Google App Engine 3 Python by Example

What I will assume about you:

If you are reading this it is likely that you are taking a module about Cloud Computing. The following will be assumed about you before you start reading this book:

- You are a competent software developer/engineer or have completed one or more years study in Computer Science prior to this course in Computer Science.
- You have a working knowledge of HTML, CSS and JavaScript. Python would be useful but you can pick it up as you go through the examples.
- You have a working knowledge of dynamic web sites and full stack development. Doesn't matter which stack it is.

If you are missing one or more of the above I would suggest you address that shortfall before carrying on with this book.

Introduction:

The aim of this book is to give you some simple examples to follow that showcase what Google App Engine with Python 3 is capable of doing. It is recommended that you write out the programs as they appear and try to understand each one as you go. The reason for this is to enable you the learner to get used to the structures of a Google App Engine application before attempting the assignments set out in the course.

The documentation and programs you see here is heavily based on the documentation already available on http://cloud.google.com/ However, this is much expanded and contains explanatory material that can be used to further understand the workings of Google App Engine 3.

I would recommend that you go through all of the programs here. Understand in depth what is happening and the proceed to work with the assignments after this. As a first step however you will need to get your environment setup to work with Google App Engine python.

Working Environment:

There are a number of things that should be installed and setup before you attempt to run any of the programs here:

- A decent syntax highlighting text editor with support for: Python, HTML, CSS, JS, Yaml, JSON, and Git integration. My recommendation here is to use Visual Studio Code which is highly customisable and has many extensions available.
- An installation of the latest version of python 3 that is accessible on the command line. Note that if you are installing on Windows using the installer make sure you select a custom

install and enable the option to setup the Environment Variables. If you don't do this step you will not be able to run python off the command line.

- An installation of Google App Engine with the app-engine-python and app-engine-python-extras installed that is also accesible from anywhere on the command line. If you follow the installer script for Google App Engine it should do this for you automatically. The installation of this is covered two sections from now.
- An installation of the Git Source Code Management system. If you are on Linux your package manager will have a package for this. For other OSes go to https://git-scm.org/ and download and install from there.

Once you have all of the above installed and setup then you are ready to go with building Google App Engine applications.

Changes from the previous edition

There are two major changes to this book compared to the previous verison.

1. Flask is no longer used as the python framework. FastAPI is now used.
2. We have moved completely away from the Datastore to Firestore instead.

As a result the code of the examples have changed dramatically and have been modernised extensively. While we don't take full advantage of the features FastAPI has to offer it is a useful framework as it's asynchronus nature (if you write code to take advantage of it) will enable you to serve much more traffic for the same amount of resources than if you done everything synchronusly.

The move to Firestore was also necessary as Datastore was being kept as a legacy option. New applications are advised to use Firestore instead as it provides more features and is quicker than Datastore.

Installing Google App Engine SDK:

NOTE: At no point during this installation will you need to sign up for a trial or for billing on your Google account. If you are offered a trial period or billing is requested then decline it immediately. The free daily limits for App Engine and Firestore are more than enough for the examples and your assignments.

First and foremost you will need to go to this link

https://cloud.google.com/sdk/docs/install/

and download the relevant version of the SDK for your machine. For the purposes of this text I will use the Linux version of this as this is the OS that I use and setup tends to involve a few more steps. Once you have unpacked/installed the SDK, you will need to move onto the initialisation steps. For Linux/Mac OS users you will need to go into your newly created google-cloud-sdk/ directory and run the install.sh script from your command line. Those of you installing on Windows install.bat should run automatically and you should see the following prompts in a command line as well.

First up it will ask

Do you want to help improve the Google Cloud SDK? (your answer is N)

Next

Modify your profile to update your $PATH and enable shell command completion (your answer is Y)

It will then ask (if you are on Linux/Mac OS, something similar will appear on Windows but the response is the same)

Enter a path to an rc file to update, or leave blank to use [default] (accept the default)

Once the script is finished open a command line and do the following to install the App Engine Python SDK

gcloud components install app-engine-python app-engine-python-extras

Once that completes installation you can proceed to setting up an app engine project in the Google Cloud Console in the next section.

Setting up an App Engine project in the cloud console

First start by going to this URL and logging in with your Gmail account if necessary.

https://console.cloud.google.com/

Go to the option to add a new project and give it a name. Don't modify the location and hit create. Once the project is created then you should see it in the dashboard. Go to the hamburger men on the top left (the three horizontal lines denoting a menu) and under "Serverless" click on "App Engine" don't go into the sub menu here. It should show you a single dialog with a button "Create Application". Click on that and when region selection is offered choose "europe-west" then "Next". On the screen that follows click "I'll do this later". At this point it may ask you to setup billing. Decline this option. We will only need billing if you wish to run your App Engine applications on the Google Cloud. For the purposes of this text we will stick to local testing only. Your App Engine project should finalise setup in a few minutes but you can start working with the examples now. However, in later examples we will need to come back to this console to enable other necessary parts or clear databases for future examples.

Final steps before starting:

Finally create a directory somewhere in your home directory (or documents directory if you are on Windows) where you will store all of these examples. For example on my machine I have a directory called:

app-engine-fastapi/

In the examples later on I will refer to this directory as your examples directory

I also have a seperate directory for each of the 10 examples. And one for the python environment that will be shared with all of the examples (this will be created at the start of the first example) so your final directory structure after you finish the book should look similar to :

app-engine-fastapi/Example01

app-engine-fastapi/Example02

app-engine-fastapi/Example03

app-engine-fastapi/Example04

app-engine-fastapi/Example05

app-engine-fastapi/Example06

app-engine-fastapi/Example07

app-engine-fastapi/Example08

app-engine-fastapi/Example09

app-engine-fastapi/Example10

app-engine-fastapi/env

app-engine-python/requirements.txt

You will notice that there is a requirements.txt file here which will be filled out in the first example.

Example 01: Hello World in Google App Engine Python 3 with FastAPI

First before we can start with anything we will need to create a python virtual environment as we will need to install things into it without messing with the global python environment. Open a command line and using the “cd” command navigate to your examples directory and run the following command (Windows and Linux)

python -m venv env

Mac OS users may need the following depending on their OS version:

python3 -m venv env

This will create a directory called “env” that contains a python virtual environment that is seperate from the regular python environment. Before you run any of the examples or attempt to install a requirements file you will need to run the following command (Linux and Mac OS)

source env/bin/activate

or if you are on Windows terminal (CMD)

./env/Scripts/activate.bat

of if you are on Windows PowerShell

./env/Scripts/activate.ps1

This will modify your terminal PATH and other variables to reference the newly created env directory first. You may notice after this command that you will see “(env)” before the start of your command line to indicate that you are in a python virtual environment. You will need to navigate to this directory and run the second of these two commands everytime you start a terminal te setup thei virtual environment before you can start running app engine applications. The only way to get out of the virtual enviornment is to exit the terminal when you are finished. Now we can get to developing the application by following the given steps.

01) In your examples directory create a new directory called Example01. In Example 01 create a single file called main.py and add the following code to it

```
1  from fastapi import FastAPI
2
3  app = FastAPI()
4
5  @app.get("/")
6  async def root():
7      return {"message" : "hello world"}
```

There are a number of things to explain here

- Line 1 is where we are importing the FastAPI library upon which our App Engine application is built. Without it none of the examples in this book will work

- Line 3 is where we are initialising FastAPI and getting it ready for use. Note that we don't specify any options in the constructor as for these examples the default will do
- Line 5 is the definition of a route that gets added to the app variable that we defined in line 3. it states what HTTP verb should be used to trigger the route in this case GET (the ".get(" part) and the URL it should be applied to. In this case / meaning if we run this on localhost this route will be triggered when a HTTP GET verb is called on http://localhost:8080/. When the route is triggered it will call the function immediatly listed below it (in this on line 6) to generate a response to this HTTP GET request.
- Line 6 is the definition of a python function that will return a reply to the get function. All this function will do is define a python dictionary and return it. When a python dictionary is to be returned FastAPI in the background converts this into JSON format before returning it as a response.
    - You will note that the async keyword is used in the definition of the function. FastAPI by default is asynchronus in nature. This means that if we define a route with the async keyword, should that function suspend while waiting for a result from another function to return FastAPI in the background can suspend our function and do something else. When the result has been returned FastAPI will come back to your function and continue executing it from where it stopped.
    - The reason for the above is that this allows higher throughput, meaning a larger number of requests can be served by the same server than if we only used synchronus behaviour.

02) in requirements.txt add the following dependancies

```
1  fastapi==0.97.0
2  google-auth==2.20.0
3  google-cloud-firestore==2.11.1
4  google-cloud-storage==2.10.0
5  Jinja2==3.1.2
6  python-multipart==0.0.6
7  requests==2.31.0
8  uvicorn==0.22.0
```

and run the following command

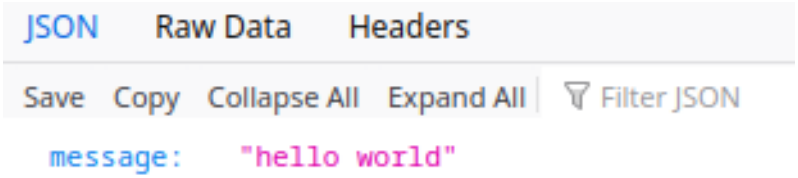
pip install -r requirements.txt

This will install all the dependancies we need to run all the examples in this book.

02) On the terminal cd into Example01 and run the following command

uvicorn main:app –reload

This will launch a FastAPI server running your application above on http://localhost:8000/ when you navigate to this URL with a browser you should see a JSON object that looks similar to this (screenshot taken in Firefox)



This is your very first admittedly not useful Google App Engine application but it is a start. In the next example we will introduce templates to start rendering normal HTML/CSS/JS content as JSON is not particularly user friendly.

Example 02: Hello World but using templates to display dynamic content

While the previous example showed us how to get up and running with a very basic FastAPI application that could be run on Google App Engine the UI was not very friendly in that it just returned a JSON object. In this example we will show how to render HTML/CSS content dynamically by using JINJA2 templates. In this example a very simple page will be rendered with a name and student number which will be provided to the JINJA2 template. If you change the name and student number in the python code the template will render that name and student number on the next refresh of the page.

01) In your examples directory create a new directory called Example02 and add the following files and directories to it

- main.py
- static/
  - styles.css
- templates/
  - main.html

02) to styles.css add the following:

```
1  body {
2      font-family: "helvetica", sans-sans-serif;
3      text-align: center;
4  }
```

Some very basic CSS here to set the Font and also align all text to the centre of the page

03) to main.py add the following:

```
1  from fastapi import FastAPI, Request
2  from fastapi.responses import HTMLResponse
3  from fastapi.staticfiles import StaticFiles
4  from fastapi.templating import Jinja2Templates
5
6  # define the app that will contain all of our routing for Fast API
7  app = FastAPI()
8
9  # define the static and templates directories
10 app.mount('/static', StaticFiles(directory='static'), name='static')
11 templates = Jinja2Templates(directory="templates")
12
13 @app.get("/", response_class=HTMLResponse)
14 async def root(request: Request):
15     return templates.TemplateResponse('main.html', {'request': request, 'name': 'John Doe', 'number': '1234567'})
```

There is quite a bit to explain here:

- Imports are on lines 1 to 4. Like the previous example we pull in the FastAPI module but there are four extra modules imported

    - Request: While we don't need this now we will need it in later examples. The Request object is how you will access parameters that have been sent to the route by the client. The most common use for this is to capture and access data entered into web forms
    - HTMLResponse: Compiles a response in the form of a HTML document such that when sent as a response to a HTTP GET verb this will render a HTML document on the clients browser
    - StaticFiles: Needed to access the static (unchanging) content that will be stored in our static/ directory
    - Jinja2Templates: Needed to process and assemble dynamic HTML content
- Line 10 is where we declare to the FastAPI library where to find all of our static files content. app.mount() takes three parameters which are:
    - "/static" refers to the route that these static files will be made available. If our application is running on localhost any URL that points to http://localhost/static/ will be handled by this sub-application (FastAPI's terminology). The sub-application is the second parameter that will handle the rest of the URL
    - "StaticFiles(directory='static')" The sub-application that will handle our static files content. "directory" points to the relative path where to find the static files content. Absolute paths are not permitted here. If you push an application with an absolute path for static content, said content will not be seen when your application is running on Google App Engine.
    - name='static' the name that FastAPI will internally use for this sub-application in this case "static"
- Line 11 is where we initialise the Jinja template engine. The only parameter we need to supply here is the relative path to the location of the templates. If you followed along with step 1 this will point at the templates/ directory you defined in that step.
- Line 13 sees a small change to our declaration of our route by adding in "response_class=HTMLResponse" This is stating to FastAPI that when this route is finished it will return a HTMLResponse that should be sent back to the client.
- Line 14 sees another change in that we add a parameter to our root() function "request: Request". This is to allow us to take the request that was sent to the route from the client's browser and use any information that is embedded in it.
- Line 15 is where we call the templating engine to render a HTMLResponse. Here we provide two parameters.
    - The first (main.html) is the template that we wish to use to render the HTMLResponse
    - The second ({'request.....}) is a python dictonary containing all of the data that you wish to pass to the template. The keys listed in this python dictionary can be used in the template itself to access the data attached to them. We will see this in the next step

04) to main.html add the following:

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>First templated HTML page through Jinja2</title>
5          <link href="{{ url_for('static', path='/styles.css') }}" rel="stylesheet"/>
6      </head>
7      <body>
8          <h1>Student name: {{ name }}</h1>
9          <h1>Student number: {{ number }}</h1>
10     </body>
11 </html>
```

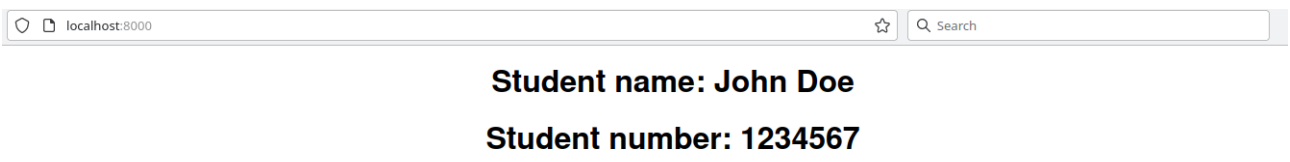
This is an example of a Jinja template. Note that most of this looks like a standard HTML document however we will discuss the parts that have been parameterised namely on lines 5, 8, and 9

- On line 5 we have a parameterisation that will generate a URL for the static content of the application. Note that in Jinja whenever we wish to use a parameter we have to encase it in double braces "{{ }}". Jinja will recognise this and will replace the double braces with the variable or output from the function used. In this case we are using the "url_for" function which will generate a full url for the relative paths given.
    - The first parameter "static" references the endpoint we wish to use. In the previous step we defined an endpoint called "static" that references all of our static content.
    - The second parameter "path=/styles.css" is the path to the required content in the endpoint declared in the first parameter.
    - Both parameters will be combined with the URL that the application is running on to generate the full path. Thus if this is running on localhost it will generate the full path as http://localhost:8000/static/styles.css
    - Thus if you were then to push this to Google App Engine it should change localhost:8000 to be whatever your domain name is on GAE and all the links and references will still work.
- Line 8 and line 9 are using references to keys that were passed in the dictionary in the previous step. Line 8 will access the value attached to the key called "name" while line 9 will access the value attached to the key called "number" both lines will take whatever values are there and render them in place of "{{ name }}" and "{{ number }}". Thus if you change the value attached to name and number in the previous step the template will render those changed values on the next page refresh

05) run the application using the following command

uvicorn main:app –reload

and when you navigate to http://localhost:8000/ you should see something similar to this



Setting up Firebase Authentication

Before we can go to the next example we will need to setup support for authentication through Firebase. While we could write our own login system, we will avoid that work and use one that is proven and reliable. Before we can integrate this into our application we will need to setup authentication through the Firebase console. At the end of this you should end up with a snippet of code that can be added to a JavaScript file to link to your login system.

01) to start goto https://console.firebase.google.com/ and login with the same Google account that you used to setup your application earlier.

02) click on “add project” and on the following screen click on “Enter your project name” and it should present you with a dropdown list of your existing Google Cloud projects and your Google App Engine application should be there. Click on that and click “Continue” then disable Google Analytics and finish.

03) you will need to wait a few seconds for it to finish creating before you see the dashboard for the project. On the menus on the left click on “Authentication” If you cannot see it, it should be under the “Build” menu.

04) click on “Get started” and choose “Email/Password” under “Native Providers”. Enable “Email/Password” and click “Save”

05) go back to the dashboard and on the “Project Overview” page you should see an icon that looks like a HTML tag “</>”. Click on that. Give your app a name, leave Firebase Hosting unchecked and click “Register App”

06) on the form that follows click “Use a <script> tag and copy the lines that define the “const firebaseConfig ...” variable and save this somewhere. You do not need the rest of the lines as we will provide our own version of this in the next example.

Example 03: Managing user authentication with the use of email/password login through firebase.

NOTE: do not attempt this example until you have completed the section on setting up firebase authentication before this example.

In this example we will add in firebase authentication to our application. Almost all cloud facing applications will need to have some form of authentication mechanism. Rather than building your own we will use Firebase Authentication to handle this for us.

NOTE: this is how login/logout is required to be implemented in assignments. Alternative solutions or libraries will not be permitted. For assignments you are required to include firebase-login.js. It must be named firebase-login.js and you are only permitted to modify the firebaseConfig variable and nothing else. This is to make correction of your assignments quicker.

01) Create an Example03 directory in your Examples directory and setup the following file structure

- static/
  - firebase-login.js
  - styles.css
- templates/
  - main.html
- main.py

02) create styles.css in the same way you created it in Example 02

03) in main.py add in the following code

```python
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
import google.oauth2.id_token
from google.auth.transport import requests

# define the app that will contain all of our routing for Fast API
app = FastAPI()

# we need a request object to be able to talk to firebase for verifying user logins
firebase_request_adapter = requests.Request()

# define the static and templates directories
app.mount('/static', StaticFiles(directory='static'), name='static')
templates = Jinja2Templates(directory="templates")
```

This looks very similar to the previous example with a few additions

- In terms of imports there are two additional imports one line 5 and line 6. These are to support the OAuth2 authentication system that Firebase is using and also to allow us to route requests to the firebase authetication system
- Line 12 creates a Firebase Request Adapter that we will use to route login information to authenticate our users.

04) in main.py add in the following code

```
18  @app.get("/", response_class=HTMLResponse)
19  async def root(request: Request):
20      # query firebase for the request token. We also declare a bunch of other variables here as we will need them
21      # for rendering the template at the end. we have an error_message there in case you want to output an error to
22      # the user in the template.
23      id_token = request.cookies.get("token")
24      error_message = "No error here"
25      user_token = None
26
27      # if we have an ID token we will verify it against firebase. If it doesn't check out then log the error message that is returned
28      if id_token:
29          try:
30              user_token = google.oauth2.id_token.verify_firebase_token(id_token, firebase_request_adapter)
31          except ValueError as err:
32              # dump this message to console as it will not be displayed on the template. use for debugging but if you are building for
33              # production you should handle this much more gracefully.
34              print(str(err))
35
36      return templates.TemplateResponse('main.html', {'request': request, 'user_token': user_token, 'error_message': error_message})
```

There are a few additions here compared to the previous example

- Lines 23 to 25 setup a number of variables that we will pass to the template on line 36. We have to declare them here before the if statement as declaring them in the if statement will cause them to go out of scope before the template is rendered and thus produce compiler errors.
- Line 23 declares an id_token variable that is populated with whatever is stored under the “token” key in the cookies attached to the request. If a user is logged in and authenticated this will be a token that when you ask firebase to validate, it will match the logged in user. If no user is logged in this will be empty.
- Line 24 declares an error message variable that in later examples or assignments can be used as a means of communicating errors to your user should a request not complete correctly. For now we will say “no error here” as a place holder.
- Line 25 is a user token that is currently set to None. This is a token that will be returned from Firebase after the id_token is validated. It will give us some information about the user including their email address.
- Line 28 checks for the existance of an id_token that we pulled from the cookies in line 23. If an id_token exists it will try and verify that token with firebase on lines 29 and 30. If validation fails an exception will be thrown which is caught on line 31 which prints the exception to console on line 34. If validation is successful the user token is returned and stored in the user_token variable.
- Line 36 then renders the main.html template and we pass the request object, user_token, and error_message to the template

05) in main.html add in the following code

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Firebase login</title>
5          <link type="text/css" href="{{ url_for('static', path='/styles.css') }}" rel="stylesheet"/>
6          <script type="module" src="{{ url_for('static', path='/firebase-login.js') }}"></script>
7      </head>
8      <body>
9          <div id="login-box" hidden="true">
10             Email:<input type="email" name="" id="email"><br/>
11             Password: <input type="password" name="" id="password"><br/>
12             <button id="login">Login</button>
13             <button id="sign-up">Sign Up</button>
14         </div>
15         <button id="sign-out" hidden="true">Sign out</button>
16
17         <!-- if we have a logged in user then show the user email address from the user_token object that was passed in -->
18         {% if user_token %}
19             <p>User email: {{ user_token.email }}</p>
20             <p>error message: {{ error_message }}</p>
21         {% endif %}
22     </body>
23 </html>
```

A bit more templating has gone on here so we will go through those bits as well as the login box

- On line 6 you will note that we have an additional reference to a static piece of content in the form of firebase-login.js. We will define this script in the following steps. It's purpose will be to interact with the firebase authetication system that we setup before we started this example. It will be responsible for validating and authenticating users. It will also affect the login box and the sign out button on this template. The end result will be that if a user is logged in the login box will be hidden and the sign out button will be visible. If there is no user logged in the login box will be visible and the sign out button will be hidden.
- The login box is defined on lines 9 to 15 by default we have set this to hidden as firebase-login.js will control its visibility. This is a very simple form that takes in an email address and password with a login and signup button. Note that no logic has been assigned to the buttons as this will be defined in firebase-login.js
- The signout button is defined on line 16. No logic has been attached to this as it will be defined in firebase-login.js
- lines 18 to 21 define some dynamic content that will appear only if a user_token exists. If the user_token has a value other than None (as was set on line 25 of main.py) then this will be rendered and made visible to the user. It will pull the email address from the user token and render it in place of {{ user_token.email }} and will take the error message and render it in place of {{ error_message }}.

06) in firebase-login.js add the following code

```
1  'use strict';
2
3  // import firebase
4  import { initializeApp } from "https://www.gstatic.com/firebasejs/9.22.2/firebase-app.js";
5  import { getAuth, createUserWithEmailAndPassword, signInWithEmailAndPassword, signOut } from "https://www.gstatic.com/firebasejs/9.22.2/firebase-auth.js"
6
7  // Your web app's Firebase configuration
8  const firebaseConfig = {
9      apiKey: [redacted]
10     authDomain: [redacted]
11     projectId: [redacted]
12     storageBucket: [redacted]
13     messagingSenderId: [redacted]
14     appId: [redacted]
15 };
```

Note the following:

- on lines 4 and 5 we pull in the necessary imports from the Firebase Javascript libraries that we need to interface with the Firebase Authentication system. We pull in dependancies to allow us to authorise, create, sign in, and sign out users from the autentication system.
- On lines 8 to 15 we include our firebase config. Note I have pixellated out my own configuration here as the intention is for you to replace this const firebaseConfig variable with the one I got you to save on step 06 in the “Setting up Firebase Authentication” section before this example. If you have not done all the steps in this section you need to do it now before you can proceed any further. This variable will direct the script to your specific firebase authetication system instance for user logins.

07) in firebase-login.js add the following code

```
17  window.addEventListener("load", function() {
18      const app = initializeApp(firebaseConfig);
19      const auth = getAuth(app);
20      updateUI(document.cookie);
21      console.log("hello world load");
22
23      // signup of a new user to firebase
24      document.getElementById("sign-up").addEventListener('click', function() {
25          const email = document.getElementById("email").value
26          const password = document.getElementById("password").value
27
28          createUserWithEmailAndPassword(auth, email, password)
29          .then((userCredential) => {
30              // we have a created user
31              const user = userCredential.user;
32
33              // get the id token for the user who just logged in and force a redirect to /
34              user.getIdToken().then((token) => {
35                  document.cookie = "token=" + token + ";path=/;SameSite=Strict";
36                  window.location = "/";
37              });
38
39          })
40          .catch((error) => {
41              // issue with signup that we will drop to console
42              console.log(error.code + error.message);
43          })
44      });
```

On line 17 we add a callback function to the client’s browser that will run the given JavaScript code when the page has been loaded.

- Line 18 and 19 will initialise the firebase authentication system ready for use by the rest of the script
- Line 20 will make a call to the updateUI function that we will define in a later step. The reason why we pass document.cookie to it is to check for the presence of the “token” cookie. If this is present updateUI will show the signout button on the UI, if not then the login box will be shown instead.
- Line 21 is a message that is logged to the console of the client browser. This won’t get dumped out to the terminal that FastAPI is running on. If you are using Firefox and want to

see the browser console hit Ctrl+Shift+I (Windows and Linux) and you will see the message there.

- Lines 24 to 44 define the logic that will called whenever the sign-up button is clicked on the page.
    - Lines 25 and 26 pull the user entered email and password from the login-box that was defined in main.html
    - Line 28 will call the createUserWithEmailAndPassword function that we imported from firebase earlier with the link to our authentication instance, email address that the user provided, and the password provided as well. If this action is successfull the code in the .then function on lines 29 to 39 will be called. If it fails the .catch() function on lines 40 to 43 will be called instead.
    - In the .then function on line 31 firebase will pass us a userCredential object, which contains a user object which we extract
    - On lines 34 to 37 we then extract the id token for that user and attach is as a cookie called “token” in the clients browser on line 35. Note the presence of two additional parameters
        - “path=/” is required to prevent the same cookie being stored on different URLs. If you omit this you will end up with a bug where a user can logout of one URL but if they visit a different URL in your application they will still be logged in. By forcing the path to be “/” we are forcing the browser to store one and only one copy of the token. Thus if we logout we logout of all URLs.
        - “SameSite=Strict” states that this cookie is not to be transfered to another site during a cross-site request. For the token this makes sense as you do not want to pass authentication tokens to another site as this would be a security risk for all users.
    - Line 36 will force a redirect to the “/” route of the application. Thus no matter what URL the user is on their browser will automatically redirect them here after a successful signup.
    - The .catch function on lines 40 to 43 will simply take the error given by Firebase authentication and will dump it in the browser console. Hopefully you won’t need this but if Firebase authentication is not working you will have some ability to diagnose the problem.

08) in firebase-login.js add the following code

```
    // login of a user to firebase
    document.getElementById("login").addEventListener('click', function() {
        const email = document.getElementById("email").value
        const password = document.getElementById("password").value

        signInWithEmailAndPassword(auth, email, password)
        .then((userCredential) => {
            // we have a signed in user
            const user = userCredential.user;
            console.log("logged in");

            // get the id token for the user who just logged in and force a redirect to /
            user.getIdToken().then((token) => {
                document.cookie = "token=" + token + ";path=/;SameSite=Strict";
                window.location = "/";
            });


        })
        .catch((error) => {
            // issue with signup that we will drop to console
            console.log(error.code + error.message);
        })
    });
```

The functionality defined here for the login button of the login box is almost exactly the same as the previous step except instead of calling createUserWithEmailAndPassword() we call signInWithEmailAndPassword() with the exact same set of parameters to attempt to login the user.

09) in firebase-login.js add the following code

```
    // signout from firebase
    document.getElementById("sign-out").addEventListener('click', function() {
        signOut(auth)
        .then((output) => {
            // remove the ID token for the user and force a redirect to /
            document.cookie = "token=;path=/;SameSite=Strict";
            window.location = "/";
        })

    });
});
```

Here we add logic for the sign out button that will call the signOut() function with our authentication system. Once this has been done we will set the “token” in the cookie to be empty and redirect the user back to the “/” route.

10) in firebase-login.js add the following code

```
// function that will update the UI for the user depending on if they are logged in or not by checking the passed in cookie
// that contains the token
function updateUI(cookie) {
    var token = parseCookieToken(cookie);

    // if a user is logged in then disable the email, password, signup, and login UI elements and show the signout button and vice versa
    if(token.length > 0) {
        document.getElementById("login-box").hidden = true;
        document.getElementById("sign-out").hidden = false;
    } else {
        document.getElementById("login-box").hidden = false;
        document.getElementById("sign-out").hidden = true;
    }
};
```

This is the updateUI function that will control the visibility of the login box and the signout button depending on if the user is logged in or not.

- Line 84 will call the parseCookieToken() function that we will define in the next step. This will take the value of cookie which may contain multiple parameters and will extract the value associated with “token”. If it exists it will return the full value. If not it will return the empty string
- Lines 87 to 93 then check to see if the length of the value of cookie is greater than 0. If it is we assume there is a valid cookie and thus the login box should be hidden and the sign out button should be made visible. If not then the login box should be made visible and the sign out button should be hidden

11) in firebase-login.js add the following code

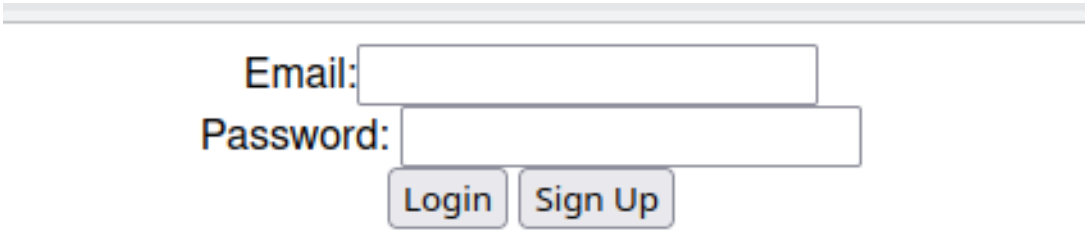
```
96  // function that will take the and will return the value associated with it to the caller
97  function parseCookieToken(cookie) {
98      // split the cookie out on the basis of the semi colon
99      var strings = cookie.split(';');
100
101     // go through each of the strings
102     for (let i = 0; i < strings.length; i++) {
103         // split the string based on the = sign. if the LHS is token then return the RHS immediately
104         var temp = strings[i].split('=');
105         if(temp[0] == "token")
106             return temp[1];
107     }
108
109     // if we got to this point then token wasn't in the cookie so return the empty string
110     return "";
111
112 };
```

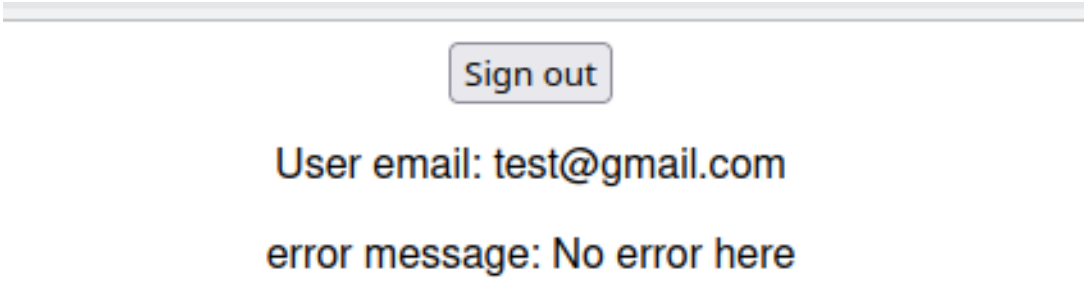
This function will take in a cookie string and will look for and return the value from the token set in that cookie.

- Line 99 will first split the cookie into multiple strings by using the semi-colon as the separator.
- Lines 102 to 107 will iterate through those strings and split them based on the equals sign. This will give us two strings. The first being an attribute and the second being a value. We check to see if any of the attributes are named “token” if it is we return the value of it immediately.
- Line 110 will only ever be reached if there is no token set on the cookie. If this is the case we will return the empty string to indicate that no valid token is set.

12) run the code with the following command and you should see output similar to the below showing what the application looks like when a user is not logged in and logged in

uvicorn main:app --reload





Setting up a Firestore database

For the following examples you will need access to a Firestore NoSQL database. To set this up you will need to go back to the Google Cloud Console (https://console.cloud.google.com/) and do the following steps.

01) Click on the hamburger menu at the top left and under “More Products” go to “Databases” and then click on “Firestore”

02) there should be no databases listed and you should have an option to “Create Database” click this. You will be given two options for “Firestore mode” You want “Native Mode”. Do not under any circumstances select “Datastore mode”. If you do none of the following examples will work.

03) click “Continue”. On the following page select “Region” for location type, select your Region and click “Create Database”

04) your database should be ready to go but give it about 5 minutes before you start accessing it from the examples as it usually takes a little time to fully create

We are not fully finished setting up Firestore here as we will need to get a key that we can link to our applications. The reason we need this is that the Firestore library we will use does not automatically know which firestore database to use. For this we will need access to our service account.

Getting access to your service account

Now that we have setup our Firestore database we will need to get a JSON file that we can link to our applications as the Firestore library by default will not know which Firestore database it has to connect to. To get this we will need to go back to the google cloud console (https://console.cloud.google.com/)

01) Click on the hamburger menu on the top left and under “More Products” go to “IAM and admin” and click on “Service Accounts” in the popup menu

02) In the list of service accounts that show up there should be one labelled “App Engine default service account” click on that one.

03) In the following screen click on “Keys” and click on “Add Key”. This will give you two options. The one you want is “Create Key”

04) In the dialog that follows choose “JSON” and then click “Create”. This will download a JSON file for you. Make sure you put this in the root of your examples directory as you will need it for all examples and your assignments as well.

Example 04: Introduction to the basics of Firestore

At this point we now have the ability to login/logout users and also generate some dynamic content. In order to make our applications much more useful we need a mechanism through which we can store data. This is where Firestore will be used. Firestore is a NoSQL database. It is designed for speed rather than expressive power. If you are coming from an SQL background you will see later that the types of queries you can perform are a lot more limited compared to SQL. However, the trade off is much increased speed. The guiding principle of NoSQL is:

“Storage is cheap, compute is expensive”

i.e. we would rather denormalise our database and duplicate data if it saves on computation costs. SQL was built at a time when storage was expensive compared to compute and thus the priority was saving disk space and using expressive queries to fill in the rest.

In this example we will show how to add data to Firestore and also how to update it through web forms. We will also take data from Firestore and pass it to our templates to show how it can be used to create dynamic content.

Note you will not be able to proceed with this example unless you have completed the previous section to setup the firestore database.

01) In your examples directory create an Example04 directory with the following file and directory structure

- static/
  - firebase-login.js
  - styles.css
- templates/
  - main.html
  - update.html
- main.py

02) use the firebase-login.js and styles.css from the previous example

03) In main.py add the following imports

```
1  from fastapi import FastAPI, Request
2  from fastapi.responses import HTMLResponse, RedirectResponse
3  from fastapi.staticfiles import StaticFiles
4  from fastapi.templating import Jinja2Templates
5  import google.oauth2.id_token;
6  from google.auth.transport import requests
7  from google.cloud import firestore
8  import starlette.status as status
```

There are a couple of additional imports that we have added in here:

- Line 2 now includes a RedirectResponse. We will use this whenever we update data. Usually updating data comes in the form of a POST request which does not return a HTML response to the client. Anytime we do a post we will perform a redirect (which uses the GET verb) to an appropriate URL to display to the client.
- Line 7 imports the Google Firestore python library. We will need this to interact with our firestore database.
- Line 8 imports a representation of HTTP status codes. We will need this when we return a RedirectResponse to indicate what kind of redirect we are performing.

04) In main.py add the following code

```
10  # define the app that will contain all of our routing for Fast API
11  app = FastAPI()
12
13  # define a firestore client so we can interact with our database
14  firestore_db = firestore.Client()
15
16  # we need a request object to be able to talk to firebase for verifying user logins
17  firebase_request_adapter = requests.Request()
18
19  # define the static and templates directories
20  app.mount('/static', StaticFiles(directory='static'), name='static')
21  templates = Jinja2Templates(directory="templates")
```

The only difference here is on line 14. This is where we initialise the Firestore library so we can interact with our database. Note that we do not provide a reference to our JSON file that contains details of our Firestore database. We will use an environment variable (provided on the command line, will show this in a later step) to tell the Firestore library where our database is.

05) In main.py add the following code

```
23  # function that we will use to retrieve and return the document that represents this user
24  # by using the ID of the firebase credentials. this function assumes that the credentials have
25  # been checked first
26  def getUser(user_token):
27      # now that we have a user token we are going to try and retrieve a user object for this user from firestore if there
28      # is not a user object for this user we will create one
29      user = firestore_db.collection('users').document(user_token['user_id'])
30      if not user.get().exists:
31          user_data = {
32              # our signup form doesn't have a name field so we will set a default that we will edit later
33              'name': 'No name yet',
34              'age': 0
35          }
36          firestore_db.collection('users').document(user_token['user_id']).set(user_data)
37
38      # return the user document
39      return user
```

This is our first function that will interact with firestore. What the function will do is try to pull an object (document in Firestore terminology) associated with the user. If it exists we will return that document. If not we will create it and return it.

- Line 29 is where we attempt to retrieve the document related to the currently logged in user. Note that the document is stored in a collection called “users”. Collections group together sets of documents that represent the same thing. In this case each document we will store under “users” should represent data relevant to an individual user. The parameter passed

to .document() is the unique identifier for that document. In this case we will use the “user_id” parameter in the user token provided to us by Firebase Authentication as this will uniquely identify each user. A document will then be stored in “user” even if said document does not exist in the database which is where the next segment comes in

- Line 30 attempts to pull the document from Firestore and check if it exists. If the document exists the body of the if statement will be skipped. If the document does not exist we first generate a python dictionary (lines 31 to 35) containing our default data about the user and then on line 36 we then write that data to firestore. Thus the next time we access the user data it will exist and it will be returned directly instead of being created
- Line 39 then returns this data to the caller.

06) In main.py add the following code

```
41  # function that we will use to validate an id_token. will return the user_token if valid, None if not
42  def validateFirebaseToken(id_token):
43      # if we dont have a token then return None
44      if not id_token:
45          return None
46
47      # try to validate the token if this fails with an exception then this will remain as None so just return at the end
48      # if we get an exception then log the exception before returning
49      user_token = None
50      try:
51          user_token = google.oauth2.id_token.verify_firebase_token(id_token, firebase_request_adapter)
52      except ValueError as err:
53          # dump this message to console as it will not be displayed on the template. use for debugging but if you are bui
54          # production you should handle this much more gracefully.
55          print(str(err))
56
57      # return the token to the caller
58      return user_token
```

First and foremost this is code for user validation that you have seen already in the previous example. What we are doing here is a little bit of refactoring by splitting it off into its own seperate function. The reason for this is we will need to validate the user token on every route we use. Rather than writing several copies of this code, having it in one function makes it smaller and simpler. In this particular function we must provide the token and it will then validate. If the token is valid it will return that token otherwise it will return None.

07) In main.py add the following code

```
61  @app.get("/", response_class=HTMLResponse)
62  async def root(request: Request):
63      # query firebase for the request token. We also declare a bunch of other variables here as we will need them
64      # for rendering the template at the end. we have an error_message there in case you want to output an error to
65      # the user in the template.
66      id_token = request.cookies.get("token")
67      error_message = "No error here"
68      user_token = None
69      user = None
70
71      # check if we have a valid firebase login if not return the template with empty data as we will show the login box
72      user_token = validateFirebaseToken(id_token)
73      if not user_token:
74          return templates.TemplateResponse('main.html', {'request': request, 'user_token': None, 'error_message': None, 'user_info': None})
75
76      # get the user document and render the template
77      user = getUser(user_token)
78      return templates.TemplateResponse('main.html', {'request': request, 'user_token': user_token, 'error_message': error_message, 'user_info': user.get()})
```

This is the “/” route that will be the first thing a user sees when they interact with our application.

- The variables declared in lines 66 to 69 are similar to the previous example with the exception of line 69, the declaration of the user variable. We will use this to store the retrieved user document from Firestore and we will pass this to the template later

- Lines 72 to 74 validate if this is a logged in user If a user is not logged in it will redirect back to the “/” route without providing any data. This is a validation and security measure. If a user is not logged in they should not be able to see any information in the application.
- Line 77 and 78 will pull the user document from Firestore and then pass it to the main.html template to be rendered for the client. Main.html will be defined in a later step. Note the use of user.get() this will pull the user document form Firestore before we attempt to render the template

08) In main.py add the following code

```
80  # add in a second route to show us a form for updating the name and age of the user
81  @app.get("/update-user", response_class=HTMLResponse)
82  async def updateForm(request: Request):
83      # there should be a token here
84      id_token = request.cookies.get("token")
85
86      # validate the token. if its not valid then redirect to / as a basic security measure as a non logged in user should not be accessing this
87      user_token = validateFirebaseToken(id_token)
88      if not user_token:
89          return RedirectResponse('/')
90
91      # get the user document and send it to the template that will show a basic form for changing this data
92      user = getUser(user_token)
93      return templates.TemplateResponse('update.html', {'request': request, 'user_token': user_token, 'error_message': None, 'user_info': user.get()})
```

Here we have defined a second route “/update-user”. If you were running this on local host you can access it through http://localhost:8000/update-user. We get this to return a HTML response as it is responding to the GET verb. We will use this route to render a simple form through which the user can update their name and age.

- Lines 84 to 89 are the validation check that you saw in the previous step. Again if a user attempts to view this page without a valid login we will redirect them back to / to force them to login.
- Lines 92 and 93 pull the currently logged in user’s document from Firestore and passes it to the update.html template (we will define this later) to render the current values stored for the user in their document

09) In main.py add the following code

```
95  # this is another version of update user but this will accept a post request and will only redirect when finished.
96  @app.post("/update-user", response_class=RedirectResponse)
97  async def updateFormPost(request:Request):
98      # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
99      id_token = request.cookies.get("token")
100     user_token = validateFirebaseToken(id_token)
101     if not user_token:
102         return RedirectResponse('/')
103
104     # pull the user document and then we will modify the name and age and update it
105     user = getUser(user_token)
106     form = await request.form()
107     user.update({"name": form['name'], "age": int(form['age'])})
108     return RedirectResponse('/', status_code=status.HTTP_302_FOUND)
```

You may have noticed that we appear to be defining the “/update-user” route again. However note the call to app.post() instead of app.get(). This route will only be triggered if the HTTP POST verb is called on “/update-user”. We set this as a redirect response as post requests generally should not return data to the caller.

- Lines 99 to 102 are our now standard user validation check. If there is no valid login we will redirect to “/” as it would be a security risk if non-logged in users could change data in an application
- Line 105 is where we retrieve the document for the currently logged in user that we need to update
- Line 106 is pulling the form from the request object that will contain the name and age that the client has entered. This is an asynchronus function, and you may have noticed that we have the await keyword in front of it. What FastAPI will do here is see if request.form() returns immediately. If so it carries on processing. If not FastAPI will suspend updateFormPost() at this point and will execute other code until request.form() indicates it has completed its work. Once completed FastAPI will unsuspend updateFormPost() and carry on processing.
- Line 107 is a call to Firestore to update our user document. The python dictionary contains a set of key value pairs. The key names should match the key names in the document. If they match the values will be updated in the document. If they don’t new key value pairs will be created in the document. Also note in the call to our form we use the values of ‘name’ and ‘age’ these should match the “id” attribute of your <input /> tags in your HTML form.
- Line 108 is where we return the redirect response. In this case we will redirect to “/” but we also include a HTTP 302 status code to indicate that the operation was a success and we are redirecting the user as a result of this.

10) In main.html add the following code

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Firestore basics</title>
5          <link type="text/css" href="{{ url_for('static', path='/styles.css') }}" rel="stylesheet"/>
6          <script type="module" src="{{ url_for('static', path='/firebase-login.js') }}"></script>
7      </head>
8      <body>
9          <div id="login-box" hidden="true">
10             Email:<input type="email" name="" id="email"><br/>
11             Password: <input type="password" name="" id="password"><br/>
12             <button id="login">Login</button>
13             <button id="sign-up">Sign Up</button>
14         </div>
15         <button id="sign-out" hidden="true">Sign out</button>
16
17         <!-- if we have a logged in user then show the user email address from the user_token object that was passe
18             we will also show the user document that has a name and age -->
19         {% if user_token %}
20             <p>User email: {{ user_token.email }}</p>
21             <p>error message: {{ error_message }}</p>
22             <p>name: {{ user_info.get('name') }}</p>
23             <p>age: {{ user_info.get('age') }}</p>
24
25             <!-- show a link to update the name and age -->
26             <a href="/update-user">Update name and age here</a>
27         {% endif %}
28     </body>
29 </html>
```

Most of the content here is the same as the previous example. The differences are in lines 22 to 26

- Lines 22 and 23 uses the user object that was pulled from Firestore and passed to this template to extract the name and age of the user for display

- Line 26 adds in a hyperlink to our “/update-user” route. Hyperlinks by default will use a HTTP GET verb on whatever URL is provided so the code you defined in step 08 will be called when this link is clicked.

11) in update.html add the following code

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Firestore basics</title>
5          <link type="text/css" href="{{ url_for('static', path='/styles.css') }}" rel="stylesheet"/>
6          <script type="module" src="{{ url_for('static', path='/firebase-login.js') }}"></script>
7      </head>
8      <body>
9          <div id="login-box" hidden="true">
10             Email:<input type="email" name="" id="email"><br/>
11             Password: <input type="password" name="" id="password"><br/>
12             <button id="login">Login</button>
13             <button id="sign-up">Sign Up</button>
14         </div>
15         <button id="sign-out" hidden="true">Sign out</button>
16
17         <!-- if we have a logged in user then show the user email address from the user_token object that was passe
18             we will also show the user document that has a name and age -->
19         {% if user_token %}
20             <p>User email: {{ user_token.email }}</p>
21             <form action="/update-user" method="post">
22                 Name:<input type="text" id="name" name="name" value="{{ user_info.get('name') }}"/><br/>
23                 Age:<input type="number" id="age" name="age" value="{{ user_info.get('age') }}"/><br/>
24                 <input type="submit" value="Update"/>
25                 <input type="submit" value="Cancel"/>
26             </form>
27         {% endif %}
28     </body>
29 </html>
```

The template provided here is very similar to the previous step. The differences being in lines 21 to 26

- Line 21 defines the form. The action attribute states which URL this form is sent to (“/update-user”), while the method attribute states which HTTP verb to use in this case POST. Thus when the client clicks the “Update” button this will be sent to the code that you defined in step 09.
- Lines 22 and 23 take in the name and age data that are currently set for this user and set this as the initial values for the form. Generally when editing a form it’s a good idea to populate it with the currently set values so the user does not need to remember them
- Lines 24 and 25 set update and cancel buttons on the form.

12) before running the code with uvicorn from the command line run the following command first, replacing <filename> with the name of the JSON service account file you downloaded prior to this example:

For Linux and Mac OS Users:

export GOOGLE_APPLICATION_CREDENTIALS=”../<filename>”

for Windows CMD users:

set GOOGLE_APPLICATION_CREDENTIALS=”../<filename>”

for Windows PowerShell users:

```
$Env:GOOGLE_APPLICATION_CREDENTIALS="../<filename>"
```

Note that the relative path above assumes you are using the same directory structure as was outlined at the beginning of the book.

Example 05: Datatypes available in Firestore

In this example you will see the datatypes the Firestore supports. The full list of datatypes are:

- strings: single dimension character based data
- integers: numerical values with no decimal point
- floats: numerical values with a decimal point
- booleans: true of false values
- datetimes: a full description of a point in time in ISO8601 format. Year, month, day, hour, second, millisecond
- geo-points: a latitude and longitude pair that represent a location on the surface of Earth.
- Arrays: allow you to store a related collection of objects in a linear indexable structure
- maps: key value pair system to store a collection of objects each under their own name (key)

This example will create a set of default values for each type when a user logs in. It will then retrieve those values and display them through a template

NOTE: Before you do this example you will want to empty your Firestore first, otherwise old data with different fields may prevent this example from working. To empty your Firestore goto http://console.cloud.google.com/firestore/ and delete your collections

01) Create an Example 05 directory in your examples directory and give it the following structure

- static/
  - firebase-login.js
  - styles.css
- templates/
  - main.html
- main.py

02) in Main.py add the following imports and setup code

```python
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
import google.oauth2.id_token;
from google.auth.transport import requests
from google.cloud import firestore
from typing import Union
import starlette.status as status
import datetime

# define the app that will contain all of our routing for Fast API
app = FastAPI()

# define a firestore client so we can interact with our database
firestore_db = firestore.Client()

# we need a request object to be able to talk to firebase for verifying user logins
firebase_request_adapter = requests.Request()

# define the static and templates directories
app.mount('/static', StaticFiles(directory='static'), name='static')
templates = Jinja2Templates(directory="templates")
```

Similar imports as before nothing really new here.

03) in Main.py add the following function

```python
# function that we will use to retrieve and return the document that represents this user
# by using the ID of the firebase credentials. this function assumes that the credentials have
# been checked first
def getUser(user_token):
    # now that we have a user token we are going to try and retrieve a user object for this user from firestore if there
    # is not a user object for this user we will create one
    user = firestore_db.collection('users').document(user_token['user_id'])
    if not user.get().exists:
        user_data = {
            # our signup form doesn't have a name field so we will set a default that we will edit later
            'string': 'No name yet',
            'int': 0,
            'float': 3.14159,
            'boolean': True,
            'datetime': datetime.datetime.now(),
            'geo-point': firestore.GeoPoint(53.3314, -6.277804),
            'array': [5, 6, 7, 8],
            'map': {"first": "hello", "second": "world"}
        }
        firestore_db.collection('users').document(user_token['user_id']).set(user_data)

    # return the user document
    return user
```

This is similar to the getUser() function from the previous example. The main difference here being the data that is stored for the user.

- Lines 35 and 36 are the string and integer types that you saw in the previous examples. Lines 37 and 38 show floating point and boolean values and how they are stored.
- Line 39 is the storage of a datetime object. This will not accept datetimes in the form of strings it must be a datetime object. Datetime has mechanisms to create datetime objects and convert ISO8601 strings into datetime objects.

- Line 40 is a Firestore geopoint. Two floating point values must be provided here with the first being latitude and the last being longitude. This particular geopoint points to Griffith College Dublin.
- Line 41 is an array which is done in the form of a python list. While this can (line python lists) mix different types in the same list, it is recommended from a programming perspective to keep the same types in a list
- Line 42 is a map which is done in the form of a python dictionary. This is a key value pair system where each key can have a different type of value associated with it.

04) in Main.py add the following function

```
49  # function that we will use to validate an id_token. will return the user_token if valid, None if not
50  def validateFirebaseToken(id_token):
51      # if we dont have a token then return None
52      if not id_token:
53          return None
54
55      # try to validate the token if this fails with an exception then this will remain as None so just return at the end
56      # if we get an exception then log the exception before returning
57      user_token = None
58      try:
59          user_token = google.oauth2.id_token.verify_firebase_token(id_token, firebase_request_adapter)
60      except ValueError as err:
61          # dump this message to console as it will not be displayed on the template. use for debugging but if you are building for
62          # production you should handle this much more gracefully.
63          print(str(err))
64
65      # return the token to the caller
66      return user_token
```

Same function as was used in the previous example so no change here.

05) in Main.py add the following function

```
69  @app.get("/", response_class=HTMLResponse)
70  async def root(request: Request):
71      # query firebase for the request token. We also declare a bunch of other variables here as we will need them
72      # for rendering the template at the end. we have an error_message there in case you want to output an error to
73      # the user in the template.
74      id_token = request.cookies.get("token")
75      error_message = "No error here"
76      user_token = None
77      user = None
78
79      # check if we have a valid firebase login if not return the template with empty data as we will show the login box
80      user_token = validateFirebaseToken(id_token)
81      if not user_token:
82          return templates.TemplateResponse('main.html', {'request': request, 'user_token': None, 'error_message': None, 'user_info': None})
83
84      # get the user document and render the template
85      user = getUser(user_token)
86      return templates.TemplateResponse('main.html', {'request': request, 'user_token': user_token, 'error_message': error_message, 'user_info': user.get()})
```

Essentially the same “/” route as the previous example where we first validate the user and redirect them back to “/” if there is no valid login. If valid we pull the user document from Firestore and pass it to the template to generate a response.

06) add the following code to main.html

```
1  <!DOCTYPE html>
2  <html>
3      <head>
4          <title>Firestore basics</title>
5          <link type="text/css" href="{{ url_for('static', path='/styles.css') }}" rel="stylesheet"/>
6          <script type="module" src="{{ url_for('static', path='/firebase-login.js') }}"></script>
7      </head>
8      <body>
9          <div id="login-box" hidden="true">
10             Email:<input type="email" name="" id="email"><br/>
11             Password: <input type="password" name="" id="password"><br/>
12             <button id="login">Login</button>
13             <button id="sign-up">Sign Up</button>
14         </div>
15         <button id="sign-out" hidden="true">Sign out</button>
16
17         <!-- if we have a logged in user then show the user email address from the user_token object that was passed in
18             we will also show the user document that has a name and age -->
19         {% if user_token %}
20             <p>User email: {{ user_token.email }}</p>
21             <p>error message: {{ error_message }}</p>
22             <p>String type: {{ user_info.get('string') }}</p>
23             <p>Int type: {{ user_info.get('int') }}</p>
24             <p>Float type: {{ user_info.get('float') }}</p>
25             <p>Boolean type: {{ user_info.get('boolean') }}</p>
26             <p>Datetime type: {{ user_info.get('datetime') }}</p>
27             {% for value in user_info.get('array') %}
28                 <p>Array index {{ loop.index0 }} {{ loop.index }} {{ value }}</p>
29             {% endfor %}
30
31             {% for key, value in user_info.get('map').items() %}
32                 <p>{{ key }}: {{ value }} </p>
33             {% endfor %}
34
35         {% endif %}
36     </body>
37 </html>
```

Most of this is the same as the previous example bar the display code between lines 19 and 35

- Most of types can be rendered directly in the browser as can bee see in lines 20 to 26. For array and map types however, these cannot be rendered directly. They must be iterated through and rendered one by one.
- Lines 27 to 29 show the Jinja for loop syntax to render an array. This for loop will take a copy of each value and render it directly. Note the user of loop.index0 and loop.index. These will give you the current iteration of the loop. Loop.index0 will give the iteration number starting from 0 while loop.index will give the iteration number starting from 1. They will always have this difference of 1
- Lines 31 to 33 show the Jinja for loop syntax to render a map. Note how for each iteration we must pull out both the key and its associated value.

07) run the application and you should see the datatypes appear when a user logs in

Example 06: Linking firestore documents together through keys

Up to this point we have only considered single objects with Firestore. However, to make any kind of database useful we need to be able to link multiple objects together through some form of referencing system. If you come from an SQL background you will be familiar with the idea of using foreign keys. Where we take a copy of the primary key of a different entity to store with our row, such that if we need to reference that object we can pass the foreign key to a query and retrieve the associated row.

We can use a similar system in Firestore wherein if we have a copy of the ID of the other document we wish to link to, we can store the ID of the other document in the current document. This is useful for performance in Firestore. By using the ID of the document we can perform direct key access and retrieve the document much quicker than if we used a query (you will see queries in a later example). The other performance benefit is you get to decide how many of the documents you will pull at any given time. e.g. if I have 100 keys linking to other documents I may only need 10 at a time for something like pagination. This will make more sense after the next example where we use maps/sub-documents to store related documents.

In this example we will create a simple address book where each of the addresses will be stored as a seperate document and we will link them with keys. We will pull all of the address objects through direct key access

NOTE: like the previous example clear out your firestore before running this application otherwise you will get unexpected errors when you try to run it.

01) create an Example06 directory in your examples directory and give it the following structure:

- static/
  - firebase-login.js
  - styles.css
- templates/
  - main.html
- main.py

02) use firebase-login.js and styles.css from the previous examples

03) in main.py add the following inputs and initialisation code. There is nothing different to the previous examples here

```python
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse, RedirectResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
import google.oauth2.id_token;
from google.auth.transport import requests
from google.cloud import firestore
import starlette.status as status

# define the app that will contain all of our routing for Fast API
app = FastAPI()

# define a firestore client so we can interact with our database
firestore_db = firestore.Client()

# we need a request object to be able to talk to firebase for verifying user logins
firebase_request_adapter = requests.Request()

# define the static and templates directories
app.mount('/static', StaticFiles(directory='static'), name='static')
templates = Jinja2Templates(directory="templates")
```

04) in main.py add the following getUser() function

```python
# function that we will use to retrieve and return the document that represents this user
# by using the ID of the firebase credentials. this function assumes that the credentials have
# been checked first
def getUser(user_token):
    # now that we have a user token we are going to try and retrieve a user object for this user from firestore if there
    # is not a user object for this user we will create one
    user = firestore_db.collection('users').document(user_token['user_id'])
    if not user.get().exists:
        user_data = {
            # for now we will use a placeholder name as this is not our focus but we will start with an empty array for our addresses
            'name': 'John Doe',
            'address_list': []
        }
        firestore_db.collection('users').document(user_token['user_id']).set(user_data)

    # return the user document
    return user
```

The main difference in data here is the address_list array. We will use this to store keys that directly reference address documents that the user has created. Every time we create an address we will take a copy of its key and append it to address_list

05) in main.py add the following validateFirebaseToken function that you have used in previous examples.

```python
# function that we will use to validate an id_token. will return the user_token if valid, None if not
def validateFirebaseToken(id_token):
    # if we dont have a token then return None
    if not id_token:
        return None

    # try to validate the token if this fails with an exception then this will remain as None so just return at the end
    # if we get an exception then log the exception before returning
    user_token = None
    try:
        user_token = google.oauth2.id_token.verify_firebase_token(id_token, firebase_request_adapter)
    except ValueError as err:
        # dump this message to console as it will not be displayed on the template. use for debugging but if you are building for
        # production you should handle this much more gracefully.
        print(str(err))

    # return the token to the caller
    return user_token
```

06) in main.py add the following function to handle the “/” route

```python
@app.get("/", response_class=HTMLResponse)
async def root(request: Request):
    # query firebase for the request token. We also declare a bunch of other variables here as we will need them
    # for rendering the template at the end. we have an error_message there in case you want to output an error to
    # the user in the template.
    id_token = request.cookies.get("token")
    error_message = "No error here"
    user_token = None
    user = None

    # check if we have a valid firebase login if not return the template with empty data as we will show the login box
    user_token = validateFirebaseToken(id_token)
    if not user_token:
        return templates.TemplateResponse('main.html', {'request': request, 'user_token': None, 'error_message': None, 'user_info': None})

    # get the user document and render the template we will need to pull the address objects as well
    # you can use get_all as well however it will not guarantee order. If order does not matter then use get_all()
    user = getUser(user_token).get()
    addresses = []
    for address in user.get('address_list'):
        addresses.append(address.get())
    return templates.TemplateResponse('main.html', {'request': request, 'user_token': user_token, 'error_message': error_message, 'user_info': user, 'address_list': addresses})
```

There are a few small differences here in lines 78 to 82 compared to previous similar functions you have seen. The beginning of the function is the same where we declare our variables and validate our user tokens. In lines 78 to 82 however, we will not only retrieve our user but also all address objects linked to them.

- Line 79 creates an empty list for holding our address documents.
- Lines 80 and 81 will take each individual key from the address_list parameter of the user document and will retrieve those documents one by one and append them to addresses.
- Line 82 will then take the user document and the collection of address documents and pass them to the main.html templates for display to the client.

07) in main.py add the following function to handle a POST verb to the “/add-address” route

```python
# route that will take in an address form and will add it to the firestore and link it to a user
# this will use a new firebase document and reference to connect it to the user
@app.post("/add-address", response_class=RedirectResponse)
async def addAddress(request: Request):
    # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
    id_token = request.cookies.get("token")
    user_token = validateFirebaseToken(id_token)
    if not user_token:
        return RedirectResponse('/')

    # pull the form containing our data
    form = await request.form()

    # create a reference to an address object note that we have not given an ID here
    # we are asking firestore to create an ID for us
    address_ref = firestore_db.collection('address').document()

    # set the data on the address object
    address_ref.set({
        'address1': form['address1'],
        'address2': form['address2'],
        'address3': form['address3'],
        'address4': form['address4']
    })

    # add the address to our current user
    user = getUser(user_token)
    addresses = user.get().get('address_list')
    addresses.append(address_ref)
    user.update({'address_list': addresses})

    # when finished return a redirect with a 302 to force a GET verb
    return RedirectResponse('/', status_code=status.HTTP_302_FOUND)
```

In this route we will first validate our user and redirect them to “/” if they are not a valid user in lines 89 to 92.

- Line 95 will pull the data from the form and like before if this is delayed FastAPI will temporarily suspend addAddress() until it returns.
- Line 99 will then create a reference to a new address object. Note that in the call to .document() we have not provided an ID. In this case Firestore will create a random id for us and store it in address_ref
- Lines 102 to 107 will take the four lines of address information that will be entered into the form and add them to an address document. The .set() function will commit these changes to Firestore
- Lines 110 to 114 will then pull the user document and extract its address_list and append the reference to the address to this list. Finally we will update the address_list in the user document and update this in Firestore
- Finally we will redirect back to "/" wherein the new address should be rendered along all previous addresses

08) in main.py add the following function to handle a POST verb to the "/update-address" route

```
118  # route that will take in an index for an address object to be deleted from firestore and the reference be removed from the user
119  @app.post("/delete-address", response_class=RedirectResponse)
120  async def deleteAddress(request: Request):
121      # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
122      id_token = request.cookies.get("token")
123      user_token = validateFirebaseToken(id_token)
124      if not user_token:
125          return RedirectResponse('/')
126
127      # pull the index from our form
128      form = await request.form()
129      index = int(form['index'])
130
131      # pull the list of address objects from the user delete the requested index and update the user
132      user = getUser(user_token)
133      addresses = user.get().get('address_list')
134      addresses[int(index)].delete()
135      del addresses[int(index)]
136      data = {
137          'address_list': addresses
138      }
139      user.update(data)
140
141      # when finished return a redirect with a 302 to force a get verb
142      return RedirectResponse('/', status_code=status.HTTP_302_FOUND)
```

Here we have a route that will take an index to an address object and will delete the address document from Firestore and will delete the reference from the user document as well.

- Lines 122 to 129 does the usual user validation before getting the form from the browser. When the form is returned we pull the value of the index attribute from it. The index will indicate which specific address the user wishes to remove. These indexes will be setup by the template in the next step.
- lines 132 to 139 will then delete the document and reference. After we pull the user from Firestore we then pull the address list from the user document and pull the required address reference using the index. We then delete the document first before we delete the reference form the list. Finally we update the list in the user and update the user before redirecting back to "/"

09) add the following code to main.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>Firestore basics</title>
        <link type="text/css" href="{{ url_for('static', path='/styles.css') }}" rel="stylesheet"/>
        <script type="module" src="{{ url_for('static', path='/firebase-login.js') }}"></script>
    </head>
    <body>
        <div id="login-box" hidden="true">
            Email:<input type="email" name="" id="email"><br/>
            Password: <input type="password" name="" id="password"><br/>
            <button id="login">Login</button>
            <button id="sign-up">Sign Up</button>
        </div>
        <button id="sign-out" hidden="true">Sign out</button>

        <!-- if we have a logged in user then show the user email address from the user_token object that was passed in
            we will also show the user document that has a name and age -->
        {% if user_token %}
            <p>User email: {{ user_token.email }}</p>
            <p>error message: {{ error_message }}</p>
            <p>Name: {{ user_info.get('name') }}</p>

            <!-- form that we will use to add in an address to our user -->
            <form action="/add-address" method="post">
                Address Line 1:<input type="text" name="address1"/><br/>
                Address Line 2:<input type="text" name="address2"/><br/>
                Address Line 3:<input type="text" name="address3"/><br/>
                Address Line 4:<input type="text" name="address4"/><br/>
                <input type="submit" value="Add Address">
            </form>

            <p>Address list</p>
            {% for address in address_list %}
                <p>Array index {{ loop.index0 }} <p>
                Address line 1:{{ address.get('address1') }}<br/>
                Address line 2:{{ address.get('address2') }}<br/>
                Address line 3:{{ address.get('address3') }}<br/>
                Address line 4:{{ address.get('address4') }}<br/>
                <form action="/delete-address" method="post">
                    <input type="hidden" value="{{ loop.index0 }}" name="index"/>
                    <input type="submit" value="Delete Address"/>
                </form>
            {% endfor %}
        {% endif %}
    </body>
</html>
```

The above is similar to what you have seen previously but with the addition of two forms: one to add a new address to the system and one to delete an address.

- Lines 25 to 31 deal with addin an address to the system. This is similar to forms you've seen previously were we take in data through a form and pass it through a POST to the "/add-address" URL. This will then be passed to the function you defined in step 07
- Lines 34 to 44 deal with displaying the addresses and also providing an option for deleting individual addresses. In this we take each address in turn and display it's contents (lines 35 to 39). However, for each address we then add a form to it containing a submit button that will send a POST to the "/delete-address" URL which will trigger the function you defined in step 08. However, we need to indicate which of the addresses we wish to delete. This is why we have the hidden parameter in line 41. By using loop.index0 we will apply a unique index to each address that will match the indexes in the array of addresses that was passed to the template.

10) run the application and you should be able to add and remove addresses from a user's address book

Example 07: Linking firestore documents together through the use of maps/subdocuments.

The previous example showed how to link multiple documents together using keys. It's a general purpose way of linking documents together and works well for many to many relationships. However, there are situations were you may have documents that will only ever belong to one other doument and they may not be shared with other documents. In cases like this it may be better to embed the document within the owning document. To show you the difference between the two forms we will take the previous example and will convert the addresses to be contained within the user document. This makes adding, updating, deleting of addresses a little quicker, however you would not be able to link the same address to multiple users or run general queries on all addresses in the system, sharing addresses between mutliple users would also be a little harder.

Be aware however that Firestore enforces a restriction with subdocuments. A document may only have one level of sub documents. It is not possible to have subdocuments within subdocuments

NOTE: like the previous example clear out your firestore before running this application otherwise you will get unexpected errors when you try to run it.

01) Create a new directory Example 07 in your examples directory.

02) in your directory create the following structure

- static/
  - firebase-login.js
  - styles.css
- templates/
  - main.html
- main.py

03) use firebase-login.js and styles.css from the previous example

04) in main.py add the following imports and setup code

```python
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse, RedirectResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
import google.oauth2.id_token;
from google.auth.transport import requests
from google.cloud import firestore
import starlette.status as status

# define the app that will contain all of our routing for Fast API
app = FastAPI()

# define a firestore client so we can interact with our database
firestore_db = firestore.Client()

# we need a request object to be able to talk to firebase for verifying user logins
firebase_request_adapter = requests.Request()

# define the static and templates directories
app.mount('/static', StaticFiles(directory='static'), name='static')
templates = Jinja2Templates(directory="templates")
```

05) in main.py add the following getUser() function which is the same as the previous example

```python
# function that we will use to retrieve and return the document that represents this user
# by using the ID of the firebase credentials. this function assumes that the credentials have
# been checked first
def getUser(user_token):
    # now that we have a user token we are going to try and retrieve a user object for this user from firestore if there
    # is not a user object for this user we will create one
    user = firestore_db.collection('users').document(user_token['user_id'])
    if not user.get().exists:
        user_data = {
            # for now we will use a placeholder name as this is not our focus but we will start with an empty array for our addresses
            'name': 'John Doe',
            'address_list': []
        }
        firestore_db.collection('users').document(user_token['user_id']).set(user_data)

    # return the user document
    return user
```

06) in main.py add the following validateFirebaseToken() function which is the same as the previous example

```python
# function that we will use to validate an id_token. will return the user_token if valid, None if not
def validateFirebaseToken(id_token):
    # if we dont have a token then return None
    if not id_token:
        return None

    # try to validate the token if this fails with an exception then this will remain as None so just return at the end
    # if we get an exception then log the exception before returning
    user_token = None
    try:
        user_token = google.oauth2.id_token.verify_firebase_token(id_token, firebase_request_adapter)
    except ValueError as err:
        # dump this message to console as it will not be displayed on the template. use for debugging but if you are building for
        # production you should handle this much more gracefully.
        print(str(err))

    # return the token to the caller
    return user_token
```

07) in main.py add the following root() function, which is slightly different from the previous example

```python
@app.get("/", response_class=HTMLResponse)
async def root(request: Request):
    # query firebase for the request token. We also declare a bunch of other variables here as we will need them
    # for rendering the template at the end. we have an error_message there in case you want to output an error to
    # the user in the template.
    id_token = request.cookies.get("token")
    error_message = "No error here"
    user_token = None
    user = None

    # check if we have a valid firebase login if not return the template with empty data as we will show the login box
    user_token = validateFirebaseToken(id_token)
    if not user_token:
        return templates.TemplateResponse('main.html', {'request': request, 'user_token': None, 'error_message': None, 'user_info': None})

    # get the user document and render the template we will need to pull the address objects as well
    # you can use get_all as well however it will not guarantee order. If order does not matter then use get_all()
    user = getUser(user_token).get()
    addresses = user.get('address_list')
    return templates.TemplateResponse('main.html', {'request': request, 'user_token': user_token, 'error_message': error_message, 'user_info': user, 'address_list': addresses})
```

The main difference here is in lines 78 and 79 because we are using subdocuments to store the addresses we do not need to pull the address documents key by key. We can pull them directly from the user document as the whole address is there.

08) in main.py add the following addAddress() function, which is slightly different from the previous example

```python
# route that will take in an address form and will add it to the firestore and link it to a user
@app.post("/add-address", response_class=RedirectResponse)
async def addAddress(request: Request):
    # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
    id_token = request.cookies.get("token")
    user_token = validateFirebaseToken(id_token)
    if not user_token:
        return RedirectResponse('/')

    # pull the form from the request
    form = await request.form()

    # create a dictionary object that will represent our address as a map we will add to the user
    address = {
        'address1': form['address1'],
        'address2': form['address2'],
        'address3': form['address3'],
        'address4': form['address4']
    }

    # add the address to our current user as a map
    user = getUser(user_token)
    addresses = user.get().get('address_list')
    addresses.append(address)
    user.update({'address_list': addresses})

    # when finished return a redirect with a 302 to force a GET verb
    return RedirectResponse('/', status_code=status.HTTP_302_FOUND)
```

Like hte previous step the main difference here is in lines 103 to 106. As we are storing a new address as a sub document within the user document we have no need to generate a key, store the address document, and then setup the reference in the user document.

09) in main.py add the following deleteAddress() function, which is slightly different from the previous example

```python
@app.post("/delete-address", response_class=RedirectResponse)
async def deleteAddress(request: Request):
    # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
    id_token = request.cookies.get("token")
    user_token = validateFirebaseToken(id_token)
    if not user_token:
        return RedirectResponse('/')

    # pull the form from the request
    form = await request.form()
    index = form['index']

    # pull the list of address objects from the user delete the requested index and update the user
    user = getUser(user_token)
    addresses = user.get().get('address_list')
    del addresses[int(index)]
    data = {
        'address_list': addresses
    }
    user.update(data)

    # when finished return a redirect with a 302 to force a get verb
    return RedirectResponse('/', status_code=status.HTTP_302_FOUND)
```

Similar to the previous step there is no need to first get the reference then delete the document before deleting the reference. As the document is embedded in the user document we simply remove the subdocument and update the user and the address no longer exists.

10) add the following code to main.html which is the same as the previous example

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Firestore basics</title>
        <link type="text/css" href="{{ url_for('static', path='/styles.css') }}" rel="stylesheet"/>
        <script type="module" src="{{ url_for('static', path='/firebase-login.js') }}"></script>
    </head>
    <body>
        <div id="login-box" hidden="true">
            Email:<input type="email" name="" id="email"><br/>
            Password: <input type="password" name="" id="password"><br/>
            <button id="login">Login</button>
            <button id="sign-up">Sign Up</button>
        </div>
        <button id="sign-out" hidden="true">Sign out</button>

        <!-- if we have a logged in user then show the user email address from the user_token object that was passed in
            we will also show the user document that has a name and age -->
        {% if user_token %}
            <p>User email: {{ user_token.email }}</p>
            <p>error message: {{ error_message }}</p>
            <p>Name: {{ user_info.get('name') }}</p>

            <!-- form that we will use to add in an address to our user -->
            <form action="/add-address" method="post">
                Address Line 1:<input type="text" name="address1"/><br/>
                Address Line 2:<input type="text" name="address2"/><br/>
                Address Line 3:<input type="text" name="address3"/><br/>
                Address Line 4:<input type="text" name="address4"/><br/>
                <input type="submit" value="Add Address">
            </form>

            <p>Address list</p>
            {% for address in address_list %}
                <p>Array index {{ loop.index0 }} <p>
                Address line 1:{{ address.get('address1') }}<br/>
                Address line 2:{{ address.get('address2') }}<br/>
                Address line 3:{{ address.get('address3') }}<br/>
                Address line 4:{{ address.get('address4') }}<br/>
                <form action="/delete-address" method="post">
                    <input type="hidden" value="{{ loop.index0 }}" name="index"/>
                    <input type="submit" value="Delete Address"/>
                </form>
            {% endfor %}
        {% endif %}
    </body>
</html>
```

11) run the application and it should function the exact same as the previous example

Example 08: Transactions and Batch operations in Firestore

While Firestore is a relatively quick database to work with there are things we can do to improve the performance of our applications. Particularly when it comes to retrieving, updating, or deleting data. Upto this point we have modified documents individually one by one. While this will suffice for small sets of data transfers it would soon become a performance bottleneck when your application starts to experience more demand due to more users. Firestore introduces batch operations that allow you to group a series of requests into one. Firestore will then take this batch as a single unit and will then run the operations in batch in one go.

The other situation that is useful in a database is transactions. This is where you need to enforce data consistency. A transaction will take a set of operations and will attempt to complete all of them without fail. Should one of the operations fail the transaction will revert Firestore to the state it was in before any transaction operations were attempted. Thus if the transaction succeeds or fails Firestore will remain in a consistent state.

NOTE: like the previous example clear out your firestore before running this application otherwise you will get unexpected errors when you try to run it.

01) Create a new directory Example 08 in your examples directory.

02) in your directory create the following structure

- static/
  - firebase-login.js
  - styles.css
- templates/
  - main.html
- main.py

03) use firebase-login.js and styles.css from the previous example

04) in main.py add the following import and setup code. This is the same as previous examples

```python
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse, RedirectResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
import google.oauth2.id_token;
from google.auth.transport import requests
from google.cloud import firestore
import starlette.status as status

# define the app that will contain all of our routing for Fast API
app = FastAPI()

# define a firestore client so we can interact with our database
firestore_db = firestore.Client()

# we need a request object to be able to talk to firebase for verifying user logins
firebase_request_adapter = requests.Request()

# define the static and templates directories
app.mount('/static', StaticFiles(directory='static'), name='static')
templates = Jinja2Templates(directory="templates")
```

05) in main.py add the following getUser() function. Note that we only store a name here as we will be using a different document collection for our batch and transaction operations

```python
# function that we will use to retrieve and return the document that represents this user
# by using the ID of the firebase credentials. this function assumes that the credentials have
# been checked first
def getUser(user_token):
    # now that we have a user token we are going to try and retrieve a user object for this user from firestore if there
    # is not a user object for this user we will create one
    user = firestore_db.collection('users').document(user_token['user_id'])
    if not user.get().exists:
        user_data = {
            # we won't use this but just to ensure some data in our user document
            'name': 'John Doe'
        }
        firestore_db.collection('users').document(user_token['user_id']).set(user_data)

    # return the user document
    return user
```

06) in main.py add the following validateFirebaseToken() function that is the same as previous examples.

```python
# function that we will use to validate an id_token. will return the user_token if valid, None if not
def validateFirebaseToken(id_token):
    # if we dont have a token then return None
    if not id_token:
        return None

    # try to validate the token if this fails with an exception then this will remain as None so just return at the end
    # if we get an exception then log the exception before returning
    user_token = None
    try:
        user_token = google.oauth2.id_token.verify_firebase_token(id_token, firebase_request_adapter)
    except ValueError as err:
        # dump this message to console as it will not be displayed on the template. use for debugging but if you are building for
        # production you should handle this much more gracefully.
        print(str(err))

    # return the token to the caller
    return user_token
```

07) in main.py add the following root() function

```
60  @app.get("/", response_class=HTMLResponse)
61  async def root(request: Request):
62      # query firebase for the request token. We also declare a bunch of other variables here as we will need them
63      # for rendering the template at the end. we have an error_message there in case you want to output an error to
64      # the user in the template.
65      id_token = request.cookies.get("token")
66      error_message = "No error here"
67      user_token = None
68      user = None
69
70      # check if we have a valid firebase login if not return the template with empty data as we will show the login box
71      user_token = validateFirebaseToken(id_token)
72      if not user_token:
73          return templates.TemplateResponse('main.html', {'request': request, 'user_token': None, 'error_message': None, 'user_info': None})
74
75      # get the user document and render the template we will need to pull the address objects as well
76      # you can use get_all as well however it will not guarantee order. If order does not matter then use get_all()
77      user = getUser(user_token).get()
78      dummy_data = firestore_db.collection('dummy-data').stream()
79      return templates.TemplateResponse('main.html', {'request': request, 'user_token': user_token, 'error_message': error_message, 'user_info': user, 'dummy_data': dummy_data})
```

The first part of the code from lines 65 to 73 are the same as previous examples were the validate the user token and render the login box if a user is not logged in. Where things change is on line 78 where we use the .stream() function on a collection rather than .get()

- If we were to use .get() without providing a document or a key it will return all documents in that collection at once. This will have two effects
    - A large spike in network bandwidth as all the documents are returned
    - A large spike in memory usage to store all the documents on the server while they are needed
- Using .stream() instead allows us to consume each document one by one which will reduce these effects. Thus if you have a large document collection that you need to go through use stream() instead of get() as this will smooth out network access and memory usage

08) in main.py add the following batchAdd() function

```
81  # route that will add four objects to the firestore by using a batch request. the idea is to add them in a single operation
82  # rather than four individual objects
83  @app.post("/batch-add", response_class=RedirectResponse)
84  async def batchAdd(request: Request):
85      # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
86      id_token = request.cookies.get("token")
87      user_token = validateFirebaseToken(id_token)
88      if not user_token:
89          return RedirectResponse('/')
90
91      # create dictionaries for four documents to add into the firestore
92      batch1 = {'name': 'first'}
93      batch2 = {'name': 'second'}
94      batch3 = {'name': 'third'}
95      batch4 = {'name': 'fourth'}
96
97      # get a batch object and add the objects
98      batch = firestore_db.batch()
99      batch.set(firestore_db.collection('dummy-data').document('1'), batch1)
100     batch.set(firestore_db.collection('dummy-data').document('2'), batch2)
101     batch.set(firestore_db.collection('dummy-data').document('3'), batch3)
102     batch.set(firestore_db.collection('dummy-data').document('4'), batch4)
103
104     # commit the batch and it should be added to the firestore
105     batch.commit()
106
107     # when finished redirect with a 302 to force a GET request back to /
108     return RedirectResponse('/', status_code=status.HTTP_302_FOUND)
```

This is how a batch operation is performed. Note that unlike transactions batch writes can partially fail and won't automatically rollback to the last good state. Generally this should be used for something where performance is required but potentially losing some data consistency is not problematic.

- Lines 86 to 89 are are usual validation code that will redirect to “/” if there is not a user logged in.
- Lines 92 to 95 is where we are defininining the set of attributes for each document that we wish to write to firestore.
- Line 98 is where we request a batch object from firestore. This is the object we need to attach all writes to before they are committed. Note that none of the set operations on the following lines will write to firestore until the commit() function is called on line 105
- Lines 98 to 102 declare four documents each with their own identifier, and set each of the dictionaries of attributes on each document. These are not written to firestore until the commit() function is called on line 105

09) in main.py add the following transactionAdd() function

```
110 # route that will add four objects to the firestore by using a transaction request. the idea is to add them in a
111 # single operation rather than four individual objects
112 @app.post("/transaction-add", response_class=RedirectResponse)
113 async def transactionAdd(request: Request):
114     # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
115     id_token = request.cookies.get("token")
116     user_token = validateFirebaseToken(id_token)
117     if not user_token:
118         return RedirectResponse('/')
119 
120     # create dictionaries for four documents to add into the firestore
121     transaction1 = {'name': 'fifth'}
122     transaction2 = {'name': 'sixth'}
123     transaction3 = {'name': 'seventh'}
124     transaction4 = {'name': 'eighth'}
125 
126     # get a transaction object and add the objects
127     transaction = firestore_db.transaction()
128     transaction.set(firestore_db.collection('dummy-data').document('5'), transaction1)
129     transaction.set(firestore_db.collection('dummy-data').document('6'), transaction2)
130     transaction.set(firestore_db.collection('dummy-data').document('7'), transaction3)
131     transaction.set(firestore_db.collection('dummy-data').document('8'), transaction4)
132 
133     # commit the transaction to the firestore
134     transaction.commit()
135 
136     # when finished redirect with a 302 to force a GET request back to /
137     return RedirectResponse('/', status_code=status.HTTP_302_FOUND)
```

Structurally this is almost the same as the previous example however with a transaction you are guaranteed consistency of data. If one operation fails, the transaction stops and any previously completed operations in the transaction are rolled back to their previous state before the transaction began

10) in main.py add the following batchDelete() function

```python
# route that will delete the first four objects using a batch operation
@app.post("/batch-delete", response_class=RedirectResponse)
async def batchDelete(request: Request):
    # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
    id_token = request.cookies.get("token")
    user_token = validateFirebaseToken(id_token)
    if not user_token:
        return RedirectResponse('/')

    # get a batch object and add the objects
    batch = firestore_db.batch()
    batch.delete(firestore_db.collection('dummy-data').document('1'))
    batch.delete(firestore_db.collection('dummy-data').document('2'))
    batch.delete(firestore_db.collection('dummy-data').document('3'))
    batch.delete(firestore_db.collection('dummy-data').document('4'))

    # commit the batch and it should be added to the firestore
    batch.commit()

    # when finished redirect with a 302 to force a GET request back to /
    return RedirectResponse('/', status_code=status.HTTP_302_FOUND)
```

A batch delete operation. Only difference here and 08 above is that were are using .delete() instead of .set() Should you wish to remove documents or collections from firestore this is a good performance orientated way of doing it.

11) in main.py add the following transactionDelete() function

```python
# route that will delete the last four objects using a transaction operation
@app.post("/transaction-delete", response_class=RedirectResponse)
async def transactionDelete(request: Request):
    # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
    id_token = request.cookies.get("token")
    user_token = validateFirebaseToken(id_token)
    if not user_token:
        return RedirectResponse('/')

    # get a transaction object and add the objects
    transaction = firestore_db.transaction()
    transaction.delete(firestore_db.collection('dummy-data').document('5'))
    transaction.delete(firestore_db.collection('dummy-data').document('6'))
    transaction.delete(firestore_db.collection('dummy-data').document('7'))
    transaction.delete(firestore_db.collection('dummy-data').document('8'))

    # commit the transaction to the firestore
    transaction.commit()

    # when finished redirect with a 302 to force a GET request back to /
    return RedirectResponse('/', status_code=status.HTTP_302_FOUND)
```

Transaction deletion operation. The only difference here and step 09 is the use of .delete() instead of .set(). Something like this would be used if you need to guarantee the deletion of all documents in the transaction

12) in main.html add the following code

```
<!DOCTYPE html>
<html>
    <head>
        <title>Firestore basics</title>
        <link type="text/css" href="{{ url_for('static', path='/styles.css') }}" rel="stylesheet"/>
        <script type="module" src="{{ url_for('static', path='/firebase-login.js') }}"></script>
    </head>
    <body>
        <div id="login-box" hidden="true">
            Email:<input type="email" name="" id="email"><br/>
            Password: <input type="password" name="" id="password"><br/>
            <button id="login">Login</button>
            <button id="sign-up">Sign Up</button>
        </div>
        <button id="sign-out" hidden="true">Sign out</button>

        <!-- if we have a logged in user then show the user email address from the user_token object that was passed in
            we will also show the user document that has a name and age -->
        {% if user_token %}
            <p>User email: {{ user_token.email }}</p>
            <p>error message: {{ error_message }}</p>

            {% for doc in dummy_data %}
                {{ loop.index0 }} {{ doc.get('name') }}<br/>
            {% endfor %}

            <form action="/batch-add" method="post">
                Batch add four objects to the firestore<input type="submit"/>
            </form>

            <form action="/transaction-add" method="post">
                Transaction add four objects to the firestore<input type="submit"/>
            </form>

            <form action="/batch-delete" method="post">
                Batch delete four objects from the firestore<input type="submit"/>
            </form>

            <form action="/transaction-delete" method="post">
                Transaction delete four objects from the firestore<input type="submit"/>
            </form>
        {% endif %}
    </body>
</html>
```

What this template will show aside from the login box is a display of all the dummy data that is currently present in firestore. Four buttons will then be added to the page allowing you to trigger the add and delete functions in both batch and transaction form.

13) run the application and your add and delete functionality should work.

Example 09: Queries in Firestore

This is perhaps you will see the biggest difference between SQL and NoSQL type databases. If you are used to SQL you are probably used to having lots of expressive power, such as the ability to query on multiple fields at once, join tables together and link multiple queries together. However, SQL in its expressiveness is not the most scalable when it comes to cloud scale applications. Remember the following:

- SQL was designed at a time when storage was more expensive than compute. Thus the philosophy was storage is more expensive than compute
- NoSQL was designed at a time when compute is more expensive than storage. Thus the philosophy is compute is more expensive than storage.

Thus NoSQL is built for compute performance rather than minimising the storage of data by eliminating redundancies & favouring normalisation (as SQL and Relational Database design principles would suggest). In NoSQL it is (pardon the choice in words) normal to denormalise your database. Redundancy is permitted if the increase in performance is worth more than the effort to ensure consistency. This principle is also taken to queries, as to ensure the speed of the database queries are much more limited in scope again to ensure performance. Some restrictions you should be aware of:

- By default you can only query on a single attribute of a collection. You cannot query on multiple attributes unless you create an index for this in Firestore. If you wish to query on multiple attributes you must create an index for each combination of attributes you wish to query on.
- Index creation requires a significant amount of time particularly if you have a large number of documents in a collection. The size of the index is roughly equivalent to the number of documents in the collection. The index will be updated every time a document is added, deleted, or updated in that collection. However, query speed using that index is much faster.
- By default every single attribute of a document will have an index created for it by Firestore. Thus you should consider using multiple queries on single attributes to effect a multi attribute query and use set operations to find the intersection of the multiple queries to find the documents that satisfy both.

In this example we will show using our dummy data from the previous example how some basic queries are performed. We will also show the steps needed to create a multi attribute query.

NOTE: like the previous example clear out your firestore before running this application otherwise you will get unexpected errors when you try to run it.

01) Create a new directory Example 09 in your examples directory.

02) in your directory create the following structure

- static/
  - firebase-login.js
  - styles.css

- templates/
  - main.html
- main.py

03) use firebase-login.js and styles.css from the previous example

04) add in the following imports and setup code to main.py. This is the same as previous examples

```python
from fastapi import FastAPI, Request
from fastapi.responses import HTMLResponse, RedirectResponse
from fastapi.staticfiles import StaticFiles
from fastapi.templating import Jinja2Templates
import google.oauth2.id_token;
from google.auth.transport import requests
from google.cloud import firestore
from google.cloud.firestore_v1.base_query import FieldFilter
import starlette.status as status

# define the app that will contain all of our routing for Fast API
app = FastAPI()

# define a firestore client so we can interact with our database
firestore_db = firestore.Client()

# we need a request object to be able to talk to firebase for verifying user logins
firebase_request_adapter = requests.Request()

# define the static and templates directories
app.mount('/static', StaticFiles(directory='static'), name='static')
templates = Jinja2Templates(directory="templates")
```

05) add in the following getUser() function to main.py. This is the same as the previous example

```python
# function that we will use to retrieve and return the document that represents this user
# by using the ID of the firebase credentials. this function assumes that the credentials have
# been checked first
def getUser(user_token):
    # now that we have a user token we are going to try and retrieve a user object for this user from firestore if there
    # is not a user object for this user we will create one
    user = firestore_db.collection('users').document(user_token['user_id'])
    if not user.get().exists:
        user_data = {
            # for now we will use a placeholder name as this is not our focus but we will start with an empty array for our addresses
            'name': 'John Doe'
        }
        firestore_db.collection('users').document(user_token['user_id']).set(user_data)

    # return the user document
    return user
```

06) add in the following validateFirebaseToken() function to main.py. This is the same as the previous example

```python
# function that we will use to validate an id_token. will return the user_token if valid, None if not
def validateFirebaseToken(id_token):
    # if we dont have a token then return None
    if not id_token:
        return None

    # try to validate the token if this fails with an exception then this will remain as None so just return at the end
    # if we get an exception then log the exception before returning
    user_token = None
    try:
        user_token = google.oauth2.id_token.verify_firebase_token(id_token, firebase_request_adapter)
    except ValueError as err:
        # dump this message to console as it will not be displayed on the template. use for debugging but if you are building for
        # production you should handle this much more gracefully.
        print(str(err))

    # return the token to the caller
    return user_token
```

07) add in the following root() function to main.py. This is the same as the previous example.

```python
@app.get("/", response_class=HTMLResponse)
async def root(request: Request):
    # query firebase for the request token. We also declare a bunch of other variables here as we will need them
    # for rendering the template at the end. we have an error_message there in case you want to output an error to
    # the user in the template.
    id_token = request.cookies.get("token")
    error_message = "No error here"
    user_token = None
    user = None

    # check if we have a valid firebase login if not return the template with empty data as we will show the login box
    user_token = validateFirebaseToken(id_token)
    if not user_token:
        return templates.TemplateResponse('main.html', {'request': request, 'user_token': None, 'error_message': None, 'user_info': None})

    # get the user document and render the template we will need to pull the address objects as well
    # you can use get_all as well however it will not guarantee order. If order does not matter then use get_all()
    user = getUser(user_token).get()
    dummy_data = firestore_db.collection('dummy-data').stream()
    return templates.TemplateResponse('main.html', {'request': request, 'user_token': user_token, 'error_message': error_message, 'user_info': user, 'dummy_data': dummy_data})
```

08) add in the following initialise() function to main.py

```python
# route that will add four objects to the firestore by using a batch request. the idea is to add them in a single operation
# rather than four individual objects
@app.post("/initialise", response_class=RedirectResponse)
async def initialise(request: Request):
    # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
    id_token = request.cookies.get("token")
    user_token = validateFirebaseToken(id_token)
    if not user_token:
        return RedirectResponse('/')

    # create dictionaries for four documents to add into the firestore
    batch1 = {'number': 1, 'name': 'first'}
    batch2 = {'number': 2, 'name': 'second'}
    batch3 = {'number': 3, 'name': 'third'}
    batch4 = {'number': 4, 'name': 'fourth'}
    batch5 = {'number': 5, 'name': 'first'}
    batch6 = {'number': 6, 'name': 'second'}
    batch7 = {'number': 7, 'name': 'third'}
    batch8 = {'number': 8, 'name': 'fourth'}
    batch9 = {'number': 9, 'name': 'first'}
    batch10 = {'number': 10, 'name': 'second'}
    batch11 = {'number': 11, 'name': 'third'}
    batch12 = {'number': 12, 'name': 'fourth'}

    # get a batch object and add the objects
    batch = firestore_db.batch()
    batch.set(firestore_db.collection('dummy-data').document('1'), batch1)
    batch.set(firestore_db.collection('dummy-data').document('2'), batch2)
    batch.set(firestore_db.collection('dummy-data').document('3'), batch3)
    batch.set(firestore_db.collection('dummy-data').document('4'), batch4)
    batch.set(firestore_db.collection('dummy-data').document('5'), batch5)
    batch.set(firestore_db.collection('dummy-data').document('6'), batch6)
    batch.set(firestore_db.collection('dummy-data').document('7'), batch7)
    batch.set(firestore_db.collection('dummy-data').document('8'), batch8)
    batch.set(firestore_db.collection('dummy-data').document('9'), batch9)
    batch.set(firestore_db.collection('dummy-data').document('10'), batch10)
    batch.set(firestore_db.collection('dummy-data').document('11'), batch11)
    batch.set(firestore_db.collection('dummy-data').document('12'), batch12)

    # commit the batch and it should be added to the firestore
    batch.commit()

    # when finished redirect with a 302 to force a GET request back to /
    return RedirectResponse('/', status_code=status.HTTP_302_FOUND)
```

This function is purely to initialise a set of test data for the queries. We create 12 documents here using a batch operation similar to the previous example.

09) add in the following filterByNumber() function to main.py

```
127  # route that will filter by a number and return display the list of objects that satisfy
128  @app.post("/filter-by-number", response_class=HTMLResponse)
129  async def filterByNumber(request: Request):
130      # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
131      id_token = request.cookies.get("token")
132      user_token = validateFirebaseToken(id_token)
133      if not user_token:
134          return RedirectResponse('/')
135
136      # pull the number from the form
137      form = await request.form()
138      num = int(form['num'])
139
140      # get a referencce to the collection and then make a query
141      dummy_data_ref = firestore_db.collection('dummy-data')
142      query = dummy_data_ref.where(filter=FieldFilter('number', '>=', int(num)))
143
144      # return the template with the filtered data
145      user = getUser(user_token).get()
146      return templates.TemplateResponse('main.html', {'request': request, 'user_token': user_token, 'error_message': 'no error here', 'user_info': user, 'dummy_data': query.stream()})
```

This is an example of the most basic kind of query you can perform in Firestore. Like most of the functions we have previously seen we start by validating the user token before pulling the required data from the form.

- Lines 141 and 142 is where the query is being performed. The first step (141) in any query is that you must get a reference to the collection you wish to perform a query on. In this case “dummy-data”
- Using that reference you call the .where() function and provide a FieldFilter that takes three parameters
    - The first is the field you wish to apply the filter
    - The second is the filter you wish to apply. In this case greater or equals
    - The third is the value to use for comparison. In this case whatever value the user has entered in the form.
- Like document collections you can use either .get() or .stream() to return the results. Get() will return all matching documents at once whereas stream() will return them to you one by one. Your use case will determine which is most appropriate.

10) add in the following filterByRange() function to main.py

```
148  # route that will filter by two numbers and return display the list of objects that satisfy
149  @app.post("/filter-by-range", response_class=HTMLResponse)
150  async def filterByRange(request: Request):
151      # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
152      id_token = request.cookies.get("token")
153      user_token = validateFirebaseToken(id_token)
154      if not user_token:
155          return RedirectResponse('/')
156
157      # pull the numbers from the form
158      form = await request.form()
159      low = int(form['low'])
160      high = int(form['high'])
161
162      # get a referencce to the collection and then make a query
163      dummy_data_ref = firestore_db.collection('dummy-data')
164      query = dummy_data_ref.where(filter=FieldFilter('number', '>=', int(low))).where(filter=FieldFilter('number', '<=', int(high)))
165
166      # return the template with the filtered data
167      user = getUser(user_token).get()
168      return templates.TemplateResponse('main.html', {'request': request, 'user_token': user_token, 'error_message': 'no error here', 'user_info': user, 'dummy_data': query.stream()})
```

Similar to the previous step except this time we are providing two comparisons on the same attribute. This is permitted by Firestore as the same single attribute index will be used for both filters. To setup the combined filter we have two where clauses:

- The first where() filter compares on the number field using greater or equals to the value of low
- The second where() filter compares on the number field using greater or equals on the value of high.
- Thus any number that falls in the range [low, high] will be included in the query and will be returned in the call to .get() or .stream()

11) add in the following filterByString() function to main.py

```
# route that will filter by two strings and return display the list of objects that satisfy
# this will pick out all the objects with a name starting with the letter f
@app.post("/filter-by-string", response_class=HTMLResponse)
async def filterByString(request: Request):
    # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
    id_token = request.cookies.get("token")
    user_token = validateFirebaseToken(id_token)
    if not user_token:
        return RedirectResponse('/')

    # get a referencce to the collection and then make a query
    dummy_data_ref = firestore_db.collection('dummy-data')
    query = dummy_data_ref.where(filter=FieldFilter('name', '>=', 'f')).where(filter=FieldFilter('name', '<', 'g'))

    # return the template with the filtered data
    user = getUser(user_token).get()
    return templates.TemplateResponse('main.html', {'request': request, 'user_token': user_token, 'error_message': 'no error here', 'user_info': user, 'dummy_data': query.stream()})
```

In this example we are comparing on strings. Note that in order to restrict the range of strings returned we have to use a combination of a greater equals and a less than FieldFilter. In this particular instance we want to return all documents where name starts with the letter f. Note that for the second FieldFilter we have set this less than g to exclude all strings starting with a g or other letter later in the alphabet.

12) add in the following filterByBoth() function to main.py

```
# route that will filter by a name and number and return display the list of objects that satisfy
# note this will require an index to be built
@app.post("/filter-by-both", response_class=HTMLResponse)
async def filterByBoth(request: Request):
    # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
    id_token = request.cookies.get("token")
    user_token = validateFirebaseToken(id_token)
    if not user_token:
        return RedirectResponse('/')

    # pull the numbers from the form
    form = await request.form()
    num = int(form['num'])
    textinput = form['textinput']

    # get a referencce to the collection and then make a query
    dummy_data_ref = firestore_db.collection('dummy-data')
    query = dummy_data_ref.where(filter=FieldFilter('number', '>=', int(num))).where(filter=FieldFilter('name', '==', textinput))

    # return the template with the filtered data
    user = getUser(user_token).get()
    return templates.TemplateResponse('main.html', {'request': request, 'user_token': user_token, 'error_message': 'no error here', 'user_info': user, 'dummy_data': query.stream()})
```

In this example we are comparing on both the number and the field. NOTE that if you try to run this without first creating the index in Firestore it will fail with a message indicating that the required index has not been created. To setup the index you will need to do the following:

- go to http://console.cloud.google.com and select the Firestore section in the hamburger menu on the top left
- open the database you created. There should be a list of options on the left. One of them should read “Indexes”, click that.
- Make sure the “Composite” tab is selected and create an index with the first field as name in ascending order and the second as number in ascending order.

13) add in the following code to main.html

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Firestore basics</title>
        <link type="text/css" href="{{ url_for('static', path='/styles.css') }}" rel="stylesheet"/>
        <script type="module" src="{{ url_for('static', path='/firebase-login.js') }}"></script>
    </head>
    <body>
        <div id="login-box" hidden="true">
            Email:<input type="email" name="" id="email"><br/>
            Password: <input type="password" name="" id="password"><br/>
            <button id="login">Login</button>
            <button id="sign-up">Sign Up</button>
        </div>
        <button id="sign-out" hidden="true">Sign out</button>

        <!-- if we have a logged in user then show the user email address from the user_token object that was passed in
            we will also show the user document that has a name and age -->
        {% if user_token %}
            <p>User email: {{ user_token.email }}</p>
            <p>error message: {{ error_message }}</p>

            {% for doc in dummy_data %}
                {{ doc.get('number') }} {{ doc.get('name') }}<br/>
            {% endfor %}

            <form action="/initialise" method="post">
                Add our initialised objects to firestore<input type="submit"/>
            </form>

            <form action="/filter-by-number" method="post">
                Find objects with number higher than
                <input type="number" name="num" value="0"/>
                <input type="submit"/>
            </form>

            <form action="/filter-by-range" method="post">
                Find objects with number between
                <input type="number" name="low" value="0"/>
                <input type="number" name="high" value="0"/>
                <input type="submit"/>
            </form>

            <form action="/filter-by-string" method="post">
                Find all objects that have a name starting with the letter f
                <input type="submit"/>
            </form>

            <form action="/filter-by-both" method="post">
                Find all objects with number greater than and name equal to
                <input type="number" name="num" value="0"/>
                <input type="text" name="textinput" value="text here"/>
                <input type="submit"/>
            </form>
        {% endif %}
    </body>
</html>
```

Nothing really new here. We have four forms that allow you to run the various queries and a button for initialising your firestore.

14) run the application and all functionality should be working.

Example 10: Cloud Storage Buckets

When it comes to data storage we've only dealt with a database upto this point. However, what if you want to store more substantial content in the form of files. This is where cloud storage buckets will be used. It gives you a place to store files and organise them into directories. In this example we will not only show how to create, download, and delete a file. We will also show how to create directories as well.

NOTE: like the previous example clear out your firestore before running this application otherwise you will get unexpected errors when you try to run it.

01) Create a new directory Example 09 in your examples directory.

02) in your directory create the following structure

- static/
  - firebase-login.js
  - styles.css
- templates/
  - main.html
- main.py
- local_constants.py

03) use firebase-login.js and styles.css from the previous example

04) in local_constants.py add the following code

```
1  # local constants for our project name and bucket name
2  PROJECT_NAME= '[redacted]'
3  PROJECT_STORAGE_BUCKET='[redacted]'
```

PROJECT_NAME and PROJECT_STORAGE_BUCKET are constants that will be used in main.py to access our bucket.

- For PROJECT_NAME go to the JSON file you downloaded for the service account and take a copy of the value listed for the attribute "project_id"
- For PROJECT_STORAGE_BUCKET take the same attribute of "project_id" and add the following to then end of it
  - .appspot.com

05) in main.py add the following imports and setup code.

```
1  from fastapi import FastAPI, Request
2  from fastapi.responses import HTMLResponse, RedirectResponse, Response
3  from fastapi.staticfiles import StaticFiles
4  from fastapi.templating import Jinja2Templates
5  import google.oauth2.id_token
6  from google.auth.transport import requests
7  from google.cloud import firestore, storage
8  import starlette.status as status
9  import local_constants
10
11
12 # define the app that will contain all of our routing for Fast API
13 app = FastAPI()
14
15 # define a firestore client so we can interact with our database
16 firestore_db = firestore.Client()
17
18 # we need a request object to be able to talk to firebase for verifying user logins
19 firebase_request_adapter = requests.Request()
20
21 # define the static and templates directories
22 app.mount('/static', StaticFiles(directory='static'), name='static')
23 templates = Jinja2Templates(directory="templates")
```

The only differences here is on line 7 where we are importing the library needed to access the storage bucket. And on line 9 where we are importing the local_constants we wrote in the previous step.

06) in main.py add the following addDirectory() function

```
25 # function that will add an empty directory to our storage bucket. Note that the passed in directory name must have a trailing
26 # slash attached to it otherwise this will store as a file
27 def addDirectory(directory_name):
28     # get access to a storage client then list the bucket we need to use using the project and bucket name from the local constants
29     storage_client = storage.Client(project=local_constants.PROJECT_NAME)
30     bucket = storage_client.bucket(local_constants.PROJECT_STORAGE_BUCKET)
31
32     # make an empty blob out the directory name and upload it to the bucket. this is the convention GCS uses to distinguish between
33     # files and directories
34     blob = bucket.blob(directory_name)
35     blob.upload_from_string('', content_type='application/x-www-form-urlencoded;charset=UTF-8')
```

The first of our functions that interact directly with the cloud storage bucket. The first two lines on line 29 and 30 will be seen in other functions as well.

- Line 29 is where the get a storage client object. We request this from the storage library and we must provide the project name to the cloud project that contains our storage bucket
- Line 30 is where we provide the full name of the storage bucket that holds our files. Assuming you completed the steps to create an App Engine project much earlier this will have already been created for you.
- Line 34 is where we create a blob (Binary Large OBject) containing the directory name passed to this function
- Line 35 is where we direct the storage library to create our directory. Note the unusual structure here. For some very odd reason the storage library doesn't have a direct function to create a directory. Instead you are required to upload the empty string with the given content

type. The storage bucket on the other end will recognise this particular combination of string and content_type and will create a the same name as the provided blob.

07) in main.py add the following addFile() function

```
37  # function that will add a file to our storage bucket.
38  def addFile(file):
39      # get access to a storage client then list the bucket we need to use using the project and bucket name from the local constants
40      storage_client = storage.Client(project=local_constants.PROJECT_NAME)
41      bucket = storage_client.bucket(local_constants.PROJECT_STORAGE_BUCKET)
42
43      # create the blob to be stored then upload the content from the source file
44      #blob = bucket.blob(file)
45      print(file)
46      blob = storage.Blob(file.filename, bucket)
47      blob.upload_from_file(file.file)
```

Similar to the function seen in the previous step but this time we are uploading a file to the storage bucket. The only difference is on lines 46 and 47

- Line 46 we create a blob that references the file that the user selected on disk and the bucket where the file will be uploaded to.
- Line 47 is where we request the library to upload the content of the file on disk to the storage bucket.

08) in main.py add the following blobList() function

```
49  # function that will return the list of blobs in the bucket
50  def blobList(prefix):
51      # get access to a storage client then list the bucket we need to use using the project and bucket name from the local constants
52      storage_client = storage.Client(project=local_constants.PROJECT_NAME)
53
54      # get the list of blobs and return it
55      return storage_client.list_blobs(local_constants.PROJECT_STORAGE_BUCKET, prefix=prefix)
```

This function is used to return the list of blobs in a given directory in the cloud storage bucket. The list of blobs will include both directories and files. Note that the format is slightly different in that we don't have to create the bucket in order to list the blobs.

- This is because in line 55 we have to provide the bucket name to the .list_blobs() function
- Also not the second parameter here of prefix. Prefix states what path you want to list the blobs of. For example assume you had the following directory structure
    - /
        - test/
            - file1.txt
        - blarg/
        - file2.txt
- If you used a prefix of / and listed the blobs you would get: test/, blarg/, and file2.txt
- If you used a prefix of /test/ and listed the blobs you would get: file1.txt

09) in main.py add the following downloadBlob() function

```python
# function that will get the contents of a blob and will return it to the caller for downloading
def downloadBlob(filename):
    # get access to a storage client then list the bucket we need to use using the project and bucket name from the local constants
    storage_client = storage.Client(project=local_constants.PROJECT_NAME)
    bucket = storage_client.bucket(local_constants.PROJECT_STORAGE_BUCKET)

    # get access to the blobname and then download it to disk
    blob = bucket.get_blob(filename)
    return blob.download_as_bytes()
```

Like other functions you have to get a reference to a storage client and a bucket before you can download the blob once you get a reference to the blob we call .download_as_bytes() to get the content from the bucket.

10) in main.py add the following getUser() function which is the same as the previous example.

```python
# function that we will use to retrieve and return the document that represents this user
# by using the ID of the firebase credentials. this function assumes that the credentials have
# been checked first
def getUser(user_token):
    # now that we have a user token we are going to try and retrieve a user object for this user from firestore if there
    # is not a user object for this user we will create one
    user = firestore_db.collection('users').document(user_token['user_id'])
    if not user.get().exists:
        user_data = {
            # for now we will use a placeholder name as this is not our focus but we will start with an empty array for our addresses
            'name': 'John Doe'
        }
        firestore_db.collection('users').document(user_token['user_id']).set(user_data)

    # return the user document
    return user
```

11) in main.py add the following validateFirebaseToken() function which is the same as the previous example.

```python
# function that we will use to validate an id_token. will return the user_token if valid, None if not
def validateFirebaseToken(id_token):
    # if we dont have a token then return None
    if not id_token:
        return None

    # try to validate the token if this fails with an exception then this will remain as None so just return at the end
    # if we get an exception then log the exception before returning
    user_token = None
    try:
        user_token = google.oauth2.id_token.verify_firebase_token(id_token, firebase_request_adapter)
    except ValueError as err:
        # dump this message to console as it will not be displayed on the template. use for debugging but if you are building for
        # production you should handle this much more gracefully.
        print(str(err))

    # return the token to the caller
    return user_token
```

12) in main.py add the following root() function

```python
@app.get("/", response_class=HTMLResponse)
async def root(request: Request):
    # query firebase for the request token. We also declare a bunch of other variables here as we will need them
    # for rendering the template at the end. we have an error_message there in case you want to output an error to
    # the user in the template.
    id_token = request.cookies.get("token")
    error_message = "No error here"
    user_token = None
    user = None

    # check if we have a valid firebase login if not return the template with empty data as we will show the login box
    user_token = validateFirebaseToken(id_token)
    if not user_token:
        return templates.TemplateResponse('main.html', {'request': request, 'user_token': None, 'error_message': None, 'user_info': None})

    # the list of files and directories that we have in storage
    file_list = []
    directory_list = []

    # get the list of blobs and sort them based on directory and files
    blobs = blobList(None)
    for blob in blobs:
        if blob.name[-1] == '/':
            directory_list.append(blob)
        else:
            file_list.append(blob)

    # get the user document and render the template we will need to pull the address objects as well
    # you can use get_all as well however it will not guarantee order. If order does not matter then use get_all()
    user = getUser(user_token).get()
    return templates.TemplateResponse('main.html', {'request': request, 'user_token': user_token, 'error_message': error_message, 'user_info': user, 'file_list': file_list, 'directory_list': directory_list})
```

The first half of the function upto line 116 is similar to what you have seen in previous examples.

- Lines 119 and 120 we declare two empty lists. One for the blobs that are files and one for the blobs that are directories.
- Line 123 calls the .blobList() function we defined in an earlier step with a prefix of None. Providing a prefix of None means we want to list the root directory of the bucket (/)
- Lines 124 to 128 iterate through all of the blob names to determine which is a file and which is a directory. If the blob name ends in a "/" then it is a directory. If it ends in anything else it is a file.
- Finally we pull the user document from storage and pass that to the template along with the list of blobs that are files, and the list of blobs that are directories

13) in main.py add the following addDirectoryHandler() function

```
135 # handler that will take in a string representing a directory and will create it in the bucket
136 @app.post("/add-directory", response_class=RedirectResponse)
137 async def addDirectoryHandler(request: Request):
138     # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
139     id_token = request.cookies.get("token")
140     user_token = validateFirebaseToken(id_token)
141     if not user_token:
142         return RedirectResponse('/')
143 
144     # do a couple of basic checks. if the string is zero length or does not end in a / then reject it
145     form = await request.form()
146     dir_name = form['dir_name']
147     if dir_name == '' or dir_name[-1] != '/':
148         return RedirectResponse('/')
149 
150     # create the directory in the bucket and then redirect
151     addDirectory(dir_name)
152     return RedirectResponse('/', status_code=status.HTTP_302_FOUND)
```

Like the previous examples the first part of this upto line 146 is validating the user and pulling the required data from the form. Line 147 then does two checks. It makes sure that the directory name is not empty (i.e. cannot create a directory with no name) and finally does the last character end in "/". If either condition is triggered the directory will not be created. Thus when you run the example later make sure to add a "/" at the end of the name or it will be rejected.

14) in main.py add the following downloadFileHandler() function

```
154 # handler that will take in a filename to download and will serve it to the user
155 @app.post("/download-file", response_class=Response)
156 async def downloadFileHandler(request: Request):
157     # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
158     id_token = request.cookies.get("token")
159     user_token = validateFirebaseToken(id_token)
160     if not user_token:
161         return RedirectResponse('/')
162 
163     # pull the form data and see what filename we have for download
164     form = await request.form()
165     filename = form['filename']
166     return Response(downloadBlob(filename))
```

Like the previous examples everything upto line 165 is validating the user and pull the required data from the form. Line 166 will then call the downloadBlob() function we defined earlier and wrap

that in a response object. When the client browser receives the response it will then attempt to download the file to the user's machine.

15) in main.py add the following uploadFileHandler() function

```python
# handler that will upload a file to the bucket. this will store it in root of the bucket
@app.post("/upload-file", response_class=RedirectResponse)
async def uploadFileHandler(request: Request):
    # there should be a token. Validate it and if invalid then redirect back to / as a basic security measure
    id_token = request.cookies.get("token")
    user_token = validateFirebaseToken(id_token)
    if not user_token:
        return RedirectResponse('/')

    # if the filename is empty then redirect back to / and do nothing
    form = await request.form()
    if form['file_name'].filename == '':
        return RedirectResponse('/', status_code=status.HTTP_302_FOUND)

    # redirect back after the directory is added
    addFile(form['file_name'])
    return RedirectResponse('/', status_code=status.HTTP_302_FOUND)
```

Similar code upto Line 180 but then on line 183 we upload the file to the storage bucket before redirecting back to / in order to show the updated file in the UI.

16) in main.html add the following code

```html
<!DOCTYPE html>
<html>
    <head>
        <title>Firestore basics</title>
        <link type="text/css" href="{{ url_for('static', path='/styles.css') }}" rel="stylesheet"/>
        <script type="module" src="{{ url_for('static', path='/firebase-login.js') }}"></script>
    </head>
    <body>
        <div id="login-box" hidden="true">
            Email:<input type="email" name="" id="email"><br/>
            Password: <input type="password" name="" id="password"><br/>
            <button id="login">Login</button>
            <button id="sign-up">Sign Up</button>
        </div>
        <button id="sign-out" hidden="true">Sign out</button>

        <!-- if we have a logged in user then show the user email address from the user_token object that was passed in
            we will also show the user document that has a name and age -->
        {% if user_token %}
            <p>User email: {{ user_token.email }}</p>
            <p>error message: {{ error_message }}</p>

            <form action="/add-directory" method="post">
                Add a directory to the bucket
                <input type="text" name="dir_name"/>
                <input type="submit" value="Add"/>
            </form>

            <form action="/upload-file" method="post" enctype="multipart/form-data">
                Upload File: <input type="file" name="file_name" />
                <input type="submit"/>
            </form>

            <h2>Directories in Bucket</h2><br/>
            {% for dir in directory_list %}
                {{ dir.name }}<br/>
            {% endfor %}

            <h2>Files in Bucket</h2><br/>
            {% for file in file_list %}
                <form action="/download-file" method="post">
                    <input type="hidden" value="{{ file.name }}" name="filename"/>
                    {{ file.name }}<input type="submit" value="Download"/><br/>
                </form>
            {% endfor %}
        {% endif %}
    </body>
</html>
```

The first two forms added here to add a directory and upload a file to the cloud storage bucket. After this the next set of code on lines 35 to 37 will iterate through all of the directory blobs and display their names. Lines 40 to 45 will then iterate through the file blobs and not only display them but will add a form with a button for downloading and a hidden attribute containing the name of the file so the handler knows which file is requested.

17) run the application and you should be able to upload and download files and create directories.