Foundations of Cloud Computing

CEN 5086-001

GROUP10 – FINAL PROJECT REPORT

Team Details:

Name	Znumber	Email Address
Akhil Yarlagadda	Z23689300	ayarlagadda2022@fau.edu
Raviteja Allu	Z23693651	rallu2022@fau.edu
Subash Gupta Karamsetty	Z23689645	skaramsetty2022@fau.edu
Sandeep Mandapati	Z23684014	smandapati2022@fau.edu
Seema Srinivas	Z23707807	ssrinivas2023@fau.edu

Project Details:

Project Name: Group10-Project1

Project ID: group10-project1

Bucket Name: group10project2

Datastore database Name: default

Cloud Run Service Name: finalapp

Secret Manager secret names: authentication-credentials, firebase-config

Application URL (External IP Address): https://finalapp-vseygp636a-uc.a.run.app

GitHub Repository Link for the code files (Public): SubashGupta/finalproject_group10 (github.com)

GCP Project link: Welcome – Group10-Project1 – Google Cloud console

Firebase Setup:

Firebase project console link: Group10-Project1 – Overview – Firebase console (google.com)

Firebase Project Name: Group10-Project1

Webapp Name: imageapp

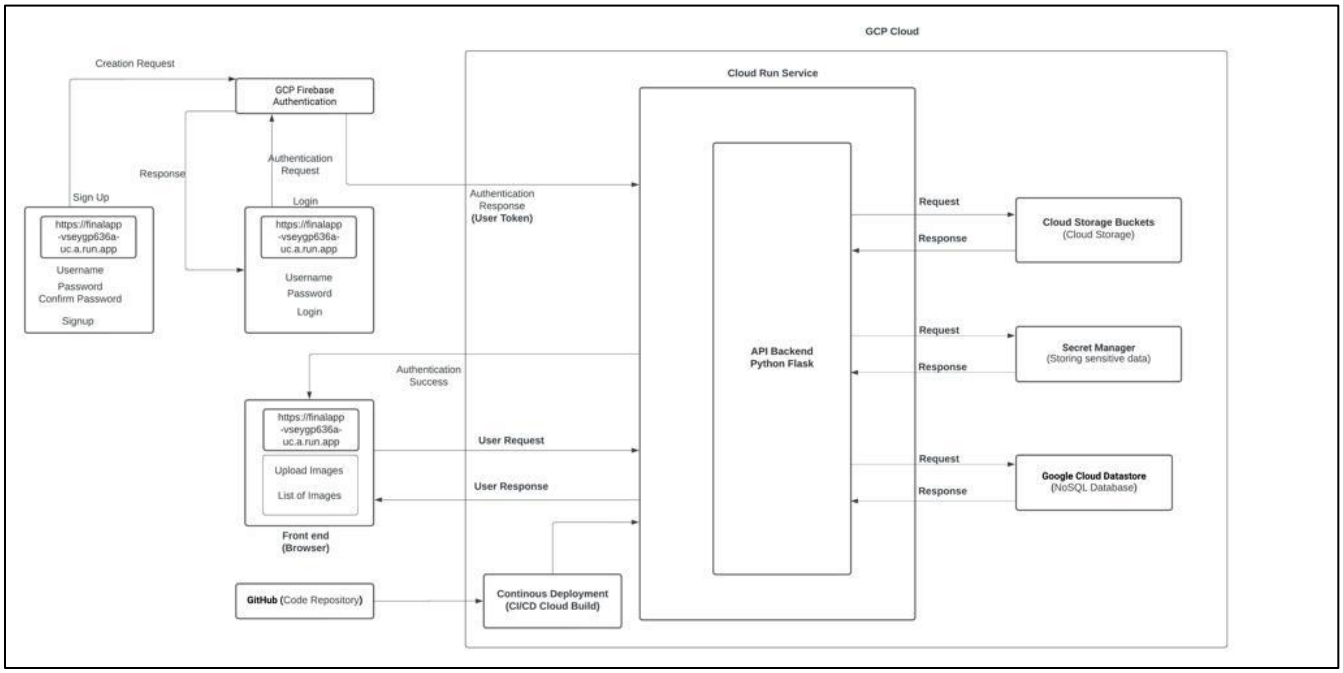
Authentication Sign-in method: Email/Password

Introduction:

The main aim of the final course project is to implement user login authentication for the application so that the users will be able to see only the images uploaded by them and implement the functionality of deleting the image and its respective metadata once the user clicks on the delete button on the specific drilled image. The functionality of the application that was developed in Project 2 remains as is with minor changes, and the continuous deployment feature implemented in Project 3 remains as is.

To design the application, we have used HTML, CSS, Python, and Flask, which is the web framework. To host the application over the internet "SERVERLESS" and make it accessible to everyone, we have used Cloud Run, which is linked to GitHub. We are storing the application's codebase on GitHub, which is the central repository enabling version control. We have implemented a continuous deployment pipeline to guarantee speed and flexibility in our development approach. It automatically deploys the latest version of our application to Cloud Run whenever changes are committed to the GitHub repository. We enabled a few GCP APIs and used a secret manager to store the AuthenticationCredentials.json and the firebase_config.json files. For the user login and signup authentication, we have used authenticate with Firebase using password-based accounts using JavaScript attached to our login and signup custom HTML templates, which will connect to Firebase, validate and authorize the login and signup, and route the user to their application home page. To store the images, we have used a storage bucket, and to store the metadata of the images, we have used the datastore database. The end-user who has the application URL can access it.

Architecture:



Project Planning and Implementation Overview:

The final project is an extension of Project 3, and based on the requirements, as there is no need to implement traffic splitting, we have removed it. We have added the Signup and Login Authentication flows, which are implemented using the GCP Firebase. We create the users in Firebase and authenticate them by creating the tokens.

Part 1 (Front End):

In the final project, we have added new flows like the User Login, User Signup, and Delete functionality, which deletes the image and its respective metadata. We also implemented sessions, logout, and change password functionalities. First, the user will create an account using signup and then login to the application using the login page. When the user logs in successfully, they will be routed to the home page. On the home page, we have the functionality to change the password, log out, and upload an image. When the user uploads an image, the image is shown in the list below under the view of saved photos. When the user clicks on one of them, they will be routed to the image page, where the image and its respective metadata are shown. Here, the user can go back to the home page by clicking the back button, or they can delete the image and the metadata using the delete button. Once the user clicks on the delete button, the image and the metadata are deleted, and the user is routed to the home page. When routing to any page, the user session is checked to see if they are in the session or not, and if the user logs out and tries to go to the home page using the URL, then the error message is flashed on the screen and routed back to the home page.

Part 2 (Back End):

The logic to create an account and login is implemented on the HTML side using the script tags to include the JavaScript code. When the user signs up for the application, the credentials are stored in Firebase as users, and an account is created. When the user logs in, the credentials are validated by the GCP Firebase Authentication, and if the credentials are valid, it generates a user authentication token, which is sent to the backend server, which is validated to see if the token is valid, unexpired, or not, and in that case, the application is redirected to the home page. When a change of password is done, the password of the user is pulled from the form response, updated with the new password, and the user is logged out. When storing the image in the bucket, for every user, an individual folder is created with the username so that when the user logs in, only the images uploaded by that user are displayed on the page. The metadata of the images is stored in the datastore, and an additional column named username is added. It will help fetch and store the metadata of the image when selected and uploaded by the user.

Part 3 (Cloud):

We have created the Firebase project, added our application as an imageapp, and tracked the user accounts. We have a secret manager that stores the credentials so that they can be fetched dynamically whenever needed by services. We have downloaded the firebase-admin-sdk key and uploaded it as a new secret with the name firebase-config secret in the secret manager. It already has the credentials.json file as a secret to access services like buckets and datastores, where we store all the uploaded images and metadata, respectively. As the credentials cannot be directly exposed in the code, we are accessing the secret manager service to fetch the credentials (key).

Also, we are using a continuous deployment setup with a cloud run, which also links to GitHub for the code repository. so that whenever there is a change in the code or commit, the changes are seen in the application.

Part 4 (GitHub):

After every change in the code, we push and commit the code to GitHub so that every developer is in sync with the code. After every commit, an auto-trigger of deployment is done so that the updated changes in the code are seen in the application.

Implementation:

We can divide the project implementation into the following parts:

a. Login/Signup Authentication & Configuration
b. Upload Image into bucket.
c. Extract and save image metadata into the datastore.
d. Fetch the image and metadata to visualize.
e. Delete the image and metadata using the delete button.
f. Change password and logout.
g. Continuous Deployment with CloudRun and GitHub

a. Login/Signup Authentication & Configuration:

1. Firebase Configuration:

- To access the application and upload and view the images, the user should have an account created. To sign up for the account, the user clicks on the signup, gives the required details, and creates an account.
- We have used the Firebase authentication functionality to create an account and login to the account with the credentials. To integrate and use Firebase, we first need to create a Firebase project and set the Firebase configuration.
- Go to the Firebase console and click on add project. As we already have a GCP project created, add it, follow the prompts, agree to the terms and conditions, and click continue. There is an option to add Google Analytics. Enable it, Next, you get a screen saying your project is ready. Now click continue.
- To configure, click on the settings icon next to the project overview and select the project settings option. We need to create a new web app, as our project is a web application. So click on the web icon to register our new app, give a name to our project app, and register it.
- Once registered, we can see the Firebase SDK, which has the configuration part. Copy it as we use it in our code.
- Now, we need to enable the authentication methods, In the Firebase console, navigate to the authentication section. Click on the 'Sign-in method' and enable

the provider “email/password”, as this is what we are using for the login and signup methods.

- We also need to have the firebase-SDK JSON key to access the firebase from the server side, i.e., python code. For this, in the Firebase console, click on the settings icon and select the project settings option. Go to the service accounts tab. Click the generate new private key button and then download the JSON file.
- We already have the secret manager API enabled and did the necessary settings. Now, upload the JSON key file to the secret manager by creating a new secret with the name “firebase-config” and uploading the key file. Now, the secret is created, and the Python file can access it to initialize Firebase admin with this secret file, the same way we used the Credentials.JSON file for securely accessing the GCP services.
- Now, the setup to use Firebase for the application authentication is done.

2. Signup:

- For performing the signup, we have designed a custom HTML template that has fields like email, password, and confirm password. Once the user hits the signup button, the JavaScript code that has the configuration to authenticate and create an account will be executed and the user account is created.
- In the HTML code we include the script tags, which contain the JavaScript code that is needed for the authentication. The following code is included in the <head> tag.

```
<script src="https://www.gstatic.com/firebasejs/9.0.2/firebase-app-compat.js"></script>
<script src="https://www.gstatic.com/firebasejs/9.0.2/firebase-auth-compat.js"></script>
  <script>
    // Your web app's Firebase configuration
    const firebaseConfig = {
        apiKey: "AIzaSyCjWGIF5xN9Vn4P-uNfTgbUvc769hR4AUY",
        authDomain: "group10-project1.firebaseapp.com",
        projectId: "group10-project1",
        storageBucket: "group10-project1.appspot.com",
        messagingSenderId: "213668284036",
        appId: "1:213668284036:web:a201b86f3d8687e85fe4d1",
        measurementId: "G-PTV2ZHD6T5"
  }
    // Initialize Firebase
    const app = firebase.initializeApp(firebaseConfig);
```

```
</script>
```

- The first 2 script tags are the Firebase libraries necessary for authentication. The firebaseConfig contains the necessary configuration details for using Firebase services in web applications. If not added in the HTML/JavaScript side, we can't initialize the Firebase authentication services.

Note: The information in firebaseConfig is meant to be publicly accessible. It doesn't include any secret keys or credentials that could compromise the security of the Firebase project.

- Once the config is included, we then use it to Initialize the Firebase.

```
<script>
   window.addEventListener('DOMContentLoaded', (event) => {
   document.getElementById('signup-form').addEventListener('submit', function(event) {
   event.preventDefault();
   var email = document.getElementsByName('username')[0].value;
   var password = document.getElementsByName('password')[0].value
   var confirmPassword = document.getElementsByName('confirmpassword')[0].value;
if(password === confirmPassword) {
            firebase.auth().createUserWithEmailAndPassword(email, password)
               .then((userCredential) => {
                  // Signed up successfully
                  // Optionally, make a POST request to your Flask `/signup` route
                  // Then redirect to login page
                  fetch('/signup', {
                     method: 'POST',
                     headers: {
                        'Content-Type': 'application/json',},
                     body: JSON.stringify({ email: email, password: password })
                  })
                  .then(response => response.json())
                  .then(data => {
                     console.log('Signup Success:', data);
                     window.location.href = '/login'; // Redirect to login page
                  })
```

```
                .catch((error) => {
                    console.error('Error:', error);
                });
            })
            .catch((error) => {
                var errorCode = error.code;
                var errorMessage = error.message;
                alert(errorMessage); // Display error message
            });
        } else { alert("Passwords do not match!"); }
    });
});
</script>
```

- The above JavaScript script code is designed for the user sign-up functionality in a web application. An event listener is attached to the sign-up form, tracking the form submission to handle the process manually. By clicking on the submit button, the script first prevents the default submission action, then collects the entered email, password, and confirm password values from the form fields.
- It checks if the entered password and confirm password match. If they don't match, an alert notifies the user of the mismatch. If the passwords match, the script uses Firebase's built-in createUserWithEmailAndPassword method to perform the user registration.
- If successful, it optionally sends the user data to a Flask backend server using a POST request and then directly redirects the user to the login page from the client side. If Firebase encounters an error like an invalid email or weak password, or an account that already exists with the given email, it displays the respective error message to the user through an alert. This process is a secure user sign-up experience, integrating front-end interactivity with Firebase authentication to create a user account.

3. **Login:**
    - For performing the login, we have designed a custom HTML template that has fields like email and password. Once the user hits the login button after entering the data, the JavaScript code that has the configuration to authenticate and login to the user's account will be executed, and the user will be logged in.
    - In the HTML code, we also include the script tags in the head, which contain the JavaScript code that is needed for the authentication, similar to the signup.
    - The logic for login is slightly different to the signup and is attached below:

```
<script>
window.addEventListener('DOMContentLoaded', (event) => {
   document.getElementById('login-form').addEventListener('submit', function(event) {
      event.preventDefault();
      var email = document.getElementsByName('username')[0].value;
      var password = document.getElementsByName('password')[0].value;
      firebase.auth().signInWithEmailAndPassword(email, password)
         .then((userCredential) => {
            userCredential.user.getIdToken().then(function(idToken) {
               // Send token to Flask backend in request headers
               fetch('/login', {
                  method: 'POST',
                  headers: {
                     'Content-Type': 'application/x-www-form-urlencoded',
                     'Authorization': 'Bearer ' + idToken
                  },
                  body: new URLSearchParams({
                     'username': email,
                     'password': password,
                     // Add other form fields if needed
                  })
               })
               .then(response => {
                  if (response.redirected) {
                     window.location.href = response.url;    }
               })
               .catch((error) => { console.error('Error:', error); });
               });
         })
         .catch((error) => {
            var errorCode = error.code;
            var errorMessage = error.message;
```

```
            alert(errorMessage); // Display error message
        });
    });
});
</script>
```

- The above JavaScript script code is designed for the user login functionality in a web application. An event listener to the login form, which tracks the form submission to handle the process programmatically is added. On clicking on the login button in the form, the script prevents the default form submission behavior and retrieves the entered email and password from the form fields.
- Now with the email and password, using Firebase's signInWithEmailAndPassword method, it attempts to authenticate the user with these credentials. If the authentication is successful, it obtains an ID token from the user's credentials and sends this token along with the email and password to the Flask backend server via a POST request.

```
import firebase_admin
from firebase_admin import credentials, auth
def access_secret_version(project_id, secret_id, version_id="latest"):
    client = secretmanager.SecretManagerServiceClient()
    name = f"projects/{project_id}/secrets/{secret_id}/versions/{version_id}"
    response = client.access_secret_version(request={"name": name})
    payload = response.payload.data.decode("UTF-8")
    return payload

#For firebase initialization fetching the credentials from secret manager
project_id2 = "group10-project1"
secret_id2 = "firebase-config"
firebaseconfig_json = access_secret_version(project_id2, secret_id2)
firebase_config = json.loads(firebaseconfig_json)
cred = credentials.Certificate(firebase_config)
firebase_admin.initialize_app(cred)
```

- We already installed and imported the firebase_admin module. We are getting the credentials of the firebase-admin-sdk which is uploaded in the secret manager by using the secret manager, and then we are initializing and authenticating the firebase-admin using the key.

```
@app.route('/login', methods=['GET','POST'])
def login():
    session.pop('user', None)
    session.pop('uid', None)
    if request.method == "POST":
        try:
            authorization_header = request.headers.get('Authorization', '')
            if 'Bearer ' in authorization_header:
                id_token = authorization_header.split('Bearer ')[1]
                verified_token = auth.verify_id_token(id_token)
                session['user'] = verified_token['email']
                session['uid'] = verified_token['uid']
                return redirect(url_for('home'))
            else:
                raise ValueError('Bearer token not found in Authorization header')
                flash("Issue in login. Login failed. Please try again. ",'error')
                return redirect(url_for("login"))
        except auth.ExpiredIdTokenError as e:
            flash(f"Token expired: {e} Please try again. ",'error')
            return redirect(url_for("login"))
        except auth.InvalidIdTokenError as e:
            flash(f"Invalid token: {e} Please try again. ",'error')
            return redirect(url_for("login"))
        except Exception as e:
            return jsonify({'status': 'error', 'message': str(e)}), 500
    else:
        return render_template("login1.html")
```

- This POST request includes the Authorization header. On the server side, under the flask route /login the POST Request is captured and processed. It first clears any existing user session data and then checks for the presence of a 'Bearer' token in the Authorization header. A Bearer Token is a cryptic string, usually generated by the server in response to a login request. The client must send this token in the HTTP Authorization header when making requests to a protected resource.

- From the Firebase-admin, we are importing **auth** and using it to verify the token. If a bearer token is found, it verifies the token using Firebase's "auth.verify_id_token" method. Upon successful verification, the route stores the user's email and unique ID in the session and redirects it to the home page, treating it as a success case. If the token is expired, invalid, or not found, appropriate error messages are flashed, and the user is redirected back to the login page. In case of any other exceptions, an error message is returned as a JSON response. This process is a secure user login experience, integrating client-side with server-side processing and authentication management to successfully login the user.

b. Upload an Image into the bucket:

- The uploaded images will be stored in the GCP storage bucket. To store the images, you first need to create a storage bucket. In the GCP console, search for the storage and click the buckets.
- Click on the Create option, choose the name of the bucket, and click Continue. The name should be unique. Leave the location type as the default multi-region. Set the storage class to the default class, which is the standard.
- The access control given to the bucket is uniform. So that all objects in the bucket will inherit the same access control settings by default. There is no need to protect the data additionally, select none in the protect object data and click the create button.
- Now, a bucket has been created. By default, public access to this bucket is prevented so that no one with the bucket link can access the data. So, the bucket is secured.
- Ensure that the service account you created can access this bucket. If not present, for the service account, click on the grant access option and give the "Storage Object Admin" role. This gives access to viewing, storing, and deleting the images in the bucket.
- **As the user's email will be unique, we are using the email address to create the folder for the user in the bucket.** Now, as the user is logged in successfully, if the user is a new user, no images have been uploaded by them already. So, no folder exists under the user's email though the user account is created. Also, on the home page, under the view of your saved photos, the list will be empty.
- Now, for the first time, when the user uploads an image, the new bucket will be created with the user's email, and the newly uploaded image will be saved in the folder. As the page reloads, we can see the image name under the "View your images" list.

```
client = storage.Client(credentials=credentials1)

foldername = session['user']

fileslist=list_files(foldername)

if fname in fileslist:
```

```
    flash(f'An Image with the same name {fname} already uploaded by the current user. Please rename the file ad try again.','error')

    return redirect(url_for("home"))

else:

imageuploadstatus = put_image_into_bucket(fname,imagefile,client,foldername)
```

- From the above lines, storage.client() is an instance of the Google Cloud Storage that was created to allow you to interact with GCP services. We are getting the user email address from the session and storing it in the variable foldername so that we use it to create a new folder or use the existing folder to save the image. We are calling the list_files function to fetch all the filenames that the user has already uploaded. Now, we are checking if the filename already exists in the uploaded files list. If yes, we are flashing an error message. Otherwise, the custom function put_image_into_bucket will be called and will let you store the image in the bucket, which takes the filename, image, instance, and foldername. The following is the function code.

```
bucket = client.get_bucket("group10project2")
image_object_name = f'{foldername}/{fname}'
```

- The above code will help you get a reference to the specific bucket. As we have the foldername parameter, we use it and store the image inside that folder in the bucket. If the folder exists the image is stored in the folder else, it creates a folder and saves the image in it.

```
blob = bucket.blob(image_object_name)

blob.upload_from_file(imagefile)
```

- We create a blob variable with the image and use it to upload the file into the bucket under the specific folder using the built-in function upload_from_file, which takes the image as a parameter.

c. Extract and save image metadata into the datastore:

- When the image is uploaded, we need to store the image metadata in the datastore database along with the image. We use the following code to communicate with the datastore database:

```
client1 = datastore.Client()

kinds = "ImageMetadata"

foldername = session['user']

status = put_metadata_into_datastore(imagefile,kinds,client1,fname,foldername)
```

- A client object for Google Cloud Datastore using the datastore.Client() constructor is created, and we have the "kind" name of our datastore already created as ImageMetadata. We pass the image, filename, kindname,

foldername, and object instance client1 as parameters to the function "put_metadata_into_datastore", which will extract the metadata from the image and push it into the datastore database.

```
data = extract_metadata(imagefile,fname)
```

- The following code is the "extract_metadata" function code, which will extract the image metadata.

```
  filename = fname
   image = Image.open(image)
   exif_data = image.getexif()
   metadata={}
   metadata['filename'] = filename
   if exif_data:
      for tag_id in exif_data:
            tag = TAGS.get(tag_id, tag_id)
            value = exif_data.get(tag_id)
            if isinstance(value, int) or isinstance(value,str) or isinstance(value, bool) or isinstance(value,float) or isinstance(value,bytes):
               metadata[tag]=value
            else:
               pass
   return metadata
```

- We first install the pillow module in Python and import it. We import the Image, ExifTags, and TAGS methods from the module. We use the getexif() function, which will let you get the image's exif data. We use the TAGS to get the metadata tag name and its value with the tag_id, save it into the dictionary, and return it. The returned metadata is passed into the put_metadata_into_datastore function.

```
new_entity = datastore.Entity(client1.key(kinds))
for key1,value1 in data.items():
   new_entity[key1]=value1
new_entity['username']=foldername
times=datetime.now().strftime('%Y-%m-%d %H:%M:%S')
new_entity['uploaded_date']=times
```

- Once we have the metadata, we create the new record instance named new_entity in the datastore under the specific kind. We save the dictionary data items into the new_entity dictionary and also add a couple more key-value pairs like **username** and **uploaded_time**. The username key will let you uniquely identify the user-uploaded image's metadata in the list of existing records, which acts as a filter when fetching the image list and image metadata.

  client1.put(new_entity)

- Finally, we push the data into the datastore using the GCP Datastore method "put" which is shown above.
- To show the list of files uploaded, under the view your saved photos on the home page, we call the list_files() function, which is below. We are querying on the datastore with the username column, fetching the results list, extracting the image names, returning the list to the home page, and displaying them as anchor tags on the home page. The following is the list_files() function code.

  kinds = "ImageMetadata"

  client1 = datastore.Client(credentials=credentials1)

  query = client1.query(kind=kinds)

  query.add_filter('username', '=', usernames) #Queries all the file names which are stored by that user.

  results = list(query.fetch())

  filenames = [entity['filename'] for entity in results if 'filename' in entity]

  return filenames

d. Fetch the image and its metadata to visualize:

- when the user clicks on a particular image to visualize it, the control comes to the route **@app.route("/files/<fname>").** We have the image name as fname. Here, we call the functions get_image_from_bucket(fname,client,foldername) and get_metadata_from_datastore(kinds,fname,client1,foldername). First, we get the foldername from the session['user']. The following is the code:

  foldername = session['user']

  kinds = "ImageMetadata"

  client = storage.Client(credentials=credentials1)

  client1 = datastore.Client(credentials=credentials1)

  image_data , mimetype = get_image_from_bucket(fname,client,foldername)

  image_bytes = image_data.read()

  # Encode the image data as base64

  image_base64 = base64.b64encode(image_bytes).decode('utf-8')

- We initialized the datastore and bucket and called the function get_image_from_bucket(). The following are the function details:

```
bucket = client.get_bucket("group10project2")
image_object_name = f'{foldername}/{fname}'
blob = bucket.blob(image_object_name)
if not blob.exists():
    print("Unable to find the image with the given filename")
image_bytes = BytesIO()
blob.download_to_file(image_bytes)
image_bytes.seek(0)
mimetype="image/jpeg"
return (image_bytes,mimetype)
```

- we get the reference to the specific bucket. As we have the foldername parameter, we use it to fetch the image inside that user's folder in the bucket. We create a blob variable with the image and check if the blob exists or not. As we work with the image, we create an empty BytesIO object, which will help us download the blob's content and write it into the BytesIO object image_bytes. This will read the binary data from the blob and store it in the BytesIO buffer. Here, the cursor will be in the last position. Hence, seek (0) is being used to put the cursor at the start of the data so that we can read and process it from the start. So, finally, we will have the image in binary form and return the image and the mimetype. We then read the image data and represent it in the base64 format in the HTML and then we call the metadata function.

```
feteched_data  = get_metadata_from_datastore(kinds,fname,client1,foldername)
```

- The following is the metadata_from_datastore function:

```
query = client1.query(kind=kinds)
query.add_filter('filename', '=', fname)
query.add_filter('username', '=', foldername)
results = list(query.fetch())  # Execute the query and retrieve the results
metadata1={}
if results:
    metadata1 = {key: value for key, value in results[0].items()}
else:
    print("No results fetched.")
return metadata1
```

- To explain the above code, we get the kind name, datastore object instance, image name, and the foldername which is the username, and pass them to the get_metadata_from_datastore method, which will create the query syntax with the kindname, and then we add two filters (additional condition) with the filename and the username, which we have them as a parameter of the function. This will fetch the required metadata of the image from the datastore, and we will pass this dictionary output to the webpage so that this data and the image are displayed.

e. Delete the image and its metadata using the delete button:

- In the visualization page, which is loaded when the user clicks on one of the available uploaded images, we have provided the back and the delete button. Once the user clicks on the back button, the user will be routed to the home page.
- When the user clicks on the delete button, immediately, the control will go to the **@app.route("/delete", methods=['GET', 'POST']).** The code is executed, and the image and its respective metadata are deleted, then, the page is routed to the home page, where a flash message is displayed on the top saying the image and its respective metadata are deleted successfully.
- In the HTML side, we have added a hidden field **<input type="hidden" name="fname" value="{{ fname }}">,** where this field can be accessed in the code when the form is submitted(when the button delete is clicked).
- The code of the delete functionality is attached below:

```
if 'user' in session:
    foldername = session['user']
    fname = request.form['fname']
    client = storage.Client(credentials=credentials1)
    client1 = datastore.Client(credentials=credentials1)
    bucket = client.get_bucket("group10project2")
    image_object_name = f'{foldername}/{fname}'
    blob = bucket.blob(image_object_name)
    kinds = "ImageMetadata"
    query = client1.query(kind=kinds)
    query.add_filter('filename', '=', fname)
    query.add_filter('username', '=', foldername)
    try:
      blob.delete()
      entities = list(query.fetch())
      for entity in entities:
        client1.delete(entity.key)
      flash(f'Successfully deleted the image and the metadata of {fname}','error')
      return redirect(url_for("home"))
    except:
      flash(f'Unable to find the image or the metadata to delete. Please try again later.','error')
      return redirect(url_for("home"))
```

```
else:
    flash('Please login before you proceed further. ', 'error')
    return redirect(url_for('login'))
```

- We first fetch the user email address, which is stored in the session. This will be helpful for finding the foldername and the username field in the bucket and the datastore database, respectively. We also retrieve the file name from the form response.
- Using these, it constructs the object name for the image in the Cloud Storage bucket ('group10project2') and the filters for querying the Datastore with the fname and the username field(variable foldername). The script then attempts to delete the image blob from the bucket and its corresponding metadata entities from the Datastore. If successful, it flashes a success message indicating that the deletion of the image and metadata is successful. If the script encounters any issues during this process (such as the image or metadata not being found), it flashes an error message and finally is routed to the home page.

f. Change password and Logout:

- When the user is logged in, On the home page, we have provided the functionality to change the password and logout. When the user clicks on the logout button, the user will be logged out of the application. The sessions will be cleared, and the user will be routed to the login page. A flash message is displayed saying “successfully logged out of the user”. The following is the code for logout.

```
@app.route('/logout', methods=['GET', 'POST'])
def logout():
   session.pop('user', None)
   session.pop('uid', None)
   flash("Successfully logged out the user. ",'error')
   return redirect(url_for('login'))
```

- Now, when the user clicks on the change password button, the user will be routed into the **/change_password** route, where a custom change password template is displayed. Once the new password and confirm password are entered, and the user clicks the update password button, the control will go to the server side, and the following code is executed.
- It first checks if a user is currently logged in by verifying the presence of a 'user' in the session. If the user is logged in and the request method is POST (indicating that the password change form has been submitted), the script retrieves the user's email and unique identifier (UID) from the session, which is already created. It then fetches the new password and the confirm password from the form.

```
if 'user' in session:
    if request.method == 'POST':
       email = session['user']
       uid = session['uid']
       #old_password = request.form.get('old_password')
       new_password = request.form.get('new_password')
```

```
            confirm_new_password = request.form.get('confirm_new_password')
            if new_password == confirm_new_password:
                try:
                    auth.update_user(uid, password=new_password)
                    flash('Password changed successfully!', 'success')
                    return redirect(url_for('login'))
                except ValueError as e:
                    flash(f'Error changing password: {e}', 'error')
            else:
                flash('New and Confirm Password Mismatch. Try again!', 'error')
                return redirect(url_for('change_password'))
        else:# Handle other HTTP methods or invalid requests
            return render_template('change_password.html')
    else:
        flash('Please login before you proceed further. ', 'error')
        return redirect(url_for('login'))
```

- If the new password and the confirm password do not match, an error message is flashed, and the page is re-routed to the change_password route. If they match, the script attempts to update the user's password using Firebase's built-in **auth.update_user** method, passing the UID and the new password. If the password change is successful, it logs out the user, clears the sessions, flashes a success message, and redirects the user to the login page. Now, the user should login with the new password to access the application. If there's an error during the update (e.g., invalid password format), it captures and displays the error message. The user if unexpectedly clicked the change password button can go back to the home page by clicking the home button.

g. Continuous Deployment with CloudRun and GitHub:

Likewise, in Project 3, we deployed our code using continuous deployment with CloudRun, which is connected to the GitHub repository.

1. GitHub:

- As continuous deployment expects a code repository to track the changes and deploy the code continuously, with the available repository options, we have selected GitHub.
- In your GitHub account, select Create a New Repository option and provide the repository name as "finalproject_group10" and description if needed, leave the repository as a public repository, and click Create repository.
- Copy the HTTPS URL to add the GitHub repository as a remote for the connection.
- Go to the folder where your code files exist and type **git init** which will initialize the git repository.
- Type **git remote add origin HTTPS URL** (which you copied), which will provide the remote repository location for the connection so that you can push the files into this repository.
- Type **git add .** which will add all the uncommitted files to git.

- Type **git commit -m "COMMIT MESSAGE"** which will commit the changes that we have made, and the message will help you gain an understanding of what changes are in this push package.
- Type **git push -u origin main**, which will push all the files into the git repository. Now, in GitHub, verify if the files were pushed successfully or not.

2. **Continuous Deployment with CloudRun:**
   - To implement the continuous deployment feature, we must enable a few Cloud Run APIs like the Cloud Run API, the Container Registry APIs, and the Cloud Build API. We can search for them in the GCP console search and enable them.
   - This next step is crucial for allowing Cloud Run to access secrets during the build and deployment process.
   - Navigate to IAM & admin and select IAM. We can see all the permissions for the project's service accounts. Now, look for the service account that will be used by CloudRun. Unless specified for the cloud run to use any other specific service account, it will use the service account that is in this format, XXXXXX-compute@developer.gserviceaccount.com, for building.
   - Now click on Edit Pencil, add the new role "**Secret Manager Secret Accessor**" and save it. Now, the role has been added to the service account. This role will allow the code to access the secret manager and get the credentials dynamically. We are not showing the key in the code as it will compromise the user account. Instead, we are connecting to the secret manager and fetching the key. If the access is not given, the code fails when trying to access the key to build the credentials to access the bucket and datastore services further.
   - Now, Search for Cloud Run and click on the Create service. Select **Continuously deploy new revisions from a source repository** and click on **Setup Cloud Build**.
   - On your right, you can see a window. Select the repository provider. In our case, it is GitHub. Now follow the prompts to authenticate with GitHub. You might be redirected to GitHub, logged in to your account, and asked to install the Google Cloud Build app on your GitHub account.
   - Now, you select the repository **(finalproject_group10)**that you created, which contains the code and required files. Now click next.
   - In the second step, i.e., build configuration, select the branch as **^main$ or put .***, which refers to any branch. For our project, we select **^main$**, which is the default.
   - As we use Dockerfile, select the build type as Dockerfile. The source location will have a default value. Leave it as is, as your Dockerfile is in the repository root, and click save.
   - Enter a name for the service and select the desired region for deployment. Choose "Allow unauthenticated invocations" for public access. Leave other settings at their defaults, review them once, and click Create to start the build and deployment process.

- The build and deployment will take a few minutes. You can observe the progress in the Cloud Run interface. Once the build is successful, you can access the application via the provided URL. The application should display the latest changes.

Security:

- In terms of security, as we are deploying the application using CloudRun, the application is secured, and the URL has “https”.
- We are using Firebase authentication, so only the user with legit credentials can access the application. When successfully logged in, the user is routed to the home page and can upload or access the images they have uploaded.
- We are using the sessions so that when routing to every page, excluding the login and signup, the user session is checked, and only the content of the page is displayed. If the user session is not present, then the user is routed to the login page, and we flash an error message asking the user to login to proceed further, making the application secure.
- The usage of the secret manager to store the keys as secrets increases the level of security for the application, as these sensitive keys should not be stored in public repositories or in the code, which can easily give access to intruders and misuse them.
- Robust security is ensured by strictly managing access to GCP services like Cloud Run, storage buckets, Datastore, Firebase, and Secret Manager. This is achieved through carefully configured Identity and Access Management (IAM) roles. We've restricted access to these services exclusively to our core project developers. By doing so, we prevent end users of the application from directly accessing these GCP services, thereby maintaining a high level of security.

PROS & CONS:

Pros:

1. Anyone can easily access the application with one click on the URL.
2. As we are using Firebase for the user login and signup authentication, The account creation and login validation are taken care of by Firebase, which provides extreme security for the user account so that only the valid user with valid credentials can log in to the account.
3. As Firebase authentication is built on Google's secure infrastructure, it provides robust security measures like that protect other Google services. It can also scale itself to accommodate a huge user count, is also of low cost, and provides real-time offline support.
4. We have created the user login to achieve security for the user-uploaded images. So, only the images that are uploaded by that user and their metadata are visible under the user account, which is achieved by the individual user folders inside the storage bucket.
5. Intruders, if they find the route to the home page, cannot go to the home page without logging into the application due to the sessions and security features.
6. Easy view, download, and upload options for the users, which are user-friendly. You Can view only the user-uploaded images with their names on the screen itself.

7. We have protected the application where our URL has HTTPS, which is achieved by using Cloud Run to run the application.
8. As each user has their own folder inside the bucket, fetching the specific image for the user is easy and takes less time, and if multiple users upload the same image, as they are saved in their respective user folders, there will be no overlap or clash with the buckets.
9. When utilizing VMs, in the case of autoscaling, there is a possibility of encountering storage challenges when retrieving user images, particularly when the images are stored in VM memory. We have solved this concern by implementing cloud storage buckets.
10. As we are using Cloud Run, in the case of heavy traffic, the application will not crash because Cloud Run will auto-scale and use load balancing to split the traffic as it creates multiple container instances.
11. As the code is being containerized, it will be very easy for cloud/DevOps engineers to deploy the container into any environment with relative ease.
12. Continuous Deployment automates the release of code changes to production, significantly reducing manual efforts and accelerating the deployment process. This ensures a faster and more reliable way to update applications, providing immediate feedback to developers and enhancing overall efficiency. The automated process is more reliable as it minimizes human error and ensures consistent application updates.
13. In CI/CD environments, monitoring changes and troubleshooting during rollouts becomes more manageable, as issues can usually be traced back to one or two recent changes, provided the right monitoring tools are in use. Rollbacks are simplified and less effort-intensive due to the small scale of changes, but this depends on having efficient orchestration tools in place.
14. By leveraging Continuous Deployment (CD), we continuously release new features and bug fixes. This improves user experience by promptly addressing issues, introducing enhancements, and keeping pace with the rapidly evolving market demands.
15. Continuous Deployment ensures shorter review times as it typically involves smaller, incremental changes that are easier to test and validate. This streamlined process facilitates rapid feedback, significantly contributing to higher-quality output and more robust code.
16. The administrator only has access to disable the user account if it is found compromised or reported by a legitimate user. Also, the administrator can generate the reset password link if a user requests it. Here, the reset password list is sent to the user’s registered email address.

Cons:

1. As our project has a simple requirement, there is no need for a person to monitor the application continuously, but for larger projects, the need for constant monitoring and immediate support escalates very high. This is very much needed for the operational teams, who must be prepared always to respond.
2. As our project has a small requirement, it works without having a separate DevOps team, but in larger companies and projects, effective implementation of Continuous Deployment (CD) demands a strong DevOps culture. Organizations without strong DevOps practices may struggle to harness CD's benefits fully.
3. As the login authentication depends on Firebase and if the Firebase is suddenly down, the user will face login-related issues as the login validation is done on the Firebase side.

4. As the user count and the number of uploads increases, the data storage costs increase rapidly, especially when there is huge traffic.
5. The application depends on multiple services; the overall user experience is impacted if either is down.

Application Instructions:

1. Please type in the URL in your local browser and click enter or click on the URL from the PDF.
2. You will be routed to the application when you click the URL, and you will be able to see the login and sign-up buttons on the initial page.

Signup:

3. If you don't have an account, click the sign-up button on the initial page. You will be routed to the sign-up page.
4. To create an account, enter the username, password, and confirm password fields, and click the sign-up button.
5. If the sign-up process is successful, you will be routed to the login page.
6. If the password length is short and less than six characters, you will be alerted with an error message saying, "Password should be at least 6 characters (auth/weak password)".
7. If there is any password mismatch, you will be alerted by an error message saying, "Passwords do not match!". The user needs to click ok and re-enter the passwords.

Login:

8. If an account already exists, you can log in. On the initial page, click on the login button. You will be routed to the login page.
9. Now, enter the username and password on the login page and click the login button.
10. If the credentials are correct, you will be routed to your user-specific account home page, where you can upload the images. If wrong, then you will be flashed an error message.
11. On the screen, you can see the list of existing images that have already been uploaded to the server, if any.

Visualizing the uploaded Images:

12. To view the previously uploaded image by the user, click on the image you like. On click, you will be routed to the My Image page, which displays the image along with the metadata of that image, the back, the delete buttons, and the image name at the top. You can click the back button to go back to the home page. You can click the delete button to delete the image and its metadata.

Upload a new image:

13. If you want to upload a new image or picture, click the choose file button. Then, a window will be opened, and you can go to the path where the desired image is located, select it, and click Open.

14. Immediately, you can see that the image has been pulled up from your device and is ready to upload.
15. Click the upload button to upload the image into your specific folder in the GCP storage bucket through the application.
16. By clicking on the upload button, the page will be refreshed, and immediately, we can see the newly uploaded image in the view your saved photos list.
17. If the user clicks on the upload button without selecting a file, an error message will be flashed on the screen asking the user to choose an image and click the upload option.
18. If the user uploads an image that was already uploaded and has the same name, then an error message is flashed on the screen, asking the user to rename the file and try again.

Delete an Image:

19. Once the user clicks on the specific image from the list, you will be routed to the My Image page, which displays the image along with the metadata of that image. On the top left, you have the buttons back and delete.
20. To delete an image, click on the delete button. Immediately, the deletion operation is performed, and the image and its respective metadata on that page are deleted from the cloud side. You will be routed to the home page, where you can see a flash message saying, "Successfully deleted the image and the metadata."
21. Parallelly, on the home page, under the saved list of photos, the deleted image name will not be shown anymore.

Download an existing image:

22. If you want to download the existing uploaded images, click on the respective image name. You will be routed to the My Image HTML page, which displays the image along with the metadata of that image. Right-click on the image and select Save the image as an option. Once the destination saving path is set, the image will be downloaded. You can route to the home page to continue accessing the application.

Change Password:

23. The change password button is also present on the home page, along with the home and logout buttons.
24. To change the password of your user account, click on the change password button on the home page. The user will be routed to the change password page, where the user must enter the new password in the new password and confirm password fields.
25. Once the user enters the new and confirm password and clicks the update password button, If the new and confirm passwords match, then the password is updated, and it will log out the user from the current session and re-route to the login page.
26. If there is a mismatch between the new password and the confirm password entered by the user, it will flash an error message saying, "New and Confirm Password Mismatch. Try again!" and route the user to the change password page again.

27. If the user unexpectedly clicks the change password button on the home page and is routed to the change password page, the user can go back to the home page by clicking the home button on the change password page.

Logout:

28. The logout button is also present on the home page, along with the home and change password buttons.
29. Once the user clicks the logout button, the current user session is cleared, and the user will be successfully logged out of the application. The user will be routed to the login page, where they can see a flash message saying, "Successfully logged out the user."
30. If you want to log back into the account, log in with your proper credentials, and you will be logged into your specific account.

Lessons learned:

In addition to designing a simple application from scratch in Project 1, building the container, and deploying the application serverless using Cloud Run in Project 2, splitting the traffic, and setting up continuous deployment in Project 3, the final course project has taught us that user authentication can be implemented in multiple ways using the Firebase authentication features. Some of them are “Authenticate with Firebase using Password-Based Accounts using JavaScript and custom login/signup templates. Another way is by logging in with the Google feature, which completely depends on the Google login process, and another way of implementation is to use Pyrebase. We have tried to use the Pyrebase user authentication feature, but we faced multiple issues due to the older Pyrebase version, which is incompatible with GCP. There is some version-related configuration issue that expects a specific version of the request’s module, which is not supported by the Google Cloud Python module, which created module-related issues, so we moved to using the custom-designed login and signup templates that authenticate with Firebase using password-based accounts using JavaScript. As the login process is to create a user-specific experience, we have made some changes to the bucket so that for each user, there is a separate folder inside the storage bucket, which ensures that data privacy is achieved. We also implemented the delete button functionality so that the user could delete the images they uploaded. We have used the same continuous deployment functionality so that when the changes are pushed into GitHub, the deployment is triggered, and the changes are seen in the application. We have encountered many errors and referred to the documentation and multiple resources to solve them. Every lesson helped us to know more about the cloud features and enhanced our problem-solving skills.