**Firebase - intro**

Firebase is what’s known as a backend service and provides services like database, authentication, file storage, cloud functions, hosting, etc. We can plug those services directly into our front-end applciations.

Firebase then takes care of all of the server-side logic, so FE developers don’t need to focus much on it. It’s an alternative back-end infrastructure to using tools like mongoDB and node.js server. Firebase handles all of that for us, letting FE to focus more on the FE and the user experience.

**Setting up webpack**

1. Install webpack and webpack-cli using: npm I webpack webpack-cli -D
2. Create a webpack.config.js file with the following code:

const path = require('path')

module.exports = {

  mode: 'development',

  entry: './src/index.js',

  output: {

    path: path.resolve(\_\_dirname, 'dist'),

    filename: 'bundle.js'

  },

  watch: true

}

**Creating a Firebase project**

1. Install firebase using npm install firebase
2. In order to create a firebase project, navigate to the firebase website, go to firebase console and click create a project.
3. Add a front-end application to get started 🡪 web app.
4. Click in continue to console.
5. Open your app and click in settings.
6. In the SDK Configuration tab, select “config” object. This object contains information about our firebase project, and we need to use it on our FE so it can connect to the firebase project and interact with it. Copy this code and paste it in index.js. Then your code in the index.js file would look something like this:

|  |  |
| --- | --- |
|  | import { initializeApp } from 'firebase/app'  const firebaseConfig = {    apiKey: "AIzaSyDmXgb\_58lO7aK\_ujN37pGlNxzWGEU0YpI",    authDomain: "fb9-sandbox.firebaseapp.com",    projectId: "fb9-sandbox",    storageBucket: "fb9-sandbox.appspot.com",    messagingSenderId: "867529587246",    appId: "1:867529587246:web:dc754ab7840c737f47bdbf"  }  // init firebase  initializeApp(firebaseConfig) |

**Creating a Firebase Database**

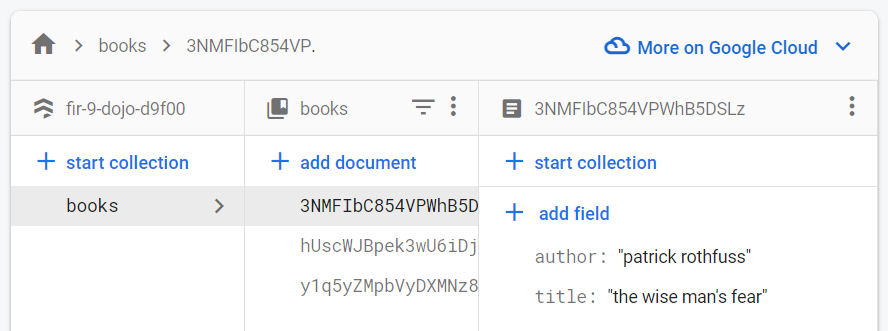
We are now connected to the Firebase backend. The next thing we want to setup a database and connect to that database from our FE, so that we can grab the data from it.

First thing to do is go to your firebase project 🡪 Creation tab 🡪 Firebase Database 🡪 Create Database 🡪 Start in test mode

Now you have a database which is split into collections and documents. For example, in the application we’re going to build, we are going to have a collection which will contain book names and authors, so we will call this collection “books”.

When we create a collection, we need to create a document inside it, which is basically an object inside your array of objects. It will also you for a document ID but you can click in AutoID and Firebase automatically creates is for you.

For example, for our “books” collection, we can add 3 documents with the properties of “author” and “title”.



**Getting data from Firebase to our FE application**

1. Initialize the Firestore Service.

The first step is to initialize our Firebase server on the FE so we can connect to it. Anytime we want to access our database we are going to use this db variable to do it.

import {getFirestore} from 'firebase/firestore'

const db = getFirestore()

1. Get a ref to a specific collection of our database.

Our reference for the books collection is now stored in side colRef variable.

getFirestore, collection} from 'firebase/firestore'

const colRef = collection(db, 'books')

1. Get the collection data.

We use the method getDocs to get all the docs inside that collection.

import {

  getFirestore, collection, getDocs

} from 'firebase/firestore'

getDocs(colRef)

This returns a promise, so we can use a then method, which will run when the task is complete. This method will take a snapshot of that collection in that moment, which is an array that contains our docs (objects).

This snapshot contains a data() property, which is where our data is stored in an elegant way. So we need to loop through this snapshot to have access to the object properties and create a new clean array, and lastly add the ID for each.

getDocs(colRef)

  .then(snapshot => {

    // console.log(snapshot.docs)

    let books = []

    snapshot.docs.forEach(doc => {

      books.push({ ...doc.data(), id: doc.id })

    })

    console.log(books)

  })

  .catch(err => {

    console.log(err.message)

  })

So the full document would look like

import { initializeApp } from 'firebase/app'

import {

  getFirestore, collection, getDocs

} from 'firebase/firestore'

const firebaseConfig = {

  apiKey: "AIzaSyDmXgb\_58lO7aK\_ujN37pGlNxzWGEU0YpI",

  authDomain: "fb9-sandbox.firebaseapp.com",

  projectId: "fb9-sandbox",

  storageBucket: "fb9-sandbox.appspot.com",

  messagingSenderId: "867529587246",

  appId: "1:867529587246:web:dc754ab7840c737f47bdbf"

}

// init firebase

initializeApp(firebaseConfig)

// init services

const db = getFirestore()

// collection ref

const colRef = collection(db, 'books')

// get collection data

getDocs(colRef)

  .then(snapshot => {

    // console.log(snapshot.docs)

    let books = []

    snapshot.docs.forEach(doc => {

      books.push({ ...doc.data(), id: doc.id })

    })

    console.log(books)

  })

  .catch(err => {

    console.log(err.message)

  })

**Adding Documents to the Database**

Consider we have the following form set up:

  <form class="add">

    <label for="title">Title:</label>

    <input type="text" name="title" required>

    <label for="author">Author:</label>

    <input type="text" name="author" required>

    <button>add a new book</button>

  </form>

In order to add a document, we need to target the form in the JS and add an event listener to it. When the user submits a title and an author, we want to target the reference we created (colRef) and add the object to it, with the respective properties and values.

We do this with the addDoc function provided by Firestore. This function is asynchronous and returns a promise. So, we can use then, which will run after the adding process is complete. And what we want to do is reset the form.

import {

  getFirestore, collection, getDocs,

  addDoc

} from 'firebase/firestore'

const addBookForm = document.querySelector('.add')

addBookForm.addEventListener('submit', (e) => {

  e.preventDefault()

  addDoc(colRef, {

    title: addBookForm.title.value,

    author: addBookForm.author.value,

  })

  .then(() => {

    addBookForm.reset()

  })

})

**Deleting Documents from the Database**

To delete a document, we need access to the documents in the database, so we need to import doc. Doc is similar to collection, but instead of giving a reference to collection, gives a reference to a document.

This doc function takes 3 arguments:

* The database
* The collection in which the document belongs
* The ID of the document we are targeting

import {

  getFirestore, collection, getDocs,

  addDoc, deleteDoc, doc

} from 'firebase/firestore'

const deleteBookForm = document.querySelector('.delete')

deleteBookForm.addEventListener('submit', (e) => {

  e.preventDefault()

  const docRef = doc(db, 'books', deleteBookForm.id.value)

  deleteDoc(docRef)

    .then(() => {

      deleteBookForm.reset()

    })

})

**Real Time Collection Data**

At the moment, in order to see the updated data in the browser we have to refresh the page. This is because the getDocs function we have used before just runs once.

We have to set up a real time listener to the Firestore collection, which listens for changes in that collection. To do that, instead of using getDocs, we need to use onSnapshot, which takes two arguments:

* The reference of the collection we want to listen to (colRef)
* Callback function, which is going to fire every time there is a change in the referenced collection, and it’s going to send a new snapshot of that collection after the change occurs.

The function will be very similar do the getDocs, but won’t run just when the page is loaded.

onSnapshot(colRef, (snapshot) => {

  let books = []

  snapshot.docs.forEach(doc => {

    books.push({ ...doc.data(), id: doc.id })

  })

  console.log(books)

})

**Firestore Queries**

At the moment, we are getting every single document inside the collection. However, we might want to filter it and just get some of them. We do that using a Firestore Query.

We need our reference to the collection to be a query reference instead. We need to import query and where functions. This query function takes two arguments:

* The collection reference.
* The where function, which will fetch any documents that follow a defined criteria.

The where function takes 3 arguments:

* The property name
* Comparison operator
* The value of the property we are looking for

For example, the query below asks for any object with the author equal to Patrick Rothfuss inside the colRef collection.

const q = query(colRef, where("author", "==", "patrick rothfuss"))

Then we have to call the onSnapshot function but this time using the query, not the collection.

onSnapshot(q, (snapshot) => {

  let books = []

  snapshot.docs.forEach(doc => {

    books.push({ ...doc.data(), id: doc.id })

  })

  console.log(books)

})

**Ordering Data & Timestamps**

The documents we get back either using getDocs on onSnapshot seem to be in no apparent order. But they actually are in alphabetic order with the ID, which is randomly generated by firebase. We can order it by a specific property (e.g. title).

We need to use orderBy function, which takes 2 arguments:

* The property we want to order by.
* “desc” for descending order or “asc” to ascending order. If we don’t pass anything it defaults to ascending order.

const q = query(colRef, where("author", "==", "patrick rothfuss"), orderBy('createdAt'))

If we do this, we are going to see the error below in the console, which is saying “we can’t create this query without first creating an index for it”. To sort it, we need to click in the link which opens your Firestore console, then click “Create Index”. The building might take a couple of minutes to do and you have to wait until the Status changes to “Enabled”.

Note that if we don’t provide the where function, it basically gets the whole collection.

const q = query(colRef, orderBy('createdAt'))

Another way we can do this is ordering by the time the object (or document) was added. We do this by adding a timestamp property each time we add a document to the collection.

Note that to add the documents with a timestamp property they need to be added through the FE application and not through the Firebase platform.

We create a timestamp by using the function serverTimestamp, and save it in the createdAt property.

  getFirestore, collection, onSnapshot,

  addDoc, deleteDoc, doc,

  query, where,

  orderBy, serverTimestamp

} from 'firebase/firestore'

const addBookForm = document.querySelector('.add')

addBookForm.addEventListener('submit', (e) => {

  e.preventDefault()

  addDoc(colRef, {

    title: addBookForm.title.value,

    author: addBookForm.author.value,

    createdAt: serverTimestamp()

  })

  .then(() => {

    addBookForm.reset()

  })

})

**Fetching a Single Document**

We have seen how to grab an entire collection using onSnapshot and getDoc. But what if we just want 1 document? For that, we need to use the getDoc function and pass a document reference. It takes the same arguments as when it’s used in the delete function:

* The database
* The collection in which the document belongs
* The ID of the document we are targeting

Then we can get the document and to something with it. Here we just log the object and its ID.

const docRef = doc(db, 'books', 'gGu4P9x0ZHK9SspA1d9j')

 getDoc(docRef)

   .then(doc => {

     console.log(doc.data(), doc.id)

   })

Much like before, we can also set up a real time listener (or a subscription) to a document as well. So, if that document ever changes, firestore will send a new version of it, after it has been changed.

onSnapshot(docRef, (doc) => {

  console.log(doc.data(), doc.id)

})

**Updating Documents**

To update properties in individual documents, we created another form in the HTML:

  <form class="update">

    <label for="id">Document id:</label>

    <input type="text" name="id" required>

    <button>update a book</button>

  </form>

For that, we need to create another function from firestore: updateDoc. This function takes 2 arguments:

* The reference to the document we want to update.
* An object, in which we can pass any properties that we want to update, and the updated value for that property.

Note that in this case we have hardcoded the updated value, but the correct way is to dynamically access the value from the form.

const updateForm = document.querySelector('.update')

updateForm.addEventListener('submit', (e) => {

  e.preventDefault()

  let docRef = doc(db, 'books', updateForm.id.value)

  updateDoc(docRef, {

    title: 'updated title'

  })

  .then(() => {

    updateForm.reset()

  })

})