Build, deploy, and run a Containerized Application using GCP. Using GCP create and validate an online meeting account.

1. Beginning with Creation of repositories in Artifact Registry for container 1, 2, 3 as shown in the image Fig 1.

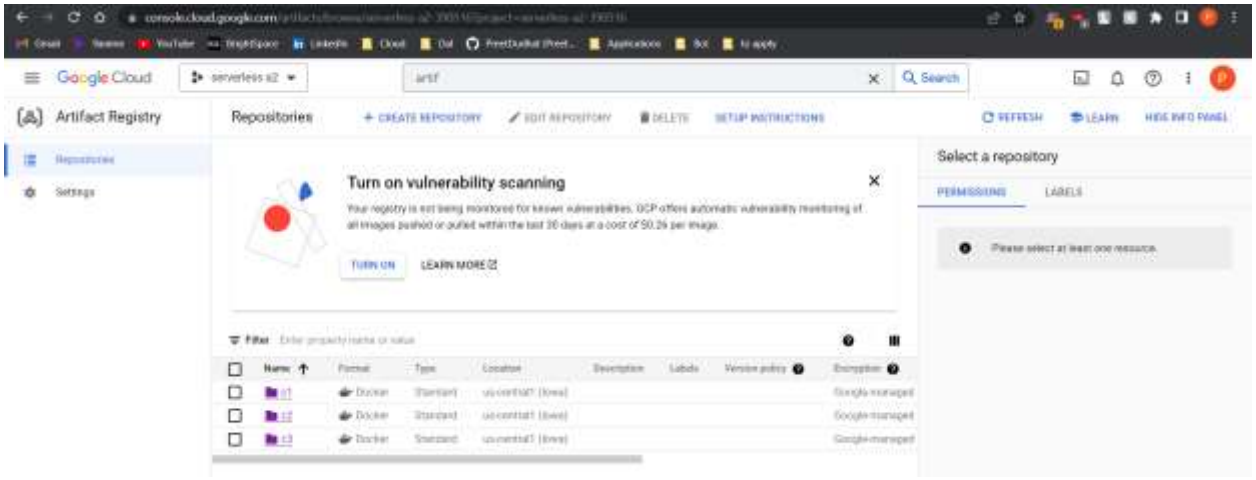

Fig 1

2. Next we will build docker images, tag them and push them to Artifact Registry as shown in Fig 1(a), 1(b), 1(c). We can verify if the images has been pushed to Artifact Registry which is shown in Fig 2, 3, 4. c1 has registration service, c2 has login service, c3 has logout and state service.

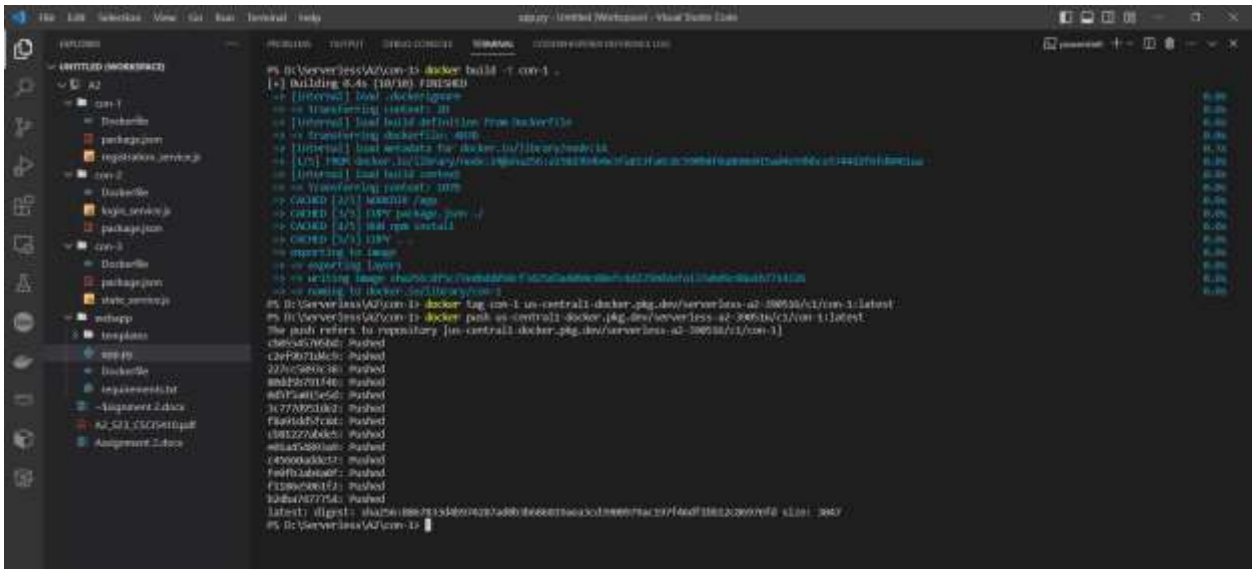

Fig 1(a)

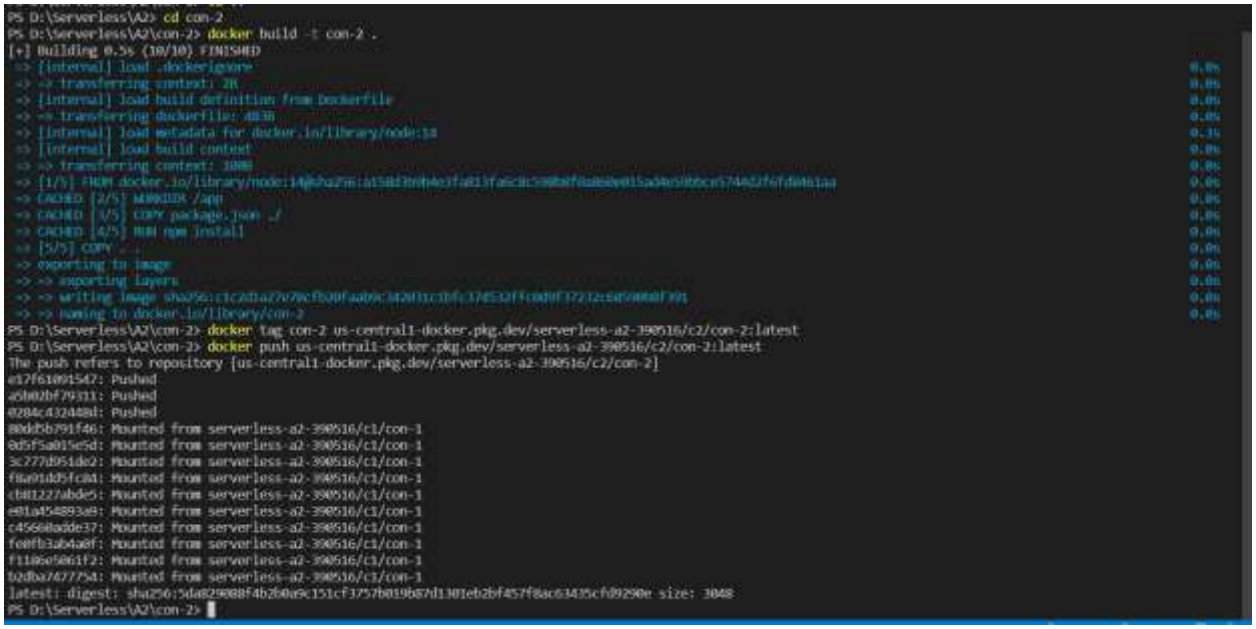
PS D:\Serverless\A2> cd con-2
PS D:\Serverless\A2\con-2> docker build -t con-2 .
[+] Building 0.5s (10/10) FINISHED
PS D:\Serverless\A2\con-2> docker tag con-2 us-central1-docker.pkg.dev/serverless-a2-390516/c2/con-2:latest
PS D:\Serverless\A2\con-2> docker push us-central1-docker.pkg.dev/serverless-a2-390516/c2/con-2:latest
The push refers to repository [us-central1-docker.pkg.dev/serverless-a2-390516/c2/con-2]
e17f61001547: Pushed
a5b02bf79331: Pushed
0284c43248bd: Pushed
80dd5b791f46: Mounted from serverless-a2-390516/c1/con-1
8d5f5a015e5d: Mounted from serverless-a2-390516/c1/con-1
3c777d951de2: Mounted from serverless-a2-390516/c1/con-1
f8a01dd5fc04: Mounted from serverless-a2-390516/c1/con-1
cb81227abde5: Mounted from serverless-a2-390516/c1/con-1
e01a454893a9: Mounted from serverless-a2-390516/c1/con-1
c45660adde37: Mounted from serverless-a2-390516/c1/con-1
fe0fb3ab4a0f: Mounted from serverless-a2-390516/c1/con-1
f1186e5061f2: Mounted from serverless-a2-390516/c1/con-1
b2dba7477754: Mounted from serverless-a2-390516/c1/con-1
latest: digest: sha256:5da829888f4b2b0a9c151cf3757b019b87d1301eb2bf457f8ac63435cfd9290e size: 3048
PS D:\Serverless\A2\con-2>

Fig 1(b)

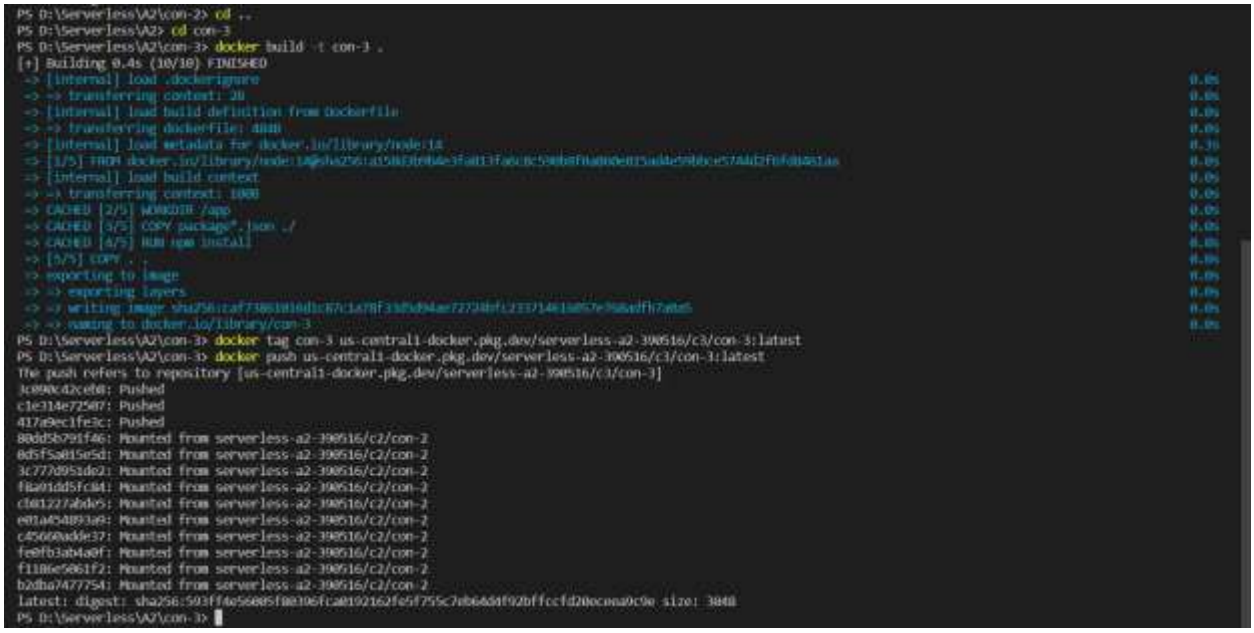
PS D:\Serverless\A2\con-2> cd ..
PS D:\Serverless\A2> cd con-3
PS D:\Serverless\A2\con-3> docker build -t con-3 .
[+] Building 0.4s (10/10) FINISHED
PS D:\Serverless\A2\con-3> docker tag con-3 us-central1-docker.pkg.dev/serverless-a2-390516/c3/con-3:latest
PS D:\Serverless\A2\con-3> docker push us-central1-docker.pkg.dev/serverless-a2-390516/c3/con-3:latest
The push refers to repository [us-central1-docker.pkg.dev/serverless-a2-390516/c3/con-3]
c1e314e7250f: Pushed
417a0ec1fe3c: Pushed
80dd5b791f46: Mounted from serverless-a2-390516/c2/con-2
8d5f5a015e5d: Mounted from serverless-a2-390516/c2/con-2
3c777d951de2: Mounted from serverless-a2-390516/c2/con-2
f8a01dd5fc04: Mounted from serverless-a2-390516/c2/con-2
cb81227abde5: Mounted from serverless-a2-390516/c2/con-2
e01a454893a9: Mounted from serverless-a2-390516/c2/con-2
c45660adde37: Mounted from serverless-a2-390516/c2/con-2
fe0fb3ab4a0f: Mounted from serverless-a2-390516/c2/con-2
f1186e5061f2: Mounted from serverless-a2-390516/c2/con-2
b2dba7477754: Mounted from serverless-a2-390516/c2/con-2
latest: digest: sha256:593ff4e56005f80306fca0192162fe5f755c7eb6604f92bffccfd20ccea0c9e size: 3048
PS D:\Serverless\A2\con-3>

Fig 1(c)

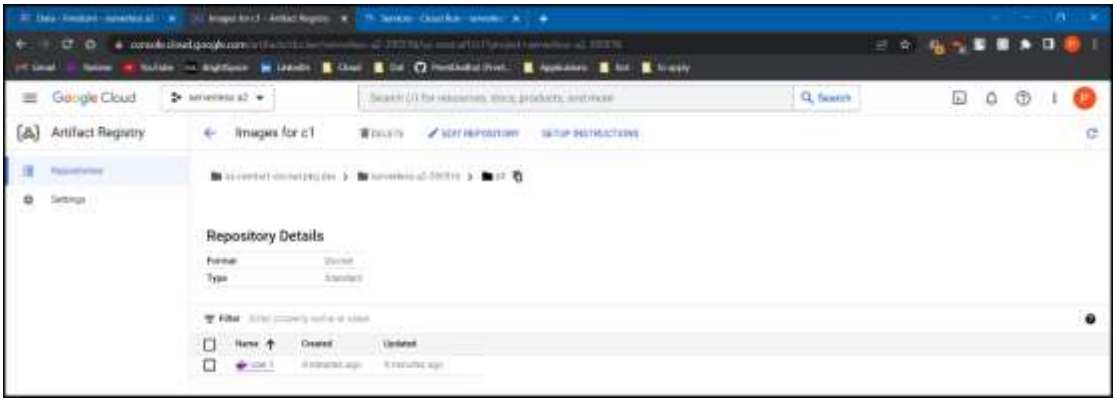

Fig 2

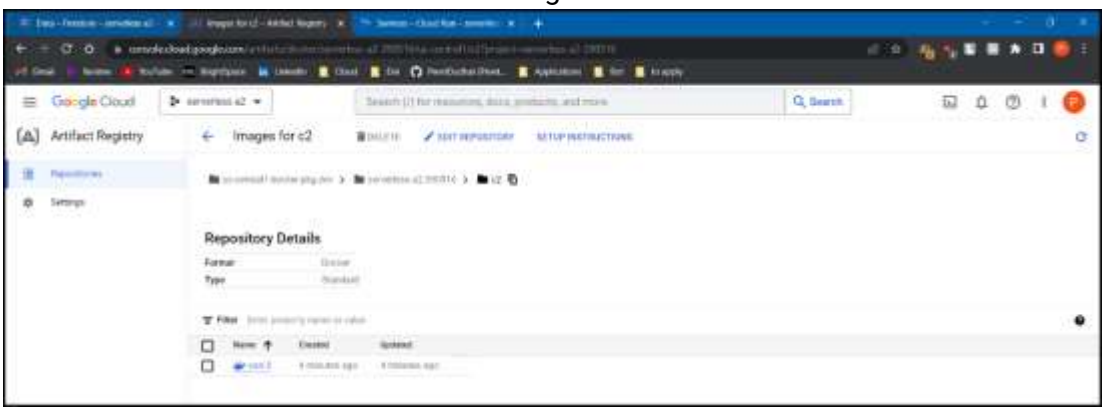

Fig 3

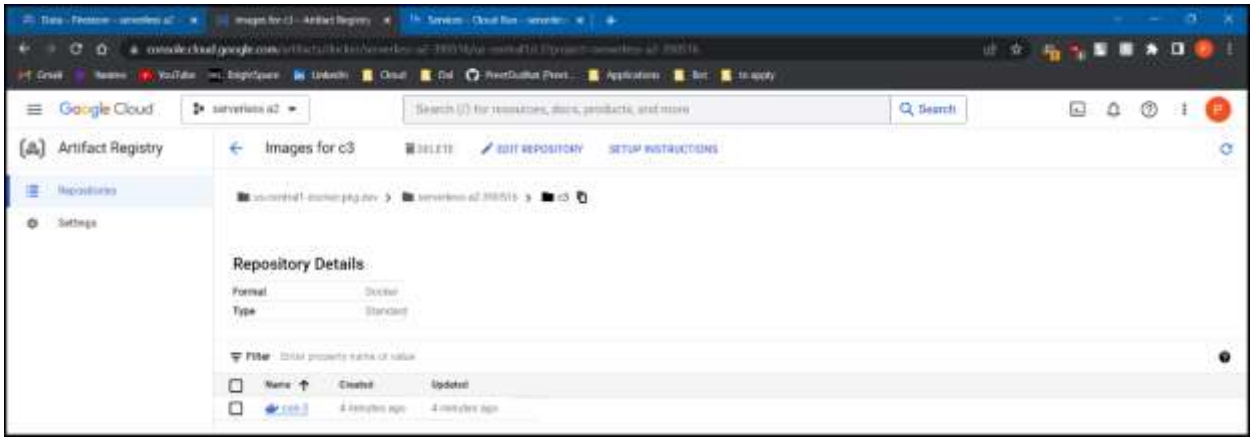

Fig 4

3. Next we will create services for each of the images we have in Artifact Registry which is shown in Fig 5. All the necessary configurations must be made which is shown in Fig 5(a), 5(b), 5(c) this configurations include Selecting image we want to deploy, checking unautorized invocations in Authentication as right now we are creating a website, and lastly setting port to what we have used in the Dockerfile for each image when we were building the image.

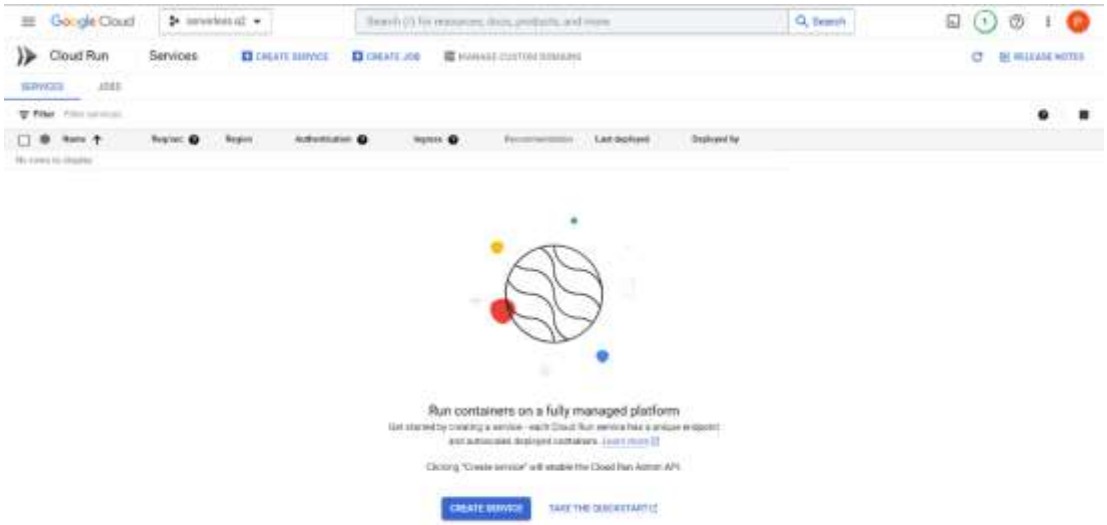
Google Cloud
Cloud Run
Services
CREATE SERVICE
CREATE JOB
Run containers on a fully managed platform
CREATE SERVICE

Fig 5

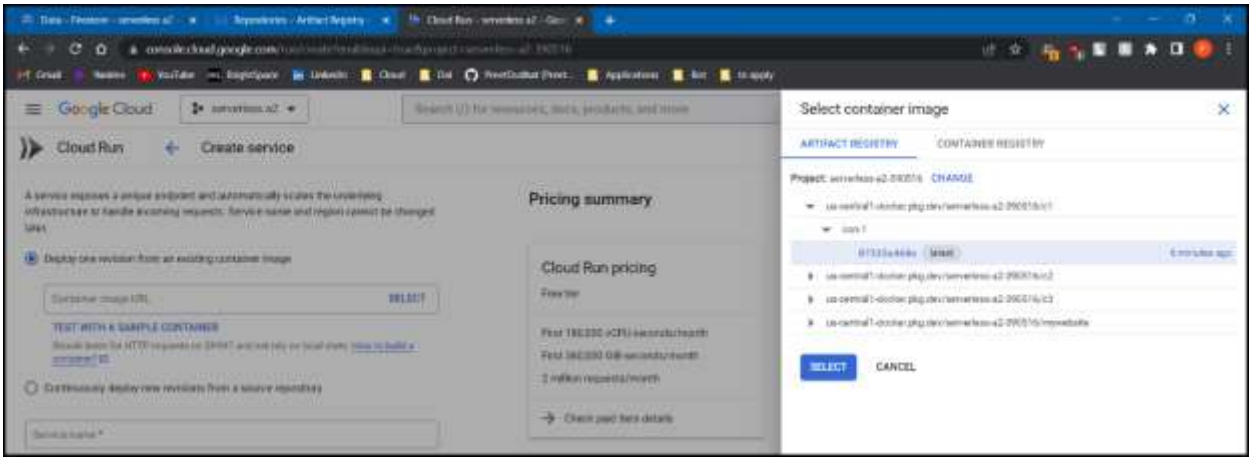
Google Cloud
Cloud Run
Create service
Pricing summary
Cloud Run pricing
Select container image
ARTIFACT REGISTRY
CONTAINER REGISTRY
SELECT
CANCEL

Fig 5(a)

Fig 5(b)

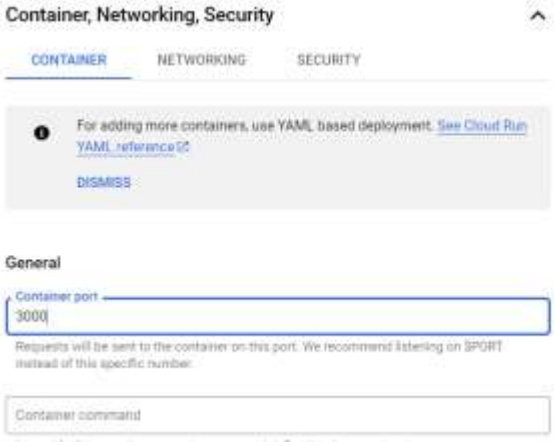

Fig 5(c)

4. Like the above, all the services that will be created with the same configuration just a minor change which is port number which are c1: 3000, c2: 3001, c3: 3003 which are shown in Fig 5(d), 5(e). After waiting for some time all the services will be created (as shown in Fig 6(a)) and we need to take the URL which can be seen in Fig 6(b), 6(c), 6(d) of each service and copy to our code to use them which is shown in Fig 7(a), 7(b), 7(c), 7(d).

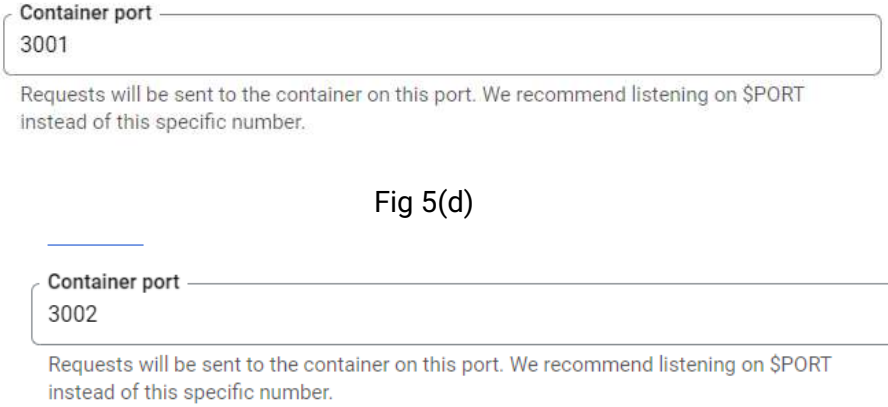

Fig 5(d)

Fig 5(e)

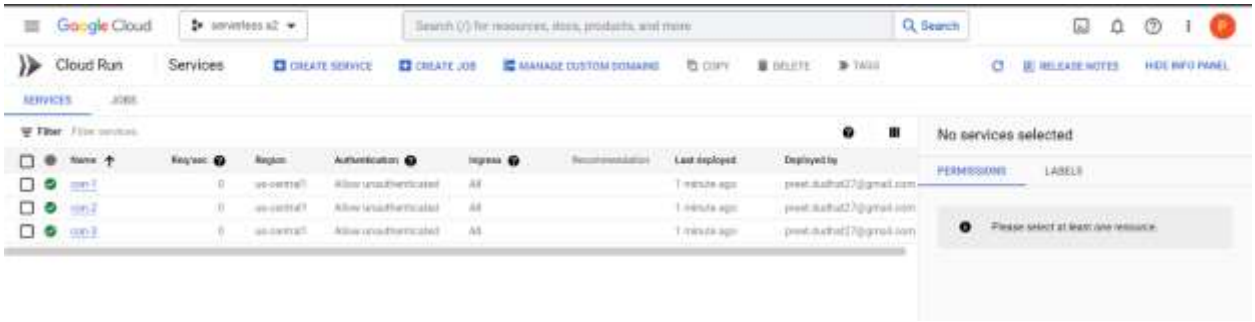

Fig 6(a)

Fig 6(b)

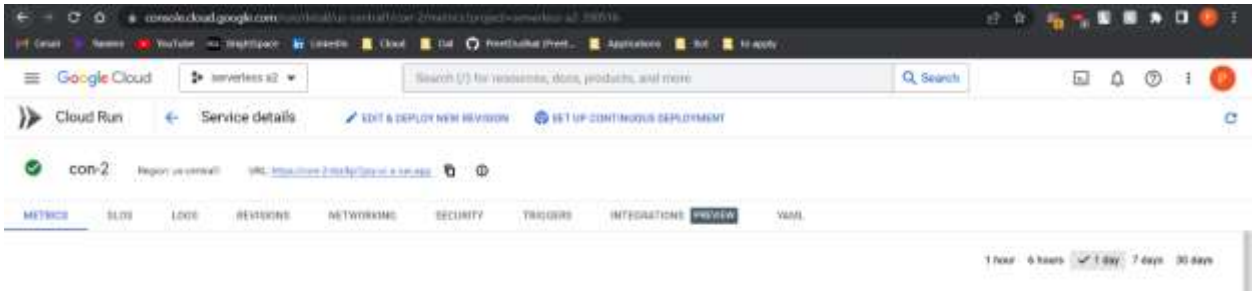

Fig 6(c)

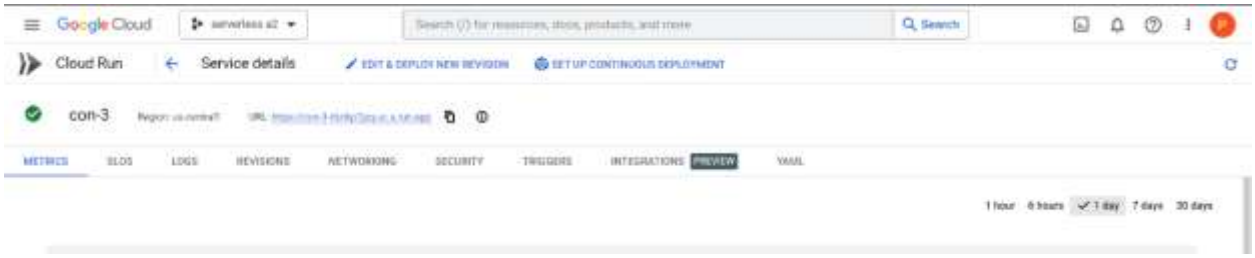

Fig 6(d)

```python
@app.route('/register', methods=['POST'])
def register():
    name = request.form['name']
    email = request.form['email']
    password = request.form['password']
    location = request.form['location']

    # Send a POST request to the registration URL to add data to Firestore
    registration_url = "https://con-1-rbzikp7jzq-uc.a.run.app/register"
    payload = {
        'name': name,
        'email': email,
        'password': password,
        'location': location
    }
    response = requests.post(registration_url, json=payload)

    return 'Registration successful! <a href="/">Click here to login</a>'
```

Fig 7 (a)

```python
@app.route('/login', methods=['POST'])
def login():
    global name  # Use the global name variable

    email = request.form.get('email')
    password = request.form.get('password')

    login_url = 'https://con-2-rbzikp7jzq-uc.a.run.app/login'
    login_data = {'email': email, 'password': password}

    headers = {'Content-Type': 'application/json'}  # Set the content type to JSON

    response = requests.post(login_url, json=login_data, headers=headers)

    if response.status_code == 200:
        data = response.json()
        if 'message' in data and data['message'] == 'Login successful':
            data = response.json()
            name = data.get('name', '')  # Update the global name variable
            return redirect(url_for('dashboard'))
        elif 'error' in data:
            return render_template('index.html', error=data['error'])

    return 'Login Failed!!!<a href="/">Click here to login</a>'
```

Fig 7(b)

```python
@app.route('/dashboard')
def dashboard():
    global name  # Use the global name variable

    online_users_url = 'https://con-3-rbzikp7jzq-uc.a.run.app/online-users'
    response = requests.get(online_users_url)

    if response.status_code == 200:
        data = response.json()
        online_users = data.get('onlineUsers', [])
        online_users = [user for user in online_users if user['name'] != name]
        return render_template('dashboard.html', online_users=online_users, name=name)
    return render_template('dashboard.html', online_users=[], name='')
```

Fig 7(c)

```python
@app.route('/logout', methods=['POST'])
def logout():
    global name  # Use the global name variable

    # Construct the JSON payload
    payload = {
        'name': name
    }
    headers = {'Content-Type': 'application/json'}  # Set the content type to JSON

    logout_url = "https://con-3-rbzikp7jzq-uc.a.run.app/logout"
    response = requests.post(logout_url, json=payload, headers=headers)
    return f"Logout successful for user: {name}! <a href='/'>Click here to login</a>"
```

Fig 7(d)

5. Now we will repeat Steps 2, 3, 4 to create artifact registry for website, service for website, which is shown in Fig 8(a), 8(b), 8(c), 9 (setting the port to 7000) respectively.

Fig 8(a)

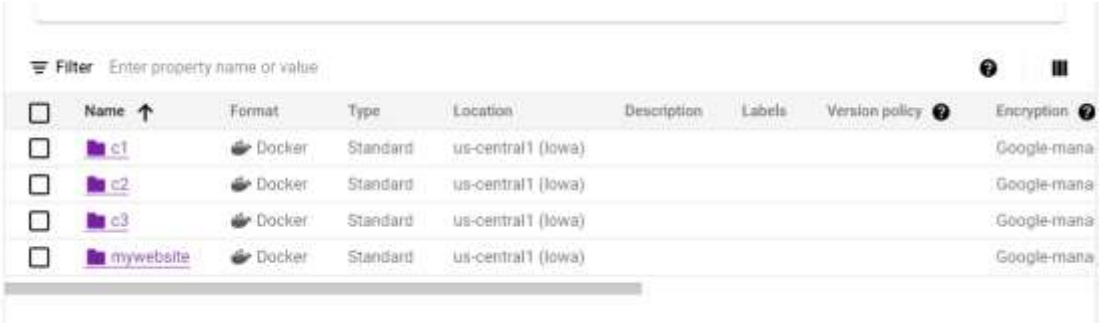

Fig 8(b)

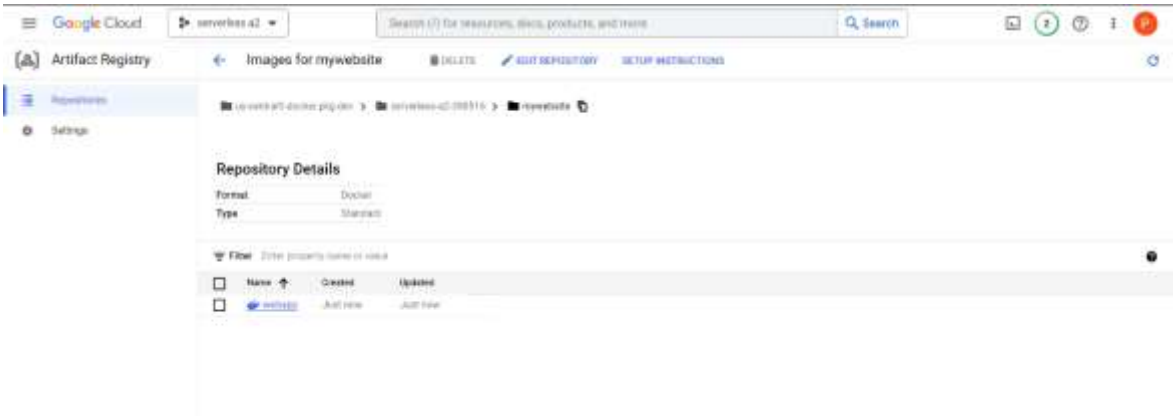

Fig 8(c)

Fig 9

6. Now as all the services are created as shown in Fig 10, we will open webapp service in which we will have a URL, clicking will open the website as shown in Fig 11,12.

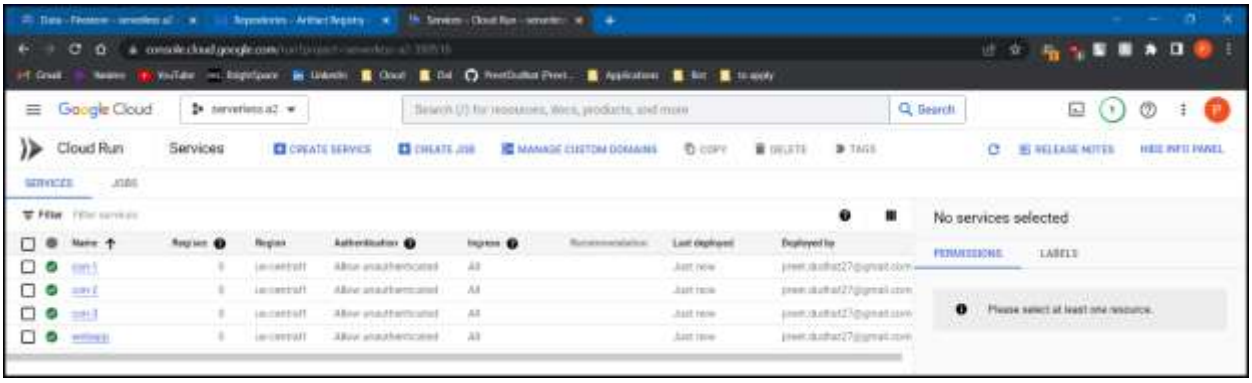

Fig 10

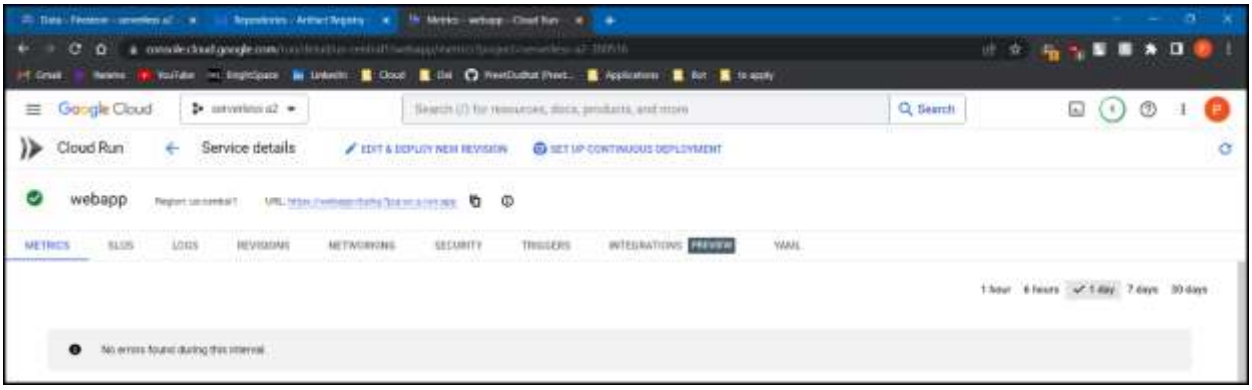

Fig 11

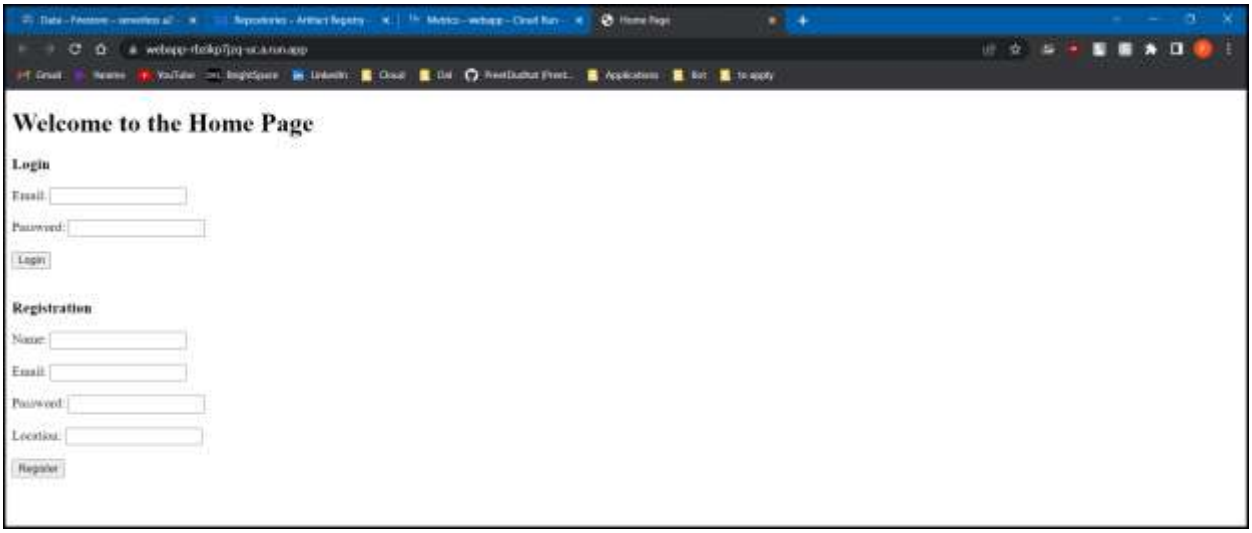

Fig 12

7. As we can see in Fig 13, Firestore has no collection as we register and login the Reg and state collection will be created. First, I have registered user Jack Sparrow as shown in Fig 14, 15. After registration we can see in Fig 16, in firestore Reg collection is created with document which have our registered details of the user.

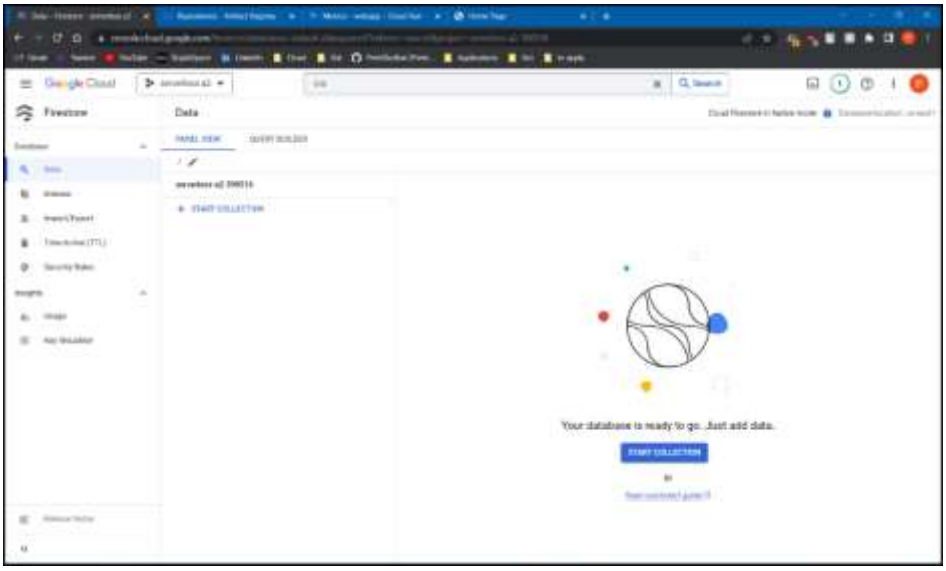

Fig 13

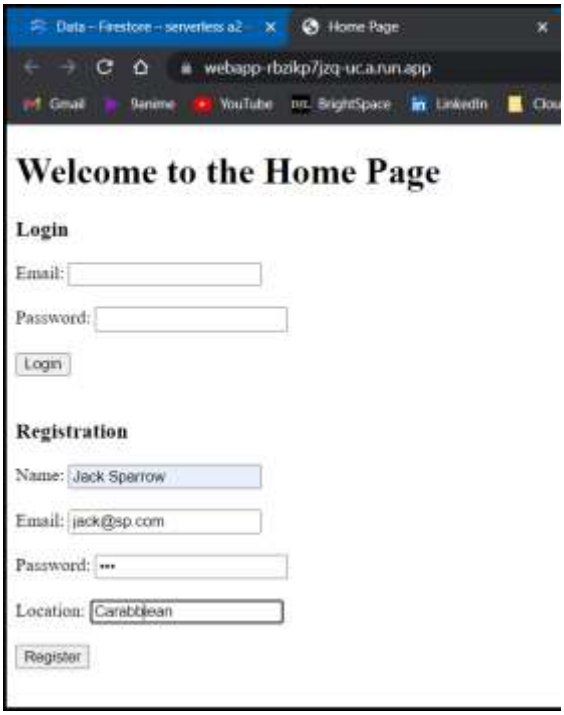

Fig 14

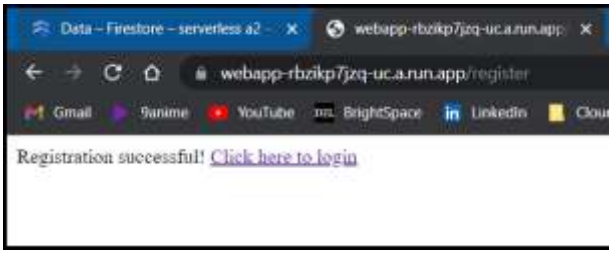

Fig 15

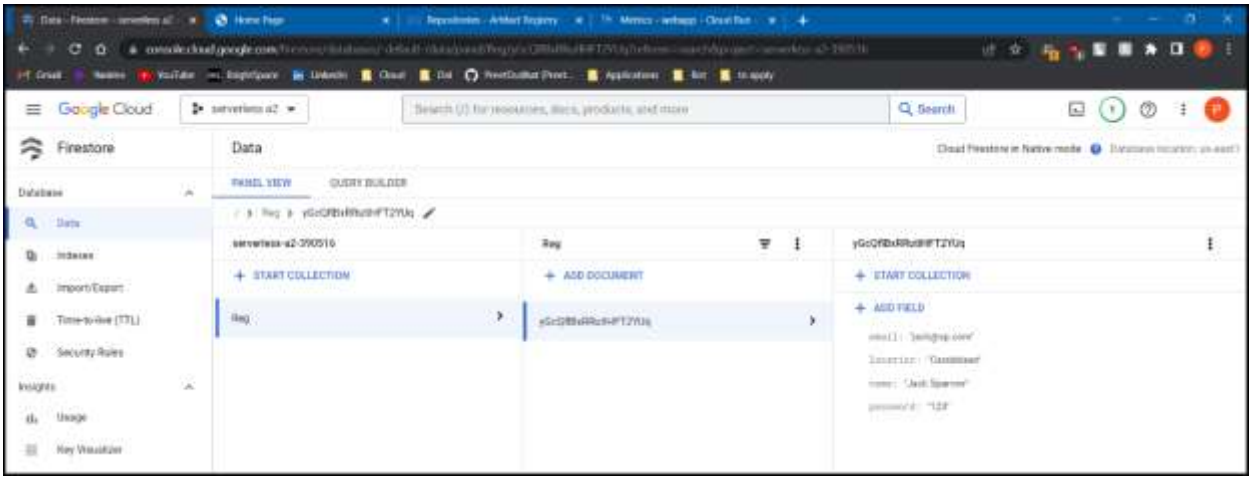

Fig 16

8. Now we will login to Jack Sparrow's account with wrong password as shown in Fig 17, 18 and with correct password as shown in Fig 19 which will open login dashboard as shownin Fig 20. We can also, see in Firestore that state service has been created as shown in Fig 21.

Fig 17

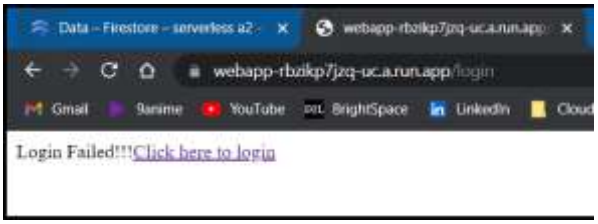

Fig 18

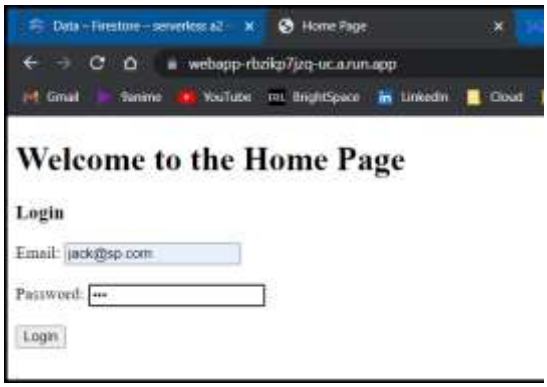

Fig 19

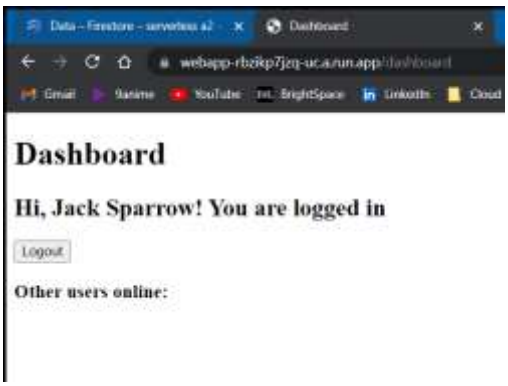

Fig 20

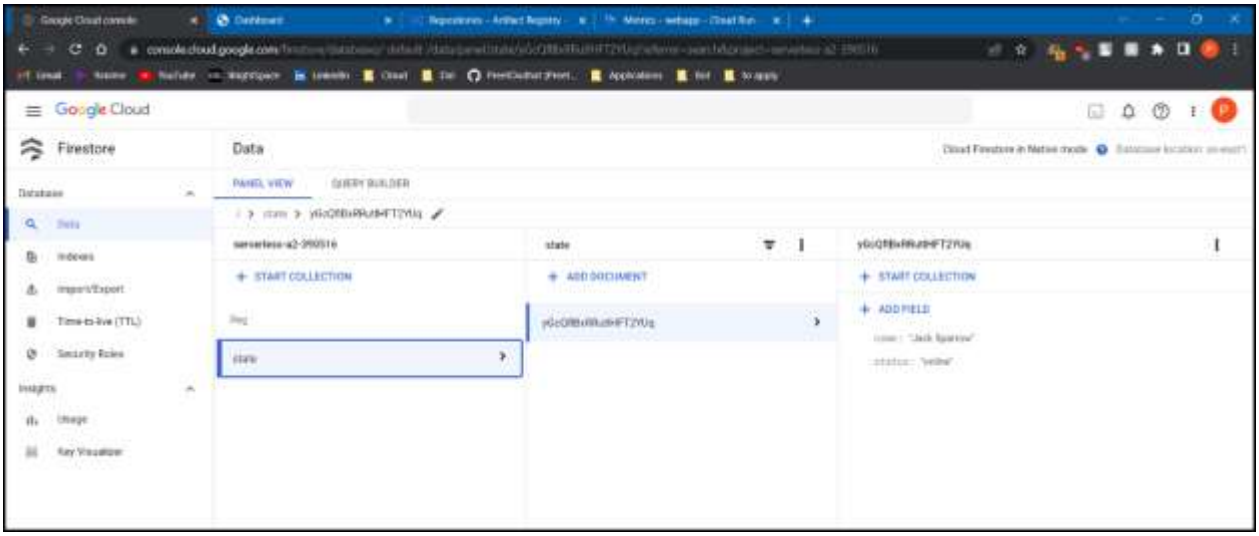

Fig 21

9. As we can see in Fig 20, that there are no other users online we will register a few users which will populate the Firestore database. So, Fig 22, 23, 24 new users are created and Fig 25, 26, 27 shows that data has been added to Firestore database.

Fig 22

Fig 23

Fig 24

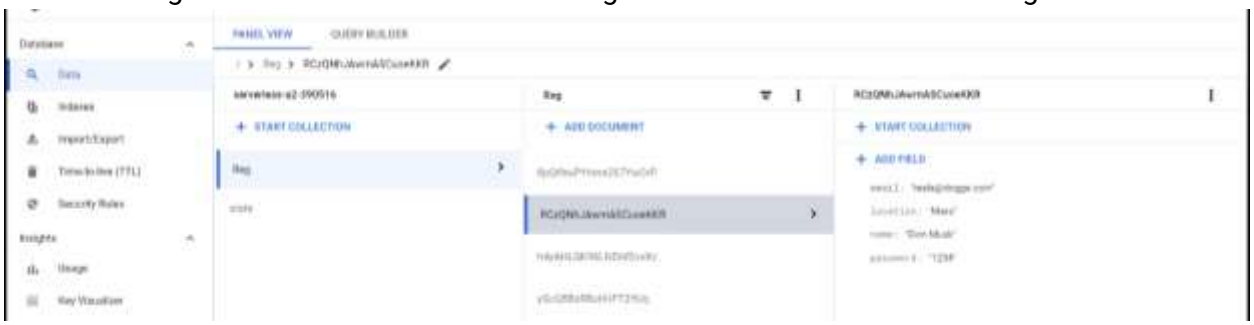

Fig 25

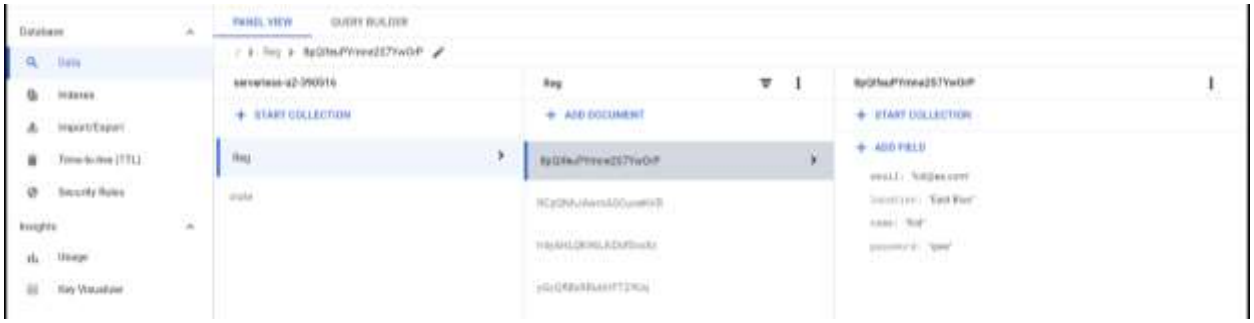

Fig 26

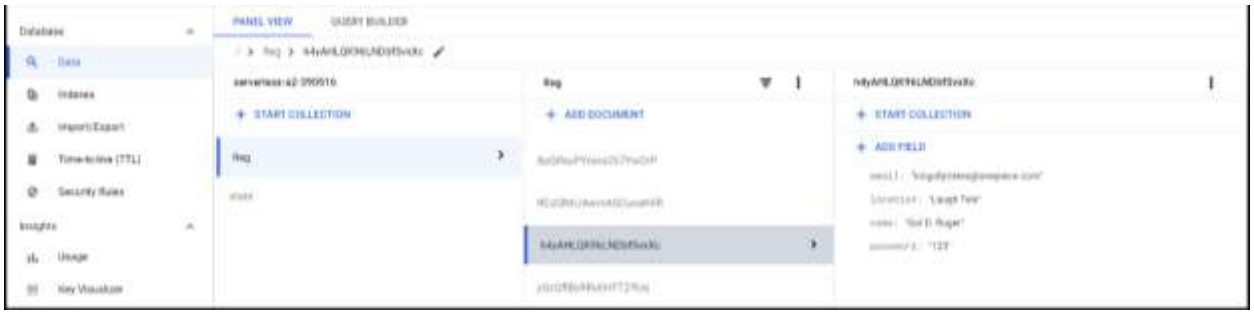

Fig 27

10. Now we will start logging in to different accounts. Note we are logging into different accounts using the link repetitively in new tabs and not logging out in any of the accounts. We can see from Fig 28, 29, 30 that other users online are been displayed as we log in to different accounts and from Fig 31, 32, 33 we can see that all the state services have been created.

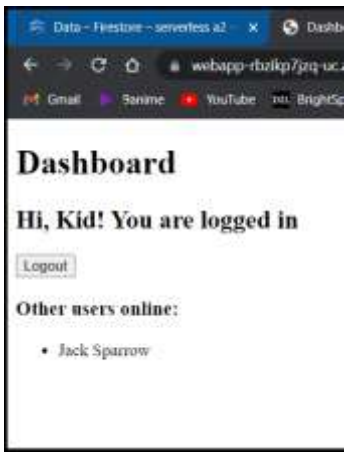

Fig 28

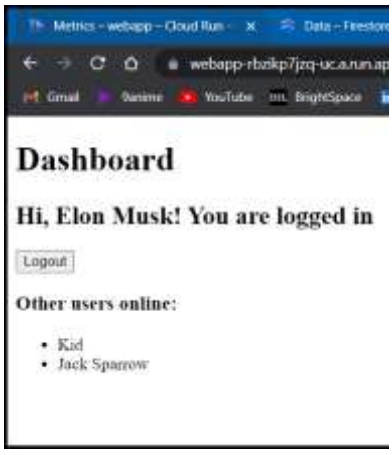

Fig 29

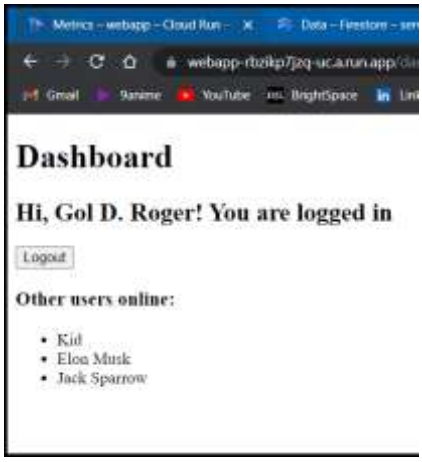

Fig 30

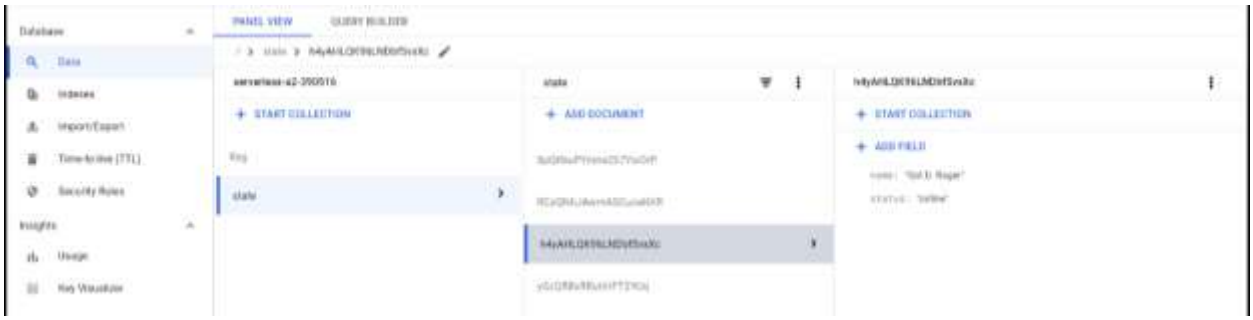

Fig 31

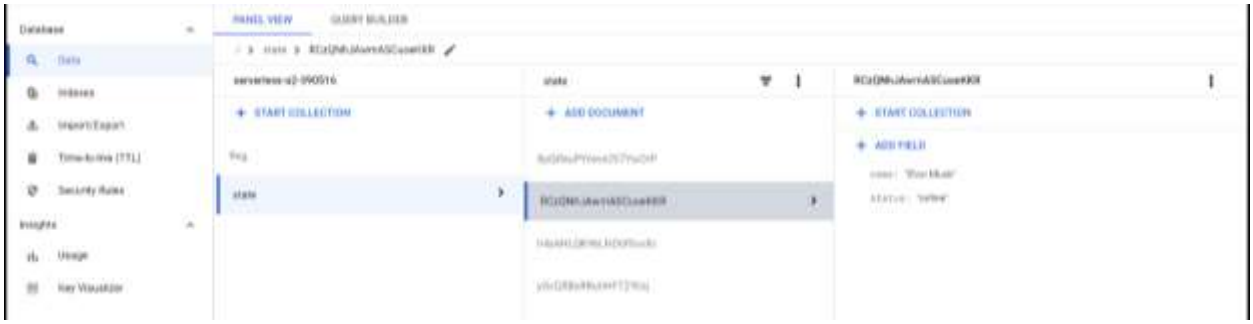

Fig 32

Fig 33

11. So, now we will begin to log out of accounts and check that the other accounts online users have been updated. First, we will logout from account Kid as shown in Fig 34. Next, we will login to account Gol D. Roger and Jack Sparrow to see if the Kid has been removed from the online users list which can be seen in Fig 35, 36.

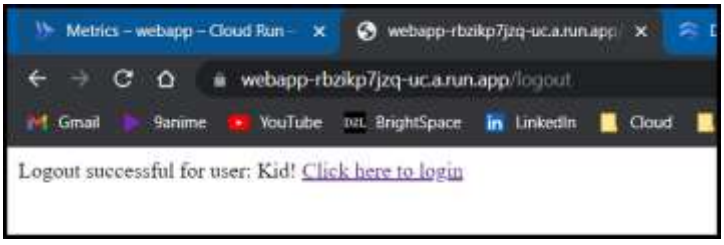

Fig 34

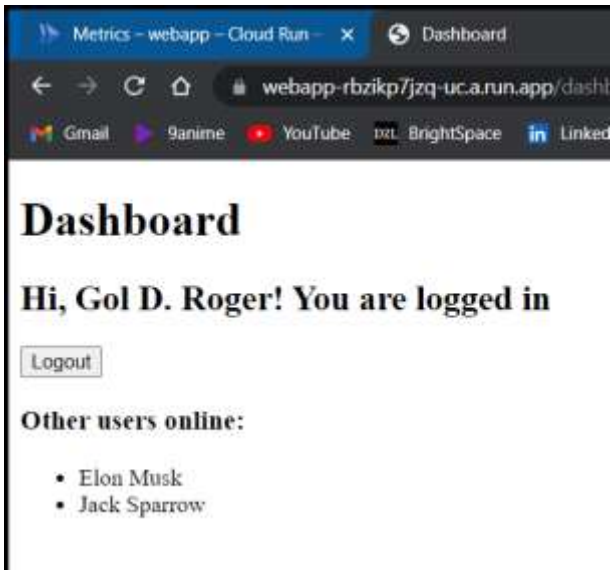

Fig 35

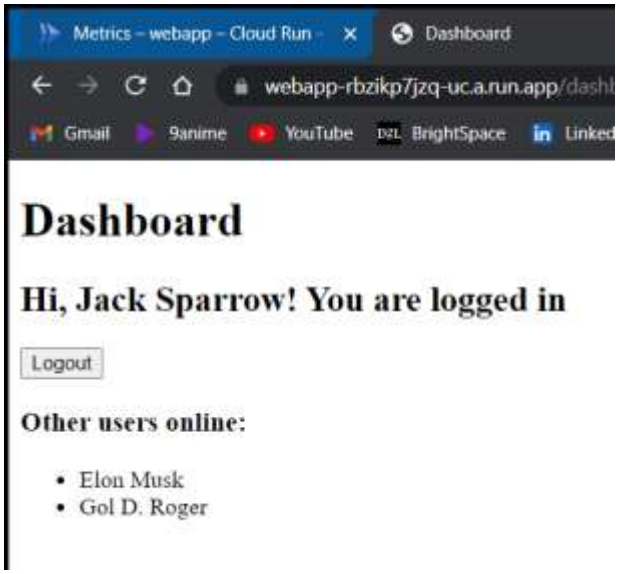

Fig 36

12. So, now we will be logging out of each user as shown in Fig 37, 38, 39 and from Fig 40, 41, 42, 43 we can see that in Firestore the status of all the users have been changed to offline.

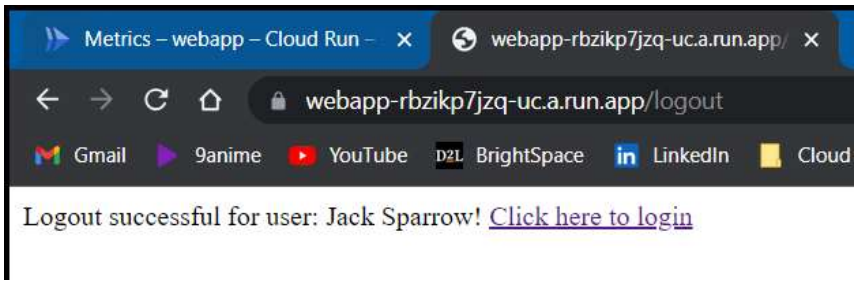

Fig 37

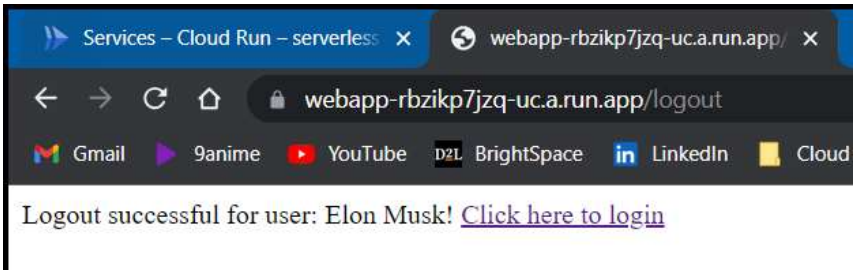

Fig 38

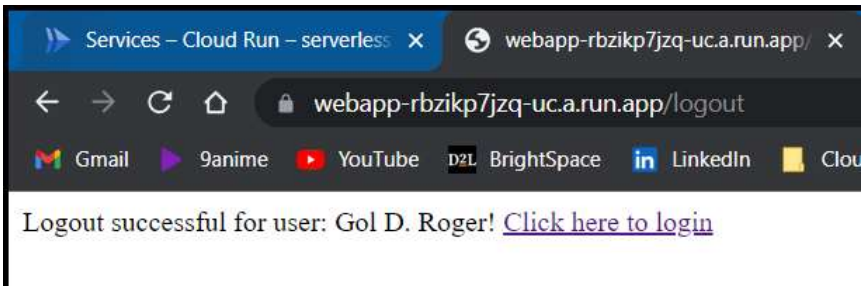

Fig 39

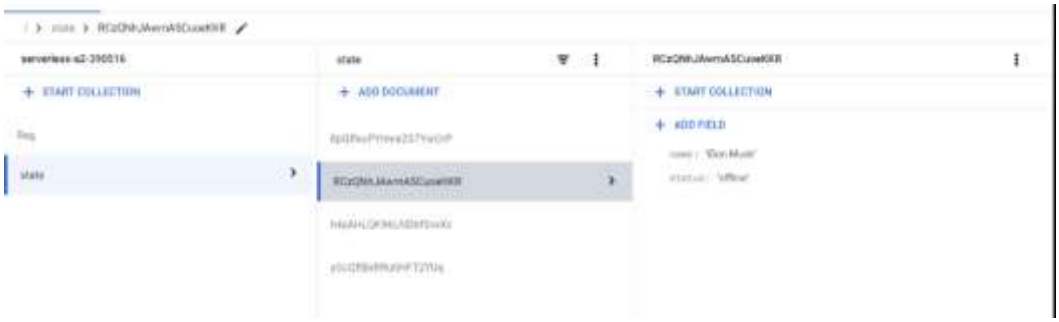

Fig 40

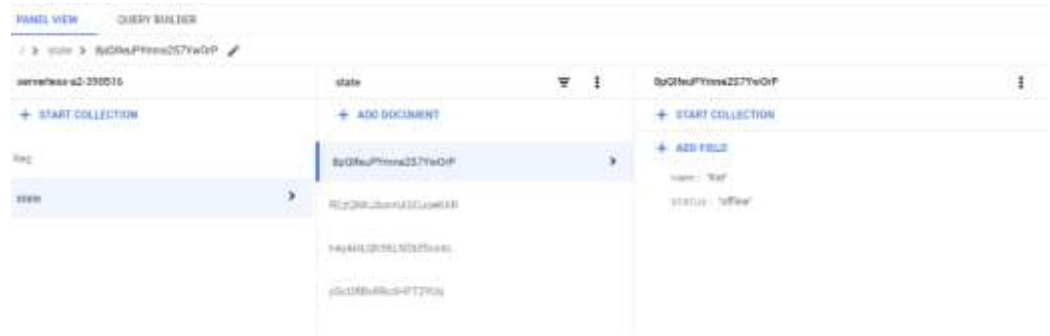

Fig 41

Fig 42

Fig 43

Test Cases:

1. First, we will be testing the Registration Service (test_reg.py) which ran successfully as can be seen in Fig 44 (command Line), 45 (Firestore), 46(cloud run log). In cloud run log it's the one which has been highlighted.

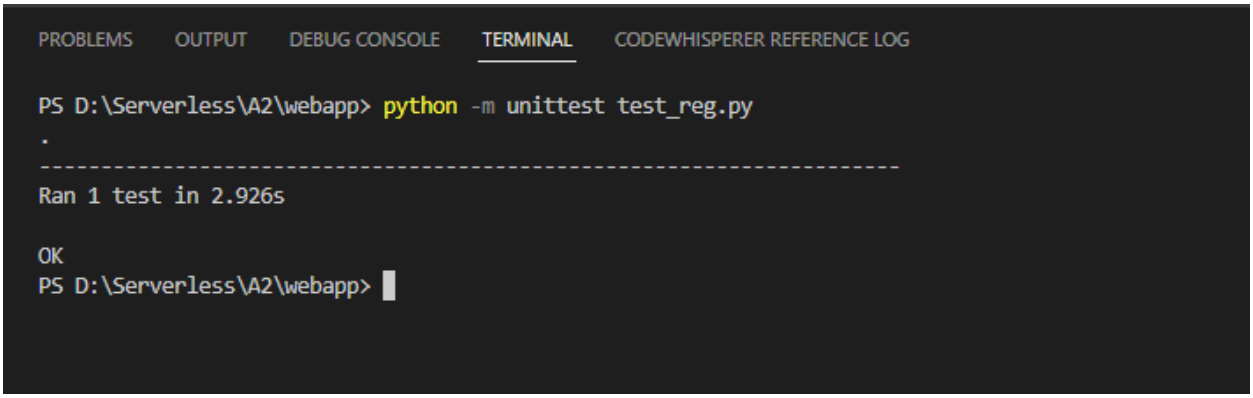

Fig 44

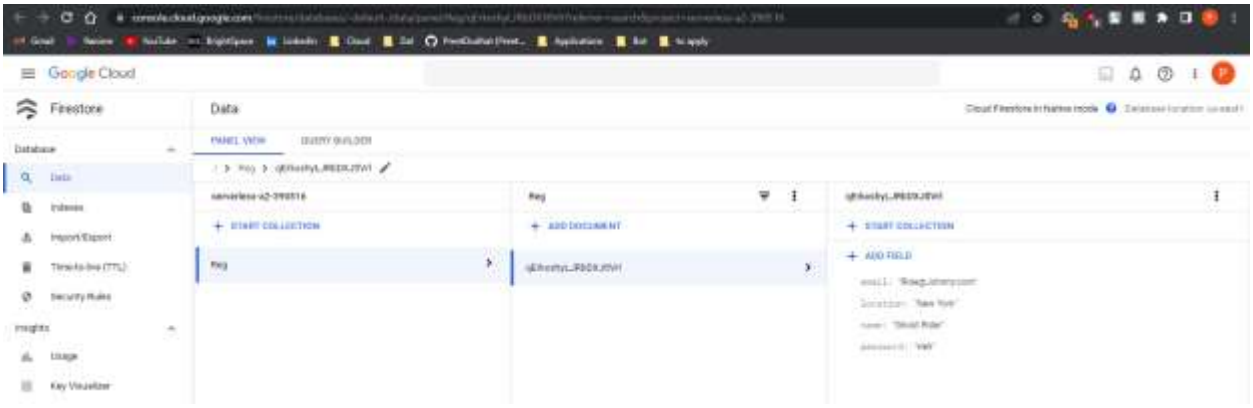

Fig 45

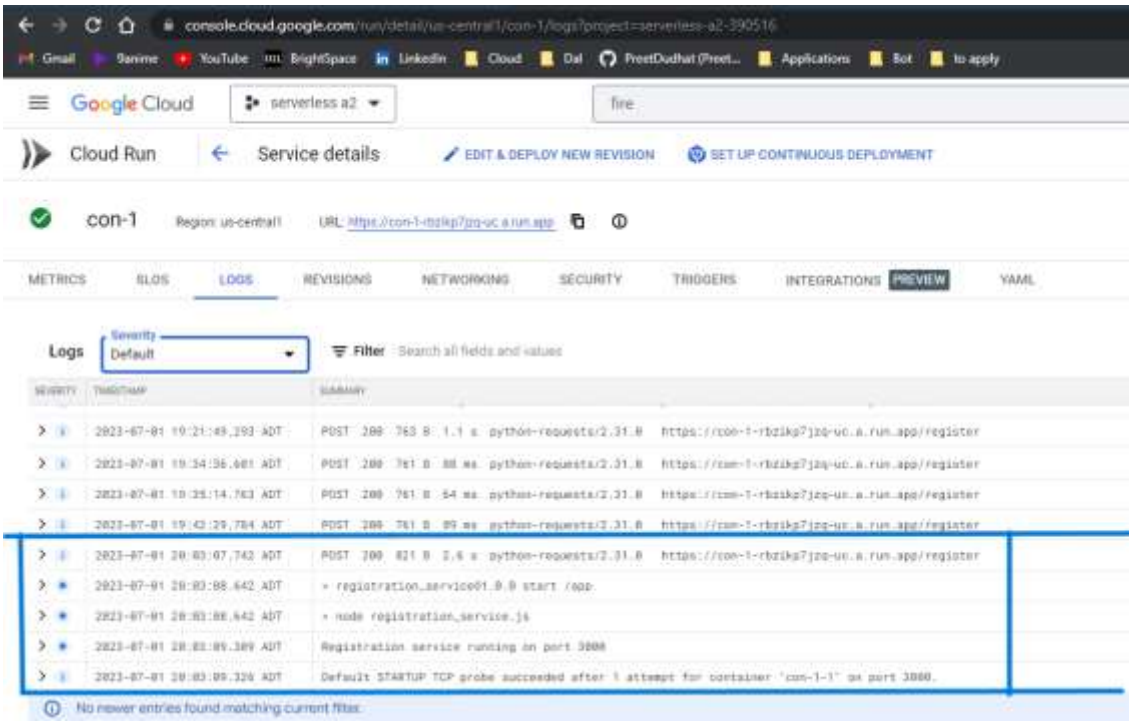

Fig 46

2. Now we will test for Log in Service, first I am going to test for wrong credentials which is shown in Fig 47, 48. This test ran successfully as in Fig 48 we can see that state is not created.
   Next with correct credentials the test is shown in Fig 49,50 which also, ran successfully as state is created and status is login as shown in Fig 50 and in Fig 51, I have shown Cloud run the error one is warning which is showing for our Login2(Wrong Credentials) and Login1(Correct Credentials).

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   CODEWHISPERER REFERENCE LOG

PS D:\Serverless\A2\webapp> python -m unittest test_reg.py
.
----------------------------------------------------------------------
Ran 1 test in 2.926s

OK
PS D:\Serverless\A2\webapp> python -m unittest test_login2.py
.
----------------------------------------------------------------------
Ran 1 test in 0.373s

OK
PS D:\Serverless\A2\webapp>
```

Fig 47

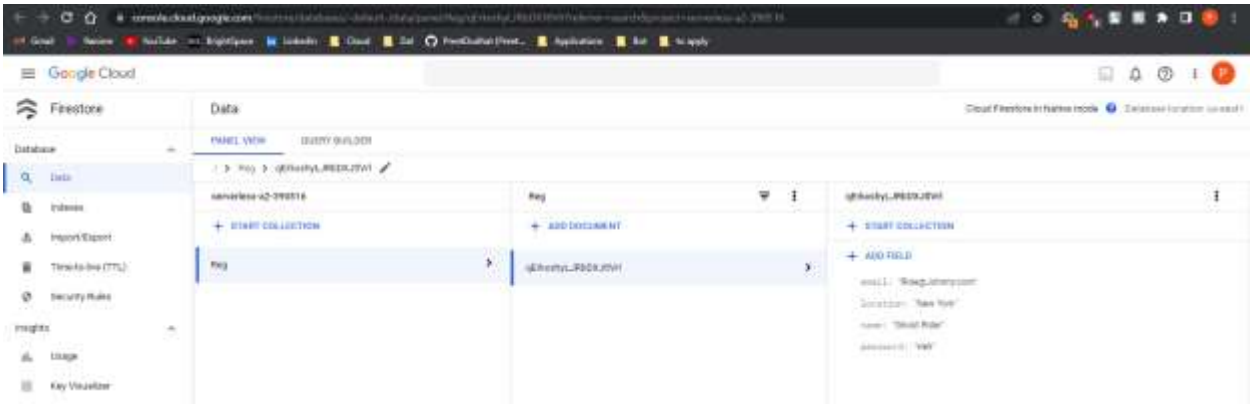

Fig 48

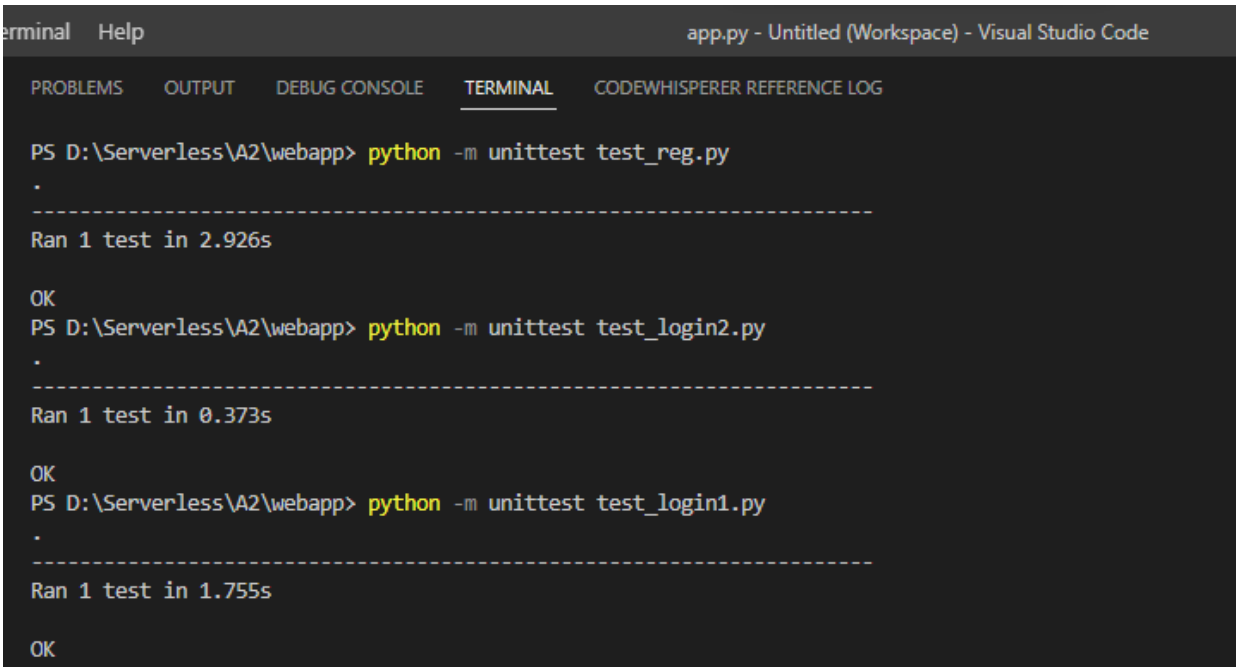

Fig 49

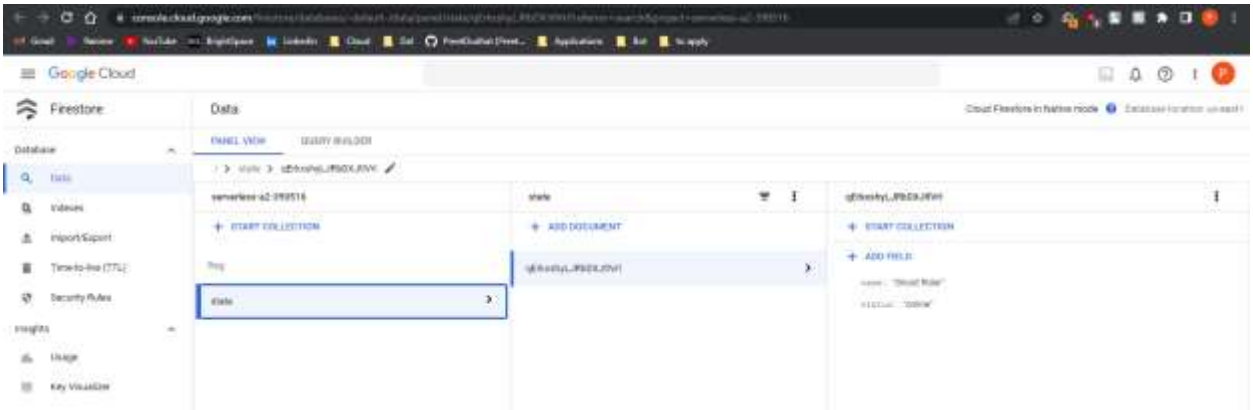

Fig 50

Fig 51

3. Now I am going to check for the last feature, which is Logout, which is shown in Fig 52, 53 and Fig 54 shows the cloud run log. The test was successful as we can see in Fig 50 the status changed to offline.

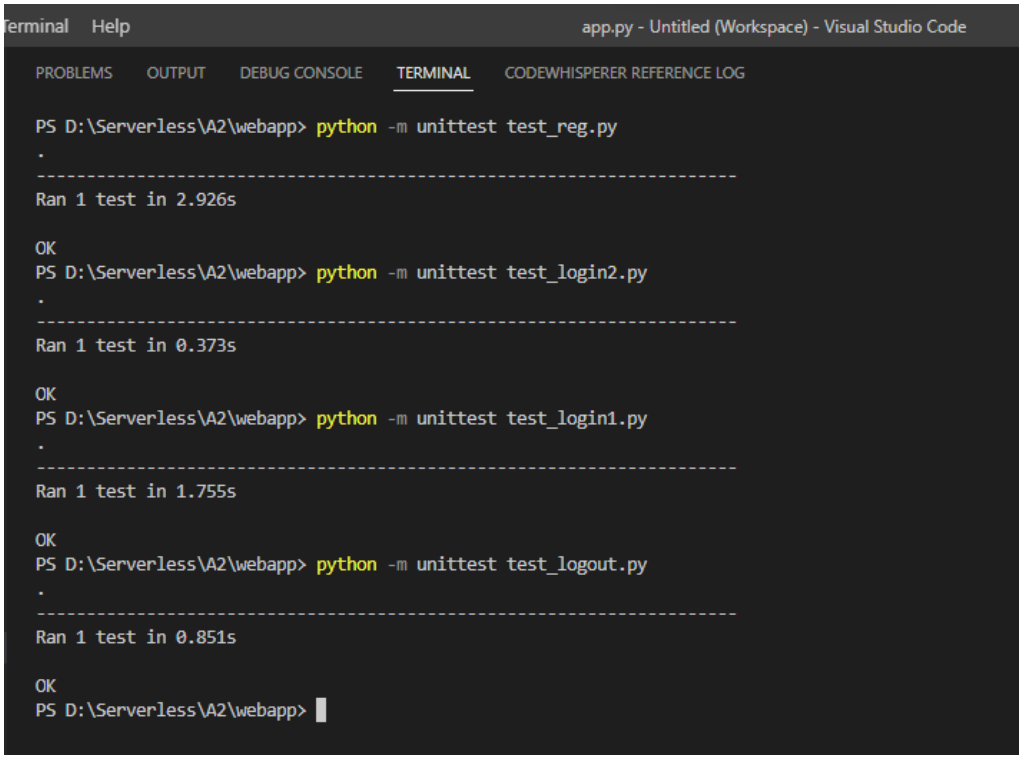

Fig 52

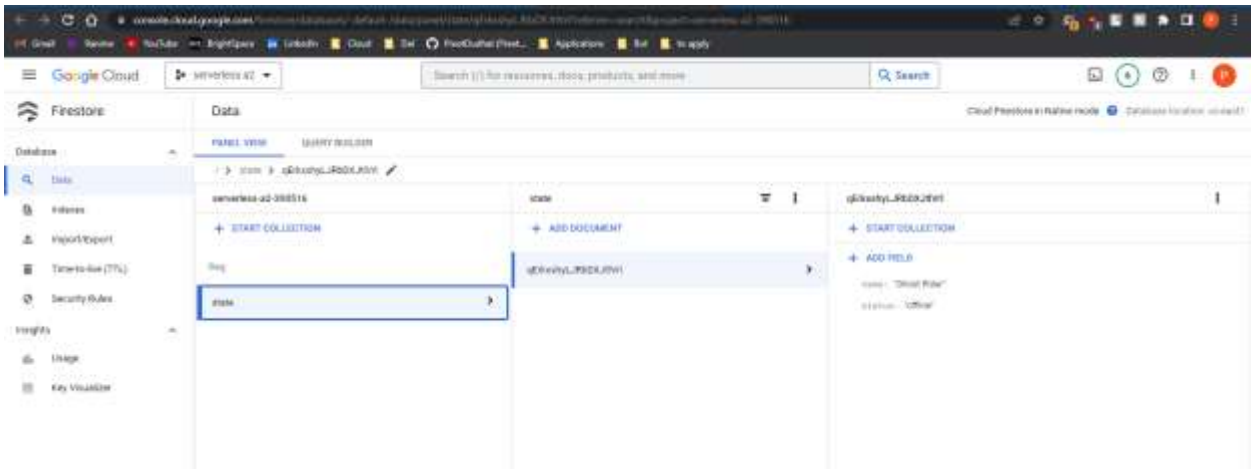

Fig 53

>	2023-07-01 20:00:39.820 ADT	POST	200	756 B	364 ms	python-requests/2.31.0	https://con-3-rbzikp7jzq-uc.a.run.app/logout
>	2023-07-01 20:05:47.120 ADT	GET	200	782 B	83 ms	python-requests/2.31.0	https://con-3-rbzikp7jzq-uc.a.run.app/online-users
>	2023-07-01 20:06:13.230 ADT	POST	200	756 B	229 ms	python-requests/2.31.0	https://con-3-rbzikp7jzq-uc.a.run.app/logout

Fig 54

Program Instructions and Pseudo Code:

Pseudo Code of Container 1 (Registration service):

1. Import express module.
2. Import Firestore module from '@google-cloud/firestore'.
3. Create an instance of the express application.
4. Create an instance of Firestore.
5. Use express.json() middleware to parse request bodies as JSON
6. Define a POST route '/register' with an async callback function (req, res)
   Try the following:
   Extract the name, email, password, and location from the request body.
   Get a reference to the 'Reg' collection in Firestore.
   Add a new document to the collection with the registration data.
   Respond with a success message (status 200)
   Catch any errors that occur:
   Log the error to the console.
   Respond with an error message (status 500)
7. Start the express application to listen on port 3000.
8. Print a message to the console indicating the registration service is running.
9. Next, Build the image of our webapp push to artifact registry.

Pseudo Code of Container 2 (Login service):

1. Import express module.
2. Import Firestore module from '@google-cloud/firestore'.
3. Create an instance of the express application.
4. Create an instance of Firestore.
5. Use express.json() middleware to parse request bodies as JSON
6. Define a POST route '/login' with an async callback function (req, res)
   Try the following:
   Extract the email and password from the request body.
   Get a reference to the 'Reg' collection in Firestore.
   Query the collection to find the user with matching email and password.
   If no matching user is found:
   Respond with an error message indicating invalid credentials (status 401)
   Return to exit the callback function.
   If a matching user is found:
   Get the data of the first document in the query snapshot.
   Get the ID of the first document in the query snapshot.
   Get a reference to the 'state' collection in Firestore.
   Set the status of the user to 'online' and store the user's name in Firestore.
   Respond with a success message and the user's name (status 200)
   Catch any errors that occur:

Log the error to the console.
Respond with an error message (status 500)

7. Start the express application to listen on port 3001.
8. Print a message to the console indicating the login service is running.
9. Next, Build the image of our webapp push to artifact registry.

Pseudo Code of Container 3 (state service and logout):

1. Import express module.
2. Import Firestore module from '@google-cloud/firestore'.
3. Create an instance of the express application.
4. Create an instance of Firestore.
5. Use express.json() middleware to parse request bodies as JSON
6. Define a GET route '/online-users' with an async callback function (req, res)

Try the following:
Get a reference to the 'state' collection in Firestore.
Query the collection to find online users with status 'online'.
Map the query snapshot documents to extract the online user data.
Respond with the array of online users (status 200)
Catch any errors that occur:
Log the error to the console.
Respond with an error message (status 500)

7. Define a POST route '/logout' with an async callback function (req, res)

Try the following:
Extract the name from the request body.
Get a reference to the 'state' collection in Firestore.
Query the collection to find the user with matching name.
If no matching user is found:
Respond with an error message indicating user not found (status 404)
Return to exit the callback function.
If a matching user is found:
Get the first document in the query snapshot.
Update the status of the document to 'offline' in Firestore.
Respond with a success message indicating successful logout (status 200)
Catch any errors that occur:
Log the error to the console.
Respond with an error message (status 500)

8. Start the express application to listen on port 3002.
9. Print a message to the console indicating the state service is running.
10. Next, Build the image of our webapp push to artifact registry.

Pseudo code for Container 4 (web app):

1. Import necessary modules: **Flask** for creating the application, **render_template** for rendering HTML templates, **request** for handling HTTP requests, **redirect** and **url_for** for URL redirection, **session** for session management, **webbrowser** for opening the default web browser, and **json** for JSON manipulation.
2. Open the **index.html** file in the default web browser.
3. Create a Flask application instance.
4. Define a global variable **name** to store the logged-in user's name.
5. Implement the **/register** route to handle POST requests for user registration. It retrieves the registration data from the request form, sends a POST request to the registration microservice, and returns a success message or an error message to the client.
6. Implement the **/login** route to handle POST requests for user login. It retrieves the login credentials from the request form, sends a POST request to the login microservice, and redirects the user to the dashboard page if the login is successful. Otherwise, it renders the **index.html** template with an error message.
7. Implement the **/dashboard** route to handle GET requests for displaying the dashboard page. It sends a GET request to the online users microservice, retrieves the online user data, and renders the **dashboard.html** template with the online users and the logged-in user's name.
8. Implement the **/logout** route to handle POST requests for user logout. It sends a POST request to the logout microservice, updates the **name** variable, and returns a success message to the client.
9. Implement the **/** route to handle GET requests for the home page. It renders the **index.html** template.
10. Build the image of our webapp push to artifact registry.

Program Instructions:

1. After pushing all these images to Artifact Registry we will create Services in cloud run.

2. Note: While creating services in Cloud run to check Unauthorized Access check mark and change the port to the one given in the above code.

3. After all the services are built, we can open our webapp and start testing the services. If there are any errors, we can go to that service and check the logs.

Commands used for building, tagging, and pushing docker images:

1. docker build -t con-1 .
   docker tag con-1 us-central1-docker.pkg.dev/serverless-a2-390516/c1/con-1:latest
   docker push us-central1-docker.pkg.dev/serverless-a2-390516/c1/con-1:latest
2. docker build -t con-2 .

docker tag con-2 us-central1-docker.pkg.dev/serverless-a2-390516/c2/con-2:latest
docker push us-central1-docker.pkg.dev/serverless-a2-390516/c2/con-2:latest

3. docker build -t con-3 .
   docker tag con-3 us-central1-docker.pkg.dev/serverless-a2-390516/c3/con-3:latest
   docker push us-central1-docker.pkg.dev/serverless-a2-390516/c3/con-3:latest
4. docker build -t webapp .
   docker tag webapp us-central1-docker.pkg.dev/serverless-a2-390516/mywebsite/webapp:latest
   docker push us-central1-docker.pkg.dev/serverless-a2-390516/mywebsite/webapp:latest

Summary of Google Cloud Run, Artifact registry, Docker Container:

1. Google Cloud Run: Google Cloud Run is a serverless compute platform that allows you to run stateless containers in a managed environment. In this application, the microservices were deployed using Google Cloud Run [3]. It provides automatic scaling and handles the underlying infrastructure, allowing developers to focus on writing code [3].
2. Artifact Registry: Artifact Registry is a fully managed Docker container registry that provides a private, secure, and scalable storage solution for container images and other artifacts [1]. In this application, the container images built for the microservices were pushed to the Artifact Registry repository [1].
3. Docker Container: Docker containers were used to package and deploy microservices. Each microservice was containerized, which means it was isolated in its own lightweight container along with its dependencies [2]. Docker allows for consistent deployment across different environments and simplifies the management of dependencies [2].

The application consisted of three microservices, each responsible for specific backend logic. The microservices communicated with Firestore, a NoSQL document database provided by Google Cloud Platform [4].

- Container #1: Responsible for accepting registration details from the front end and storing them in the Firestore database. It received the registration data (Name, Password, Email, Location) and stored it in the "Reg" collection in Firestore.
- Container #2: Responsible for validating login information by checking it against the values in the Firestore database. Once a user logged in successfully, their

state was updated to "online" in the Firestore database. This microservice verified the login information and managed the user's authentication.

- Container #3: Responsible for extracting state information from the Firestore database, such as who is currently online. It maintained the user session from login to logout. When a user clicked on logout, the session expired, and the state item in the Firestore database was updated accordingly.
- Container #4: Responsible for interacting with the above backend microservices for registration, login, and state management.

To test the application, test cases were written to cover various scenarios and functionalities. Screenshots were provided as evidence for each step, demonstrating the successful execution of the application.

Overall, Google Cloud Run was used to deploy the containerized microservices, GCR/Artifact Registry served as the repository for storing the container images, and Docker containers were utilized for packaging and managing the microservices. These technologies together provided a scalable, managed environment for running the backend logic of the application and interacting with the Firestore database.