1. Promise用法,接受回调函数，继续返回promise，可以连锁
    A. Let promise = function_return_promise();
    B. promise.then(res=>{xxx}).then(res=>{xxx})
2. async, await
    A. res = await promise; 可以直接获得promise的结果
    B. async function my_function();
        a. 只有在async修饰的function里才可以调用await
        b. async function返回的对象也是promise
        c. async function可以紧跟在await之后
3. firebase开发
    A. firebase是google基于web的noSQL数据库,类似mongoDB
        a. document,类似一个json,有唯一ID(自动生成)和各个field
        b. collection,类似一个表,可以存放很多document，也有唯一ID(手动指定)
    B. 开发步骤
        a. npm install firebase
        b. 从官网copy firebase config并放在secret.js中等待导出
        c. .gitignore中添加secret.js防止泄露
        d. 在App.js开始处初始化

09. Promises + Async/Await, Firebase

Promises

As you've noticed by now, React Native, and JavaScript more generally, and interactive applications more generally use a lot of callbacks. We've mostly seen these for handling user interface events such as `onClick` or `onChangeText`, but we've also used them with timers to run code asynchronously. Once you start dealing with apps that have both a client and server component (as it the case with essentially *all* web apps), you start having to deal with operations that could take a long time (relatively speaking, in computer terms), and so you start having to deal with more and more asynchrony.

Traditionally these operations would be done with callbacks, too, using a pattern like

```
doSomethingLengthy(dataToSend, callbackWhenItsDone);
```

As you've noticed it's common to define callbacks inline, like so:

```
doSomethingLengthy(dataToSend, (dataFromOperation) => {
  // do stuff after something lengthy operation ends
});
```

But many developers have found that it's not uncommon to have chains of operations where each step in the chain needs to wait until the previous one completes. Here's a pseudo-example:

```
authenticateUser(username, password, (userInfo) => {
  getUserRole(userInfo, (userRole) => {
    getUserData(userInfo, userRole, (data) => {
      // do something with the data
      // and so on
      // and so on...
    }
  }
```

回调链非常不优雅

```
}
```

While we haven't seen this situation and, frankly, we might not see it much this semester, but it's a thing and as a result the People Who Make JavaScript developed a new and honestly pretty clever technique for handling asynchronous "do this THEN do that," the *Promise.*

The basic idea is that a function can return a `Promise` rather than a conventional result (e.g., an Object, Array, string, ...). A Promise is a kind of "wrapper" around the data returned from a function that only becomes available when the data is available —however long that might take.

The general pattern for using a `Promise` is as follows:

```
let aPromise = functionThatReturnsAPromise();
aPromise.then((functionReturn) => {
  // do something with functionReturn
  // and/or do other stuff
}
```

Let's look at a more concrete example. For the next several examples, cd into your `week9PromiseAsync` directory and create the file `fetchDataMuseSWAPI.js` to try things out.

Fetching data from a server via HTTP can take a while. In the olden days (before Promises), a fetch operation would required a callback and have looked something like this (note that this code won't work with `node-fetch`, since it doesn't use callbacks, but Promises):

```
let url = 'https://api.datamuse.com/words?rel_rhy=forgetful';
fetch(url, (result) => {
  // extract data from result
  // do something with it
});
```

With Promises, a fetch operation looks like this (put this into a new file called `fetchDataMuseSWAPI.js` and try it out via `$ node fetchDataMuseSWAPI.js`:

```
import fetch from 'node-fetch';

function fetchDataMusePromise1() {
  let url = 'https://api.datamuse.com/words?rel_rhy=forgetfu
l';
  let thePromise = fetch(url);
  thePromise.then((result) => {
    console.log(result); // try to do something with the data
  });
}

fetchDataMusePromise1();
```

回调函数

Inspecting the result tells us that perhaps we need to do a bit more work to get the actual data we're working for:

```
Response {
  size: 0,
  [Symbol(Body internals)]: {
    body: Gunzip {
      _writeState: [Uint32Array],
      _readableState: [ReadableState],
      _events: [Object: null prototype],
      _eventsCount: 6,
      _maxListeners: undefined,
      _writableState: [WritableState],
      allowHalfOpen: true,
      bytesWritten: 0,
```

```
      _handle: [Zlib],
      _outBuffer: <Buffer 5b 7b 22 77 6f 72 64 22 3a 22 66 72
65 74 66 75 6c 22 2c 22 73 63 6f 72 65 22 3a 33 39 38 2c 22 6e
75 6d 53 79 6c 6c 61 62 6c 65 73 22 3a 32 7d 2c 7b ... 16334 m
ore bytes>,
...
```

It turns out that the `result` that is provided as the payload of the Promise returned by `fetch()` is a `Response` object (a custom object that is part of the `node-fetch` package), and the `Response` itself is a `Promise` that resolves to the actual data (which can be accessed as `json` or `text`).

```
function fetchDataMusePromise2() {
  let url = 'https://api.datamuse.com/words?rel_rhy=forgetfu
l';
  let thePromise = fetch(url);
  thePromise.then((result) => {
    return result.json(); // this returns a Promise
  })
  .then((jsonData) => {   // this runs when the result.json()
Promise "resolves"
    console.log(jsonData); //deal with data
  });
}

//fetchDataMusePromise1();
fetchDataMusePromise2();
```

And now we see what we were looking for:

```
[
  { word: 'fretful', score: 398, numSyllables: 2 },
  { word: 'regretful', score: 302, numSyllables: 3 },
```

```
  { word: 'threatful', score: 129, numSyllables: 2 },
  { word: 'netful', score: 28, numSyllables: 2 },
  { word: 'let phil', numSyllables: 2 },
  { word: 'met phil', numSyllables: 2 },
  { word: 'set fill', numSyllables: 2 }
]
```

We can tidy up our code a bit, and you'll often see Promise code written more like this:

```
function fetchDataMusePromise3() {
  let url = 'https://api.datamuse.com/words?rel_rhy=forgetful'
  fetch(url)
    .then(result => result.json())
    .then(res => console.log(res));
}

//fetchDataMusePromise1();
//fetchDataMusePromise2();
fetchDataMusePromise3();
```

这步因为转JSON也是异步的

Let's consider what this would look like with regular callbacks. It would have to work something like this:

```
let url = 'https://api.datamuse.com/words?rel_rhy=forgetful';
fetch.(url, (result) => {
  result.json((data) => { //it's a callback inside a callback!
    // finally deal with data
  });
});
```

Following the chain of callbacks is a bit challenging, but honestly following Promise chains isn't that much more intuitive. So switching to Promises cleans things up a bit, but doesn't seem like a huge win does it? We will shortly see examples of

"Promise chains," which are sequences of calls that all return promises, wherein each stage of the chain has to wait for the previous stage to finish before doing its job. But the story doesn't stop with Promises—it continues onto a programming construct that is legitimately more comprehensible and manageable: `async/await`.

Async/Await

Promises were introduced to address the problem of "callback hell," and, while they don't really save any keystrokes compared to the callback approach, many developers feel that they help with readability and maintainability. There are also advantages in terms of error handling but we're not going to get into them here (if you find yourself working with Promises a lot in the future, I would recommend spending more time learning about their ins and outs—we are just scratching the surface here).

Even so, Promises are still quite callback-y, in the sense that you don't get to write code in the conventional function call-and-return style that developers are most comfortable with. For example, you still can't write

```
let myJson = fetch(url).json();
```

But have to write something like this, which still won't work like "normal" procedural code:

```
function fetchDataMusePromise4() {
  let url = 'https://api.datamuse.com/words?rel_rhy=forgetfu
l';
  let myJson = {};
  let thePromise = fetch(url);

  thePromise.then((result) => {
    return result.json(); // this returns a Promise
  })
  .then((jsonData) => {   // this runs when the result.json()
Promise "resolves"
    //deal with data
    myJson = jsonData;
```

```
  });
  console.log(myJson); // won't work!
}

//fetchDataMusePromise1();
//fetchDataMusePromise2();
//fetchDataMusePromise3();
fetchDataMusePromise4();
```

In addition to still being pretty cumbersome, you still couldn't use myJson outside of the callback right after calling `fetch()` because it would still be assigned to `{}` until sometime later when both of the Promises (from `fetch()` and `result.json()`) resolve (or complete).

To facilitate synchronous-style programming while still supporting asynchrony, the keywords `async` and `await` were introduced. `await` effectively allows you to call functions that return Promises as if they were regular functions. `await` pauses execution at the line where it is declared and waits for the Promise to resolve, returning the Promise result as the return value of the function. For example:

```
function fetchDataMuseAsync () {
  let url = 'https://api.datamuse.com/words?rel_rhy=forgetfu
l';
  let result = await fetch(url);
  let myJson = await result.json();自动等待promise运行完
  console.log(myJson);
}

//fetchDataMusePromise1();
//fetchDataMusePromise2();
//fetchDataMusePromise3();
//fetchDataMusePromise4();
```

```
fetchDataMuseAsync();
```

To use this approach, you simply put `await` in front of a function call, where the function returns a Promise. The rest is handled for you.

But there's one more thing—you can only use `await` inside of a function that is declared using the `async` keyword. This indicates to the rest of the code that the function will run asynchronously. It *also* forces your function to return a `Promise` instead of whatever you define as the return value. Your return value will be packaged inside a `Promise` and then provided to the caller via `then()`, as we have seen before. To make the `await` example above work, we need to wrap it in an `async` function, like so:

```
async function fetchDataMuseAsync () {
  let url = 'https://api.datamuse.com/words?rel_rhy=blue';
  let result = await fetch(url);
  let myJson = await result.json();
  console.log(myJson);
}

//fetchDataMusePromise1();
//fetchDataMusePromise2();
//fetchDataMusePromise3();
//fetchDataMusePromise4();
fetchDataMuseAsync(); 可以继续调用then
```

Now You Try

For this exercise, let's first look at how we would use the Star Wars API with `fetch()` and `Promises` to get information about a character's name and species. Information about a character can be retrieved from `https://swapi.dev/api/people/{id}/` where `{id}` is the character's ID within SWAPI. The character with id=2 is C-3PO, and that's who we'll use for our example.

Use your browser to look at the data for C-3PO: `https://swapi.dev/api/people/2/`, and note where to find the character's name

and species information. Note also that `species` is an array of URLs, which means you need to fetch the url at `json.species[0]` to get information about the character's species (such as the species name).

```
function fetchSWAPIPromise() {
  let url = 'https://swapi.dev/api/people/2/';
  let charName, speciesName = '';
  fetch(url)
    .then((result) => {
      return result.json(); // this is a Promise
    })
    .then((json) => {
      charName = json.name;
      return fetch(json.species[0]); // also a Promise
    })
    .then((result) => {
      return result.json(); // guess what? A Promise!
    })
    .then((json) => {
      speciesName = json.name; // finally not a Promise! We're
done!
      console.log(charName, 'is a', speciesName);
    })
}
fetchSWAPIPromise();
```

Your job is to write the function `fetchSWAPIAsync()` using the `await` keyword to have the equivalent functionality as `fetchSWAPIPromise()`. Fill in the missing code:

```
async function fetchSWAPIAsync() {
  let url = 'https://swapi.dev/api/people/2/';
  let charName, speciesName = '';
```

```
  // MISSING CODE

  console.log(characterName, 'is a', speciesName);
}
fetchSWAPIAsync();
```

Now You Try #1

Write and test your solution for MISSING CODE. Paste your solution into slido.

Solution: https://github.com/SI669-internal/week9PromiseAsync

Persistent Data

You may have noticed that so far all the data we enter into the apps we build disappears when the app restarts. This is because all the data we have created has existed only in our apps' memory, and not on any kind of persistent storage. *Data persistence* is the ability to store data "permanently" such that it survives across app sessions. You're probably already familiar with ways to store application data in an (effectively) permanent way, such as databases and data files (e.g., csv, json, etc.). We're going to look at how to use a particular kind of database—a Cloud Storage service—to manage our application data persistently.

In most modern applications, data is stored on a server connected to the Internet. Each client app connects to the "application server" over the net and uses it to fetch and store the data it needs. Some apps will also have "local storage" (i.e., a small database or data file on the mobile device itself) for certain types of data—but usually only data that it wouldn't be disastrous to lose (like, say, replaceable login credentials or cached data that could speed up app loading), but it's probably pretty rare to see any kind of app these days that doesn't use some kind of server on the backend.

Most application servers do a lot more than just store and fetch data. Any application logic that takes advantage of data from multiple people, such as recommendation systems or social network feed aggregators, pretty much *need* to

run on the backend, because no single client app would have all the information it would need to operate. Some operations, like fading out a button when it's being pressed by the user, pretty much *need* to run on the client app, or the "front end," because it would take too long to send the signal to the server and get the response. By the time the animation was rendered, the user's finger would be long gone. For many other bits of application logic, it may be possible to implement them on the frontend *or* the backend. We won't get into the nuances of when to put what functionality where (frontend/backend), but it's worth knowing that this is a conversation that often needs to happen.

Backend-as-a-service

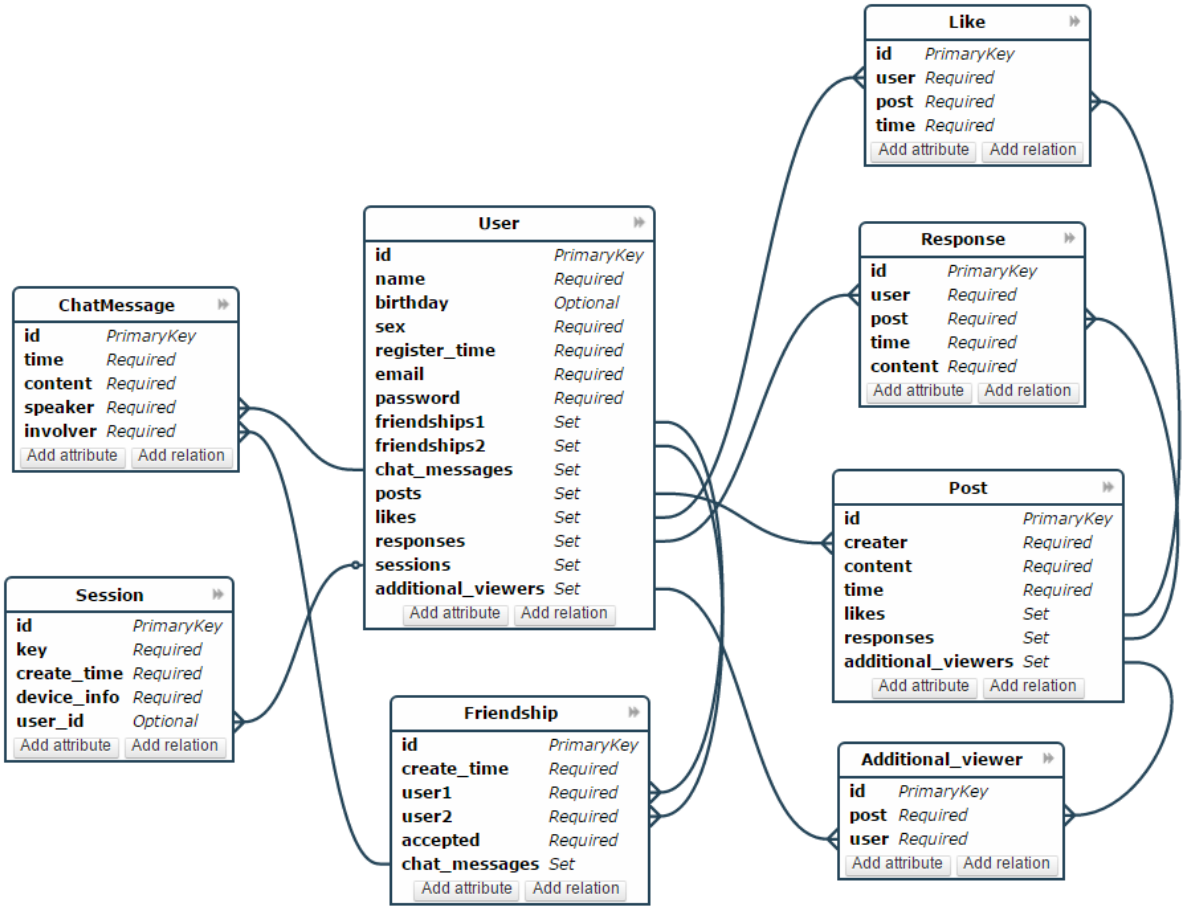
In this course, we are not really going to get into backends at all. We're just going to use one kind of backend, and we're just going to use it "out-of-the-box" without doing much customization at all. We're going to use a type of backend that is sometimes called "Backend-as-a-Service" (or BaaS), which is essentially a generic *Backend* (i.e., it provides basic services like data storage that can be used by all different kinds of applications with little to no modification) that is provided as a *Service,* meaning that a service provider (like, in our case Google) actually runs the server and takes care of most of the painful aspects of running application servers like storing data securely and robustly, scaling bandwidth and storage when needed, and keeping the backend up and running and hacker-free 24 hours a day, 7 days a week, 365 days a year.

BaaS doesn't work for all types of applications (as noted above, some types of function basically require a custom-built backend), but it works for a lot of applications, and it will work just fine for everything we're going to do in this class.

NoSQL

Another concept to cover before we dive into Firebase is the concept of "NoSQL Databases". Many of you have probably worked with SQL databases before. They're great, and they are exactly what you need in lots of cases. They are particularly useful when you are dealing with data that is highly *relational*, that is lots of different concepts linked together that are needed to work in concert for your application.

Here is an example of a relational model for what appears to be a social networking application, that has been turned into a database model.



SQL databases make it relatively easy for you to specify these of relationships (e.g., all **Post**s have exactly one **creater** (who is a **User**) and a *Set* each of **likes, responses,** and **additional_viewers**. Each of these linked entity types (**Like, Response,** and **Additional_viewer**) have their own fields, including links to other entities, and so forth.

SQL databases also give you a handy language for getting relational data out of a database efficiently and concisely, by issuing queries like

```
  SELECT * FROM ChatMessage JOIN User ON speaker = User.id WHE
RE User.name = "Mark"
```

But for some applications, SQL is overkill. There are three main arguments that are usually advanced for using NoSQL over SQL: scalability, flexibility, and simplicity.

- Scalability means it is easier to redistribute a NoSQL database over different physical machines once the data gets too big to fit on whatever machines it used to fit on. Since the different batches of data are purposefully set up to be unrelated, there are fewer constraints on where you can move different parts of the database if it turns out you need to.
- Flexibility means that you can more easily change your mind about how the data is modeled and stored. With SQL, you have to define the data model (i.e., the tables, rows, primary keys, foreign keys, etc.) up front and it can be very cumbersome to redefine the model once you've started building significant chunks of your application. Many NoSQL databases basically store all of the data as JSON objects. Changing how you want to represent your data is as easy as changing the format of the JSON objects (or Python/JavaScript objects, which already map nicely onto JSON) you produce. That's not necessarily effort-free, but it's generally less effort than "migrating the database" which is what often needs to happen if you change the design of a SQL database that already has data in it.
- Simplicity is the third advantage claimed for NoSQL. If you know JSON, you know NoSQL. And everyone knows JSON nowadays, right? No special skills (e.g., writing SQL statements) required!

Introducing Firebase

Firebase is Google's BaaS offering. Firebase started as separate company in 2011 and was acquired by Google in 2014 for an undisclosed amount of money. According to some sources, there are more than a million developers and thousands of apps using Firebase, including companies like Instacart, Twitch, Alibaba, CBS, and NPR (what specifically they are using it for, I don't know). Anyway, it's a serious piece of infrastructure, but it's also pretty easy to get started with, as you hopefully saw when you prepped for this week.

Firestore

Firestore is Firebase's NoSQL database. Actually it's *one of* Firebase's NoSQL databases—the other is called "Realtime database." The differences between the two are pretty subtle and not really relevant to us at this point, but since Firestore is newer and has more exciting adjectives associated with it ("scalable!", "available!", "offline!", "advanced!") we'll go with it. There are a few flavors of NoSQL database running around out there, but one of the most popular is the "Document Database,"

of which Firestore is an example. The core idea is that all data items are stored as "documents," which you can more or less think of as JSON objects. Here is an example of a document you might find in a Firestore database (represented as JSON):

```
'person1': {
  'firstName': 'Mark',
  'lastName': 'Newman'
}
```

In this example, `person1` is the unique ID of the document, and `firstName` and `lastName` are fields (aka 'keys' or 'properties') of the document.

Documents are stored within *Collections*, which are simply groupings of documents.

A Firestore database, then, is a set of *Collections* that contain *Documents.* That's pretty much it! Of course it gets a bit more complicated when you realize that Documents can contain other Collections, which can contain other Documents, and so on. But that isn't really that much more complicated, when you realize that it's the same as pretty much all other hierarchical organization systems, such as file systems or URLs or dictionaries/JS Objects/JSON Objects.

Since all Firestore Documents must exist inside of Collections, we might place our `person1` document inside a collection like so:

```
[database root]: {
  'people': {
    'person1': {
      'firstName': 'Mark',
      'lastName': 'Newman'
    },
    'person2': {
      'firstName': 'Colleen',
      'lastName': 'Van Lent'
```

```
    }
  }
}
```

... where here ‘people’ is a collection at the top level of the database that contains 2 documents.

As with JSON (and python dicts, and JS objects), all collections that share a common parent must have unique IDs (aka keys), as must all documents that share a common parent collection. So you can’t do this:

```
[database root]: {
  'people': {
    'person1': {
      'firstName': 'Mark',
      'lastName': 'Newman'
    }
  },
  'people': {
    'person2': {
      'firstName': 'Colleen',
      'lastName': 'Van Lent'
    }
  }
}
```

or this

```
[database root]: {
  'people': {
    'person1': {
      'firstName': 'Mark',
      'lastName': 'Newman'
    },
```

```
    'person1': {
      'firstName': 'Colleen',
      'lastName': 'Van Lent'
    }
  },
}
```

While it's productive to think of Firestore databases being structured *like* JSON (or JS Objects) and sharing many of the same constraints, strictly speaking they are *not* JSON, as the differences between Documents and Collections provide structure that JSON key-value pairs don't. Indeed if you create the above structure in Firestore using the Firebase console, it will look like:

This simple examples contains all of the major structural elements of a Firestore database: Collections, Documents, and Fields. An additional note about Fields: unlike JSON fields (i.e., values associated with keys) which can only be strings, Firestore fields can have a variety of different data types, as you can see in the document creation UI.

You'll also note that Firestore allows you to "Auto-ID" documents, meaning that Firestore will assign a unique ID for you, either in the console or, more commonly, when you create new documents in code. Collections, however, cannot be auto-ID'ed. You need to name your collections something meaningful and unique.

Here's a summary of Firestore data model fundamentals:

- At the Top-level, the DB contains only collections
- Collections contain (only) documents
- Documents can contain *fields* or *collections*
- Fields have data types (string, number, boolean, map, array, ...)
- Sibling collections must have unique IDs (names)
- Sibling documents must have unique IDs
- Document IDs can (and often are) auto-generated. Collection IDs aren't.

Announcements

- Assignments have been re-shuffled
  - DYU4 due next week (11/4)
  - HW5 due in 2 weeks (11/11)

    - Proj 2 due in 3 weeks (11/18)
- Final Project is unchanged
    - Proposal due next week (11/4)
    - Plan due in 3 weeks (11/18)
    - After 11/18 it's all Final Project, all the time!
-

==========App of the Week==========

==========Theme: CRUDdier Apps==========

==========Seung Jun Kim==========

Accessing Firebase from React Native

To access Firebase from React Native, we will use the JavaScript `firebase` package. This package provides JS methods and objects to access and manipulate data stored in Firebase (along with many other things). We'll take a brief tour through the Firebase and Firestore functions and data structures, with a focus on seeing how to use Firestore to perform persistent CRUD operations.

We will use the `week9LM1K` project you created for this week's prep to look at the basic Firebase operations.

```
$ git clone git@github.com:SI669-internal/week9LM1K.git
```

Initializing Firebase and obtaining a Database Reference

Before you can access any Firebase services (such as Firestore), you need to initialize Firebase by passing in a `config` object that contains all of your secret, project-specific URLs and credentials. We'll start our good habits now by putting our `firebaseConfig` into a `Secrets.js` module and exclude that file from `git` using `.gitignore`. Create `Secrets.js` in your `week8FireStarter` project directory, and put your `firebaseConfig` in there (you can get this from your Firebase project, as we did in the prep video), adding the `export` keyword so you can `import` it into other files.

```
// Secrets.js
export const firebaseConfig = {
  apiKey: "XXXXXXXXXXXXXXXXXXX",
  authDomain: "YOUR-PROJECT-ID.firebaseapp.com",
  databaseURL: "https://YOUR-PROJECT-ID.firebaseio.com",
  projectId: "YOUR-PROJECT-ID",
  storageBucket: "YOUR-PROJECT-ID.appspot.com",
  messagingSenderId: "XXXXXXXXXX",
  appId: "X:XXXXXX:web:XXXXXXXXXX"
};
```

Now create a `.gitignore` file, and exclude `Secrets.js`, along with other stuff we should exclude.

```
# .gitignore
node_modules/**/*
Secrets.js

# Annoying Mac OS file
.DS_Store
```

You can check to make sure the exclusion will work by creating a git repo and checking the status.

```
$ git status
```

You should see, at least, a modified `.gitignore` , but you should *not* see `Secrets.js` as an "untracked" or "modified" file.

Next modify `App.js` to import `firebaseConfig` rather than declaring it. Then import the other stuff we'll need from the firebase module (don't get rid of the react/react-native imports):

```
import { firebaseConfig } from './Secrets';
import { initializeApp } from 'firebase/app';
import { initializeFirestore, collection,
  doc, getDoc, setDoc, addDoc, deleteDoc
} from "firebase/firestore";
```

Now initialize Firebase and Firestore, using the technique for initializing Firestore that was presented in the prep video. This should be done at the "top level" of your App.js file, i.e., not inside your `App` function:

```
const app = initializeApp(firebaseConfig);
const db = initializeFirestore(app, {
  useFetchStreams: false
});
```

To understand what's happening throughout the rest of the lecture, I'll point you towards specific Firebase (mostly Firestore) documentation, but for your own Firebase work you'll need to be able to track this kind of thing down. There are two primary resources you'll find helpful:

- Firestore Guides (conceptual overviews and example walkthroughs of common use cases like "Add and Manage Data" and "Read Data")
- Firestore "Web" API (detailed documentation on each Firestore function and object, e.g., `getDoc()`, `DocumentReference`, `QuerySnapshot`, etc.)

Reading Data

Now we're ready to get some data! We will start by fetching all of the documents from the `"people"` collection at the top level of our firestore database, and

updating the `peopleList` state variable with the data we retrieve. This will ensure we get the latest, greatest copy of our people list from persistent storage every time we run the app.

For this we'll need

- `collection()` (top-level firestore function that returns a `CollectionReference`)
- `CollectionReference` (a firestore class/data type that provides access to a collection)
- `query()` (a firestore function that constructs a query)
- `getDocs()` (a firestore function that runs a query and returns a `QuerySnapshot`)
- `QuerySnapshot` (a firestore class/data type that provides access to a set of query results)
  - `QuerySnapshot.docs` (the javascript array within a `QuerySnapshot` that contains the `DocumentSnapshot`s returned by the query)
- `DocumentSnapshot` (a firestore class/data type that encapsulates a single document)
  - `DocumentSnapshot.data()` (returns a javascript object containing the document's field data)
  - `DocumentSnapshot.id` (a string with the document's firebase ID)

```
useEffect(()=>{

  // a function inside a function!
  async function loadInitList() {
    const initList = [];
    const collRef = collection(db, 'people');
    const q = query(collRef);
    const querySnapshot = await getDocs(q);
    querySnapshot.docs.forEach((doc)=>{
      let person = doc.data();
      person.key = doc.id;
      initList.push(person);
    });
    setPeopleList(initList);
```

初始化，从firebase中读取所有文档，并放入initList中

```
    }
    loadInitList();

}, []);
```

useEffect不能接受一个async函数，只能接受一个返回async function的函数，所以多套了一层

Note: it turns out that making the function you pass to `useEffect()` "`async`" causes an error because `async` functions, by definition, return a `Promise` and the only thing that a `useEffect()` function can return is another function. To avoid this, we need to define an `async` function *inside* our `useEffect()` function, and then call it *within the same function.* Crazy.

When you try this out, you should see the contents of your Firestore "people" collection displayed on the screen, rather than the hardcoded list with John, Jen, and Jia.

Next, let's look at making sure that new people we add to the list get added to Firebase as well. We'll want to start by looking at the part of the code where a person gets added to the `peopleList` state variable. This is in the `onPress` handler for the Add/Save button, i.e., here:

```
        <Button title={mode === 'add'? 'Add' : 'Save'}
          onPress={()=>{
            if (mode === 'add') {
              let d = Array.from(peopleList);
              d.push({
                firstName: firstNameInput,
                lastName: lastNameInput,
                key: '' + Math.floor(Math.random() * 10000000)
              });
              setPeopleList(d);
              setFirstNameInput('');
              setLastNameInput('');
            } else {
```

```
              let d = Array.from(peopleList);
              let idx = d.findIndex((elem)=>elem.key === selec
tedItemKey);
              d[idx].firstName = firstNameInput;
              d[idx].lastName = lastNameInput;
              setPeopleList(d);
              setFirstNameInput('');
              setLastNameInput('');
              setMode('add');
              setSelectedItemKey('none');
            }
          }}
        />
```

For now we're only interested in the "Add" functionality, so we'll focus on the code in bold above.

For this task, we'll use some Firestore stuff we already used, like `collection()`, `CollectionReference`, and `DocumentReference`. In fact, all that's new here is

- `addDoc()` (a firestore function that adds an object to firebase as a document and returns a `DocumentReference` to the new document)

We'll do this as a separate function (within `function App()`, mind you!), so that our JSX doesn't get too cluttered and also to more elegantly handle the asynchronous aspects of what we're doing. Our `addPerson()` function will go right before the `return` statement.

```
  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);
    setPeopleList(pList);
    pList.push(personObj);
    setFirstNameInput('');
    setLastNameInput('');
  }
```

And now we simply need to call this function when the “Add” button is clicked:

```
        <Button title={mode === 'add'? 'Add' : 'Save'}
          onPress={()=>{
            if (mode === 'add') {
              addPerson(firstNameInput, lastNameInput);
              let d = Array.from(peopleList);
              d.push({
                firstName: firstNameInput,
                lastName: lastNameInput,
                key: '' + Math.floor(Math.random() * 10000000)
              });
              setPeopleList(d);
              setFirstNameInput('');
              setLastNameInput('');
            } else {
              let d = Array.from(peopleList);
              let idx = d.findIndex((elem)=>elem.key === selec
tedItemKey);
              d[idx].firstName = firstNameInput;
              d[idx].lastName = lastNameInput;
              setPeopleList(d);
```

```
              setFirstNameInput('');
              setLastNameInput('');
              setMode('add');
              setSelectedItemKey('none');
            }
          }}
        />
```

Now You Try #2

In our `addPerson()` function, we added the person to Firebase before updating the `peopleList` state variable.

1. Could we have done it in the other order? Why or why not?

Answer in slido.

Deleting Data

To delete data, we'll need to use two new functions:

- `doc()`:
  - takes a "reference" (i.e., either a database or `CollectionReference`)
  - and a "path" (which can be provided as a Unix-style path like `collection/document`, or as a "path" followed by "pathSegments", when it's more convenient to provide the document path in parts.
  - returns a `DocumentReference`
  - For example, a reference to the document with ID "person1" inside the collection "people" could be obtained in two ways with the same end result:
    - `doc(db, "people/person1")`
    - `doc(db, "people", "person1")`
- `deleteDoc()`: takes a `DocumentReference` and deletes it. Returns a "void" Promise (i.e., a Promise that doesn't resolve with any data)

You're going to implement delete. To get you started, here's what you're going to write for the onPress handler for the delete button rendered inside each list item:

```
    <Button title='Delete'
```

```
      onPress={()=> {
        deletePerson(item);
        let d = Array.from(peopleList);
        let idx = d.findIndex((elem)=>elem.key === item.key);
        d.splice(idx, 1);
        setPeopleList(d);
      }}
    />
```

Your job is therefore to implement `deletePerson(person)` such that, when called, the person is deleted from both Firebase and the UI.

Now You Try #3

Implement `deletePerson(person)` and paste your solution into slido. This one shoudn't give slido any trouble, so no need to make any special edits before submitting.

Updating Data

I'll bet you know where this is going next. For update, you just need one new function! Guess what's it called?

- `updateDoc()`: takes a `DocumentReference` and a javascript object, and updates the referenced document in Firebase with the supplied data. Note that the object that is provided as the second argument should have all of the document fields specified *other* than the key. The key is embedded in the `DocumentReference` and, ideally, should not be added as a field within the document also. So to update the document at "people/person1" to have the name "Tom Finholt", you would do the following:

```
    const docRef = doc(db, "people", "person1");
    await updateDoc(docRef, {firstName: "Tom", lastName: "Finholt"});
```

Key

Like `deleteDoc()`, `updateDoc()` returns a void Promise.

Now You Try #4

Implement `updatePerson(...)` and paste your solution into slido. To test it out, you'll also need to change the `onPress` handler for the "Add/Save" button to call `updatePerson()` (ask yourself, why "Add/Save" and not "Edit", so also paste your amended `onPress` handler into your slido solution. Your slido post should look something like this:

```
// updatePerson implementation
function implementation goes here

// onPress handler for "Add/Save" button
onPress={...}
```

If you follow these guidelines, it shouldn't give slido any trouble.

Notes on Project 2

A big part of your task will be to insert the Firebase operations we've gone over today into the ListMaker app. This includes figuring out which DataModel operations to modify and how (particularly how to deal with Promises/async/await appropriately), as well as things like figuring out how and when to load existing data into the app during startup.

I would encourage you to get started on at least this part sooner rather than later, because we haven't gone over *exactly* how to integrate these operations into a complete app, especially one that uses a separate DataModel as we have done with the ListMaker series. You have all of the tools at your disposal, but figuring out how to apply them may take a bit of trial and error. I'll be happy to help, but I'll want to see that you've given it a good try on your own first. Talking through strategies, architectures, etc. with your classmates is strongly encouraged—just don't share actual code ("pseudo code"—like the kind you might write on a whiteboard—is OK).

There are a few other challenges, including figuring out some new UI widgets that we haven't explicitly gone over, and these may take a bit of trial and error too, but you've done this kind of thing a few times now so I don't anticipate major difficulties there.

This project will require writing a decent amount of code. My own solution clocks in a bit above 500 lines across four files, and yours might even run a bit longer (or shorter, of course). This isn't a massive piece of software, by any means, but for many of you it might be among the largest programs you've written up to this point, and larger programs introduce their own problems in terms of complexity, readability, testability, integration, etc. This is all to say, yet again, don't wait til the last minute!

Conclusion

Asynchronous execution is fundamental in much JavaScript code. We've looked at event callbacks previously, and today we added `Promise`s and `async/await` to the toolbox of approaches for dealing with asynchronous execution. Firebase is a BaaS that can be used to provide persistent storage for mobile apps. Firebase Firestore is a NoSQL database provides a simple data model consisting only of Documents and Collections. The `firebase` JavaScript module provides an API you can use in JS-based apps (including React Native apps) to create, read, update, and delete data in a Firestore database. The `firebase` module makes heavy use of Promises, since all operations require server communication and therefore are best performed asynchronously. To use Firebase in apps that support CRUD, you need to make sure that all data-altering operations (Create, Update, and Delete) are performed *both* in Firebase and in the app's data model.