Google App Engine 3 Python by example

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 rather than copy paste out the programs within they should be typed out by hand. The reason for this is to enable 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 understanding 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 then proceed to work with 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. The recommendation here is to use the Github editor Atom which is highly customisable and has many extensions available.
- An installation of python 3.8 that is accessible from anywhere on the command line. i.e. your PATH variable has been modified to find it
- An installation of google app engine with the app-engine-python and app-engine-python-extras installed that is also accessible from anywhere on the command line. i.e. your PATH variable has been modified to find it

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

Changes since python 2.7

The python 2.7 version of App Engine now no longer exists. For those of you who came from such an environment there will be many adjustments that will be seen here. No longer are you restricted to webapp2 there is the flexibility to use many Python based web frameworks. The main one used here is Flask but Django is also an option. Also the google login/logout service has now been removed and in its place is the use of Firebase Authentication using OAuth2 as the authentication mechanism. The datastore has also been replaced in favour of using Firebase as it is an established, and robust NoSQL database.

With all of the above sorted its time to dive into our first example which is the basic hello world application just to get off the ground and started developing

Example 01: Hello World in Google App Engine Python 3

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 local python environment. The instructions you see here are taken from:

https://cloud.google.com/appengine/docs/standard/python3/quickstart

Specifically the Linux/OSX version. There are Windows versions of these as well on the same page. First open a command line and navigate to the directory where you will write the code for these examples. When you get there run the following command:

```
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. After this we will need to run the following command

```
source env/bin/activate
```

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 at 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 above two commands everytime you start a terminal to setup the virtual environment before you can start running app engine applications. The only way to get out of the virtual environment is to exit the terminal when you are finished. Now we can get to developing the application by following the given steps

01) create a new directory for this example

02) inside that directory create a file called requirements.txt and add the following text into it

```
Flask==1.1.2
```

The **requirements.txt** file is there to tell Google App Engine what additional libraries are needed in order to run your application. In this case we are stating that we need the Flask web framework and specifically version 1.1.2 in order to run our application. If we are developing the application locally we will need to install the requirements by hand which will be covered in a later step. Any additional libraries you will need must be specified in this file.

Note that it is not possible to add any Python library of your choosing to this as Google App Engine runs a restricted version of Python3. Google App Engine may also have a deny-list of libraries that are not permitted to be installed as well and your chosen library(ies) may appear on that list.

03) Create a file called ‘app.yaml’ and place the following code into it

```
runtime: python38
```

The **app.yaml** file tells google app engine how to setup and initialise your application in preparation for running. For our first application this is very simple as we are stating that we will use the Python 3.8 runtime to run our application. In later examples we will have more detailed versions of this file as we will start to add additional details for more complex applications

04) create a file called ‘main.py’ and add the following code into it

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello world'

if __name__ == '__main__':
    app.run(host='127.0.0.1', port=8080, debug=True)
```

This is the basis of our hello world application. The first line is an import statement that will give us access to the Flask framework so we can build our application. The next line defines a Flask application object. This object will be needed to run our application as it contains all the necessary support to receive requests and to generate and send replies. The **__name__** part takes the name of the current file and passes it as part of the Flask constructor.

The line that follows this **@app.route(‘/’)** is a Flask annotation that specifies the following method should be called if the URL of the host is called without anything after the / the so called root URL. When you run this application locally and point your browser to http://localhost:8080/ this annotation will be triggered and the **hello()** function below it will be called to serve a response.

The **hello()** function is very simple in nature. It will return the string “hello world” to the browser. Note that whatever the value of the return statement is in a function tagged by “@app.route” that will be the response that the user will see.

Finally we have an if statement that says if this file has been mentioned as part of a call to python on the command line the code guarded by this if statement will be run. In this case the **app.run()** function here will run the application and will set the server at 127.0.0.1 (localhost) on port 8080 and debug functionality will be enabled. When you are developing applications debug should always be set to true so you can attach a debugger if needed and also debugging messages will be logged to console.

05) open a console and navigate to the project directory. Make sure you have created your environment and sourced it as shown above and run the following command

```
pip install -r requirements.txt
```

This will read in the requirements file and will install the necessary libraries

06) run your application by executing the following command

```
python main.py
```

This will start your Flask application and if you navigate to **localhost:8080** in your web browser you will see the message “Hello world” in plain text if everything is working correctly.

Example 02: A more complex app with user authentication and NoSQL storage. Version 1: setting up a file structure and some basic components

The second application will be more extensive than the first. It will introduce two core components that are needed in every application that is cloud facing: user authentication and data storage. Both components will use the external service Firebase. User authentication will use OAuth2 (a standard authentication method) while data storage will be handled by the NoSQL Firebase database. Note that we won't get to that yet but there will be multiple versions of this example that will build towards it

If you have not used a NoSQL database before and you are familiar with SQL databases there are a few things you will need to be aware of:

- Firebase's NoSQL database is key-value pair system not a relational table. It is designed for speed and to support cloud scale applications
- As a result of the point above you will not have queries on the same expressive power that SQL provides. This means you will have to think about how you structure your data as this will have an effect on the kind of queries you can perform.
- The advantage however though is if you can structure your data to use less queries and use more direct key access on average your application data access will be quicker than if you used an SQL database

We will expose a lot of these point in the development of this application when we get to the database but for now we will follow along with the following google documentation example

https://cloud.google.com/appengine/docs/standard/python3/building-app/writing-web-service

and we will provide detailed notes along the way.

01) create a new directory for this application and inside that directory create the following directory and file structure

- app.yaml
- main.py
- requirements.txt
- static/
  - script.js
  - style.css
- templates/
  - main.html

The first three files should be familiar to you from the previous example. However there are now two extra directories. The static directory is there for any static files that you will serve as part of your application. Generally we will store JavaScript, CSS, and any static HTML files here along with all other static assets. The templates directory will contain HTML templates that are used to generate HTML pages on the fly. These can be customised with data from your python code to alter the display of the template each time a request for that template is required. We will fill in code for these additional files in the next steps and will explain what is happening as we go along.

01) first we will define the text in **requirements.txt**. Like the previous application you should add in the following:

```
Flask==1.1.2
```

02) next we will define the code in **app.yaml** which will be the following

```
runtime: python38
handlers:
- url : /static
  static_dir : static

- url : /.*
  script : auto
```

The first line is the same as the previous example but we have some additional information here in the form of handlers. Handlers specify what your application should do in certain scenarios. Here we have two handlers defined by **url**. This hander will state what will happen if certain URLs are triggered by either the user or the application. The first handler will state what will happen if the URL http://localhost:8080/static is triggered. The line below it states that anything that hits this URL should serve its content directly from the static content directory called **static**. We will use this to serve up JS and CSS files. Finally the second of the URL handlers states what happens to any other URL that is not served by the static handler above. The **script : auto** command will redirect Flask to find the python script with the appropriate **@app.route** annotation for the URL provided. Note that this last catch all handler is always required if you are going to serve static content of any description. If it is omitted then your application will not work.

03) next we will define **main.py**

```
import datetime
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/')
def root():
    dummy_times = [datetime.datetime(2018, 1, 1, 10, 0, 0),
                   datetime.datetime(2018, 1, 2, 10, 30, 0),
                   datetime.datetime(2018, 1, 3, 11, 0, 0),
                   ]

    return render_template('index.html', times=dummy_times)

if __name__ == '__main__':
    app.run(host='127.0.0.1', port=8080, debug=True)
```

At the beginning of the script we will import the **datetime** library which will be used later in the script. We also have our definition of the application object as before. Note that we have an additional import from Flask called **render_template**. What **render_template** is designed to do is to trigger HTML templates for dynamic rendering. What this permits us to do is pass python objects to a HTML template that will be dynamically rendered with whatever data is present in those python objects. The templates can be parameterised so you can serve up dynamic content on different requests.

In the **root()** function we have defined some sample data of the **datetime** class type. We will do this just to show how template rendering will work as we have yet to define a datastore of any description. Finally on the last line of this function we will call render_template to render the template called **index.html** (which it will look for in the **templates/** directory) and we will pass a times variable with the **dummy_times** we defined on the previous line. These times will be used to dynamically render the **index.html** template.

04) in **templates/index.html** add in the following HTML

```
<!doctype html>
<html>
<head>
  <title>Datastore and Firebase Auth Example</title>
  <script src="{{ url_for('static', filename='script.js') }}"></script>
  <link type="text/css" rel="stylesheet" href="{{ url_for('static',
filename='style.css') }}">
</head>
<body>

  <h1>Datastore and Firebase Auth Example</h1>

  <h2>Last 10 visits</h2>
  {% for time in times %}
    <p>{{ time }}</p>
  {% endfor %}

</body>
</html>
```

This is your first HTML template. Note that this is a Jinja template we will use as it is the templating engine favoured by Flask. Most of the HTML will be familiar to you so we will only focus on the templated parts. First we will start off with the template defined in the script and link tags. Note the form **`{{ url_for(‘static’, filename=’script.js’) }}`** and similar for the **style.css** file. This asks the Jinja templating engine to run the **url_for()** function that will generate a url out of the base url http://localhost:8080/ (or whatever URL your app is hosted on when running the app on GAE in the cloud) it will add **static/** to it and it will finally add the filename **script.js** to it. The resulting URL will be http://localhost:8080/static/script.js and this gets sent back to the Flask server. Flask will see the static part and will use the static handler defined in **app.yaml** to serve that content from the **static/** directory you defined as part of your application structure back in step 01). It will do the same for the **style.css** file as well.

The other templating part we will look at is the **`{% for time in times %}` … `{% endfor %}`** part. This asks Jinja to perform a standard python loop for everything defined between **for** and **endfor**. Here we are defining a loop variable **time** which will take a reference to each object in the times list that was pass as part of the **render_template** call earlier. It will then run that loop body for each object. In this case the loop body is pretty simple we have a paragraph tag with **`{{ time }}`** in between it. This asks the Jinja to call the **__str__()** function of the python object and display the string that it returns in this place. As we have defined three time objects in our times list this loop will iterate three times.

05) put the following code in script.js

```
'use strict';
window.addEventListener('load', function () {
  console.log("Hello World!");
});
```

06) put the following code in style.css

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

And run the application. It should display a message in the top centre of the view and then three dates and times underneath

Example 03: Adding to the previous example to include access to the cloud datastore.

In this example we will add access to the datastore as a place to store our data for our application. Here we will use the google cloud datastore which is a NoSQL Key-Value pair database to store all of our application data. Before carrying on with this example the first thing you will need to do is create an application instance on Google Cloud by following the details below. At the end of this you should have a JSON file that will contain all of your authentication details for the application instance you just created. This JSON file will be required to enable your application to access the database

https://cloud.google.com/docs/authentication/getU+0074ing-started

In this example we will introduce some of the most basic operations you will require as part of any Google App Engine application. The ability to write and retrieve data from a database. Here we will do something very simple. Everytime a visit is made to the root page we will take the current system time and will store it in the datastore. We will also retrieve the last 10 visit times from the datastore and will display them as part of the template.

01) take a copy of the complete code of the previous example into a new directory

02) add the following line into requirements.txt

```
google-cloud-datastore==1.15.0
```

This is the python library that will allow our application to interact with, store and retrieve data from the datastore.

03) in main.py add the following import to the list of imports

```
from google.cloud import datastore
```

This will bring in the module we will need to access in order to be able to interact with the datastore. Anytime we want to store or retrieve data we will need to use this **datastore** module

04) in main.py add the following code after the declaration of the **app** variable

```
datastore_client = datastore.Client()
```

This will create a datastore client through which we can make requests to do CRUD operations on the datastore

05) in main.py add the following function after the code in 04) but before the definition of **root()**

```
def store_time(dt):
    entity = datastore.Entity(key = datastore_client.key('visit'))

    entity.update({'timestamp' : dt})
    datastore_client.put(entity)
```

This method takes in a single parameter **dt** which will be a python **datetime** object. We will use the time defined in this object to store a timestamp of when the user last accessed the page. The first line of the method defines a datastore **Entity** object. Entity objects are used to store a collection of attributes together that belong to one entity. In this case we are defining an entity object of type **visit** that will be stored in our datastore. In the following line we are adding an attribute to the entity called **timestamp** that will have the value of **dt** that was passed into the function. Note that update will add attributes if they are not defined in the entity. If the attribute already exists in the entity it will overwrite the value for that attribute. Finally in the last line we are asking the datastore client to store the entity in the datastore.

06) in main.py add the following function after the code in 05) but before the definition of “root()”

```
def fetch_times(limit):
    query = datastore_client.query(kind='visit')
    query.order = ['-timestamp']

    times = query.fetch(limit=limit)
    return times
```

This function will retrieve the last **limit** number of **visit** entities from the datastore using a very basic query and will return that set of **visit** entities back to the caller in a regular python list object. The function starts by asking the datastore client to create an empty query on **visit** entities. This kind of query will return back all **visit** entities in the datastore. The line immediately after this states the order in which we want the visit entities to be returned. The **-timestamp** part states that we wish to first order by the timestamp attribute with the **-** indicating that we wish to order in decreasing order. The next line will then ask the query object to fetch our **visit** entities from the datastore to the limit defined by the **limit** integer that was passed into the function. Finally we return the list of **visit** entities from the query to the caller of the function.

07) replace the “root()” function to have the following code

```
def root():
    store_time(datetime.datetime.now())
    times = fetch_times(10)

    return render_template('index.html', times=times)
```

Note how we have gotten rid of the dummy times variable we can use the Datastore to get the list of times that we wish to display to the user. We start by first calling the **store_time()** function we defined in 05) above with the current system time. What this will do is store a new **visit** entity in the datastore with the current system time set as the timestamp. The next line will then call our **fetch_times()** function that we defined in 06) and it will fetch a maximum of the 10 most recent visit times. Finally those fetched times are then passed to the **render_template()** function for **index.html** and the times are passed under the variable **times**.

08) Before you go to run this project you will need the JSON file nearby to access the datastore. Before you run your application in your command line you will need to set the session variable GOOGLE_APPLICATION_CREDENTIALS with the location of this JSON file. In my case I have the JSON file above the directory this project runs in so in Linux I would run the following command to set that session variable

```
export GOOGLE_APPLICATION_CREDENTIALS=”../app-engine-3-testing.json”
```

where `app-engine-3-testing.json` is my JSON file. After this run the project as normal and every time you visit the root page now a new visit time will be added and the list will be dynamically updated.

Example 04: adding Firebase authentication to the basic template.

Being able to dynamically store and modify data is one requirement of PaaS applications. However, there will be a need to distinguish information between different users and to be able to segregate access and data between different users. In order to do this we will need to have access to an authentication mechanism for our application. For this we will use Firebase as the authentication mechanism as it supports a standard authentication method in OAuth2. The first thing you will need to do is add your application to the Firebase console and permit user access to it using the "Add Firebase to your Cloud Application" section of the following page:

https://cloud.google.com/appengine/docs/standard/python3/building-app/adding-firebase

Once you have this done follow the steps below. Once this is setup go to the project settings page and under the general tab you will find your API key and other parameters that you will need to link up with Firebase authentication. It will also provide sample javascript that has all of these parameters filled in so you can copy paste into your code later.

01) take a complete copy of the previous application and put it in a new directory

02) first we are going to modify templates/index.html and add the following code into the **<head>** tag

```
<script src="https://www.gstatic.com/firebasejs/ui/4.4.0/firebase-ui-auth.js"></script>
<link type="text/css" rel="stylesheet"
href="https://www.gstatic.com/firebasejs/ui/4.4.0/firebase-ui-auth.css" />
```

This will pull in the necessary JavaScript and CSS files to allow your user interface to map onto the Firebase authentication services. We will need to add some more to the template however to display these services and allow a user to use them.

03) add a new file to the static directory called **app-setup.js** and add in the following code

```
  var firebaseConfig = {
    apiKey: "<API_KEY>",
    authDomain: "<PROJECT_ID>.firebaseapp.com",
    databaseURL: "https://<DATABASE_NAME>.firebaseio.com",
    projectId: "<PROJECT_ID>",
    storageBucket: "<BUCKET>.appspot.com",
    messagingSenderId: "<SENDER_ID>",
  };
  firebase.initializeApp(firebaseConfig);
```

This is necessary for attaching firebase to your application. The template above can be autofilled for you by looking for the sample code in the General tab of the Settings page of the application you defined in the Firebase console earlier. This segment of code contains your API key and all the necessary links to enable your application to use Firebase to create and authenticate users. You will need this for every template you have that requires the use of firebase authentication however, we have refactored it out here such that we can include it in any template that requires it without having to copy paste the code each time

04) at the bottom of the <**body**> tag add in the following lines

```
<script src="https://www.gstatic.com/firebasejs/7.14.5/firebase-app.js"></script>
<script src="https://www.gstatic.com/firebasejs/7.8.0/firebase-auth.js"></script>
<script src="{{ url_for('static', filename='app-setup.js') }}"></script>
```

This pulls in additonal scripts that are needed to get firebase authentication up and running. The last script import here is to pull in the additonal script we defined in 03) above

05) replace the HTML/template part for the last 10 visits with the following HTML/template code

```
<div id="firebase-auth-container"></div>
  <button id="sign-out" hidden="true">Sign out</button>
  <div id="login-info" hidden="true">
      <h2>Login Info</h2>
      {% if user_data %}
        <dl>
            <dt>Name:</dt><dd>{{ user_data['name'] }}</dd>
            <dt>Email:</dt><dd>{{ user_data['email'] }}</dd>
            <dt>Last 10 Visits:</dt><dd>
                {% for time in times %}
                  <p>{{ time['timestamp'] }}</p>
                {% endfor %}
            </dd>
        </dl>
      {% elif error_message %}
        <p>Error Message: {{ error_message }}</p>
      {% endif %}
  </div>
```

There is a lot to process here but most will not be visible until the later steps. The **firebase-auth-container** will be used by the firebase scripts for showing login options in a later step. There is also a sign-out button that is currently hidden until we get the rest of the firebase authentication sorted. In the following templated code we have now given the template the ability to react to different sets of data being passed to the template. There is a **{% if user_data %}** that will display the first part of the template between here and **{% elif … %}** if a **user_data** object has been passed to the template. If no **user_data** has been provided but an error_message has been provided then the template under **{% error_message %}** will be displayed instead. Note that in the user_data template we are going to parameterise the template with the **user_data['name']** and **user_name['email']** parts. This will allow us to show the name and email of the user who is logged in.

06) in **script.js** clear out the code currently in the function attached to **load** and replace it with the following code

```
document.getElementById('sign-out').onclick = function() {
        // ask firebase to sign out the user
        firebase.auth().signOut();
};
```

This will attach a function to the onclick property of the **sign-out** button defined in our template earlier. If that button is clicked then it will ask firebase authentication to sign out the currently logged in user.

07) add the following code to the function attached to **load** under the code defined in 05) above

```
var uiConfig = {
        signInSuccessUrl: '/',
        signInOptions: [
            firebase.auth.GoogleAuthProvider.PROVIDER_ID,
            firebase.auth.EmailAuthProvider.PROVIDER_ID
        ]
};
```

This variable will be used to configure the firebase authentication widget in a later step. But there are some important parameters that can be explained now. **SignInSuccessUrl** will tell firebase where to redirect to if the sign in was successful. In this case we will redirect to http://locahost:8080/ which will be picked up by the **root()** function in main.py. Secondly the **signInOptions** variable contains a list of authentication methods that we will accept for login. Here we state we can use the Google login mechanism or standard email addresses as a login mechanism.

08) add the following code to the function attached to **load** under the code defined in 06) above.

```
firebase.auth().onAuthStateChanged(function(user) {
        if(user) {
            document.getElementById('sign-out').hidden = false;
            document.getElementById('login-info').hidden = false;

            console.log('Signed in as ${user.displayName} (${user.email})');

            user.getIdToken().then(function(token) {
                document.cookie = "token=" + token;
            });
        } else {
            var ui = new firebaseui.auth.AuthUI(firebase.auth());
            ui.start('#firebase-auth-container', uiConfig);

            document.getElementById('sign-out').hidden = true;
            document.getElementById('login-info').hidden = true;
            document.cookie = "token=";
        }
    }, function(error) {
        console.log(error);
        alert('Unable to log in: ' + error);
    });
```

There is a lot of code to discuss here. First however is that this code will only run if the **onAuthStateChanged()** function is called by firebase. This will only get called if a user has just logged in or logged out. The attached code is reacting to those events. In **function(user)** we have two branches. The first reacts to a user who just logged in and the second reacts to a user who just logged out.

In the first branch we start by making visible the **sign-out** button and the **login-info** divider. We then log a message to the console stating which user was logged in. Finally we get the token that represents the user and attach it to a cookie for this document. We will use this authentication token for two purposes: 1. to identify the currently logged in user. 2. to maintain session information for this user.

In the second branch we start by creating a new firebase authentication widget and set it with the parameters defined in 06) above. We then ask firebase to start that UI widget and display it to the user. In the final part we hide the sign-out button and the login-info divider before clearing the cookie that was previously set.

Finally we have an error handling function that will display an alert box if login does not work. The alert will show the error returned by firebase and that error will also be logged to console.

09) in main.py add the following imports

```
import google.oauth2.id_token
from flask import Flask, render_template, request
from google.auth.transport import requests
```

This will be needed to read and use the token that we attached as part of the cookie in the **index.html** template we recently modified. The requests module will enable us to link up with firebase authentication in our code. Be very careful as you have two similar imports here **request** from **flask** and **requests** from **google.auth.transport**. We can’t do much about the naming so you will need to be careful about the use of either in code.

10) add the following code after code for retrieving a datastore client

```
firebase_request_adapter = requests.Request()
```

This will be needed to interact with firebase directly so we can verify users at any stage in our application. It is recommeneded that at all times throughout your application requests should be checked to see what user it relates to at all times.

11) Add the following code into the **root()** function before the call to store time

```
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    times = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)
        except ValueError as exc:
            error_message = str(exc)
```

This code will first request the user token from the cookies in the **request** object that was sent with this request. It will also set some other variables to **None** as not all of these will be set at the same time. The following if statement will only be run if there was a token in the cookies. What the **try catch** block will do is contact the firebase authentication service and will verify that the token belongs to a user of that application. If this is successful it will return us identification of the user who has that token. Otherwise it will only throw back an error message.

12) change the call to render_template in the root() function with the following code

```
return render_template('index.html', user_data=claims, times=times,
error_message=error_message)
```

This will now render **index.html** with some additional parameters **user_data** and **error_message**. These variables correspond to the calls to **user_data** and **error_message** in **index.html**

13) add the following dependencies to **requirements.txt** install them and run the application

```
google-auth==1.21.2
requests==2.24.0
```

When you run the application you should be able to register and login to the application. If you are logged out you should only have an option to login. When you login you should see your user information (as put in on registration) along with the last 10 access times and a sign out button.

Example 05: tagging visit times to individual users by using connecting multiple entity types together.

In the previous example we got to the point where we could use firebase authentication to register and login users. We were also able to store data, however the data was not tied down to any specific user. In this example we will show how using multiple keys connected together we can connect multiple entity types together. Here will we connect a visit entity to a user entity such that we can track the visit times specifically for that user.

01) take a complete copy of the previous examples code

02) replace the code of the **store_time()** function with the following code

```
def store_time(email, dt):
    entity = datastore.Entity(key = datastore_client.key('User', email, 'visit'))

    entity.update({'timestamp' : dt})
    datastore_client.put(entity)
```

This will now tie down a visit entity to a user entity by using an ancestor key as defined in the call to **datastore_client.key()**. The call to this function states that the first part of the key will identify a User entity by an email address, while the second part of the key will be an autogenerated key that will be used to identify the visit object of that user. The rest of the function is pretty much the same bar there is an additonal email parameter provided as part of the function call.

03) replace the code of the **fetch_times()** function with the following code

```
def fetch_times(email, limit):
    ancestor_key =  datastore_client.key('User', email)

    query = datastore_client.query(kind='visit', ancestor=ancestor_key)
    query.order = ['-timestamp']

    times = query.fetch(limit=limit)
    return times
```

Similar to **store_times()** above we have an email parameter added to the function. But note how at the start we define an ancestor key for the User entity with an email address. We then modify the query to say that we only wish to pull back the visit objects that have our ancestor key set, i.e. it will only pull back the visit entities belonging to this user.

04) replace the code in **root()** with the following code

```
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    times = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            store_time(claims['email'], datetime.datetime.now())
            times = fetch_times(claims['email'], 10)
        except ValueError as exc:
            error_message = str(exc)

    return render_template('index.html', user_data=claims, times=times,
error_message=error_message)
```

Here we have moved the calls to **store_time()** and **fetch_times** into the try catch block the reason for this is that if we don't get claims for the user then this code would fail each time.

05) define a new file **index.yaml** and add this code to it

```
indexes:
- kind: visit
  ancestor: yes
  properties:
  - name: timestamp
    direction: desc
```

This file will define an index for the visit entities. Be aware that any entity that has an ancestor relationship you must define indexes for. In this case we are declaring that the visit entity has an ancestor connected to it. We also state that it should be indexed in descending order of timestamp. The reason we do this is that is the main query we will run on visit entities will be timestamps in descending order. This will vastly speed up those queries. You will need to set this on your cloud datastore before your application will function properly. When you run this application you will start to see visit times belonging to a single user.

Example 06: Properties supported by Datastore Entities

In this example we will explore the entities in the basic template examples in a lot more detail. We will show what properties and datatypes they support. When a user signs in to this simple example a User object will be created for them and it will be prepopulated with sample data. At a later stage we will also show how to update some values of an entity by showing how to handle a POST request and entity updates.

01) take a copy of the stripped out template and fill in the details for the firebase authentication script

02) in **main.py** add in the follow function

```
def retrieveUserInfo(claims):
    entity_key = datastore_client.key('UserInfo', claims['email'])
    entity = datastore_client.get(entity_key)

    return entity
```

This function will take in a user claims object similar to the previous examples. It will use the email address from this **claims** object as the identifier for a key for a **UserInfo** entity. This will tie down a **UserInfo** object to a specific user login. The following line will then attempt to retrieve the user info object from the datastore. One of two things will happen here:

- If the entity exists in the datastore then it will be retrieved and returned to you.
- If the entity does not exist in the datastore the default value of **None** will be returned.

Finally we return whatever was returned by the **get()** function to the caller

03) in **main.py** add in the following function

```
def createUserInfo(claims):
    entity_key = datastore_client.key('UserInfo', claims['email'])
    entity = datastore.Entity(key = entity_key)
    entity.update({
        'email': claims['email'],
        'name': claims['name'],
        'creation_date': datetime.datetime.now(),
        'bool_value': True,
        'float_value': 3.14,
        'int_value': 10,
        'string_value': 'this is a sample string',
        'list_value': [1, 2, 3, 4],
        'dictionary_value': {'A' : 1, 'B': 2, 'C': 3}
    })

    datastore_client.put(entity)
```

This function will first create a key for the **UserInfo** that is tied down to the currently logged in user’s email address. It will then create a **UserInfo** Entity using that key. The update method provides a dictionary of values to add to this **UserInfo** object. Here we show most of the datatypes supported by Entity objects. The first two are **strings** with the user email and name. The next is a **datetime** object that takes the current system time. Then we have a **boolean**, **float**, **integer**, another **string** value. These are the basic datatypes supported. The last two values are a python list and a python dictionary which can be composed of the basic datatypes or **keys** or **entities**. We will deal with the latter two in a later example.

04) replace the code of the root() function with the following code.

```
def root():
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    user_info = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token, 
firebase_request_adapter)

            user_info = retrieveUserInfo(claims)
            if user_info == None:
                createUserInfo(claims)
                user_info = retrieveUserInfo(claims)
        except ValueError as exc:
            error_message = str(exc)

    return render_template('index.html', user_data=claims, error_message=error_message, 
user_info=user_info)
```

In the first section we are declaring our variables as before but now we have removed the times variable and replaced it by **user_info**. Like before we have an if statement checking if there is a token attached to a cookie and will try to verify the claims of that cookie. However, this is where the major difference from the previous example begins. We first try to retrieve the the user information from the datastore using the function defined in 03) above. If the user has logged in before then that entity will be returned and the code will skip the two lines below the if. However if the user is logging in for the first time then we will create a **UserInfo** object store it in the datastore and retrieve it again for the purposes of the template. Note how for the template we are passing the **user_info** variable to the template instead of the times like the previous examples.

05) in **index.html** change the templated code between the login-info **<div>** tags to the following:

```
{% if user_data %}
  <dl>
      <dt>Name:</dt><dd>{{ user_data['name'] }}</dd>
      <dt>Email:</dt><dd>{{ user_data['email'] }}</dd>
      <dt>Creation Date:</dt><dd>{{ user_info['creation_date'] }}</dd>
      <dt>Boolean Value:</dt><dd>{{ user_info['bool_value'] }}</dd>
      <dt>Float Value:</dt><dd>{{ user_info['float_value'] }}</dd>
      <dt>Int Value:</dt><dd>{{ user_info['int_value'] }}</dd>
 <dt>String Value:</dt><dd>{{ user_info['string_value'] }}</dd>
      <dt>List Value:</dt><dd>{{ user_info['list_value'] }}</dd>
      <dt>Dictionary Value:</dt><dd>{{ user_info['dictionary_value'] }}</dd>
  </dl>
{% elif error_message %}
  <p>Error Message: {{ error_message }}</p>
{% endif %}
```

Like the previous example we will render two different templates depending on if a user is logged in or not. In the logged in section all we are doing here is outputting the values of the user object we created for the user as pulled back from the datastore. When you run this application at this point and login as a user you should see all 8 values output correctly.

06) in **main.py** add in the following function

```
def updateUserInfo(claims, new_string, new_int, new_float):
    entity_key = datastore_client.key('UserInfo', claims['email'])
    entity = datastore_client.get(entity_key)

    entity.update({
        'string_value': new_string,
        'int_value': new_int,
        'float_value': new_float
    })
    datastore_client.put(entity)
```

This function has the same form as the **createUserInfo** function where we create a key and pull in the entity attached to the current user. However, when it comes to updating the entity we will only update the values we have new information for. We do not need to reset any of the other values in the entity that are not changing. Finally we put the new values in the datastore.

07) add a new function in **main.py** with the following code

```
@app.route('/edit_user_info', methods=['POST'])
def editUserInfo():
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    user_info = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            new_string = request.form['string_update']
            new_int = request.form['int_update']
            new_float = request.form['float_update']
            updateUserInfo(claims, new_string, new_int, new_float)

        except ValueError as exc:
            error_message = str(exc)

    return redirect("/")
```

This method will be responsible for updating the **UserInfo** entity when a form we will define in **index.html** has been submitted. The first thing of note here is the application route. The first part of the application route states that this function will be called if the URL points to **edit_user_info** (if running locally this would be http://localhost:8080/edit_user_info). The second part of this annotation states that this method will only accept the "POST" HTTP verb. Meaning the function will only be triggered if the URL is contacted with a post request.

The first part of the function will pull the user claims as before while in the **try catch** block after the user has been verified we then use the **request.form** object to pull back the text associated with the **string_update**, **int_update**, and **float_update** parts of the form (we will define this in the **index.html** template in the next step). Finally we pass this information to the **updateUserInfo()** function to update the user info with those new values. Once the values have been updated the very last thing that will happen is a redirect will be called to / which will force update the main page to reflect the new content that was just entered into this user object.

08) in **index.html** add the following code after the **</dl>** tag in the login info section of **index.html**

```
<form action="/edit_user_info" method="post">
            Int update:<input type="number" value="0" name="int_update"/><br/>
            Float update:<input type="number" value="0.0" name="float_update",
step="any"/><br/>
            String update:<input type="text" value="" name="string_update"/><br/>
            <input type="submit" value="Update values" name="submit_button"/>
</form>
```

This will define a standard HTML form that will provide a place for the user to update the string value, integer value, float value. The three input fields here will be the fields retrieved by the **request.form** object mentioned in 07 above. When you run this application now you will see that you have additional fields for entering in user data and when you update them they will update not only in the datastore but also on the page.

Example 07: Linking Multiple Entities through the use of Keys.

One of the main advantages of a key-value pair database is that access to entities is extremely fast if you have the key to the object. Direct Key access tends to be much quicker than using queries. In this example we will show a simple address book application where a user will be able to add and delete addresses from an address book. We will link a user entity and an address entity through a list of keys in the user object. We will also deal with entity deletion here.

01) take a complete copy of the stripped out template

02) add the following function to **main.py**.

```
def retrieveUserInfo(claims):
    entity_key = datastore_client.key('UserInfo', claims['email'])
    entity = datastore_client.get(entity_key)

    return entity
```

This is the same as the **retrieveUserInfo()** function in previous examples

03) add the following function to **main.py**

```
def createUserInfo(claims):
    entity_key = datastore_client.key('UserInfo', claims['email'])
    entity = datastore.Entity(key = entity_key)
    entity.update({
        'email': claims['email'],
        'name': claims['name'],
        'address_list': []
    })

    datastore_client.put(entity)
```

This is the same as the **createUserInfo()** function in previous examples. The only exception here is that we have an empty list initialised for the **address_list** property of the **UserInfo** entity. We will be storing keys for **Address** entities in this list

04) add the following function to **main.py**

```
def retrieveAddresses(user_info):
    # make key objects out of all the keys and retrieve them
    address_ids = user_info['address_list']
    address_keys = []
    for i in range(len(address_ids)):
        address_keys.append(datastore_client.key('Address', address_ids[i]))

    address_list = datastore_client.get_multi(address_keys)
    return address_list
```

This function will pull out the list of **Address** entities referenced by the passed in **UserInfo** entity (**user_info**). The first two lines will pull the list of keys directly from the **user_info** entity provided. It will also initalise an empty list where we will store all of our key objects. The reason why we do this is we will do a single retrieve request with all the keys rather than do individual retrieves for each key. The reason for this is a single pull with multiple keys is more efficient at this task.

The for loop that follows will pull out each key from the list and create a datastore key out of it before appending it to the **address_keys** list. In the final two lines we provide the list of key objects to **get_multi()** which will pull back the entities associated with each key. It will then return the list of retrieved entities back to the caller.

05) rewrite the **root()** function to have the following code

```
@app.route('/')
def root():
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    user_info = None
    addresses = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            user_info = retrieveUserInfo(claims)
            if user_info == None:
                createUserInfo(claims)
                user_info = retrieveUserInfo(claims)

            addresses = retrieveAddresses(user_info)
        except ValueError as exc:
            error_message = str(exc)

    return render_template('index.html', user_data=claims, error_message=error_message,
user_info=user_info, addresses=addresses)
```

This **root** function is very similar to previous examples. The only changes of note are the following:

- In the variable declarations at the start of the function we have added an extra variable for an address list called **addresses**
- In the call to **render_template** at the end we also pass this list of addresses to the template (defined in a future step)
- In the try block where we verify our user and pull back their user information we make a call to the **retrieveAddresses()** function defined in 04) above with the retrieved **user_info** object. We will need the address entities for templating at a later stage

06) add the following function to **main.py**

```
def createAddress(claims, address1, address2, address3, address4):
    # 63 bit random number that will serve as the key for this address object. not sure
why the data store doesn't like 64 bit numbers
    id = random.getrandbits(63)

    entity_key = datastore_client.key('Address', id)
    entity = datastore.Entity(key = entity_key)
    entity.update({
        'address1': address1,
        'address2': address2,
        'address3': address3,
        'address4': address4
    })
    datastore_client.put(entity)

    return id
```

This function will be tasked with creating an address entity, persisting it in the datastore and returning the **id** of the created entity to the caller of the function. We will need to return this **id** to the caller so we can store the **id** in the relevant **UserInfo** entity and link the entities together. To start with the function will take in five parameters. We have a **claims** object and four strings representing a four line address.

In the first line of the function we are using a random number generator to generate a 63-bit integer. From what I have read of the google documentation it appears entities support 64-bit keys. However, during testing if I used a 64-bit key there would be buggy behaviour where by it would only sometimes accept an entity with a 64-bit key and store it in the datastore. Other times nothing would get stored. 63-bit keys appear to get around this issue. We will also return this key at the end of the function to the caller for storage in a **UserInfo** entity. In the following lines we generate an **Address** key with the random id. We then add the four address lines to that entity and store it in the datastore, before returning the generated id to the caller.

07) add the following function to **main.py**

```
def addAddressToUser(user_info, id):
    address_keys = user_info['address_list']
    address_keys.append(id)
    user_info.update({
        'address_list': address_keys
    })
    datastore_client.put(user_info)
```

This function is used to add an **id** to an **Address** entity to the provided **UserInfo** entity (**user_info**). In the first two lines we pull the current list of keys from the user and append the new id to this list of keys. After this we update the **address_list** property of the entity before commiting the updated **UserInfo** entity to the datastore.

08) add the following function to **main.py**

```
@app.route('/add_address', methods=['POST'])
def addAddress():
    id_token = request.cookies.get("token")
    claims = None
    user_info = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            user_info = retrieveUserInfo(claims)
            id = createAddress(claims, request.form['address1'],
request.form['address2'], request.form['address3'], request.form['address4'])
            addAddressToUser(user_info, id)

        except ValueError as exc:
            error_message = str(exc)

    return redirect('/')
```

This function will be responsible for taking in an address through a HTML form, creating an **Address** entity out of it, and link that entity to the **UserInfo** entity of the current user. It will listen for a POST verb on the **/add_address** relative URL. Like other handlers we will get the **id_token** from the user and verify the user before proceeding any further. After the user is verified we use some of the functions we defined in previous steps. First we pull the **UserInfo** entity for the current user. We then create an **Address** entity out of the address lines provided in the form (defined in a later step) we then take the **id** returned to us for the **Address** entity and attach it to the current user. Once all that is done we then redirect back to the root URL to refresh the list of addresses the user has.

09) add the following function to **main.py**

```
def deleteAddress(claims, id):
    user_info = retrieveUserInfo(claims)
    address_list_keys = user_info['address_list']

    address_key = datastore_client.key('Address', address_list_keys[id])
    datastore_client.delete(address_key)

    del address_list_keys[id]
    user_info.update({
        'address_list' : address_list_keys
    })
    datastore_client.put(user_info)
```

This function will be used for deleting an address from the current user. The function takes in a user **claims** object along with an index to the address key that needs to be removed from the datastore. In the first pair of lines we pull the **UserInfo** entity and the list of address keys from the datastore. In the next pair of lines we make a **Key** object out of the key stored in the list. When that key is provided to the **delete()** function it instructs the datastore to remove that **Address** entity with the given key. In the final set of lines we remove the key we just deleted from the **UserInfo's** list of keys before updating it in the **UserInfo** object and persisting it to the datastore.

10) add the following function to **main.py**

```
@app.route('/delete_address/<int:id>', methods=['POST'])
def deleteAddressFromUser(id):
    id_token = request.cookies.get("token")
    error_message = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token, 
firebase_request_adapter)
            deleteAddress(claims, id)
        except ValueError as exc:
            error_message = str(exc)

    return redirect('/')
```

This function will be called whenever we wish to delete an address from the current user. The first thing to note is the **@app.route** annotation as it looks a little different to the ones you have seen previously. Here we have **/delete_address/<int:id>** what this means is that the number that appears immediately after **/delete_address/** will be treated as an integer and will be passed into the attached handler with a variable name of **id** as seen in the function definition on the line below.

Like the other handlers we will first verify our user before proceeding any further. But when the user is verified we will call our **deleteAddress()** function with the index that was passed in through the URL. e.g. if **/delete_address/0** is called **id** will be 0 if **/delete_address/1** is called then **id** will be 1 etc. Finally we will redirect to the root to update the list of addresses once the current address has been removed.

11) In **index.html** replace the template code between the **login-info <div>** tags with the following:

```
<h2>Login Info</h2>
{% if user_data %}
  <dl>
      <dt>Name:</dt><dd>{{ user_data['name'] }}</dd>
      <dt>Email:</dt><dd>{{ user_data['email'] }}</dd>
  </dl>

  <!-- form for adding in a new address -->
  <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" name="button"/>
  </form>
  <br/>

  <!-- block that will print out the addresses in turn -->
  {% for address in addresses %}
      Address {{ loop.index - 1 }}<br/>
      Address line 1:{{ address.address1 }}<br/>
      Address line 2:{{ address.address2 }}<br/>
      Address line 3:{{ address.address3 }}<br/>
      Address line 4:{{ address.address4 }}<br/>
      <br/>
      <form action="/delete_address/{{ loop.index - 1 }}" method="post">
          <input type="submit" name="delete"/>
      </form>
  {% endfor %}
{% elif error_message %}
  <p>Error Message: {{ error_message }}</p>
{% endif %}
```

The error message section is the same as previous examples howver the **user_data** section has changed. Like previous examples the first part uses the provided user data to print out the name and email address of the currently logged in user. In the second part we have a HTML form that will take in four lines of text that will form an address. When the submit button on this form is clicked it will trigger the **/add_address** URL which will add the address into the current user's address book.

Finally we have a looping template block that has two jobs: to display the addresses that the user has added to their address book. And also to add delete buttons to each of the addresses should a user wish to delete that address. In both take note of the **{{ loop.index – 1 }}** part. **loop.index** is a special variable maintained by Jinja noting the current iteration of the for loop. It starts at 1 rather than zero, hence the **-1** part to bring it back to zero. We use this loop index to setup the dynamic addresses for deletion. On the first pass **/delete_address/{{ loop.index – 1 }}** will become **/delete_address/0**. On the second pass this becomes **/delete_address/1** and so on.

When you run this application you should be able to add and remove addresses from the application. The data should also be segregated between users. While it looks like a lot of effort to setup keys the speed up is worth it as direct key access is always quicker than running a query. This kind of direct key access is useful in many situations particularly where you may have a recursive relationship between entities of the same type.

Example 08: Linking multiple entities together through sub entities

In this example we will take the previous example and implement it through the use of sub entities rather than through keys. The advantage of an approach like this is that when the **UserInfo** entity is retrieved, it will also retrieve all of the address entities at the same time. The downside is that entities will take longer to retrieve and update as you are pulling back entire entities at the same time rather than keys. The decision to use subentities or keys to link multiple entities together should be on a case by case basic. You will need to think carefully about how data is accessed and queried to determine which is the best approach to use.

A lot of the code in this example will be very similar to the previous example and we will note any changes as we go along.

01) take a complete copy of the stripped out template

02) in **main.py** add the following function

```
def retrieveUserInfo(claims):
    entity_key = datastore_client.key('UserInfo', claims['email'])
    entity = datastore_client.get(entity_key)

    return entity
```

This is the same as the previous example

03) in main.py add the following function

```
def createUserInfo(claims):
    entity_key = datastore_client.key('UserInfo', claims['email'])
    entity = datastore.Entity(key = entity_key)
    entity.update({
        'email': claims['email'],
        'name': claims['name'],
        'address_list': []
    })

    datastore_client.put(entity)
```

This is the same as the previous example

04) in **main.py** rewrite the **root()** function to have the following code

```
@app.route('/')
def root():
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    user_info = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)
            user_info = retrieveUserInfo(claims)
            if user_info == None:
                createUserInfo(claims)
                user_info = retrieveUserInfo(claims)

        except ValueError as exc:
            error_message = str(exc)

    return render_template('index.html', user_data=claims, error_message=error_message,
user_info=user_info)
```

The big difference here between this and the previous example is that we no longer use a function to retrieve the addresses of the user. As they will be sub entities contained within the **UserInfo** entity they will be automatically retrieved when we retrieve the **UserInfo** entity.

05) add the following function to **main.py**

```
def createAddress(address1, address2, address3, address4):
    entity = datastore.Entity()
    entity.update({
        'address1': address1,
        'address2': address2,
        'address3': address3,
        'address4': address4
    })

    return entity
```

This **createAddress()** function is a little different to the **createAddress()** function in the previous example. The first difference is that we no longer create a key for or store the entity directly in the datastore. Here we create a simple entity with all the address fields we require and will return it to the caller. The intention here is that when the entity is added to the **UserInfo** object we do not need to do a seperate put command for the **Address** or need a key for it as it will be put into the datastore when we put the **UserInfo** entity into the datastore.

06) add the following function to **main.py**

```
def addAddressToUser(user_info, address_entity):
    addresses = user_info['address_list']
    addresses.append(address_entity)
    user_info.update({
        'address_list': addresses
    })
    datastore_client.put(user_info)
```

This function will take in two parameters: the retrieved user information from the data store, and the new address object that is to be added to this user. The function will then first retrive the list of **address** entities from the current user info and will then append the new **address** entity to it. As the list has been updated the updated user info object must then be put in the datastore to store the changes that have been made.

07) add the following function to **main.py**

```
@app.route('/add_address', methods=['POST'])
def addAddress():
    id_token = request.cookies.get("token")
    claims = None
    user_info = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            user_info = retrieveUserInfo(claims)
            address = createAddress(request.form['address1'], request.form['address2'],
request.form['address3'], request.form['address4'])
            addAddressToUser(user_info, address)

        except ValueError as exc:
            error_message = str(exc)

    return redirect('/')
```

This function will be responsible for taking a new address object from the user and appending it to the current user's information in the datastore. It is set to listen for POST requests on the **/add_address** relative URL. Like the **root()** function defined earlier the first things the function will do is pull the access token and verify the user's claims before attempting anything else. Should the user claims work out we will then retrieve the user's data from the datastore and then create an address object out a form (to be defined later) that has been filled in by the user. Normally this would be a good place to do data validation on any input that comes in through a form but this is omitted for the sake of the example. We then add this address using the **addAddressToUser()** function defined in 06) above. Assuming all is ok we will then redirect back to the root page to force the user display to update with the new address.

08) in **main.py** add in the following function

```
def deleteAddress(claims, id):
    user_info = retrieveUserInfo(claims)
    address_list = user_info['address_list']

    del address_list[id]
    user_info.update({
        'address_list' : address_list
    })
    datastore_client.put(user_info)
```

This function will be used to delete an **address** from a user object. It takes in two parameters one being the **claims** of the user so we can retrieve the user object and the other being an index into the list of addresses that the user wants to remove. The first two lines will pull out the user info object and will then get the **address** list of the user. The **del** command on the next line is the command that will actually remove the address from the address list. Note that this matches standard python syntax for removing an element from a python list. We then update the address list of the user and put it back in the datastore to store the updated changes.

09) in **main.py** add in the following function

```
@app.route('/delete_address/<int:id>', methods=['POST'])
def deleteAddressFromUser(id):
    id_token = request.cookies.get("token")
    error_message = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)
            deleteAddress(claims, id)
        except ValueError as exc:
            error_message = str(exc)

    return redirect('/')
```

This function will be responsible for removing an address when it is requested by the user. It is also an example of something called a dynamic route in Flask. Note the URL of the route in the attached annotation **/delete_address/<int:id>** What this means to Flask is that if a URL is passed to **/delete_address/** with an integer added after it then that request will be redirected to this function regardless of what integer is place there. The **<int:id>** states that an integer should be expected and if found should be assigned the variable name **id**. This variable **id** then gets passed into the parameter list of the function itself.

Like previous routes that we have defined we will first check the user claims before doing any functionality. Once verified we will call the **deleteAddress()** function defined in 08) above to remove the address from the user before redirecting to the root URL to refresh the page automatically for the user.

10) In **index.html** replace the template code between the **login-info <div>** tags with the following

```
<h2>Login Info</h2>
  {% if user_data %}
    <dl>
        <dt>Name:</dt><dd>{{ user_data['name'] }}</dd>
        <dt>Email:</dt><dd>{{ user_data['email'] }}</dd>
    </dl>

    <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" name="button"/>
    </form>
    <br/>

    {% for address in user_info['address_list'] %}
        Address {{ loop.index - 1 }}<br/>
        Address line 1:{{ address.address1 }}<br/>
        Address line 2:{{ address.address2 }}<br/>
        Address line 3:{{ address.address3 }}<br/>
        Address line 4:{{ address.address4 }}<br/>
        <br/>
        <form action="/delete_address/{{ loop.index - 1 }}" method="post">
            <input type="submit" name="delete"/>
        </form>
    {% endfor %}
```

This code is pretty similar to the code for the template as the last example. There are no significant changes here.

Example 09: Different ways of adding and deleting multiple entities to/from the data store.

Up to now we have only dealt with adding, updating, or deleting one entity at a time with the datastore. However, to save on API calls and also to offer the potential for speedup there are multiple different ways that can be used to add or remove entities. The last two Batching and Transactions allow you to combine multiple operations of different types either for speedup or consistency purposes respectively.

In this example we will show with the use of dummy data how all of thee operations function. Please note that in order to take advantage of these operations you will need to have the physical entities themselves or have the keys for accessing those entities.

NOTE: that for this example you will need to have a look at the datastore directly to see the changes being made as we will not display them directly on the returned page.

01) take a complete copy of the stripped out template

02) in **main.py** add the following function

```
def createUserInfo(claims):
    entity_key = datastore_client.key('UserInfo', claims['email'])
    entity = datastore.Entity(key = entity_key)
    entity.update({
        'email': claims['email'],
        'name': claims['name'],
        'address_list': []
    })

    datastore_client.put(entity)
```

This is the same as previous examples

03) in **main.py** add the following function

```
def retrieveUserInfo(claims):
    entity_key = datastore_client.key('UserInfo', claims['email'])
    entity = datastore_client.get(entity_key)

    return entity
```

This is the same as previous examples

04) in **main.py** add the following function

```
@app.route('/')
def root():
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    times = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            user_info = retrieveUserInfo(claims)
            if user_info == None:
                createUserInfo(claims)
                user_info = retrieveUserInfo(claims)

        except ValueError as exc:
            error_message = str(exc)

    return render_template('index.html', user_data=claims, error_message=error_message)
```

This is very similar to previous examples

05) add the following function to **main.py**

```
def createDummyData(number):
    entity_key = datastore_client.key('DummyData', number)
    entity = datastore.Entity(key = entity_key)
    entity.update({
        'number': number,
        'squared': number ** 2,
        'cubed': number ** 3
    })

    return entity
```

This function will be responsible for creating an entity that we will use as part of our batch operations. It’s a simple entity that will store a number it’s square and also its cube. We will use the number itself as the **id** for the entity. This will allow us to create multiple independent entities that can be used as part of a batch or a transaction. This will be an important detail later on.

06) add the following function to **main.py**

```
@app.route('/multi_add', methods=['POST'])
def multiAdd():
    id_token = request.cookies.get("token")
    error_message = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            entity_1 = createDummyData(1)
            entity_2 = createDummyData(2)
            entity_3 = createDummyData(3)
            entity_4 = createDummyData(4)

            datastore_client.put_multi([entity_1, entity_2, entity_3, entity_4])

        except ValueError as exc:
            error_message = str(exc)

    return redirect('/')
```

This is similar in structure to the **root** function but there are some small differences. For a start this will listen for a post action on the **/multi_add** relative URL. After checking user claims this will generate four **DummyData** entities. Once the entities are created it will then create a list out of all four entities before passing this list to the **put_multi()** function. This function will take a list of entities (they don’t all have to share the same type) and in a single API call submit them to the datastore. This will be quicker than sending four seperate put requests and will save on API calls too.

07) add the following function in **main.py**

```
@app.route('/batch_add', methods=['POST'])
def batchAdd():
    id_token = request.cookies.get("token")
    error_message = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token, 
firebase_request_adapter)

            entity_5 = createDummyData(5)
            entity_6 = createDummyData(6)
            entity_7 = createDummyData(7)
            entity_8 = createDummyData(8)

            batch = datastore_client.batch()
            with batch:
                batch.put(entity_5)
                batch.put(entity_6)
                batch.put(entity_7)
                batch.put(entity_8)

        except ValueError as exc:
            error_message = str(exc)

    return redirect('/')
```

This is similar to 06) above however the difference here is that we are using a batch operation to store the entities instead of **put_multi()**. We also have the additional **with** syntax to be aware of. Prior to this syntax we use the batch() function of the datastore client to prepare a batch operation for us. The **with batch:** syntax is used to setup a batch operation without having to write the necessary try catch functionality as it is handled by this syntax through the use of a **ContextManager** in the background. Whatever operations are put here (and this can be a mix of put, delete, or other operations) will be collected into on operation. When the end of the **with** block is reached the batch operation is then sent to the datastore to be completed. Like **put_multi** this will reduce the time needed and the number of API calls needed to commit these changes to the datastore. However there is one additional benefit to this. In a batch operation if there are multiple objects that are independent of each other (are not linked or tied to each other) then they will be committed in parallel and the overall set of operations will complete in a shorter timeframe.

08) in **main.py** add the following function

```
@app.route('/transaction_add', methods=['POST'])
def transactionAdd():
    id_token = request.cookies.get("token")
    error_message = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            entity_9 = createDummyData(9)
            entity_10 = createDummyData(10)
            entity_11 = createDummyData(11)
            entity_12 = createDummyData(12)

            transaction = datastore_client.transaction()
            with transaction:
                transaction.put(entity_9)
                transaction.put(entity_10)
                transaction.put(entity_11)
                transaction.put(entity_12)

        except ValueError as exc:
            error_message = str(exc)

    return redirect('/')
```

This is similar to 07) above however the key difference here is that we are using a transaction to store the entities instead of a batch operation. The difference being is that a transaction will take additional steps to ensure all data is safely and consistently committed to the datastore. Should one of the operations fail the entire transaction will be rolled back to ensure the datastore remains in a consistent state.

09) add the following function into **main.py**

```
@app.route('/multi_delete', methods=['POST'])
def multiDelete():
    id_token = request.cookies.get("token")
    error_message = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token, 
firebase_request_adapter)

            entity_1_key = datastore_client.key('DummyData', 1)
            entity_2_key = datastore_client.key('DummyData', 2)
            entity_3_key = datastore_client.key('DummyData', 3)
            entity_4_key = datastore_client.key('DummyData', 4)

            datastore_client.delete_multi([entity_1_key, entity_2_key, entity_3_key, 
entity_4_key])

        except ValueError as exc:
            error_message = str(exc)

    return redirect('/')
```

This is similar to 06) above with a couple of significant differences. The first being that instead of generating entities for the dummy data we are only generating keys instead. Subsequently we are then calling the **delete_multi()** function with that list of keys to remove all four entities at once. Again this will save on API calls and will be quicker than issuing four seperate delete operations at once.

10) add the following code into **main.py**

```
@app.route('/batch_delete', methods=['POST'])
def batchDelete():
    id_token = request.cookies.get("token")
    error_message = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            entity_5_key = datastore_client.key('DummyData', 5)
            entity_6_key = datastore_client.key('DummyData', 6)
            entity_7_key = datastore_client.key('DummyData', 7)
            entity_8_key = datastore_client.key('DummyData', 8)

            batch = datastore_client.batch()
            with batch:
                batch.delete(entity_5_key)
                batch.delete(entity_6_key)
                batch.delete(entity_7_key)
                batch.delete(entity_8_key)

        except ValueError as exc:
            error_message = str(exc)

    return redirect('/')
```

This is similar to 07) above however like the previous step we are using the batch to delete multiple entities in the same batch. Similar to addition if these entities are independent of each other then the deletions will occur in parallel.

11) add the following function into **main.py**

```
@app.route('/transaction_add', methods=['POST'])
def transactionAdd():
    id_token = request.cookies.get("token")
    error_message = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            entity_9 = createDummyData(9)
            entity_10 = createDummyData(10)
            entity_11 = createDummyData(11)
            entity_12 = createDummyData(12)

            transaction = datastore_client.transaction()
            with transaction:
                transaction.put(entity_9)
                transaction.put(entity_10)
                transaction.put(entity_11)
                transaction.put(entity_12)

        except ValueError as exc:
            error_message = str(exc)

    return redirect('/')
```

This is similar to 08) above and like the previous two steps we are using the transaction to delete the entities here. If one of the deletes fails along the way the transaction will detect this and will rollback the transaction to the last known good consistent state.

12) change the contents of the **login-info <div/>** tags to have the following HTML code

```
      <h2>Login Info</h2>
      {% if user_data %}
        <dl>
            <dt>Name:</dt><dd>{{ user_data['name'] }}</dd>
            <dt>Email:</dt><dd>{{ user_data['email'] }}</dd>
        </dl>

        <form action="/multi_add" method="post">
            Batch add four entities through put_multi: <input type="submit"
value="Click here"/>
        </form>

        <form action="/batch_add" method="post">
            Batch add four entities through batch: <input type="submit" value="Click
here"/>
        </form>

        <form action="/transaction_add" method="post">
            Batch add four entities through transaction: <input type="submit"
value="Click here"/>
        </form>

        <form action="/multi_delete" method="post">
            Batch delete four entities through delete_multi: <input type="submit"
value="Click here"/>
        </form>

        <form action="/batch_delete" method="post">
            Batch delete four entities through batch: <input type="submit" value="Click
here"/>
        </form>

        <form action="/transaction_delete" method="post">
            Batch delete four entities through transaction: <input type="submit"
value="Click here"/>
        </form>

      {% elif error_message %}
        <p>Error Message: {{ error_message }}</p>
      {% endif %}
```

This will simply add 6 buttons to trigger the add and delete functionality of the 6 functions we added in the previous steps. After you click each one make sure you check the datastore to see if those changes occurred.

Example 10: Basic queries with the datastore

In this example we will show how some basic queries can be performed with the datastore. One thing you should be aware of in advance is that the expressive power you may be used to with SQL will not be available here. The queries can only be done on a single entity type and only permit the operators of >, <, =, >=, and <= there is no operator for not equals. However this can be easily replicated by combining the <, and > operators to avoid the value that you do not wish to retrieve.

In the example below we will show some basic queries that can be run on entities in the datastore. Queries are capable of working on the following datatypes: **int**, **string**, **boolean**, and **datetime**. Should your application require queries it would be wise to consider in advance the kind of queries that you will need to answer and structure your data such that you can answer all queries by using a combination of the basic query types that are supported here.

One thing to note however. When queries retrieve entities from the datastore you may run into an issue with the eventual consistency nature of the datastore. Eventual consistency ensures that at some point in the future updates will be commited to the datastore but does not specify when. Where this normally occurs is when you insert an entity into the datastore then attempt to retrieve it immediately afterwards using a query on the next request. It may not show up as a result of the query as it may not have been written to the datastore yet. However if you give it some time and try the same query again the entity will show up after it has been committed. The only way to force strong consistency (i.e. when an entity has been added or updated you can see the changes immediately) is to use direct key access instead of a query to access the entity.

01) take a copy of the stripped out template.

02) in **main.py** add the following function

```
def createDummyData(name, id, boolean):
    entity_key = datastore_client.key('DummyData', id)
    entity = datastore.Entity(key = entity_key)
    entity.update({
        'name': name,
        'id': id,
        'boolean': boolean
    })

    return entity
```

Like the previous example we will need to put some data in the datastore to see how the queries will work. In this data we have three fields, one of the string type, an integer, and a boolean. We will use these to run our queries.

03) in **main.py** replace the **root()** function with this **root()** function

```
@app.route('/')
def root():
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    times = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            query = datastore_client.query(kind='DummyData')
            result = query.fetch()

        except ValueError as exc:
            error_message = str(exc)

    return render_template('index.html', user_data=claims, error_message=error_message,
data=result)
```

Like the previous examples we will check the credentials of the user first before we proceed with any further processing. After the claims have been verified is where we see the changes here. First we have a call to the **datastore_client.query()** function and we have specified the kind of entity that we wish the query to work on (in this case **DummyData** entities). Note that this will not fetch the entities straight away as this will enable us to add additional filters (as you will see in later steps) but we have to explicitly call a fetch when we are ready (as we do on the next line). If no filters have been specified as part of a query the default fetch behavour is to return the entire set of entities of the requested kind. Once we have this list we will then pass it to the render template as we wish to display the data to the user so they can experiment with some basic queries.

03) in **index.html** replace the HTML code in the **login-info <div/>** tag with the following code

```
      <h2>Login Info</h2>
      {% if user_data %}
        <dl>
            <dt>Name:</dt><dd>{{ user_data['name'] }}</dd>
            <dt>Email:</dt><dd>{{ user_data['email'] }}</dd>
        </dl>

        <form action="/initialise_dummy_data" method="post">
            initialise the entities in the datastore<input type="submit"/>
        </form>

        <form action="/pull_entity_by_id" method="post">
            pull an entity by a given id <input type="number" name="id"><input 
type="submit"/>
        </form>

        <form action="/pull_entity_by_name" method="post">
            pull an entity by name <input type="text" name="name"><input 
type="submit"/>
        </form>

        <form action="/query_multiple_attribs" method="post">
            example filter on multiple attributes<input type="submit"/>
        </form>

        {% for i in data %}
        Name: {{ i['name'] }}<br/>
        ID: {{ i['id'] }}<br/>
        Boolean: {{ i['boolean'] }}<br/>
        {% endfor %}

      {% elif error_message %}
        <p>Error Message: {{ error_message }}</p>
      {% endif %}
```

The first thing to note here is that we have four seperate forms for triggering different basic query types. The first however will be used to add some initial data to the datastore for the queries to work. The second will pull back a single entity that has a matching ID number, The third will pull back a single entity that has a matching name, and finally the last one will pull back entities that have an ID less than four but the boolean is also set to true. We also have a **{% for %} {% endfor %}** towards the end of the code that will take the data parameter passed to the template and it will render whatever **DummyData** entities are listed there to the web page so you can see what data is being retrieved as part of a query.

04) in **main.py** add the following function

```
@app.route('/initialise_dummy_data', methods=['POST'])
def initialiseDummyData():
    entity_1 = createDummyData("foo", 1, True)
    entity_2 = createDummyData("bar", 2, False)
    entity_3 = createDummyData("baz", 3, False)
    entity_4 = createDummyData("wookie", 4, True)

    datastore_client.put_multi([entity_1, entity_2, entity_3, entity_4])

    return redirect('/')
```

This is a simple function where we create four sets of dummy data and put them in the datastore.

05) in **main.py** add the following function

```
@app.route('/pull_entity_by_id', methods=['POST'])
def pullEntityById():
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    times = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            if request.form['id'] == '':
                return redirect('/')

            id = int(request.form['id'])

            query = datastore_client.query(kind='DummyData')
            query.add_filter('id', '=', id)
            result = query.fetch()

        except ValueError as exc:
            error_message = str(exc)

    return render_template('index.html', user_data=claims, error_message=error_message,
data=result)
```

Like the other handlers the first thing we do here is check the user claims before going any further. Once this is done however this is where our processing changes. Immediately after the claims have been checked we look at the form to see if an id number has been requested. If not then this will return the empty string and the if statement here will redirect back to the **root()** function to display all the entities that are stored in the datastore. However, if this is not the case we will then retrieve the id from the form and construct a query. The most important part of the query is the **add_filter()** function. This function takes three parameters

1. the name/key field the filter is to be applied on
2. the comparison operator <, >, =, >=, <=
3. the value to be compared

In this case we are filtering on the **id** field and we want the **id** to match the one that was passed into the form. Thus one of two things will happen with the fetch command. If it exists then the result list will contain a single entity. If not the result list will be empty. Finally at the end of the function we make a call to render the template of **index.html** with the given result to show the user the entity they retrieved through their search.

06) in **main.py** add the following function:

```
@app.route('/pull_entity_by_name', methods=['POST'])
def pullEntityByName():
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    times = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            if request.form['name'] == '':
                return redirect('/')

            name = request.form['name']

            query = datastore_client.query(kind='DummyData')
            query.add_filter('name', '=', name)
            result = query.fetch()

        except ValueError as exc:
            error_message = str(exc)

    return render_template('index.html', user_data=claims, error_message=error_message,
data=result)
```

This is almost the exact same as 04) above except this time we are doing a string comparison on the name field rather than a numerical comparison on the id field.

07) add the following function to **main.py**:

```
@app.route('/query_multiple_attribs', methods=['POST'])
def queryMultipleAttribs():
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    times = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            query = datastore_client.query(kind='DummyData')
            query.add_filter('id', '<', 4)
            query.add_filter('boolean', '=', True)
            result = query.fetch()

        except ValueError as exc:
            error_message = str(exc)

    return render_template('index.html', user_data=claims, error_message=error_message,
data=result)
```

This is similar to the last two functions that were added, however this time we have an example of a compound query on more than one attribute. Here we are querying on the **id** field having a value less than four and also the boolean field has been set to true.

08) finally we must create an **index.yaml** file and add the following to it

```
indexes:
- kind: DummyData
  properties:
  - name: boolean
- name: id
```

This needs to be done for the last query as for some reason (not sure why) google app engine requires that indices for the id and boolean fields be present in order to run the final query. Make sure to upload this index to your datastore before you run this example.

Example 11: Working with the Google Cloud Storage system to upload and download files.

In this example we will show how to work with the Google Cloud Storage system to upload and download blobs (Binary Large Objects) that are too large to be stored in the datastore. By default whenever an application is created a default Google Cloud Storage bucket is also created that has the same name as the application. For testing and development purposes the first 5GB of this is free and will be more than enough for interacting with in this course and for assignments.

The example we have here will show how to create directories in the storage system and also how to upload and download files through the user's browser.

01) take a copy of the stripped out template.

02) add a file called **local_constants.py** and add the following code to it

```
PROJECT_NAME='<your project name goes here>'
PROJECT_STORAGE_BUCKET='<your storage bucket goes here>'
```

These constants are refactored out to their own file to make them easy to reuse and also to prevent us from having to copy and paste them everytime we need to use these constants. You can fill in the appropriate values here by finding the same values you have already defined in app-setup.js If you want to have a look at the storage bucket contents go to

http://console.cloud.google.com/

and click on storage to see your buckets

03) in **main.py** add the following function

```
def createUserInfo(claims):
    entity_key = datastore_client.key('UserInfo', claims['email'])
    entity = datastore.Entity(key = entity_key)
    entity.update({
        'email': claims['email'],
        'name': claims['name'],
    })

    datastore_client.put(entity)
```

This is very similar to previous examples

04) in **main.py** add the following function

```
def retrieveUserInfo(claims):
    entity_key = datastore_client.key('UserInfo', claims['email'])
    entity = datastore_client.get(entity_key)

    return entity
```

This is very similar to previous examples

05) in **main.py** add the following function

```
def blobList(prefix):
    storage_client = storage.Client(project=local_constants.PROJECT_NAME)

    return storage_client.list_blobs(local_constants.PROJECT_STORAGE_BUCKET,
prefix=prefix)
```

This function will first generate a Google Cloud Storage client that will be set to work on the project that is defined by the constants in 02) above. In the next step we ask the storage client to list all of the blobs in the storage bucket attached to this project and return that list to the caller.

You may also be wondering why we have a prefix variable. It is possible to create directory structures in a storage bucket. The prefix can be used to filter out the blob list to only include the blobs that exist under a particular subdirectory e.g. if I have a directory called **test/** and I specify a prefix of **test/** I will only see the directories and blobs under that **test/** directory

06) in **main.py** replace the **root()** function with the following code

```
@app.route('/')
def root():
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    times = None
    user_info = None
    file_list = []
    directory_list = []

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            user_info = retrieveUserInfo(claims)
            if user_info == None:
                createUserInfo(claims)
                user_info = retrieveUserInfo(claims)

            blob_list = blobList(None)
            for i in blob_list:
                if i.name[len(i.name) - 1] == '/':
                    directory_list.append(i)
                else:
                    file_list.append(i)

        except ValueError as exc:
            error_message = str(exc)

    return render_template('index.html', user_data=claims, error_message=error_message,
user_info=user_info, file_list=file_list, directory_list=directory_list)
```

This has a little more processing in it compared to previous examples but not much more. Like before we will check the user's credentials and if the user object does not exist for this user we will generate it for them. In the following lines we will then use the **blobList()** function defined previously to pull out the list of blobs stored in the bucket. We then seperate the blobs into two seperate lists of directories and files. Any blob with a trailing / is a directory and any blob without this trailing slash is a file. We then pass these two lists to the template, along with all other required information.

07) in **index.html** replace the HTML code in the **login-info <div/>** tag with the following

```
<h2>Login Info</h2>
{% if user_data %}
  <dl>
      <dt>Name:</dt><dd>{{ user_data['name'] }}</dd>
      <dt>Email:</dt><dd>{{ user_data['email'] }}</dd>
  </dl>

  <form action="/add_directory" method="post">
      Directory Name: <input type="text" name="dir_name"/><input type="submit"/>
  </form>

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

  <h1>list of directories</h1><br/>
  {% for i in directory_list %}
  {{ i.name }}<br/>
  {% endfor %}

  <h1>list of files</h1><br/>
  {% for i in file_list %}
  <form action="/download_file/{{ i.name }}" method="post">
  {{ i.name }}<input type="submit"/><br/>
  </form>
  {% endfor %}

{% elif error_message %}
  <p>Error Message: {{ error_message }}</p>
{% endif %}
```

This has four main parts. In the first form we will have a form for adding a directory blob to the storage bucket. The second form we will use to select a file from the user's machine for uploading to the bucket. Then we have a **for** loop that will output the list of directories in the bucket. And finally we have a **for** loop that will generate a form for each file in the bucket that could potentially be download through the use of a dynamic url consisting of **/download_file/** followed by the name of the file itself.

08) in **main.py** add the following function

```
def addDirectory(directory_name):
    storage_client = storage.Client(project=local_constants.PROJECT_NAME)
    bucket = storage_client.bucket(local_constants.PROJECT_STORAGE_BUCKET)

    blob = bucket.blob(directory_name)
    blob.upload_from_string('', content_type='application/x-www-form-
urlencoded;charset=UTF-8')
```

This function will be responsible for adding a directory to the bucket. In the first two lines we get access to the storage client and then request the storage client to act on the bucket we have defined for this project. In the third line we initialise a blob object with the name of the directory we wish to define. The final line is what generates the directory in the bucket itself. It looks strange but this is the convention that Google Cloud Storage uses to add directories into a bucket for all directories.

09) in **main.py** add the following function

```
def addFile(file):
    storage_client = storage.Client(project=local_constants.PROJECT_NAME)
    bucket = storage_client.bucket(local_constants.PROJECT_STORAGE_BUCKET)

    blob = bucket.blob(file.filename)
    blob.upload_from_file(file)
```

This function is responsible for adding a file to the bucket from the user’s system. Note that is similar in structure to 08) above but the only real difference here is in the last two lines where we have to set the filename of the blob (which is provided by the incoming file parameter to this function) and ask for the blob to be uploaded from the contents of that provided file

10) in **main.py** add the following function

```
def downloadBlob(filename):
    storage_client = storage.Client(project=local_constants.PROJECT_NAME)
    bucket = storage_client.bucket(local_constants.PROJECT_STORAGE_BUCKET)

    blob = bucket.blob(filename)
    return blob.download_as_bytes()
```

This function will be responsible for getting the content of a file such that it can be downloaded by a user. There are methods that allow you to download a file directly but these are only to be used by the application itself as it will download a file in the same directory as where the application resides. If you want to send a file to a user’s browser you will need to get the bytes first and generate a Response object out of it which we will do in a later step when we define the handler for downloading a file.

11) in **main.py** add the following function

```
@app.route('/add_directory', methods=['POST'])
def addDirectoryHandler():
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    times = None
    user_info = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            directory_name = request.form['dir_name']
            if directory_name == '' or directory_name[len(directory_name) - 1] != '/':
                return redirect('/')

            user_info = retrieveUserInfo(claims)
            addDirectory(directory_name)

        except ValueError as exc:
            error_message = str(exc)

    return redirect('/')
```

This handler will be responsible for adding a directory to the bucket. Like the previous handlers we have defined the first thing we do is check that we have a valid user before doing any other processing. After this has been checked we pull the entered directory name from the form the user has provided. If no directory name was provided or the directory does not have a trailing slash then we will redirect back to root as there is nothing we can add here. After this we when use the **addDirectory()** function to add the directory to our storage bucket before redirecting back to root to show that the directory has been added.

12) in **main.py** add the following function

```
@app.route('/upload_file', methods=['post'])
def uploadFileHandler():
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    times = None
    user_info = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

            file = request.files['file_name']
            if file.filename == '':
                return redirect('/')

            user_info = retrieveUserInfo(claims)
            addFile(file)

        except ValueError as exc:
            error_message = str(exc)

    return redirect('/')
```

This function is responsible for uploading a file to the bucket. Like the other handlers we check the claims of the user and if this checks out then pull the file from the form that was submitted by the user. Before attempting to upload the file we first check to see if a file was selected. If not we redirect to **root** as there is nothing to do. We then add the file which will upload the file to the bucket using the functionality defined earlier.

13) in **main.py** add the following function

```
@app.route('/download_file/<string:filename>', methods=['POST'])
def downloadFile(filename):
    id_token = request.cookies.get("token")
    error_message = None
    claims = None
    times = None
    user_info = None
    file_bytes = None

    if id_token:
        try:
            claims = google.oauth2.id_token.verify_firebase_token(id_token,
firebase_request_adapter)

        except ValueError as exc:
            error_message = str(exc)

    return Response(downloadBlob(filename), mimetype='application/octet-stream')
```

This function is similar to previous handlers in that we check the user claims first before proceeding. However, at the end of the function in order to send the file through the user's browser we must get the bytes of the file using the **downloadBlob()** function defined earlier. To ensure the file gets downloaded and is not displayed in the browser we use the octet-stream mimetype. Finally this all gets wrapped up in a response object so Flask will generate a response and send it directly to the browser of the user