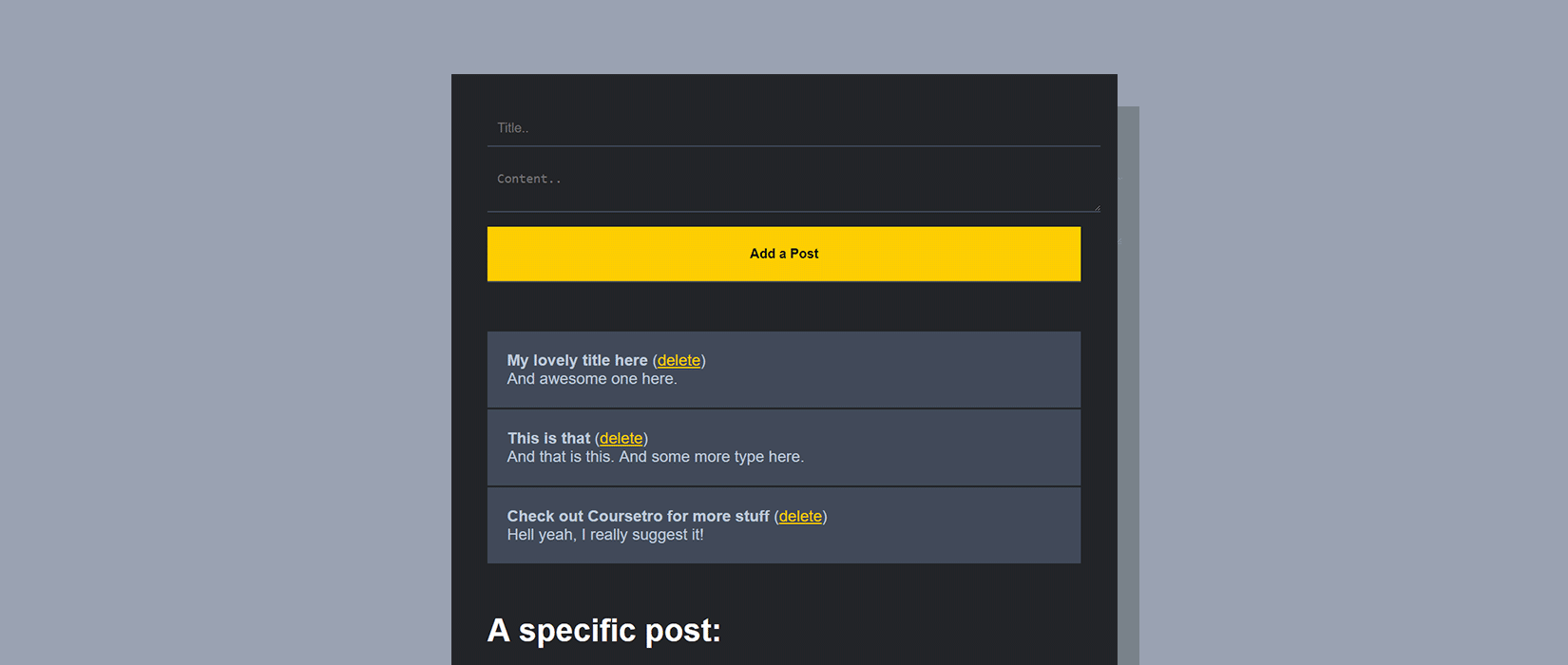
**Use Angular with Google's Cloud Firestore - Tutorial**

BY GARY SIMON - OCT 14, 2017



Firebase was Google's original realtime database, but there were some obvious shortcomings. Fortunately, they've released a new database called [**Cloud Firestore**](https://firebase.google.com/docs/firestore/).

Firestore is realtime, but offers several key advantages over the Firebase realtime database:

* **Structure**  
  The Firebase realtime db returned a large JSON tree. Firestore stores your data in objects called documents, which are grouped into collections. Within these collections, you can have more collections called subcollections up to 100 levels deep.
* **Querying**  
  This is the best part of Firestore. You can query for documents without having to retrieve all of the other data in other subcollections. This means you receive a faster response. You can also use where **clauses**, and even multiple where clauses if you add an index.
* **Scaling**  
  If your dataset in the Firebase realtime db got large, scaling could become an issue. With Firestore's new hierarchical structure, this is no longer an issue. Your queries will remain the same speed regardless of the size of your stored data.

There are other benefits, like an improved pricing model and fetching data just one time is also easier.

In this tutorial though, we're going to focus on creating a simple Angular app that uses the AngularFire5 library to read and write from Cloud Firestore -- so let's get started!

**[[](https://coursetro.com/courses/19/Learn-Angular-5-from-Scratch---Angular-5-Tutorial?utm_source=in_article&utm_campaign=article&utm_medium=Learn+Angular+5+from+Scra)](https://coursetro.com/courses/19/Learn-Angular-5-from-Scratch---Angular-5-Tutorial?utm_source=in_article&utm_campaign=article&utm_medium=Learn+Angular+5+from+Scra)**

**[Learn Angular 5 from Scratch - Angular 5 Tutorial](https://coursetro.com/courses/19/Learn-Angular-5-from-Scratch---Angular-5-Tutorial?utm_source=in_article&utm_campaign=article&utm_medium=Learn+Angular+5+from+Scra)**

**[[](https://coursetro.com/courses/18/Build-a-Beautiful-CryptoCurrency-App-using-Ionic-3?utm_source=in_article&utm_campaign=article&utm_medium=Build+a+Beautiful+CryptoC)](https://coursetro.com/courses/18/Build-a-Beautiful-CryptoCurrency-App-using-Ionic-3?utm_source=in_article&utm_campaign=article&utm_medium=Build+a+Beautiful+CryptoC)**

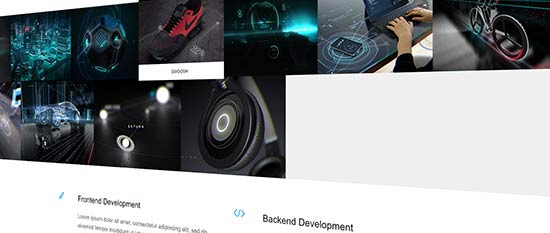
**[Build a Beautiful CryptoCurrency App using Ionic 3](https://coursetro.com/courses/18/Build-a-Beautiful-CryptoCurrency-App-using-Ionic-3?utm_source=in_article&utm_campaign=article&utm_medium=Build+a+Beautiful+CryptoC)**

**[[](https://coursetro.com/courses/13/Create-a-MEAN-App-Called-CodePost---Full-Stack?utm_source=in_article&utm_campaign=article&utm_medium=Create+a+MEAN+App+Called+)](https://coursetro.com/courses/13/Create-a-MEAN-App-Called-CodePost---Full-Stack?utm_source=in_article&utm_campaign=article&utm_medium=Create+a+MEAN+App+Called+)**

**[Create a MEAN App Called CodePost - Full Stack](https://coursetro.com/courses/13/Create-a-MEAN-App-Called-CodePost---Full-Stack?utm_source=in_article&utm_campaign=article&utm_medium=Create+a+MEAN+App+Called+)**

**[[](https://coursetro.com/courses/12/Learn-Angular-4-from-Scratch?utm_source=in_article&utm_campaign=article&utm_medium=Learn+Angular+4+from+Scra)](https://coursetro.com/courses/12/Learn-Angular-4-from-Scratch?utm_source=in_article&utm_campaign=article&utm_medium=Learn+Angular+4+from+Scra)**

**[Learn Angular 4 from Scratch](https://coursetro.com/courses/12/Learn-Angular-4-from-Scratch?utm_source=in_article&utm_campaign=article&utm_medium=Learn+Angular+4+from+Scra)**

**[[](https://coursetro.com/courses/10/Create-a-Personal-Portfolio-using-Angular-2-&-Behance?utm_source=in_article&utm_campaign=article&utm_medium=Create+a+Personal+Portfol)](https://coursetro.com/courses/10/Create-a-Personal-Portfolio-using-Angular-2-&-Behance?utm_source=in_article&utm_campaign=article&utm_medium=Create+a+Personal+Portfol)**

**[Create a Personal Portfolio using Angular 2 & Behance](https://coursetro.com/courses/10/Create-a-Personal-Portfolio-using-Angular-2-&-Behance?utm_source=in_article&utm_campaign=article&utm_medium=Create+a+Personal+Portfol)**

**If you prefer watching a video..**

Be sure to Subscribe to the **[Official Coursetro Youtube Channel](http://youtube.com/user/designcourse)** for more videos.

**Prerequisites**

As always, you will need **[Nodejs](http://nodejs.org/)** with NPM before you proceed. You can check if you have it by running the following commands in your console:

> node -v

> npm -v

If you have them installed, you can use npm to install the Angular CLI (Command Line Interface) tool, which will help us start the Angular project quickly.

To install it, run:

> npm install @angular/cli@latest -g

Once installed, you're ready to move onto the next step.

**Starting the Project**

We're going to use the Angular CLI to start a new project:

> ng new firestore

Once finished, hop into the new firestore directory with **cd firestore** and run the following commands to install the following packages:

> npm install angularfire2 firebase --save

This will install the 2 necessary packages for interacting with Firebase and Firestore.

Next, run the following command to launch the project in the browser:

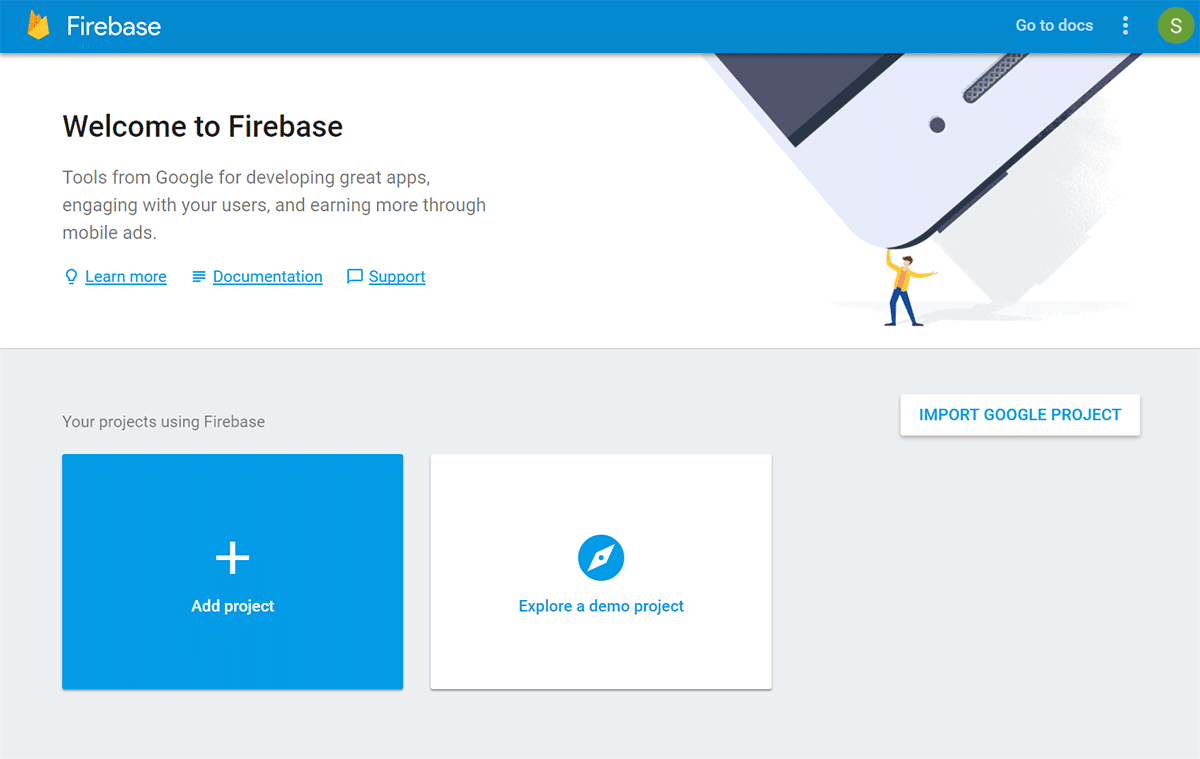
> ng serve

You can now access the project by visiting **[http://localhost:4200](http://localhost:4200/)** in the browser.

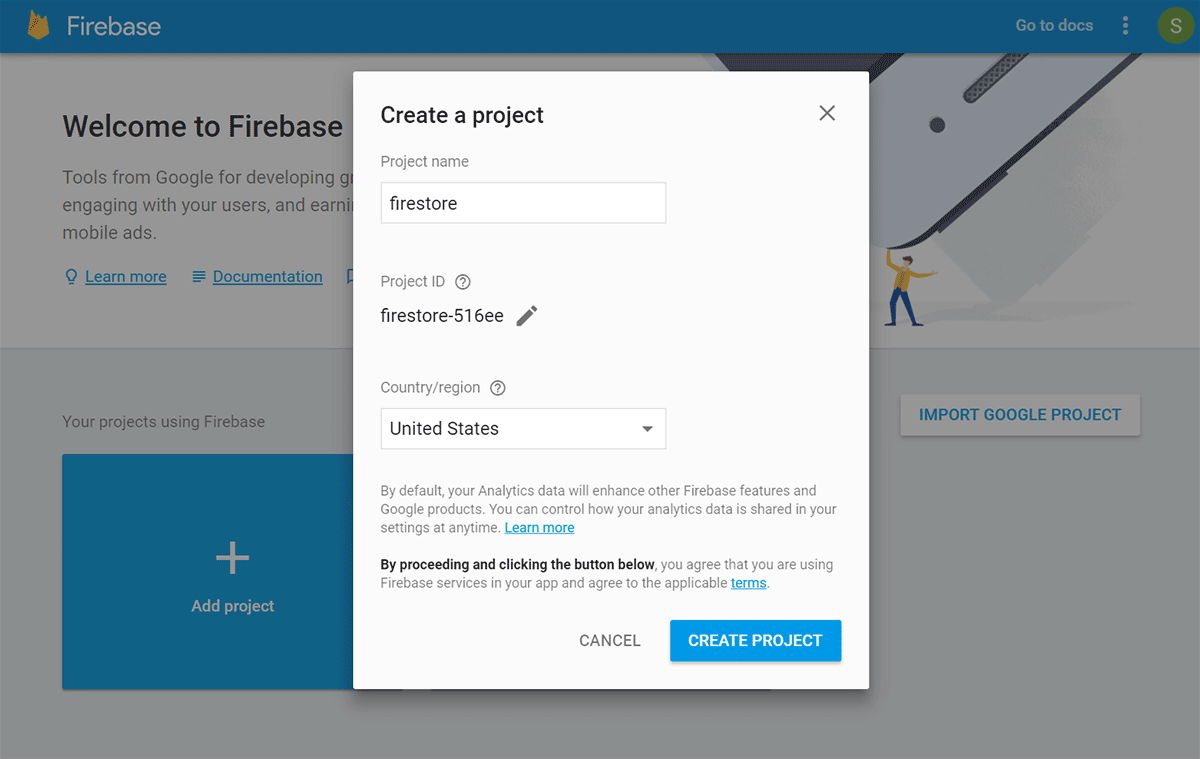
**Starting a New Cloud Firestore Project**

Login to your Google account and start a new tab on your browser. Visit the **[Firebase page](https://firebase.google.com/)** and click the Get Started button.

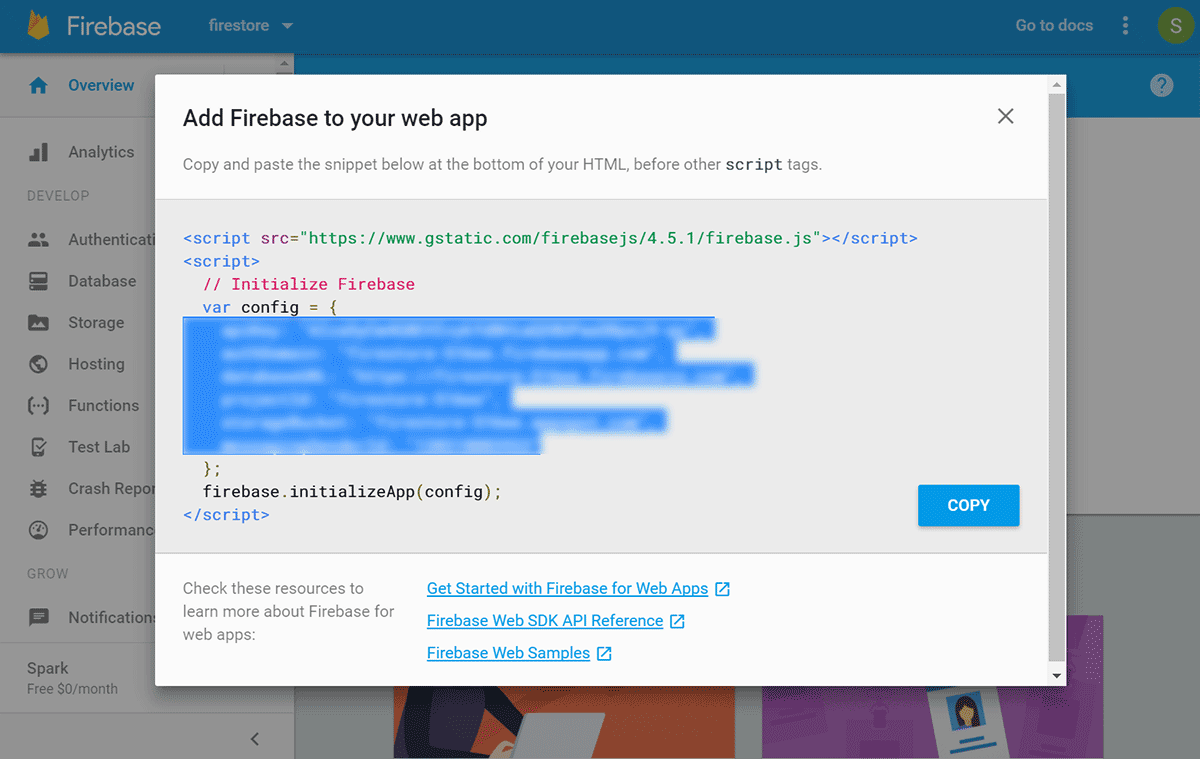
Click **Add project**



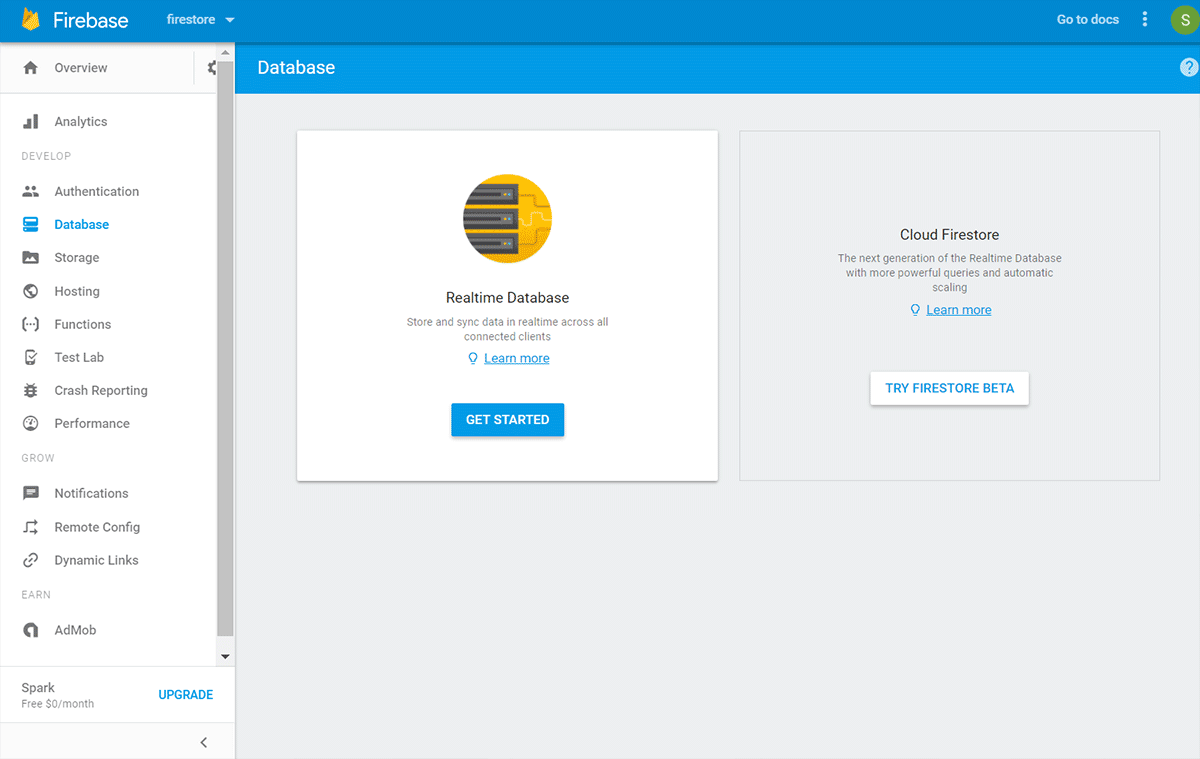
Next, give it a project name and hit **Create Project**



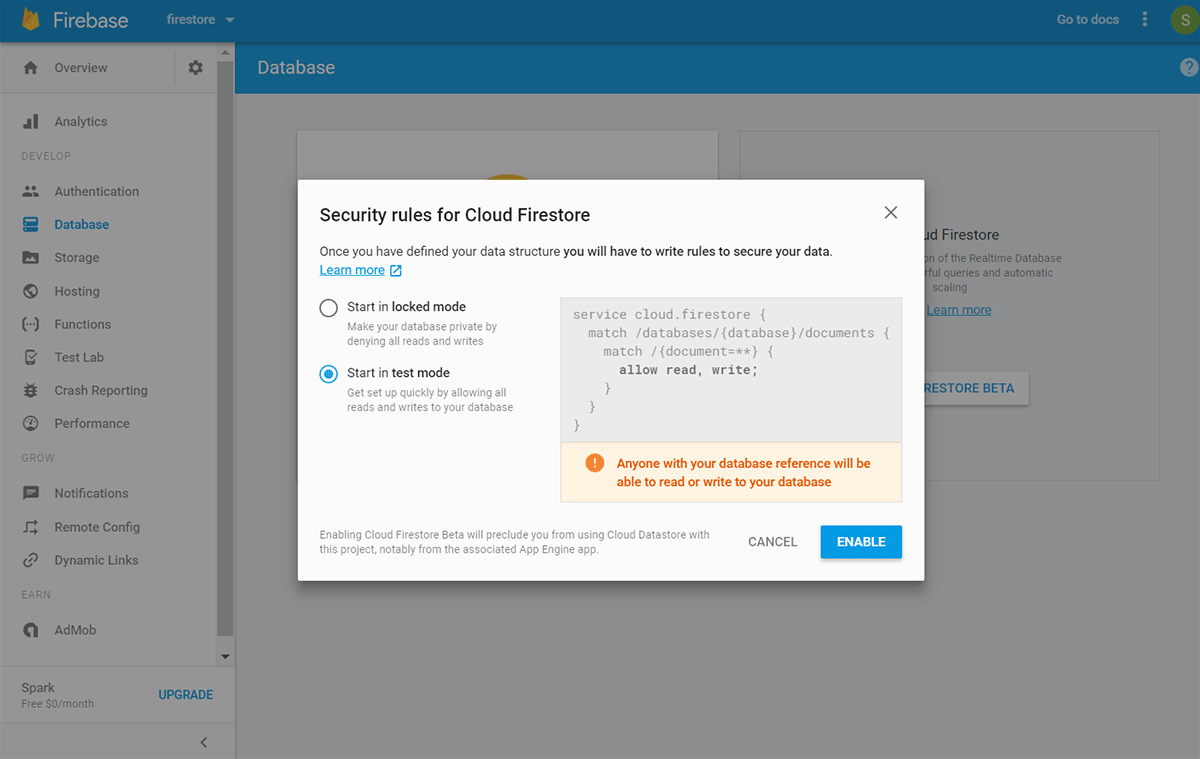
Click on the **Add Firebase to your web app** button and copy the following properties and values. Save them in a text document for later use:



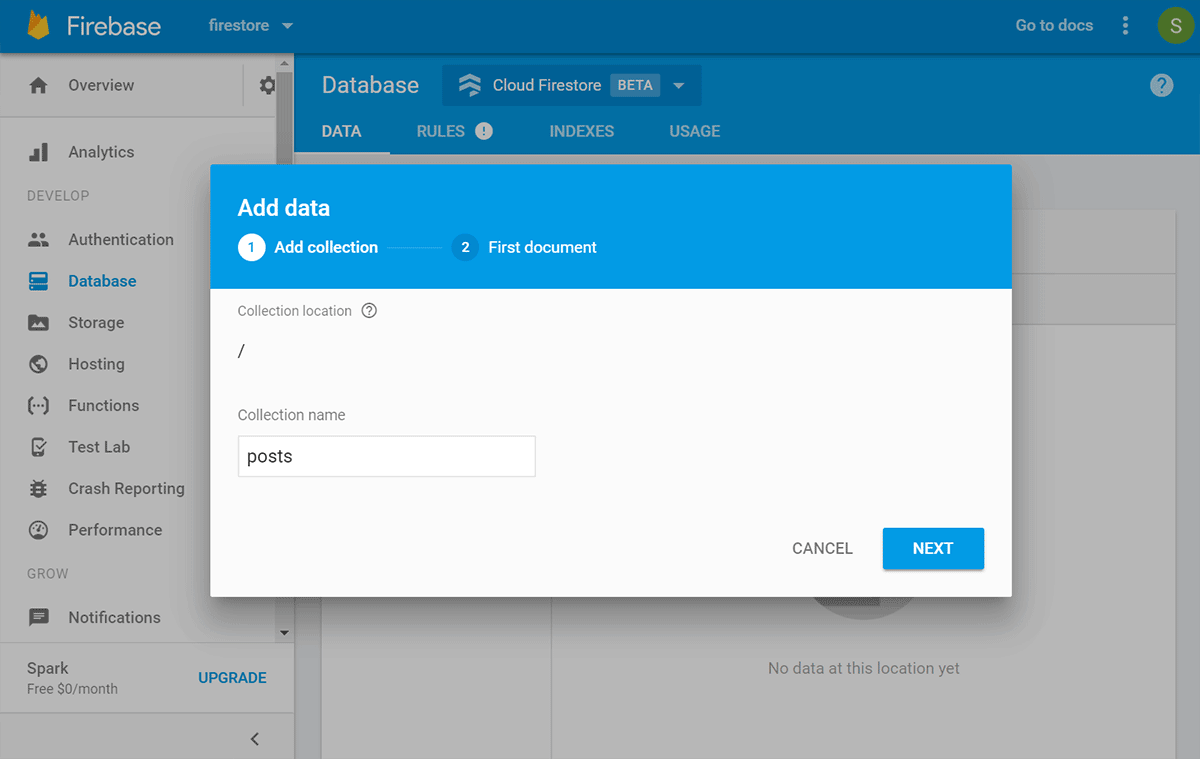
Click on the **Database** menu to the left, and then choose **Try Firestore Beta** (**Note**: It is currently in beta at the time of writing this tutorial, but that may have changed. This process should still remain the same).



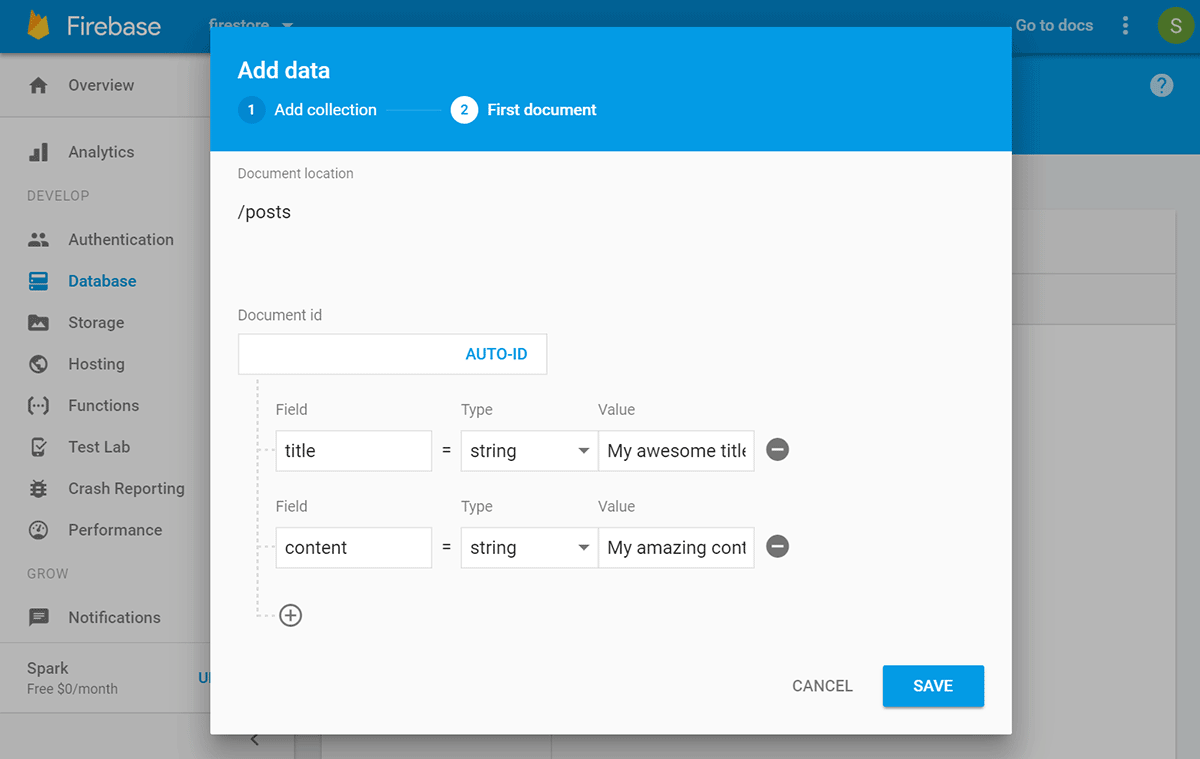
On the next screen shown below, choose **test mode** and then click **Enable**

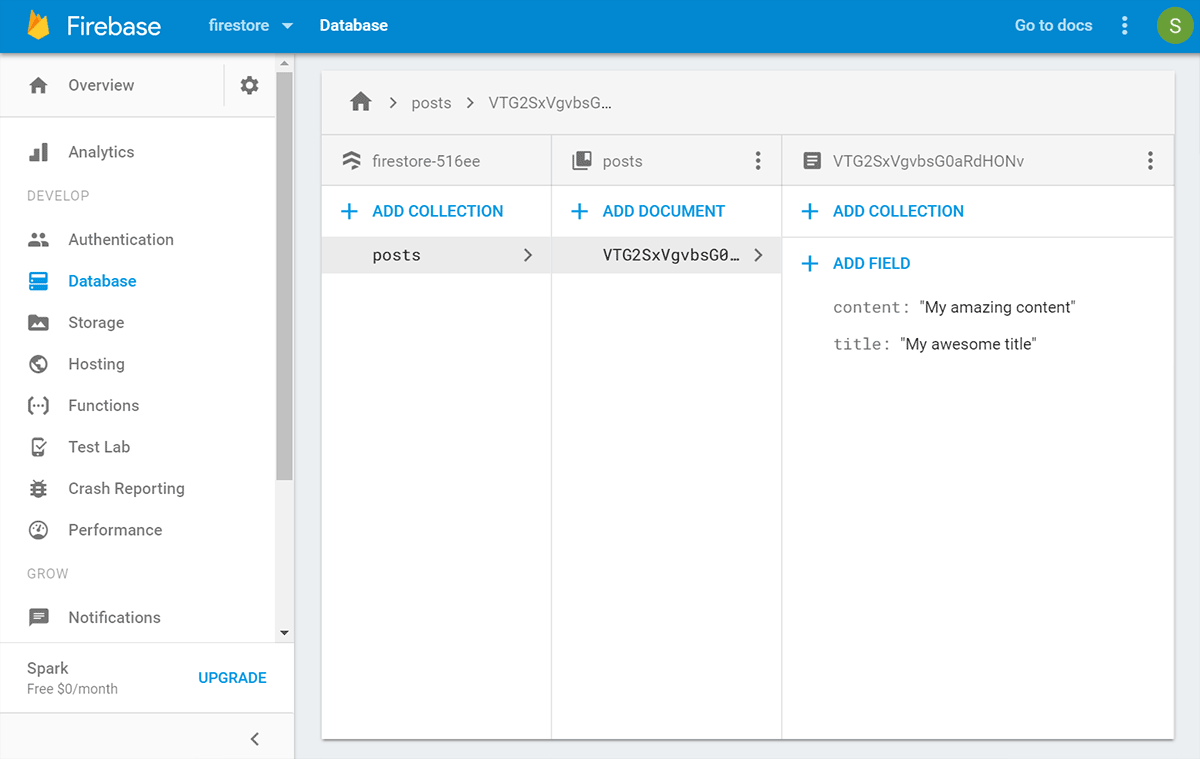


On the next screen, click **Add Collection** and name the collection "posts"



Then, add 2 fields of type string (title, content), and provide them with some initial value. Then, click **Save**





Great! That's all there is to the setup of our Firestore project for now.

**Integrating Firestore into the Angular Project**

Open up VS Code, or your favorite code editor and navigate to the **/src/app/app.module.ts** file.

Here, we have to import a couple libraries and add them to the imports section:

// Other imports removed for brevity

import { AngularFireModule } from 'angularfire2';

import { AngularFirestoreModule } from 'angularfire2/firestore';

// Paste in your credentials that you saved earlier

var firebaseConfig = {

apiKey: "",

authDomain: "",

databaseURL: "",

projectId: "",

storageBucket: "",

messagingSenderId: ""

};

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

AngularFireModule.initializeApp(firebaseConfig), // Add this

AngularFirestoreModule // And this

],

providers: [],

bootstrap: [AppComponent]

})

 Next, head over to the **app.component.ts** file located in the same folder, and add the following 3 import lines:

// (Standard component import up here)

import { AngularFirestore, AngularFirestoreCollection, AngularFirestoreDocument } from 'angularfire2/firestore';

import { Observable } from 'rxjs/Observable';

import 'rxjs/add/operator/map';

Further down this file, in the class section, add the following, which will create an instance of AngularFirestore through DI (Dependency Injection):

export class AppComponent {

constructor(private afs: AngularFirestore) {}

ngOnInit() {

}

}

Now, we're ready to rock!

**Retrieving a Collection**

If you were paying attention, you noticed we imported **AngularFirestoreCollection** and **AngularFirestoreDocument**.

We use these to interact with either the collections, which store a bunch of documents, or to retrieve specific documents. Let's focus on retrieving a specific collection.

First, we're going to define an Interface, which will help us define the structure of the data associated with our posts collection:

// Imports up here..

interface Post {

title: string;

content: string;

}

// @Component Decorator here..

In the class, add the following:

export class AppComponent {

postsCol: AngularFirestoreCollection<Post>;

posts: Observable<Post[]>;

constructor(private afs: AngularFirestore) { }

ngOnInit() {

this.postsCol = this.afs.collection('posts');

this.posts = this.postsCol.valueChanges();

}

}

First, we define two properties **AngularFirestoreCollection** of type **Post** (the interface defined above), and **posts** which will hold the **Post** array that's returned.

In the **ngOnInit()** lifecycle hook, which fires when the component loads, we bind **postsCol** to our AngularFirestore instance using the **.collection** method, wherein we pass the name of our collection (**posts**).

Then, we bind **posts** to **postsCol** and use the **.valueChanges()** method, which provides us with an Observable.

Head over to the **app.component.html** template, delete everything currently there and add the following:

<ul \*ngFor="let post of posts | async">

<li>

<strong>{{ post.title}}</strong>

<br>

{{post.content}}

</li>

</ul>

We use \***ngFor** to iterate over our **posts** object with the async pipe. Then, we use interpolation on **post** along with the field names we defined in Firestore.

Check out your browser and you will see the first document we created earlier! Go ahead and modify the title or content of the existing document and you will notice that the result in the browser instantly changes; it's realtime.

**Adding a Document**

We only have a single document that we added manually through Firestore, so, let's give ourselves the ability to add documents to the **posts** collection.

Being that we're already in the **app.component.html** file, add the following HTML above the unordered list:

<input type="text" [(ngModel)]="title" name="title" placeholder="Title..">

<textarea [(ngModel)]="content" name="content" placeholder="Content.."></textarea>

<input type="submit" value="Add a Post" (click)="addPost()">

Fairly simple, we're just adding a textfield and a textarea input, both with **ngModel** to communicate the value to and from the component class, and then we're calling **addPost()** on click from the button.

**Note**: Normally, you would probably want to use Angular's Reactive Forms for something like this, but that's a bit more setup. You can check out my **[Reactive Forms Tutorial](https://coursetro.com/posts/code/66/Angular-4-Reactive-Forms-Tutorial)** though to learn how to set it up.

Right now, this will error. That's because **ngModel** doesn't exist on input fields. We need to import the **FormsModule** in the **app.module.ts** file:

import { FormsModule } from '@angular/forms';

@NgModule({

imports: [

FormsModule

],

})

Next, in the **app.component.ts** file, at the top, add two more properties for our **ngModel** 2-way data binding:

title:string;

content:string;

Then, add the following **addPost()** method:

addPost() {

this.afs.collection('posts').add({'title': this.title, 'content': this.content});

}

We use the **add** method and pass in an object consistent with the structure of our collection structure.

Fill out the form and click "Add a Post", it will not add the post to Firestore and show it in the list below!

**Designating a Custom ID**

In our **addPost()** method above, the **.add()** method will automatically generate an ID for us. You know what's really ugly though?

This:  
**[mysite.com/posts/EFJISOFEW3891FOMG](http://mysite.com/posts/EFJISOFEW3891FOMG)**

This, however, is not ugly:  
**[mysite.com/posts/my-unique-post-id](http://mysite.com/posts/my-unique-post-id)**

To designate a custom ID, we simply need to adjust the **addPost()** method as shown:

addPost() {

this.afs.collection('posts').doc('my-custom-id').set({'title': this.title, 'content': this.content});

}

The **.doc** method accepts a string which is your unique ID, then you change **.add** to **.set**.

If you now add a new row through the form and visit the Firestore database, you will see the new ID.

**Note**: Ordinarily, you would use some mechanism to ensure the ID is unique.

**Retrieving a Single Document**

Sometimes, you need access to a specific document. In the case of blog posts, you retrieve a collection of blog posts on a home page, and when a user clicks on a title, they're taking to the full post.

We're not going to do exactly that, but we will demonstrate that same basic concept to show you how to retrieve a single document.

Unfortunately, our current **posts** property does not contain the ID, which is considered metadata.

In order to return the metadata, we have to make some changes to our code.

Just beneath where we defined the **interface Post**, add:

interface PostId extends Post {

id: string;

}

 Next, in the class, let's change the **posts** property to type **any**

posts: any;

Then, we need to redefine **this.posts** in **ngOnInit()** to use **snapshotChanges()**:

this.posts = this.postsCol.snapshotChanges()

.map(actions => {

return actions.map(a => {

const data = a.payload.doc.data() as Post;

const id = a.payload.doc.id;

return { id, data };

});

});

**snapshotChanges()** differs from the previous method because in addition to providing you with the document data, it also returns other metadata, which includes the ID. Seems like a lot of work just to get a single ID, huh?

Now, let's make some changes in the template.

Replace the **ul** element with the following:

<ul \*ngFor="let post of posts | async">

<li (click)="getPost(post.id)">

<strong>{{ post.data.title}}</strong>

<br>

{{post.data.content}}

</li>

</ul>

There's only a few changes here, but we're adding a click event to call a method **getPost()** where we're passing in the newly acquired id. We also had to change the object structure based on the mapping we did in the class.

Underneath this **ul**, add the following:

<h1>A specific post:</h1>

<h3>{{ (singlePost | async)?.title }}</h3>

<p>{{ (singlePost | async)?.content }}</p>

Once a user clicks on one of the results from the **<li>** elements, we will show the data associated with that specific post; just to demonstrate that we retrieved a single document.

Going back to our component class, let's add that new method:

getPost(postId) {

this.postDoc = this.afs.doc('posts/'+postId);

this.post = this.postDoc.valueChanges();

}

Let's define those properties above at the top of the component class:

postDoc: AngularFirestoreDocument<Post>;

post: Observable<Post>;

With any luck, once we click on one of the results, it will call our method and display the post data retrieved.

**Deleting a Document**

Navigate to the template and modify our **<strong>** element:

<strong>{{ post.data.title}}</strong> (<a href="#" (click)="deletePost(post.id)">delete</a>)

In the component class, define the new method:

deletePost(postId) {

this.afs.doc('posts/'+postId).delete();

}

Give it a try!

**Where Clauses**

Let's say for instance that we want to return documents in a collection that meet some sort of criteria. We can use a where method for this.

To do so, we modify our **this.postsCol**:

this.postsCol = this.afs.collection('posts', ref => ref.where('title', '==', 'coursetro'));

If you save this and look at the browser, it shows no results. If you add a new title with "coursetro", you will see it displays.

Unfortunately, if you try changing == to !=, it will not work. You can, however, use other operators like >, <, >=, <=.

**Other Query Options**

Aside from **where**, you can also use:

* orderBy
* limit
* startAt
* startAfter
* endAt
* endBefore

To learn more about these, check out the **[official docs](https://firebase.google.com/docs/firestore/query-data/queries)**.

**Let's Style This Already!**

This tutorial is just about over, but being a full stack dev., I can't leave you guys with such an ugly project!

Head over to the **/src/styles.css** file and add:

body {

background: #222427;

padding: 2em;

color:#fff;

font-family: 'Arial';

}

Then in **/src/app/app.component.css** add:

input, textarea {

display:block;

width: 100%;

margin-bottom:15px;

padding:10px;

background:none;

border-bottom: 1px solid #414a59 **!important**;

border:0;

color:#fff;

}

input[type='submit'] {

background-color:#ffcc00;

cursor:pointer;

color:#000;

font-weight:bold;

margin-bottom:50px;

padding:20px;

}

ul {

list-style-type:none;

width:100%;

margin:0;padding:0;

}

ul li {

background-color:#414a59;

padding:20px;

margin-bottom:2px;

color:#bec8d9;

}

ul li a {

color:#ffcc00;

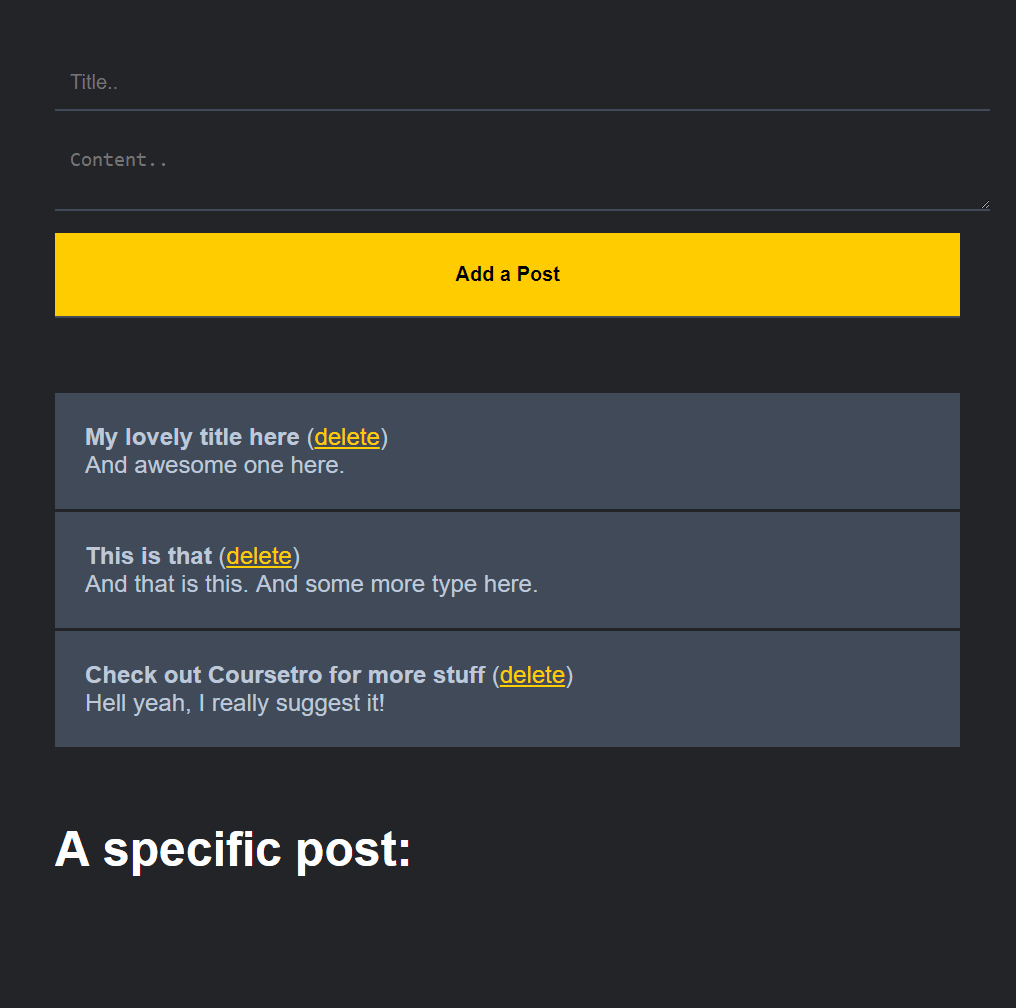
}

h1 {

margin-top:50px;

}

Save, and there we go, the result is now much more appealing.



**Conclusion**

Hopefully you were able to learn quite a bit here, and this is by no means an exhaustive tutorial. There's a lot more you can learn about Firestore, so once again, be sure to check out their **official docs** to learn more.