Part B Summary and Steps

I completed the following tasks to build and deploy a containerized application using Google Cloud Platform (GCP):

Containerized Microservices: I created three microservices, each responsible for a specific backend logic of the application. The microservices were containerized using Docker, allowing for easy deployment and scalability.

Firestore Database: I chose Firestore as the database for my application. I created two collections: "Reg" to store user registration data and "state" to store user state information such as online/offline status and timestamp.

Container #1 - Registration: I developed Container #1 to handle user registration. It received registration details from the frontend and stored them in the "Reg" collection of Firestore.

Container #2 - Login Validation: Container #2 was responsible for validating user login information by comparing it with the values in the database. When a user successfully logged in, their state in Firestore was updated to "online".

Container #3 - State Extraction: I implemented Container #3 to extract state information from the Firestore database. It maintained the user's session from login to logout and updated the user's state to "offline" upon logout.

Docker Image Creation and Artifact Registry: I created Docker images for each container and pushed them to the Artifact Registry repository. This ensured that the container images were securely stored and versioned.

Deployment on Cloud Run: I deployed the container images on Cloud Run, a serverless platform provided by GCP. Cloud Run handled the scaling and management of the application, allowing it to run efficiently and handle incoming requests.

Web Page Development: I developed three simple web pages using the technology of my choice. These pages enabled users to register, login, and view online users based on the data retrieved from Firestore.

Testing and Test Case Creation: I wrote test cases to verify the functionality and reliability of the application. Using tools like Supertest, I conducted API testing and captured screenshots as evidence of the testing process.

Note:- Please wait for some time to load the Front end website, it takes up to 10 seconds to load

Front end Deployment link - https://front-end-i6trs4yawa-uc.a.run.app

Container1(Registration) Backend Hosted url - https://container1-i6trs4yawa-uc.a.run.app

Container 2(Login) Backend Hosted url - https://container2-i6trs4yawa-uc.a.run.app

Container 3(Status) Backend Hosted Url - https://container3-i6trs4yawa-uc.a.run.app

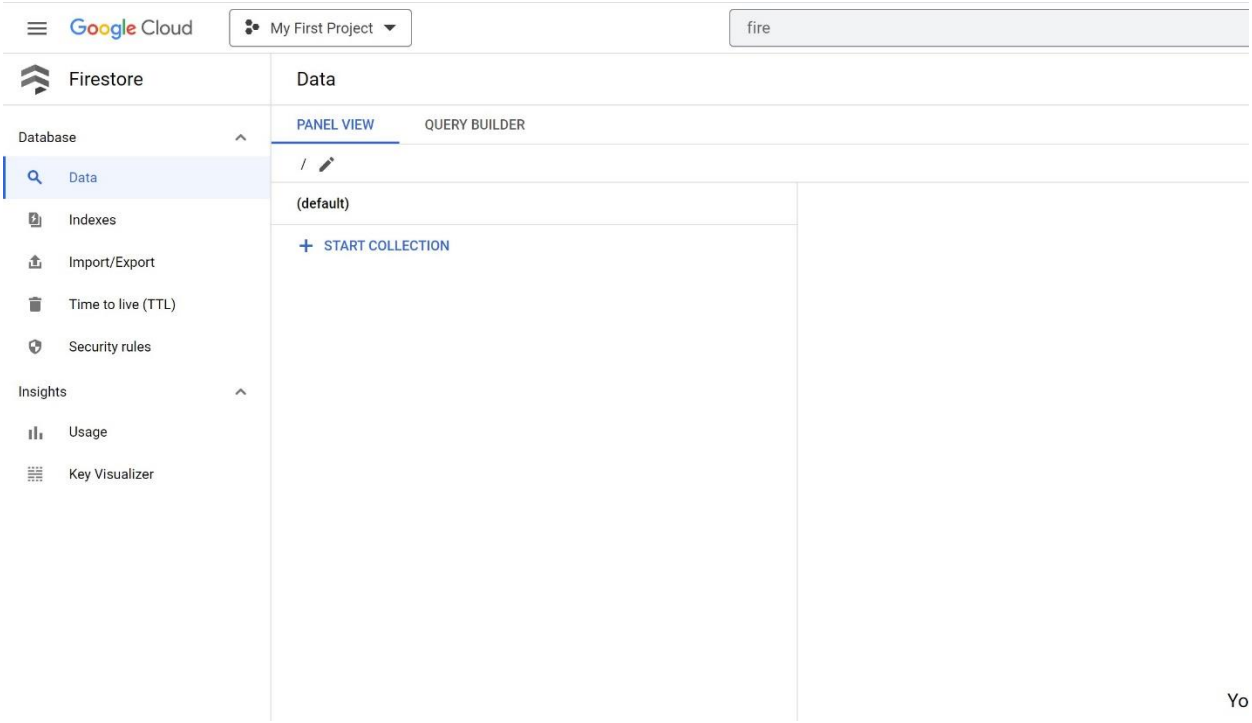


Fig 1: Empty Firestore DB (No data)

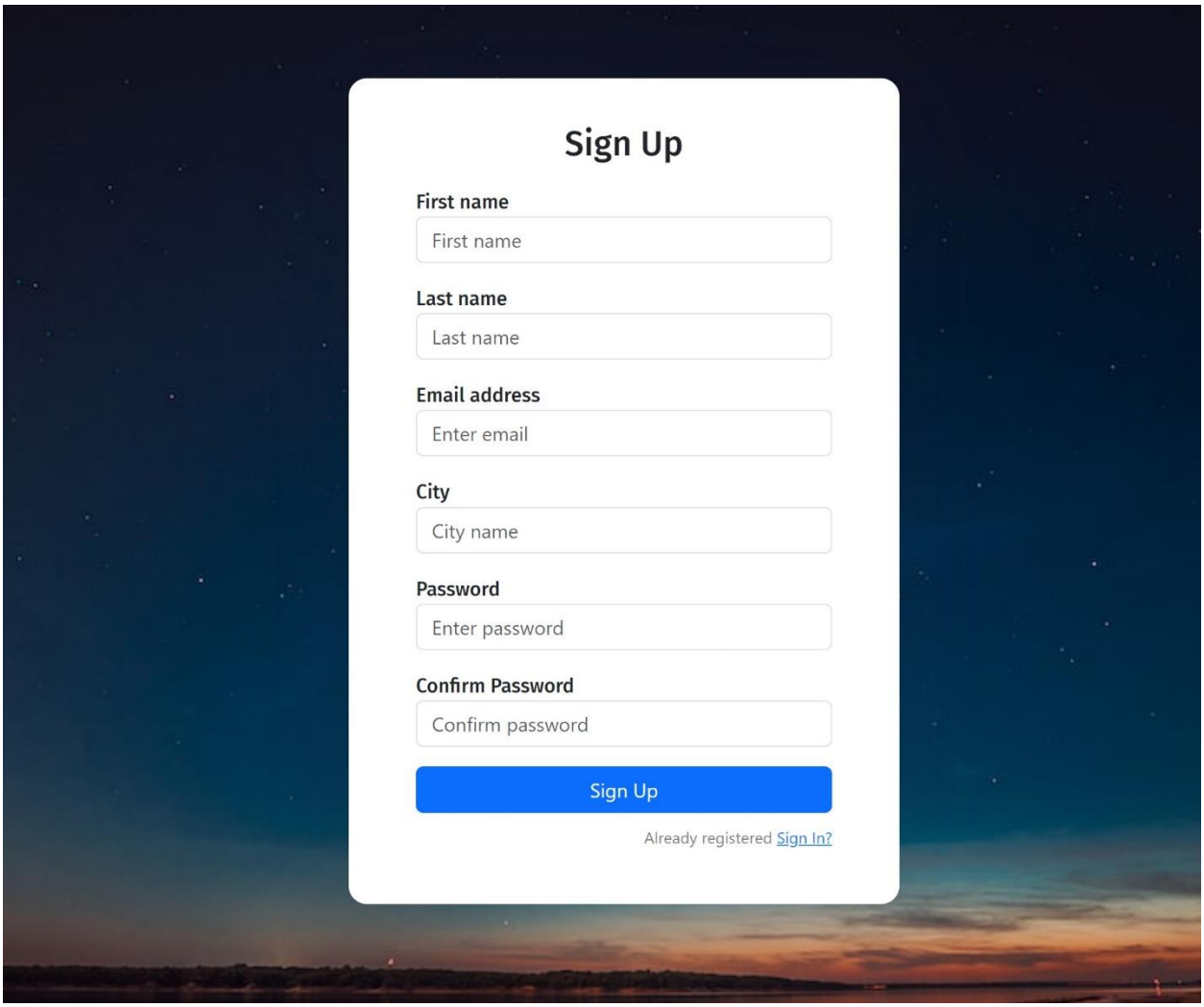


Fig 2: User Sign Up Page

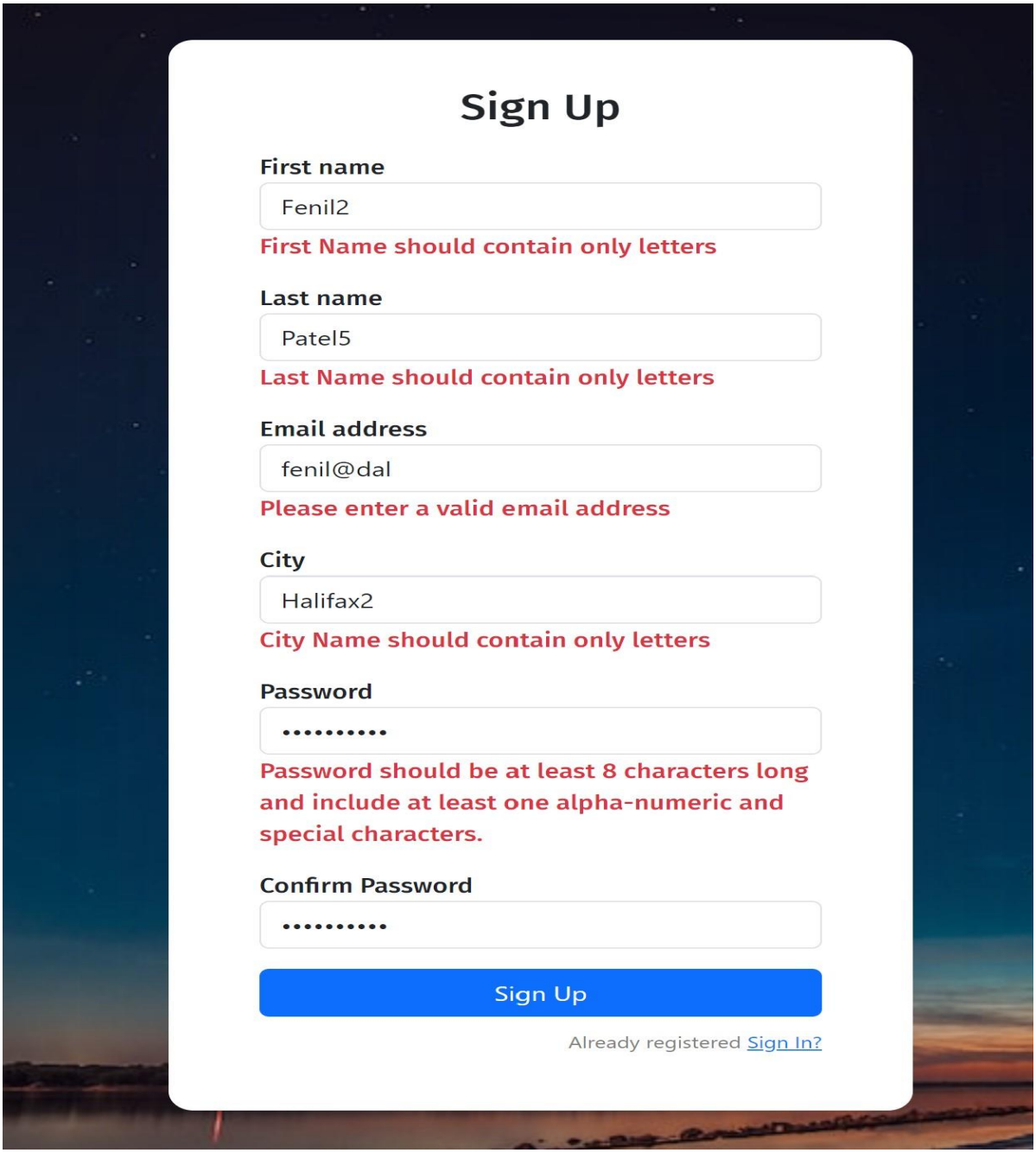


Fig 3: Implemented Front End Validation logic into Sign Up Page

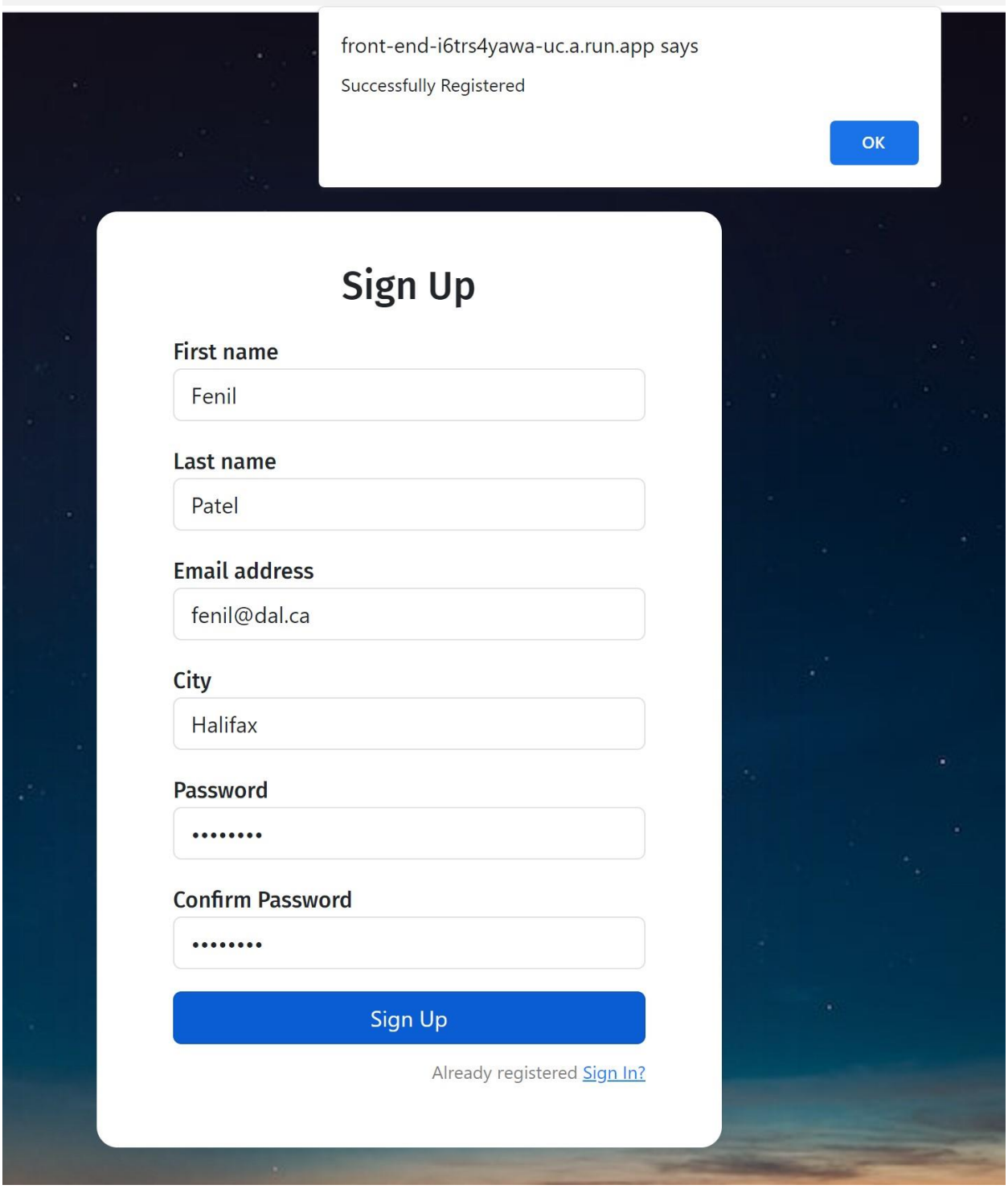


Fig 4: Successful Registration of the User

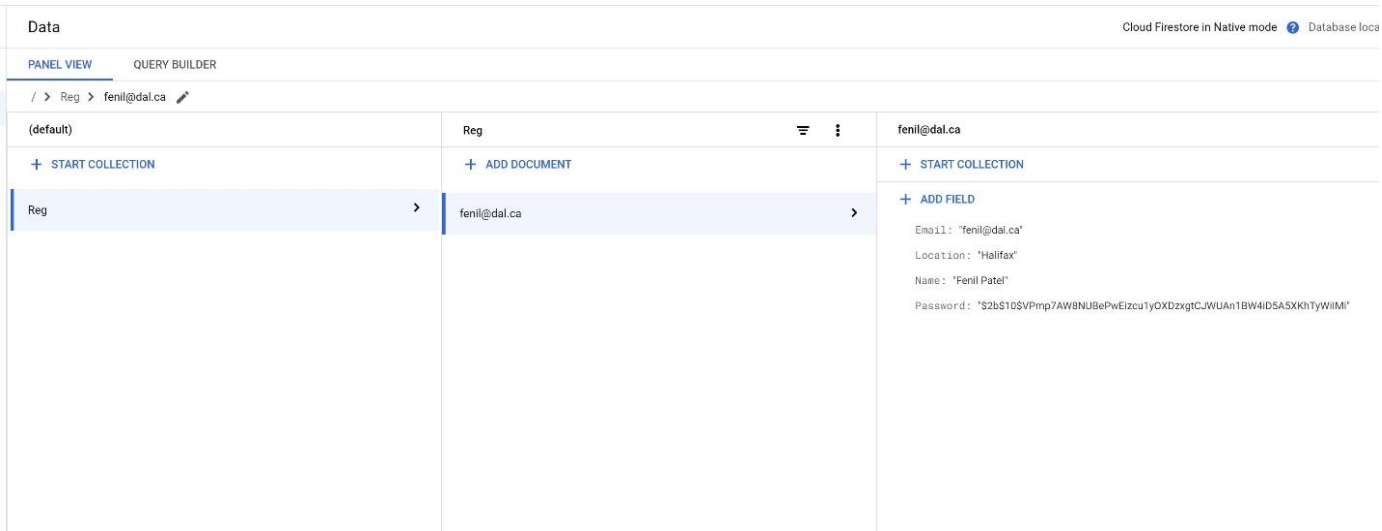


Fig 5 : Registration Entry into Reg Collection of Firestore db

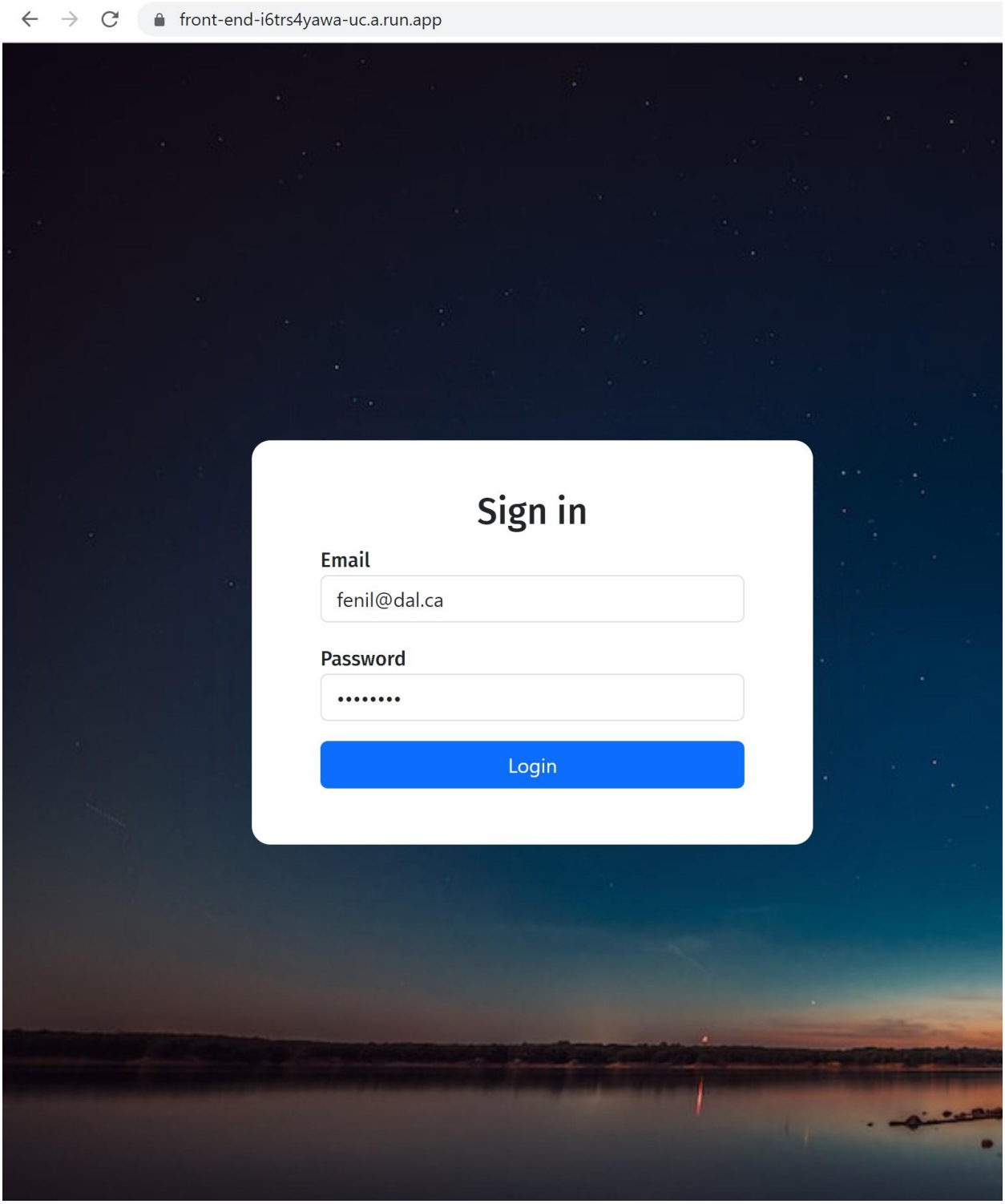


Fig 6 : User Redirects to Sign In Page After Sign Up

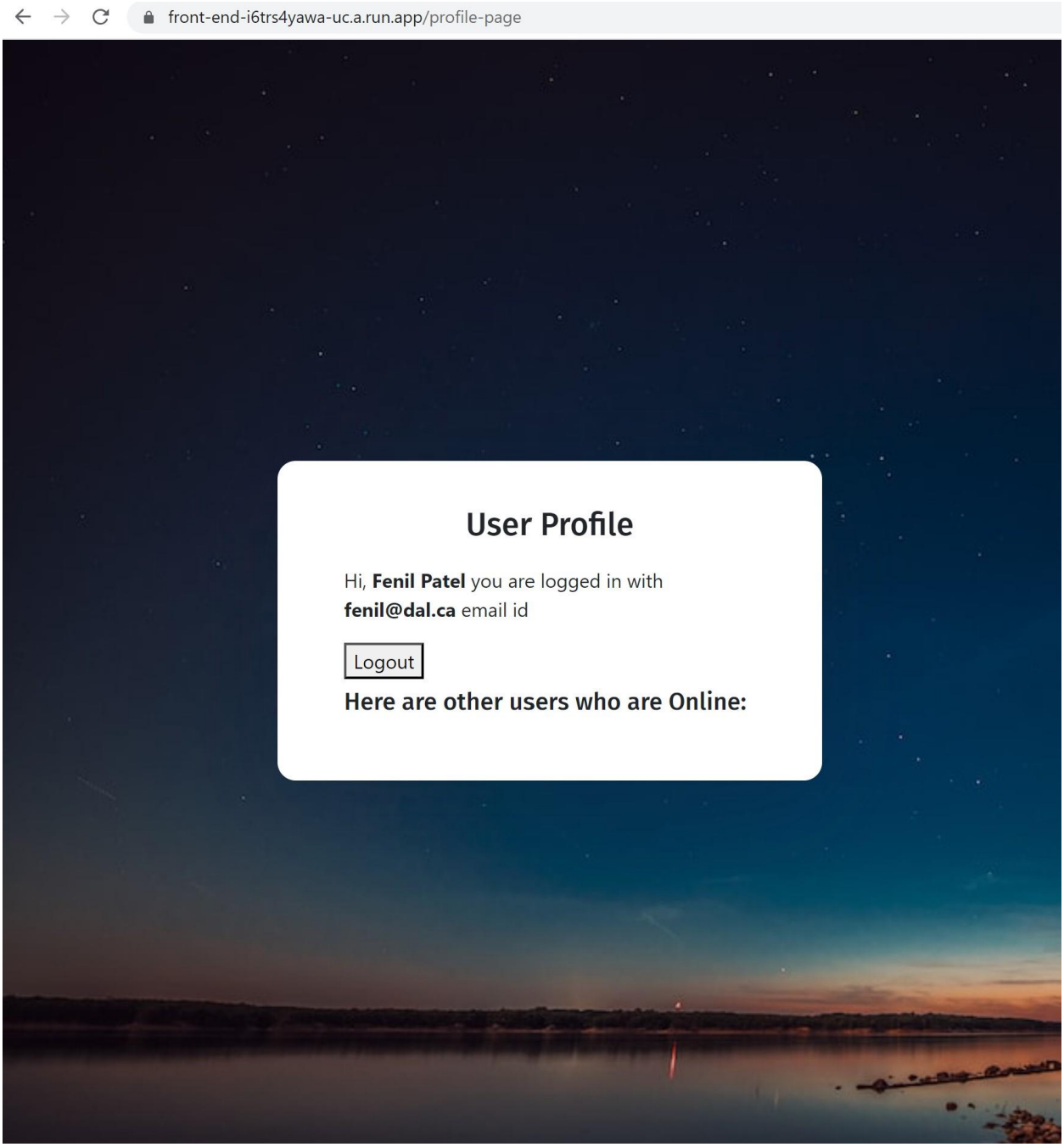


Fig 7: User Logs in, Sees User Profile Page

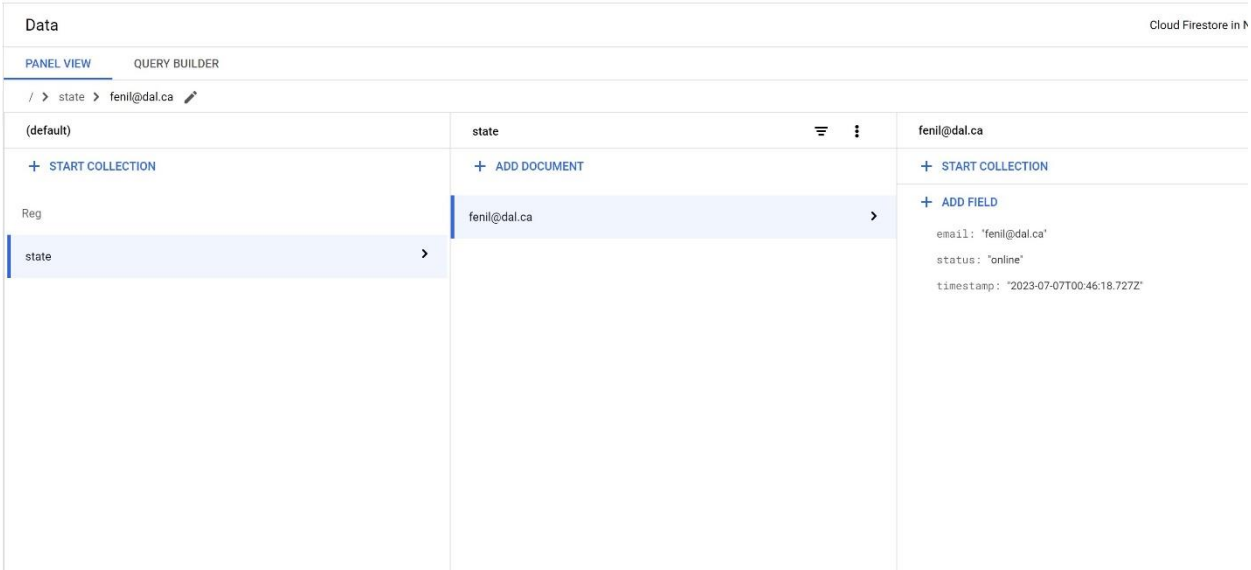


Fig 8 : After Login User state will created and Status will be shown

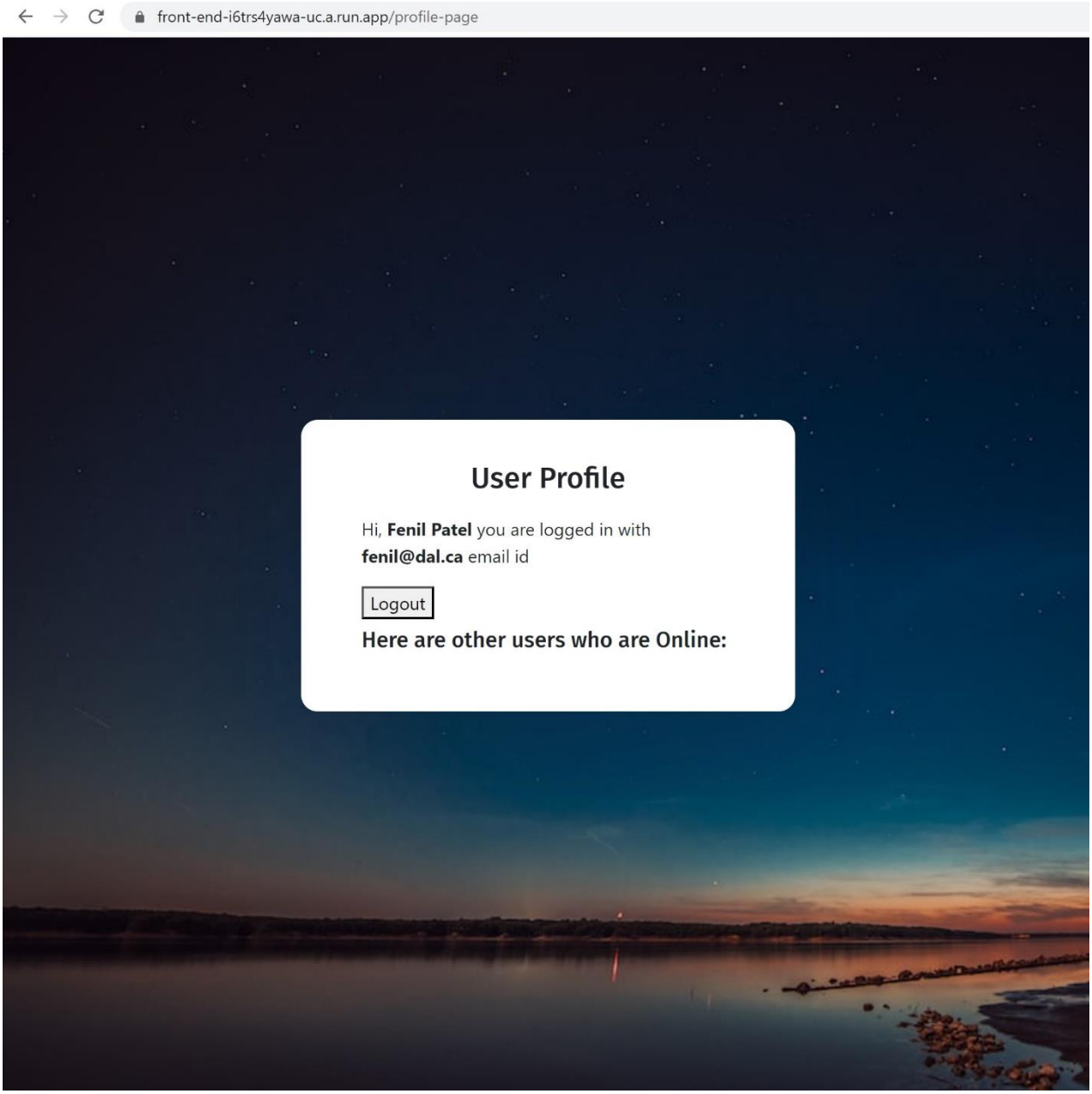


Fig 9 : User Logout through Logout button

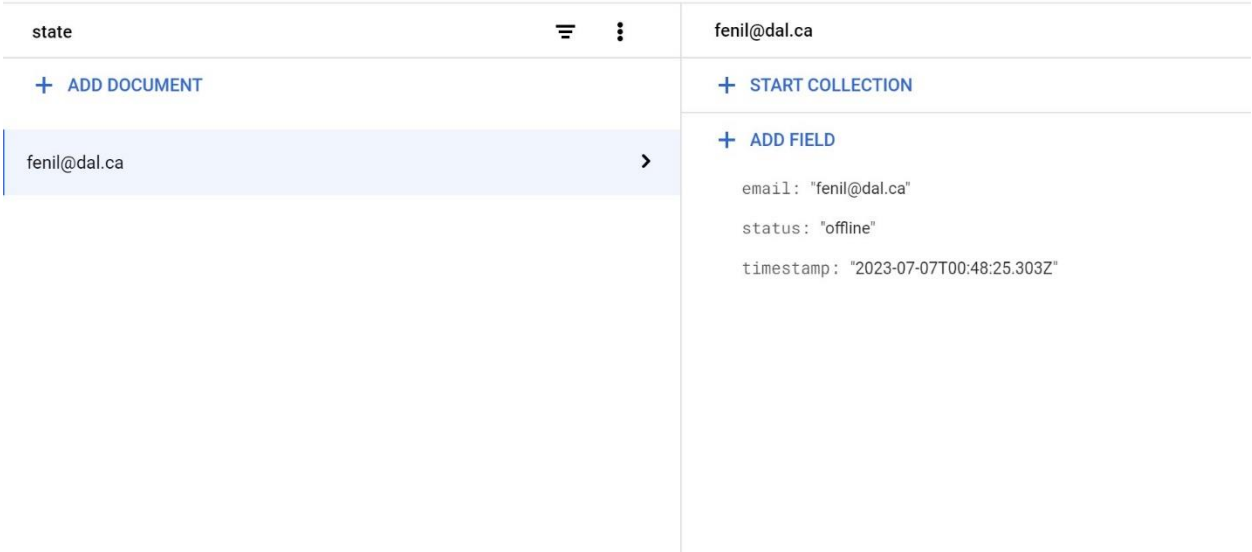


Fig 10: After Successful Log out, User Status will be turn into Offline

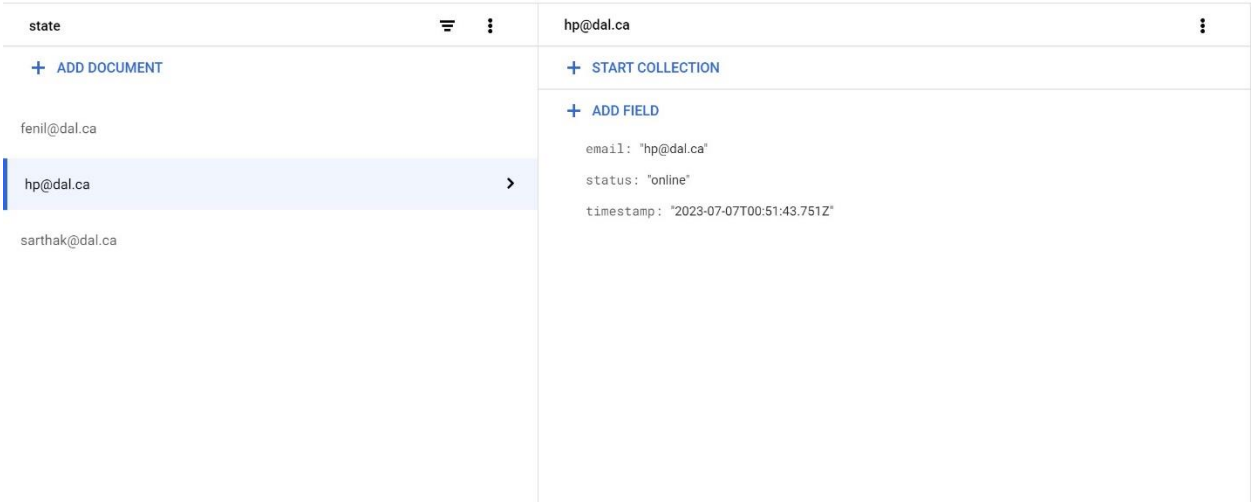


Fig 11: Showcasing Other users who are logged in and are online

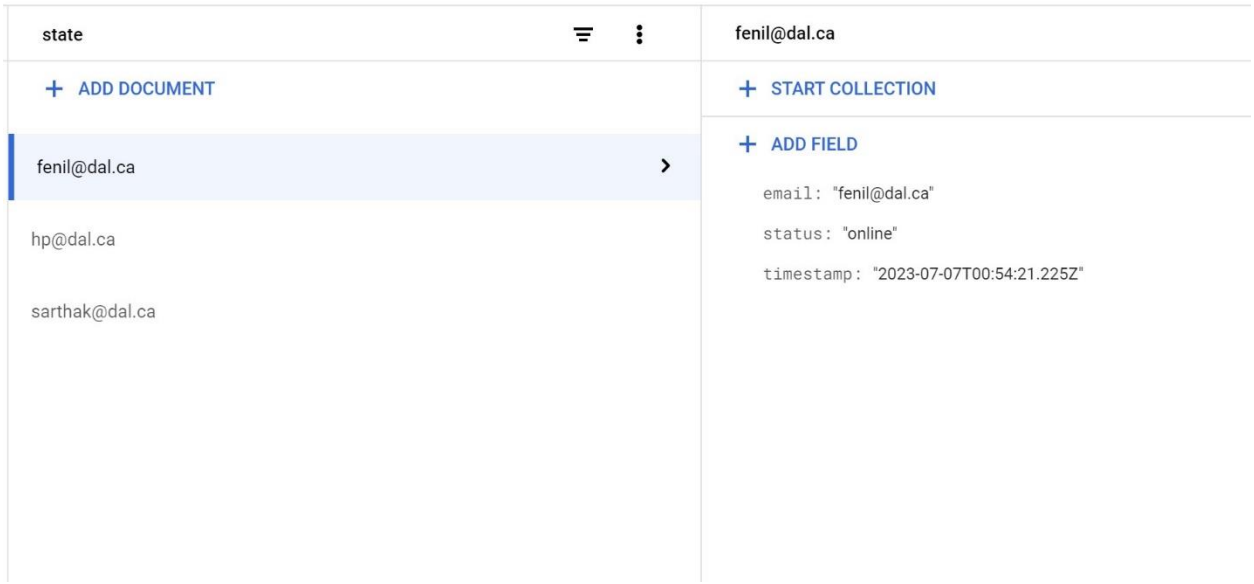


Fig 12: Showcasing Other users who are logged in and are online

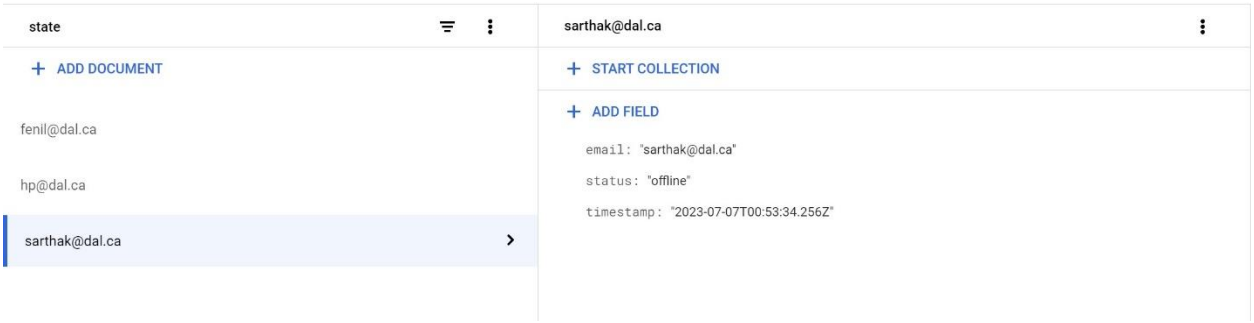


Fig 13: User Sarthak Is offline

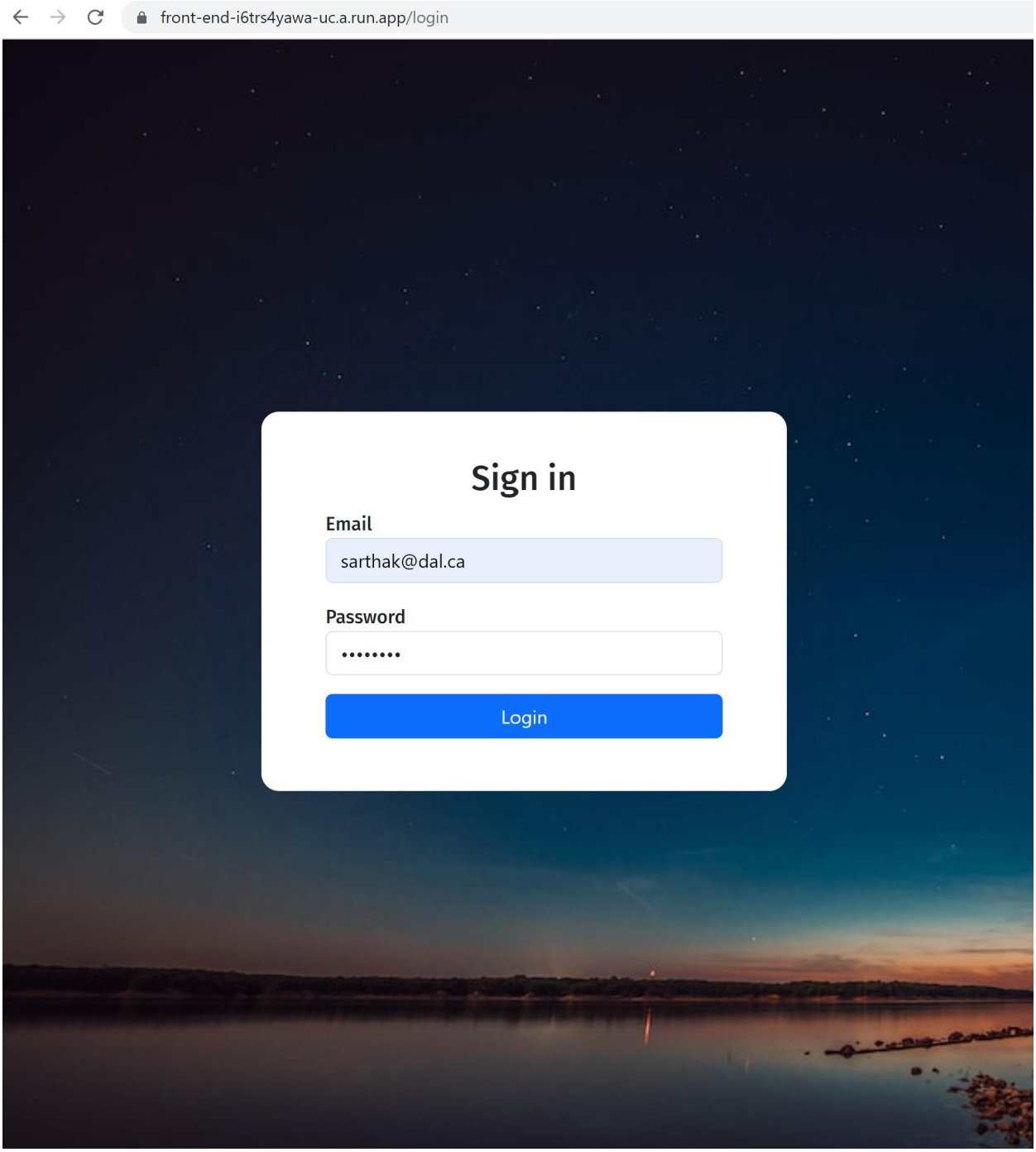


Fig 14: User Sarthak Logs in

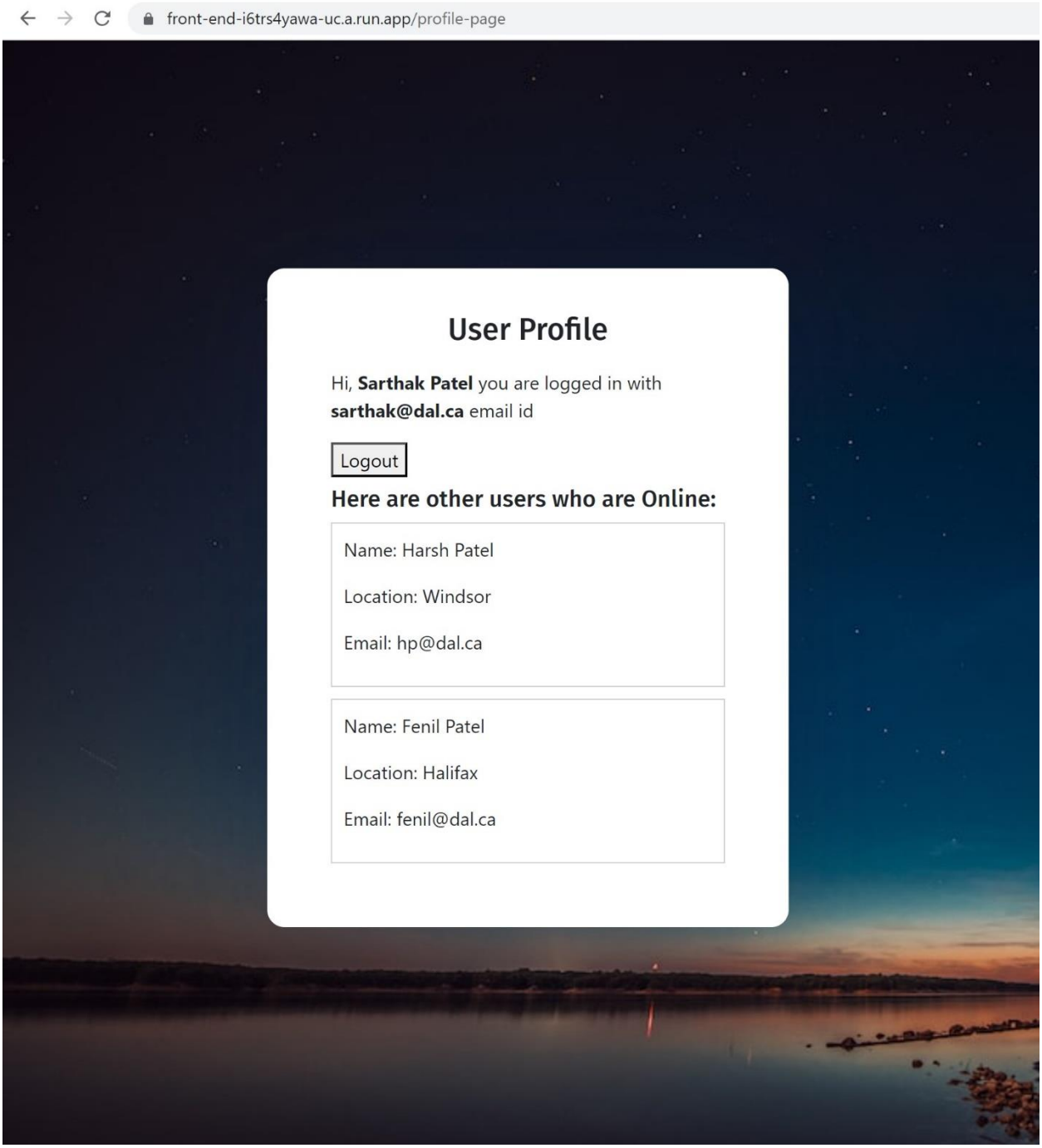


Fig 15 : After Successful Log in, User Profile Will be displayed, and Other Users, who are online will be displayed with their Name, Location and Email

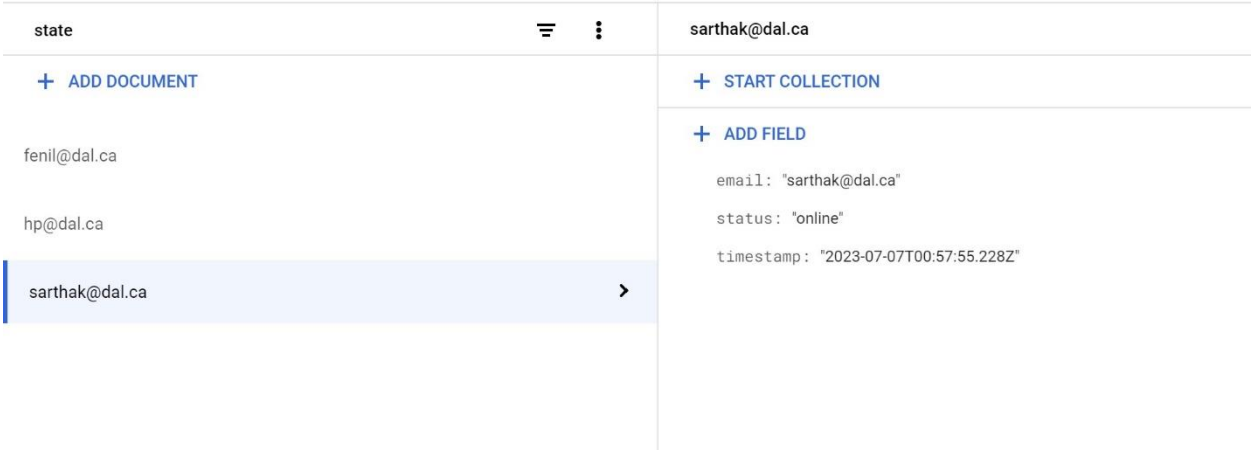


Fig 16: User Sarthak's Status Will be turned to online after Login

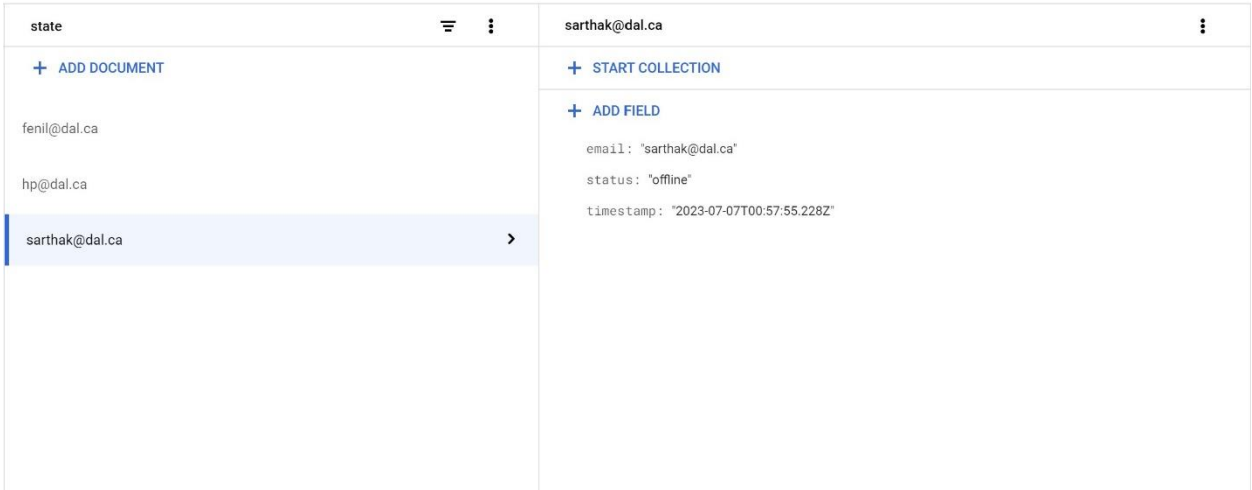


Fig 17: When User Sarthak Logs out, then His Status will be turned into offline

```
app.post( path: '/login',  handlers: async (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody, LocalsObj> ) : Promise<...>  => {
    try {
        // Retrieve email and password from the request body
        const email = req.body.email;
        const password = req.body.password;

        // Fetch user data from the Firestore collection
        const userObj : DocumentSnapshot<DocumentData>  = await usersDb.doc(email).get();

        // Check if the user exists
        if (!userObj.exists) {
            return res.status( code: 404).json( body: { message: 'User not found' });
        }

        // Retrieve stored password from user data
        const userData = userObj.data();
        const storedPassword = userData.Password;
        // Compare the provided password with the stored password
        const passwordMatch = await bcrypt.compare(password, storedPassword);
        if (!passwordMatch) {
            return res.status( code: 401).json( body: { message: 'Invalid password' });
        }
        // Password is valid, proceed with login logic
        // Update the user's status in the "state" collection of Firestore
        const stateData : {...}  = {
            email: email,
            status: 'online',
            timestamp: new Date().toISOString()
        };
        const response : WriteResult  = await stateDb.doc(email).set(stateData);

        // Set sessionID cookie and send user data in the response
        res.cookie( name: 'sessionID', email,  options: { httpOnly: true });
        res.send( body: { userData: userData });

    } catch (error) {
        res.status( code: 500).send( body: { message: 'Server error' });
    }
```

Fig 18: Login API Backend (Container 2) NodeJS Code

```
// Logout route
app.post( path: '/logout',  handlers: async (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody, LocalsObj> ) : Promise<void>  => {
    try {
        // Retrieve email from the request body
        const email = req.body.email;

        // Update the user's status in the "state" collection to "offline"
        const stateDoc : DocumentReference<DocumentData>  = stateDb.doc(email);
        await stateDoc.update({ status: 'offline' });

        // Clear the sessionID cookie and send the success message in the response
        res.clearCookie( name: 'sessionID');
        res.send( body: { message: 'Logout successful' });

    } catch (error) {
        res.status( code: 500).send( body: { message: 'Server error' });
    }
});

// Start the server
const port : string | number  = process.env.PORT || 3003;
app.listen(port,  hostname: () : void  => {
    console.log(`Server is running on port ${port}`);
});

// Export the app
module.exports = app;
```

Fig 19: Logout API Backend (Container 3) Nodejs Code

```
// Get online users route
app.get('/online-users', async (req : Request<P, ResBody, ReqBody, ReqQuery, LocalsObj> , res : Response<ResBody, LocalsObj> ) : Promise<void>  => {
    try {
        // Retrieve sessionID from the cookie
        const sessionID = req.cookies.sessionID;

        // Query the "state" collection to get online users
        const query : QuerySnapshot<DocumentData>  = await stateDb.where( fieldPath: 'status',  opStr: '==',  value: 'online').get();

        // Retrieve user data for each online user
        const onlineUsers : any[]  = [];
        const promises : unknown[]  = query.docs.map(async (doc) : Promise<void>  => {
            const email = doc.data().email;
            const regDoc : DocumentSnapshot<DocumentData>  = await usersDb.doc(email).get();
            const userData = regDoc.data();
            onlineUsers.push(userData);
        });

        await Promise.all(promises);

        // Send the online users data in the response
        res.send( body: { onlineUsers: onlineUsers });

    } catch (error) {
        res.status( code: 500).send( body: { message: 'Server error' });
    }
});
```

Fig 20 : online-users API Backend (Container 3) Nodejs Code

```
{} package.json    JS app.js    Dockerfile    {} package-lock.json    app.test.js ×    {} sigma-rarity.json
const request : function((Function | Server), ... | {...}  = require('supertest');
const app = require('./app'); // Import the main app file

describe('Registration', () : void  => {
    test('should register a new user', async () : Promise<void>  => {
        // Define the user data for registration
        const userData : {...}  = {
            firstName: 'Narendra',
            lastName: 'Modi',
            email: 'namo@dal.ca',
            password: 'namo@12345',
            city: 'Delhi',
        };

        // Send a POST request to the '/create' endpoint with the user data
        const response = await request(app)
            .post('/create')
            .send(userData);

        // Assert the response
        expect(response.status).toBe( expected: 200); // Expect the response status to be 200 (OK)
        expect(response.body).toEqual(expect.anything()); // Expect the response body to be defined and not empty
    });
});
```

Fig 21: Test Case for Registration API

```
  console.log
    {
      firstName: 'Narendra',
      lastName: 'Modi',
      at log (app.js:28:17)

 PASS  ./app.test.js
  Registration
    √ should register a new user (1034 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        1.931 s, estimated 2 s
Ran all test suites.
Jest did not exit one second after the test run has completed.
```

Fig 22: Shows Test Cases Passed for Registration API

```
describe('Login', () : void  => {
    test('should login a user', async () : Promise<void>  => {
        // Prepare test data
        const email : string  = 'namo@dal.ca';
        const password : string  = 'namo@12345';

        // Create a user in the database
        const userJson : {...}  = {
            Name: 'Narendra',
            Location: 'Delhi',
            Email: email,
            Password: await bcrypt.hash(password,  salt: 10)
        };
        await usersDb.doc(email).set(userJson);

        // Perform the login request
        const response = await request(app)
            .post('/login')
            .send({ email, password });

        // Assert the response
        expect(response.status).toBe( expected: 200);
        expect(response.body.userData).toBeDefined();
        expect(response.body.userData.Email).toBe(email);
        expect(response.headers['set-cookie']).toBeDefined();
        expect(response.headers['set-cookie'][0]).toContain( expected: 'sessionID');

        // Cleanup: Delete the user from the database
        await usersDb.doc(email).delete();
    });
```

Fig 23: Test Case for Login API

```
test('should return 404 for non-existing user', async () : Promise<void>  => {
    // Perform the login request with a non-existing user
    const response = await request(app)
        .post('/login')
        .send({ email: 'nonexisting@example.com', password: 'password123' });

    // Assert the response
    expect(response.status).toBe( expected: 404);
    expect(response.body.message).toBe( expected: 'User not found');
});

test('should return 401 for invalid password', async () : Promise<void>  => {
    // Prepare test data
    const email : string  = 'namo@dal.ca';
    const password : string  = 'namo@12345';

    // Create a user in the database
    const userJson : {...}  = {
        Name: 'Narendra',
        Location: 'Delhi',
        Email: email,
        Password: await bcrypt.hash(password,  salt: 10)
    };
    await usersDb.doc(email).set(userJson);

    // Perform the login request with an invalid password
    const response = await request(app)
        .post('/login')
        .send({ email, password: 'wrongpassword' });

    // Assert the response
    expect(response.status).toBe( expected: 401);
    expect(response.body.message).toBe( expected: 'Invalid password');

    // Cleanup: Delete the user from the database
    await usersDb.doc(email).delete();
});
```

Fig 24: More Test Cases on Login API

```
PASS  ./app.test.js
  Login
    √ should login a user (1231 ms)
    √ should return 404 for non-existing user (87 ms)
    √ should return 401 for invalid password (466 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        2.678 s, estimated 4 s
Ran all test suites.
Jest did not exit one second after the test run has completed.
```

Fig 25: Test Cases Passed for Login

```
const request : function((Function | Server), ... | {...}  = require('supertest');
const app = require('./app');

// Mocking Firebase Admin SDK
jest.mock('firebase-admin', () : {...}  => {
    // Mock Firestore functionality
    const firestore : {...}  = {
        collection: jest.fn(() : {...}  => firestore),
        doc: jest.fn(() : {...}  => firestore),
        where: jest.fn(() : {...}  => firestore),
        get: jest.fn(() : Promise<{...}>  => Promise.resolve( value: { docs: [] })),
        update: jest.fn(() : Promise<void>  => Promise.resolve()),
    };

    return {
        initializeApp: jest.fn(),
        firestore: jest.fn(() : {...}  => firestore),
        credential: {
            cert: jest.fn(),
        },
    };
});

```

Fig 26: Mock for Test cases for Status container

```
// Test suite for Online Users
describe('Online Users', () : void  => {
    // Test case: should retrieve online users
    it('should retrieve online users', async () : Promise<void>  => {
        // Mock data for online users
        const mockData : [{email: string, status: strin...  = [
            {
                email: 'gaurav@nirma.in',
                status: 'online',
                timestamp: '2023-07-04T19:08:50.987Z',
            },
        ];

        // Mock the Firestore "get" method to return the mock data
        const mockGet = jest.fn(() : Promise<{...}>  => Promise.resolve( value: { docs: mockData }));

        // Mock the app.locals with the necessary mock implementations
        app.locals = {
            stateDb: {
                where: jest.fn(() => app.locals.stateDb),
                get: mockGet,
            },
            usersDb: {
                doc: jest.fn(() => app.locals.usersDb),
                get: jest.fn(() : Promise<{...}>  => Promise.resolve( value: { data: () : {Location: string, Name: string}  => ({ Name: 'gaurav panchal', Location: 'ahmedabad' }) }
            },
        };

        // Make a request to the "/online-users" endpoint
        const response : string[]  = await request(app).get('/online-users');

        // Assert that the response status is 200
        expect(response.status).toBe( expected: 200);
    });
});
```

Fig 27: Test Cases for Status Container

```
PASS  ./app.test.js
  Online Users
    √ should retrieve online users (43 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        0.75 s, estimated 2 s
Ran all test suites.
Jest did not exit one second after the test run has completed.
```

Fig 28: Test Cases Successfully Passed for Status Contianer

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    CODEWHISPERER REFERENCE LOG

> react-scripts build

Creating an optimized production build...
One of your dependencies, babel-preset-react-app, is importing the
"@babel/plugin-proposal-private-property-in-object" package without
declaring it in its dependencies. This is currently working because
"@babel/plugin-proposal-private-property-in-object" is already in your
node_modules folder for unrelated reasons, but it may break at any time.

babel-preset-react-app is part of the create-react-app project, which
is not maintianed anymore. It is thus unlikely that this bug will
ever be fixed. Add "@babel/plugin-proposal-private-property-in-object" to
your devDependencies to work around this error. This will make this message
go away.

Compiled successfully.

File sizes after gzip:

 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/node:14
 => [auth] library/node:pull token for registry-1.docker.io
 => [1/5] FROM docker.io/library/node:14@sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa
 => [internal] load build context
 => => transferring context: 354.79MB
 => CACHED [2/5] WORKDIR /app
 => [3/5] COPY package*.json ./
 => [4/5] RUN npm install
 => [5/5] COPY . .
 => exporting to image
 => => exporting layers
 => => writing image sha256:e3df3517322b08025b1b5033fa133f0406ec540392ed5ad8de3b1ab8bd20a4d8
 => => naming to us-central1-docker.pkg.dev/sigma-rarity-378302/my-docker-repo/front-end:latest
PS C:\Users\fenil\OneDrive\Desktop\Serverless\Assignment 2\Tutorial3> docker push  us-central1-docker.pkg.dev/sigma-rarity-378302/my-docker-repo/front-end:latest
The push refers to repository [us-central1-docker.pkg.dev/sigma-rarity-378302/my-docker-repo/front-end]
14994d5082eb: Pushed
94239d2f6dad: Pushed
f8fd42ba3471: Pushed
0d5f5a015e5d: Pushed
3c777d951de2: Pushed
f8a91dd5fc84: Pushed
cb81227abde5: Pushed
e01a454893a9: Pushed
c45660adde37: Pushed
fe0fb3ab4a0f: Pushed
f1186e5061f2: Pushed
b2dba7477754: Pushed
latest: digest: sha256:e34c95023f5e86fe1e2c303b9fb825f34f4238320e2d6b4201ec2d84117f5687 size: 3056
```

Fig 29: Building and Pushing Front-End docker image into Artifact Registry

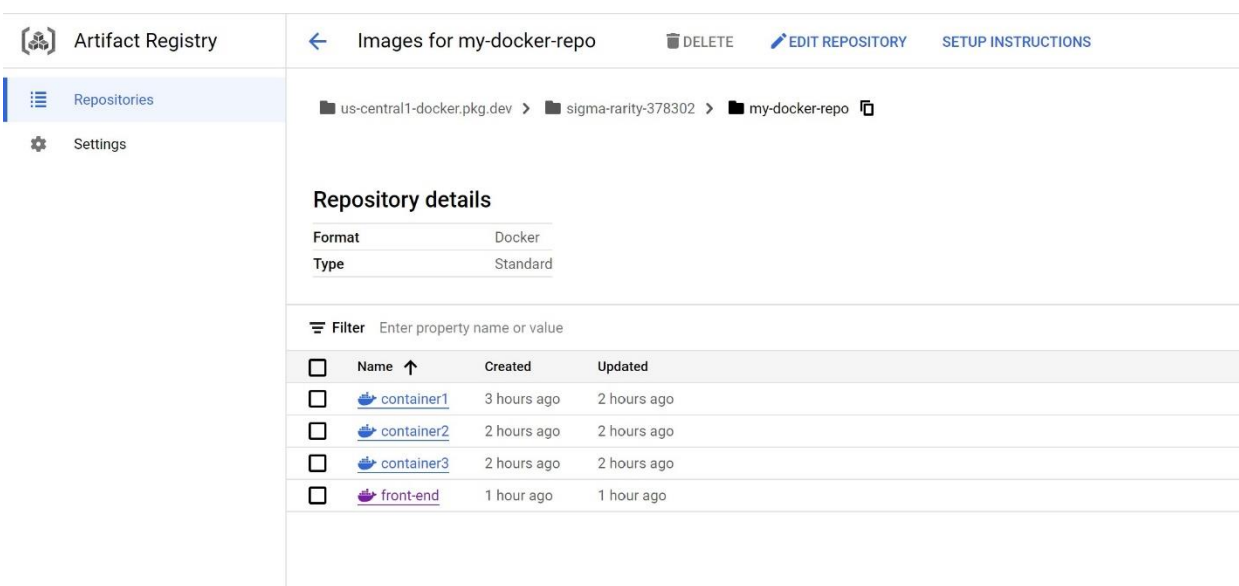


Fig 30: Docker Image is uploaded into Artifact Registry

```
 => => transferring dockerfile: 443B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/node:18.14.0
 => [auth] library/node:pull token for registry-1.docker.io
 => [1/6] FROM docker.io/library/node:18.14.0@sha256:5ab5e06df2c58dabb18b963636c2fab16b10b7150d13434a12024f599aa645bc
 => [internal] load build context
 => => transferring context: 80.69MB
 => CACHED [2/6] WORKDIR /app
 => [3/6] COPY . .
 => [4/6] RUN npm uninstall bcrypt
 => [5/6] RUN npm install bcryptjs
 => [6/6] RUN npm install bcrypt
 => exporting to image
 => => exporting layers
 => => writing image sha256:77702e6fdc05aa2e0cc0d89f3a964a31ec1ba2e7fc4c160c74c48767a449dc48
 => => naming to us-central1-docker.pkg.dev/sigma-rarity-378302/my-docker-repo/container1:latest
PS C:\Users\fenil\OneDrive\Desktop\Serverless\Assignment 2\registration> docker push  us-central1-docker.pkg.dev/sigma-rarity-378302/my-docker-repo/container1:latest
The push refers to repository [us-central1-docker.pkg.dev/sigma-rarity-378302/my-docker-repo/container1]
a233209af893: Pushed
adae1e0e4198: Pushed
2f53357e2903: Pushed
f15798e3fa04: Pushed
7adc16751dc3: Layer already exists
70c5854ab5fe: Layer already exists
a7e5b414b5d8: Layer already exists
aff26125b01e: Layer already exists
4c92897e605e: Layer already exists
0b6859e9fff1: Layer already exists
11829b3be9c0: Layer already exists
dc8e1d8b53e9: Layer already exists
9d49e0bc68a4: Layer already exists
8e396a1aad50: Layer already exists
latest: digest: sha256:806fe514a37069b7c86cb15b08daf8789b162d094c621aaef7e0d479609679c3 size: 3265
PS C:\Users\fenil\OneDrive\Desktop\Serverless\Assignment 2\registration>
```

Fig 31: Docker Image of Registration code is built and being pushed to Artifact Registry

```
 => exporting to image
 => => exporting layers
 => => writing image sha256:01998889889380087bd8213485c49a49156ca67024f6f1309403b64677d0c9b3
 => => naming to us-central1-docker.pkg.dev/sigma-rarity-378302/my-docker-repo/container2:latest
PS C:\Users\fenil\OneDrive\Desktop\Serverless\Assignment 2\login> docker push us-central1-docker.pkg.dev/sigma-rarity-378302/my-docker-repo/container2:latest
The push refers to repository [us-central1-docker.pkg.dev/sigma-rarity-378302/my-docker-repo/container2]
782c51e652b7: Pushed
86a68a1be15c: Pushed
50125fa2eef0: Pushed
8e6611318f0d: Pushed
7adc16751dc3: Layer already exists
70c5854ab5fe: Layer already exists
a7e5b414b5d8: Layer already exists
aff26125b01e: Layer already exists
4c92897e605e: Layer already exists
0b6859e9fff1: Layer already exists
11829b3be9c0: Layer already exists
dc8e1d8b53e9: Layer already exists
9d49e0bc68a4: Layer already exists
8e396a1aad50: Layer already exists
latest: digest: sha256:0e987696ae6b36406f9a872d12077d98d18fa221c82dd7054d7800312a338a16 size: 3265
PS C:\Users\fenil\OneDrive\Desktop\Serverless\Assignment 2\login>
```

Fig 32: Docker Image of the Login Code is built and being Pushed into Artifact Registry

```
PS C:\Users\fenil\OneDrive\Desktop\Serverless\Assignment 2\state> docker build -t us-central1-docker.pkg.dev/sigma-rarity-378302/my-docker-repo/container3:latest .
[+] Building 18.4s (12/12) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 441B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/node:18.14.0
 => [auth] library/node:pull token for registry-1.docker.io
 => [1/6] FROM docker.io/library/node:18.14.0@sha256:5ab5e06df2c58dabb18b963636c2fab16b10b7150d13434a12024f599aa645bc
 => [internal] load build context
 => => transferring context: 81.93MB
 => CACHED [2/6] WORKDIR /app
 => [3/6] COPY . .
 => [4/6] RUN npm uninstall bcrypt
 => [5/6] RUN npm install bcryptjs
 => [6/6] RUN npm install bcrypt
 => exporting to image
 => => exporting layers
 => => writing image sha256:021f3d45002c417ad865caf3f7a425a5919b47e105dd5120f7ac15e51abec40b
 => => naming to us-central1-docker.pkg.dev/sigma-rarity-378302/my-docker-repo/container3:latest
PS C:\Users\fenil\OneDrive\Desktop\Serverless\Assignment 2\state> docker push us-central1-docker.pkg.dev/sigma-rarity-378302/my-docker-repo/container3:latest
The push refers to repository [us-central1-docker.pkg.dev/sigma-rarity-378302/my-docker-repo/container3]
a1c851cad389: Pushed
59a1bc64cd2e: Pushed
792cf281168b: Pushed
13a516de65a9: Pushed
7adc16751dc3: Layer already exists
70c5854ab5fe: Layer already exists
a7e5b414b5d8: Layer already exists
aff26125b01e: Layer already exists
4c92897e605e: Layer already exists
0b6859e9fff1: Layer already exists
11829b3be9c0: Layer already exists
dc8e1d8b53e9: Layer already exists
9d49e0bc68a4: Layer already exists
8e396a1aad50: Layer already exists
latest: digest: sha256:232b4541e955b631496c7f9ed6e996273e9a77543440e681fb17a7817fe0a300 size: 3265
PS C:\Users\fenil\OneDrive\Desktop\Serverless\Assignment 2\state>
```

Fig 33: Docker Image of status code is built and being pushed to artifact registry

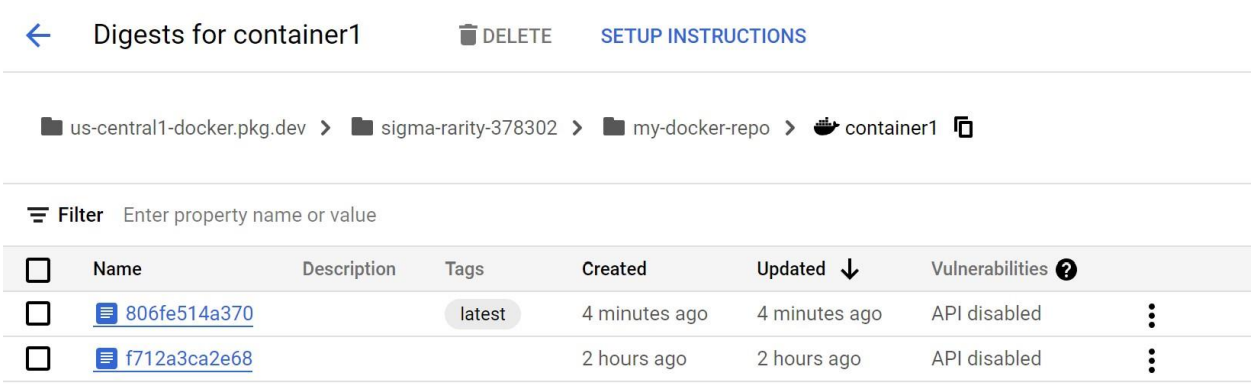


Fig 34: Docker Image of Container 1 in artifact Registry

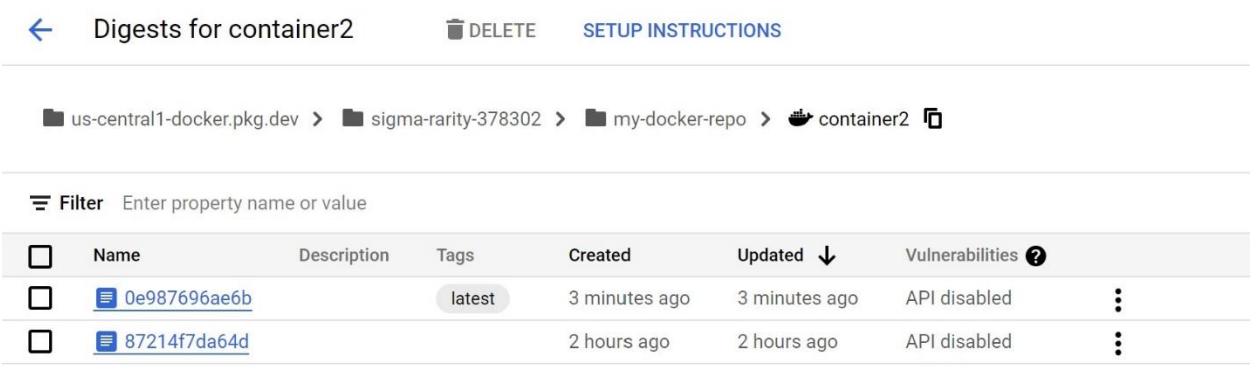


Fig 35: Docker Image of Container 2 in artifact Registry

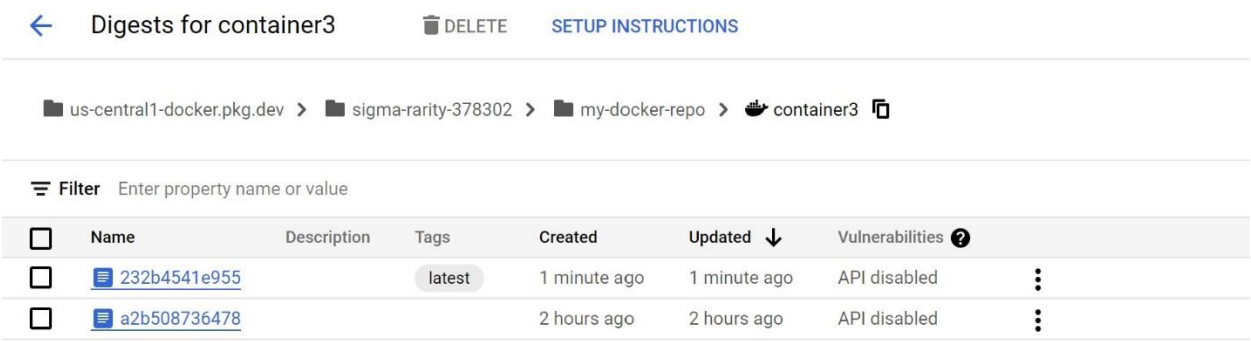


Fig 36: Docker Image of Container 3 in Artifact Registry

us-central1-docker.pkg.dev > sigma-rarity-378302 > my-docker-repo > front-end

Filter Enter property name or value

Name	Description	Tags	Created	Updated	Vulnerabilities
e34c95023f5e		latest	1 hour ago	1 hour ago	API disabled

Fig 37: Docker Image of front-end code into Artifact Registry

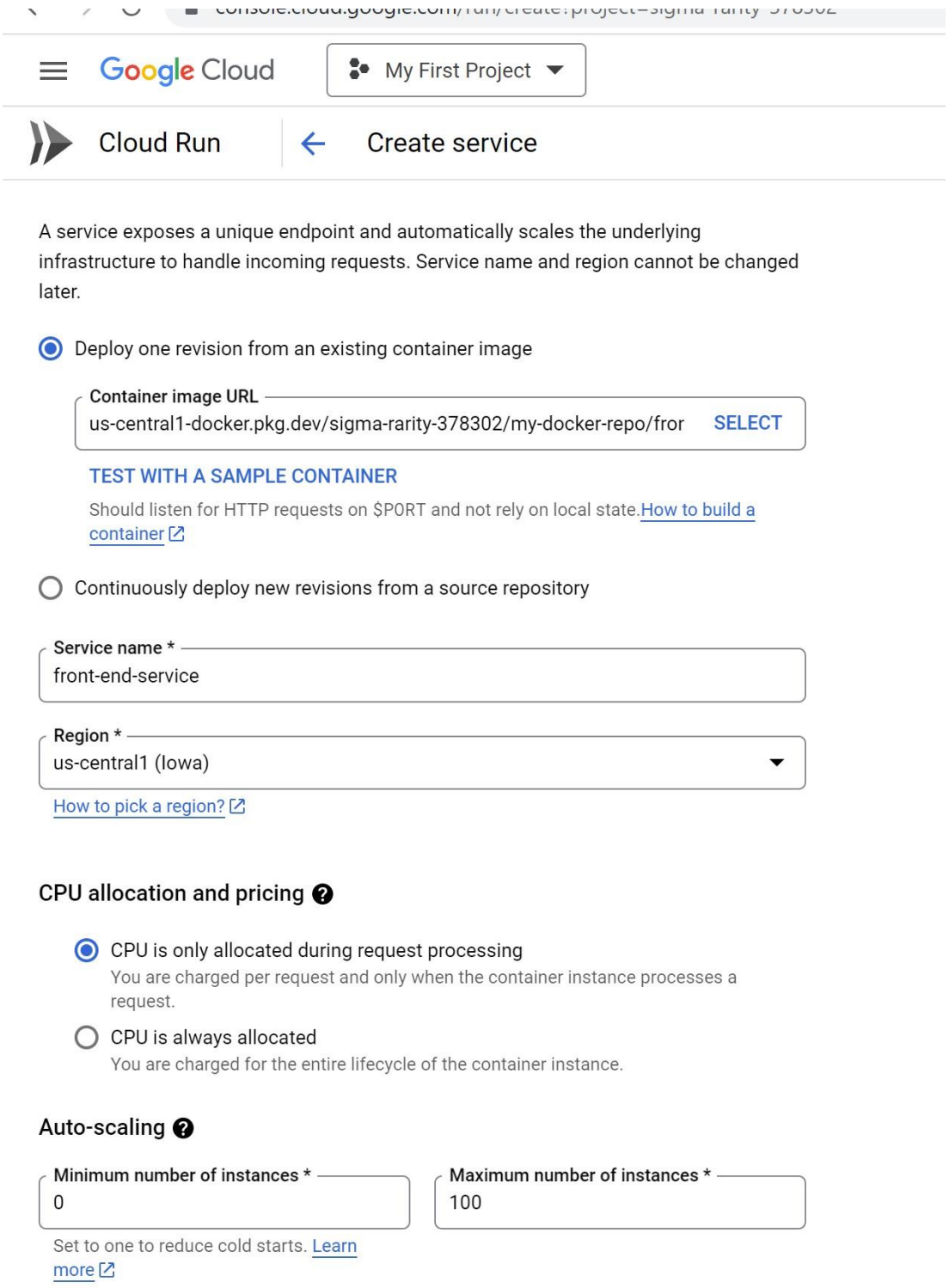


Fig 38: Creating a Cloud Run Service for Front-End Container image

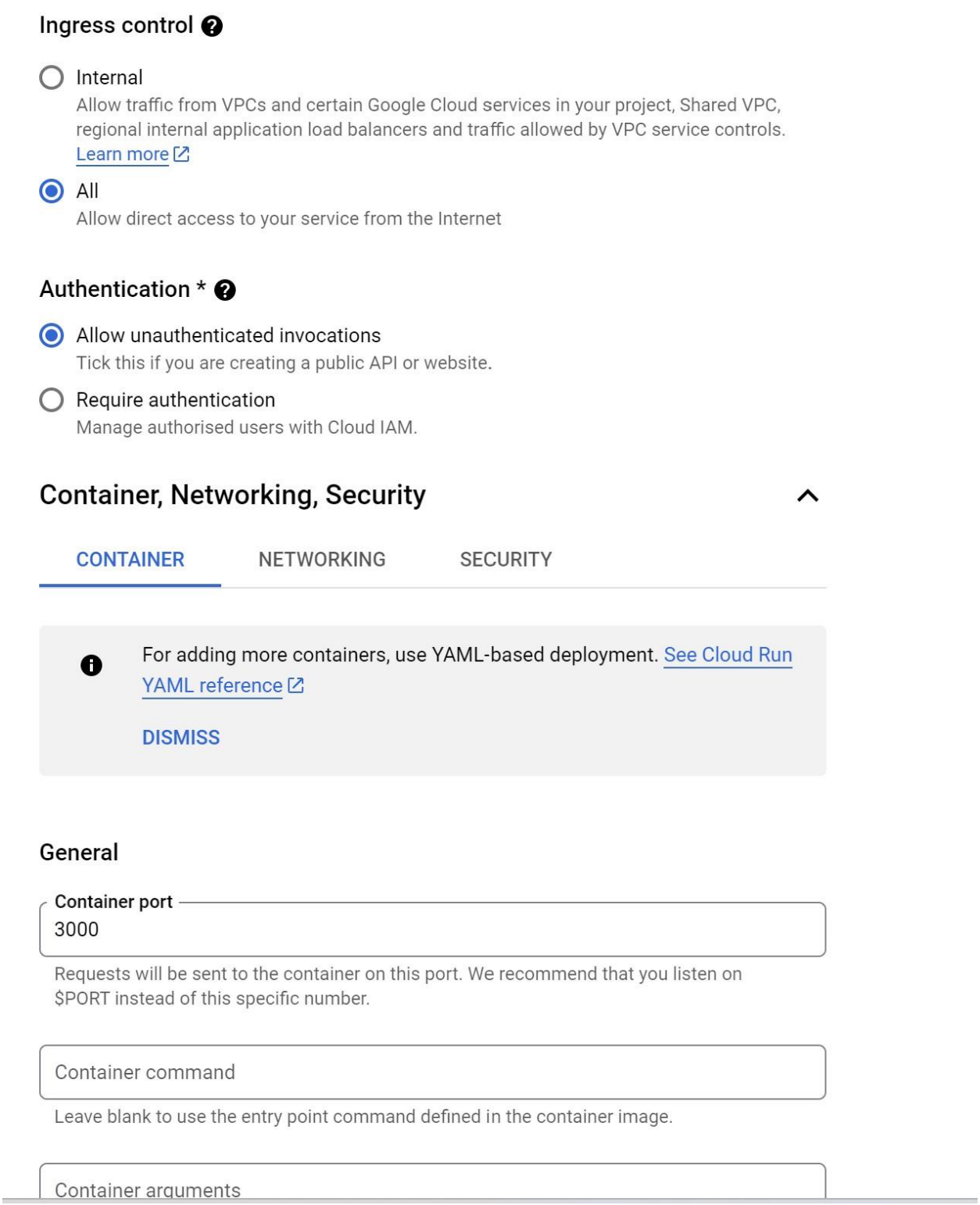


Fig 39: Configuring Front end Service

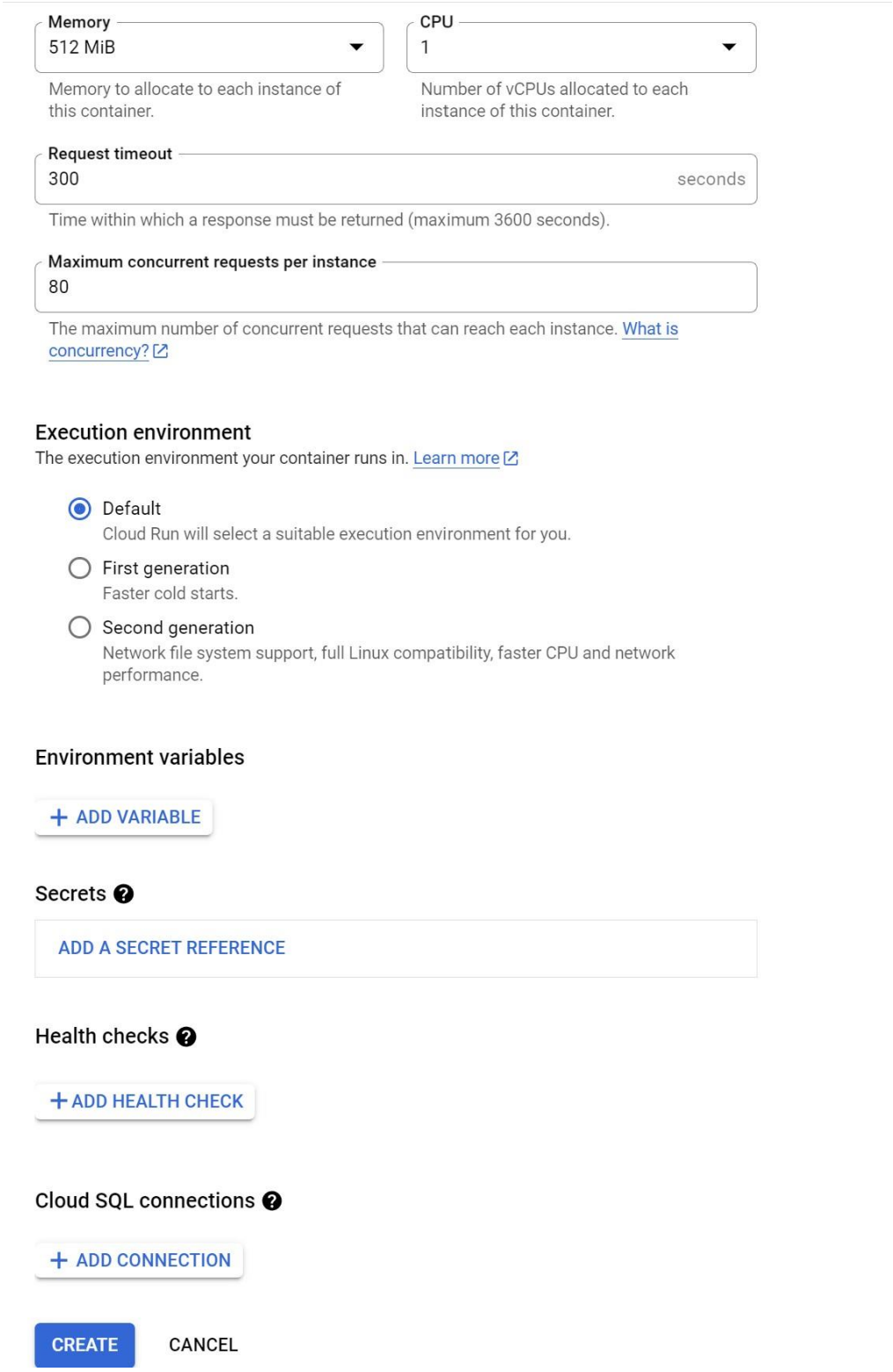


Fig 40 : Creating the Front end Service for Cloud Run

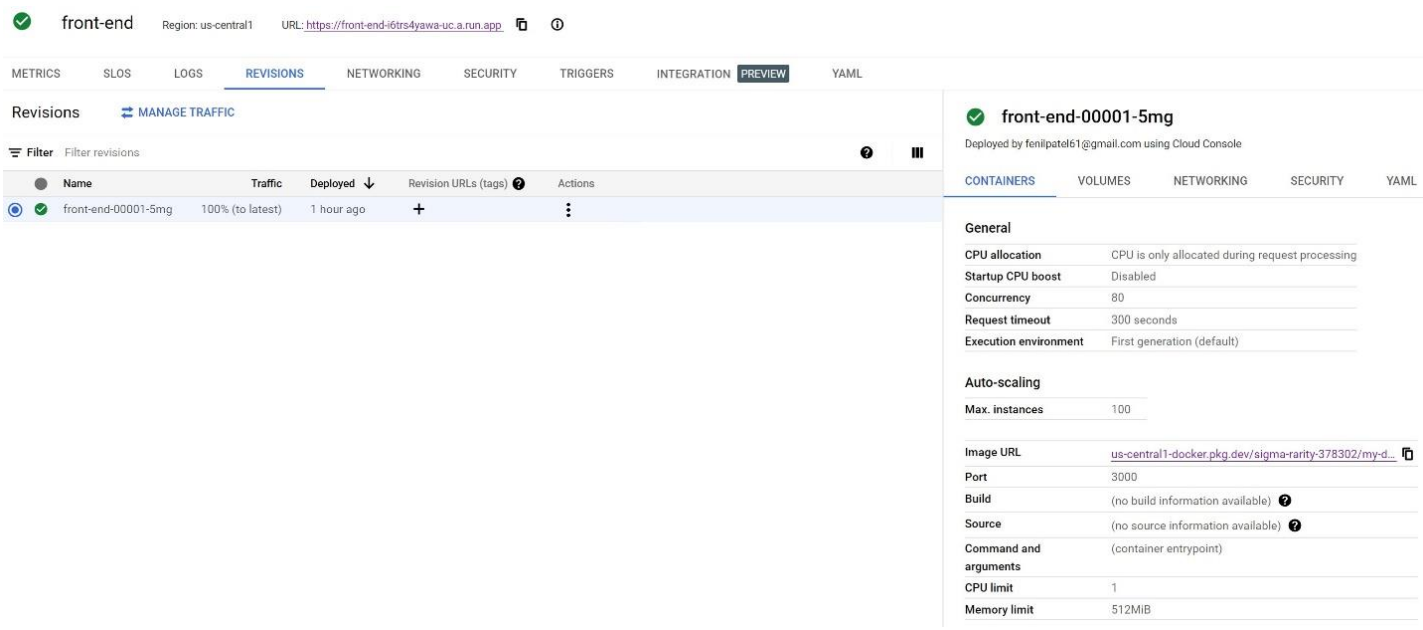


Fig 41: Front end service is running on cloud run at port 3000.

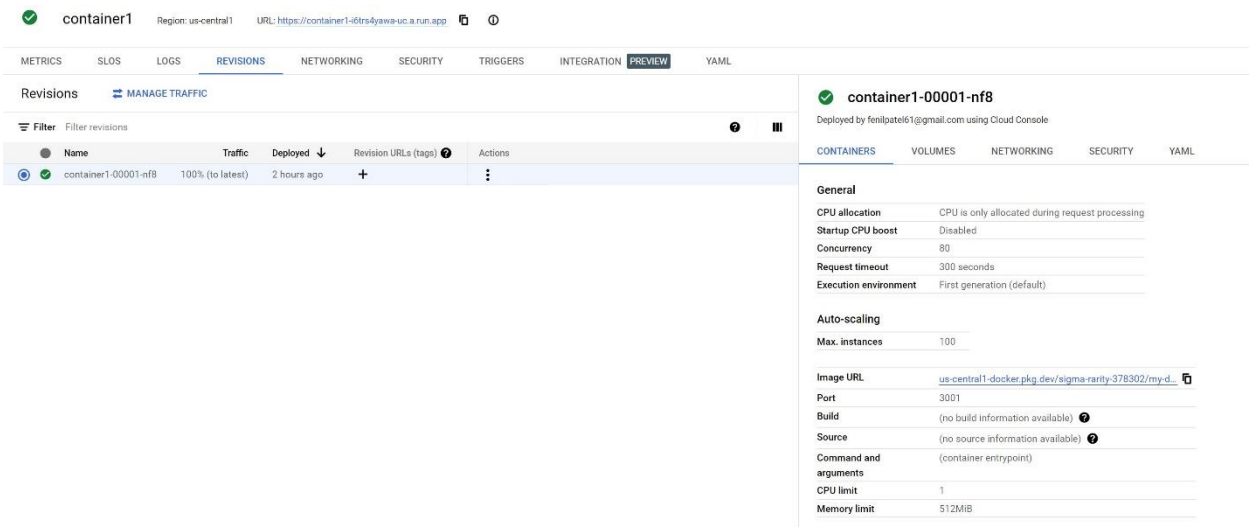


Fig 42: Same configuration followed for creating a cloud run service for container 1 Image, and it is running at port 3001

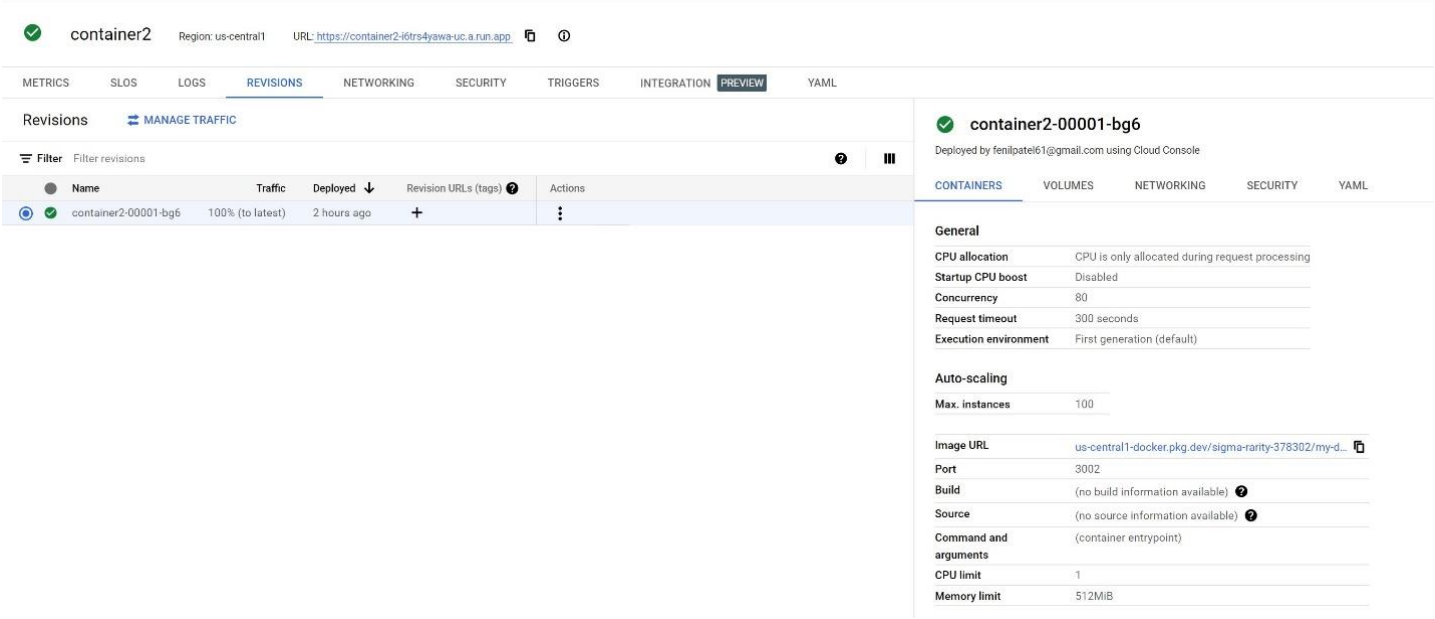


Fig 43: Same configuration followed for creating a cloud run service for container 2 Image, and it is running at port 3002

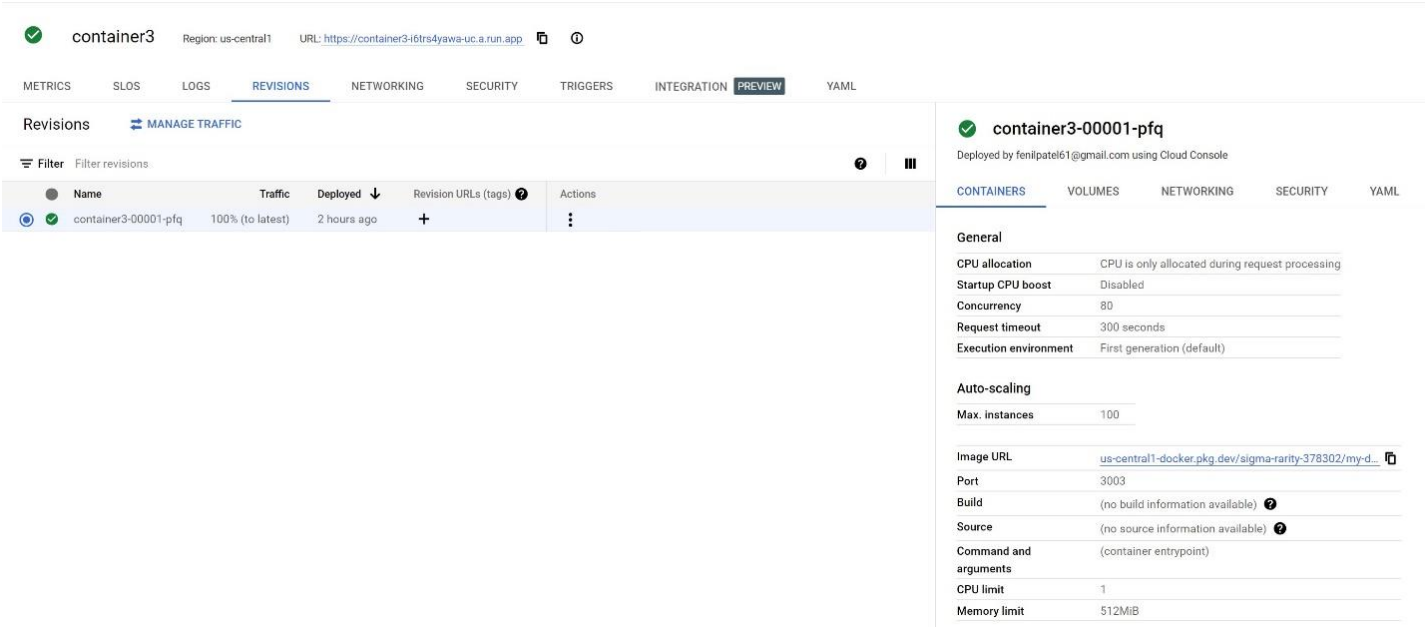
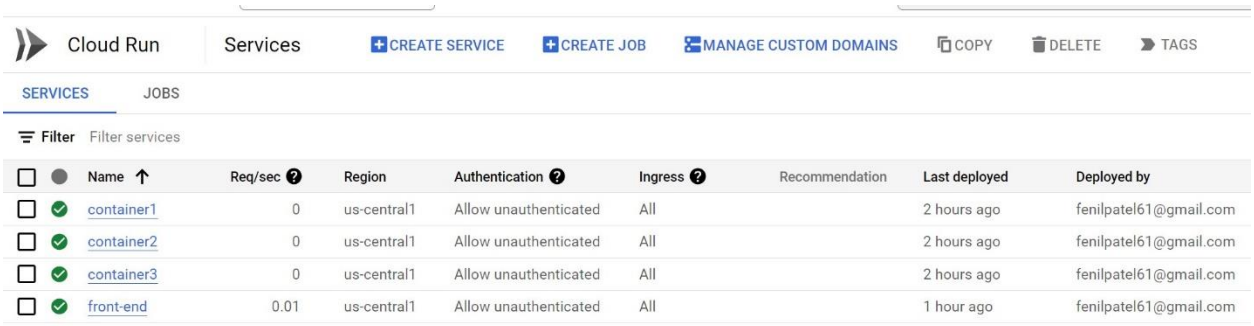


Fig 44: Same configuration followed for creating a cloud run service for container 2 Image, and it is running at port 3003



Name	Req/sec	Region	Authentication	Ingress	Recommendation	Last deployed	Deployed by
container1	0	us-central1	Allow unauthenticated	All		2 hours ago	fenilpatel61@gmail.com
container2	0	us-central1	Allow unauthenticated	All		2 hours ago	fenilpatel61@gmail.com
container3	0	us-central1	Allow unauthenticated	All		2 hours ago	fenilpatel61@gmail.com
front-end	0.01	us-central1	Allow unauthenticated	All		1 hour ago	fenilpatel61@gmail.com

Fig 45: All the Services running on Cloud Run for the Front end, all three containers

References:

Deploying Containers to Cloud Run :- https://medium.com/google-cloud/deploying-containers-to-cloud-run-in-5mins-b03f1d8d4a64

How To Write Unit Tests In NodeJS With JEST Test Library :- https://medium.com/bb-tutorials-and-thoughts/how-to-write-unit-tests-in-nodejs-with-jest-test-library-a201658829c7