



**DALHOUSIE
UNIVERSITY**

CSCI 5410

Serverless Data Processing (Summer 2023)

MASTER OF APPLIED COMPUTER
SCIENCE

Assignment-2 (Part B): Containerized Application

Name: **Jainil Sevalia** | Banner Id : **B00925445** | Email:jn498899@dal.ca

GitLab Link : https://git.cs.dal.ca/sevalia/csci5410-summer-23-b00925445/-/tree/A2?ref_type=heads

Table of Figures:

Figure 1 : Reg collection created on Fire store.....	3
Figure 2: state collection created on fire store.....	3
Figure 3 : Application business logic Code for container -1.....	5
Figure 4 : built docker image and tagged it	6
Figure 5 : Docker image pushed on GCP Artifact Registry.....	6
Figure 6 : Artifact Registry console GCP.....	7
Figure 7 : Container-1 Running on Cloud Run service.....	7
Figure 8 : Postman Sending request to container to store the User Data.....	8
Figure 9 : Container-1 running successfully on GCP and getting all the data in the Firestore.....	8
Figure 10 : Null email edge case for registration.....	9
Figure 11 : Invalid form of email handling.....	9
Figure 12: Already email registered email handling.....	10
Figure 13 : Test cases for Container-1.	11
Figure 14 : Application business logic Code for container -2.....	13
Figure 15 : Build docker image for container 2.....	14
Figure 16 : Docker image tagged with project name to push on Artifact Registry.....	14
Figure 17 : Docker Image build and pushed for container-2	15
Figure 18: Artifact Registry GCP console.	15
Figure 19 : Creating service for container 2.....	16
Figure 20 : Container 2 is running on Cloud Run service.....	16
Figure 21: Postman Sending request to container to validate the User Data.....	17
Figure 22 : User status updated in firestore.	17
Figure 23 : Null credential field passed in request payload.....	18
Figure 24 : non-Registered user tried to login.	18
Figure 25 : Test Cases for container 2.....	19
Figure 26: Application business logic Code for container -3 (Session API).....	21
Figure 27: Logout API endpoint in container 3.	22
Figure 28 : Build image for container 3.....	22
Figure 29: Docker image tagged for container 3.	23
Figure 30 : Docker image build and pushed on the GCP artifact registry.....	23
Figure 31: Artifact Registry GCP console.	24
Figure 32 : Creating Cloud Run Service for Container-3.	24
Figure 33: Container-3 Running on Cloud Run service.....	25
Figure 34 : GET request to get all online users.....	25
Figure 35 : GET request to Logout user.....	26
Figure 36: Logout will update the user status to offline in state collection.	26
Figure 37 : Test Cases for container 3.....	27
Figure 38 : Docker Image building of Frontend.	28
Figure 39 : Docker image tagged for frontend container.	28
Figure 40 : Frontend Image Pushed to Artifact Registry.....	29
Figure 41 : Artifact Registry - Fronted Image.....	29
Figure 42 : Creating Cloud Run service for Frontend Container.	30
Figure 43: Frontend Running on Cloud Run.	30
Figure 44: User Registration page(Build using React).	31
Figure 45 : User Document Successfully created in Firestore.....	31
Figure 46 : Login Page.....	32
Figure 47 : Online User List after login.....	32

B. Your database should contain only 2 collections. One collection is “Reg” to contain registration data (Name, Password, Email, Location), another collection is “state” to contain user state (online, offline, timestamp etc.) information.

The screenshot shows the Firebase Cloud Firestore interface. On the left, the navigation bar includes 'Log X', 'Images', 'Services', 'Data -', 'Build N...', 'Get dalli', 'Push all', 'Assign', 'Querying', 'java -', 'Array -', 'localhost', 'React /', 'Logs E...', 'Write...', and a '+' button. Below this is the 'Project Overview' section with 'Firestore Database' selected. The main area shows the 'Reg' collection under the 'login-microservice' database. A single document for 'Test@gmail.com' is listed, containing the following fields:

- email: "Test@gmail.com"
- location: "Halifax, Canada"
- name: "test"
- password: "Test@123"

Figure 1 : Reg collection created on Fire store.

The screenshot shows the Firebase Cloud Firestore interface, similar to Figure 1. The 'Reg' collection is visible, and the 'state' collection is now the active one. It contains one document for 'Test@gmail.com' with the following fields:

- status: "Online"
- timestamp: 25 June 2023 at 20:02:20 UTC-3

Figure 2: state collection created on fire store.

C. Code and the required dependencies of Container #1 are responsible for accepting registration details from frontend and store it in backend database.

Container – 1:

- Business logic/ Function for Registration:

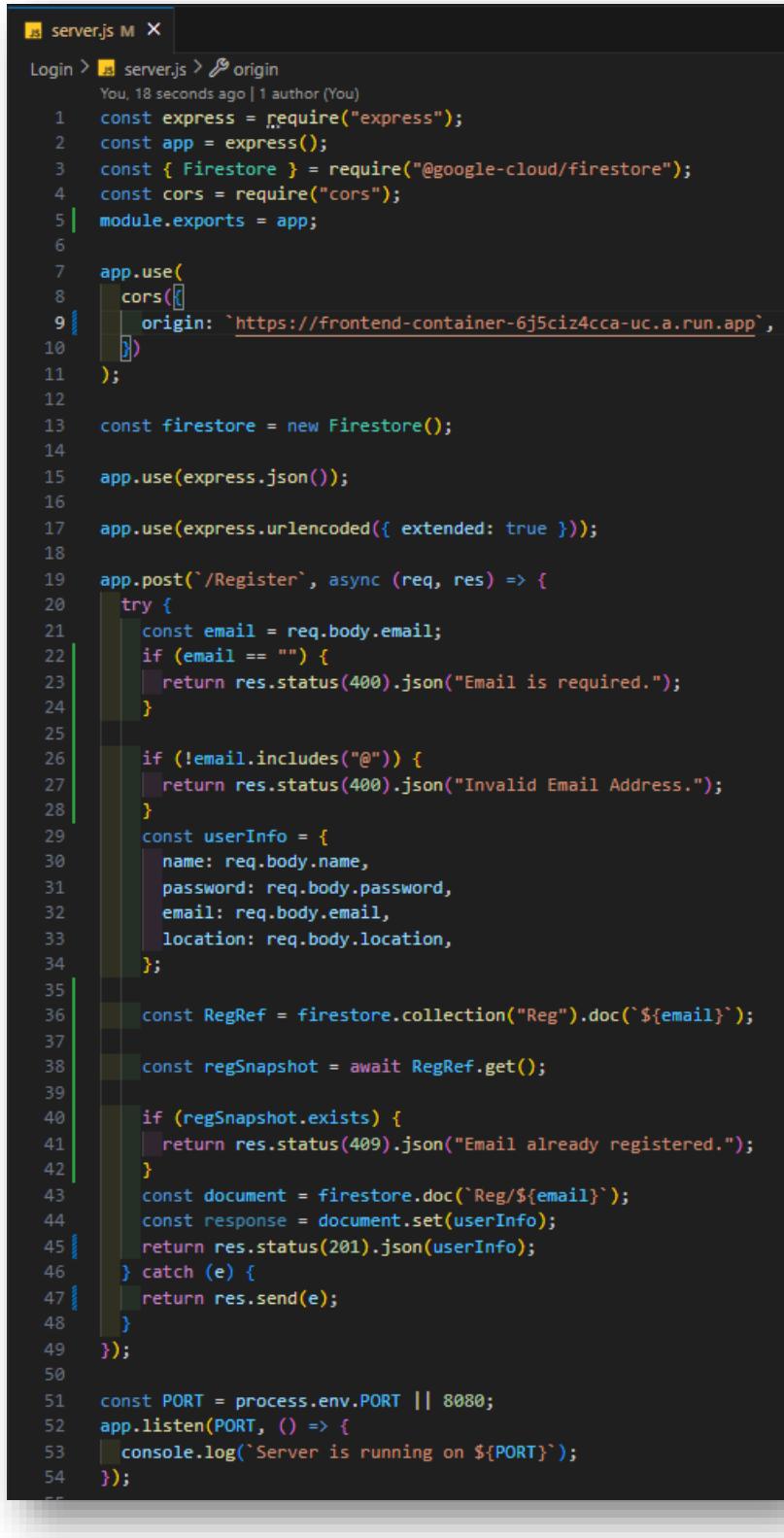
```
app.post(`/Register`, async (req, res) => {
  try {
    const email = req.body.email;
    if (email == "") {
      return res.status(400).json("Email is required.");
    }

    if (!email.includes("@")) {
      return res.status(400).json("Invalid Email Address.");
    }
    const userInfo = {
      name: req.body.name,
      password: req.body.password,
      email: req.body.email,
      location: req.body.location,
    };

    const RegRef = firestore.collection("Reg").doc(` ${email} `);

    const regSnapshot = await RegRef.get();

    if (regSnapshot.exists) {
      return res.status(409).json("Email already registered.");
    }
    const document = firestore.doc(`Reg/${email}`);
    const response = document.set(userInfo);
    return res.status(201).json(userInfo);
  } catch (e) {
    return res.send(e);
  }
});
```



The screenshot shows a code editor window with the file 'server.js' open. The code is written in JavaScript and contains logic for a registration endpoint. It uses Express.js, Firestore, and CORS. The code includes validation for email format and uniqueness, and it logs the server's port.

```
server.js M X
Login > server.js > origin
You, 18 seconds ago | 1 author (You)
1 const express = require("express");
2 const app = express();
3 const { Firestore } = require("@google-cloud/firestore");
4 const cors = require("cors");
5 module.exports = app;
6
7 app.use(
8   cors({
9     origin: `https://frontend-container-6j5ciz4cca-uc.a.run.app`,
10   })
11 );
12
13 const firestore = new Firestore();
14
15 app.use(express.json());
16
17 app.use(express.urlencoded({ extended: true }));
18
19 app.post(`/Register`, async (req, res) => {
20   try {
21     const email = req.body.email;
22     if (email == "") {
23       return res.status(400).json("Email is required.");
24     }
25
26     if (!email.includes("@")) {
27       return res.status(400).json("Invalid Email Address.");
28     }
29     const userInfo = {
30       name: req.body.name,
31       password: req.body.password,
32       email: req.body.email,
33       location: req.body.location,
34     };
35
36     const RegRef = firestore.collection("Reg").doc(` ${email}`);
37
38     const regSnapshot = await RegRef.get();
39
40     if (regSnapshot.exists) {
41       return res.status(409).json("Email already registered.");
42     }
43     const document = firestore.doc(`Reg/${email}`);
44     const response = document.set(userInfo);
45     return res.status(201).json(userInfo);
46   } catch (e) {
47     return res.send(e);
48   }
49 });
50
51 const PORT = process.env.PORT || 8080;
52 app.listen(PORT, () => {
53   console.log(`Server is running on ${PORT}`);
54 });
55
```

Figure 3 : Application business logic Code for container -1

- Build docker image and tagged it with specific name before pushing it on artifact registry.

The screenshot shows the Visual Studio Code interface. The left sidebar displays a file tree for a project named 'LOGIN_CONTAINERIZED_MICROSERVICES'. The 'Dockerfile' file is open in the editor, showing the following content:

```

FROM node:alpine
WORKDIR /app
COPY server.js package.json ./
RUN npm install
CMD ["npm", "start"]

```

The terminal tab at the bottom shows the command 'docker build . -t container-1' being run, with the output indicating a successful build:

```

[+] Building 33.9s (9/9) FINISHED
   => [internal] load build definition from Dockerfile
   => [internal] load .dockerignore
   => [internal] transfer context: 2B
   => [internal] load metadata for docker.io/library/node:alpine
   => [internal] load build context
   => transferring context: 1.93kB
   => CACHED [2/4] WORKDIR /app
   => [3/4] COPY server.js package.json ./
   => [4/4] RUN npm install
   => exporting layers
   => writing image sha256:064945d0adfa2ff89330b33a94740a1dbe048af624101f15b045944d2f07e1
--> naming to docker.io/library/container-1

```

The status bar at the bottom right indicates the Docker extension is active.

Figure 4 : built docker image and tagged it.

- Docker image pushed using command mentioned in below image.

The screenshot shows the Visual Studio Code interface with the same project structure and Dockerfile content as Figure 4. The terminal tab shows the command 'docker push us-east1-docker.pkg.dev/login-microservice/login-microservices/container-1' being run, with the output showing the image being pushed to Google Cloud Registry:

```

[+] Building 12.39s (9/9) FINISHED
   => [internal] load build definition from Dockerfile
   => [internal] load .dockerignore
   => [internal] transfer context: 2B
   => [internal] load metadata for docker.io/library/node:alpine
   => [internal] load build context
   => transferring context: 1.93kB
   => CACHED [2/4] WORKDIR /app
   => [3/4] COPY server.js package.json ./
   => [4/4] RUN npm install
   => exporting layers
   => writing image sha256:064945d0adfa2ff89330b33a94740a1dbe048af624101f15b045944d2f07e1
--> naming to docker.io/library/container-1
latest: digest: sha256:89f793cf65995c5bfecfb90e275b242273b2e7cd03d03d4101aafb4779fe8 size: 1783

```

The status bar at the bottom right indicates the Docker extension is active.

Figure 5 : Docker image pushed on GCP Artifact Registry.

- GCP Artifact Registry shows the pushed image successfully.

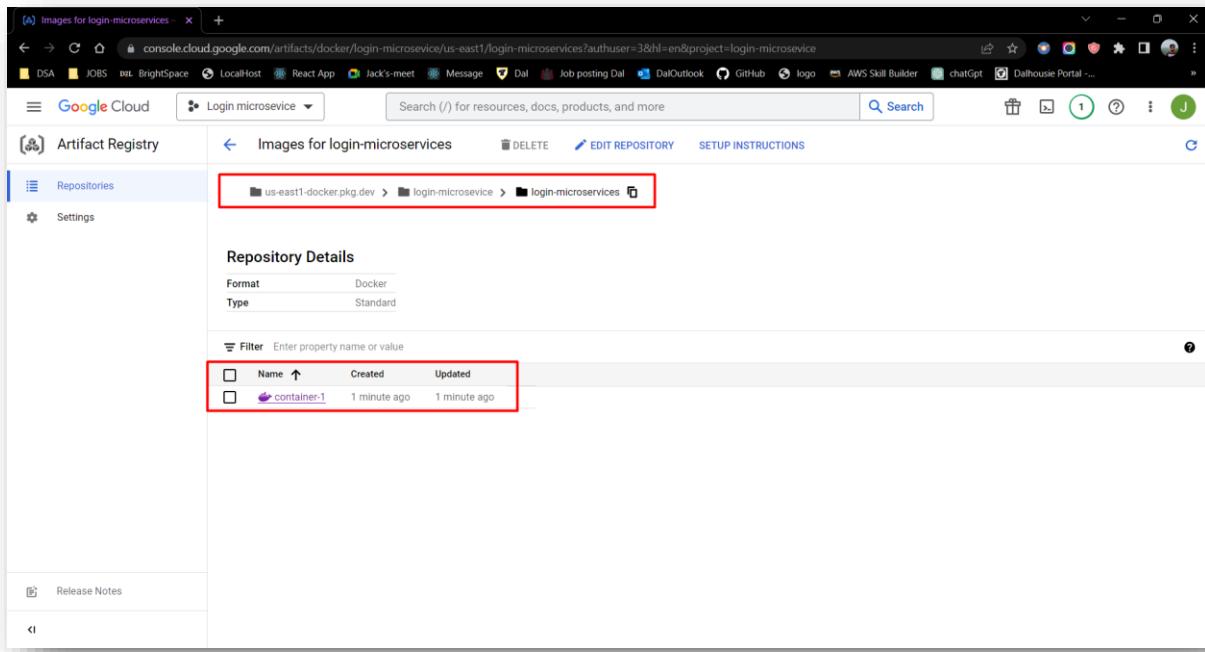


Figure 6 : Artifact Registry console GCP.

- Created New Service in Cloud Run to Run the container 1.

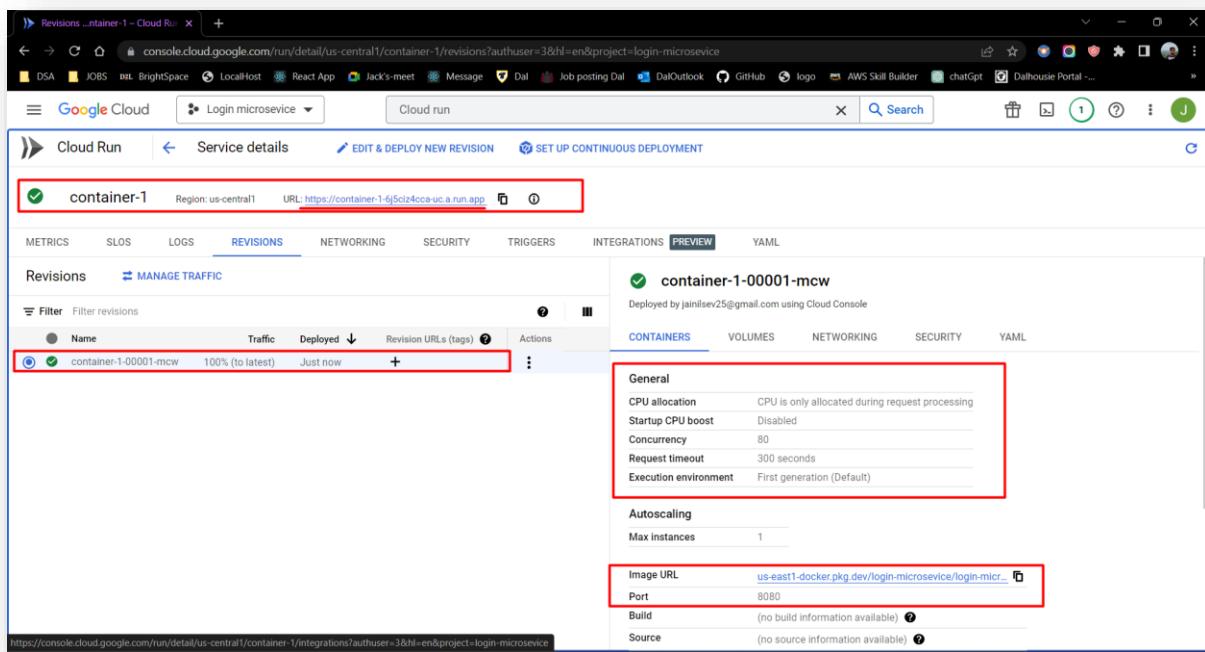


Figure 7 : Container-1 Running on Cloud Run service.

- Testing the running container by sending POST request from postman to that container.

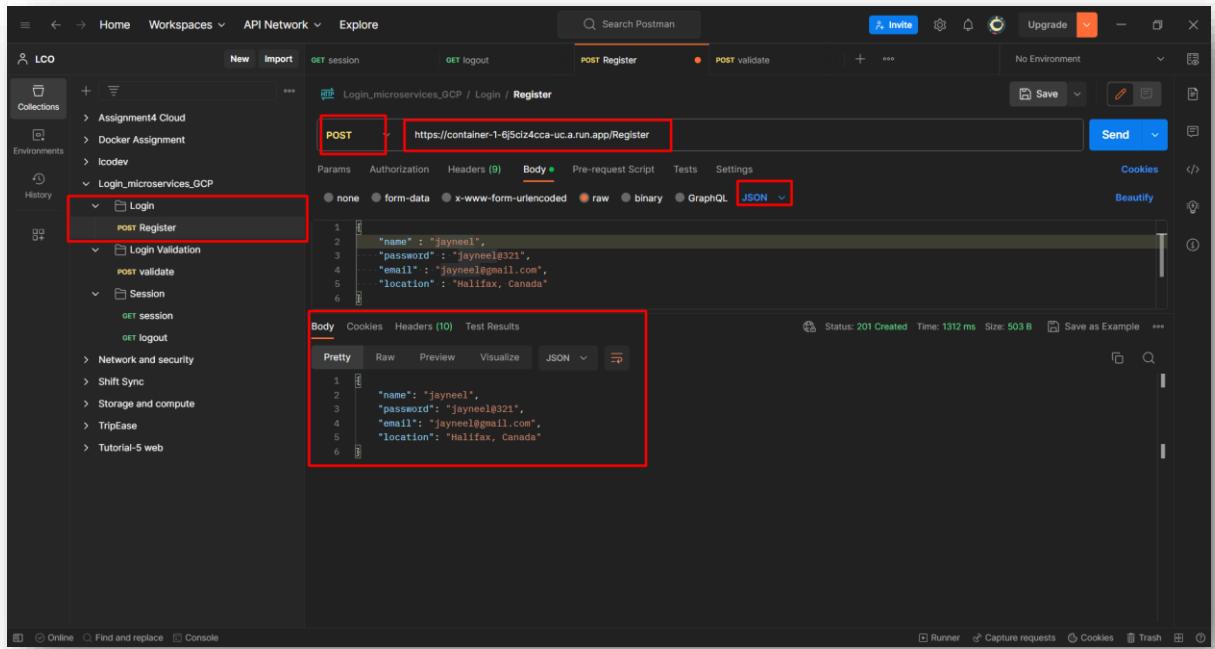


Figure 8 : Postman Sending request to container to store the User Data.

- Successfully got data in firestore.

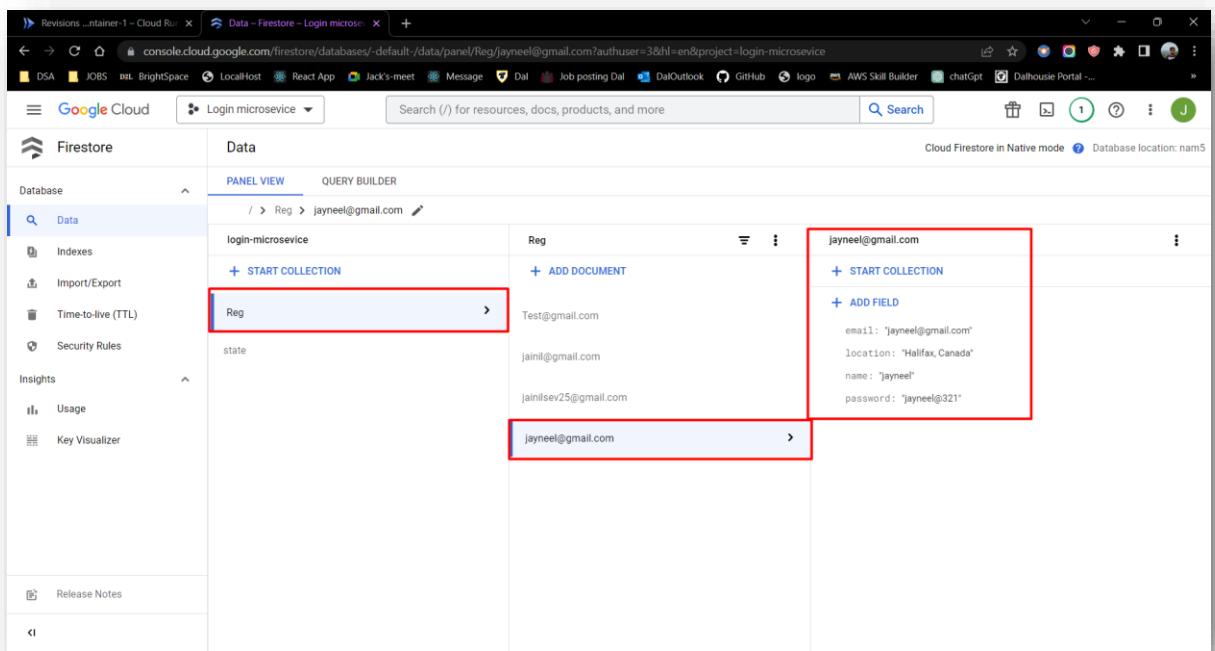


Figure 9 : Container-1 running successfully on GCP and getting all the data in the Firestore.

- Handled Edge cases for Registration microservices.
- Null email handling.

The screenshot shows the Postman interface with a collection named 'Login_microservices_GCP'. A red box highlights the 'post Register' request under the 'Login' folder. The request URL is 'https://container-1-6j5clz4cca-uc.a.run.app/Register'. The 'Body' tab is selected, showing a JSON payload with fields: name, password, location, and email (which is empty). The 'Test Results' section shows a response status of 400 Bad Request with the message 'Email is required.' highlighted in a red box.

Figure 10 : Null email edge case for registration.

- Invalid form of email handling.

The screenshot shows the Postman interface with the same collection 'Login_microservices_GCP'. A red box highlights the 'post Register' request under the 'Login' folder. The request URL is 'https://container-1-6j5clz4cca-uc.a.run.app/Register'. The 'Body' tab is selected, showing a JSON payload with fields: name, password, location, and email set to 'jayneel@gmail.com'. The 'Test Results' section shows a response status of 400 Bad Request with the message 'Invalid Email Address.' highlighted in a red box.

Figure 11 : Invalid form of email handling.

- Already registered email handling.

The screenshot shows the Postman application interface. On the left, the 'Collections' sidebar is open, showing a collection named 'Login_microservices_GCP'. Within this collection, a 'post Register' item is selected and highlighted with a red box. The main workspace shows a POST request to 'https://container-1-6j5clz4cca-uc.a.run.app/Register'. The 'Body' tab is selected, displaying a JSON payload:

```
1 {  
2   "name": "jayneel",  
3   "password": "jayneel8321",  
4   "email": "jayneel@gmail.com",  
5   "location": "Halifax, Canada"  
6 }
```

The response status is '409 Conflict' with the message 'Email already registered.' highlighted with a red box in the 'Test Results' section.

Figure 12: Already email registered email handling.

H (For container 1). Test Cases for /Registration endpoint.

- Successfully registered user.
 - Empty email. (Email is required).
 - Email validation (Invalid email).
 - User already exists with email. (Email already registered).

The screenshot shows a Visual Studio Code interface with two tabs open:

- server.js**: Contains server-side logic for handling registration requests. It includes validation for required fields like email and password, and checks if the email is already registered in a Firestore database.
- registration.test.js**: A unit test for the registration endpoint. It uses the `request` module to simulate POST requests to the '/Register' endpoint and assert the expected responses based on different input scenarios.

The status bar at the bottom displays the file path: "File: Login_Containerized_microservices.GCP - Visual Studio Code".

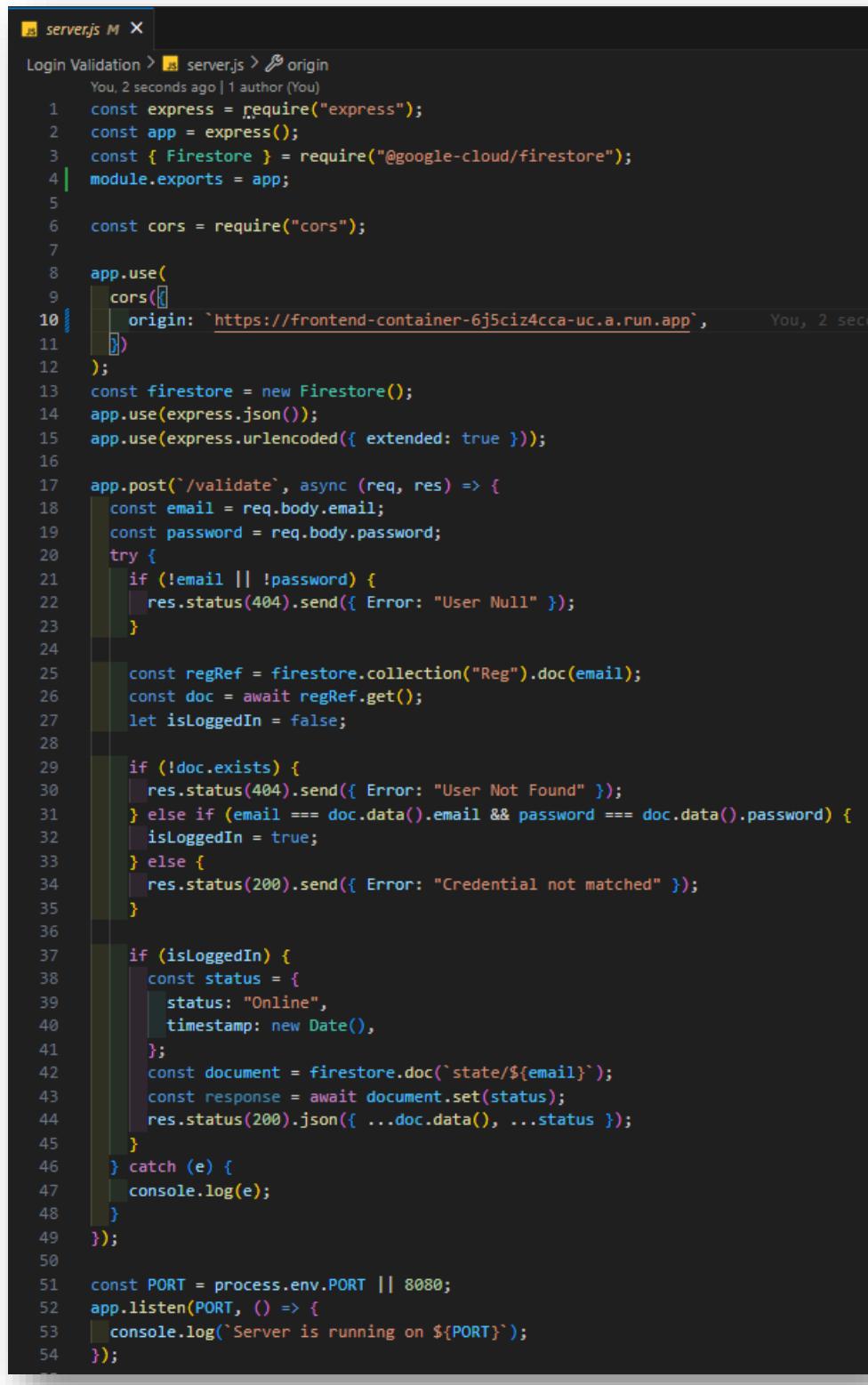
Figure 13 : Test cases for Container-1.

D. Container – 2:

- Business logic/ Function for Login:

```
app.post(`/validate`, async (req, res) => {
  const email = req.body.email;
  const password = req.body.password;
  try {
    if (!email || !password) {
      res.status(404).send({ Error: "User Null" });
    }
    const regRef = firestore.collection("Reg").doc(email);
    const doc = await regRef.get();
    let isLoggedIn = false;
    if (!doc.exists) {
      res.status(404).send({ Error: "User Not Found" });
    } else if (email === doc.data().email && password ===
    doc.data().password) {
      isLoggedIn = true;
    } else {
      res.status(200).send({ Error: "Credential not matched" });
    }
    if (isLoggedIn) {
      const status = {
        status: "Online",
        timestamp: new Date(),
      };
      const document = firestore.doc(`state/${email}`);
      const response = await document.set(status);
      res.status(200).json({ ...doc.data(), ...status });
    } } catch (e) {
  }});
});
```

- Application business logic Code for container -2



```
server.js M X
Login Validation > server.js > origin
You, 2 seconds ago | 1 author (You)
1 const express = require("express");
2 const app = express();
3 const { Firestore } = require("@google-cloud/firestore");
4 module.exports = app;
5
6 const cors = require("cors");
7
8 app.use(
9   cors({
10     origin: `https://frontend-container-6j5ciz4cca-uc.a.run.app`,
11   })
12 );
13 const firestore = new Firestore();
14 app.use(express.json());
15 app.use(express.urlencoded({ extended: true }));
16
17 app.post('/validate', async (req, res) => {
18   const email = req.body.email;
19   const password = req.body.password;
20   try {
21     if (!email || !password) {
22       res.status(404).send({ Error: "User Null" });
23     }
24
25     const regRef = firestore.collection("Reg").doc(email);
26     const doc = await regRef.get();
27     let isLoggedIn = false;
28
29     if (!doc.exists) {
30       res.status(404).send({ Error: "User Not Found" });
31     } else if (email === doc.data().email && password === doc.data().password) {
32       isLoggedIn = true;
33     } else {
34       res.status(200).send({ Error: "Credential not matched" });
35     }
36
37     if (isLoggedIn) {
38       const status = {
39         status: "Online",
40         timestamp: new Date(),
41       };
42       const document = firestore.doc(`state/${email}`);
43       const response = await document.set(status);
44       res.status(200).json({ ...doc.data(), ...status });
45     }
46   } catch (e) {
47     console.log(e);
48   }
49 });
50
51 const PORT = process.env.PORT || 8080;
52 app.listen(PORT, () => {
53   console.log(`Server is running on ${PORT}`);
54 });
55
```

Figure 14 : Application business logic Code for container -2.

- Build docker image and tagged it with specific name before pushing it on artifact registry.

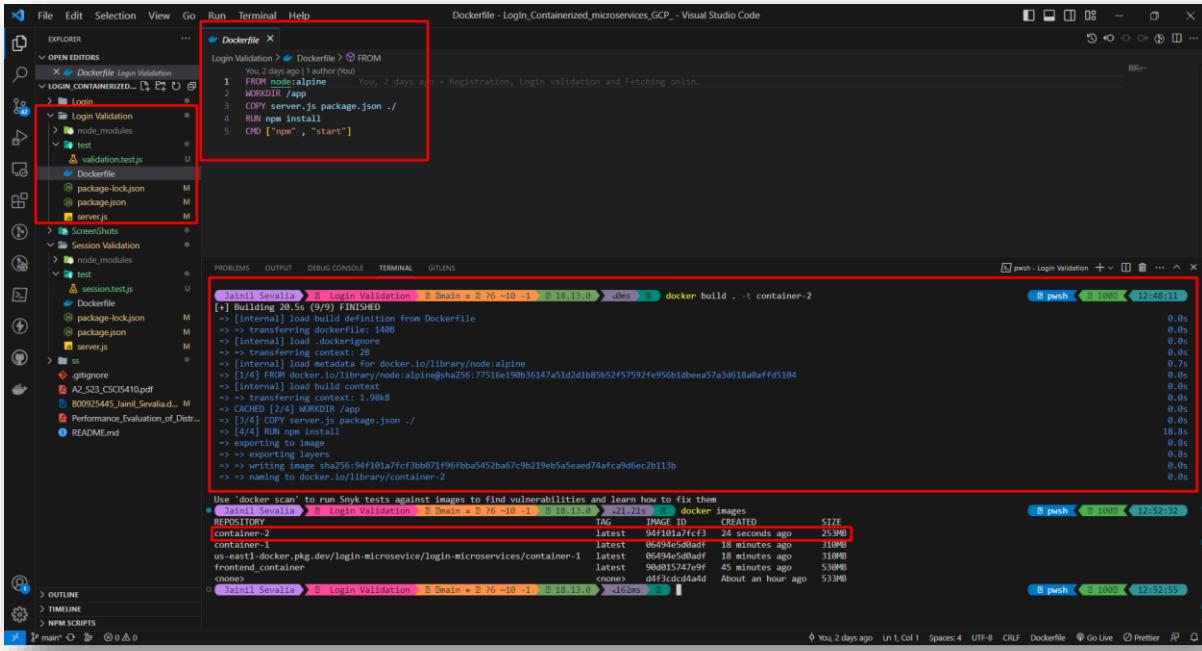


Figure 15 : Build docker image for container 2.

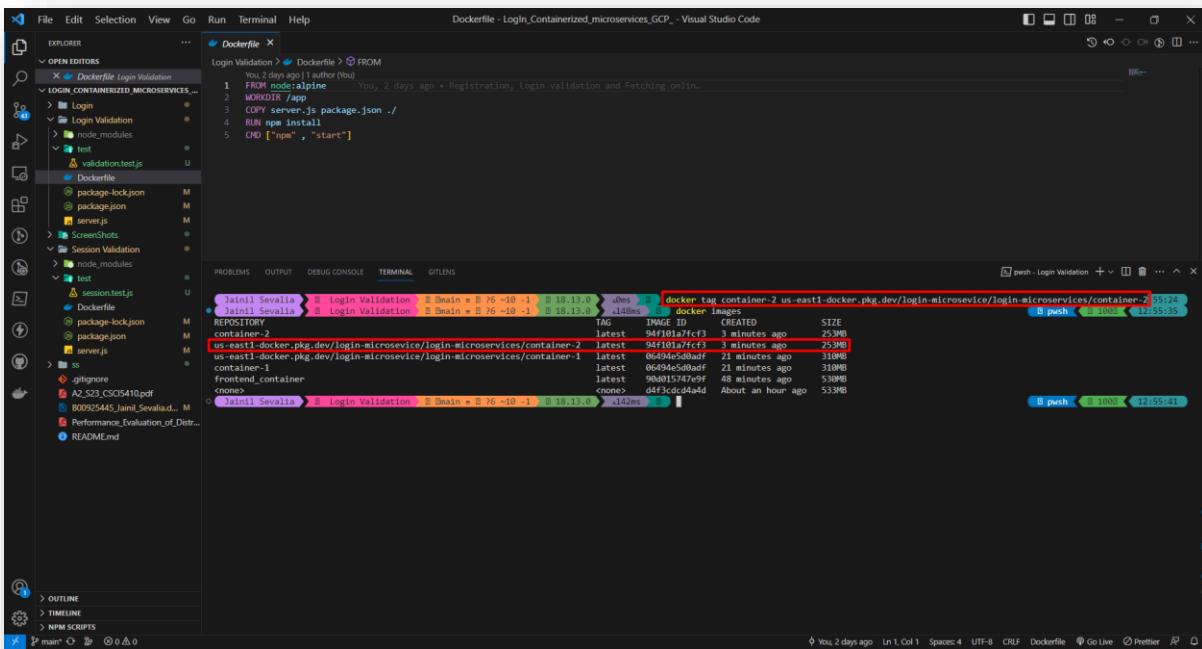


Figure 16 : Docker image tagged with project name to push on Artifact Registry.

- Docker image pushed using command mentioned in below image.

The screenshot shows the Visual Studio Code interface with the Dockerfile open in the editor. The terminal window displays the command to tag and push the Docker image:

```
docker tag container-2 us-east1-docker.pkg.dev/login-microservice/container-2:latest
docker push us-east1-docker.pkg.dev/login-microservice/container-2
```

The terminal output shows the image was tagged and pushed successfully:

```
Using default tag: latest
The push refers to repository [us-east1-docker.pkg.dev/login-microservice/container-2]
719870165752: Pushed
9164370fbafc: Layer already exists
6a615ae4a03: Layer already exists
8e0f3a0a03: Layer already exists
8879c9f177e: Layer already exists
78a82f2fa2dd: Layer already exists
latest: digest: sha256:cba5d742febf618ac75826da2ae234d5df6998a409b9d1530002fb846aaa36 size: 1784
Dockerfile: Login Validation: 1m 10s ago
  1. FROM node:10-alpine
  2. WORKDIR /app
  3. COPY server.js package.json .
  4. RUN npm install
  5. CMD ["npm", "start"]
```

Figure 17 : Docker Image build and pushed for container-2

- GCP Artifact Registry shows the pushed image successfully.

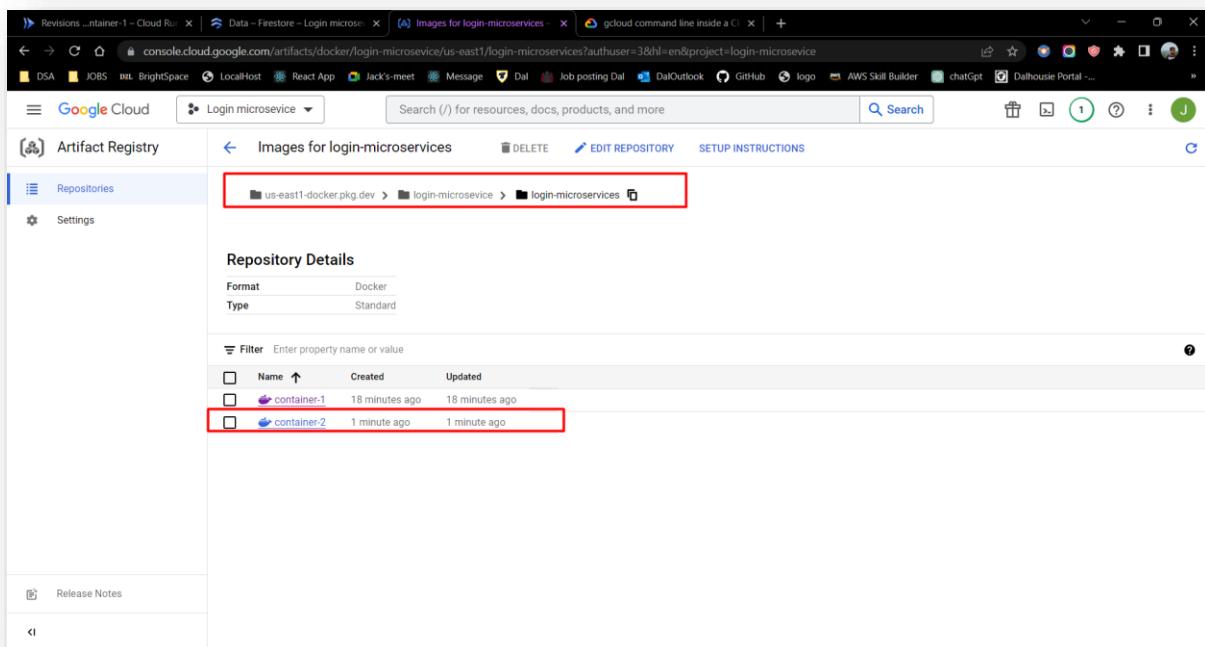


Figure 18: Artifact Registry GCP console.

- Creating service for container 2 on Cloud Run.

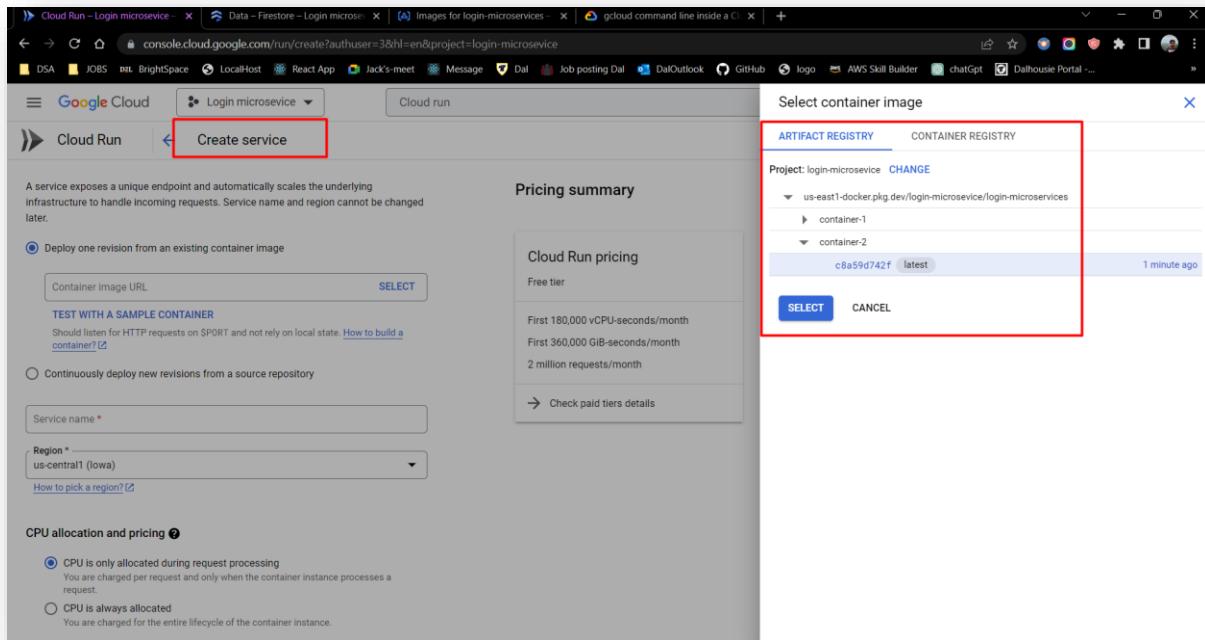


Figure 19 : Creating service for container 2.

- Container is running on the GCP cloud run service, which is shown below.

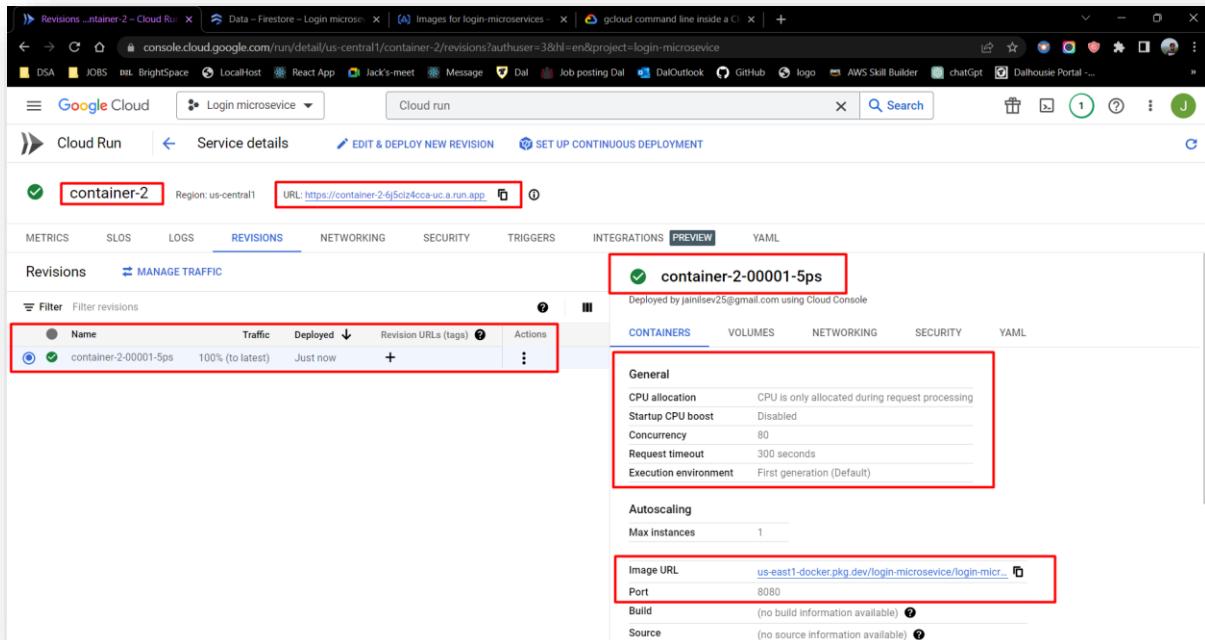


Figure 20 : Container 2 is running on Cloud Run service.

- Testing the container by sending request to container using postman.

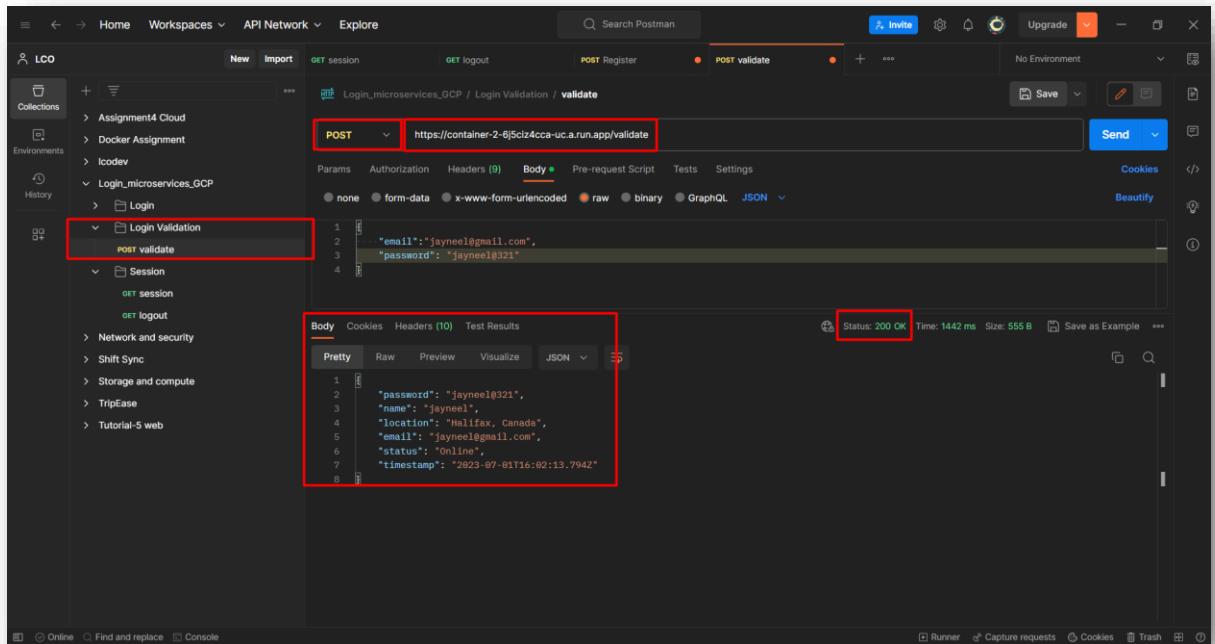


Figure 21: Postman Sending request to container to validate the User Data.

- User Status updated in firestore.

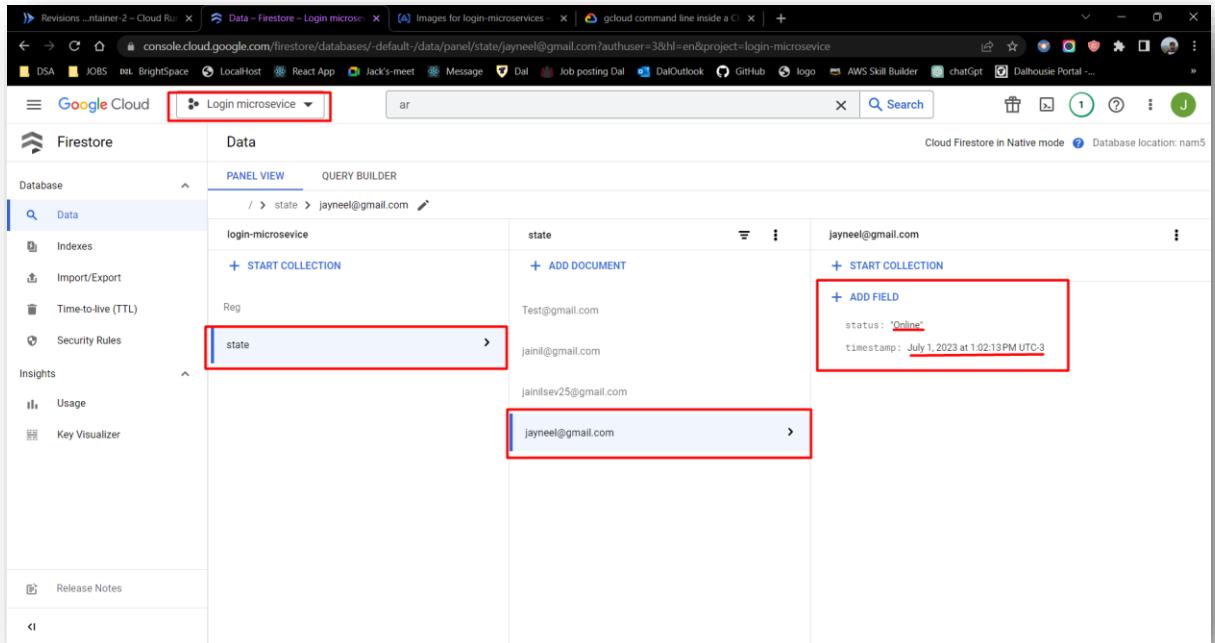


Figure 22 : User status updated in firestore.

- Edge cases handling.
- Null credential field passed in request payload.

The screenshot shows the Postman interface with a collection named 'LCO'. A POST request is selected with the URL `https://container-2-65ciz4cca-uc.a.run.app/validate`. The 'Body' tab is active, showing a JSON object with an empty 'password' field. The response status is 404 Not Found, with the error message "Error": "User Null".

Figure 23 : Null credential field passed in request payload.

- User tried to login using not registered user.

The screenshot shows the Postman interface with a collection named 'LCO'. A POST request is selected with the URL `https://container-2-65ciz4cca-uc.a.run.app/validate`. The 'Body' tab is active, showing a JSON object with a fake email ('fakeUser@gmail.com') and a valid password ('jayneel8321'). The response status is 404 Not Found, with the error message "Error": "User Not Found".

Figure 24 : non-Registered user tried to login.

H (For Container -2). Test Cases for /Login endpoint.

- Successfully Logged In user.
- Empty Credentials. (Email and Password is required).
- Valid Credentials (Incorrect Email or Password).
- Non-existing user. (Invalid Email).

```
validation.test.js - Login_Containerized_microservices_GCP - Visual Studio Code
```

```
server.js M
```

```
validation.test.js
```

```
File Edit Selection View Go Run Terminal Help
```

```
validation.test.js
```

```
server.js
```

```
app.post('/validate', async (req, res) => {
  const email = req.body.email;
  const password = req.body.password;
  try {
    if (!email || !password) {
      res.status(404).send({ Error: "User Null" });
    }
  } catch (e) {
    console.log(e);
  }
  const regRef = firestore.collection("Reg").doc(email);
  const doc = await regRef.get();
  let isLoggedInIn = false;
  if (!doc.exists) {
    res.status(404).send({ Error: "User Not Found" });
  } else if (email === doc.data().email && password === doc.data().password) {
    isLoggedInIn = true;
  } else {
    res.status(200).send({ Error: "Credential not matched" });
  }
  if (isLoggedInIn) {
    const status = {
      status: "Online",
      timestamp: new Date(),
    };
    const document = firestore.doc(`state/${email}`);
    const response = await document.set(status);
    res.status(200).json({ ...doc.data(), ...status });
  }
} catch (e) {
  console.log(e);
}
});
```

```
it("should return an error for incorrect credentials", (done) => {
  request(app)
    .post('/validate')
    .send({
      email: "johndoe@example.com",
      password: "wrongpassword",
    })
    .expect(200)
    .end((err, res) => {
      if (err) return done(err);
      assert.deepStrictEqual(res.body, {
        Error: "Credential not matched",
      });
    });
});

it("should return an error for a non-existing user", (done) => {
  request(app)
    .post('/validate')
    .send({
      email: "nonexisting@example.com",
      password: "123456",
    })
    .expect(404)
    .end((err, res) => {
      if (err) return done(err);
      assert.deepStrictEqual(res.body, {
        Error: "User Not Found",
      });
    });
});
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
```

```
4 passing (2s)
```

```
Jainil-Sevalia [2]: login.js [2]
```

```
File 22 → 23 : 13:13:3 27
```

```
Ln 79, Col 1 Spaces: 2 UTF-8 CRLF {} Go Live ⌂ Prettier ⌂
```

Figure 25 : Test Cases for container 2.

E. Container – 3:

- Business logic/ Function for Session and logout:

```
app.get(`/session`, async (req, res) => {
  const stateRef = firestore.collection("state");

  const allOnlineUsers = await stateRef.where("status", "==",
"Online").get();
  let userArray = [];
  let response = [];

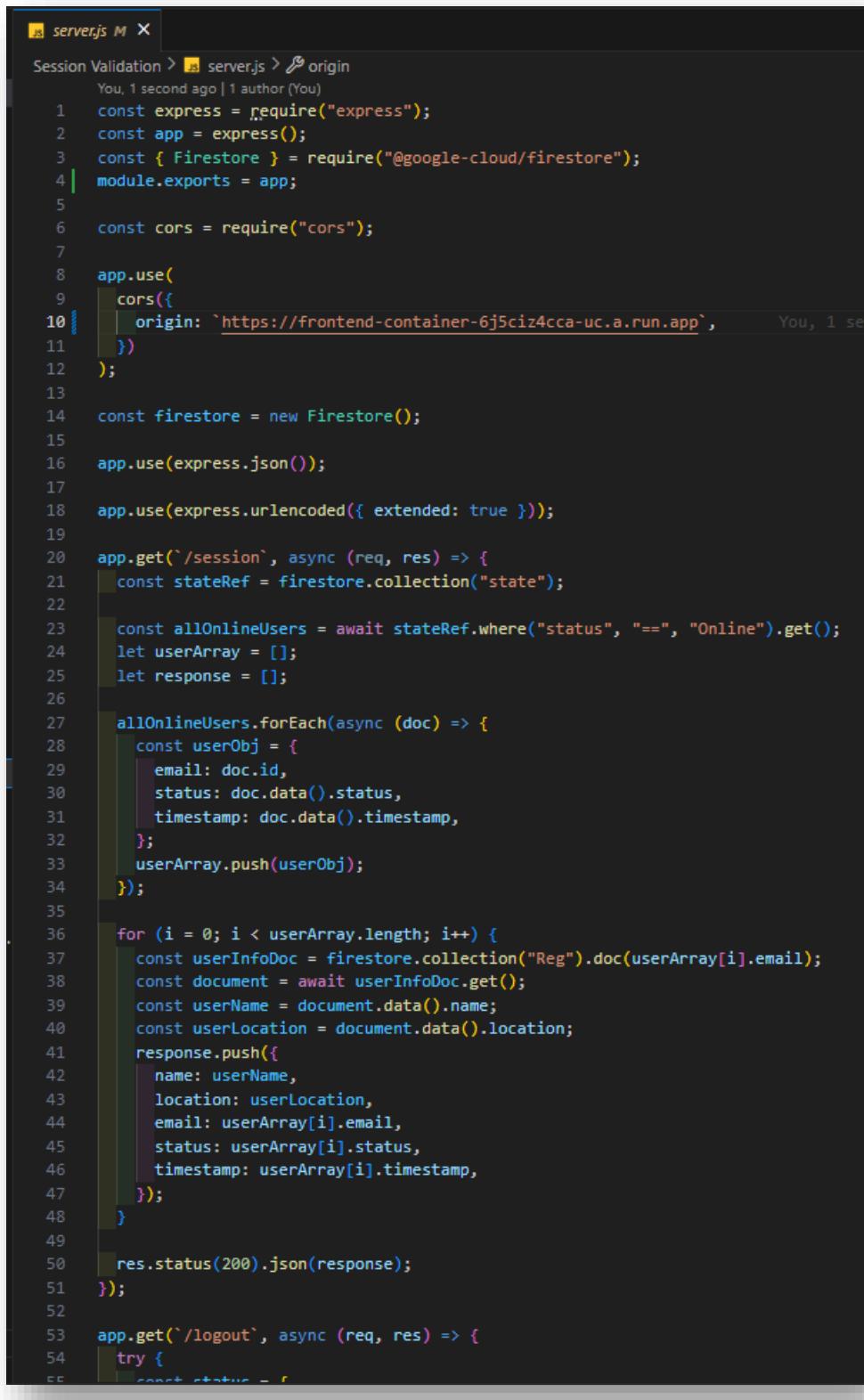
  allOnlineUsers.forEach(async (doc) => {
    const userObj = {
      email: doc.id,
      status: doc.data().status,
      timestamp: doc.data().timestamp,
    };
    userArray.push(userObj);
  });

  for (i = 0; i < userArray.length; i++) {
    const userInfoDoc =
firestore.collection("Reg").doc(userArray[i].email);
    const document = await userInfoDoc.get();
    const userName = document.data().name;
    const userLocation = document.data().location;
    response.push({
      name: userName,
      location: userLocation,
      email: userArray[i].email,
      status: userArray[i].status,
      timestamp: userArray[i].timestamp,
    });
  }

  res.status(200).json(response);
});

app.get(`/logout`, async (req, res) => {
  try {
    const status = {
      status: "Offline",
      timestamp: new Date(),
    };
    const email = req.query.email;
    const stateRef = firestore.collection("state").doc(email);
    const doc = await stateRef.set(status);
    res.sendStatus(200);
  } catch (e) {}});
});
```

- Application business logic Code for container -3

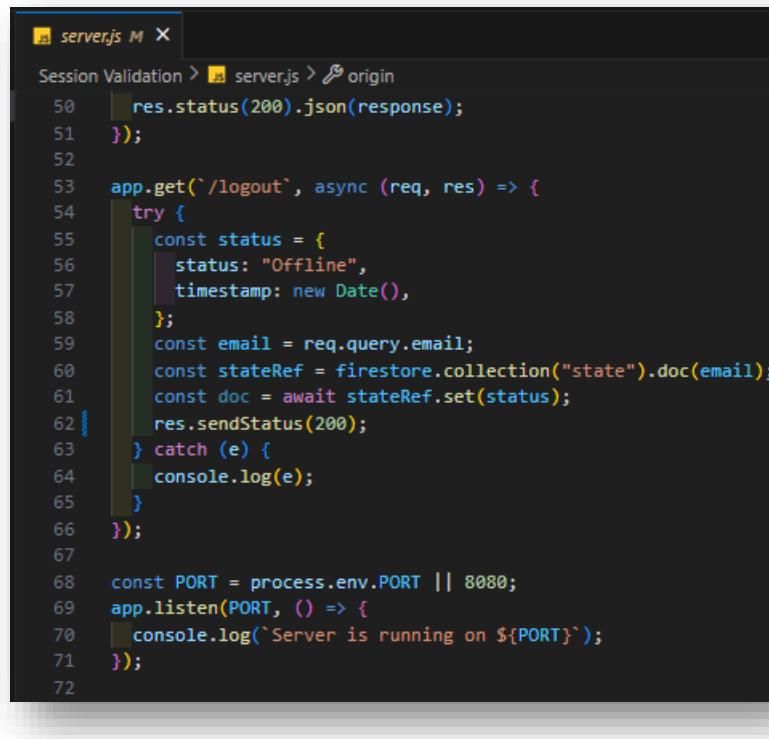


The screenshot shows a code editor window with a dark theme. The file is named `server.js`. The code is a Node.js application using Express and Firestore. It handles session validation and user information retrieval.

```
server.js M
Session Validation > server.js > origin
You, 1 second ago | 1 author (You)
1 const express = require("express");
2 const app = express();
3 const { Firestore } = require("@google-cloud/firestore");
4 module.exports = app;
5
6 const cors = require("cors");
7
8 app.use(
9   cors({
10     origin: `https://frontend-container-6j5ciz4cca-uc.a.run.app`, You, 1 se
11   })
12 );
13
14 const firestore = new Firestore();
15
16 app.use(express.json());
17
18 app.use(express.urlencoded({ extended: true }));
19
20 app.get('/session', async (req, res) => {
21   const stateRef = firestore.collection("state");
22
23   const allOnlineUsers = await stateRef.where("status", "==", "Online").get();
24   let userArray = [];
25   let response = [];
26
27   allOnlineUsers.forEach(async (doc) => {
28     const userObj = {
29       email: doc.id,
30       status: doc.data().status,
31       timestamp: doc.data().timestamp,
32     };
33     userArray.push(userObj);
34   });
35
36   for (i = 0; i < userArray.length; i++) {
37     const userInfoDoc = firestore.collection("Reg").doc(userArray[i].email);
38     const document = await userInfoDoc.get();
39     const userName = document.data().name;
40     const userLocation = document.data().location;
41     response.push({
42       name: userName,
43       location: userLocation,
44       email: userArray[i].email,
45       status: userArray[i].status,
46       timestamp: userArray[i].timestamp,
47     });
48   }
49
50   res.status(200).json(response);
51 });
52
53 app.get('/logout', async (req, res) => {
54   try {
55     const status = r
```

Figure 26: Application business logic Code for container -3 (Session API).

- Logout API endpoint in container 3.

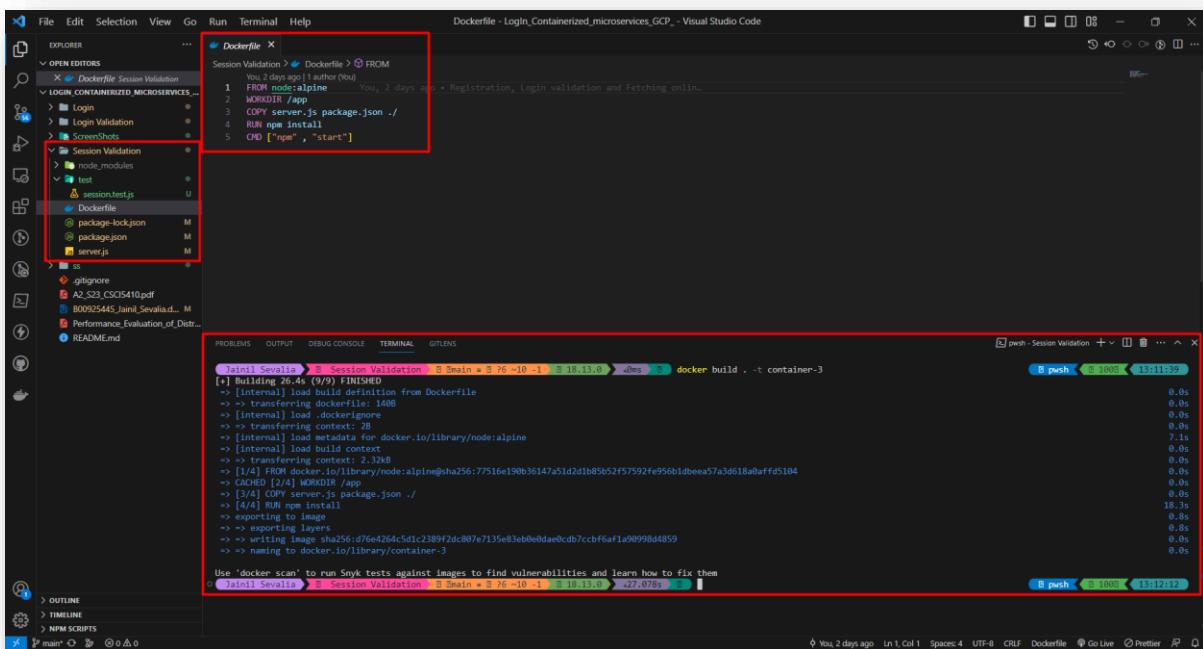


```

 50     res.status(200).json(response);
 51   });
 52 
 53   app.get('/logout', async (req, res) => {
 54     try {
 55       const status = {
 56         status: "Offline",
 57         timestamp: new Date(),
 58       };
 59       const email = req.query.email;
 60       const stateRef = firestore.collection("state").doc(email);
 61       const doc = await stateRef.set(status);
 62       res.sendStatus(200);
 63     } catch (e) {
 64       console.log(e);
 65     }
 66   });
 67 
 68   const PORT = process.env.PORT || 8080;
 69   app.listen(PORT, () => {
 70     console.log(`Server is running on ${PORT}`);
 71   });
 72 
```

Figure 27: Logout API endpoint in container 3.

- Build docker image of container 3.



Dockerfile - Login_Containerized_microservices_GCP - Visual Studio Code

```

FROM docker.io/library/node:alpine
WORKDIR /app
COPY server.js package.json .
RUN npm install
CMD ["npm", "start"]

```

docker build . -t container-3

Building 26.4s (9/9) FINISHED

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

Figure 28 : Build image for container 3.

- Docker image tagged to push it on Artifact Registry.

```

Dockerfile - Login_Containerized_microservices_GCP - Visual Studio Code

File Edit Selection View Go Run Terminal Help
OPEN EDITORS Dockerfile Session Validation
Session Validation > Dockerfile > FROM
You 2 days ago | 1 author (You)
1 FROM node:alpine
2 WORKDIR /app
3 COPY server.js package.json /
4 RUN npm install
5 CMD ["npm", "start"]

Dockerfile
package-lock.json
package.json
server.js
README.md
ss
A2_S23_CSIS410.pdf
B00925445_Jainil_Sevalia...
Performance_Evaluation_of_Distr...
README.md

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL GITLENS
Jainil_Sevalia [Session Validation] 3 Main + 3 76 -> 1 18.13.0 337ms
Jainil_Sevalia [Session Validation] 3 Main + 3 76 -> 1 18.13.0 337ms
REPOSITORY TAG IMAGE ID CREATED SIZE
container-3 latest d7f6d76d5d1 About a minute ago 25.9MB
us-east1-docker.pkg.dev/login-microservice/container-3 latest d7f6d76d5d1 About a minute ago 25.9MB
container-2 latest 944f01a7fcf3 29 minutes ago 25.9MB
us-east1-docker.pkg.dev/login-microservice/container-1 latest 06494e560af 39 minutes ago 31.0MB
container-1 latest 06494e560af 39 minutes ago 31.0MB
frontend_container
none latest 90a0157e9f About an hour ago 53.0MB
<none> d4f3c8d44d 2 hours ago 53.0MB

B push 1800 13:13:28
You, 2 days ago Ln 1, Col 1 Spaces: 4 UTF-8 CRLF Dockerfile Go Live Prettier

```

Figure 29: Docker image tagged for container 3.

- Docker image pushed using command mentioned in below image.

```

Dockerfile - Login_Containerized_microservices_GCP - Visual Studio Code

File Edit Selection View Go Run Terminal Help
OPEN EDITORS Dockerfile Session Validation
Session Validation > Dockerfile > FROM
You 2 days ago | 1 author (You)
1 FROM node:alpine
2 WORKDIR /app
3 COPY server.js package.json /
4 RUN npm install
5 CMD ["npm", "start"]

Dockerfile
package-lock.json
package.json
server.js
README.md
ss
A2_S23_CSIS410.pdf
B00925445_Jainil_Sevalia...
Performance_Evaluation_of_Distr...
README.md

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL GITLENS
Jainil_Sevalia [Session Validation] 3 Main + 3 76 -> 1 18.13.0 337ms
Jainil_Sevalia [Session Validation] 3 Main + 3 76 -> 1 18.13.0 337ms
Using default tag: latest
The push refers to a repository [us-east1-docker.pkg.dev/login-microservice/container-3]
454d4533d44: Pushed
5ab5f46ff44: Pushed
91643d7bbafc: Layer already exists
6ad153d4a83: Layer already exists
8c77sec4a4c: Layer already exists
3b301a1a533: Layer already exists
78a822f62a2d: Layer already exists
latest: digest: sha256:7a013828e41dfdb83210ec4fac1064d1ccb492f09f735f64ace90699f3c6b650 size: 1784
B push 1800 13:13:48
You, 2 days ago Ln 1, Col 1 Spaces: 4 UTF-8 CRLF Dockerfile Go Live Prettier

```

Figure 30 : Docker image build and pushed on the GCP artifact registry.

- GCP Artifact Registry shows the pushed image successfully.

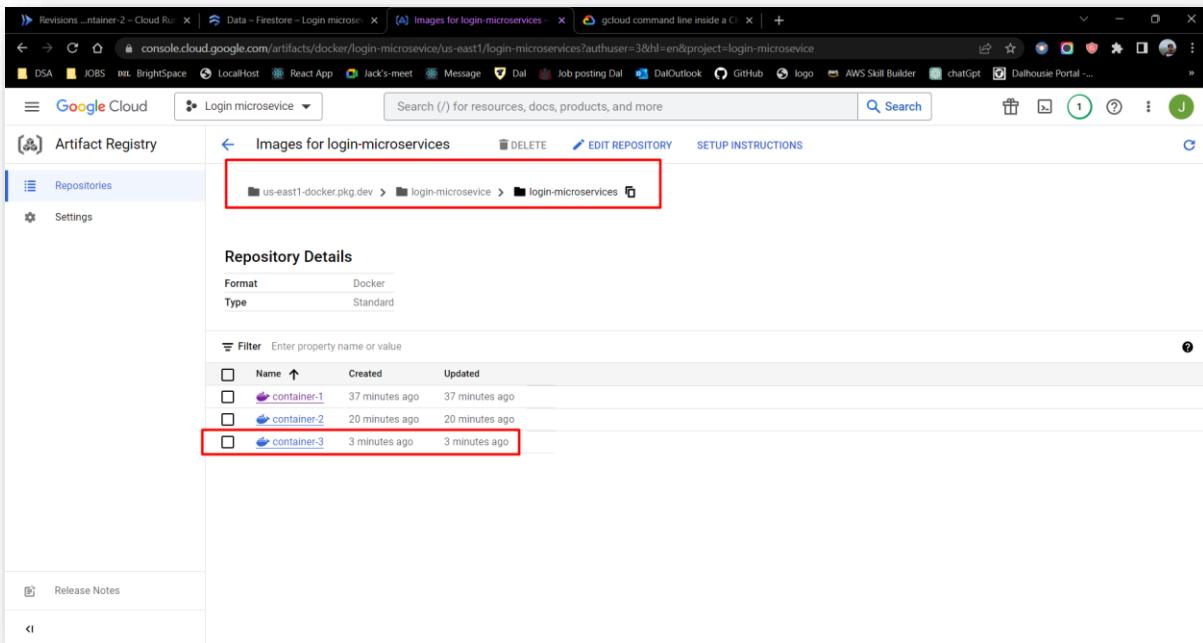


Figure 31: Artifact Registry GCP console.

- Created cloud Run service using pushed image for container 3.

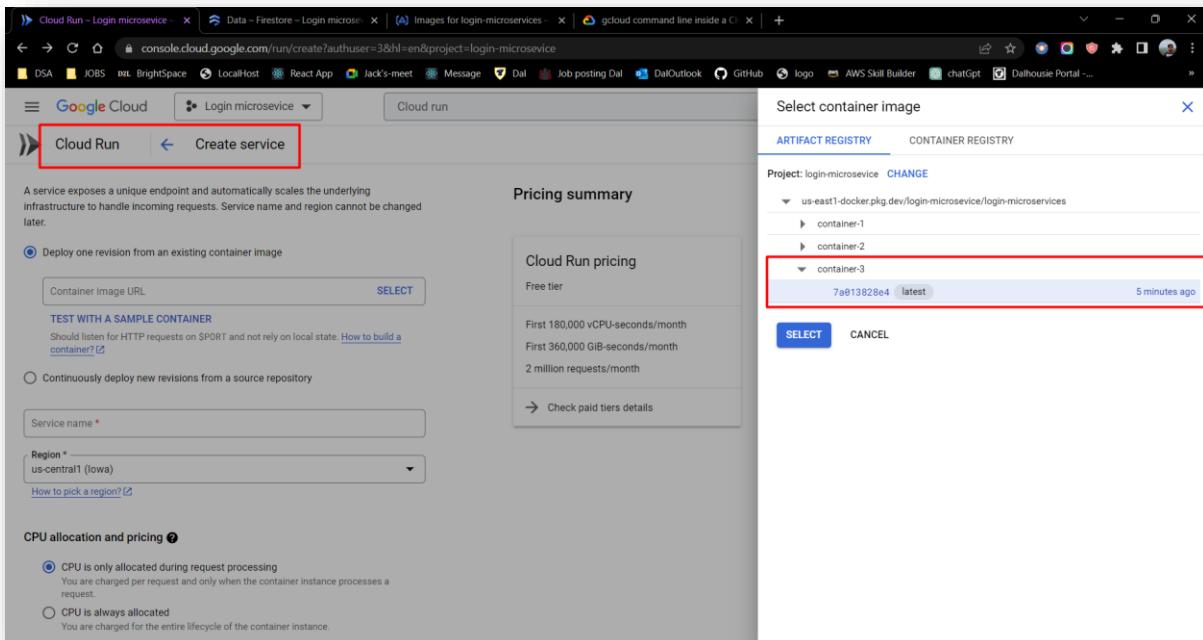


Figure 32 : Creating Cloud Run Service for Container-3.

- Container is running on the GCP cloud run service, which is shown below.

The screenshot shows the Google Cloud Platform Cloud Run Service Details page for a service named 'container-3'. The URL <https://container-3-6j5clz4cca-uc.a.run.app/session> is highlighted. A specific revision, 'container-3-00001-8vv', is selected and its details are displayed in the right panel. The revision's configuration includes an image URL (`us-east1-docker.pkg.dev/login-microservice/login-micr...`) and a port (`8080`).

Figure 33: Container-3 Running on Cloud Run service.

- Testing the container by sending request to container using postman.
 - GET request to get all online users.

The screenshot shows a Postman collection named 'Login_microservices_GCP'. A specific POST request named 'session' is highlighted. The response body is a JSON array of user objects:

```

[{"name": "test", "location": "Halifax, Canada", "email": "test@gmail.com", "status": "Online", "timestamp": {"_seconds": 1687734140, "_nanoseconds": 9356000000}}, {"name": "jayneel", "location": "Halifax, Canada", "email": "jayneel@gmail.com", "status": "Online", "timestamp": {"_seconds": 1688227333, "_nanoseconds": 7948000000}}
]

```

Figure 34 : GET request to get all online users.

- GET request to Logout(end session) of current user.

The screenshot shows the Postman interface. In the left sidebar, under the 'Session' collection, there are two items: 'GET session' and 'GET logout'. The 'GET logout' item is highlighted with a red box. The main panel shows a GET request to 'https://container-3-6j5ciz4cca-uc.a.run.app/logout?email=jayneel@gmail.com'. The response status is 'Status: 200 OK' with a time of '171 ms' and a size of '391 B'. The response body is '1 OK'.

Figure 35 : GET request to Logout user.

- Logout will update the status of user in state collection to offline.

The screenshot shows the Google Firestore Data View. On the left, the 'Data' tab is selected under the 'state' collection. A document for 'jayneel@gmail.com' is shown with the following fields:

Field	Type	Value
status	String	'Offline'
timestamp	Timestamp	July 1, 2023 at 1:23:15PM UTC-3

Figure 36: Logout will update the user status to offline in state collection.

H(For Container3). Test Cases for /Logout and /Session endpoint.

- /Session endpoint Should return online users with their information.
- /Logout Should update the user status to offline and return the updated status.

The screenshot shows the Visual Studio Code interface with two code files and a terminal window.

server.js - Login_Containerized_microservices_GCP - Visual Studio Code

```

1  // server.js
2  const express = require('express');
3  const app = express();
4  const cors = require('cors');
5  const mongoose = require('mongoose');
6  const sessionValidation = require('./sessionValidation');
7  const session = require('./session');
8
9  app.use(cors());
10 app.use(express.json());
11
12 app.get('/session', async (req, res) => {
13   const userArray = [
14     {
15       name: 'Jainil',
16       location: 'Montreal',
17       email: 'userArray[1].email',
18       status: userArray[1].status,
19       timestamp: userArray[1].timestamp,
20     }
21   ];
22
23   res.status(200).json(response);
24 });
25
26 app.get('/logout', async (req, res) => {
27   try {
28     const status = {
29       status: 'Offline',
30       timestamp: new Date(),
31     };
32
33     const email = req.query.email;
34     const stateRef = firestore.collection("state").doc(email);
35     const state = await stateRef.set(status);
36     res.sendStatus(200);
37   } catch (e) {
38     console.error(`Error: ${e.message}, 2 days ago + Registration, Login validation and Fetching online.`);
39   }
40 });
41
42 const PORT = process.env.PORT || 8080;
43
44 mongoose.connect('mongodb://127.0.0.1:27017/test', { useNewUrlParser: true, useUnifiedTopology: true });
45
46 module.exports = app;

```

session.test.js - Visual Studio Code

```

1  // session.test.js
2
3  describe('Session Endpoint', () => {
4    it('should return online users with their information', (done) => {
5      request(app)
6        .get('/session')
7        .expect(200)
8        .end((err, res) => {
9          if (err) return done(err);
10
11          assert.strictEqual(Array.isArray(res.body), true);
12          assert.ok(res.body.length > 0);
13        })
14      done();
15    });
16  });
17
18 });
19
20
21 describe('Logout Endpoint', () => {
22   it('should update the user status to offline and return the updated status', (done) => {
23     request(app)
24       .get('/logout?email=jainil@gmail.com')
25       .expect(200)
26       .end((err, res) => {
27         if (err) return done(err);
28
29         assert.strictEqual(res.status, 200);
30       })
31     done();
32   });
33 });

```

Terminal

```

Terminate batch job (Y/N)? y
Jainil Sevalia: Session Validation: 25 → 10 → 1 18.15.0 ↵ 14.957s ↵ npx mocha
Server is running on 8080

Session Endpoint
✓ should return online users with their information (1790ms)

Logout Endpoint
✓ should update the user status to offline and return the updated status (120ms)

2 passing (2s)

```

Figure 37 : Test Cases for container 3

G. Frontend and Final testing using UI.

- Docker image building of frontend project.

The screenshot shows the Visual Studio Code interface with the Dockerfile open in the editor. The terminal window below shows the command `docker build -t frontend_container` being run, and the output of the build process. The Dockerfile content is:

```

FROM node:alpine
WORKDIR /app
COPY .
RUN npm install
CMD ["npm", "start"]

```

The terminal output shows the build process completed successfully:

```

[+] Building 578.5s (9/9) FINISHED
=] [internal] load build definition from Dockerfile
=> transferring dockerfile: 119B
=] [internal] load .dockerignore
=] [internal] transfer context: 311.8kB
=> [internal] load metadata for docker.io/library/node:alpine
=> [internal] load build context
=> transferring context: 311.8kB
=> [1/4] FROM docker.io/library/node:alpine@sha256:77516e190b36147a51d2d1b85b52f57592fe956b1dbea57a3d618a0affd5104
=> CACHED [2/4] WORKDIR /app
=> CACHED [3/4] COPY .
=> CACHED [4/4] RUN npm install
=> exporting to image
=> exporting layers
=> writing image sha256:90d015747e9f3aff08d5d71f2971c71bd626ba772271af532b4ebb28c02faa4
=> naming to docker.io/library/frontend_container

```

Figure 38 : Docker Image building of Frontend.

- Docker image tagged.

The screenshot shows the Visual Studio Code interface with the Dockerfile open in the editor. The terminal window below shows the command `docker tag frontend_container us-east1-docker.pkg.dev/login-microservice/login-microservices/frontend_container` being run, and the output of the tagging process. The Dockerfile content is identical to Figure 38.

The terminal output shows the tag command completed successfully:

```

[+] Building 311.8MB
=> [1/4] FROM docker.io/library/node:alpine@sha256:77516e190b36147a51d2d1b85b52f57592fe956b1dbea57a3d618a0affd5104
=> CACHED [2/4] WORKDIR /app
=> CACHED [3/4] COPY .
=> CACHED [4/4] RUN npm install
=> exporting to image
=> exporting layers
=> writing image sha256:90d015747e9f3aff08d5d71f2971c71bd626ba772271af532b4ebb28c02faa4
=> naming to docker.io/library/frontend_container

```

Below the terminal, the Docker Images table shows the tagged image:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
container-1	latest	d76e264c5df	24 minutes ago	25.9MB
us-east1-docker.pkg.dev/login-microservice/login-microservices/container-1	latest	96494e5d0adef	24 minutes ago	25.9MB
us-east1-docker.pkg.dev/login-microservice/login-microservices/container-2	latest	94f101a7cf3	44 minutes ago	25.9MB
container-2	latest	94f101a7cf3	44 minutes ago	25.9MB
us-east1-docker.pkg.dev/login-microservice/login-microservices/container-1	latest	06494e5d0adef	About an hour ago	31.0MB
container-1	latest	06494e5d0adef	About an hour ago	31.0MB
us-east1-docker.pkg.dev/login-microservice/login-microservices/frontend_container	latest	98d8015747e9f	About an hour ago	53.9MB
frontend_container	latest	98d8015747e9f	About an hour ago	53.9MB
us-east1-docker.pkg.dev/login-microservice/login-microservices/frontend_container	none?	98d8015747e9f	About an hour ago	53.9MB
us-east1-docker.pkg.dev/login-microservice/login-microservices/frontend_container	none?	d4f3ccdc44d	2 hours ago	23.9MB

Figure 39 : Docker image tagged for frontend container.

- Push Docker image to Artifact Registry.

The screenshot shows the Visual Studio Code interface with the Dockerfile open in the editor. The terminal window at the bottom shows the command being run:

```
On [Jainil-Sevalia]: ~ Frontend For Microservices [18:15:0] 0ms ⌂ docker push us-east1-docker.pkg.dev/login-microservice/login-microservices/frontend_container
```

The terminal output shows the progress of the push operation, indicating layers are already exists for most components. The final message shows the image was pushed successfully.

Figure 40 : Frontend Image Pushed to Artifact Registry.

- Artifact Registry Console.

The screenshot shows the Google Cloud Artifact Registry console. The URL in the address bar is `console.cloud.google.com/artifacts/docker/login-microservice/us-east1/login-microservices?authuser=3&hl=en&project=login-microservice`. The page displays the repository details for the `us-east1-docker.pkg.dev/login-microservice/login-microservices` repository. The table lists several Docker containers, with the last one, `frontend_container`, highlighted with a red border.

Name	Created	Updated
<code>container-1</code>	59 minutes ago	59 minutes ago
<code>container-2</code>	42 minutes ago	42 minutes ago
<code>container-3</code>	24 minutes ago	24 minutes ago
<code>frontend_container</code>	Just now	Just now

Figure 41 : Artifact Registry - Fronted Image.

- Create new Cloud Run Service for Frontend Container.

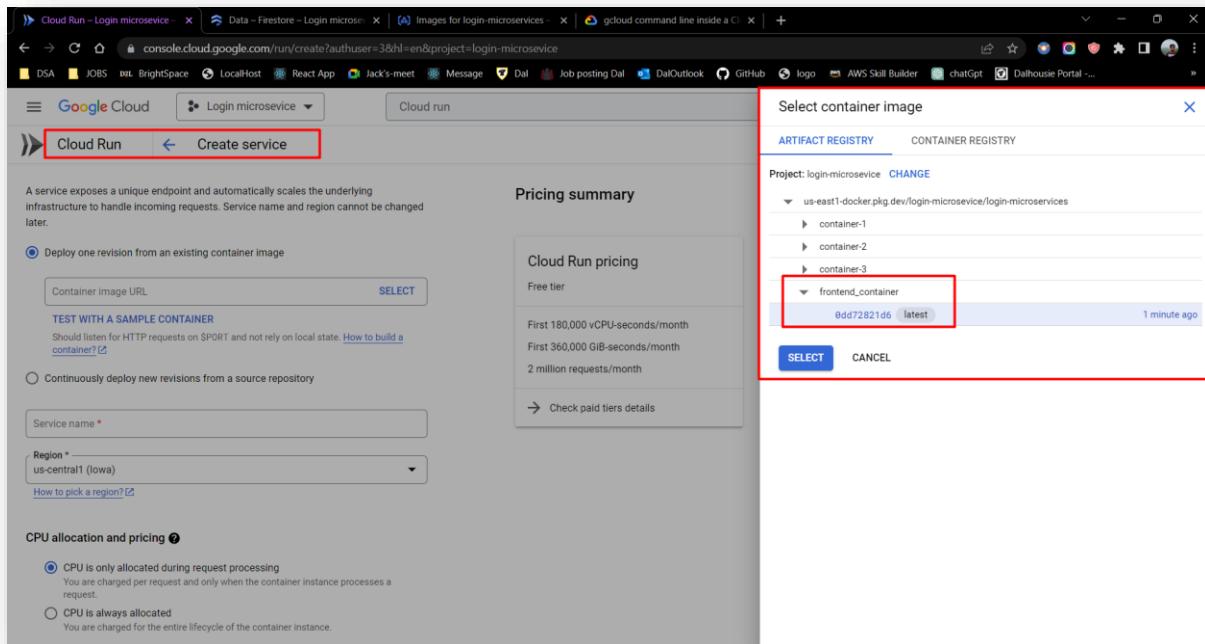


Figure 42 : Creating Cloud Run service for Frontend Container.

- Frontend Running on Cloud Run service.

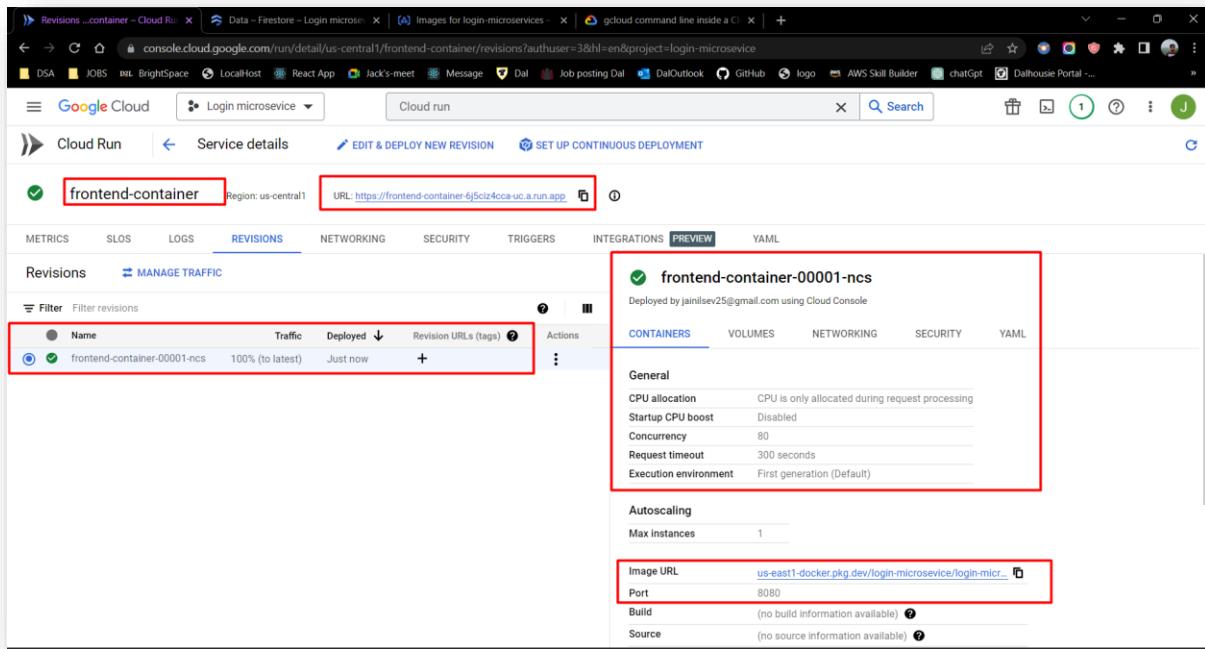


Figure 43: Frontend Running on Cloud Run.

- User Registration page:

The screenshot shows a web browser window with a registration form titled "Registration Form". The form fields include Name (Jainil Sevalia), Email (jainilsevalia@gmail.com), Password (redacted), and Location (Halifax, Canada). A "Submit" button is at the bottom.

Figure 44: User Registration page(Build using React).

- User Document created successfully in Firestore Database.

The screenshot shows the Google Cloud Firestore console. It displays a collection named "Reg" under the database "login-microservice". A document with the email "jainilsevalia@gmail.com" has been added and is shown with its details: name, location, email, and password. The entire document and its field values are highlighted with red boxes.

Figure 45 : User Document Successfully created in Firestore.

- Login Page:

The screenshot shows a web browser window with a single tab open to 'frontend-container-6j5clz4cca-uc.a.run.app/login'. The page title is 'Login Form'. It contains two input fields: one for 'Email' containing 'jainilsevalia@gmail.com' and one for 'Password' containing '.....'. Below the password field is a red 'Login' button.

Figure 46 : Login Page.

- Online user page:

The screenshot shows a web browser window with a single tab open to 'frontend-container-6j5clz4cca-uc.a.run.app/profile'. The page title is 'Online Users'. It displays two user profiles:
1. **Jainil Sevalia**: Email - jainilsev25@gmail.com, Location - Halifax, Online from - 22:12.
2. **Jayneel**: Email - jayneel@gmail.com, Location - Halifax, Canada, Online from - 13:23.
At the bottom left is a red 'Logout' button.

Figure 47 : Online User List after login.

I. Summary:

Google Cloud Run provided me with a serverless compute platform to run my stateless containers on a fully managed environment [1]. It handled infrastructure management, autoscaling, and load balancing, allowing me to focus on deploying and running my microservices [1]. I deployed my containerized microservices as Cloud Run services, benefiting from automatic scaling based on traffic and seamless load balancing.

To encapsulate my backend logic, I leveraged Docker containers. Docker allowed me to automate the deployment, scaling, and management of my applications using containerization [2]. I packaged each microservice into a separate Docker container, ensuring consistent behavior and easy deployment across different environments [2]. This approach simplified the deployment process and provided a lightweight and consistent runtime environment for my microservices.

For storing, managing, and distributing my container images, I used GCR/Artifact Registry. These managed container registries provided by Google Cloud Platform allowed me to push my Docker images, making them readily available for deployment [3]. By pushing the images to GCR/Artifact Registry, I centralized my repository and facilitated the deployment of my containerized microservices to Google Cloud Run [3].

I leveraged Google Cloud Run, Docker containers, and GCR/Artifact Registry to build and deploy my microservices-based application. Google Cloud Run provided a serverless compute platform, Docker enabled containerization and simplified deployment, and GCR/Artifact Registry served as a centralized repository for my container images [4]. This combination of technologies allowed for scalability, portability, and easy deployment while abstracting away infrastructure management concerns [4].

In my application, I utilized React as the frontend technology, leveraging its component-based architecture to develop three web pages - registration, login, and online users. React's efficiency and virtual DOM ensured a smooth user experience. For the backend, I employed Node.js as the runtime environment, coupled with Express.js as the web application framework. This combination allowed me to build containerized microservices responsible for user registration, login validation, and fetching online user information from the Firestore database. The integration of React, Node.js, Express.js and mocha.js [5] facilitated the development of a responsive and dynamic application, complementing the deployment of microservices on Google Cloud Run.

References:

- [1] “Google cloud documentation”, documentation. [Online]. Available: <https://cloud.google.com/docs> [Accessed Jul. 1, 2023].
- [2] “Reference documentation,” Docker Documentation. [Online]. Available: <https://docs.docker.com/reference/> [Accessed Jul. 1, 2023].
- [3] “Push and pull images”, “artifact registry documentation”, “google cloud,” Google. [Online]. Available: <https://cloud.google.com/artifact-registry/docs/docker/pushing-and-pulling> [Accessed Jul. 1, 2023].
- [4] “Cloud run documentation”, “cloud run documentation”, “google cloud,” Google. [Online]. <https://cloud.google.com/run/docs> [Accessed Jul. 1, 2023].
- [5] “The fun, simple, flexible JavaScript test framework,” Mocha. [Online] <https://mochajs.org/> [Accessed Jul. 1, 2023].