**Firebase - intro**

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

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.

**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

}

* **Mode**: can be development or production
* **Entry**: path to the entry file (where we want webpack to initially look for the JS source file)
* **Dirname:** path to the folder we want our files to be put in to (in this case inside the 'dist' folder. We don’t need to go back to target the 'dist' folder because this is an absolute path, not a relative one.
* **Filename:** name for the output file.
* **Watch:** if true, when we run webpack what our entry file and every time we make a change, it’s going to bundle up the new code into our output file, as well as any imports inside it.

**Running webpack**

1. Go to **package.json** file search for “script” and on the following code nested on it: "build": "webpack" (the name of the script is optional, but in this case we have chosen "build". Now, when we run this script, it will automatically look for the **webpack.config** file and run webpack according to our configuration.
2. Run **npm run build** in the terminal (build is the name that we chose for the script). Webpack will take our source code and bundle it into the output file inside the 'dist' folder. It’s also going to look for changes in our index.js file, so it re-bundles every time we make a change and save the file.
3. In your HTML file, set the source to your output file name (in this case'bundle.js').

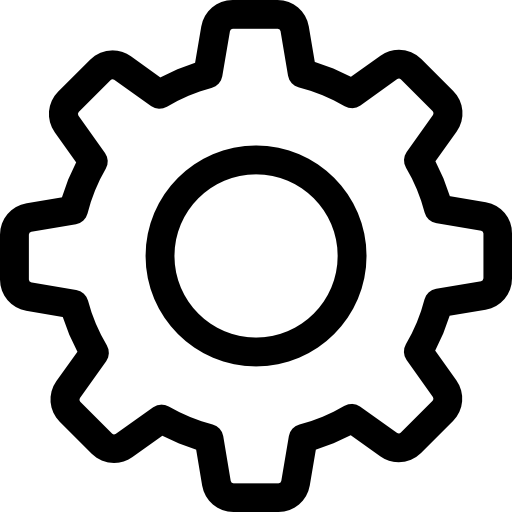
<script src="bundle.js"></script>

**Setting up Firebase**

First thing to do is install firebase and create a project:

1. Install firebase using **npm install firebase** in your project directory.
2. In order to create a firebase project, navigate to the firebase website, go to firebase console and click create a project.
3. Click add project to add a front-end application and get started
4. Choose the name of the project and click continue.
5. Disable “enable Google Analytics on this projects” and click create project.
6. Once your project is created, click in continue.

Then we have to register: a FE application for the project

1. Navigate to the dashboard of your project.
2. Click in the type of App you want to create (iOS, Android or Web).
3. Give it a name and click register app.
4. Click in Continue to Console.
5. Click in your app and click on the 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**

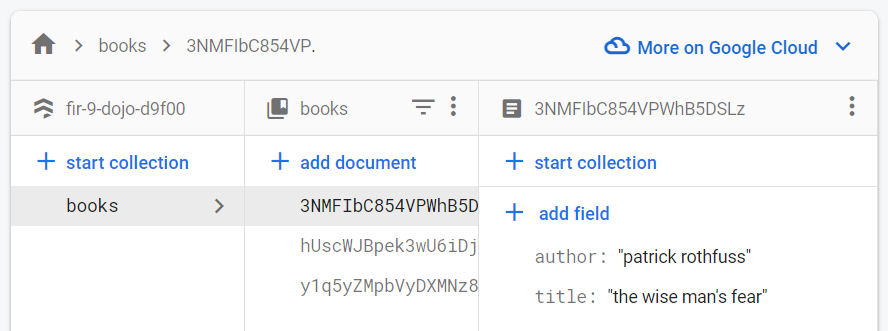
We are now connected to the Firebase backend. The next thing we want to **setup a database and connect to that database from our FE**, so that we can grab the data from it. To do that

1. Go to your firebase project.
2. Click in the Creation tab on the left-hand side.
3. Click in Firestore Database
4. Click in Create database
5. Select Start in test mode and click in Advance.
6. Select the Cloud Firestore location and click in activate.

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 ask 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**

To connect our FE app to the database and grab the documents follow the steps below. The setup is going to be done in our source file index.js.

1. **Initialize the Firestore Service** -The first step is to initialize or connect 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 inside 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 **documents** (or objects).

This snapshot contains a data property, where our data is stored in an elegant way. We need to **loop through this** snapshot **to have access to the object properties** and **create a new clean array**. Lastly, add the id for each document (given in the firestore dabase).

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 complete document would look like this:

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 use reset to clear 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 a collection, **gives a reference to a document**.

This doc function takes 3 arguments:

* The database (db)
* The collection in which the document belongs ('books')
* The ID of the document we are targeting, which in this case we input it in the form and we target the value with deleteBookForm.id.value

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**.

To do that, 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 (colRef)
* The where function, which will **fetch any documents that follow a defined criteria**.

The where function takes 3 arguments:

* The property name that we want to look or
* 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**

**Ordering Data by ascending or descending order**

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 by default**, 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('title', desc))

If we do this, we are going to see the error below in the console saying “we can’t create this query without first creating an index for it”.

To sort it, we need to

1. click in the link which opens your Firestore console,
2. Go to the Indexes tab.
3. click “Add index”. The building might take a couple of minutes to do and you have to wait until the Status changes to “Enabled”.

**Ordering Data with Timestamps**

Another way we can do this is **ordering by the time the object (or document) was added**. To add the documents with a timestamp property they need to be added through the FE application and not through the Firebase platform. We do this by adding a createdAt property each time we add a document to the collection.

  addDoc(colRef, {

    title: addBookForm.title.value,

    author: addBookForm.author.value,

**createdAt: serverTimestamp()**

  })

  .then(() => {

    addBookForm.reset()

  })

Now instead of ordering by ascending or descending order, we order it using our createdAt property, which contains timestamps created by the function serverTimestamp Note that if we don’t provide the where function, it basically gets the whole collection.

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

So, the complete code would look like this:

  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 getDocs. 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 (db)
* The collection in which the document belongs ('books')
* The ID of the document we are targeting ('gGu4P9x0ZHK9SspA1d9j')

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 (docRef)
* An object, in which we can pass any properties that we want to update, and the updated value for that property (title: 'updated title')

Note that in this case we have hardcoded the updated value to'updated title', 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()

  })

})