BUILD, DEPLOY AND RUN A CONTAINERIZED APPLICATION USING GCP

CSCI 5410 – Serverless Data Processing
Assignment 2 –Part A

Dhrumil Amish Shah (B00857606)
dh416386@dal.ca

Project Git Repository

https://git.cs.dal.ca/dashah/csci-5410-f2021-b00857606-dhrumil-amish-shah/-/tree/main/assignment_2_code

Firebase Firestore Setup

Figures 1, 2, 3, 4, 5, and 6 display the screenshots of the configuration setup of the Firebase Firestore database. Initially, there is no data in the Firestore database.

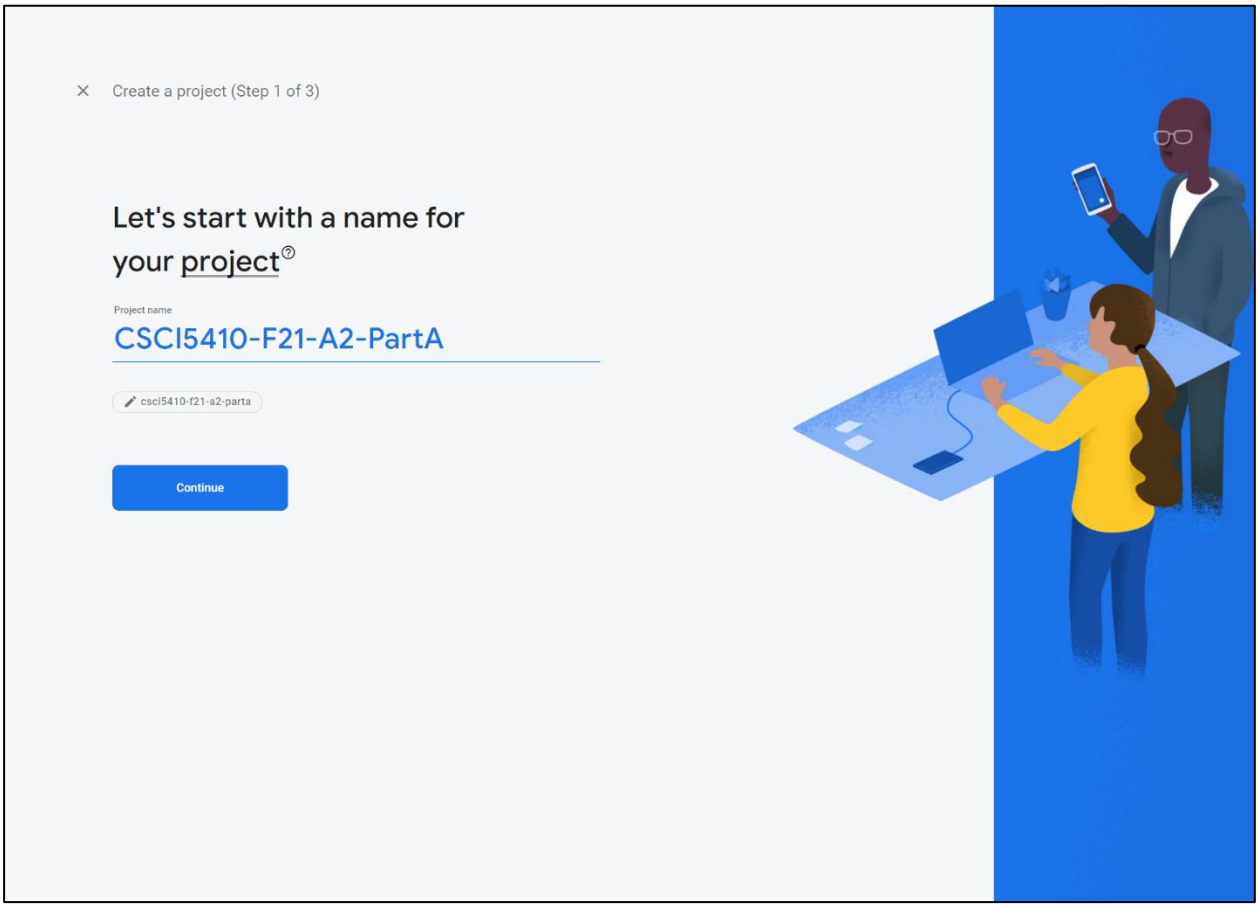

Figure 1 - Firebase Firestore project creation (Project name - CSCI5410-F21-A2-PartA) [1]

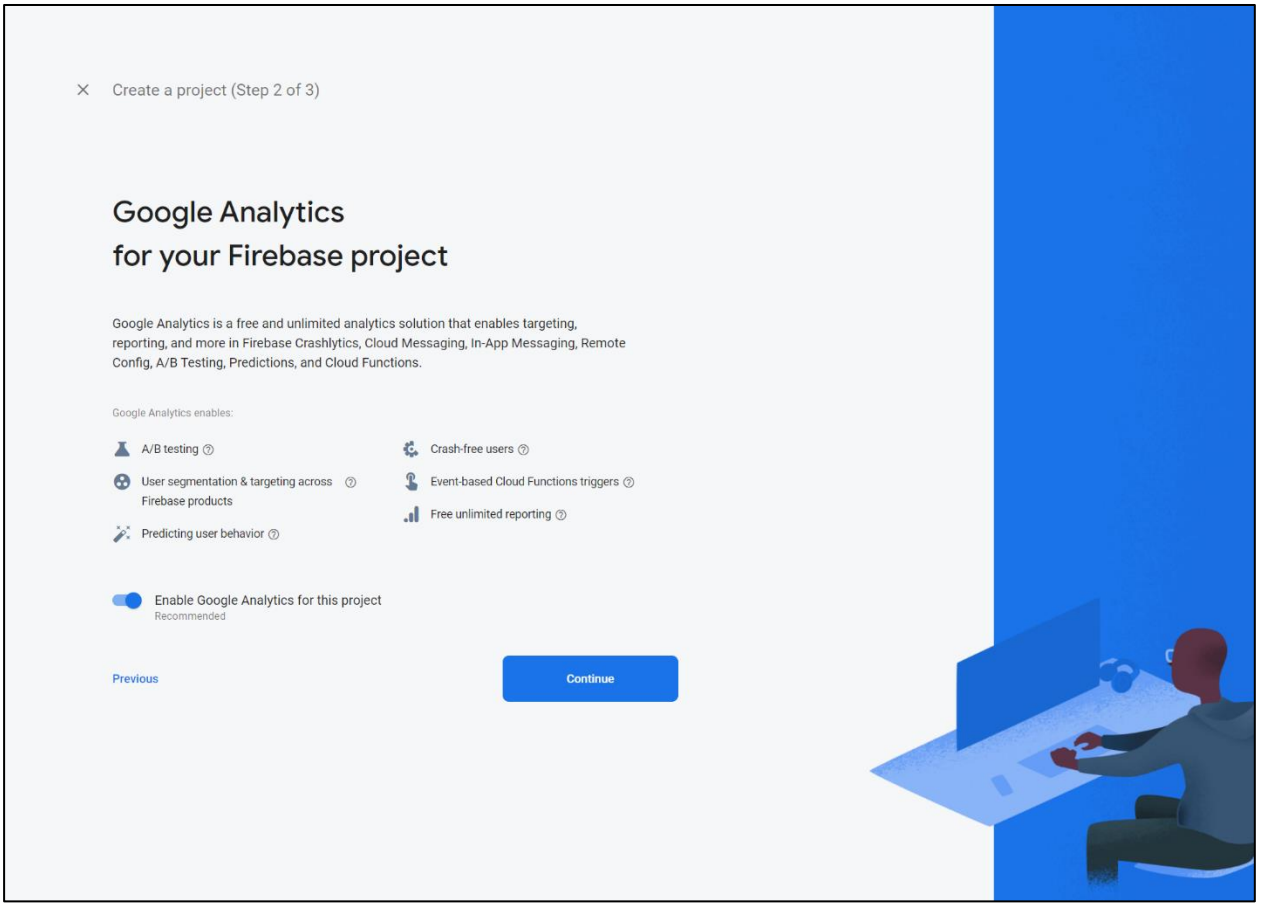

Figure 2 - Firebase Firestore project creation (Project name - CSCI5410-F21-A2-PartA) (contd.) [1]

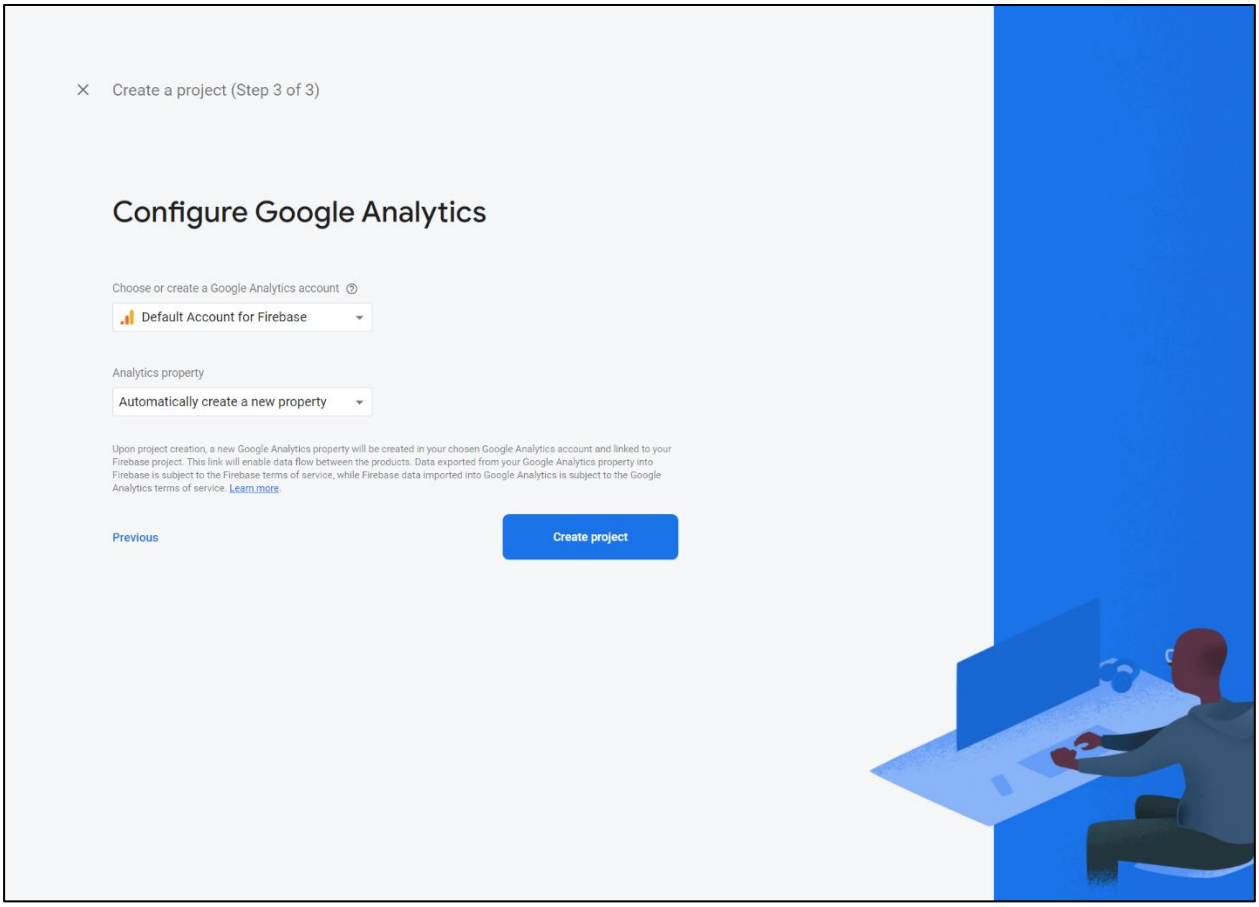

Figure 3 - Firebase Firestore project creation (Project name - CSCI5410-F21-A2-PartA) (contd.) [1]

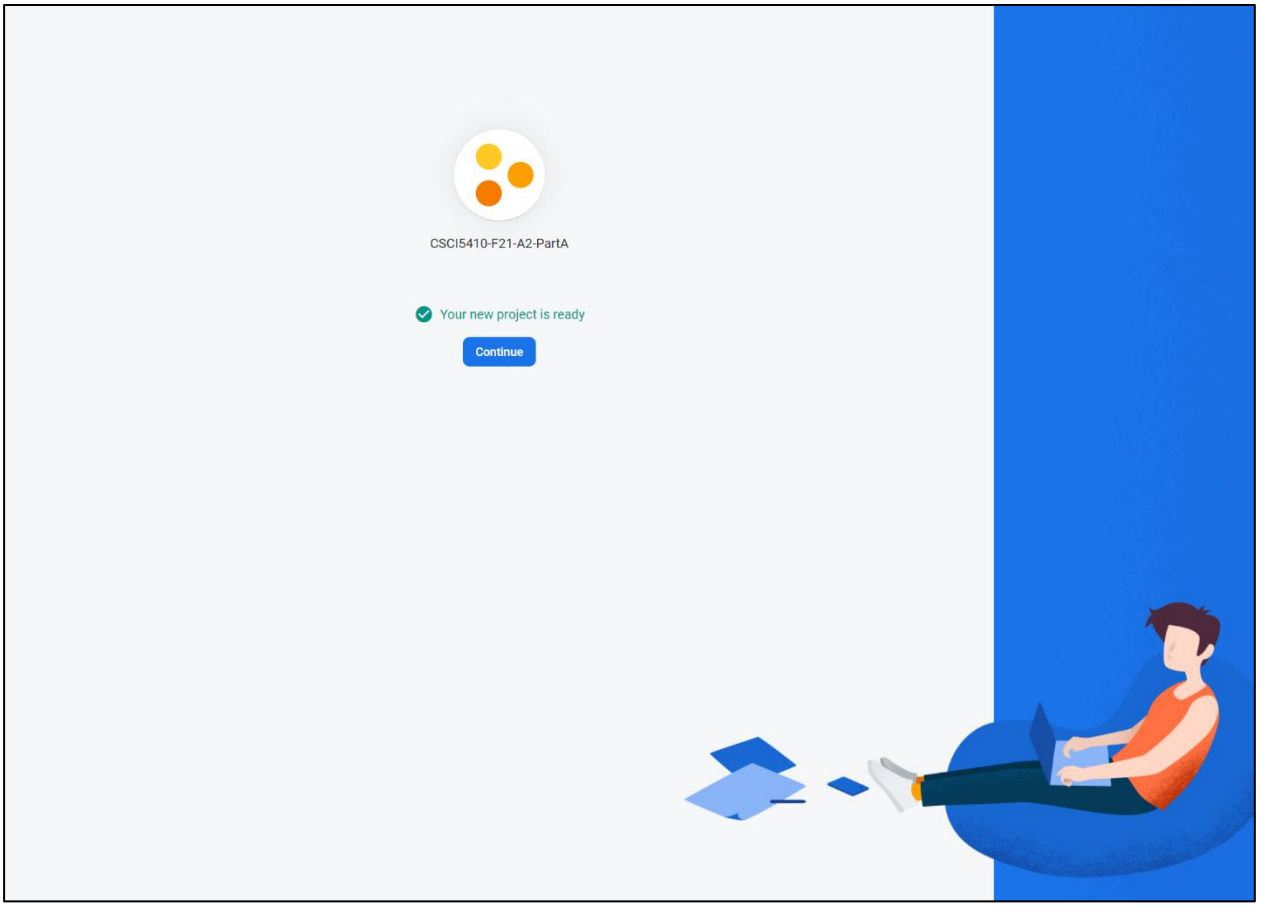

Figure 4 - Firebase Firestore project creation (Project name - CSCI5410-F21-A2-PartA) (contd.) [1]

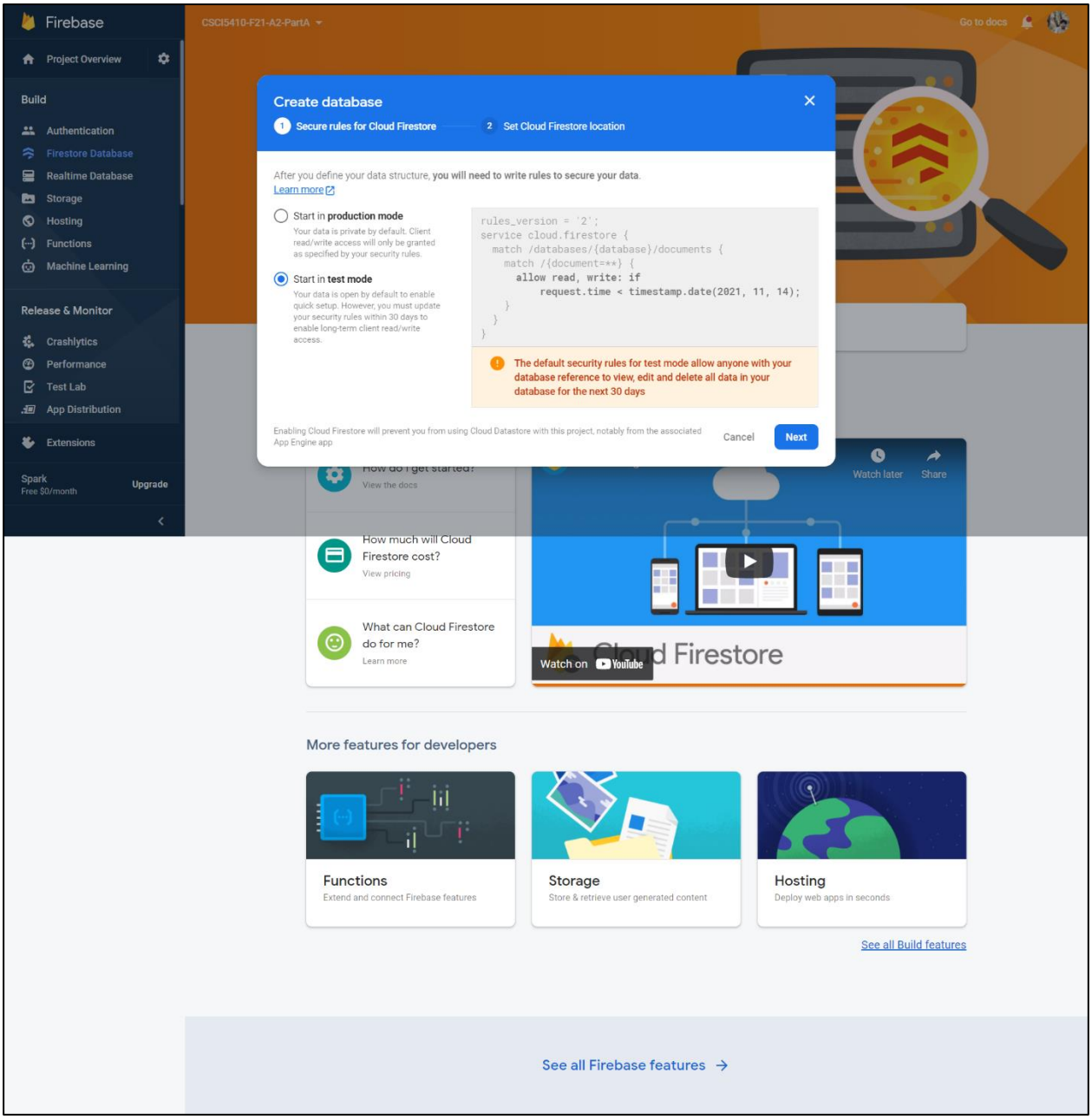

Figure 5 - Firebase Firestore project creation (Project name - CSCI5410-F21-A2-PartA) (contd.) [1]

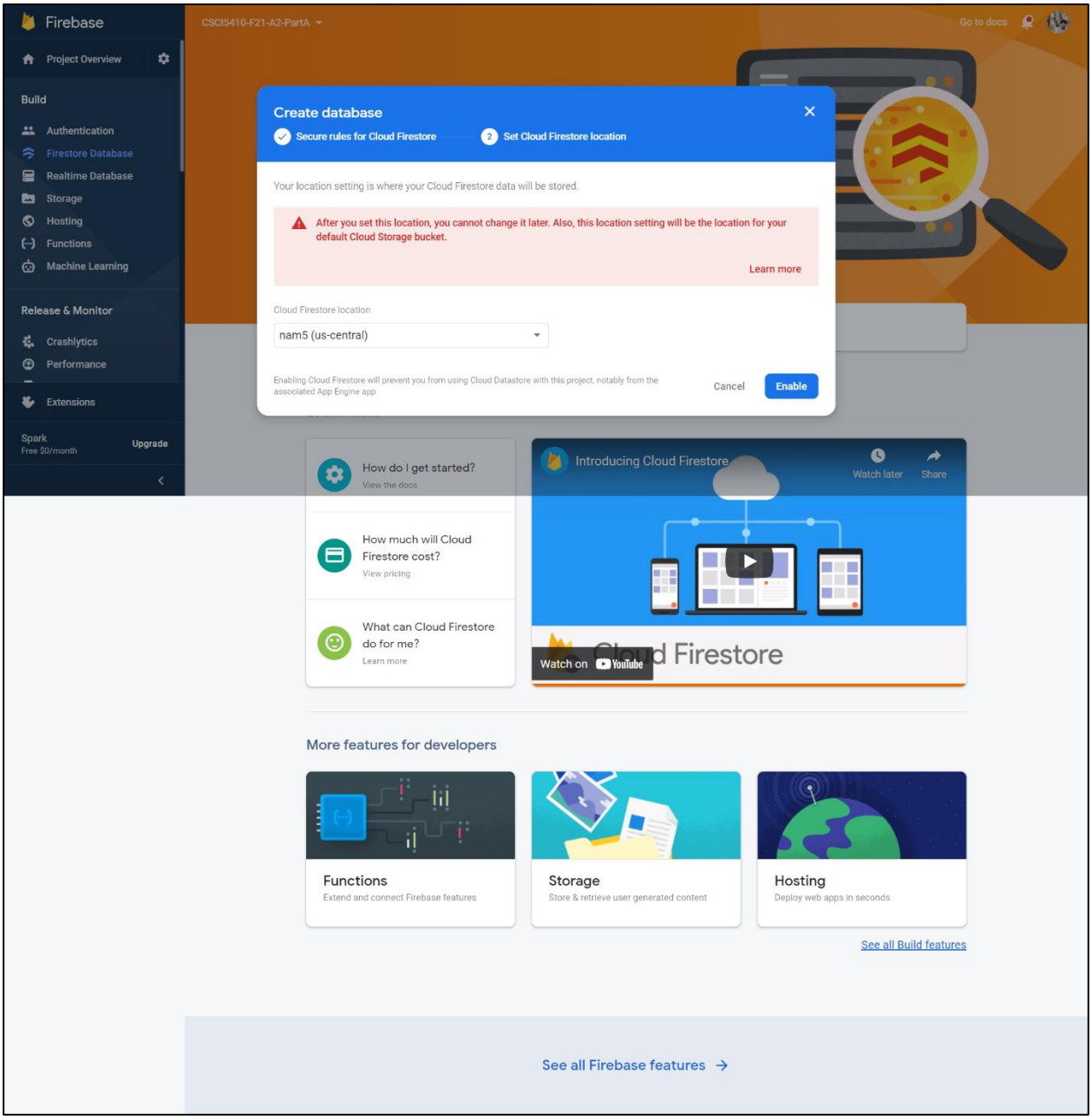

Figure 6 - Firebase Firestore project creation (Project name - CSCI5410-F21-A2-PartA) (contd.) [1]

Figure 7 displays the empty Firestore database (i.e., Database with no users registered).

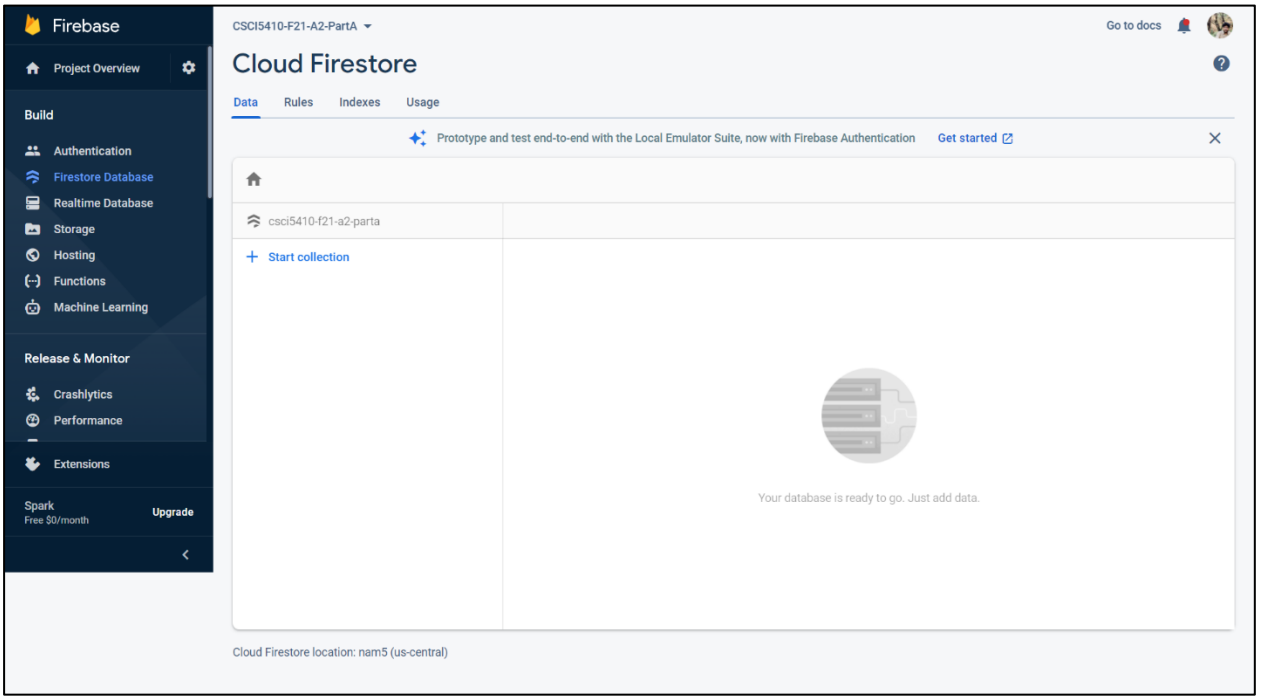

Figure 7 – Empty Firestore database (i.e., Database with no users) [1]

Container Images (Registration, Login, and Profile React applications)

Container 1 – Registration React Application

The **container_1_app** Registration React application is build using Docker. This application is responsible for accepting the registration information (i.e., first name, last name, email, and password) from a user. The accepted information is stored in the Firestore database. On successful registration, the user is redirected to the Login React application (i.e., **container_2_app** Login React application). Further, the status of the user registered is set to **offline**.

Figures 8 and 9 displays the Registration React application Docker image creation and successful creation of docker image in the dashboard. The name of the Docker image is **container_1_app**.

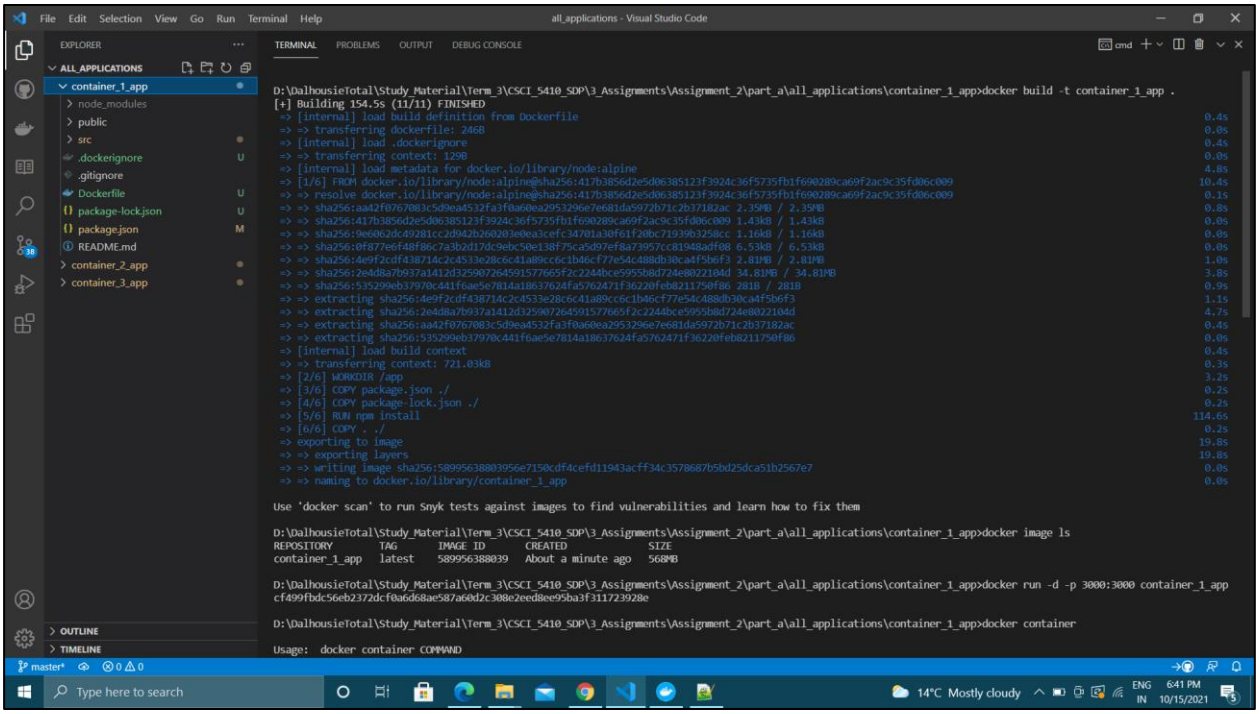

Figure 8 - Docker image creation of container_1_app Registration React application

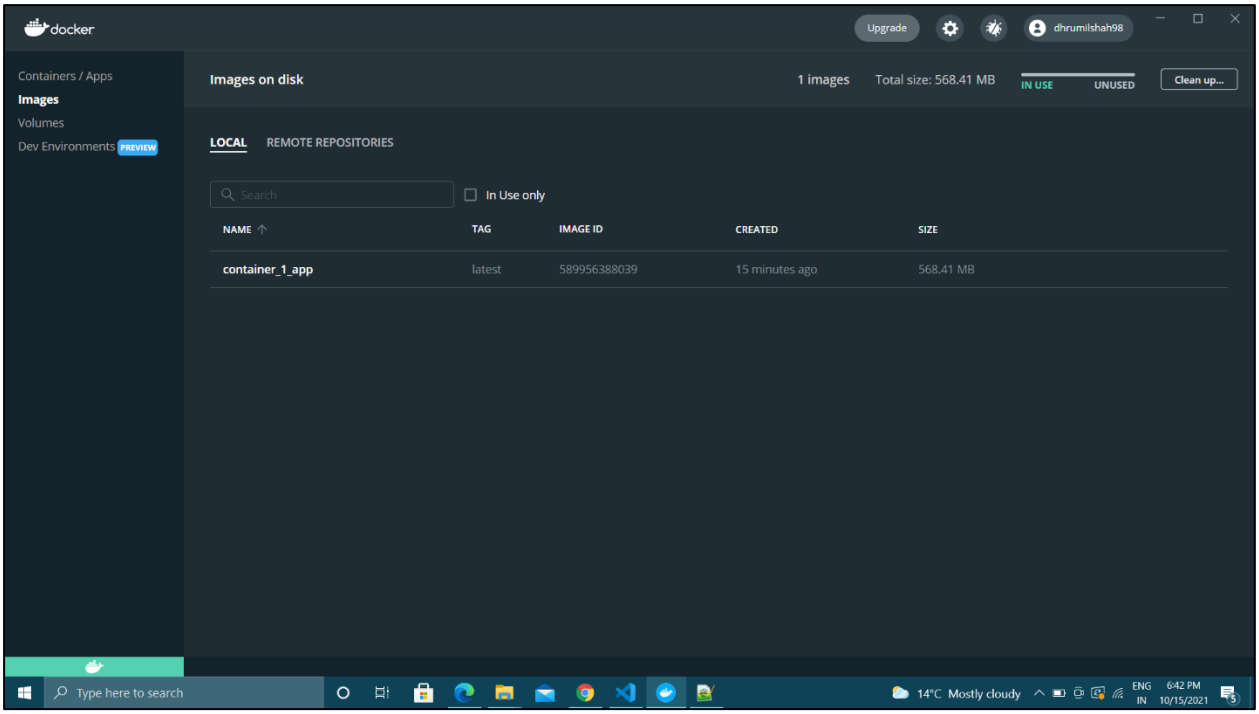

Figure 9 - Successful creation of Docker image of container_1_app Registration React application [2]

Container 2 – Login React Application

The **container_2_app** Login React application is build using Docker. This application is responsible for accepting the login information (i.e., email, and password), from a user and validate the information. The accepted information is verified in the Firestore database. On successful login, the user is redirected to the profile React application (i.e., **container_3_app** Profile React application). Further, the status of the logged-in user is set to **online**.

Figures 10 and 11 display the Login React application Docker image creation and successful creation of docker image in the dashboard. The name of the image is **container_2_app**.

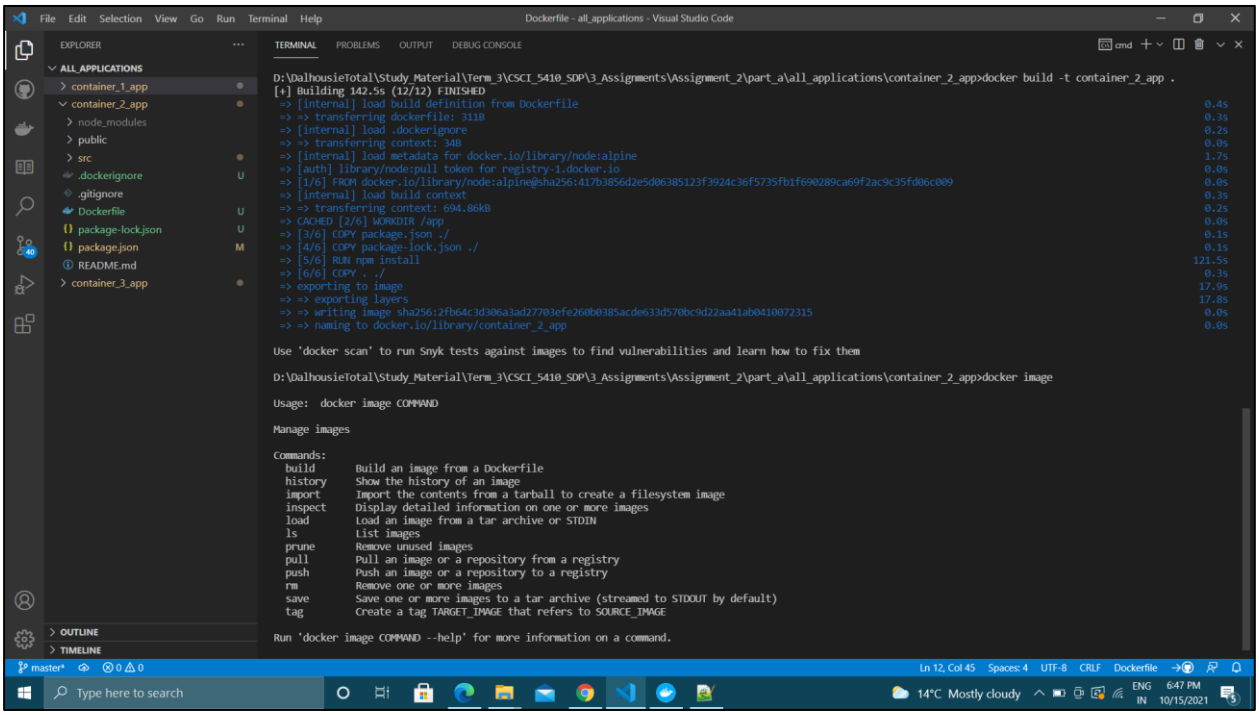

Figure 10 - Docker image creation of container_2_app Login React application

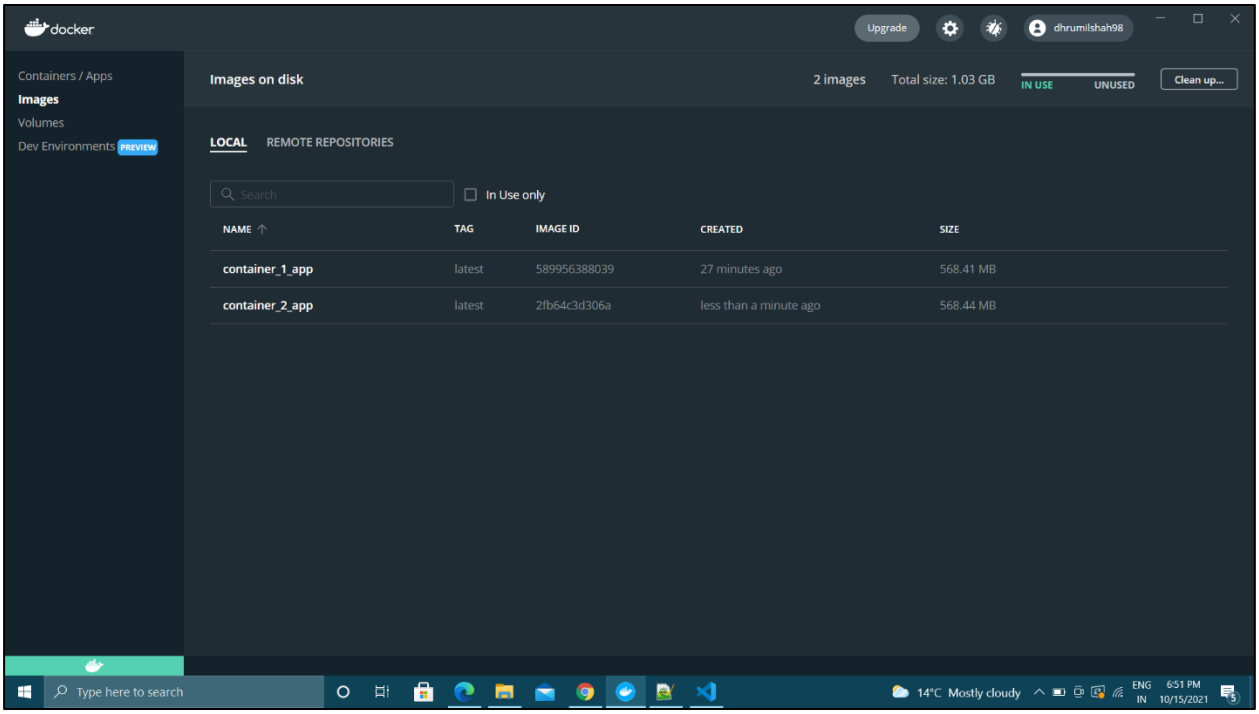

Figure 11 - Successful creation of Docker image of container_2_app Login React application [2]

Container 3 – Profile React Application

The **container_3_app** Profile React application is build using Docker. This application is responsible for displaying the profile information (i.e., first name, last name and email) of the logged-in user as well as the first name and last name of all the other online users. Also, there is a logout button that logs out the user and updates the status to **offline.** Further, on logout, the user is redirected back to the Login React application (i.e., **container_2_app** Login React application).

Figures 12 and 13 display the Profile React application Docker image creation and successful creation of docker image in the dashboard. The name of the image is **container_2_app**.

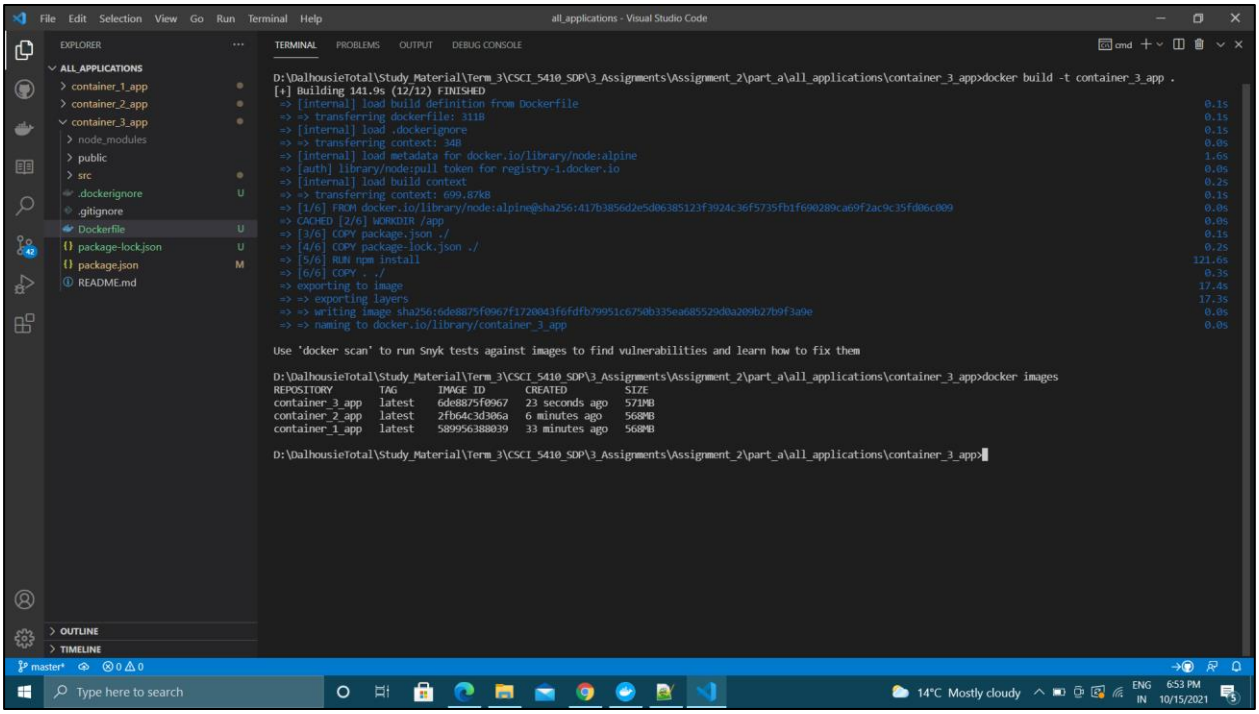

Figure 12 - Docker image creation of container_3_app Profile React application

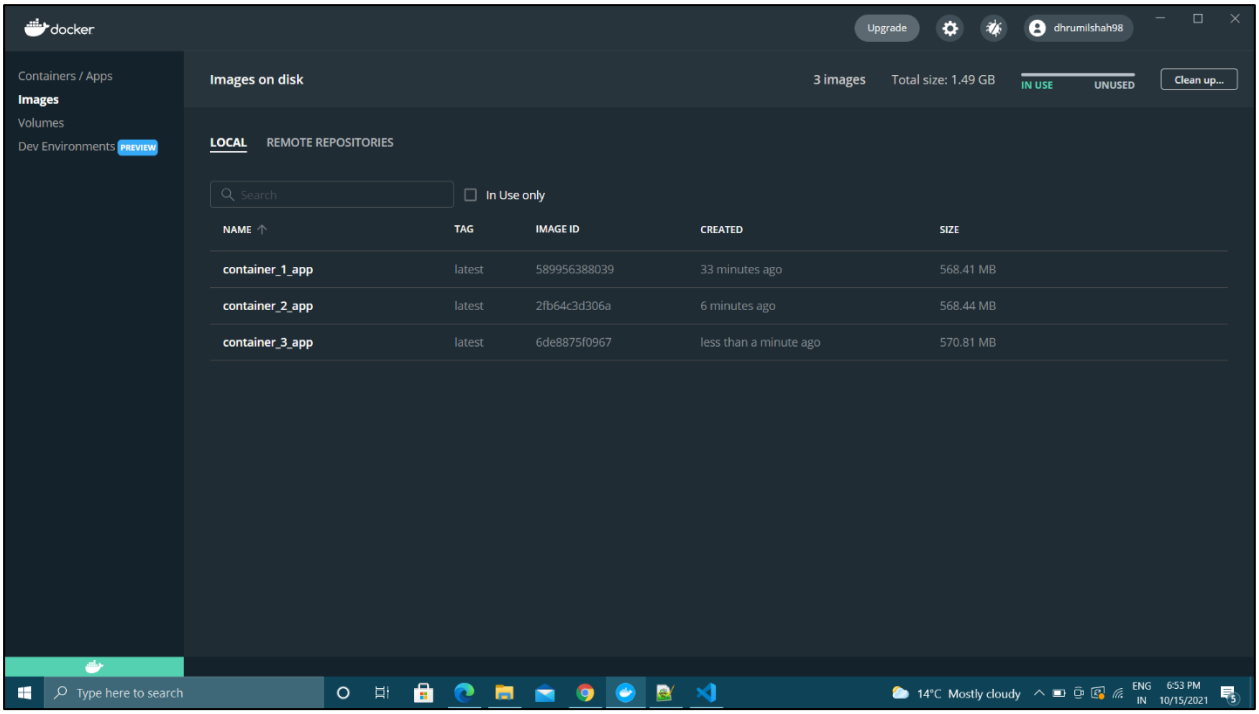

Figure 13 - Successful creation of Docker image of container_2_app Profile React application [2]

Running the Docker Images

Three Docker images namely Registration React application, Login React application and Profile React application runs on different ports namely 3000, 3001 and 3002 respectively.

Figure 14 displays the successful running of three Docker containers on three different ports.

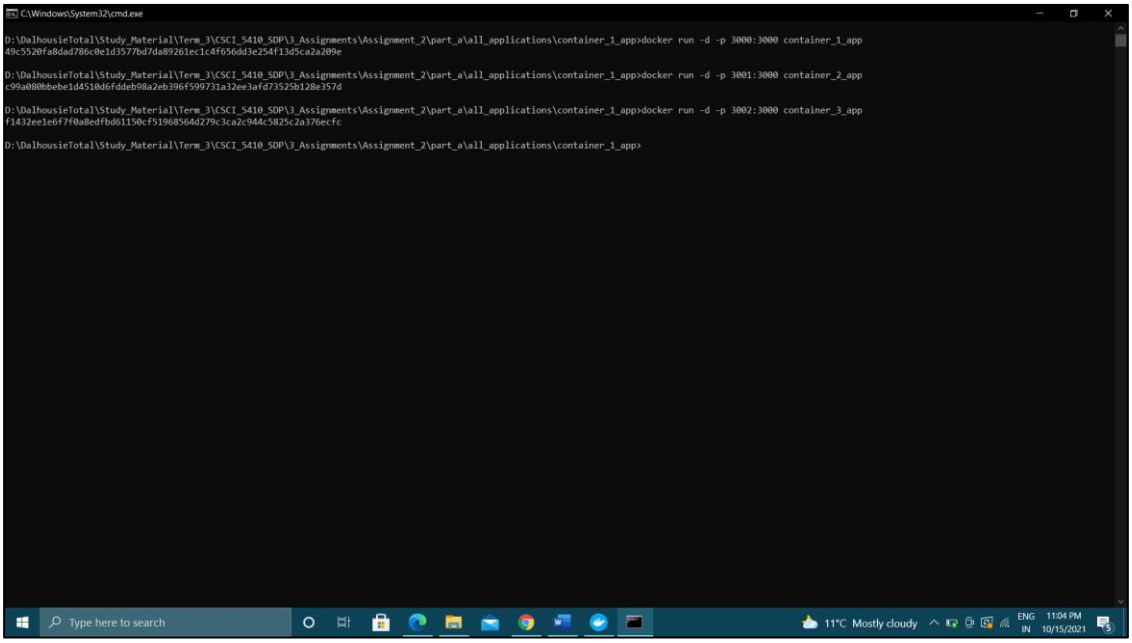

Figure 14 - Successful running of three Docker containers on ports 3000, 3001, and 3002 respectively

Figure 15 displays the successful running of three Docker containers in the Docker Dashboard.

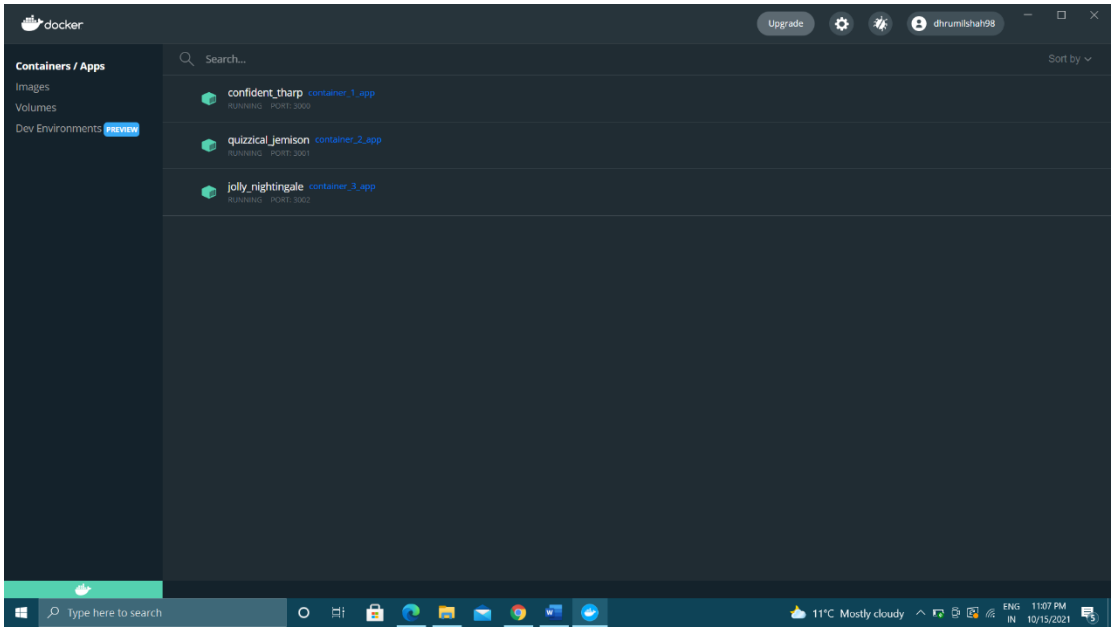

Figure 15 - Successful running of three Docker containers in the Docker Dashboard [2]

Website Working and Firestore Images

Figure 16 displays the registration page displayed using the first Docker container (i.e., container_1_app).

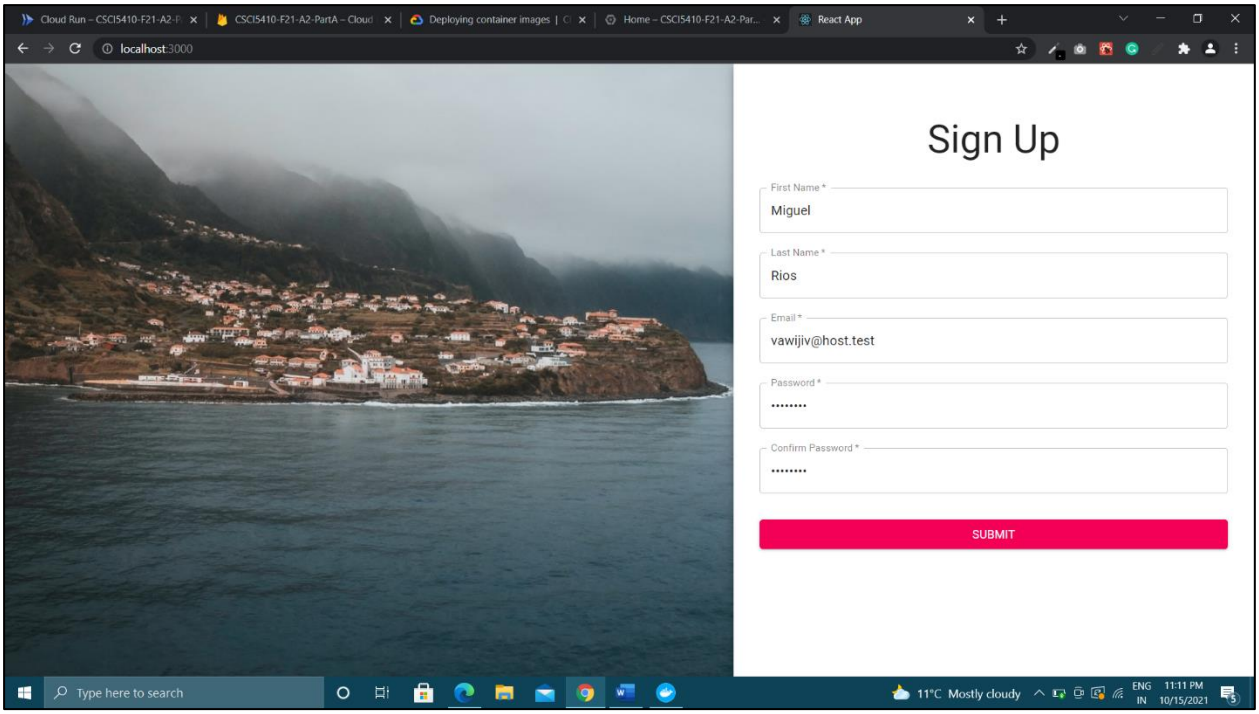

Figure 16 - Registration application displayed using container_1_app

Figure 17 displays that on successful registration, data is stored in the Firebase Firestore database with status value **offline**.

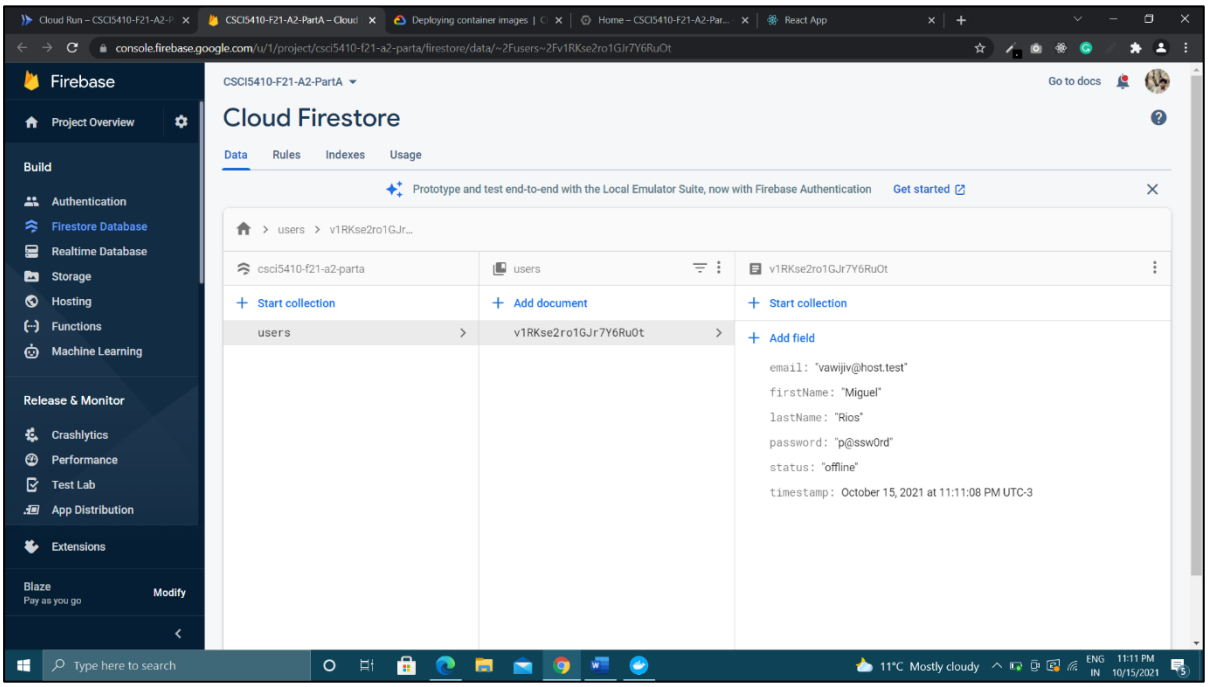

Figure 17 - Successful user registration with status value offline [1]

Figure 18 displays the login page displayed using the second Docker container (i.e., container_2_app).

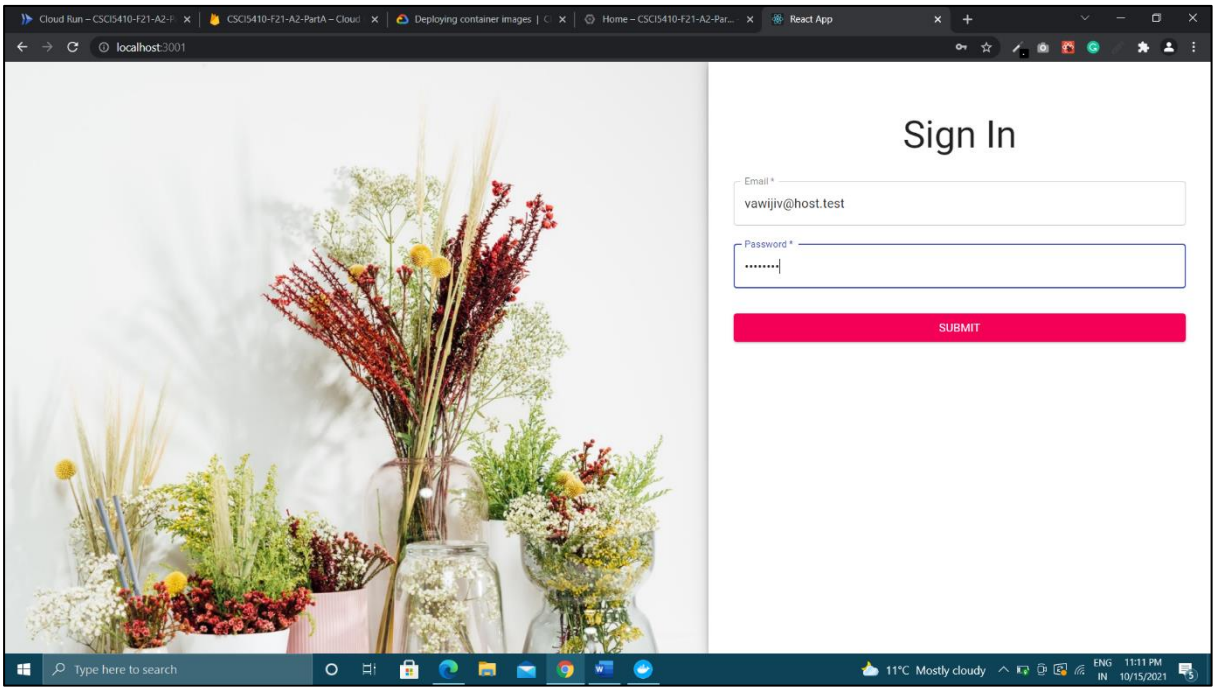

Figure 18 – Login application displayed using conatiner_2_app

Figure 19 displays the profile application displayed using the third Docker container (i.e., container_3_app).

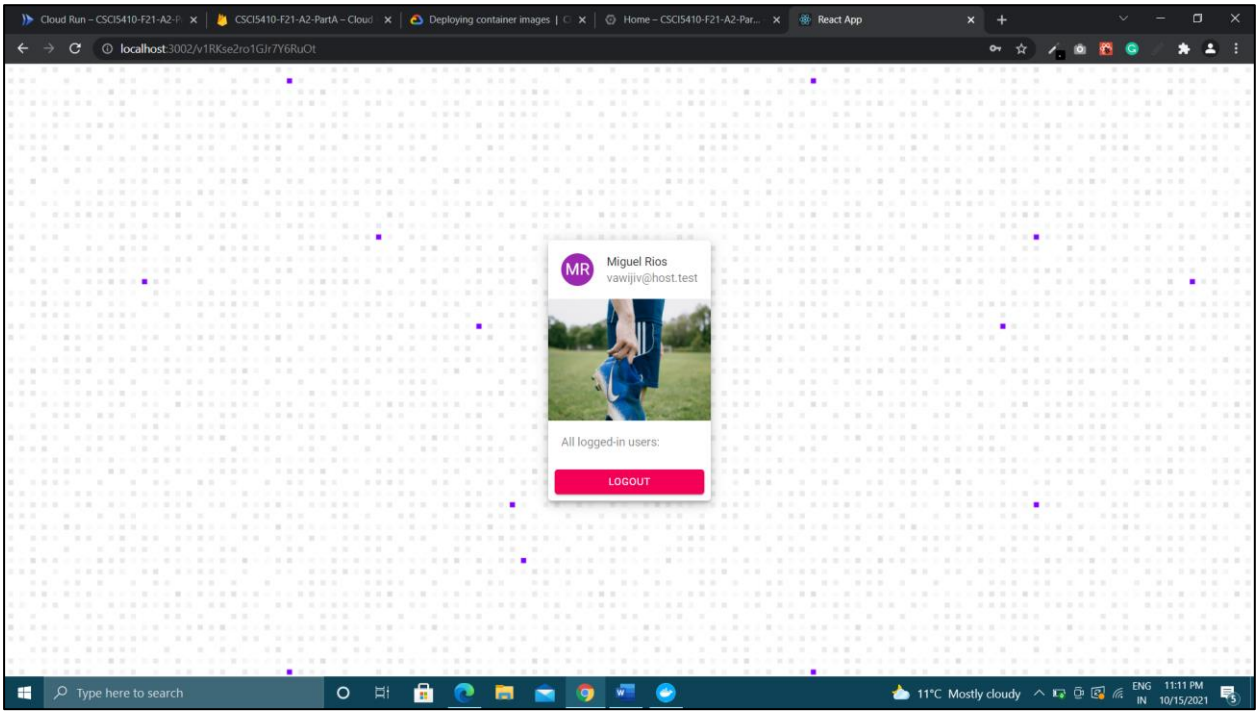

Figure 19 - Profile application displayed using container_3_app

Figure 20 displays the successful login of the user with the status value online.

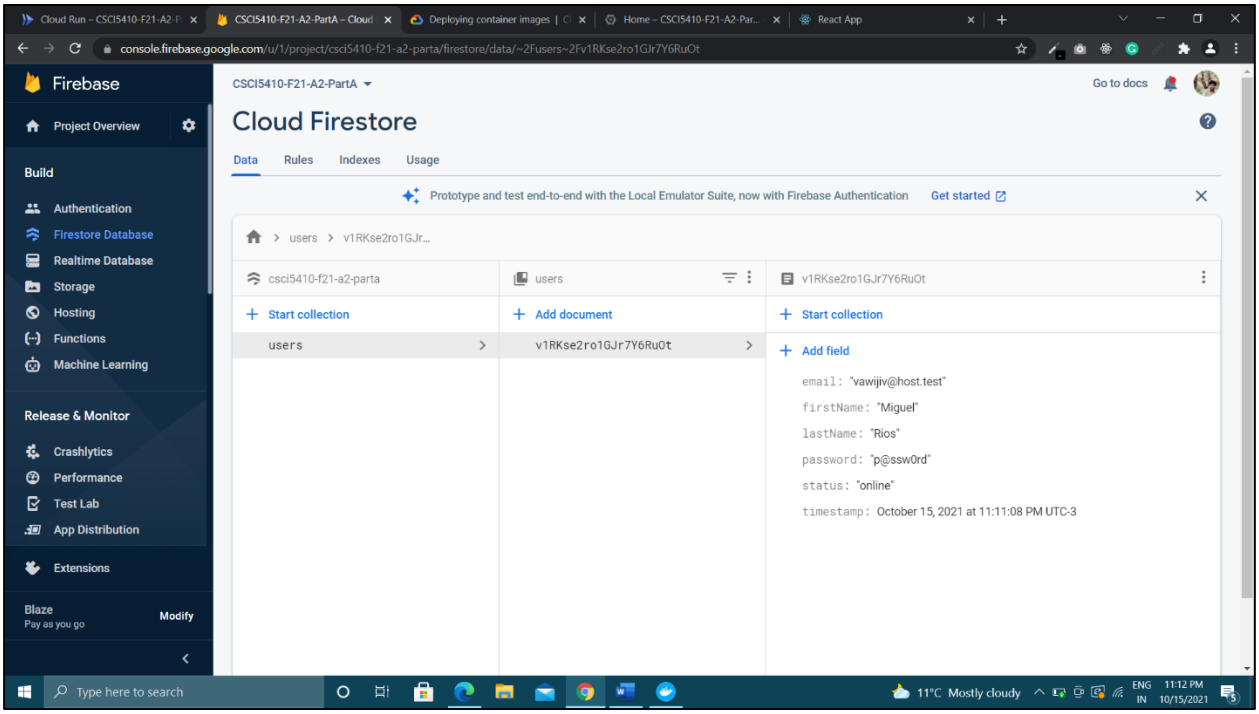

Figure 20 - Successful login changes the value of status to online [1]

Figure 21 displays the successful user logout data in Firestore database with the status changed to offline.

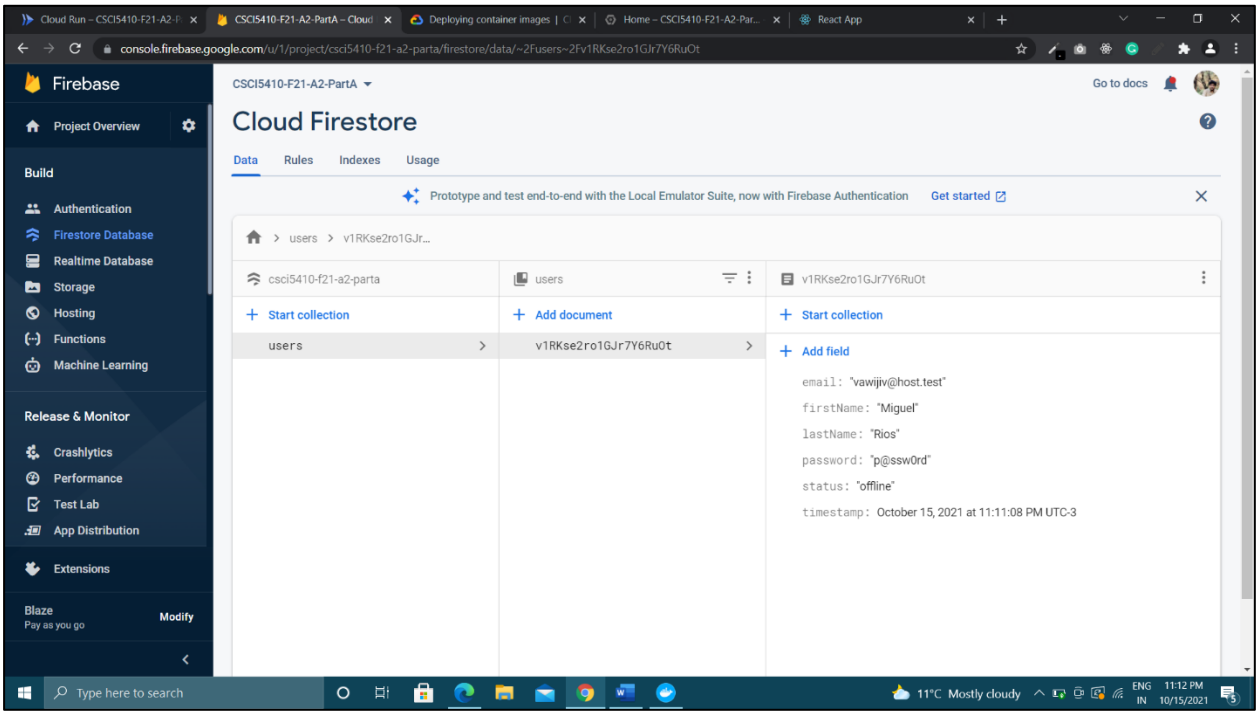

Figure 21 - Successful logout changes the status value to offline [1]

Figure 22 displays the information of the other logged-in users (i.e., users with status – online).

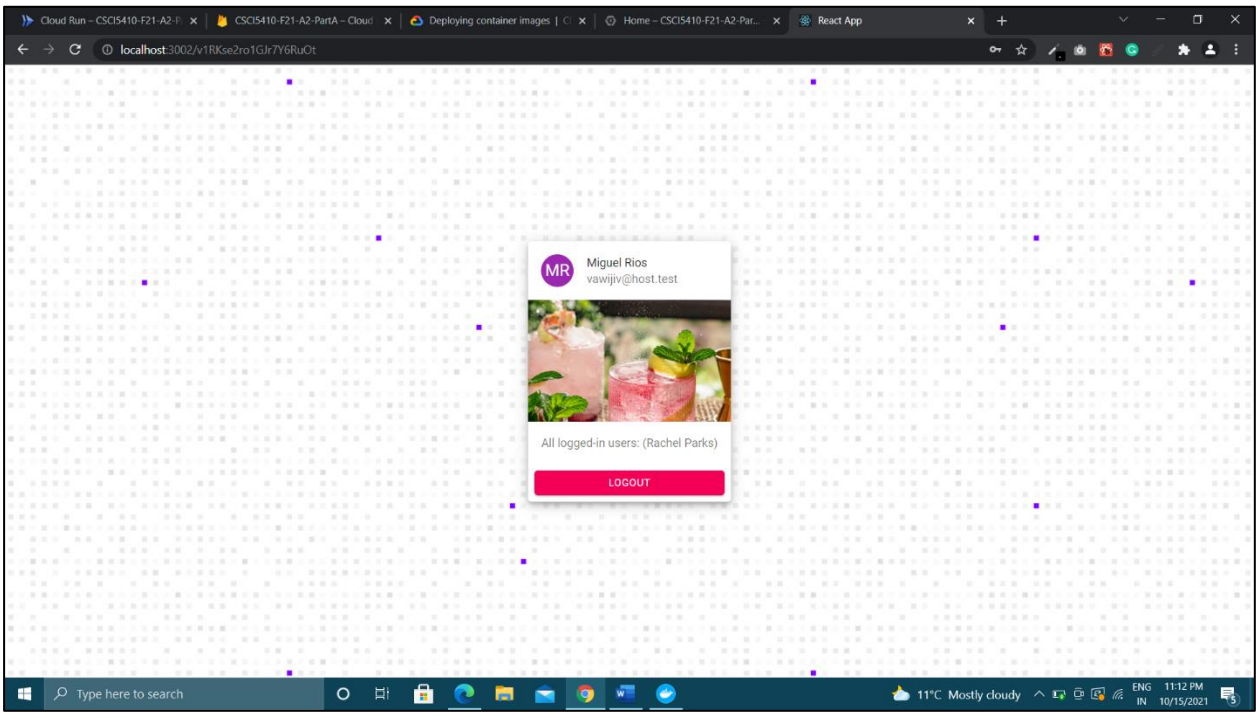

Figure 22 - Successful display of all the other users who are online

Figure 23 displays the Firestore data of other logged-in users (i.e., status – online).

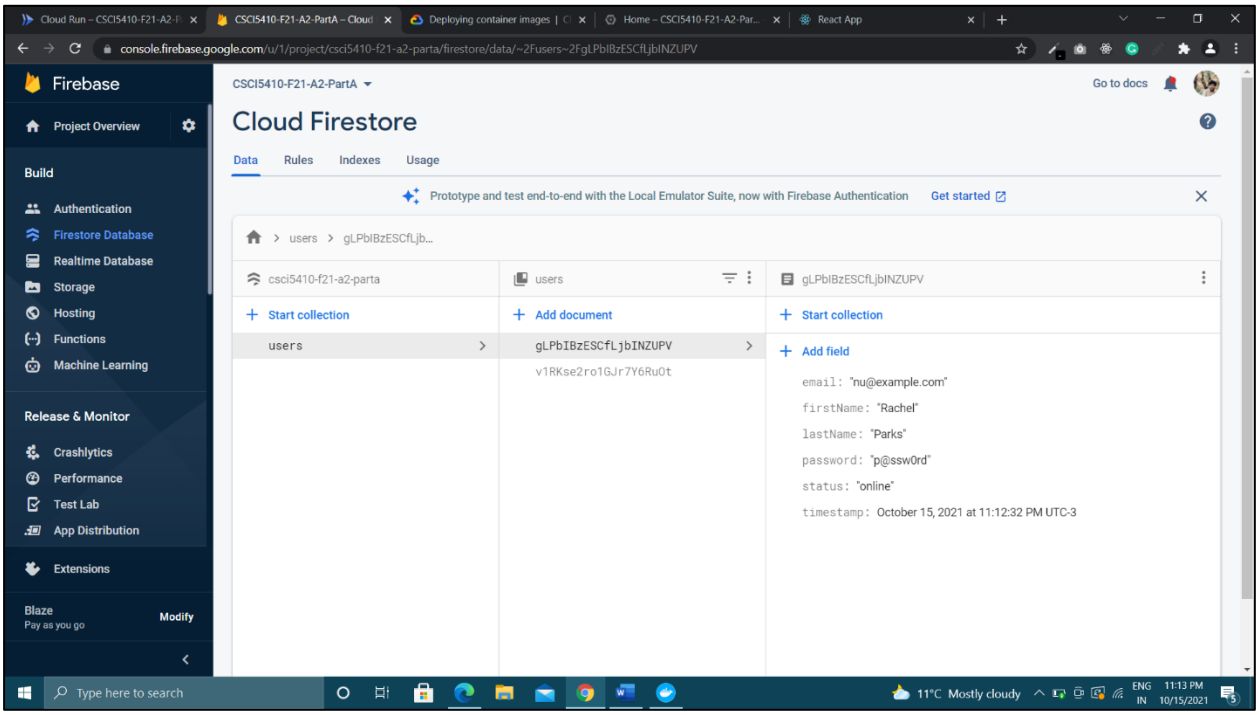

Figure 23 - Users with status online are displayed in the Profile application [1]

Docker and Google Container Registry

Steps

Step 1 – Downloaded Google Cloud SDK [3] from Google's official website and installed it on my machine.

Step 2 – Logged-in to Google Cloud SDK using my Google account.

Step 3 – Selected the project CSCI5410-F21-A2-PartA to work and copied the project id csci5410-f21-a2-parta for tagging the docker images.

Step 4 – View all docker images using command **"docker images"** (In my case I wanted to upload 3 Docker images container_1_app, container_2_app, and container_3_app)

Step 5 – Tag all the three images using commands
- docker tag container_1_app gcr.io/csci5410-f21-a2-parta/container_1_app
- docker tag container_2_app gcr.io/csci5410-f21-a2-parta/container_2_app
- docker tag container_3_app gcr.io/csci5410-f21-a2-parta/container_3_app

Step 6 – Configured Cloud Shell to authenticate my machine using command
- net localgroup docker-users DESKTOP-4P1CDV0\admin /add

Step 7 – Configure Docker with the following command:
- gcloud auth configure-docker

Step 8 – Push images to Google Container Registry
- gcloud docker push gcr.io/csci5410-f21-a2-parta/container_1_app
- gcloud docker push gcr.io/csci5410-f21-a2-parta/container_2_app
- gcloud docker push gcr.io/csci5410-f21-a2-parta/container_3_app

Figure 24 shows tagging of three docker images namely container_1_app, container_2_app, and container_3_app.

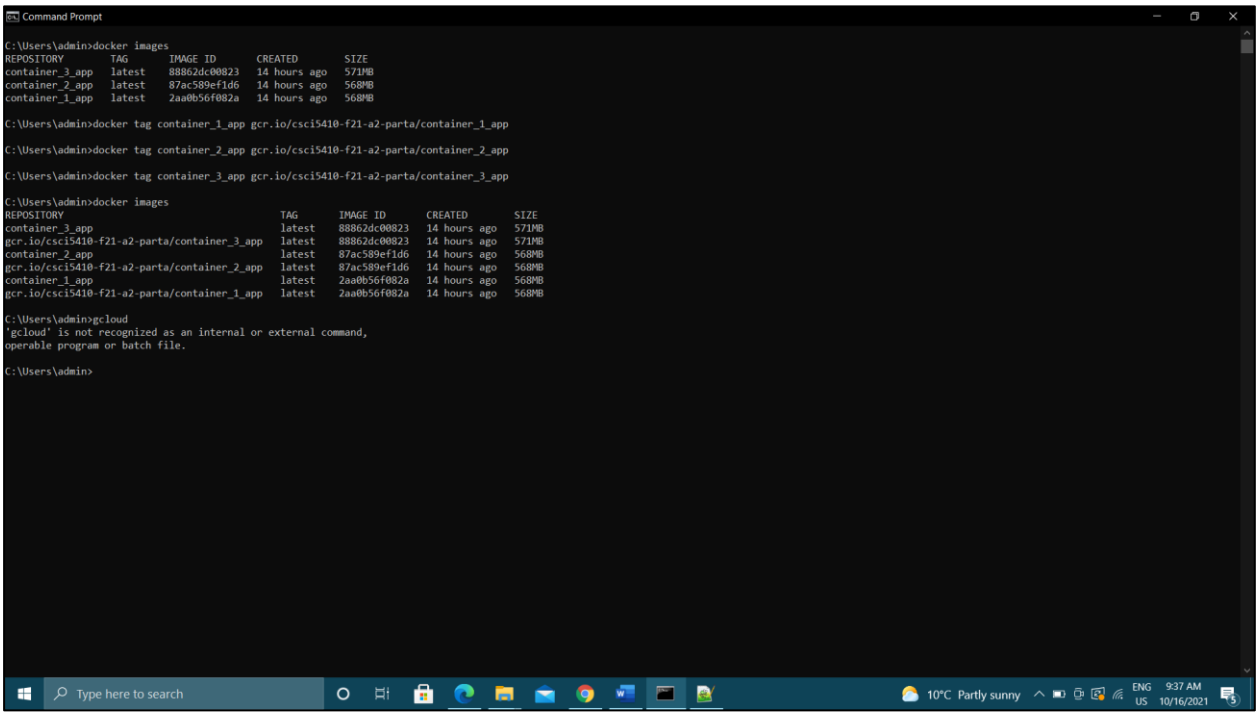

Figure 24 – Tagging of three docker images namely container_1_app, container_2_app, and container_3_app

Figure 25 shows pushing three tagged docker images to Google Container Registry. These images are inside the project – csci5410-f21-a2-parta.

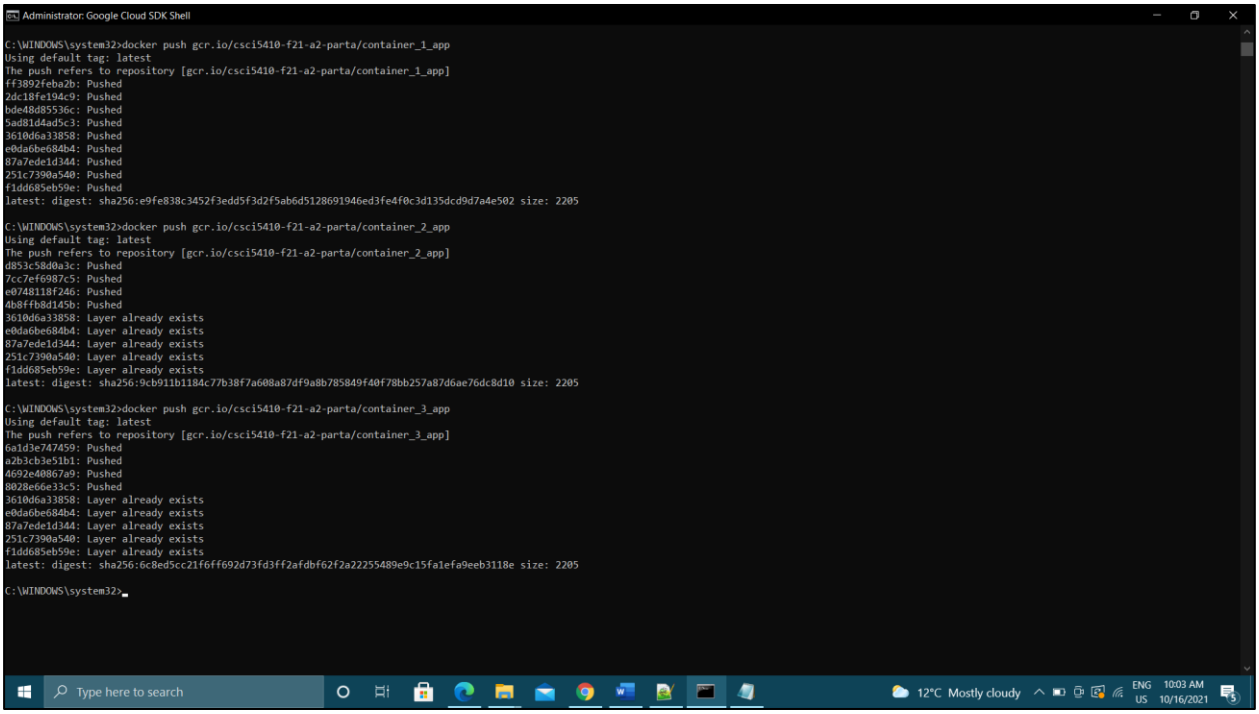

Figure 25 - Docker push to Google Container Registry

Figure 26 displays three Docker images uploaded to Google Container Registry service.

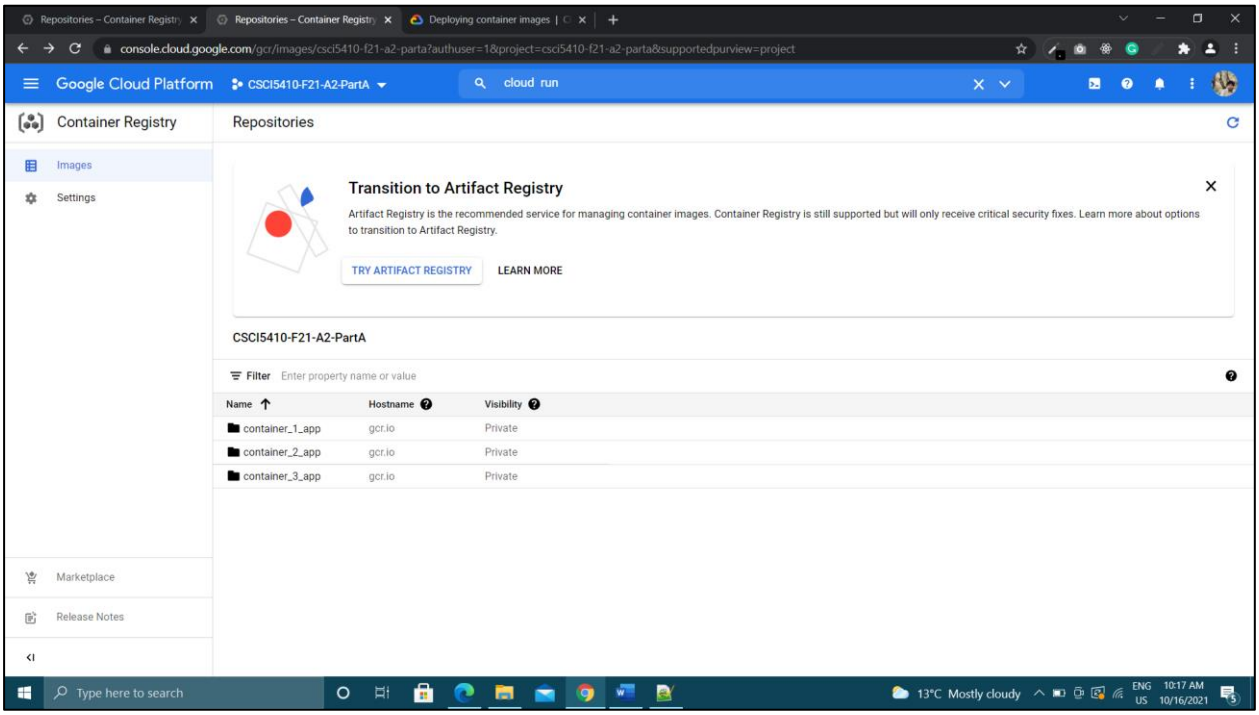

Figure 26 - Docker images uploaded to Google Container Registry [4]

Google Cloud Run and Website hosted on GCR

Figure 27, 28, 29, 30, 31, 32, 33, 34, and 35 displays the configuration of Docker images namely container_1_app, container_2_app, and container_3_app on Google Cloud Run.

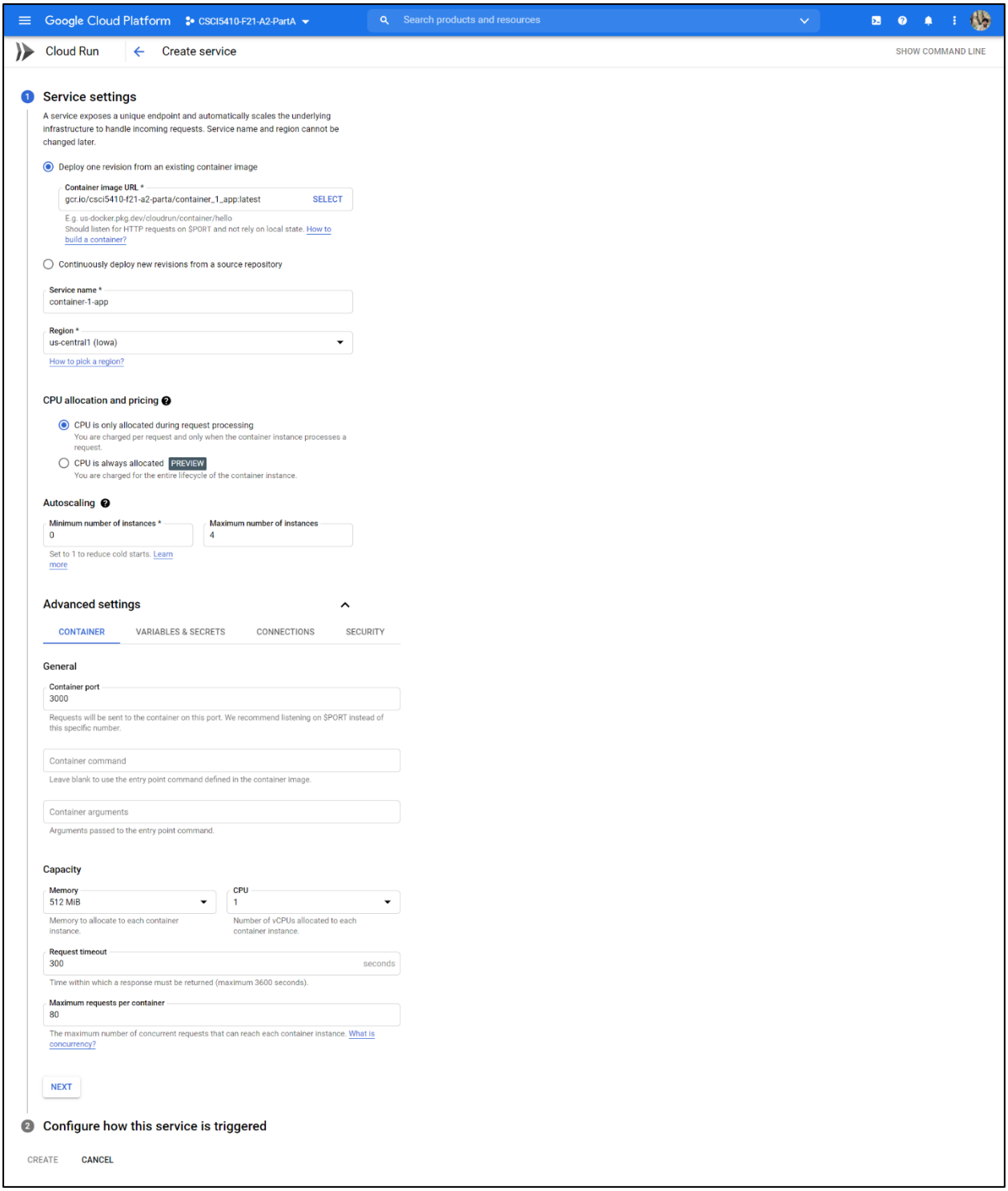

Figure 27 - Configuration of container_1_app Docker image on Google Cloud Run [5]

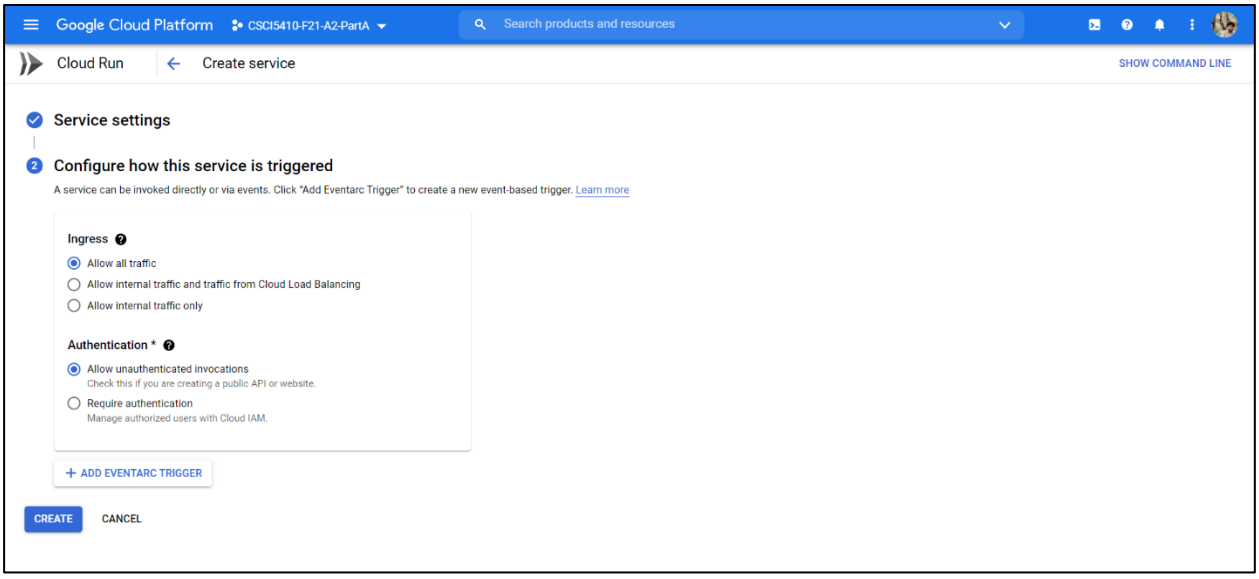

Figure 28 - Configuration of container_1_app Docker image on Google Cloud Run (contd.) [5]

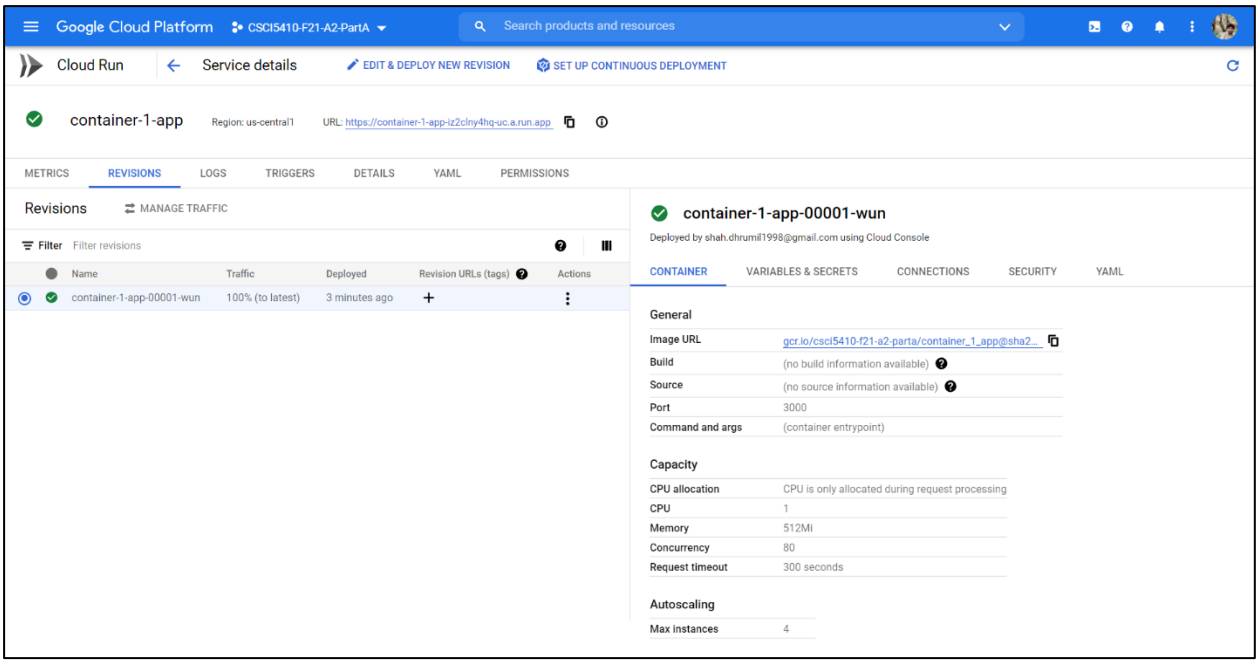

Figure 29 - container_1_app (React Registration application) hosted on Google Cloud Run [5]

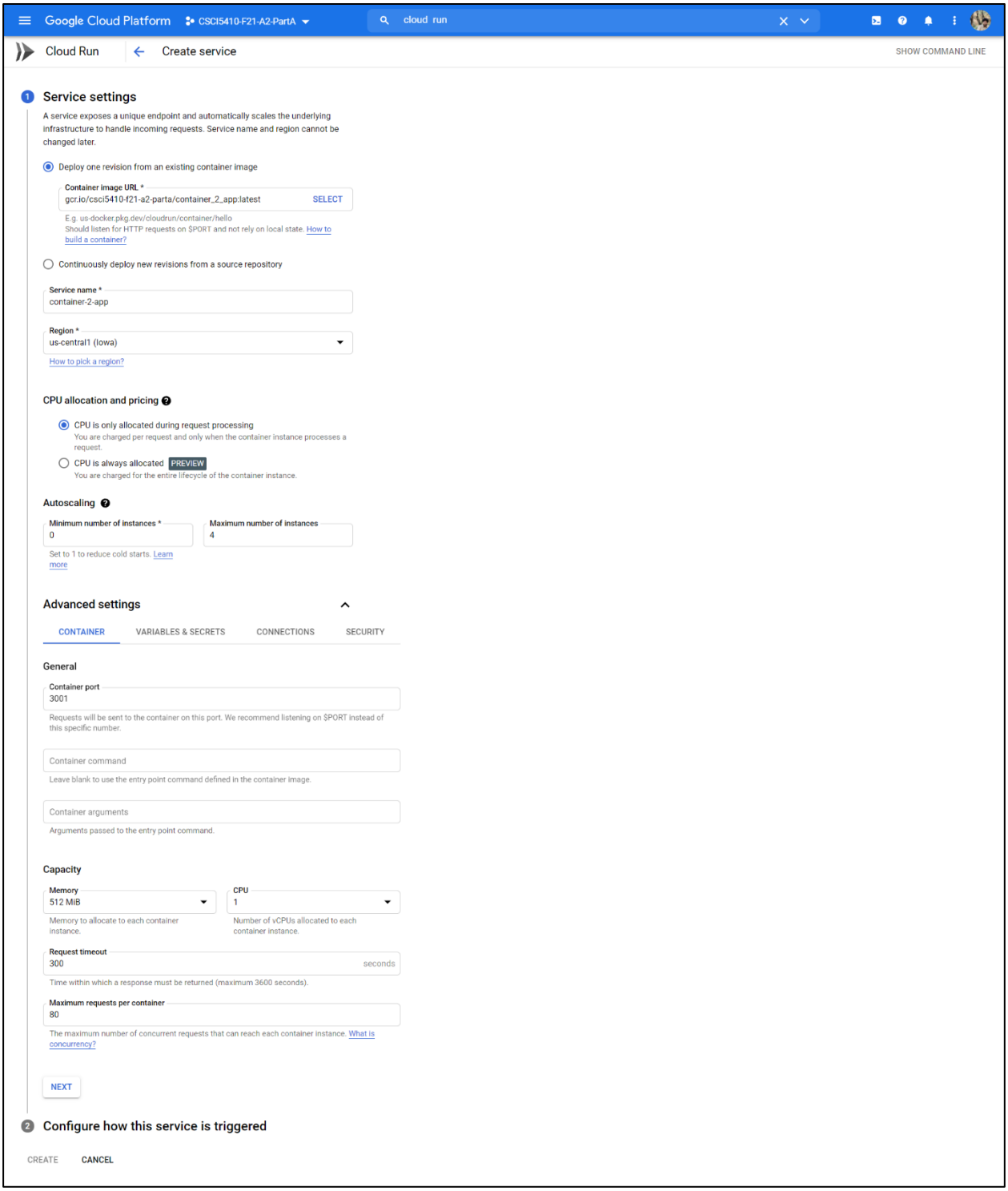

Figure 30 - Configuration of container_2_app Docker image on Google Cloud Run [5]

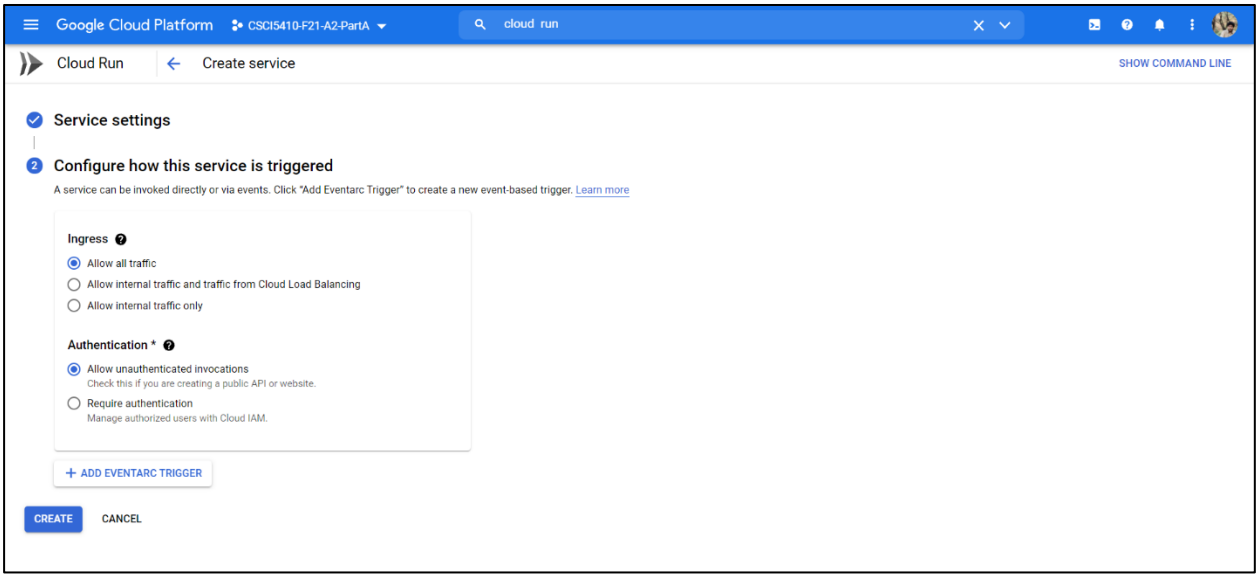

Figure 31 - Configuration of container_2_app Docker image on Google Cloud Run (contd.) [5]

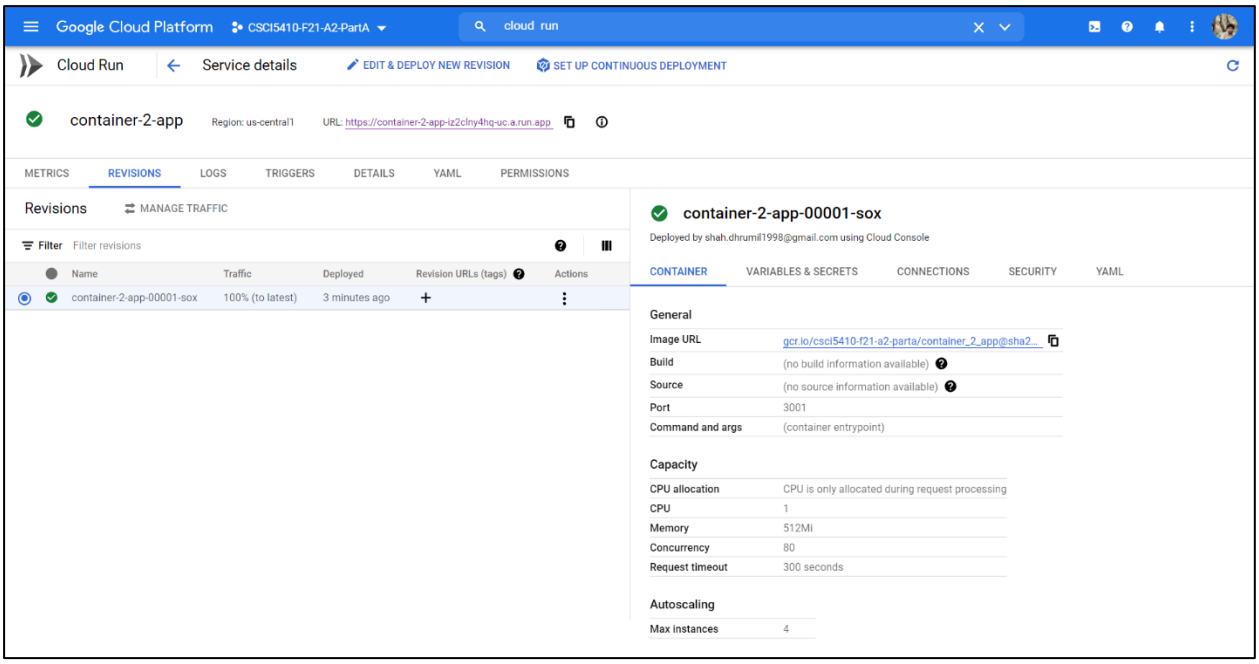

Figure 32 - container_2_app (Login React application) hosted on Google Cloud Run [5]

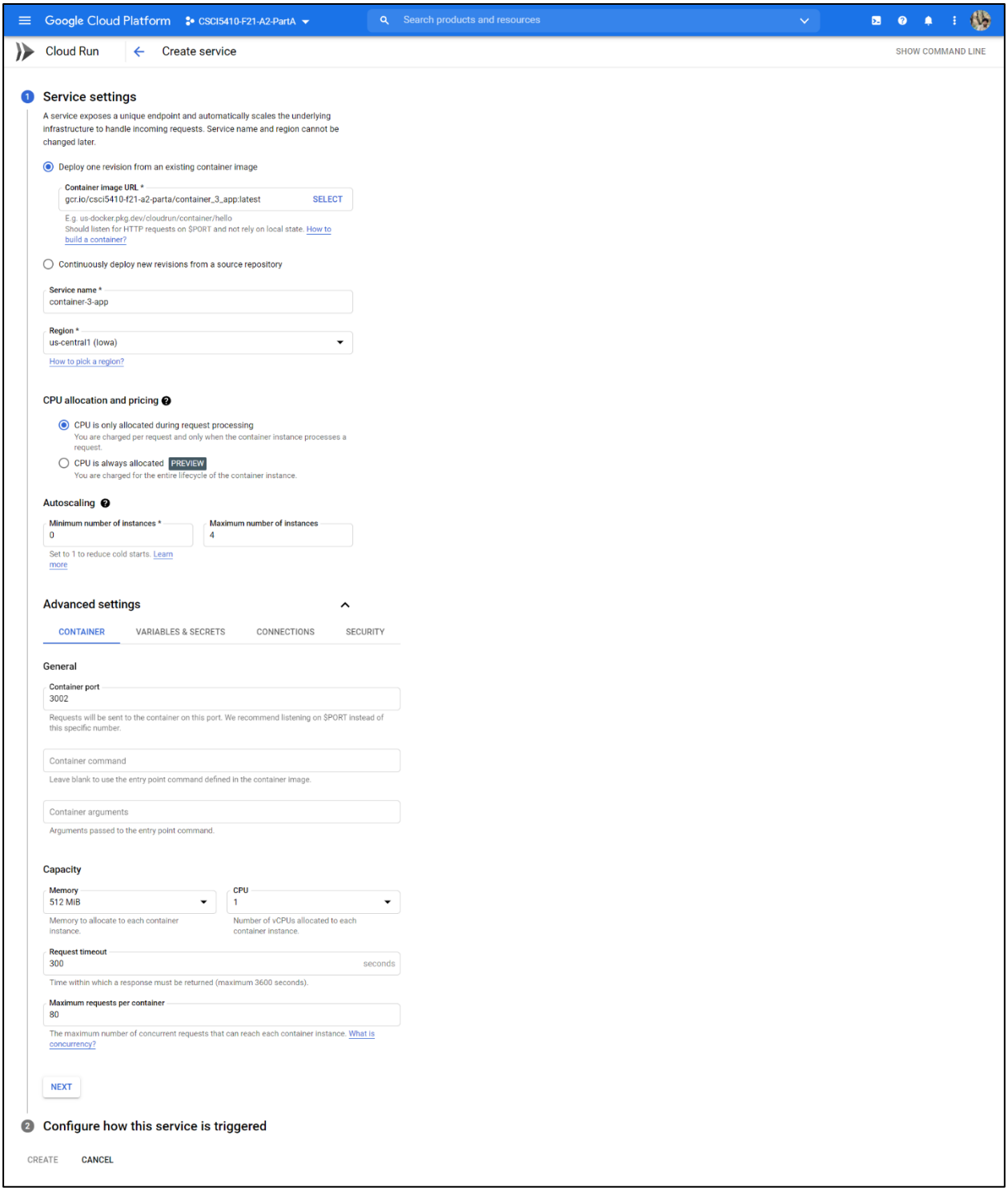

Figure 33 - Configuration of container_2_app Docker image on Google Cloud Run [5]

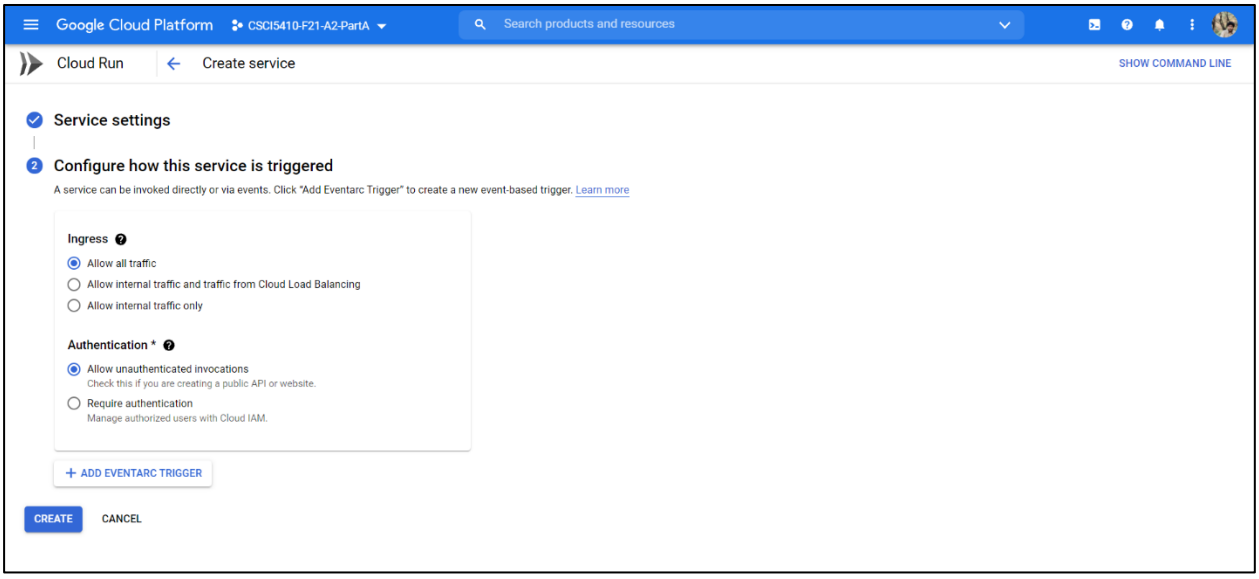

Figure 34 - Configuration of container_2_app Docker image on Google Cloud Run (contd.) [5]

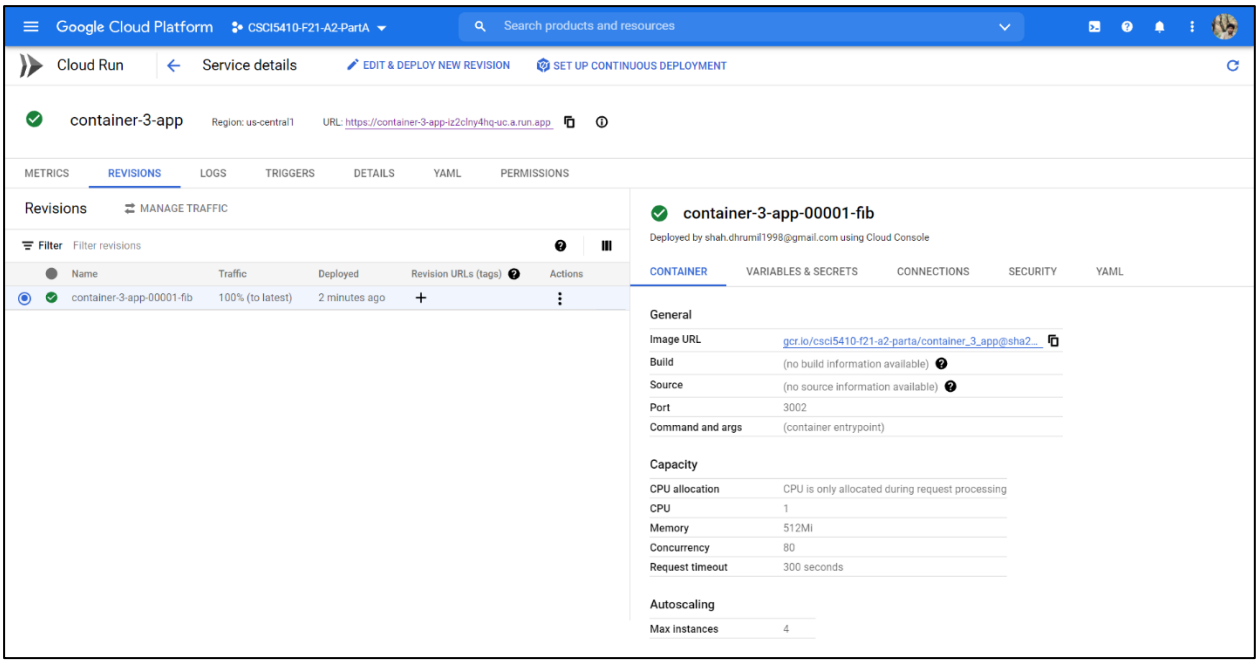

Figure 35 - container_3_app (Profile React application) hosted on Google Cloud Run [5]

Figure 36, 37 and 38 displays the website hosted on Google Cloud Run.

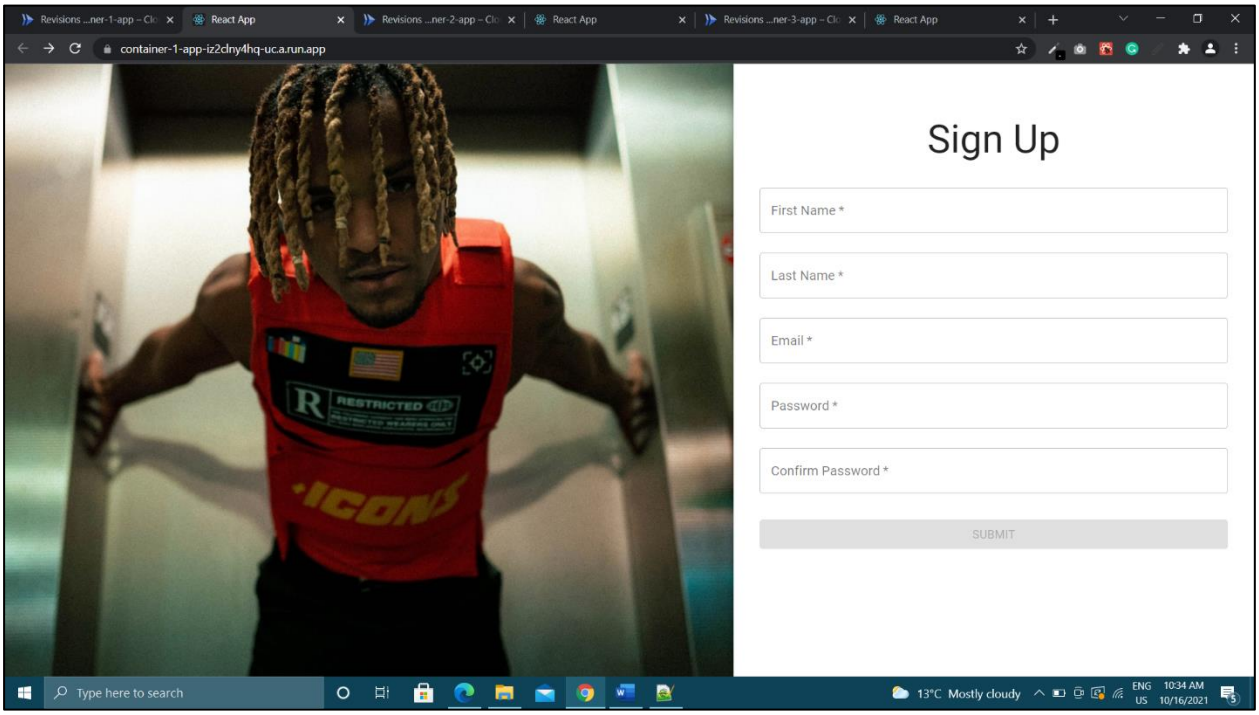

Figure 36 - container_1_app (Registration React application) website hosted on Google Cloud Run

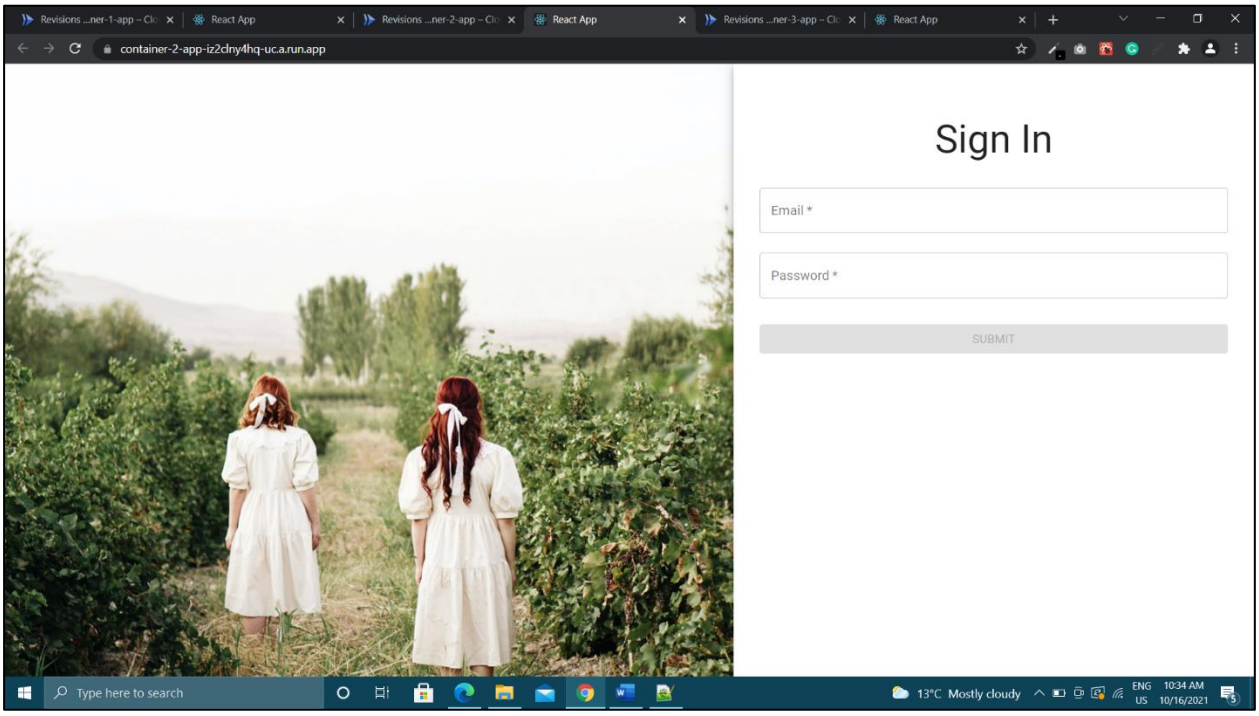

Figure 37 - container_2_app (Login React application) website hosted on Google Cloud Run

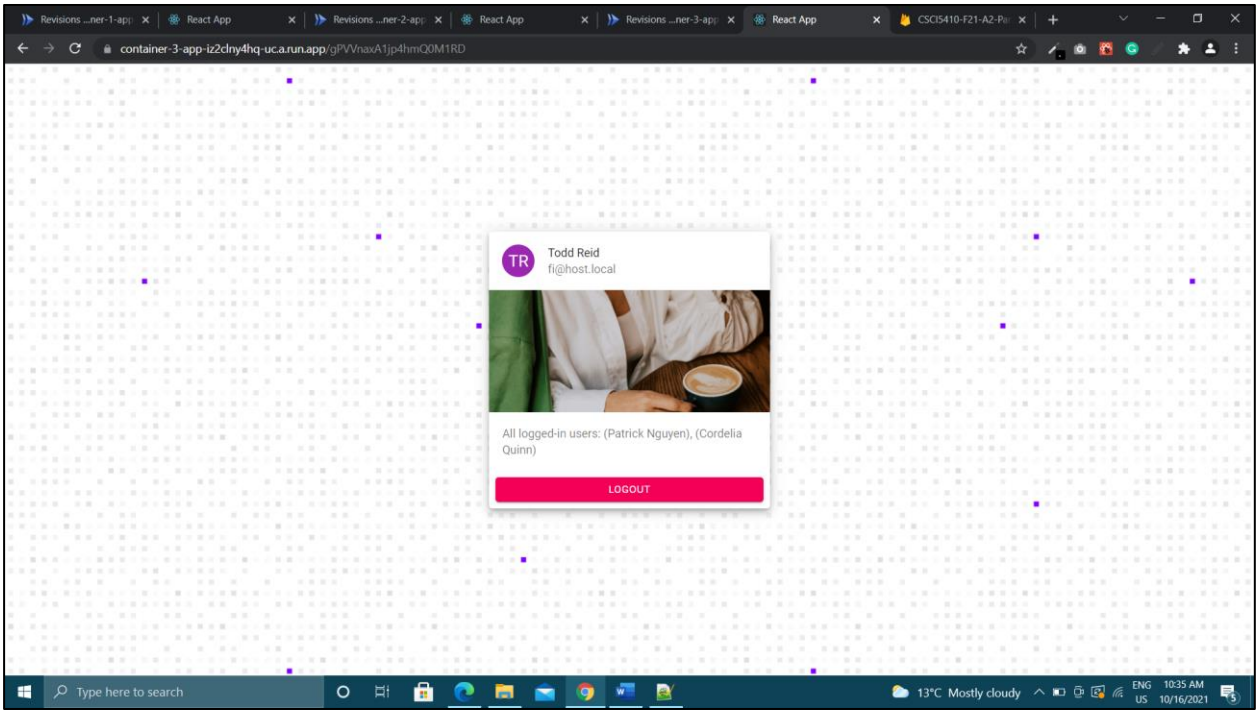

Figure 38 - container_3_app (Profile React application) hosted on Google Cloud Run

Test Cases and Important Code

⇨ Method to validate first name, last name, email, password and confirm password. (Registration React application)

```
const validate = (e) => {
      switch (e.target.name) {
         case 'firstName':
            const isFNameCorrect = RegExp(/^[A-Za-z\d]+$/).test(e.target.value);
            if (e.target.value === "" || e.target.value === null) {
               errors["firstName"] = "First name is required."
               errors["firstNameValid"] = false;
            } else if (!isFNameCorrect) {
               errors["firstName"] = "First name can only have alpha-numeric characters."
               errors["firstNameValid"] = false;
            } else {
               errors["firstName"] = "";
               errors["firstNameValid"] = true;
            }
            break;
         case 'lastName':
            const isLNameCorrect = RegExp(/^[A-Za-z\d]+$/).test(e.target.value);
            if (e.target.value === "" || e.target.value === null) {
               errors["lastName"] = "Last name is required."
               errors["lastNameValid"] = false;
            } else if (!isLNameCorrect) {
               errors["lastName"] = "Last name can only have alpha-numeric characters."
               errors["lastNameValid"] = false;
            } else {
               errors["lastName"] = "";
               errors["lastNameValid"] = true;
            }
            break;
         case 'email':
            // Email regex source - https://stackoverflow.com/questions/46155/how-to-validate-an-email-address-in-javascript
                                                const isEmailCorrect = RegExp(/^(([^<>()[\]\\.,;:\s@"]+(\.[^<>()[\]\\.,;:\s@"]+)*)|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/).test(e.target.value);
            if (e.target.value === "" || e.target.value === null) {
               errors["email"] = "Email name is required."
               errors["emailValid"] = false;
            } else if (!isEmailCorrect) {
               errors["email"] = "Please enter a valid email address."
               errors["emailValid"] = false;
```

```
            } else {
                errors["email"] = ""
                errors["emailValid"] = true;
            }
            break;
        case 'password':
            const isPasswordCorrect = RegExp(/^[A-Za-z\d@$!%*#?&_]+$/).test(e.target.value);
            if (e.target.value === "" || e.target.value === null) {
                errors["password"] = "Password is required."
                errors["passwordValid"] = false;
            } else if (e.target.value.length < 8) {
                errors["password"] = "Password must be at least 8 characters long."
                errors["passwordValid"] = false;
            } else if (!isPasswordCorrect) {
                errors["password"] = "Password can only have alpha-numeric and special characters."
                errors["passwordValid"] = false;
            } else {
                errors["password"] = ""
                errors["passwordValid"] = true;
            }
            break;
        case 'confirmPassword':
            if (e.target.value === "" || e.target.value === null) {
                errors["confirmPassword"] = "Confirm password is required."
                errors["confirmPasswordValid"] = false;
            } else if (e.target.value !== userData.password) {
                errors["confirmPassword"] = "Confirm password must match the password field."
                errors["confirmPasswordValid"] = false;
            } else {
                errors["confirmPassword"] = ""
                errors["confirmPasswordValid"] = true;
            }
            break;
        default:
            break;
    }
    setErrors(errors);
};
```

⇨ Method to store the data in Firebase Firestore (Registration React application) – User registration process.

```
const storeDataInFirestore = async () => {
    try {
        const docRef = await addDoc(collection(firestoreDB, "users"), {
            firstName: userData.firstName,
            lastName: userData.lastName,
            email: userData.email,
            password: userData.password,
            status: 'offline',
            timestamp: serverTimestamp(),
        });
        console.log("Document written with ID: ", docRef.id);
        updateUserData({
            firstName: '',
            lastName: '',
            email: '',
            password: '',
            confirmPassword: '',
        });
        window.location.href = 'http://localhost:3001/';
    } catch (e) {
        console.error("Error adding user: ", e);
    }
}
```

⇨ Method to perform user login (Login React application) – User login

```
const loginUser = async () => {
    try {
            const queryObj = query(collection(firestoreDB, "users"), where("email", "==",
userData.email));
        const querySnapshot = await getDocs(queryObj);
        var userExists = false;
        querySnapshot.forEach(async (userDoc) => {
            if (userData.password === userDoc.data().password) {
                userExists = true;
                console.log("User found!");
                console.log(userDoc.id, " => ", userDoc.data());
                const userDocRef = doc(firestoreDB, "users", userDoc.id);
                await updateDoc(userDocRef, {
                    status: "online",
                });
```

```
                window.location.href = 'http://localhost:3002/' + userDoc.id;
            } else {
                userExists = false;
            }
        });
        if (!userExists) {
            console.error("User not found!");
        }
    } catch (e) {
        console.error("Error reading user: ", e);
    }
};
```

⇨ Method to fetch all the users who are online. (Profile React application)

```
useEffect(async () => {
    const onlineUsersQueryObj = query(collection(firestoreDB, "users"), where("status", "==", "online"));
    const onlineUserQuerySnapshot = await getDocs(onlineUsersQueryObj);
    var allOnlineUsers = "";
    var firstTime = true;
    onlineUserQuerySnapshot.forEach(async (userDoc) => {
        if (userDoc.data().status === "online") {
            if (userDoc.id !== userDocId) {
                if (firstTime) {
                    allOnlineUsers = "(" + userDoc.data().firstName + " " + userDoc.data().lastName + ")";
                    firstTime = false;
                } else {
                    allOnlineUsers = allOnlineUsers + ", " + "(" + userDoc.data().firstName + " " + userDoc.data().lastName + ")";
                }
            } else {
                setLoggedInUser({
                    firstName: userDoc.data().firstName,
                    lastName: userDoc.data().lastName,
                    email: userDoc.data().email,
                });
            }
        }
    });
    setOnlineUsers(allOnlineUsers);
});
```

⇨ Method to perform logout (Profile React application)

```
const logout = async () => {
    const userDocRef = doc(firestoreDB, "users", userDocId);
    await updateDoc(userDocRef, {
        status: "offline",
    });
    window.location.href = 'http://localhost:3001/';
}
```

Summary

Serverless Data Processing allows containerizing applications and running them remotely using the cloud services provided by providers like Amazon, Google, Microsoft Azure, and so on. The modules visited for this exercise are Docker containers, Google Container Registry, and Google Cloud Run.

Docker Containers are services in the form of platform to create, run, and deploy applications in various containers, each of which works independently irrespective of the other containers. These containers contain the web applications required to be run on the cloud platform. A docker container is a light weight package of software that works in isolation. Running these docker containers on cloud platforms, helps the applications to be accessed from anywhere.

Google Container Registry is a service by Google used to store private images. The service uses Artifact Registry functions like creating multiple registries in same or multiple regions, and so on. The registry helps store this images from Cloud Build, deploy them to Google Cloud Run directly using services such as Google Cloud Run, Compute Engine, Kubernetes Engine. Identity and Access Management control provides consistent credentials and access control. The containers created here are securely built from the images. The uploaded images in the Container Registry are then deployed on one of the engines provided by Google to run these containers remotely from anywhere.

Google Cloud Run is a computing platform managed by Google that eases the job of developers for deploying and running containers that may be invoked by multiple requests or events. It is serverless in nature, which means it hides away the infrastructure so that developers can focus on the development and not on the infrastructure and management. The process to use Google Cloud Run is straightforward as GCR automates most of the process. Job of developer is to simply tell where the container is, what memory, CPU are required, and click “Create”. The application can be accessed using the http endpoint.

References

[1] Google, "Cloud Firestore," Google, [Online]. Available: https://firebase.google.com/products/firestore. [Accessed 15 October 2021].

[2] Docker, "Developers Love Docker," Docker, [Online]. Available: https://www.docker.com/. [Accessed 14 October 2021].

[3] Google, "Installing Cloud SDK," Google, [Online]. Available: https://cloud.google.com/sdk/docs/install. [Accessed 15 October 2021].

[4] Google, "Container Registry," Google, [Online]. Available: https://cloud.google.com/container-registry. [Accessed 15 October 2021].

[5] Google, "Cloud Run," Google, [Online]. Available: https://cloud.google.com/run. [Accessed 15 October 2021].