EXPRESS AND FIREBASE NOTES

The following document is a compilation of my notes for studying Express and Firebase. As I have been learning about these technologies, I have been taking notes on important concepts, commands, and code snippets that I have found useful. This document serves as a collection of these notes, organized by topic, in order to provide a comprehensive reference for myself as I continue to learn and work with these tools. I hope that these notes may also be helpful to others who are new to Express and Firebase and are looking for a concise and organized resource to help them get started.

1 Creating a Server

To create an Express server using JavaScript, follow these basic steps:

1. Import the Express module:

```
const express = require('express');
```

2. Create an instance of the Express application:

```
const app = express();
```

3. Define the routes for handling incoming requests. This is done using the ‘app‘ object’s methods such as ‘get()‘, ‘post()‘, ‘put()‘, ‘delete()‘, etc. Here’s an example of defining a route that handles GET requests:

```
app.get('/', (req, res) => {
  res.send('Hello, World!');
});
```

4. Start the server by listening on a specified port using the ‘listen()‘ method:

```
const PORT = 3000;
app.listen(PORT, () => {
  console.log(`Server listening on port ${PORT}`);
});
```

With these steps, you should have a simple Express server that can handle incoming requests and respond to them accordingly. Of course, there are many more advanced features and techniques you can use with Express, but these basic steps should get you started. Now if you visit the Express server at localhost:3000, you should be able to see the ”Hello, World!” string you entered in the res.send() function.

2 HTTP Requests

In the previous example, we demonstrated the use of the GET request to handle incoming requests. The GET request is one of the HTTP request methods used to retrieve data from a server. In addition to the GET request, there are other HTTP request methods that can be used to perform different actions on server data.

Here's a list of some of the other common HTTP request methods including GET:

- **GET:** retrieves a resource from the server. This is typically used for reading data from the server, such as fetching a list of items or a specific item.
- **POST:** Used to submit data to be processed to a specified resource
- **PUT:** Used to update a specified resource with new data
- **DELETE:** Used to delete a specified resource
- **PATCH:** Used to update a part of a specified resource with new data

Each of these request methods has a specific use case and can be used to perform different actions on server data. This is all nice and dandy, but we also want to be able to create a Firebase database so that we can successfully run CRUD operations on our application.

3 Firebase Configuration

To connect to Firebase and use Firestore, the first step is to install the firebase-admin package in your project.

You can do this by running the following command in your terminal:

```
npm install firebase-admin --save
```

After the package is installed, you need to create a new file called firebase.js in your project. This file will contain the code that sets up the connection to Firebase and exports a Firestore client instance that can be used throughout your application We will now enter the following code within the firebase.js file:

```
// Import the necessary functions from the Firebase Admin App library
const { initializeApp, cert } = require('firebase-admin/app')

// Import the necessary function from the Firebase Firestore library
const { getFirestore } = require('firebase-admin/firestore')

// Load the service account credentials from a JSON file
const serviceAccount = require('./creds.json')

// Initialize the Firebase Admin App with the service account credentials
initializeApp({
    credential: cert(serviceAccount)
})

// Get a Firestore client instance
const db = getFirestore()

// Export the Firestore client instance for use in other parts of the application
module.exports = { db }
```

This code is a simple Firebase admin module that sets up a Firestore client instance and exports it for use in other parts of the application.
Here's a breakdown of the code:

1. **firebase-admin/app**: the Firebase Admin App library.
2. **firebase-admin/firestore**: the Firebase Firestore library.
3. The **serviceAccount** object is loaded from a JSON file, which contains the credentials for the Firebase project.

4. The **initializeApp**: function is called and passed the **serviceAccount** object as the **credential** option. This function initializes the Firebase Admin App with the given credentials.

5. The **getFirestore**: function is called to get a Firestore client instance.

6. Finally, the **db** object is exported as a module so that it can be used in other parts of the application.

The overall goal of this code is to provide a centralized way of setting up a Firestore client instance that can be reused throughout the application, rather than having to set up the Firestore client in every module that needs it. This makes it easier to manage the Firestore client and makes the code more organized and maintainable.

4 Create Record

Start by writing the following code to create a new record at the /addAuthor endpoint.

```
app.post("/addAuthor", async (req, res) => {
  try {
    const { author, quotes } = req.body;

    // get reference to the author's document
    const authorRef = db.collection("authors").doc(author);
    const doc = await authorRef.get();

    if (!doc.exists) {
      // if the author doesn't exist, create a new document with the author and quotes
      const newAuthor = await authorRef.set({
        author,
        quotes: [quotes],
      });
    } else {
      // if the author exists, append the new quote to the existing quotes array
      const existingQuotes = doc.data().quotes;
      const updatedQuotes = [...existingQuotes, quotes];
      const update = await authorRef.update({
        quotes: updatedQuotes,
      });
    }

    res.status(200).send({ message: "Quote added successfully." });
  } catch (error) {
    console.error(error);
    res.status(500).send({ message: "Internal server error." });
  }
});
```

What is req.body?

The *"req.body"* is an "object" derived from the request "body" of a HTTP POST, PUT, or PATCH request. This object can be destructored so that the properties are extracted from the request. These extracted values do have to be from the request body, as that is where they are expected to be submitted by the client.

So in our example, we are using destructoring to extract "author" and "quotes" properties from the "req.body" object.

```
const { author, quotes } = req.body;
```

What does *await* do in *const doc = await authorRef.get()*?

The await keyword is an alternative and much simpler way to handle asynchronous operations/promises in JavaScript. Therefore, in our example, we are using the “await” keyword to await the result of a Promise returned by the “get()” method of a Firestore database reference.

The “get()” method is an asynchronous operation that returns a Promise that resolved with a “DocumentSnapshot” containing the data of the requested document. Thus, by using “await” with the “get()” method, we are telling the program to wait until the **Promise** is resolved with the **DocumentSnapshot** object before continuing with the execution of the code.

Why is it important to use *await*?

If we did not use *“await”* in this line, the program would continue executing without waiting for the **Promise** to resolve, and the doc variable would contain an unresolved **Promise** instead of the **DocumentSnapshot** object. By using “await”, we can ensure that the “doc” variable contains the expected value before moving on to the next line of code.

What is happening in the following block?:

```
const newAuthor = await authorRef.set({
  author,
  quotes: [quotes],
});
```

So similar to the *get()* function above, the *set()* function also returns a promise that will be resolved and assigned to the *newAuthor* constant variable. That is why the *await* keyword is used here. Additionally, the “quotes: [quotes]” syntax is creating a new array with a single element that contains the value of the “quotes” variable. So, in effect, we’re creating a new document in a Firestore collection that represents an author, and the document contains two fields: an *author* field that contains the name of the author, and a *quotes* field that contains an array with a single element representing a quote from the author.

What is happening in the following block?:

```
const existingQuotes = doc.data().quotes;
const updatedQuotes = [...existingQuotes, quotes];
const update = await authorRef.update({
    quotes: updatedQuotes,
});
```

In this code, we're retrieving the existing quotes from the *doc* object by accessing the *quotes* field using dot notation (doc.data().quotes). We're then creating a new array called *updatedQuotes* that contains all the existing quotes and the new quotes value. We're using the spread syntax to spread the elements of existingQuotes into the new array, and then adding the quotes variable as a new element at the end of the array. Finally, we're updating the Firestore document with the new *updatedQuotes* array and using the await keyword since the *update()* function returns a Promise.

So, in effect, we're taking the existing quotes, creating a new array that includes the existing quotes and the new quote, and updating the Firestore document with the new array. This way, we're appending the new quote to the existing quotes array in the document.