Foundations of Cloud Computing

CEN 5086-001

GROUP10 - PROJECT 2 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

Application URL (External IP Address): View Your Photos (imageapp-vseygp636a-uc.a.run.app)

GitHub Repository Link for the code files (Public): SubashGupta/Group10-Project1 (github.com)

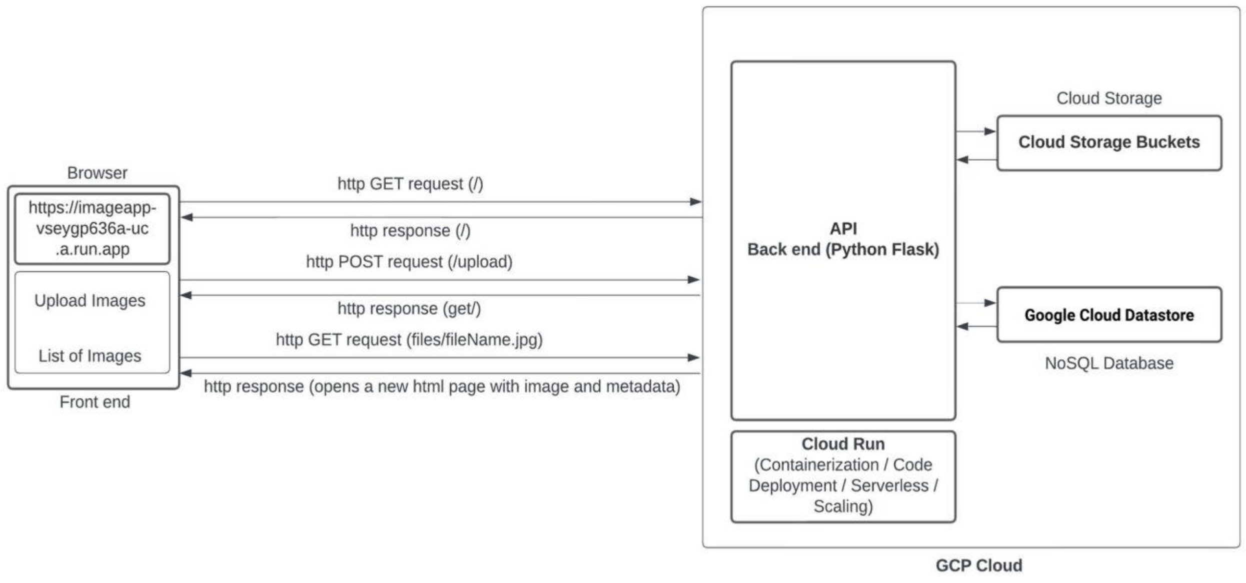
GCP Project link: Welcome – Group10-Project1 – Google Cloud console

Introduction:

The main aim of Project 2 is to slightly modify the existing application such that on clicking on the image, instead of opening the image directly, we show the image on a new HTML page along with some image information (metadata) and completely host it serverless using the Cloud Run functionality of the GCP platform. The existing application behavior of allowing the user to upload the image, view the list of uploaded images, visualize by clicking the image, and download it 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 and Docker. 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:


Browser
https://imageapp-vseygp636a-uc.a.run.app
Upload Images
List of Images
Front end
http GET request (/)
http response (/)
http POST request (/upload)
http response (get/)
http GET request (files/fileName.jpg)
http response (opens a new html page with image and metadata)
API
Back end (Python Flask)
Cloud Storage
Cloud Storage Buckets
Google Cloud Datastore
NoSQL Database
Cloud Run
(Containerization / Code Deployment / Serverless / Scaling)
GCP Cloud

Project Planning and Implementation Overview:

Project 2 is an extension of Project 1, and based on the requirements to make it serverless, we have used new services provided by the Google Cloud Provider, such as cloud storage buckets, Google Cloud Datastore, and Cloud Run. We created the architecture of our web application, which has three main parts: the front end, the back end, and cloud deployment, based on the requirements.

Part 1 (Front End):

In addition to Project 1, we have made some modifications to the front-end UI part. In Project 2, when the user uploads the image and clicks on the image to view it, instead of showing the image in a new tab, we show the image as well as the metadata of the image on an HTML page.

Part 2 (Back End):

In project 2, we have made some major changes when compared with project 1. Here, the code for storing images in the server's memory is replaced with cloud storage for image storage. We also made some changes to the existing APIs created in Project 1, and the APIs are:

1. '/upload' POST API: This API is designed for file uploads. This endpoint processes the incoming files and stores the image in Cloud Storage Buckets, stores the metadata that is extracted from the image into the Cloud Datastore, which is a NOSQL database, and then redirects to the home page, which displays the updated image list fetched from the bucket, which includes the recently uploaded image. It seamlessly integrates with the application's upload functionality.

2. '/files/{FileName.jpg}' GET API: This endpoint serves as the gateway for retrieving the specific images from the cloud storage bucket and their respective metadata from the cloud datastore. By providing the desired filename of the image as a parameter in the URL, users can access and view the requested image.

Part 3 (Cloud):

We have used the cloud for the new services like cloud storage, Google Cloud Datastore, and Cloud Run that have been implemented in this project. Cloud storage is used for storing the images when they are uploaded. Google Cloud Datastore is used for storing the metadata that is extracted from the image, and it provides scalability. We also used Cloud Run, which streamlines the deployment process by containerizing the Flask application, facilitating automatic updates, and enabling serverless execution. It also supports load balancing and auto-scaling to manage varying workloads efficiently when there is heavy traffic.

Implementation:

We can divide the project implementation into the following parts:

a. Service account creation
b. The bucket creation.
c. The datastore creation
d. Code-level modifications
e. Deploying the code serverless using Cloud Run and Dockerfile

a. Service account creation

1. Firstly, we need to create a service account. A service account is a special account that helps the user and services like cloud storage access GCP resources securely. They can be used for authentication and authorization in various GCP services.
2. To create the service account, you need to log in to the GCP console, and in the search, type "Service Account." Or in the products, under IAM & Admin, you can find the service accounts option. Click it.

3. On the top, you have the "create a new service account" option. On clicking on it, we need to provide the service account's name and ensure it is unique. A service account ID is parallelly created based on the name provided.
4. Click Create and Continue, and then give the account access to the project. We need to select the IAM role here. "Storage Object Admin" is the appropriate role as we need to access the bucket to push the images, fetch the images, and view the images. This admin role has all the above-mentioned responsibilities. Hence, this will be the best fit.
5. We need to give the user access to the service account. Who should be the user, and who is the admin? In our case, the user and the admin are given to the same person, i.e., the account holder.
6. On clicking done, the user account is successfully created. Now, on the right, click on the action 3 dots, select manage keys, click on add key, and select create a new key. Select the key in the JSON format and click Create. A key for authentication is created and is automatically downloaded. If you have an existing key, you can use it.
7. A service account has been successfully created. We can now use this service account to interact with the services of GCP, which has been granted permission.

b. The bucket creation:

1. To create the storage bucket to store the images, you need to log in to the GCP console and, in the search, type "Buckets." Or in the products, under cloud storage, you can find the buckets option. Click it.
2. Click on the Create option at the top of the page. 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.
3. 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.
4. There is no need to protect the data; additionally, select none in the protect object data and click the create button.
5. 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.
6. Ensure that the service account you created can access this bucket. If not present, click on the grant access option and give the "Storage Object Admin" role.

c. The datastore creation:

1. To create the datastore database to store the image's metadata, you need to log in to the GCP console, and in the search, type "datastore." Or in the products, you have the datastore directly. Click on it.
2. Click on the Create database option. You will be shown two options: either datastore mode or native mode. For our simple application, we have selected the datastore mode, which is a simple and economical storage model.

3. You need to select the region to store the data. For our application, I have selected nam5, which is the United States region.
4. You can create an entity (database record). Click on Create Entity. By default, the name of the database is default, and you can change it or leave it like that. The "kind" should be mentioned. We have selected "ImageMetadata" as our kind. Kind is similar to the table in the RDBMS. We have the numeric id as the key identifier, which is the default. Leave it like that. Click on the Create button. We can use the kind to filter related data.

d. Code level modifications:

1. In our previous code, we made some changes to make it run serverless.
2. Initially, the code to save the images in local memory is removed.

   os.environ['GOOGLE_APPLICATION_CREDENTIALS'] = "AuthenticationCredentials.json"

3. This line will set the environment variable that specifies the location of the JSON file that has the Google Cloud service account credentials. We need this line when we connect to services like databases and buckets. By using the environment variables, there is no need to specify the path explicitly to the credentials file to access the GCP services.
4. We use the following code block to communicate with the storage bucket:

   client = storage.Client()
   imageuploadstatus = put_image_into_bucket(fname, imagefile, client)
   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.
   The custom function put_image_into_bucket will let you store the image in the bucket, which takes the filename, image, and instance.

   bucket = client.get_bucket("group10project2")
   The above code will help you get a reference to the specific bucket.
   As we have a folder inside the bucket to store the images, we will use the **image_object_name = f'images/{fname}'** line.

   We create a blob variable with the image and use it to upload the file into the bucket using the built-in function upload_from_file, which takes the image as a parameter.
   blob = bucket.blob(image_object_name)
   blob.upload_from_file(imagefile)

5. we use the following code to retrieve the image from the bucket:

   image_bytes = BytesIO()
   blob.download_to_file(image_bytes)

```
image_bytes.seek(0)
mimetype="image/jpeg"
return (image_bytes, mimetype)
```

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.

6. We use the following code to communicate with the datastore database:

```
client1 = datastore.Client()
kinds = "ImageMetadata"
status = put_metadata_into_datastore(imagefile, kinds, client1,  fname)
```

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, 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.

```
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
```

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 to the main function.

Once we have the metadata, we create the new record instance, the new_entity in the datastore, under the specific kind. We save the dictionary data items into the new_entity dictionary and push the data into the datastore using the GCP Datastore method "put" which is shown below:

```
new_entity = datastore.Entity(client1.key(kinds))

for key1,value1 in data.items():

   new_entity[key1]=value1

client1.put(new_entity)
```

7. To fetch the metadata of a particular image we use the following code:

```
def get_metadata_from_datastore (kinds, fname, client1):
   query = client1.query(kind=kinds)
   query.add_filter('filename', '=', fname)
   results = list(query.fetch())  # Execute the query and retrieve the results
   metadata1 = {}
   if results:
      # Convert the results to a dictionary for JSON response
      metadata1 = {key: value for key, value in results[0].items()}
   else:
      print("No results fetched.")
```

To explain the above code, we get the kind name, datastore object instance, and image name and pass them to the get_metadata_from_datastore method, which will create the query syntax with the kindname, and then we add a filter (additional condition) with the filename, which we have as a parameter input to 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. Deploying the code serverless using Cloud Run and Dockerfile

1. We must create a Dockerfile to put the code inside a container for deploying the code. You can save the Docker file in the.txt format.
2. You should install the Docker desktop application to build and push the file.
3. Once the Docker application is installed, we need to create the Dockerfile and the requirements.txt file.

4. In the requirements.txt file, we need to add all the dependencies of the project code file with Python and make sure no module we have imported for our code is missing.
5. We must create the Dockerfile once the requirements file is ready. In the Docker file, we must write all the parameters needed to build it successfully, as below:

   Dockerfile:
   ```
   #Use the official Python 3.10 image based on Alpine linux
   FROM python:3.10-alpine

   # Set the working directory which is specific to the cloudrun repository
   WORKDIR /app

   # copy the requirements file which has the dependencies of the python code.
   COPY requirements.txt .

   # Install the needed packages which are specified in requirements.txt
   RUN pip install --trusted-host pypi.python.org -r requirements.txt

   #copy all the dependency files in our working directory to the Cloudrun repository
   working directory /app
   #Python application code is being copied.
   COPY Project.py .

   #Templates we used for the UI are being copied.
   COPY ./templates ./templates

   #Service account credentials which we are using to communicate is being copied.
   COPY AuthenticationCredentials.json .

   #PORT EXPOSURE for running the application
   EXPOSE 8080

   #The command to run the Python application.
   CMD ["python", "Project.py"]
   ```

6. Now, we need to build and push the docker file into the Google Cloud Repository, and then we deploy it into the cloud run using the gcloud.
   We need to run the following commands in the current directory of your local machine where the dockerfile and the code files are present.

a. **docker build -f Dockerfile.txt -t imageapp .**

   This line states that we will be building a docker image named imageapp based on the instructions mentioned in the Dockerfile.txt also mentions that the current directory is the build context.

b. **docker tag imageapp gcr.io/group10-project1/imageapp:latest**

   The command creates a new tag for a Docker image called "imageapp." This tag is given a repository reference, specifying the Google container registry and the project name. The "latest" tag indicates that this is the latest version of the image. Tagging Docker images makes managing and deploying containers in cloud environments easier.

c. **gcloud auth configure-docker**

   This command configures Docker to authenticate using your Google Cloud credentials, allowing you to push images to GCR and ensuring that you have the necessary permissions to access the registry.

d. **docker push gcr.io/group10-project1/imageapp:latest**

   This command uploads the tagged Docker image to the specified location in the Google Container Registry (GCR). In this case, it's pushing the image with the "latest" tag, indicating the latest version. This step is essential for making the container image available in the specified container registry, which can be accessed and deployed in cloud environments.

e. **gcloud run deploy --image gcr.io/group10-project1/imageapp:latest --platform managed --region us-central1 --allow-unauthenticated**

   The containerized application is being deployed into Google Cloud run. Here, our image in the Google Cloud registry is being deployed. Here, platform managed means we want Google to take care of all the server management, traffic, scaling, and any other tasks that define the meaning of serverless. We are selecting the region as us-central1 and allowing unauthenticated means anyone with the URL can access the application without providing any authentication details or credentials.

PROS & CONS:

Pros:

1. Anyone can easily access the application with one click on the URL.
2. Easy view, download, and upload options for the users, which are user-friendly.
3. You Can view the uploaded images with their names on the screen itself.
4. We have protected the application where our URL has HTTPS, which is achieved by using Cloud Run to run the application.
5. 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.
6. 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.
7. 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.

Cons:

1. There is no user-specific login for the application. In the case of millions of users, all the user's images will be shown in a single view, which is clumsy and not an effective application.
2. Without user authentication, anyone with the URL can access the application and its data and misuse it.
3. Anyone with the image URL link can view the image and its metadata directly, which is not the best practice.

4. As the uploaded files of all users are maintained in a single bucket, it is very hard to distinguish files based on the user.
5. As it is a single bucket, if millions of users have their files in the bucket, the time taken to fetch the single file from this large bucket is longer. To improve it, if a bucket is created for each user, then searching for a file that is uploaded by the user is easy and takes less time.
6. As we are using a single bucket, if multiple users upload an image with the same name, then it will cause an issue when fetching the appropriate requested file because the image might be overridden with the newly uploaded image, or it may be going on with the versioning of the image. Only the latest version image is fetched, which is wrong.

Application Instructions:

1. Please type in the URL in your local browser and click enter or click on the URL here.
2. you will be routed into the application when you click on the URL.

3. On the screen, you can see the list of existing images that have already been uploaded to the server.

Visualize the existing images:

4. To view the existing uploaded image, click on the image you like. On click, you will be routed to the My Image HTML page which displays the image along with the metadata of that image and the image name at the top. You can switch back to the application tab to continue accessing the application.

Upload a new image:

5. 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.
6. Immediately, you can see that the image has been pulled up from your device and is ready to upload.
7. Click the upload button to upload the image into the GCP storage bucket.
8. On 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.
9. 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 select an image and then click the upload option.

Download an existing image:

9. 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. Do a right-click on the image and select the Save the image as option. Once the destination saving path is selected, the image will be downloaded. You can switch back to the application tab to continue accessing the application.

Improvements:

When millions of users are present, this application is unsuccessful, as managing the user's images will be very tough.

So, as an improvement for this application, we can create user authentication and create a bucket for each user so that the bucket stores the images uploaded by that user, and also fetching the user-uploaded images is quick in this case.

Lessons learned:

In addition to designing a simple application from scratch, Project 2 has taught us how to make your application serverless and independent on the VM to store the images and metadata. First, we need to learn how to establish the connection to the storage bucket and the datastore database. We installed and authenticated the Google SDK before testing the code. With this project, we have learned how to connect to the storage bucket, the need for a service account, and how to extract the metadata from an uploaded image and push it into the datastore database. We also learned how to get the expected metadata and the expected images every time the user clicks on a particular image. We came across multiple errors in connecting to cloud storage. We also learned that the images we downloaded may or may not have metadata, or very little metadata. Some images have metadata of different datatypes unsupported by the cloud datastore database. We have encountered many errors, and we learned how to solve them. We learned how to deploy the code serverless now using the Cloud Run service. We learned the need for the Dockerfile, how to build and run it, and then use Cloud Run to deploy it for a successful running application.