**Firebase Notes**

**Intro**

* Firebase is a backend as a service.
* This means it provides backend services such as a database, authentication, file storage, cloud functions, hosting, and many other features.
* It is an alternative to setting up our own backend infrastructure with mongodb and nodejs

**Firebase 9 vs 8**

* The major change is that firebase 9 now adopts a more modular and functional approach which means we only import the firebase functions that we need.
  + Ex: version 9 project
    - ­Graphical user interface, text, application

      Description automatically generated
    - Notice we only imported the getAuth and onAuthStateChanged functions
* In contrast, firebase 8 used a more object-oriented approach where we directly called those firebase methods on firebase objects.
  + Ex: version 8 project
    - Text

      Description automatically generated
    - Notice we imported everything from firebase auth.
* By using a more modular approach, we can take advantage of tree-shaking. Tree shaking is where any unused code/functions can be removed from the final bundled JavaScript file. To do that, we will need to use a module bundler such as webpack

**Setting up webpack**

* In order to take advantage of tree shaking in firebase, we need to use a module bundler such as webpack.
* A module bundler is a tool that takes pieces of JavaScript and their dependencies and bundles them into a single file, usually for use in the browser.
* It is that js bundle that we link to in our html page.
* If we create an app using a cli tool such as ‘create-react-app’, then a modular bundler is often automatically setup for you. In those cases, we don’t need this section and can just start using firebase.
* However, if we are making an application using just html and vanilla JS, we need a modular bundler.
* Set up the following folder
* Graphical user interface, application

  Description automatically generated
* Make sure you have node js installed
* Make sure the project folder name is valid (has no spaces)
* Inside the project folder, the folder that contains dist and src, type in the cmd line: npm init -y
* Run: npm i webpack webpack-cli -D
* Create a webpack.config.js file in the project folder. Inside the folder is where we config what we want the webpack to do. We want it to look at our src/index.js file and any other imports and bundle all of that code into a single bundle file.
* To do this, we need to export an object that represents our webpack configuration by copy pasting the following into the webpack.config.js file:
  + <https://raw.githubusercontent.com/iamshaunjp/Getting-Started-with-Firebase-9/lesson-2/webpack.config.js>
  + Text

    Description automatically generated
* Module.exports exports an object from the file and this object has properties that represents the configuration of webpack.
* The mode property can be production or development. Since we are developing, we choose development.
* The entry property is the path to an entry file. This path tells webpack where to look for our index.js file.
* The output property is an object.
  + This object has a property called path which is the path to where we want the output file to be put into. We want it inside the dist folder. In order to create this path, we need to use path module which we require in line 1. This is a core node module. We cannot use a relative path here, we must have an absolute path which is why we need node.
  + The object has a property called filename which is just the filename of the output file.
* The watch property is true which means that when we run webpack, every change we make is going to bundle up the new code into the bundle.js file.
* Now, we want to run webpack which we can do by running a custom script in our package.json file.
* Text

  Description automatically generated
* We didn’t have to name it build, we could have named it whatever wanted.
* Now, when we run the build command, the webpack command is ran which runs webpack according to our configuration.
* We can test that our webpack is working properly by adding a console.log(“hi”) to our index.js file.
* Then, in cmd: npm run build
* Recall that build was the name of the script
* This script will make webpack take our source code and bundle it into the bundle.js output file in the disc folder. As well, it will also be watching our index.js file for changes so that it rebundles every time we make a change and then save the file.
* Note that we also need to link to the new bundle.js file in the index.html file.

**Setting up firebase**

* Sign up for a firebase account
* Go to Firebase console at console.firebae.google.com which is where all of our firebase projects are listed.
* Generally, each different application we create will have a new firebase project for it
* Create a new project, type in any project name you want, you can remove google analytics (having google analytics is optional), click create project, and then click continue to go to the dashboard for that project
* We need to create a frontend project to connect with the dashboard which will manage the backend of the project
* We can create a web app so click the web app icon right under the project name, give it any name you want, we don’t need to check the set up firebase hosting, click register app, then click continue to console.
* Graphical user interface, text, application, chat or text message

  Description automatically generated
* Notice that 1 app icon which represents the app we just registered. This is the app that our frontend will connect to. Click on 1 app then click on the cog to go to settings then scroll down and toggle config to get the config object.
* Graphical user interface, text, application, email

  Description automatically generated
* This config object contains information about our firebase project that will allow our frontend to connect to our backend
* In index.js, paste in the config object.
* However, while this config object contains information, we are not doing anything with the information. Thus, we first need to install firebase by running in a new cmd terminal (not the webpack run build terminal): npm install firebase
* Now, we can use some firebase functions to initialize our app.
* Text

  Description automatically generated

**Firestore Setup**

* Now that we connected our frontend to the firebase backend, we can set up a database
* First, we need to enable that database
* The database we will be using is the Firestore database. There is an older realtime database, but Firestore will be used instead.
* Graphical user interface, application

  Description automatically generated
* Click create database, start in test mode, then click next. Choose a firestore location, click enable, and now, we have a database created for us.
* This database is split up into collections and documents.
* We can have a collection of certain datatypes. For example, we could have a datatype called books and they would all be in a collection called books.
* To make a collection, click start collection, give it a collection id/name such as books.
* Then, we need to create our first document inside that collection. Each document has to have a document id so if we need to grab a document from the database, we can use the id. We can click auto-id to automatically generate an id.
* Inside the document, we have different properties/fields and values. For example, a field could be title and its value could be harry potter. We can also specify the datatype to be a string, number. We could have another field to be author and its value could be jk rowling.
* Then, we can click save and that’ll create a new id inside the firestore database.
* We can also create some more dummy data.

**Fetching firestore data**

* Firstly, we need to initialize the firestore service on the frontend so we can connect to it.
* To do that, import: getFireStore. Note that this naming convention of getService is common for many firebase services
* We can execute the getFireStore function to initialize the firestore service. Thus, we say const db = getFireStore(). This db constant will represent our database connection. Anytime we reach out to get data, we’re going to use the db constant.
* Now, we need to get a reference to a specific collection in our database.
* We will import a function called collection that will enable us to get a reference to a specific collection. This collection function takes in two arguments. The first argument is the database we will be looking in which in our case is db. The second argument is the collection we will be looking for such as ‘books’. This function returns a collection reference. Note that the collection we are looking for – the second argument – does not have to exist in the database. For example, we could say const ref = collection(db, ‘random\_stuff’) and it would not raise an error.
* We can import the getDocs function that will enable us to get the collection data. We pass in a collection reference as argument. In our example, we pass in colRef to the argument of the getDocs. This getDocs returns a promise. We can then add a .then method which takes in a snapshot object of that collection in that moment in time when we reach out to get it. We can then simply log out snapshot.docs and this docs is a property of snapshotl which represents all of the documents.
* As of now, our code can might look like the following:
* Text

  Description automatically generated
* The console would log the following:
* Text

  Description automatically generated
* We probably don’t need all of these properties, we probably just want the data and the id.
* The data is grabbed by using a data function. We can grab the id from the id property.
* We can now go through each document in snapshot.docs and then use the spread operator to get all the fields of the data along with the id property.
* Text

  Description automatically generated
* Text

  Description automatically generated

**Adding and Deleting Firebase Documents**

* Firstly, we can add a form to our html file that will allow the user to add information/delete information.
* Text

  Description automatically generated
* To add documents, we need to import the addDoc function from firebase which will allow us to add a new document to a specific collection.
* Then, we can use that function by executing it. The function takes in two arguments, the first is a collection reference which is the collection we want to add to. The second argument is an object that represents the new document that we want to add to the particular collection. Note that the first argument, the collection reference does not need to reference an existing collection. It can reference a collection with a new name which means a document will be added to the database under a new collection name.
* This addDoc function is asynchronous so we can attach a .then method. Inside the .then method, we can reset the fields of the form to be empty for better use experience.
* Text

  Description automatically generated
* To delete documents, we need to import the deleteDoc and doc functions from firebase.
* The doc function is similar to the collection function in the sense that we get a reference. However, instead of getting a reference to a collection which is what the collection function does, the doc function gets us a reference to a doc. The doc function takes in 3 arguments. The first argument is a database. The second argument is the collection. The third argument is the id of the document that we want to reference to.
* Now that we have a refence to a specific document by using the doc function, we can now delete the document by using the deleteDoc function.
* The deleteDoc function takes in 1 parameter which is a reference to a document that we want to delete. We can execute the function. Moreover, this function is asynchronous so we can add a .then method. Inside the .then method, we can reset the fields of the form to be empty for better use experience.
* A screenshot of a computer

  Description automatically generated with medium confidence
* Note that if we delete all the documents inside a collection, the collection does not get deleted. Rather, the collection is just empty.
* Note that if we try to delete a document that does not exist, nothing happens, there is no error.

**Realtime Collection Database**

* If we add/delete data, in order to see the new data, we’d have to refresh the page.
* Thus, if we were using something like react to render that data, in order for the rerender to occur, we would need to refresh the page which is not ideal. Instead, it should automatically rerender.
* Thus, we can set up a realtime listeners (aka a subscription) to the firestore collection which listens for changes to that collection and sends back the updated data inside the collection once the change happens.
* Currently, our getDocs function as shown below only runs once:
  + A screenshot of a computer

    Description automatically generated with medium confidence
* Instead of using getDocs, we will import the onSnapshot from firebase.
* The onSnapshot function takes two arguments. The first is a collection refence to a collection which is the collection that we want to listen to changes to. The second argument is a function that executes every time there is a change in the collection. As well, this function executes once initially as well. This function takes in a snapshot as a parameter much like how the .then method attached to the getDocs method takes in a snapshot.
* Text

  Description automatically generated
* Now notice whenever we change our books collection, the updated data is logged to the console.

**Firestore queries**

* Currently with the onSnapshot function, it’s second argument (the function) is executed everytime the data inside the colRef collection reference changes.
* Sometimes, we don’t want every document in a collection, but just certain documents. For example, documents with title === ‘green’.
* To do so, we will use the firestore query to make our collection reference to be a query refence instead.
* We will need to import ‘query’ and ‘where’ functions from firebase
* The query function takes in a collection refence as it’s first argument. The second argument is optional and is the where function. The where function takes in 3 parameters. The first is a property name such as ‘author’. The second is a comparison such as ‘==’. The third is a value to compare with with. Note that we need to use ‘==’ and not ‘===’.
* 
* In this above line of code, q is a query refence to all the documents inside the collection that colRef refences that satisfy the property that the author property of the document is equal to 23.
* Now on the onSnapshot function, we can replace the first parameter of colRef (the entire collection reference), to q(the query reference) so that the second parameter of onSnapshot (the function) is executed only when q changes.
* Text

  Description automatically generated
* Thus, when we first load the page, the onSnapshot’s parameter function is executed. In our case, it logs out all the documents inside the query reference. Upon adding/deleting documents that satisfy the query reference where function, the onShapshot’s parameter function is executed again to log the updated document data.
* Text, letter

  Description automatically generated
* Note that if we get the above error, click the link.

**Timestamps and Ordering Data**

* When we retrieve documents, they documents are sorted by id by default.
* We can also order the documents by properties such as ‘title’.
* To do so, we will import the orderBy function from firestore.
* We can add orderBy as a parameter to the query function. orderBy takes in two arguments. The first is the property we are ordering by. The second is the option and are the values of ‘asc’ or ‘desc’ to indicate if we should order in ascending or descending order. By default, the document’s are sorted in ascending order based on our specified property so we can not include the second argument of the sortedBy function if we want to sort in ascending order.
* Ex: 
* We can also still include the where function as a parameter in the query function if we still want.
* Notice now, when we log the data, it is sorted in ascending order based on the title.
* Text

  Description automatically generated
* We can use the time the data was added as an ordering property.
* To add this property, we cannot use a js Date object. Rather, we need to import the serverTimestamp function from firebase which gives us the time upon executing the function.
* Ex:
  + Text

    Description automatically generated
  + 
  + Text

    Description automatically generated
* Notice in the console logs out the updated data twice whenever we add data (doesn’t apply for deleting data).
* Also notice the value of the createdAt properties are different, as the first log has a value of null, while the second does not.
* This is because we added the serverTimestamp function which takes some time to add the timestamp to the document. When we add a document to firestore, the query reference changes (since there is a new document and our query reference doesn’t have any comparison since we removed that parameter). This new document has a property called createdAt but it’s value is null since it takes firestore some time to add the timestamp. Nonetheless, there is still a change which causes the parameter function of onSnapshot to execute. This logs out the data of the snapshot with the value of the createdAt property being null. Shortly after, firestore adds the time to the createdAt property which is a change to the query reference which causes the parameter function of onSnapshot to execute. This logs out the data of the snapshot with the value of the createdAt property being an actual time.

**Fetching Single Documents**

* Instead of getting an entire collection of documents, we may only want to get a single document.
* To do so, we first need to create a document refence (just like what we did to refence the document we want to delete)
* We can import the getDoc function which is similar to getDocs. Instead of passing in a collection refence which is what getDocs uses, we pass in a document reference to geDoc. Instead of returning a promise with a resolve value of a snapshot of a collection of documents which is what getDocs returns, getDoc returns a promise with a resolve value of a snapshot the document we pass into getDoc as a parameter.
* Instead of using getDoc, we could use a real time database with onSnapShot as shown below. In the below example, we are listening to changes to a single document (refenced by the docRef document reference) and executing a function upon changes (this function just prints out the doc data).
* Text

  Description automatically generated
* Now, if we update the docRef with the specified id in the firebase console, it will cause the parameter function in the onSnapshot function to execute, logging out the updated data of the document that docRef references. Note that changing this one document is also considered a change to the entire collection so the onSnapshot function that listens to changes on the entire collection also executes, logging out all the collection data.
* Ex console output: Text

  Description automatically generated Note that the onSnapshot function for the document is executed second since it comes after the onSnapshot function for the collection in our code.

**Updating Documents**

* To update a document, we first need a document reference such as: 
* Then we need to use the updateDoc function that we import from firebase. This function takes in a document refence (which is the document we want to update) as its first parameter. The second parameter is an object with properties and values (which represent the fields we want to change and the value we want to change to). Note that we only need to pass in the properties we want to change. The updateDoc function is asynchronous and so we can add a .then method.
* Graphical user interface, text, application

  Description automatically generated

**Firebase Auth Setup**

* Firebase auth uses a JSON web token to authenticate users who sign up/login/logout. When a user is logged in, this web token is sent to the firebase servers on every request, and firebase will enable that user to have access to certain features on the site such as reading firestore data/uploading files.
* To enable firebase auth, go to the firebase console, click/create the desired project, then click authentication on the left, click get started, click email and password, click enable for just the top toggle, click save.
* Now that we enable firebase auth on the backend, we need to initialize this service on the frontend
* To do so, import the getAuth function from auth
* Now, we can execute the getAuth function and store the return value in a constant. By executing the function, we initialize the authentication services and we use the constant whenever we want to do something with authentication such as sign in/out.
* Note that we still need the frirebaseConfig object and the initalizeApp method from before.

**Firebase Signing Users Up**

* We will be using username and password combinations to sign in users
* First, we need to create a form in the html to allow users to create a username and password
* A screenshot of a computer

  Description automatically generated with medium confidence
* We also need to attach an event handler when the signup form is submitted. Inside the event listener function, we will handle the logic with signing up users
* First, we need to import the createUserWithEmailAndPassword function from firebase auth.
* This function takes in 3 parameters. The first parameter is the auth object. The second argument is the email. The third argument is the password.
* Executing the function returns a promise. If the promise resolves, then the new user is signed up and the function which is passed as a parameter to the .then is executed. This function takes in a user credential object (the resolve of the promise) which contains information about the user that just signed up. Thus, we can log out cred.user
* We also need a catch block in case there is an error such as the user did not sign up with a strong enough password.
* Text

  Description automatically generated
* When we check the console after creating a user, we see:
* Timeline

  Description automatically generated with low confidence

**Logging Users In and Out**

* To log users out, we can create a button in the html which logs them out upon click.
* In the js, we need to add an event listener to the button to handle the logic.
* To sign out a user, we need to import the signOut function from firebase auth.
* This signOut function takes in the auth object as its parameter. Executing this function signs out the user and returns a promise so we can add a .then or .catch method.
* Text

  Description automatically generated
* To log users in, we need to create a form in the html which allows them to enter their login info.
* In the js, we need to add an event lister to the form to handle the logic upon form submission.
* To sign in a user, we need to import the signInWithEmailAndPassword function from firebase auth.
* This signInWithEmailAndPassword takes in 3 arguments. The first is the auth object. The second is the email. The third is the password.
* Executing this function logs the user in and returns a promise so we can add a .then or .catch method. If the promise is resolved, then the .then method takes in a function whose parameter is the user credential object of the user that just signed in.
* Text

  Description automatically generated

**Listening to Auth Changes**

* We can set up a real-time subscription to the status of the current user on the website.
* Thus, every time there is an authentication status change (user signs up, logs in, logs out), then some function is executed.
* To do so, we need to import the onAuthStateChange function from firebase auth.
* This function takes in two arguements, the first is the auth object. The second is a function that is executed each time there is an authentication status change. This function takes in a parameter which is the user credential object. If the auth status change was a user signing up, the user credential object would represent the user that just signed up. If the auth status change was a user logging in, the user credential object would represent the user that just logged in. If the auth status change was a user logging out, the user credential object would be null.
* Text

  Description automatically generated
* Notice that when we first load the page, the function is immediately fired. This is because firebase is initially looking to see if we have a user logged in.

**Unsubscribing from Changes**

* If we have subscriptions to collection data, document data, and auth data (which comes from using the onSnapshot and onAuthStateChanged functions), we should unsubscribe when we no longer need them.
* To do so, we can create a button in the html that will allow us to unsubscribe upon click.
* Note that onSnapshot and onAuthStateChanged are functions that return an unsubscribe function.
* Text

  Description automatically generated
* Text

  Description automatically generated with medium confidence
* Text

  Description automatically generated with medium confidence
* unsubCol, unsubDoc, and unsubAuth are all functions that unsubscribe when executed.
* Text

  Description automatically generated

**Cloud Storage**

* Cloud Storage for Firebase is built for app developers who need to store and serve user-generated content, such as photos or videos.
* <https://firebase.google.com/docs/storage/web/create-reference>
* <https://www.youtube.com/watch?v=-IFRVMEhZDc>
* Text

  Description automatically generated
* Text

  Description automatically generated

**Setting Up Storage**

* Create a default Cloud Storage bucket by going to the firebase console, select storage, click get started, select a location, click done.
* Cloud Storage for Firebase provides a declarative rules language that allows you to define how your data should be structured, how it should be indexed, and when your data can be read from and written to. By default, read and write access to Cloud Storage is restricted so only authenticated users can read or write data. To get started without setting up Authentication, you can configure your rules for public access. This does make Cloud Storage open to anyone, even people not using your app, so be sure to restrict your Cloud Storage again when you set up authentication. To set up public ass, go to storage, rules, then change the line to: 
* To connect your bucket to your app, have the following:
  + Text

    Description automatically generated
  + Make sure to have the storageBucket property in your config object
* Notice in the code above, we imported the getStorage function from firebase storage. This function will enable us to get the instance of storage which will enable us to later create references in the storage bucket.

**Creating References**

* Your files are stored in a Cloud Storage bucket. The files in this bucket are presented in a hierarchical structure, just like the file system on your local hard disk, or the data in the Firebase Realtime Database. By creating a reference to a file, your app gains access to it. These references can then be used to upload or download data, get or update metadata or delete the file. A reference can either point to a specific file or to a higher level node in the hierarchy.
* In order to upload or download files, delete files, or get or update metadata, you must create a reference to the file you want to operate on. A reference can be thought of as a pointer to a file in the cloud. References are lightweight, so you can create as many as you need, and they are also reusable for multiple operations.
* To create a reference, we will need to import the ref function from firebase storage which will allow us to create references. This ref function takes in the instance of storage as an argument and returns a reference to that instance of storage.
* This reference points to the root of your Cloud Storage bucket: 
* We can also point to locations lower in the tree.
  + Ex: To point to an ‘image’ folder inside the bucket: . Note that the ‘images’ folder does not already exist, it’ll be created once we actually make a change to the image folder.
  + Ex: To point to a file inside ‘image’ folder: 
* Note that creating refences doesn’t actually make any changes to the storage, we are simply pointing at locations in the storage, not changing them.

**Upload Files**

* Text

  Description automatically generated
* When the user enters a file, we get the file using: 
* To upload the file, import the uploadBytes function from firebase storage. This function takes two parameters. The first is the reference to where in the bucket a file should be added. The second is the file object. This function returns a promise with a snapshot object as it’s resolve value. This snapshot object is an object representation of the file we just uploaded I think (im not sure).
* Text

  Description automatically generated
* If it successfully uploads, we can see the image appearing in our bucket in the firebase console.
* Application

  Description automatically generated with low confidence

**Downloading Files via URL**

* To download a file via it’s url, we need to import the getDownloadURL from firebase storage. This function takes in one parameter, which is the reference to the file we want to download. This function returns a promise with an url as it’s resolve value. This url is the url of the file we downloaded.
* 
* Ex url: 

**Deleting Files**

* To delete a file, we need to import the deleteObject from firebase storage. This function takes in one parameter, which is the reference to the file we want to delete. This function returns a promise so we can add a .then method if we want.
* 

**Security Rules**

* If we failed to set up backend security rules properly, anyone could easily alter our database without permission
* By default, the firebase rules might look something like the following:
* Text, letter

  Description automatically generated
* Rules are defined in their own language which resembles JS.
* The line ‘match /databases/{database}/documents’ points to the root of our database.
* The line ‘match /{document=\*\*} ‘ points to paths in our database we want to apply rules to. In our case, by saying document=\*\*, we are matching every document in the database so all documents have to adhere to the security rules.
* We then use the allow keyword followed by the operation that we want to set a rule for. In this case, ‘allow read, write’, the operations are to read or write. If don’t have anything after the line ‘allow read, write’, anyone can read or write to it.
* We can also add a Boolean expression to apply rule logic.
* For example, if we wanted to lock down all documents so none of them could be read or written to, we could have the following:
* Text, letter

  Description automatically generated
* In the above example, if we try to query any document from the frontend, we will get an error.
* There are different types of requests that we can apply security rules to.
* Text

  Description automatically generated with medium confidence
* The get request allows us to read a specific document
* The list request allows us to read a collection query
* The create/update/delete requests allows us to create/update/delete data
* In addition, we could say allow read which combines the get and list request into a single rule. Similarly, we could also say combine write which would combine the create/update/delete requests into a single rule.
* Pointing to data
  + Now, we will learn how to point to documents/collections that we want the rules to be applied to
  + As of now, the path we have is /{document=\*\*} which tells your rules to be applied everything inside your current path. (our current path is the root of our database). This is useful when we have a rule applied to many collections such as verifying that a user is authenticated.
  + We can also apply rules to a specific document by using that document’s path
  + For example:
  + Suppose we have the following document in firestore
  + Graphical user interface, text, application

    Description automatically generated
  + We can apply a rule to it by using it’s path (highlighted in blue) of: /todos/GqQ5wmAt1GervwmQZVdDqY1PSsK2/todos/8IMuJTTpC8fLNsmHDGF5
  + Now we can apply that document’s path in our security rules (as shown on line 4):
  + Text

    Description automatically generated
  + In the above example, read and write operations are only allowed for that specific document. Nothing else can be read or accessed.
  + However, instead of hard-coding a path, we can use a wild-card and replace it with a bracket and a variable name. The value of this wild-card variable is evaluated at run time.
  + Ex:
  + Graphical user interface, text, application

    Description automatically generated
  + In the above example, any document within the /todos/GqQ5wmAt1GervwmQZVdDqY1PSsK2/todos/ collection can be read and written to.
* Authentication
  + Suppose we want any user to be able to read any document within a collection. However, suppose we also want documents within that collection to be deleted only if the user is authenticated.
  + To check if a user is authenticated, we can say ‘request.auth != null’.
  + Then the security rules would look like the following:
  + Graphical user interface, text, application

    Description automatically generated
  + However, ‘request.auth != null’ doesn’t read very well. Instead, we can create a function.
  + Graphical user interface, text, application

    Description automatically generated with medium confidence
  + Notice we created the isSignedIn function and called it in the logic for allowing the delete operation.
  + Suppose we want to determine if the user is the owner of a specific document. An example might be a user profile where other users can read the profile, but only the user owner can write to it.
  + The code could look like the following:
  + Graphical user interface, text, application

    Description automatically generated
  + Let’s say we want to determine if a user has a verified email as well. To chain statements, we need to use &&. We could also use ‘or’ instead of ‘and’.
  + Graphical user interface, text, application, email

    Description automatically generated

Existing/Incoming Data

* + Suppose we have a document in the database that has a ‘name’ and ‘price’ property as shown below:
  + Chart

    Description automatically generated with medium confidence
  + Suppose we want to read that document only if the price > 10. To do so, we need to get the existing data within the document by saying ‘resource.data’.
  + Graphical user interface, text, application

    Description automatically generated
  + Suppose we want to update a document in the database with a new ‘price’, but only if the new price is greater than the existing price. To do so, we need to get the incoming data within the request by saying ‘request.resource.data’.
  + To get the existing data, we say ‘resource.data’
  + To get the incoming data, we say ‘request.resource.data’
  + We might need existing data when you have to update the price of an item, but the price must be above some threshold.
  + Graphical user interface, text, application

    Description automatically generated

Role-based authorization

* Suppose we have the following firestore database:
* Graphical user interface, text, application

  Description automatically generated
* Graphical user interface, application

  Description automatically generated
* Suppose we want only users with a role of ‘reader’ to be able to read any document within the products collection. Suppose we want only users with a role of ‘admin’ to be able to write to any document within the products.
* Graphical user interface, text, application

  Description automatically generated
* Then the security rules would look like the above. The getUserData makes use of the get keyword. The get keyword points to a given document based on its absolute path. We write out the path like normal, but we replace wild card paths of {wildCardExample} with $(wildCardExample). The .data method added at the end reads the data within that document. So the getUserData returns the data within a given user’s document.
* We then user the getUserData function in the security rules to saying if the user has a role of reader, then they can read any product within the product collection. We also say that if the user has a role of admin, then they can read any product within the product collection.
* Ex:
* Graphical user interface, text, application

  Description automatically generated
* Since the user is not authenticated, there the getUserData function cannot locate a document since request.auth.uid is null
* However, if we try to make the same get request, but this time log in with a user with id of GqQ5wmAt1GervwmQZVdDqY1PSsK2, then we get the following:
* Graphical user interface, text, application

  Description automatically generated
* This is because in our firestore, there was a users collection with a document inside it. That document’s path was /users/ GqQ5wmAt1GervwmQZVdDqY1PSsK2. Since GqQ5wmAt1GervwmQZVdDqY1PSsK2 was the uid of the user that made the get request, we are able to go to the /users/ GqQ5wmAt1GervwmQZVdDqY1PSsK2 document and read its data which is displayed to the right (the return value of getUserData()). We then check its ‘reader’ status which is true so we successfully read the data within the /product/G9XCEcM54qRJOOYcE08G document.
* Note that if we continue with this example but try to write to the /product/G9XCEcM54qRJOOYcE08G document, we get an error since the user with uid of G9XCEcM54qRJOOYcE08G does not have admin privileges.

Time

* Suppose we want a document to be changed a limited number of times in a given time frame, say 1 time a minute. Then we can do the following:
* Graphical user interface, text, application

  Description automatically generated
* Note in order to do the above, the document must have a ‘createdAt’ property.

Other Security Stuff

* Note that to test these security rules, we can use the Rules playgroundGraphical user interface, text, application

  Description automatically generated
* Text

  Description automatically generated with medium confidence
* A picture containing text

  Description automatically generated
* Graphical user interface, text, application

  Description automatically generated

**Cloud Functions**

* Firebase is a backend as a service. Thus, we don’t have to set up our own server to build websites, enabling serverless computing.
* However, sometimes we still want to run our own server-side code. But we don’t have a server to do that on since Firebase is serverless.
* Thus, firebase provides cloud functions that allow us to write code on firebase servers. This code can interact with other firebase services such as firestore, storage, authentication.
* Each bit of code we write is packed up into a function and deployed to firebase
* One example of using cloud functions is when a new user signs up and we want to add that user to firestore. However, we don’t want the frontend to have write permissions to the database. Thus, we can write and deploy a cloud function to listen to new user sign ups and create a new database record for new users.
* Diagram

  Description automatically generated with medium confidence
* Cloud functions run in a node.js environment, meaning we can write cloud functions in js or typescript.
* Cloud functions can respond to HTTP triggers and background triggers.
* Background triggers are events that occur in background of the app.
* HTTP triggers are used to directly invoke a firebase function.
* Graphical user interface, application

  Description automatically generated

**Cloud functions setup**

* Go to firebase console, create a new project, give it a name, we can remove google analytics, and then create project.
* Now create a folder on the desktop and inside this folder in cmd run ‘npm i -g firebase-tools’. To check that this package is installed, run in cmd: ‘firebase --version’
* Then, run in cmd ‘firebase login’. If we need to logout for the future, run ‘firebase logout’
* Then, run in cmd ‘firebase init’
* Text

  Description automatically generated
* Now, we have 3 files and a ‘functions’ folder.
* Text

  Description automatically generated
* .firebaserc indicates the name of the firebase project that we’re connected to
* firebase.json declares some properties about our firebase project (we don’t have any right now)
* The ‘functions’ folder is where our cloud functions live.
* Notice the package.json file inside the ‘functions’ folder already has two dependencies, ‘firebase-admin’ and ‘firebase-functions’.

**Setting up Emulator**

* Firebase emulator is a set of advanced tools for developers looking to build and test apps locally using Cloud Firestore, Realtime Database, Cloud Storage, Authentication, Cloud Functions, Pub/Sub, and Firebase Hosting. It provides a rich user interface to help you get running and prototyping quickly.
* To initlaize the emulator, we run in cmd: ‘firebase init emulators’. We click the emulators we need (such as auth, firestore, functions) and we make sure that we set them up in the console.
* Text

  Description automatically generated
* To start the emulator, we run in cmd: ‘firebase emulators:start’
* Text

  Description automatically generated
* Notice inside the first box it gives us a link to view the emulator UI which we can click
* Inside this emulator, we can add/delete users from auth, add/delete/update items from firestore, and view function logs.

**HTTPS functions**

* To create a function, we have to create it locally. Then, we deploy the ‘functions’ folder and we deploy that to firebase so those functions sit on firebase servers.
* Recall that cloud functions use a node.js environment so the index.js will be written via node.js
* The first thing we do in this ‘index.js’ file is require the ‘firebase-functions’ module which has a return value that we can store in a constant. Let’s call that constant functions.
* Whenever we want to create a function, we will need to use the ‘functions’ constant.
* To create an endpoint request function, we create the following:
* Text

  Description automatically generated
* Here, we created a helloWorld function. We used the ‘functions’ constant, then specified that we wanted to use an HTTP trigger via the ‘https’ prop, then specified that we wanted to create a request endpoint function via ‘onRequest’. This onRequest function takes in a callback function which gets executed whenever we make a request to a certain URL (more on this URL later). This callback function takes in a request and response object as its parameters (like in node). We just simply log ‘hello world’ then respond with a ‘hello world’.
* When we save the index.js file, we get the following in the cmd:
* 
* The above shows a URL which was the URL that was previously mentioned. When we go to that URL, we get the following:
* Graphical user interface, text, application, chat or text message

  Description automatically generated
* We see hello from firebase since a the firebase servers sent a response to the webpage.
* We also see the following new messages in the cmd:
* A screenshot of a computer

  Description automatically generated with medium confidence
* We can also see the above messages when we go to the emulator, functions, then logs.
* Graphical user interface, text, application

  Description automatically generated
* For HTTP functions, we need to make sure we always respond (by always executing the ‘method’).

**Callable functions**

* A callable function is directly callable from our code.
* Text

  Description automatically generated
* Here, we created a myCallable cloud function. We used the ‘functions’ constant, then specified that we wanted to use an HTTP trigger via the ‘https’ prop, then specified that we wanted to create a callable function via ‘onCall’. The ‘onCall’ method in a callback function which gets executed whenever we call this function from elsewhere inside our code. This callback function takes in two paramters. The first is ‘data’ which represents any data that we pass to this function. The second is a ‘context’ object which has additional information available such as authentication status of the user. This callback function simply returns a string as shown above.
* We can now use this myCallable function in other parts of our code.
* Suppose we have an HTML page with firebase CDN’s as shown below:
* Text

  Description automatically generated
* Now suppose we have a button with a class of ‘call’ in that HTML file.
* Now suppose we link the following vanilla JS file to the HTML file.
* Text

  Description automatically generated
* Notice how in this function when the button is clicked, we create a reference to the cloud function and store it in a constant called myCallable. When we execute the myCallable function, we pass in a parameter of {name : ‘grant’}. We then go to the cloud function that the myCallable function references. This takes some time to reach the cloud function so myCallable is an async function. When we eventually reach the cloud function, the cloud function returns ‘hello grant’. When the cloud function finishes, the myCallable function resolves its promise so we tag on a .then. The .then gives us access to a result object and if we do result.data, we get the string ‘hello grant’ which is what the cloud function returns.
* Note that callable cloud functions can throw errors that we can catch in the .catch method.
* Note that we likely don’t want to send the actual error message since it can contain sensitive infromation
* For example, consider the following:
  + Index.js file:
  + Text

    Description automatically generated
  + Js file that is linked to html:
  + Text

    Description automatically generated

**Background functions**

* Firebase Admin:
  + When we use background functions, we need to first require ‘firebase-admin’ which is a dependency that was already installed. We can store the result of this require method in a constant called ‘admin’. We use this ‘admin’ constant to initialize our application so that we can use other firebase serves such as firestore and authentication.
* Return Value:
  + Many of these background functions are asynchronous. For these functions, we return a JavaScript promise with .then and .catch attached to it.
  + In vanilla JS functions, returning a promise with a .then/.catch would return a Pending promise since the promise is not await, but rather immediately returned.
  + Graphical user interface, text, application

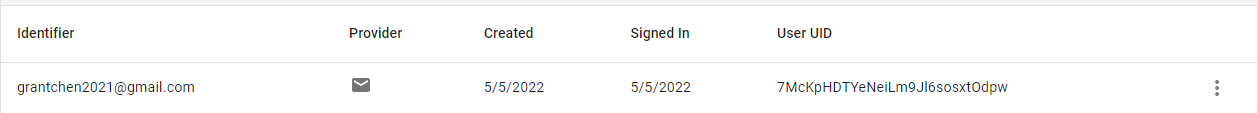
    Description automatically generated
  + However, with cloud functions, cloud functions keep running until the promise is resolved or rejected
  + Text

    Description automatically generated
  + Note that we don’t need to add a .then or .catch method if it’s not needed. The point is when we return a Promise, the promise will resolve/reject.
  + Text

    Description automatically generated
  + Note that for synchronous background functions, the we must add ‘return true’ or ‘return null’ instead of ‘return’ or not adding ‘return’ at all.

**Auth Functions**

* With auth triggers, we can run cloud functions whenever a user is added/deleted. For example, when a user is created, we might want to add their information to firestore and also send a welcome email. When a user is delete, we might want to remove their information from firestore.
* We will create a cloud function that will be invoked when a new user is added and when a user is deleted. When a new user is created, we store their email in the database. If a user is deleted, we delete the document that stored the user information.
* The code is shown below (notice the use of requiring ‘firebase-admin’):
* Text

  Description automatically generated
* Notice that for both these functions, we return a promise with a .then and .catch attached to it. Thus, the cloud function keep running until the promise is resolved/reject.
* Now, if we go to the emulator, manually add a new user as shown below:
* 
* Now, we see the following logged to the console and cmd:
* A screenshot of a computer

  Description automatically generated with medium confidence
* As well, we see the following changes to firestore (which now includes that newly created user):
* Graphical user interface, application

  Description automatically generated
* Now, we can delete that user manually as shown below:
* Graphical user interface, text, application, Word

  Description automatically generated
* Now, we will see the following logged to the console and cmd:
* A screenshot of a computer

  Description automatically generated with medium confidence
* As well, we see the following changes to firestore (which is now empty):
* Graphical user interface, application, Teams

  Description automatically generated

**Firestore Functions**

* Cloud Firestore function triggers
* The ‘firebase-functions’ module that we required from can export a functions.firestore object that allows you to create handlers tied to specific Cloud Firestore events.
* Graphical user interface, text, application, email

  Description automatically generated
* Specifying documents/collecitons:
  + Specify a single document:
    - If you want a cloud function to be invoked only when a specific document is written to, we can do the following:
    - Graphical user interface, text, application

      Description automatically generated
  + Specifying a group of documents:
    - If you want a cloud function to be invoked when any document in a specific collection is written to, we can do the following:
    - Text

      Description automatically generated
  + Note that we can also use the wildcard notation for collections as well:
    - Text

      Description automatically generated
* Event Triggers
  + Trigger a function when a new document is created
    - Text

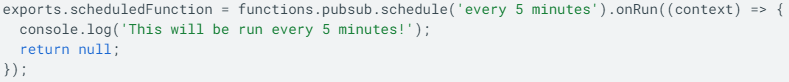
      Description automatically generated
  + Trigger a function when a document is updated
    - Text

      Description automatically generated
  + Trigger a function when a document is deleted
    - Graphical user interface, text

      Description automatically generated
  + Trigger a function for all changes to a document
    - Graphical user interface, text, application

      Description automatically generated

**Scheduled Functions**

* If you want to schedule functions to run at specified times, use functions.pubsub.schedule().onRun()
* For example, to run a function every five minutes with App Engine cron.yaml syntax, do something like this:
* 
* Both Unix Crontab and App Engine syntax are supported by Cloud Scheduler. For example, to use Crontab to select a specific timezone in which to run a scheduled function, do something like this:
* A picture containing graphical user interface

  Description automatically generated

**Storage Functions**

* <https://firebase.google.com/docs/functions/gcp-storage-events>

**Deployment**

* As of now, the cloud functions only exist locally. We need to deploy them to firebase servers.
* Go to the root folder in go to cmd
* If we want to deploy the entire project, we run in cmd: “firebase deploy”
* If we want to deploy just the cloud functions, we run in cmd: “firebase deploy --only functions’
* If we want to deploy just a specific cloud function, we run in cmd: “firebase deploy --only functions:CLOUD\_FUNCTION\_NAME”
* If we get an error, make sure we are on the blaze pricing plan.
* As well, if we included storage as one of the cloud function triggers, make sure to run ‘firebase deploy’
* When the functions are finished deploying, we should be able to see them in the functions section of the firebase dashboard:
* Graphical user interface, text, application

  Description automatically generated
* We should also get the following in cmd:
* Text

  Description automatically generated
* To remove a cloud function, we can do remove it via the dashboard or by redeployment
* Dashboard:
  + Graphical user interface, text, application

    Description automatically generated
* Redeployment:
  + Below we comment out the updateUser function:
  + Text

    Description automatically generated
  + We then deploy our application again and enter ‘yes’ to the prompt to proceed with deletion.
  + Text

    Description automatically generated