10. Realtime Updates in Firebase

Last time we looked at how to create (`addDoc()`), read (`getDoc()`, `getDocs()`), update (`setDoc()`, `updateDoc()`), and delete (`deleteDoc()`) data from Firebase. We also saw that when doing CRUD with Firebase, we need to take care to manage our in-application data structures as well as Firebase data.

Recall, for example, how we handled adding a person to our “person list”:

```
async function addPerson(firstName, lastName) {
  const collRef = collection(db, 'people');
  let personObj = {
    firstName: firstName,
    lastName: lastName
  }; // leave the key out for now
  let docRef = await addDoc(collRef, personObj);
  personObj.key = docRef.id;
  let pList = Array.from(peopleList);
  pList.push(personObj);
  setPeopleList(pList);
  setFirstNameInput('');
  setLastNameInput('');
}
```

In this example, lines 2-7 are all about managing data in Firebase (adding a person to the ‘people’ collection). Lines 8-11 are about managing in-application data (aka “state”). In line 8, we need to do a funny little dance to make sure that the keys we’re using in Firebase are the same as the keys we’re using in our application. This is important, because we need those keys to be in sync when we do delete:

```
async function deletePerson(person) {
  const docRef = doc(db, "people", person.key);
```

```
    await deleteDoc(docRef);
    let pList = Array.from(peopleList);
    let idx = pList.findIndex((elem)=>elem.key === person.key);
    pList.splice(idx, 1);
    setPeopleList(pList);
  }
```
(lines 2-3 are about Firebase, lines 4-7 are about state)

and update:

```
  async function updatePerson(key, firstName, lastName) {
    const docRef = doc(db, "people", key);
    await updateDoc(docRef, {firstName: firstName, lastName: lastName});
    let pList = Array.from(peopleList);
    let idx = pList.findIndex((elem)=>elem.key === key);
    pList[idx].firstName = firstNameInput;
    pList[idx].lastName = lastNameInput;
    setPeopleList(pList);

    // reset ui
    setFirstNameInput('');
    setLastNameInput('');
    setMode('add');
    setSelectedItemKey('none');
  }
```
(lines 2-3 are about Firebase, lines 4-8 are about state. In both `deletePerson()` and `updatePerson()`, note the important role played by the `key` associated with each person.)

This approach works fine for single-client apps (like our List Maker series), where the database effectively acts as persistent storage for a single-user app. When we start looking at multi-user applications, we'll have to do things a bit differently. In particular we'll have to *listen for realtime updates* from specific Firebase entities, such as documents or collections. To look at this, let's start with a basic Message Board app that is not multi-user (yet). In fact it's not even connected to Firebase (yet).

The `week10MessageBoard` project you should have cloned (at the last minute) will be our starting point. Walking through our starting point, we start with our imports and Firebase setup—just the standard recipe:

```
import React, { useEffect, useState } from 'react';
import { Button, FlatList, StyleSheet, Text, TextInput, View }
from 'react-native';
import { initializeApp, getApps } from 'firebase/app';
import {
  initializeFirestore, collection, getDocs, query,
  doc, addDoc, getDoc, onSnapshot
} from "firebase/firestore";
import { firebaseConfig } from './Secrets';

let app;
if (getApps().length == 0){
  app = initializeApp(firebaseConfig);
}
const db = initializeFirestore(app, {
  useFetchStreams: false
});
```

One new wrinkle here is

```
let app;
if (getApps().length == 0){
  app = initializeApp(firebaseConfig);
}
```

… which can help prevent some annoying errors that are thrown when `initializeApp()` is run more than once. Firebase doesn't seem to like this.

Next, we set up our state variables and stub out a `useEffect()`, which we'll need soon enough.

```
export default function App() {
  const [inputText, setInputText] = useState('');
  const [authorText, setAuthorText] = useState('');
  const [messages, setMessages] = useState([]);

  useEffect(()=>{

  });
```

And then our UI, which starts with two text fields for a "message" and an "author"…

```
  return (
    <View style={styles.container}>
      <View style={styles.inputContainer}>
        <Text>Message:</Text>
        <TextInput
          style={styles.inputBox}
          value={inputText}
          onChangeText={(text)=>setInputText(text)}
        />
      </View>
      <View style={styles.inputContainer}>
        <Text>From:</Text>
        <TextInput
          style={styles.inputBox}
          value={authorText}
          onChangeText={(text)=>setAuthorText(text)}
        />
      </View>
```

... a button, which adds the message to the list of `messages` when pressed:

```
      <View>
        <Button
          title="Send"
          onPress={()=>{
            setMessages(oldMessages=>{
              let newMessages = Array.from(oldMessages);
              let ts = Date.now();
              newMessages.push({
                author: authorText,
                text: inputText,
                timestamp: ts,
                key: '' + ts
              });
              return newMessages;
            });
            setInputText('');
          }}
        />
      </View>
```

... and finally a place to display all of the messages that have been posted:

```
      <View style={styles.messageBoardContainer}>
        <FlatList
          data={messages}
          renderItem={({item})=>{
            return (
              <View style={[
                styles.messageContainer
              ]}>
                <Text style={styles.messageText}>
                  {item.author}: {item.text}</Text>
              </View>
            );
          }}
        />
      </View>
    </View>
  );
}
```

At the bottom is some fairly uninspired styling that we don't need to get into.

Let's review what we learned last week.

Now You Try

Modify the behavior of the "Send" button so that new messages are not just added to the `messages` state list, but also to the Firebase collection `messageBoard`. You may want to refer back to Week 9's lecture for a refresher. This should work even if the `messageBoard` doesn't exist yet (`addDoc` will create the collection if needed). The structure of the `message` you add to Firebase should be:

```
{
   author: XXXX,
   text: YYYY,
   timestamp: new Date()
}
```

remember that you need to retrieve the `id` that Firebase creates and add it as a `key` to your `message` after creation. Verify that your new messages are correctly added to your Firestore database. Paste your `onPress` handler function into slido.

Did you get an error message like “unexpected reserved word ‘await’”? Do you remember what you need to do to solve that?

Add a few messages, make sure they’re in Firebase, then reload the app. What happened to your messages? I guess you need to load them on startup! Try that next…

```
          onPress={async ()=>{
            let newMsg = {
              author: authorText,
              text: inputText,
              timestamp: new Date()
            };
            let newDocRef = await addDoc(collection(db, 'messa
geBoard'), newMsg);
            newMsg.key = newDocRef.id;
            setMessages(oldMessages=>{
              let newMessages = Array.from(oldMessages);
              newMessages.push(newMsg);
              return newMessages;
            });
            setInputText('');
          }}
```

Now You Try

Modify `useEffect()` so that it loads all of the messages from the `messageBoard` collection into the `messages` state list. Again, refer to Week 9’s lecture if needed. Remember to make sure `useEffect()` only runs once. Test this by reloading the app to find your long-lost messages. Add a few more and reload again to make sure it’s really working. Recall that you can’t make your `useEffect` callback function `async`, so you’ll need to define a function-within-a-function and then call it, all within `useEffect` (as we did last week).

Paste your revised `useEffect()` into slido.

Checkpoint: https://github.com/SI669-internal/week10MessageBoard/tree/staticFirebase

Realtime Updates

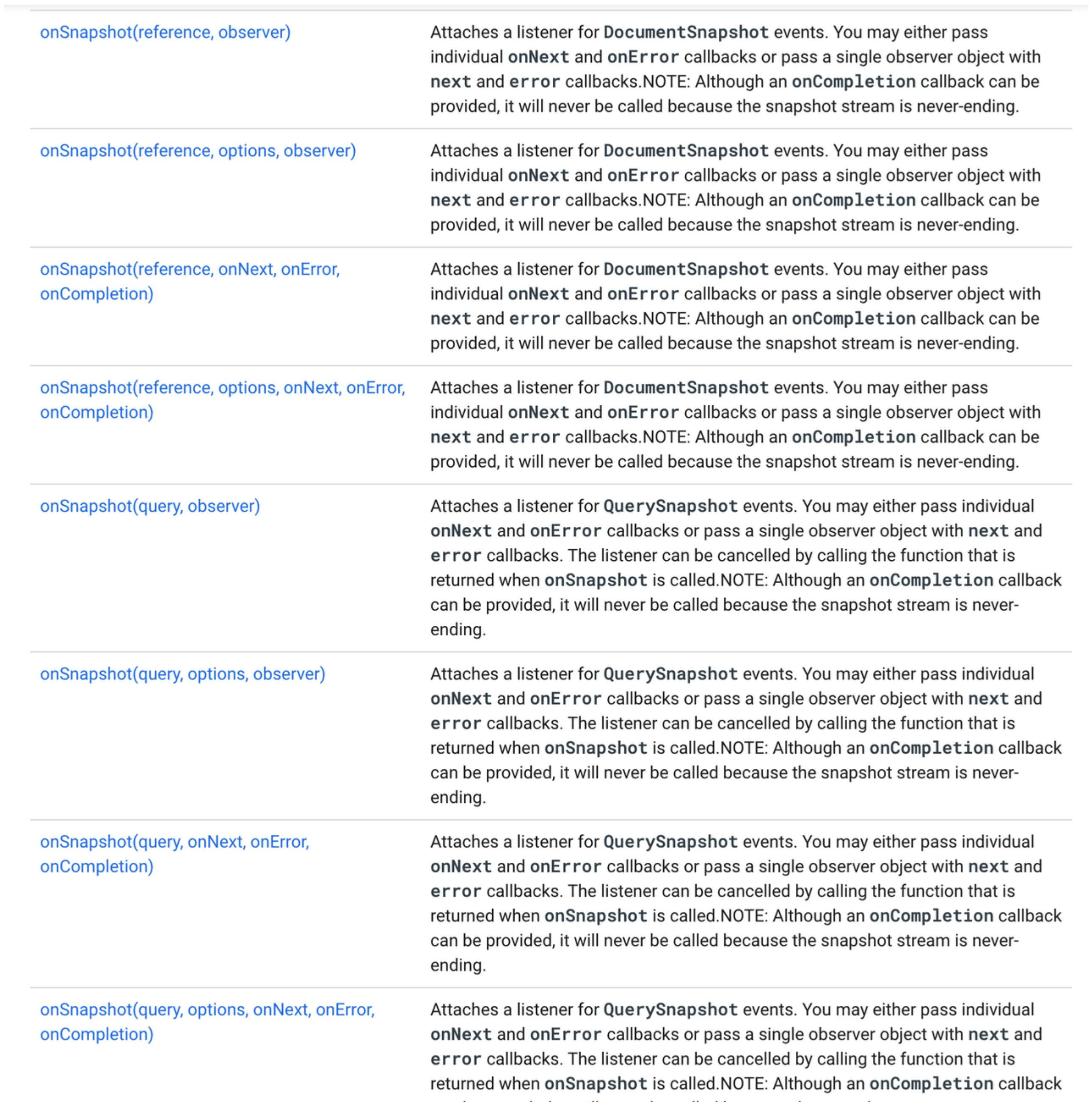
多用户协同交互数据库

Now let's try an experiment. Get together with a partner, and both run the *same app.* You can do this by scanning one partner's QR code. This will also work if one partner is using a simulator and the other is using a phone, or, if you prefer to go it alone, if you have both a simulator and a phone.

Both of you add messages. What you should see is that both partners' messages show up in Firebase but each partner can only see their own messages in their own app. When you refresh, you should both see everyone's messages. But why can't you see your partner's messages when they are posted?

You can probably see why—you are only retrieving data from Firebase on startup, and only adding data after that (i.e., not reading it). To be able to read up-to-date data from a Firestore document or collection, you need to *listen for realtime updates.* This can be done with the `onSnapshot()` function.

This is a top-level Firestore function that comes in many flavors. So many, in fact, that I couldn't fit them all in a single screenshot (see them all at https://firebase.google.com/docs/reference/js/firestore_):



onSnapshot(reference, observer)	Attaches a listener for `DocumentSnapshot` events. You may either pass individual `onNext` and `onError` callbacks or pass a single observer object with `next` and `error` callbacks.NOTE: Although an `onCompletion` callback can be provided, it will never be called because the snapshot stream is never-ending.
onSnapshot(reference, options, observer)	Attaches a listener for `DocumentSnapshot` events. You may either pass individual `onNext` and `onError` callbacks or pass a single observer object with `next` and `error` callbacks.NOTE: Although an `onCompletion` callback can be provided, it will never be called because the snapshot stream is never-ending.
onSnapshot(reference, onNext, onError, onCompletion)	Attaches a listener for `DocumentSnapshot` events. You may either pass individual `onNext` and `onError` callbacks or pass a single observer object with `next` and `error` callbacks.NOTE: Although an `onCompletion` callback can be provided, it will never be called because the snapshot stream is never-ending.
onSnapshot(reference, options, onNext, onError, onCompletion)	Attaches a listener for `DocumentSnapshot` events. You may either pass individual `onNext` and `onError` callbacks or pass a single observer object with `next` and `error` callbacks.NOTE: Although an `onCompletion` callback can be provided, it will never be called because the snapshot stream is never-ending.
onSnapshot(query, observer)	Attaches a listener for `QuerySnapshot` events. You may either pass individual `onNext` and `onError` callbacks or pass a single observer object with `next` and `error` callbacks. The listener can be cancelled by calling the function that is returned when `onSnapshot` is called.NOTE: Although an `onCompletion` callback can be provided, it will never be called because the snapshot stream is never-ending.
onSnapshot(query, options, observer)	Attaches a listener for `QuerySnapshot` events. You may either pass individual `onNext` and `onError` callbacks or pass a single observer object with `next` and `error` callbacks. The listener can be cancelled by calling the function that is returned when `onSnapshot` is called.NOTE: Although an `onCompletion` callback can be provided, it will never be called because the snapshot stream is never-ending.
onSnapshot(query, onNext, onError, onCompletion)	Attaches a listener for `QuerySnapshot` events. You may either pass individual `onNext` and `onError` callbacks or pass a single observer object with `next` and `error` callbacks. The listener can be cancelled by calling the function that is returned when `onSnapshot` is called.NOTE: Although an `onCompletion` callback can be provided, it will never be called because the snapshot stream is never-ending.
onSnapshot(query, options, onNext, onError, onCompletion)	Attaches a listener for `QuerySnapshot` events. You may either pass individual `onNext` and `onError` callbacks or pass a single observer object with `next` and `error` callbacks. The listener can be cancelled by calling the function that is returned when `onSnapshot` is called.NOTE: Although an `onCompletion` callback

The upshot is that you can listen for updates on lots of Firebase objects (Document/Collection References, Queries), and can provide a number of options, callbacks, etc. To get the basic idea, we'll focus on the simplest (and I would argue most useful) version:

```
onSnapshot(query, onNext, ...)
```

实时监听query对应的数据库对象，每次发现更新后调用回调函数

In this version, we provide a `query` that we want to monitor (which can be as simple as a `collection()`) and a callback function that will run whenever there's an update (called here `onNext`, to indicate that it is called whenever the "next" update comes).

For our purposes, we'll want to invoke `onSnapshot()` to monitor the `messageBoard` collection, and we'll want to provide a callback handler that updates our `message` state variable with the newest version of the `messageBoard` collection's contents. To wit:

```
    onSnapshot(collection(db, 'messageBoard'), (qSnap) => {
      // deal with the update
    });
```

How should we "deal with the update"? A simple (though perhaps not optimal, if you're dealing with a very large collection) solution is to simply create a new list from the QuerySnapshot that is provided to our callback (here called `qSnap`), and update our `messages` state variable with the new list. This will refresh the entire message board list, and ensure that we have the latest, greatest data from the `messageBoard` collection. How we do this is essentially identical to how we process the `QuerySnapshot` when we are loading messages at startup, so see if you can fill in the body of the `onSnapshot()` callback with the code that will refresh `messages` whenever there's an update.

Now You Try

Paste your revised `useEffect()` into slido.

Notes:

- It turns out `onSnapshot()` runs immediately after it's called to provide the beginning state of the Document, Collection, or Query specified. Do you still need to call `getDocs()` at startup?
- What happens when you add a new message in the "Send" button's `onPress` handler? Is there any code there that could be cleaned up?

Checkpoint: https://github.com/SI669-internal/week10MessageBoard/tree/firebase

Announcements

- HW5 out now. Last Homework!
- Final Projects
  - I'm excited!
  - My goal: feedback by next Weds (giving you ~10 days to work on your Plan)
- Last ~1 hour today: work time & "office hours" for Proj 2 and HW 5

~~~~~~ **Take a break** ~~~~~~

((((((((((((((((((((((((((((((((((((App of the Week))))))))))))))))))))))))))))))))))))

👪👩‍🦽👨‍👨‍👦💑👩‍🚀🧎‍♂️🐋👩‍👩‍👧‍👧💆 **Theme: Social Connection** 😾 👨‍👧‍👦👩‍👧‍👧🤷 🧞‍♀️💑🐶 🐤 👨‍👩‍👧‍👧💑

vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv Zhijie Zhao & Jiayao Wu vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

^^^ Qi Sun & Shijie Gao. ^^^

Firebase Queries

While Firebase is a "NoSQL" database, it does have some SQL-like qualities, particularly when it comes to specifying queries and their parameters. The `query()` function can be used to construct queries using `where`, `orderBy`, and `limit` statements. Each of these "statements" are implemented as functions in their own
~~~~~~

right (examples soon), and you can add more or less arbitrary numbers of them to a `query()` definition, simply by adding more and more arguments. If there are multiple `where()` arguments, they are treated as if they were `AND`ed together. Note that you will need to add `where`, `orderBy`, and `limit` to your imports from `'firebase/firestore'` in order to use them. Some examples:

```
// find all messages authored by Bob
let q = query(collection(db, 'messageBoard'),
              where('author', '==', 'Bob'));

// find all messages authored by Alice OR Bob
let q = query(collection(db, 'messageBoard'),
              where('author', 'in', ['Alice','Bob']));

// find all messages by Alice or Bob posted after Nov. 1
let q = query(collection(db, 'messageBoard'),
              where('author', 'in', ['Alice','Bob']),
              where('timestamp', '>', new Date('Nov 1, 202
1'));
```

You may have noticed that the ordering of messages in your Message Board can get scrambled sometimes. This is because the ordering returned by Firestore can be arbitrary (it also might be because you and your partner's device clocks are not synced, but that's another whole Ball of Wax). To get control of the order of your messages, you can use `orderBy()`:

```
  useEffect(()=>{
    const q = query(collection(db, 'messageBoard'),
                    orderBy('timestamp', 'desc'));
    onSnapshot(q, (qSnap) => {
      let newMessages = [];
      qSnap.docs.forEach((docSnap)=>{
        let msg = docSnap.data();
        msg.key = docSnap.id;
        msg.timestamp = msg.timestamp.toDate();
        newMessages.push(msg);
      });
      setMessages(newMessages);
    });
  }, []);
```

To make sure that the messages are ordered correctly, you can print out the timestamp information in your Message Board:

```
        <FlatList
          data={messages}
          renderItem={({item})=>{
            console.log(item);
            return (
              <View style={[
                styles.messageContainer
              ]}>
                <Text style={styles.messageText}>
                  {item.author}: {item.text}
                  <Text style={{fontSize: 9}}> (
                  {item.timestamp
                    .toLocaleDateString('en-us', {
                      month:"numeric",
                      day:"numeric",
                      hour:"numeric",
                      minute:"numeric",
                      seconds: "numeric"
                    })}
                  )</Text>
                </Text>
```

```
               </View>
             );
           }}
         />
```

Now You Try

You can also `limit()` the number of results returned by your `onSnapshot()` query. Modify your query so that you only see the *three most recent messages* on your message board. Paste your revised `query()` function call into slido.

Final checkpoint: https://github.com/SI669-internal/week10MessageBoard/blob/queries/App.js