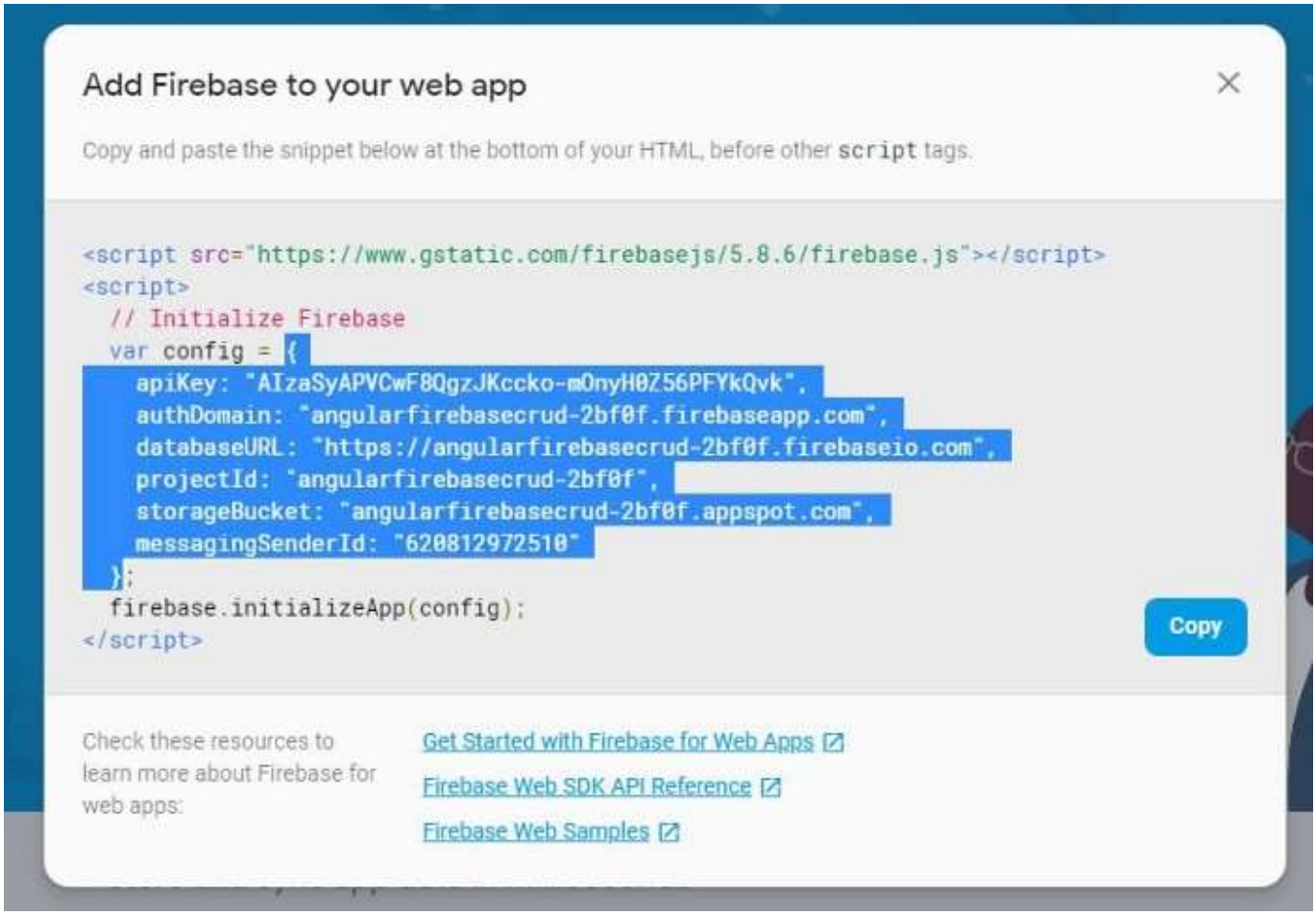
Installing the connectors

To connect your Angular app to Firebase, you're going to need install `firebase` and `@angular/fire` packages. This will give you access to `AngularFireModule` and `AngularFirestoreModule`. Use the following command in your terminal to install them.

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

Implementing the connectors

Go back to your Firebase web console and grab the config details to use in your Angular app. This is located on your *Project Overview* page. It looks something like this:



You'll need to use your own credentials. The above config won't work for you.

Copy the highlighted section of the code, navigate to your `environment.ts` file (located inside `environments` folder) and paste the config details inside the `environment` object as `firebaseConfig`. See below for example:

```
    firebaseConfig: {
      apiKey: "xxx",
      authDomain: "xxxx.firebaseapp.com",
      databaseURL: "https://xxxx.firebaseio.com",
      projectId: "xxxx",
      storageBucket: "xxxx.appspot.com",
      messagingSenderId: "xxxxx"
    }
};
```

Import the packages you installed earlier and the `environment.ts` file into your `app.module.ts`.You'll need to initialize the `AngularFireModule` with the `firebaseConfig` that you've just done the set up for. See example below:

```
import { environment } from "src/environments/environment";
import { AngularFireModule } from "@angular/fire";
import { AngularFirestoreModule } from "@angular/fire/firestore";
...
@NgModule({
...
imports: [
...
    AngularFireModule.initializeApp(environment.firebaseConfig),
    AngularFirestoreModule
  ]
...
})
```

Setting up Firestore in a service

Import `AngularFirestore` into your `orders.service.ts` file and declare it in the constructor so your service knows about it.

```
...
import { AngularFirestore } from '@angular/fire/firestore';
...
export class OrdersService {
constructor(   private firestore: AngularFirestore   ) {}
...
}
```

C is for Create

In order to create a new record in your brand spanking new Firestore database, you're going to need to call `.add()`. We're going to do this inside the `orders.service.ts` file.

To do this, you'll need to specify the `collection` name and what data you want to push to the database. In our case, it's `coffeeOrders`.

The example code below uses a promise to return the Firebase call and then lets you decide what you want to do after it's all done.

```
...
createCoffeeOrder(data) {
    return new Promise<any>((resolve, reject) =>{
        this.firestore
            .collection("coffeeOrders")
            .add(data)
            .then(res => {}, err => reject(err));
    });
}
...
```

To call this function in your component, navigate to `orders.component.ts` and inside the `onSubmit()` function and action handler, make a call the `createCoffeeOrder()` through `ordersService`.

The example below does some processing of the data by mapping the `coffeeOrder` array to the form value `coffeeOrder`. I've also created a `data` variable for destructuring purposes (along with not having to pass in a massively long name into `createCoffeeOrder()`).

```
...
  onSubmit() {
      this.ordersService.form.value.coffeeOrder = this.coffeeOrder;
      let data = this.ordersService.form.value;
```

```
            /*do something here....
            maybe clear the form or give a success message*/
        });
  }
...
```

And viola! You've now created a record from your localhost Angular app to your Firestore database.

R is for Read

In order to display your coffee orders data, you're going to need a read function in your `orders.service.ts`. The following code example will give you all the values stored in your `coffeeOrders` collection.

```
...
  getCoffeeOrders() {
    return
     this.firestore.collection("coffeeOrders").snapshotChanges();
  }
...
```

To use this function, you'll need to call it from your `orders-list.component.ts`. You can do this by importing the `OrdersService` into the file and initialize it in in the `constructor`.

```
...
import { OrdersService } from "../shared/orders.service";
...

export class OrderListComponent implements OnInit {
    constructor(private ordersService:OrdersService){}
...
}
```

returned results from your database via `subscribe()`. We will use this to iterate over and display in `order-list.component.html`

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

  coffeeOrders;

   getCoffeeOrders = () =>
      this.ordersService
      .getCoffeeOrders()
      .subscribe(res =>(this.coffeeOrders = res));

...
```

To use `coffeeOrders` in your `order-list.component.html`, use `*ngFor` to loop through the returned array. You also need to do a little of inception and do another loop for the `coffeeOrder` part to get your list of coffees for each customer. There are more efficient ways to do this but for this tutorial, see the example code below:

```
...
<tbody>
<tr *ngFor="let order of coffeeOrders">
   <td>{{ order.payload.doc.data().orderNumber }}</td>
   <td>{{ order.payload.doc.data().customerName }}</td>
   <td><span
        *ngFor="let coffee of order.payload.doc.data().coffeeOrder">
       {{ coffee }}
      </span>
   </td>
</tr>
</tbody>
...
```

And there you have it. You've now hooked up read capabilities to your Angular application.

do something to the line item based on the change. Because `snapshot()` keeps track of any changes that happens, you don't need to do any tracking or polling to the database.

We're going to create another data cell in our table with a 'check' icon from Google Materialize Icons and hook it up to a `(click)` event that will call the function `markCompleted()` in your `order-list.component.ts`. We're also going to pass in the particular order for this loop.

We're going to set a `[hidden]` attribute with the `completed` value to the data cell so that it can dynamically determine if we want to have the 'check' icon on display. We've originally set this value as `false` when we first created the form in `orders.service.ts`.

```
...
 <td [hidden]="order.payload.doc.data().completed"
     (click)="markCompleted(order)">
    <i class="material-icons">check</i>
</td>
...
```

Inside `order-list.component.ts`, create the function `markCompleted()` that uses the injected `data` to pass to a function called `updateCoffeeOrder()` in `orders.service.ts`.

```
...
  markCompleted = data =>
    this.ordersService.updateCoffeeOrder(data);
...
```

Inside `orders.service.ts` create the handling function. This function will connect and call your Firestore database based the selected collection and document id. We already know that our collection is called `coffeeOrders` and we can find the document id based on the parameters passed in from the component function call.

means that you only update the value-key pair passed in rather than replacing the entire document with what you passed in.

```
...
updateCoffeeOrder(data) {
   return
       this.firestore
       .collection("coffeeOrders")
       .doc(data.payload.doc.id)
       .set({ completed: true }, { merge: true });
}
...
```

Now when you click on the 'check' icon in your view, it will update your database and disappear because your `[hidden]` attribute is now set to `true`.

D is for Delete

For the final action, we're going to set everything up in a similar fashion as the update process — but instead of updating the record, we're going to delete it.

Set up another data cell with a click event handler that calls a `deleteOrder()` function. We're going to pass in the instance of the `order` data in the loop when the 'delete_forever' icon gets clicked. You'll need this for the document id. See code below for example:

```
...
 <td [hidden]="order.payload.doc.data().completed"
     (click)="deleteOrder(order)">
    <i class="material-icons">delete_forever</i>
</td>
...
```

Inside your `order-list.component.ts` create the associated function that calls a `deleteCoffeeOrder()` function inside your `orders.service.ts`.

```
...
```

Inside your `orders.service.ts` file, create the `deleteCoffeeOrder()` function and use the `data` injected to figure out what the document id is. Like update, you need to know both the collection name and the document id to correctly identify which record you want to delete. Use `.delete()` to tell Firestore that you want to delete the record.

```
...
deleteCoffeeOrder(data) {
   return
       this.firestore
       .collection("coffeeOrders")
       .doc(data.payload.doc.id)
       .delete();
}
...
```

Now when you click on the 'delete_forever' icon, your Angular app will fire off and tell Firestore to delete the specific record. Your record will vanish from the view when the specified document is deleted.

Final Application

You can find the Github repository for the entire working project here. You'll need to create your own Firebase database and hook it up yourself by updating the config files. The code itself is not perfect but I've kept it very minimal so you can see how Firestore works in an Angular app without having to go through a jungle of code.

I've tried to condense the tutorial down as much as possible without missing out any details. It was a hard mix but as you can see above, a lot of it is mostly the Angular code and not much Firestore. Firebase Firestore can do much more complex queries but for demonstration purposes, I've kept it simple. In theory, if your data structure remains the same, you can swap out Firestore and put in different set of API connections with minimal refactoring required.

projects without the need to create an entire backend and server to support it. Cost wise, it's not too bad and test projects are free to boot up (and they don't make you enter your credit card details until you're ready to switch plans). Overall, it took more time to create the form and the interface surrounding the app than the actual act of connecting the application up to Firestore. I haven't added form validation to this tutorial due to length but will create another post about it in the future.

I hope this walk through tutorial has been helpful to you or at least make a good reference point. I'll be doing a React equivalent in the next few days so stay tuned if you're interested.

Lets stay connected and join my weekly awesome web digest newsletter list. Thank you for reading.❤

Aphinya