ASSIGNMENT 2: PART A

TITLE: Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security

ACM Reference:

[1] S. Garg and S. Garg, "Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security," 2019 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR), 2019, pp. 467-470, doi: 10.1109/MIPR.2019.00094.

Summary:

The paper discusses about cloud computing, which boosts productivity and reduces costs [1]. The studies also highlight efficient software deployment and maintenance, which reduces administrative work [1]. One such technique is the IaaS transition, which is required to manage automated resources effectively [1]. Many organisations have their own architectural designs that make virtualization simple [1]. Using containerization, this is achievable [1]. Virtualization is carried out using open-source IT solutions like Terraform, which manages numerous cloud infrastructures and significantly decreases the amount of manual work [1]. Continuous development, which is used to create an environment that is highly competitive, heavily relies on automation [1]. This approach is sometimes referred to as a CI/CD pipeline [1].

Developers can evaluate the code quality and whether it passes testing with the use of continuous integration (CI) [1]. Through automated testing, this procedure aids in problem resolution more quickly [1]. Continuous Delivery (CD), on the other hand, concentrates on frequently deploying smaller updates [1]. The benefit of continuous delivery is that it lowers the risk associated with deployment because smaller changes are tracked and processed through automated changes prior to production [1]. This method aids the development team in implementing features more quickly, allowing them to deliver minor updates regularly and advance at a faster rate [1]. The only disadvantage of continuous delivery is that it requires a lot of time and has a challenging, dynamic culture [1]. Docker may be used to implement CI/CD and will validate the necessary data in the container to build the application [1]. To keep everything updated, Docker caches the outcomes in the Dockerfile and invalidates the most recent change [1]. The Jenkins platform running within a Docker container and the Jenkins master running inside the Docker engine are examples of architectural components of Docker [1]. The Dockerfile is used to construct the application, and the GitHub repository serves as its host [1]. Finally, the Jenkins master will use a GitHub plugin to trigger a Jenkins job whenever a developer pushes a modification to the repository [1].

Continuous integration and continuous delivery give rise to containers that are more reliable and efficient than traditional virtualization [1]. Docker containers enable local software development regardless of the host system [1]. They offer agility in addition to portability [1]. Docker provides both system management and environmental configuration [1]. A client-server architecture is implemented by Docker, and it also has a conceptual Docker architecture with a Docker client and a Docker daemon, in which a user interacts with the Docker client rather than a daemon directly [1]. The namespaces, control groups, docker daemon, Linux kernel capabilities, and security are the best practises for container security [1]. If the contents are processed as non-root, containers are often secure [1]. Third-party containers, however, are unsafe [1]. Use a private registry like Docker Trusted Registry in place of third-party containers. The industry suggests a variety of solutions for Docker security, including Docker Bench and CoreOS's Clair [1]. Future industrial automation will be changed by cloud computing, and this will benefit the open source community by enabling developers to start building applications even more quickly [1].

ASSIGNMENT 2

Build, deploy and run a Containerized Application using GCP

Git Link: https://git.cs.dal.ca/umatiya/csci5410-f23-b00899642/-/tree/A2

The primary objective of this Assignment is to work on the cloud computing containerization application using docker.

Below are the steps that I followed to work on the Assignment:

Using GCP, I created and validated an Online meeting account. Below are the steps that I followed:

Step A):

TO DO: Create 3 microservices and package these in three different containers, which will be responsible for the front-end and backend logic in this application, The database you will be using here is, Firestore.

PROCEDURE:

1. **I first created 3 containers for 3 microservices (Refer Fig. 1):**
   - container-1_Register_App (For Registration)
   - container-2_Login_App (For Login)
   - container-3_State-App (For knowing the user's state i.e. whether user is online or offline)

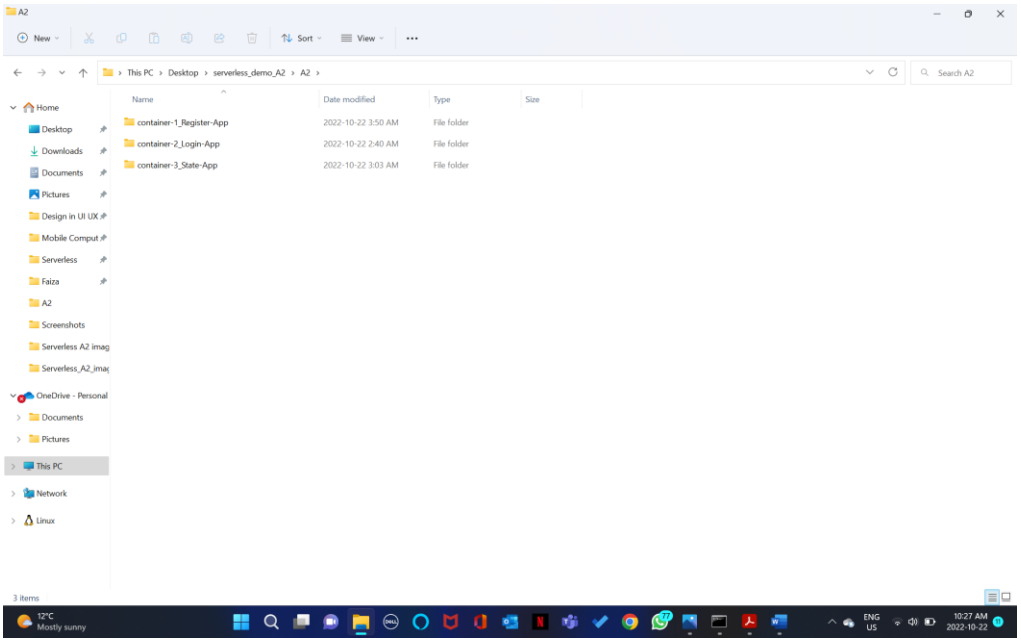


Figure 1: Created 3 container folders for 3 microservice

2. **Setting up the environment [1]:**
    - I preferably used VS Code as my IDE.
    - I installed Node.js and npm (Refer Fig. 2) using the command “npm install server -g”.
    - After that I created first react-app i.e. registration app using the command “npm create-react-app register-app” (Refer Fig. 3).
    - After writing the command “npm start”, our react-app launches successfully as shown in the Fig. 4.

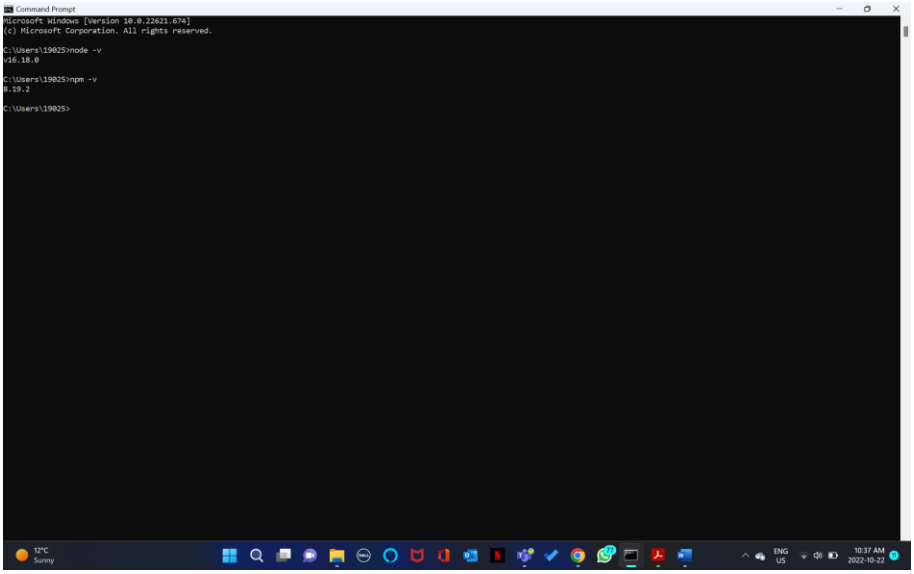


Figure 2: Installed Node.js and npm

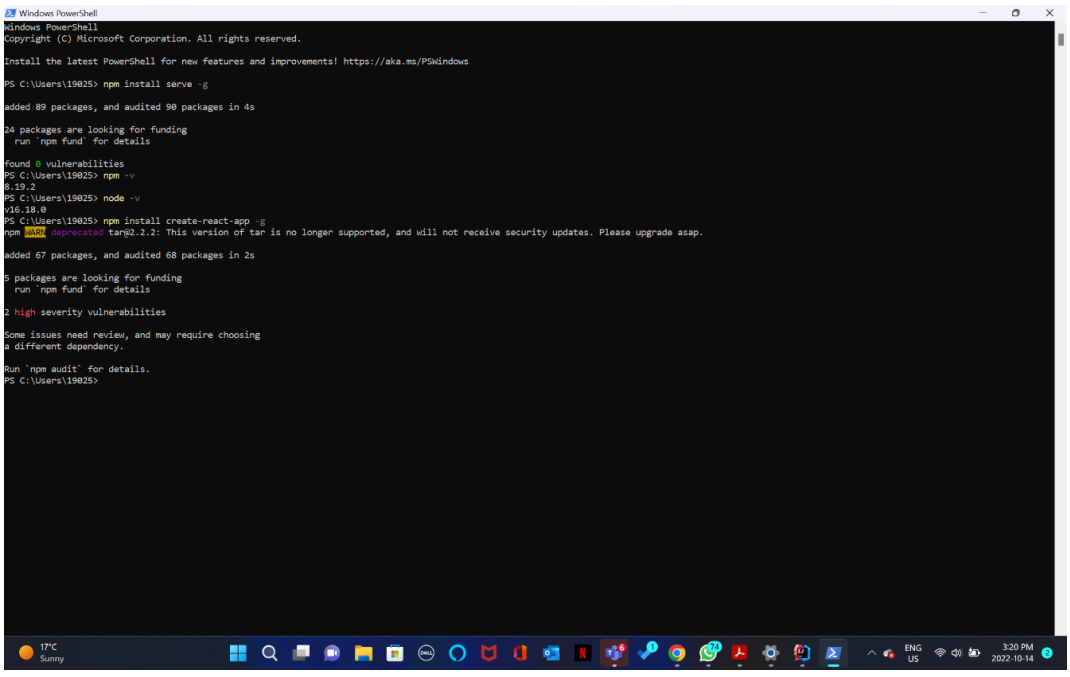


Figure 3: container created using node command

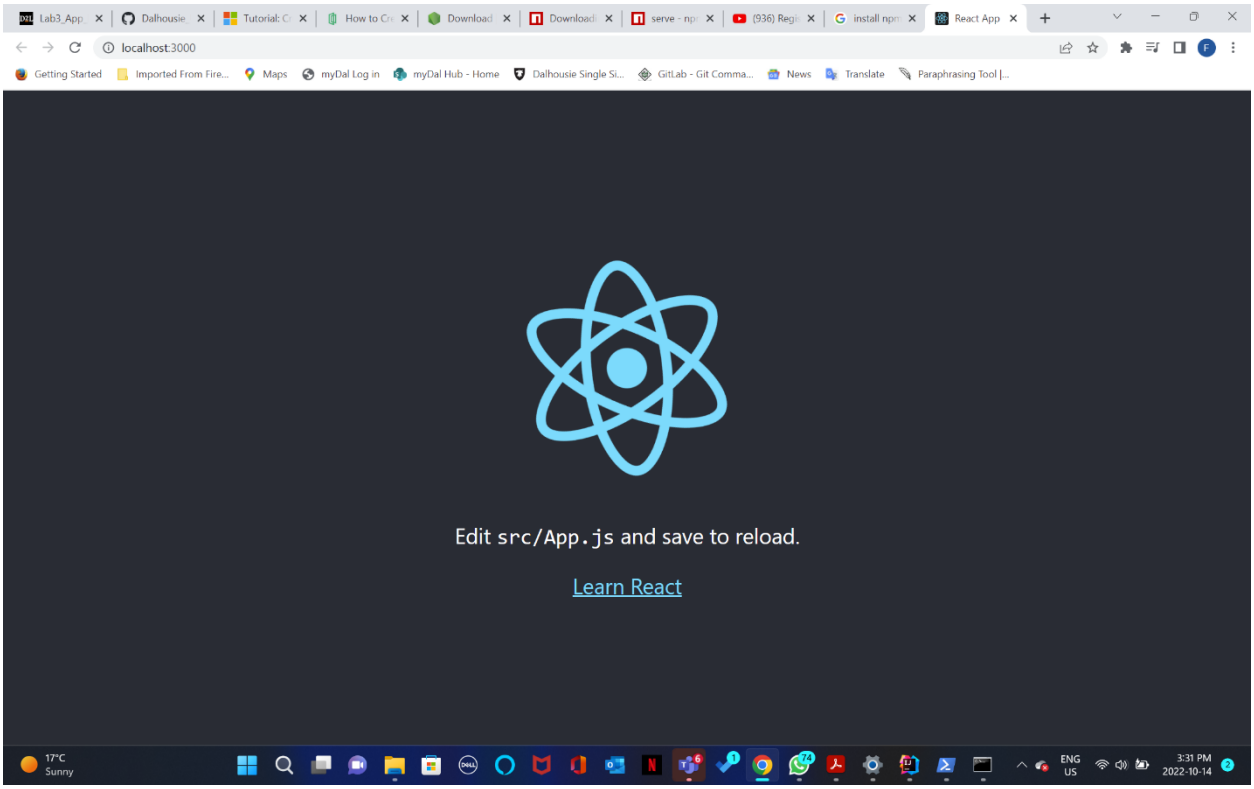


Figure 4: React-app build successfully

Step B):

TO DO: Code and required dependencies in Container #1 are responsible for accepting registration details from frontend and store it in backend database.

PROCEDURE:

I created Registration form inside the "**container-1_Register-App**". The folder consists of 4 files namely:

1. Registration.jsx
2. Firebase.jsx
3. Style.css
4. App.js

Steps I followed to create Registration form:

1. I wrote the front-end code in **Registration.jsx** for the registration which contains the User's Name, Email , Password and Location (Refer Fig. 5 & 6). Refer Fig. 7 & 8 for the CSS part which is present in the file **Style.css**.

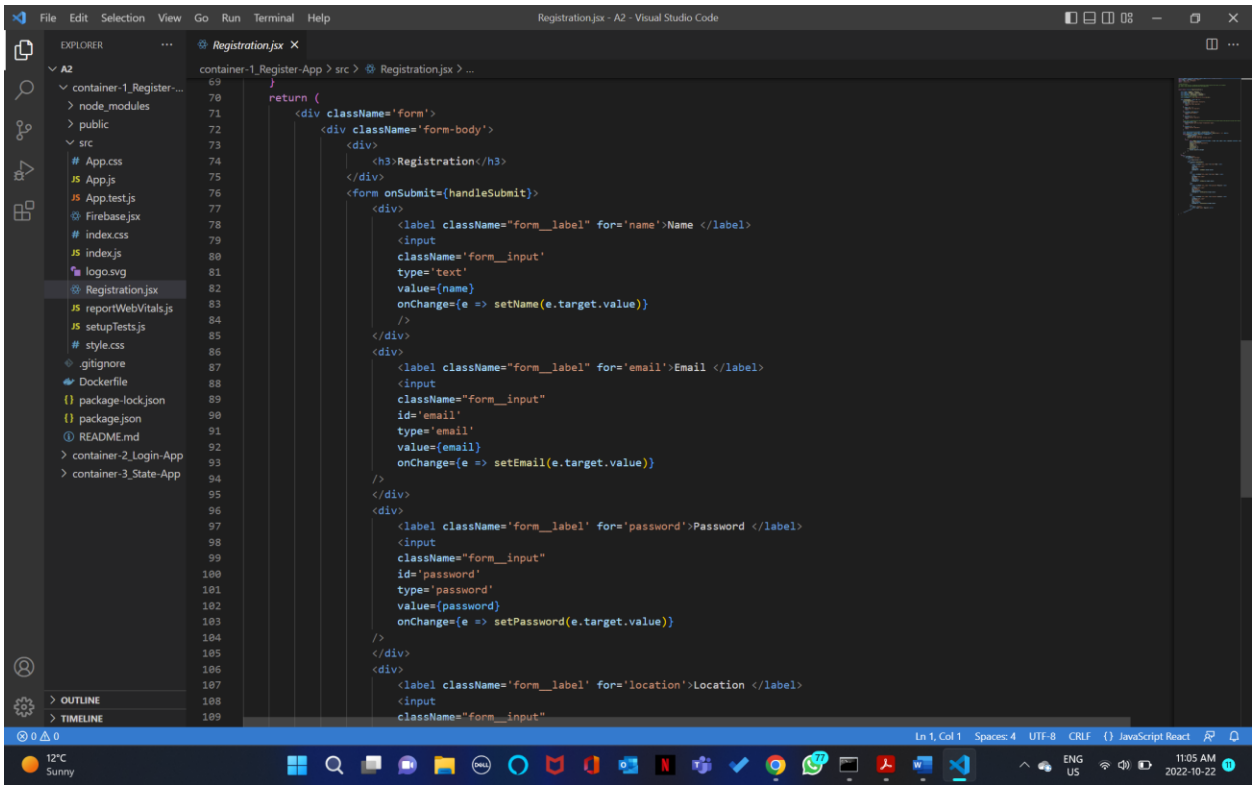


Figure 5: Front-end code for registration form

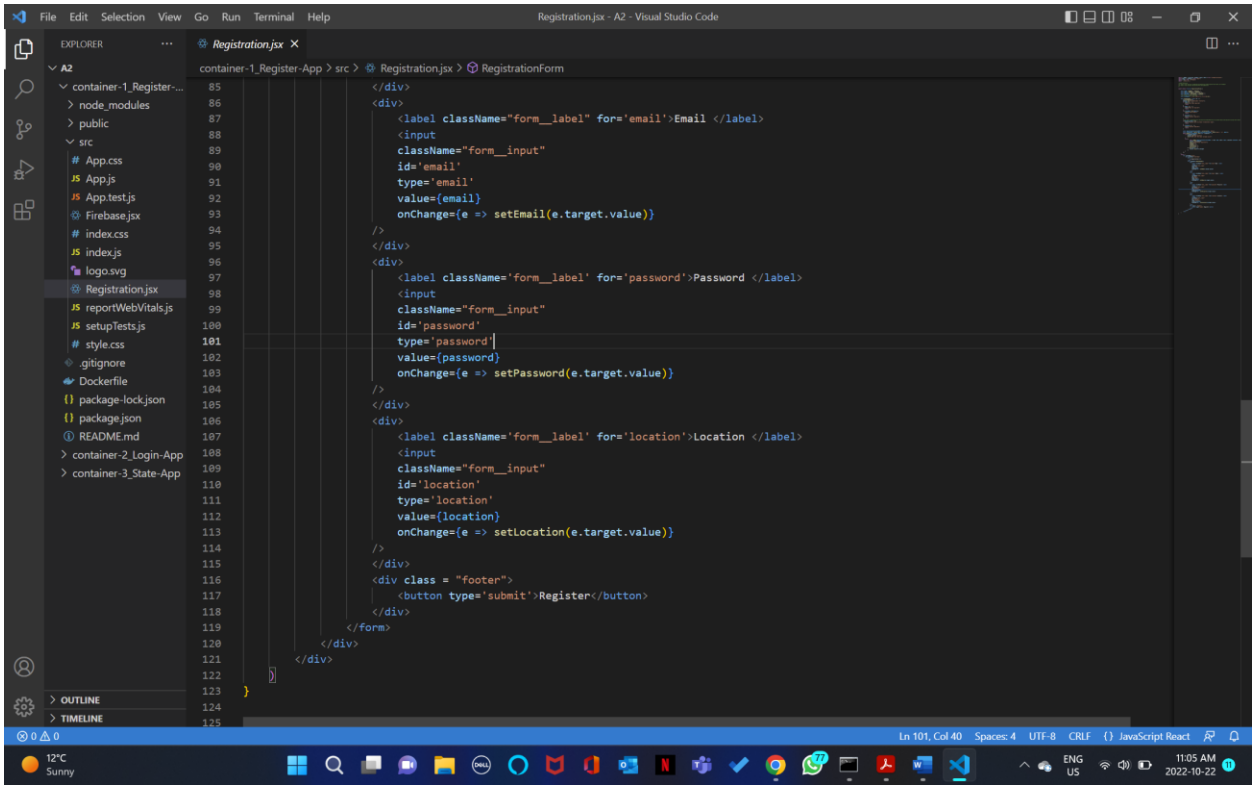


Figure 6: Front-end code for registration form

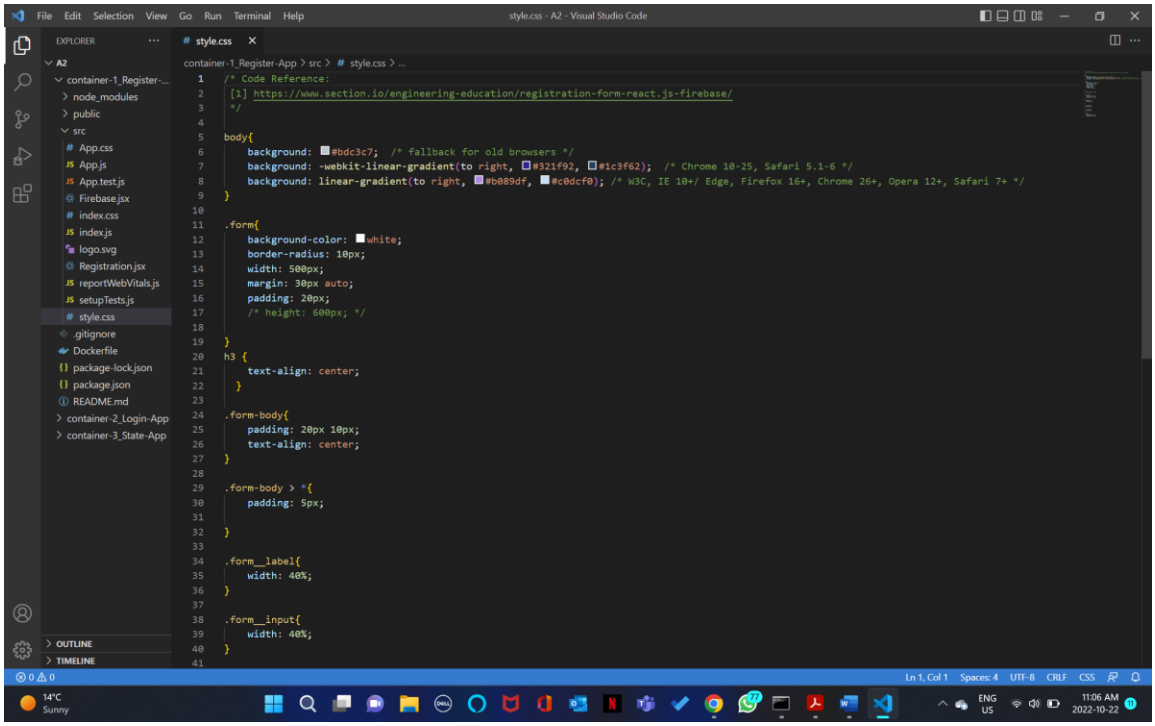


Figure 7: CSS Style code for registration form

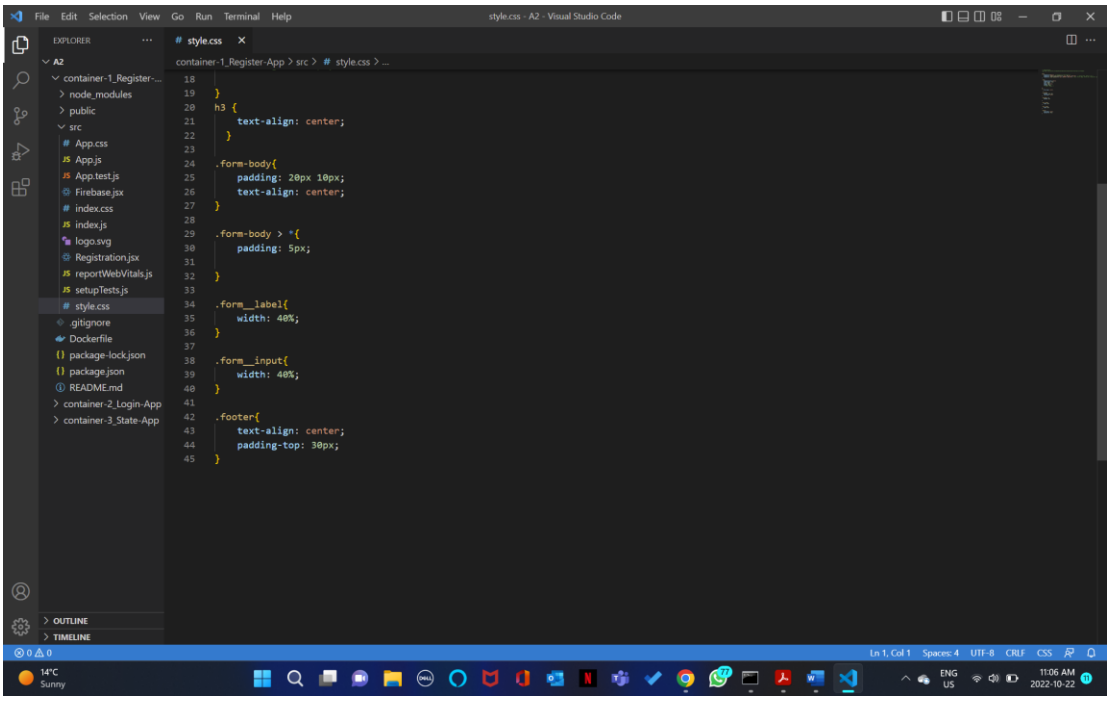


Figure 8: CSS Style code for registration form

2. To display the content to the website we mention the class in the **App.js** file [1]. The file consists of the necessary imports like **RegistrationForm** from its absolute path i.e. "**./Registration**" as shown in the Fig. 9.

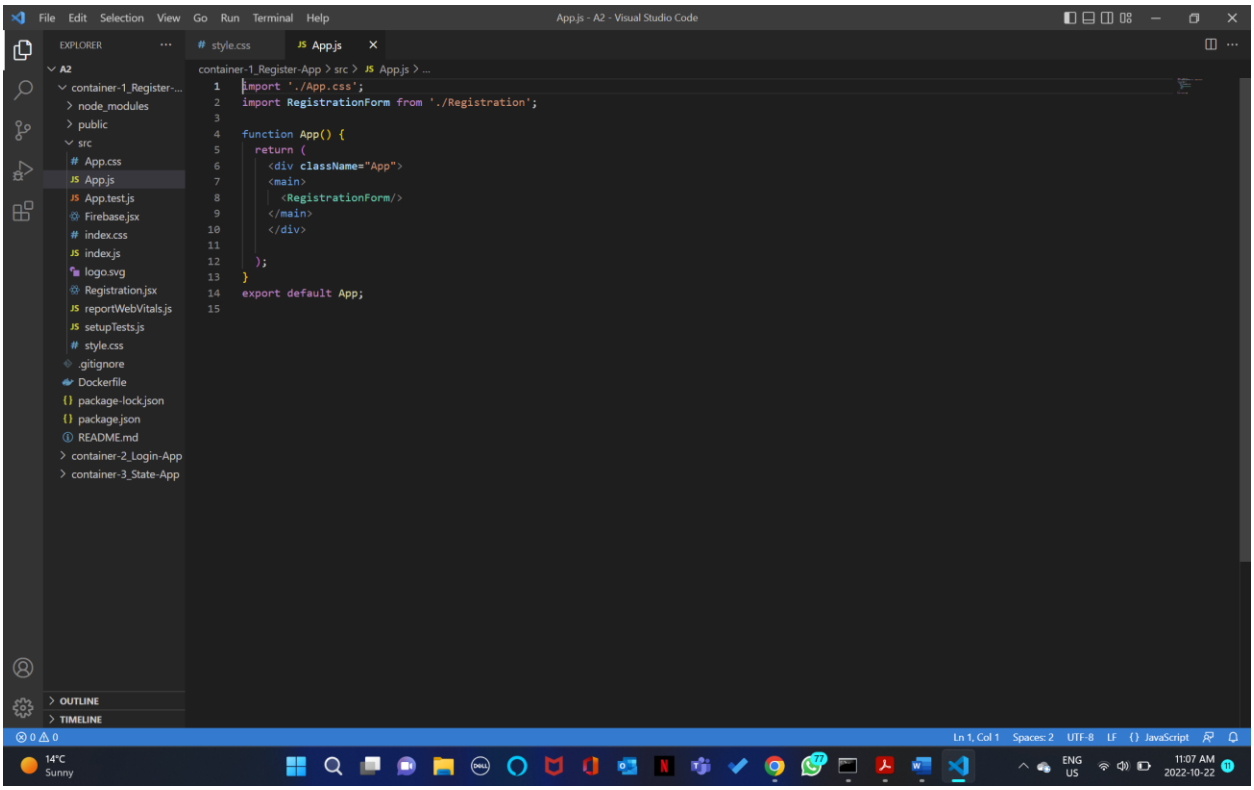


Figure 9: App.js file

3. I implemented hooks in the registration [1] i.e "useState" hook to maintain state of a variable which one can update dynamically using setState [1]. Refer Fig. 10 to see the hook I used .

```
const [name, setName] = useState('')
const [email, setEmail] = useState('')
const [password, setPassword] = useState('')
const [location, setLocation] = useState('')
```

Figure 10: Hooks in registration form

4. I used handleSubmit function in which we'll get all the values that are filled in form [1] (Refer Fig. 11). This function is used to connect the app with the firebase [1]. I implemented all the test-cases like whether the password and email is valid or not, whether the user entered empty string in any of the fields or not. This will prompt an

error message like “Invalid email”, “Invalid password”, “User name required” etc (Refer Fig. 11). This test-cases will see in the following steps. If the user entered correct values then the user will be successfully registered (Refer Fig. 12). This condition is mentioned in the “else” part of the Fig. 12 which will also create a database named “Users” in the firestore which will see in further steps.

```
const handleSubmit = async (e) => {
    e.preventDefault();
    console.log("in handlesubmit function");
    if (name === '') {
        alert("User name required")
        return
    }
    if (email === '') {
        alert("Email id required")
        return
    }
    if(!validEmail.test(email)){
        alert("Invalid email")
        return
    }
    if (password === '') {
        alert("Password required")
        return
    }
    //[3] https://stackoverflow.com/questions/68115335/how-to-validate-email-and-password-using-react-hooks
    if(password.length < 8){
        alert("Password must be atleast 8 characters long");
        return
    }
    if (location === '') {
        alert("Location required")
        return
    }
```

Figure 11: handleSubmit function in container-1_Register_App

```
const registrationCollectionRef = collection(db, 'Users')
const validateRef = query(registrationCollectionRef, where("email", "==", email));
await getDocs(validateRef).then(async (snapshot) => {
    if (!snapshot.empty) {
        console.log("User exists")
        alert("User with this email already exists")
    }
    else {
            await addDoc(registrationCollectionRef, { name: name, email: email, password: password, location: location }).then(response => {
            console.log(response)
            alert("Successfully registered");
            setName("");
            setEmail("");
            setPassword("");
            setLocation("");
        }).catch(error => {
            console.log(error.message)
        })
    }
});
```

Figure 12: Else part of handleSubmit function in container-1_Register_App

5. Next step is to connect is the registration form with the Firebase.

- Click on the “Get Started” button as shown in the Fig. 13.
- Go to “Add Project” to create new Project as shown in the Fig. 14.
- It will take you to new page and will ask you to enter name for new project as shown in the Fig. 15.
- I created the project named: “CSCI-5410-Serverless-A2” as shown in the Fig. 16.
- Next go to “project Overview” and in that go to “Firebase Products and Features” and click on the Cloud Firestore. which is shown in Fig. 17.
- To build a connection, we need to copy code from the firebase page which I found under “Project settings” as shown in the Fig. 18. Copy the provided code and paste it in the “**Firebase.jsx**” file as shown in the Fig 19.

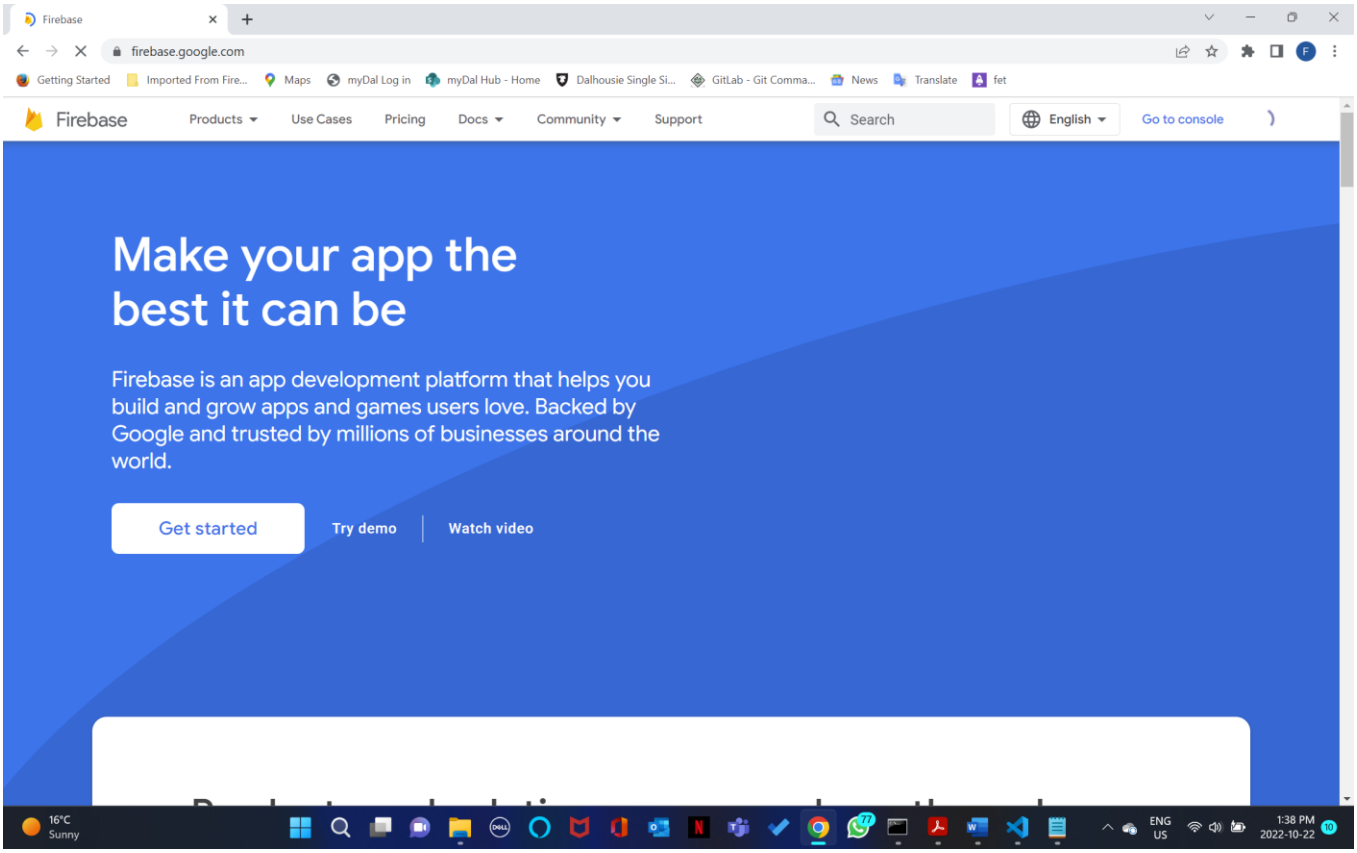


Figure 13: Firebase webpage

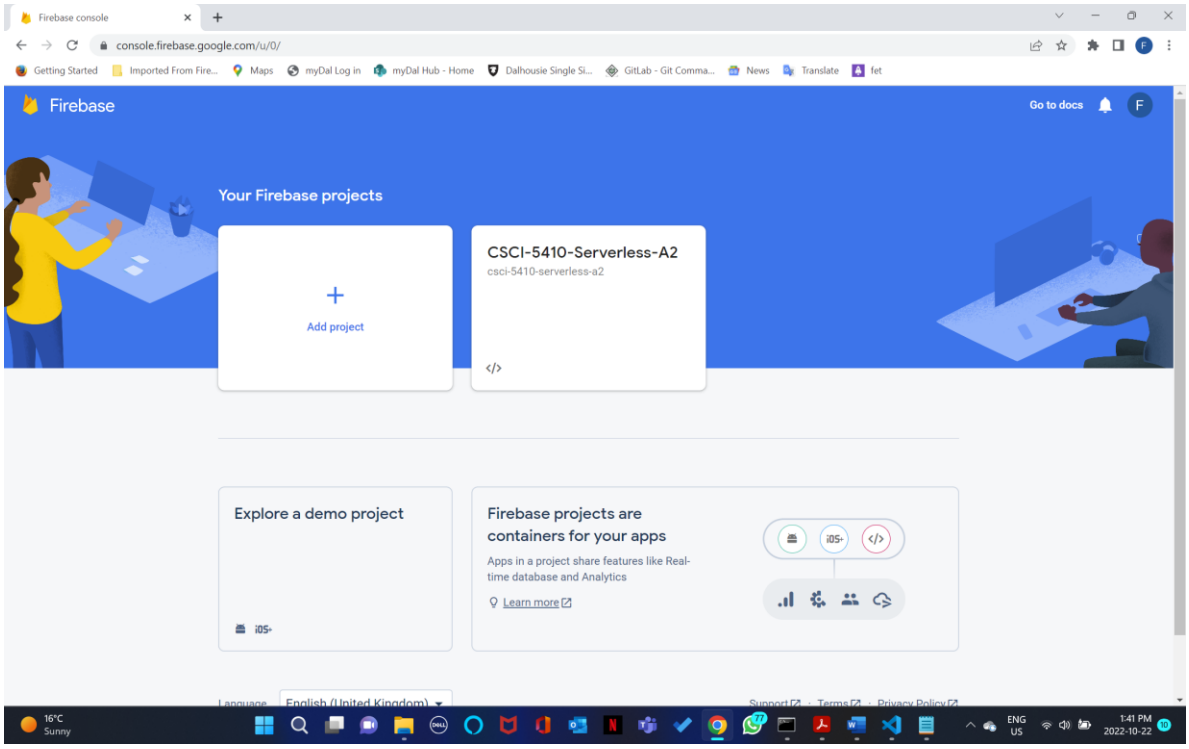


Figure 14: Firebase console webpage

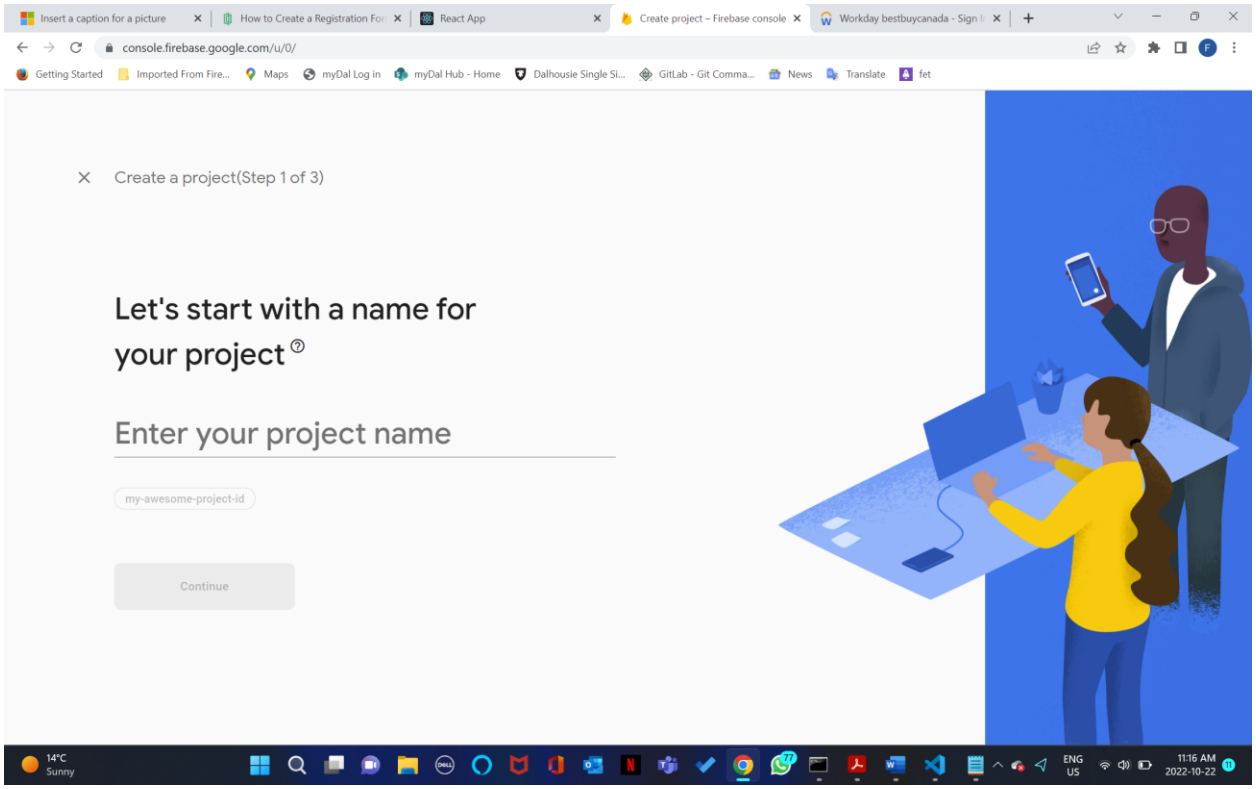


Figure 15: Firebase console webpage

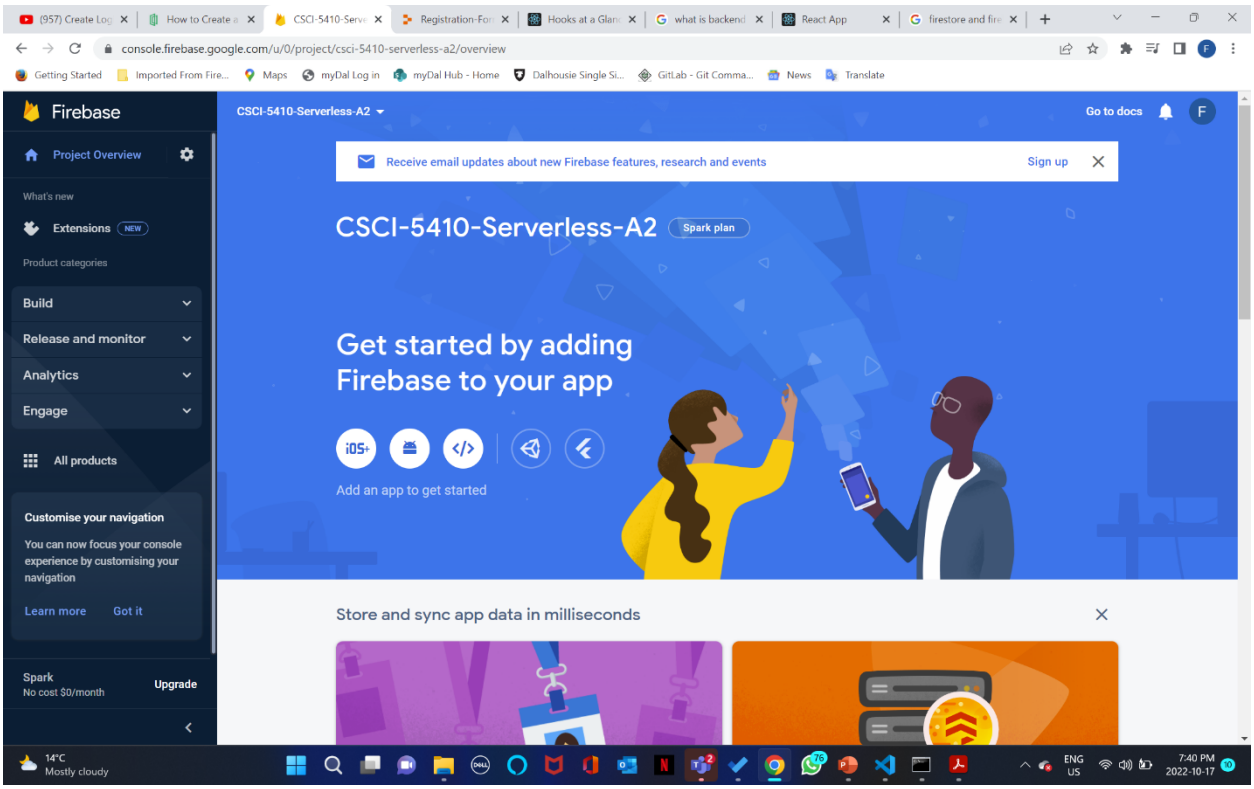


Figure 16: Firebase console webpage having the project "CSCI-5410-Serverless-A2"

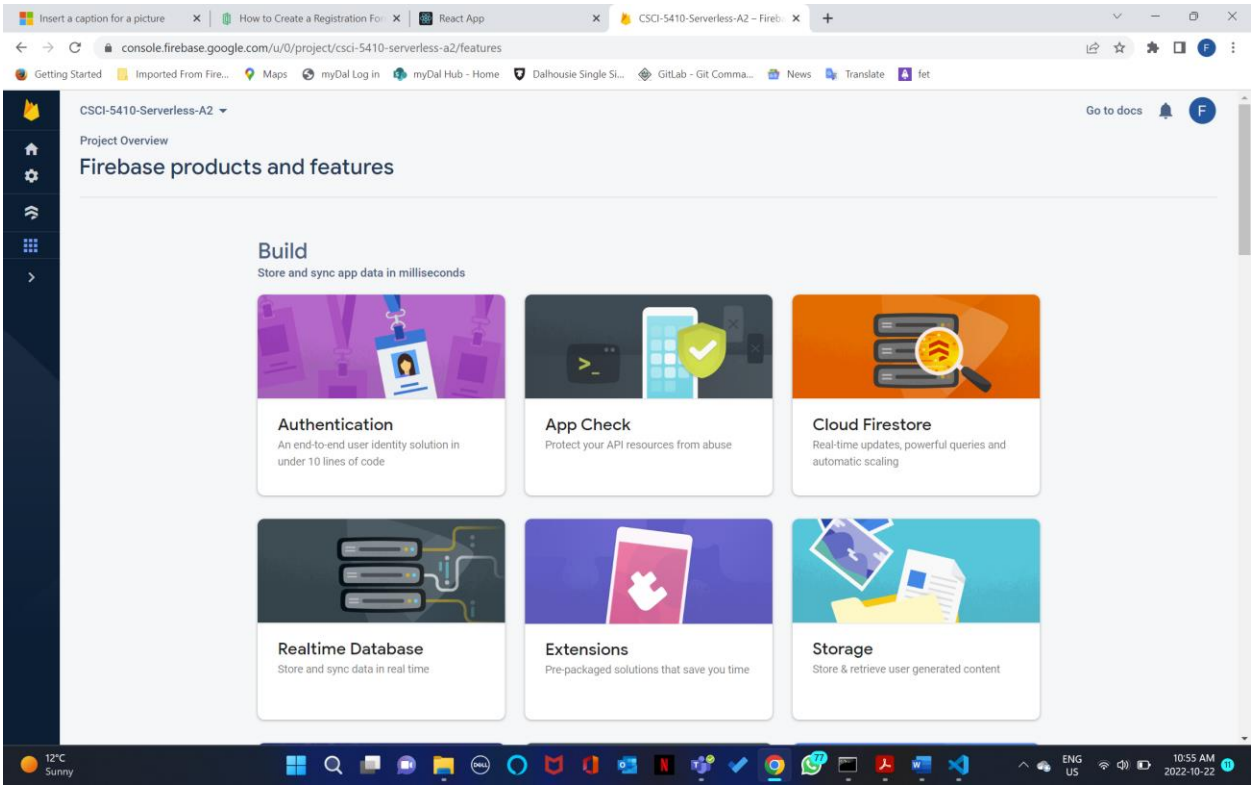


Figure 17: Firebase products and features page

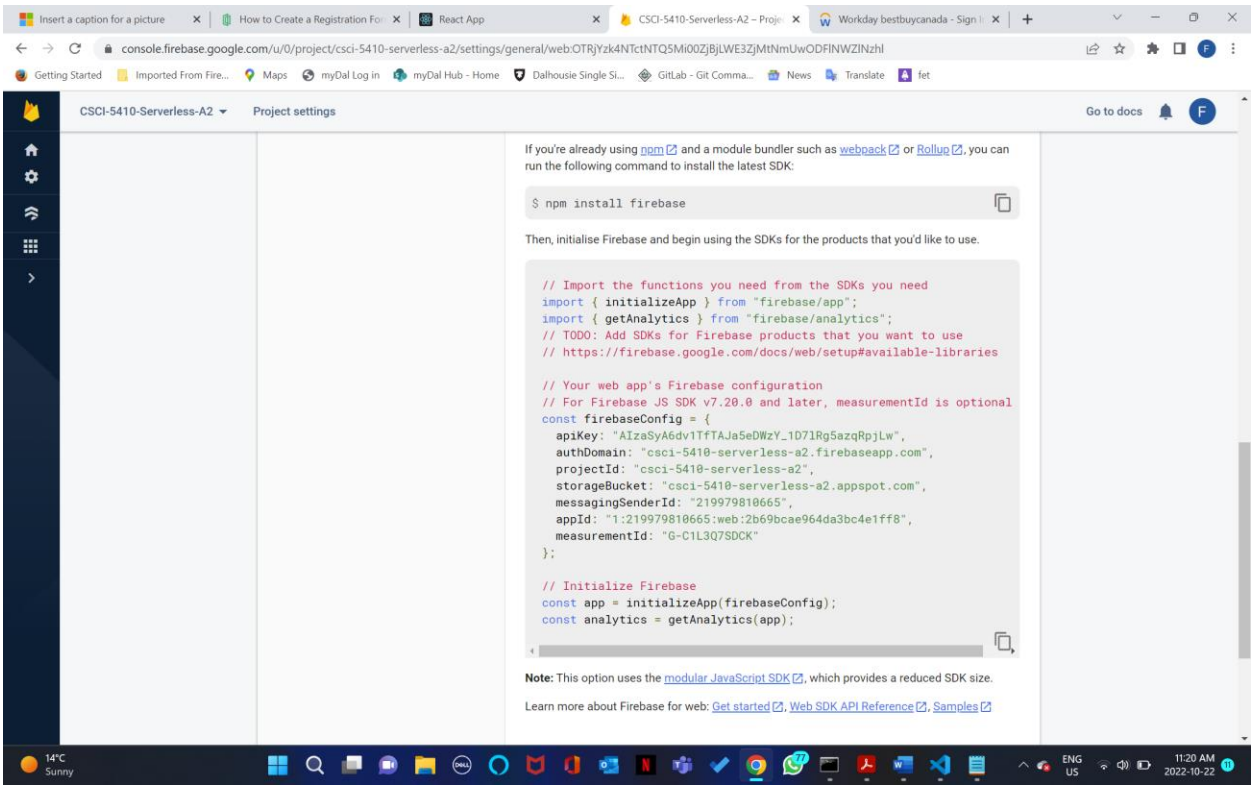


Figure 18: Firebase connection code

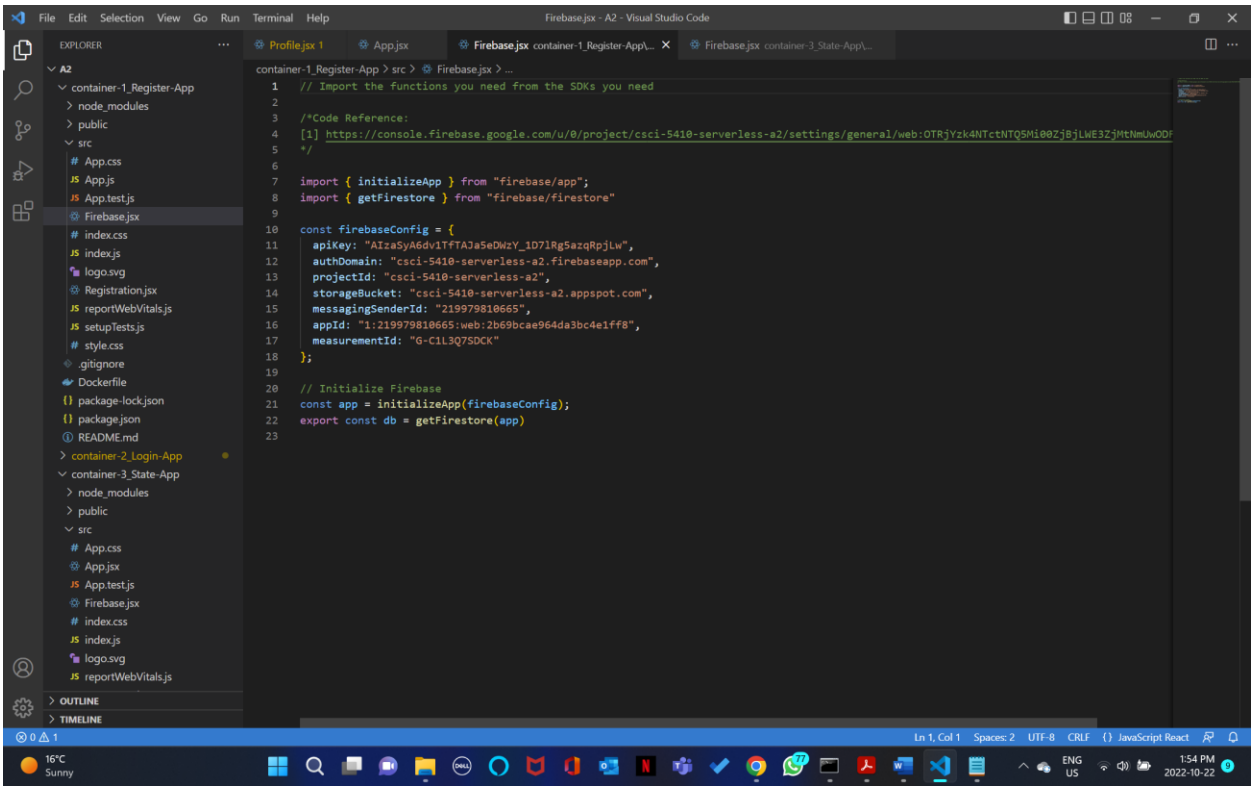


Figure 19: Firebase connection code in the "Firebase.jsx" file

6. Before running the command we see that the Firestore is empty, there is no database as shown in the Fig. 20.

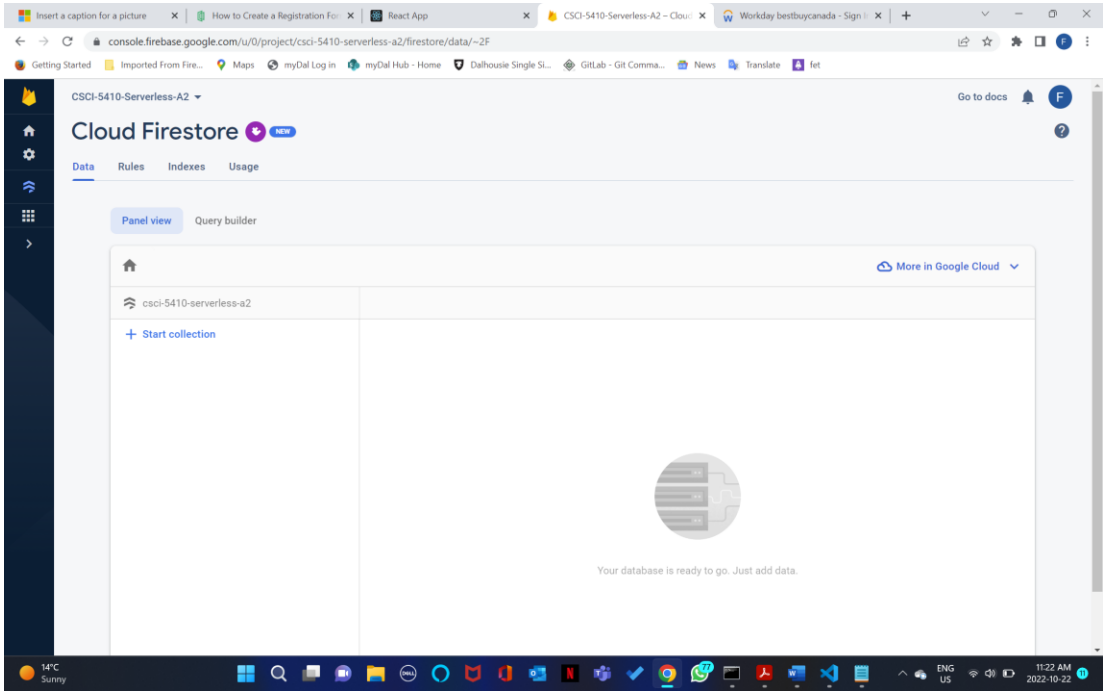


Figure 20: Cloud Firestore without any collection

7. Now Run the command by providing the path of the container and write “npm start” to launch the register app as shown in the Fig 21 & 22. The registrationForm is launched successfully which is shown in the Fig. 23.

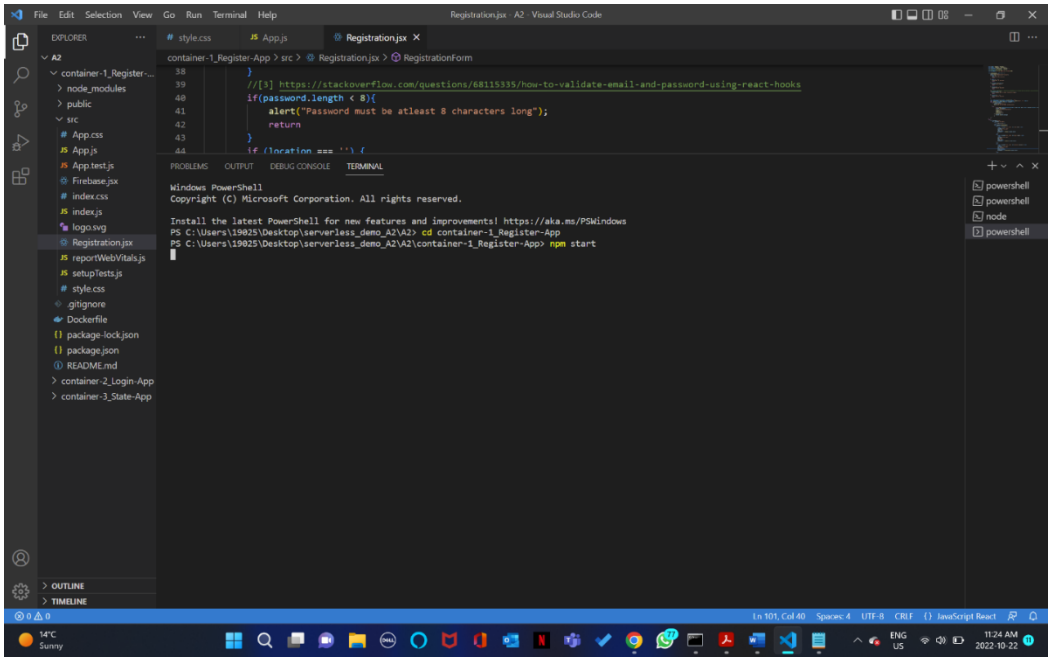


Figure 21: Running the app with the npm commands

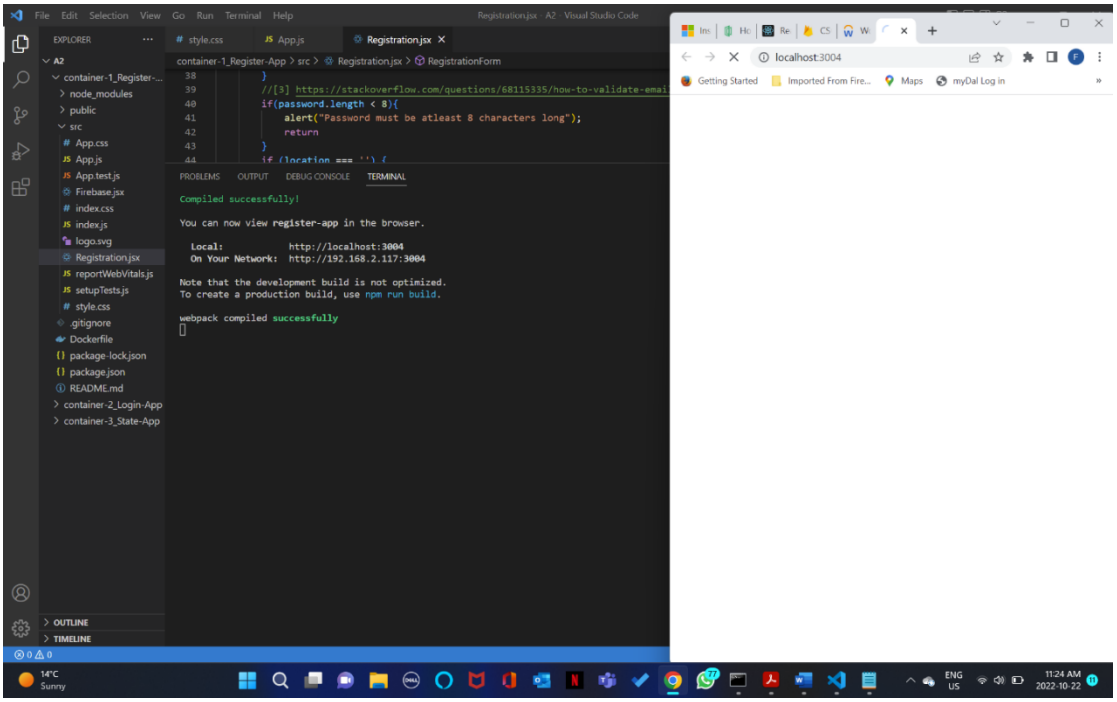


Figure 21: Running the app with the npm commands

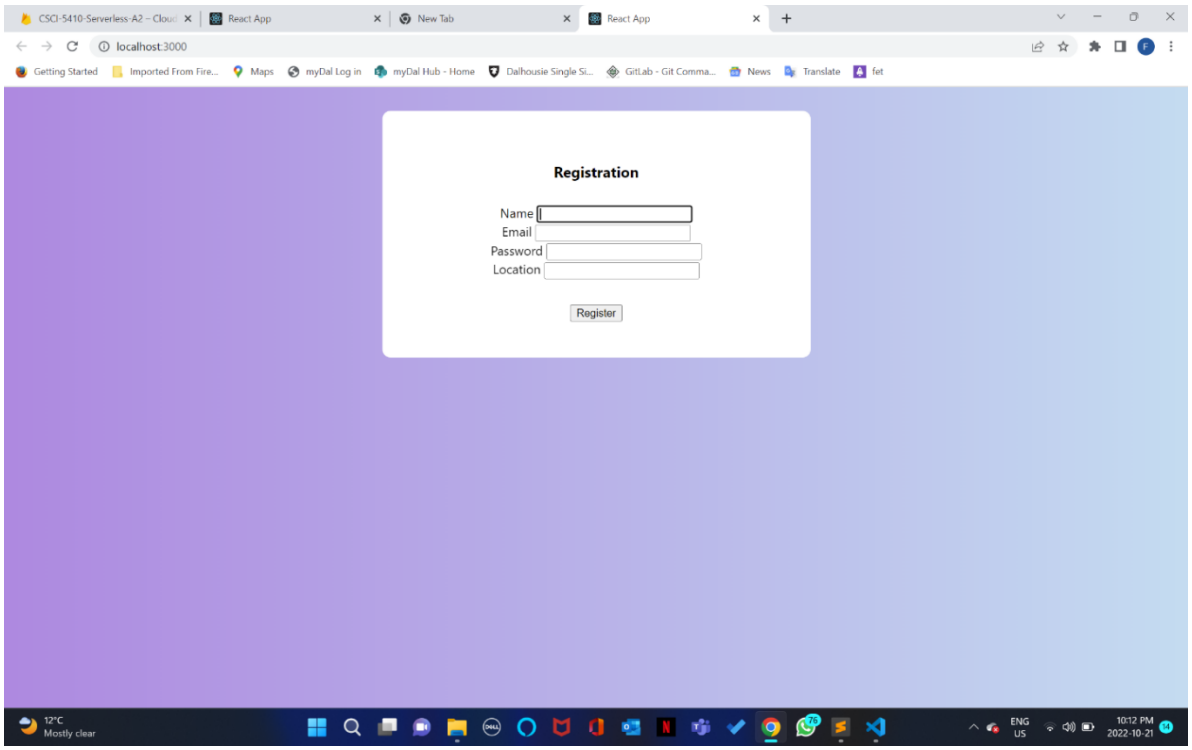


Figure 21: Registration form

8. After launching the app, the user filled the details as shown in the Figures below.

I provided different Test Cases while registering the form which is shown below:

- **Test case 1:** Fig 22. Prompts the alert message “Email id required” when user skipped entering the “email”. This will not allow user to register without entering the email id.

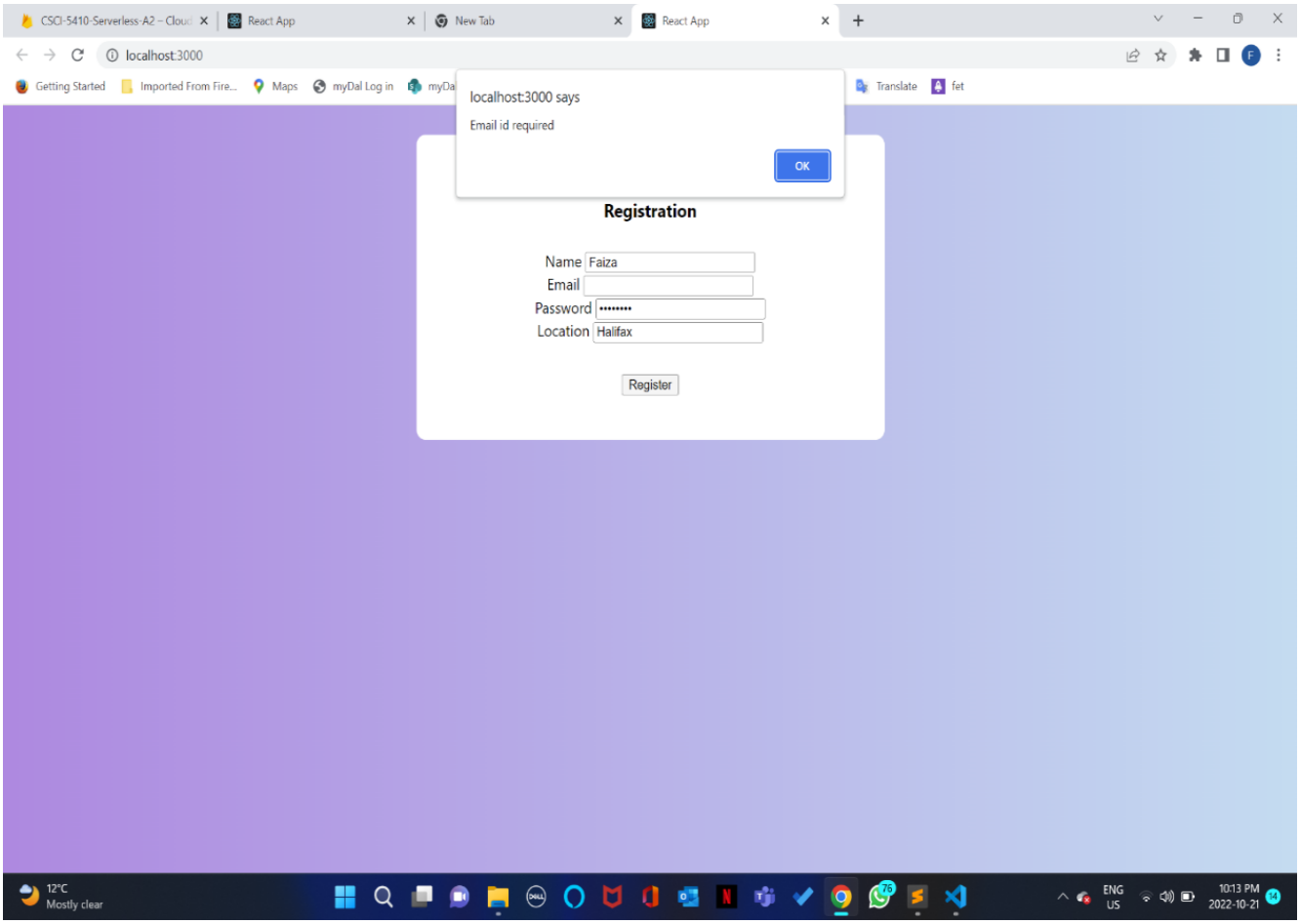


Figure 22: Alert message – Email id required

- **Test case 2:** If the user entered the email but the email entered is invalid i.e. not following the email regex pattern or conditions, then the user will be prompted to enter valid email that follows the email naming conventions (Refer Fig. 23).

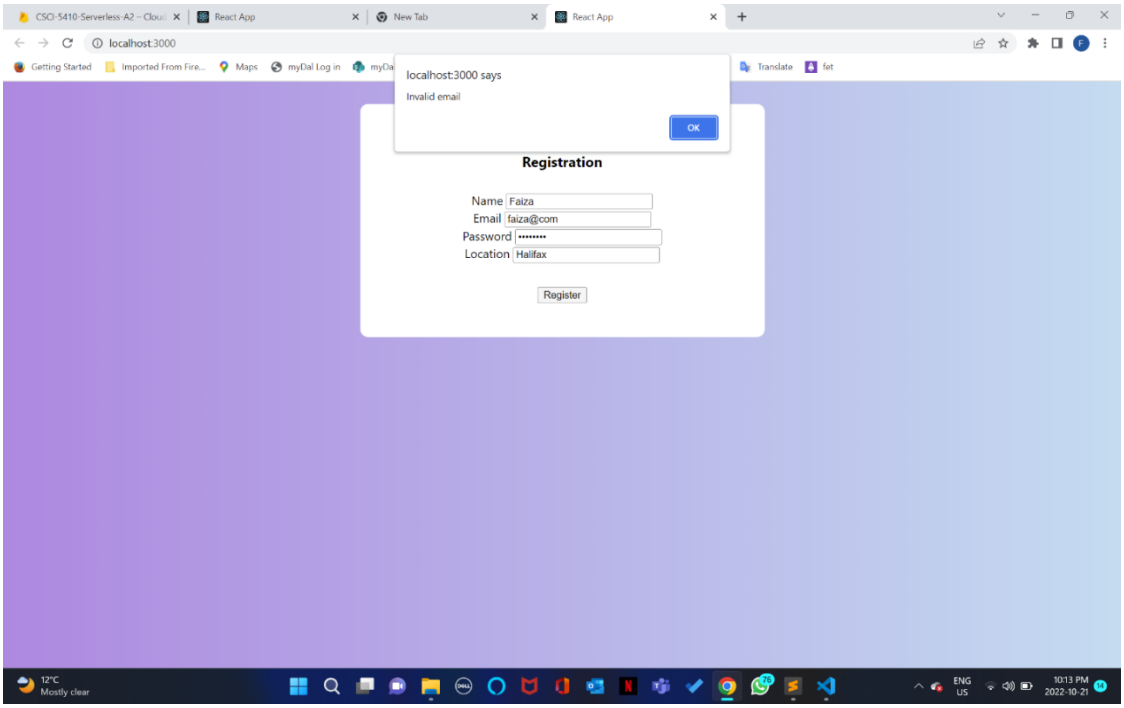


Figure 23: Alert message – Invalid email

- **<u>Test case 3:</u>** If the user entered the password but the password entered is less than 8 characters then the user will be prompted to enter password that is atleast 8 characters long. We need long passwords for security reasons. Refer Fig. 24 to see the alert message.

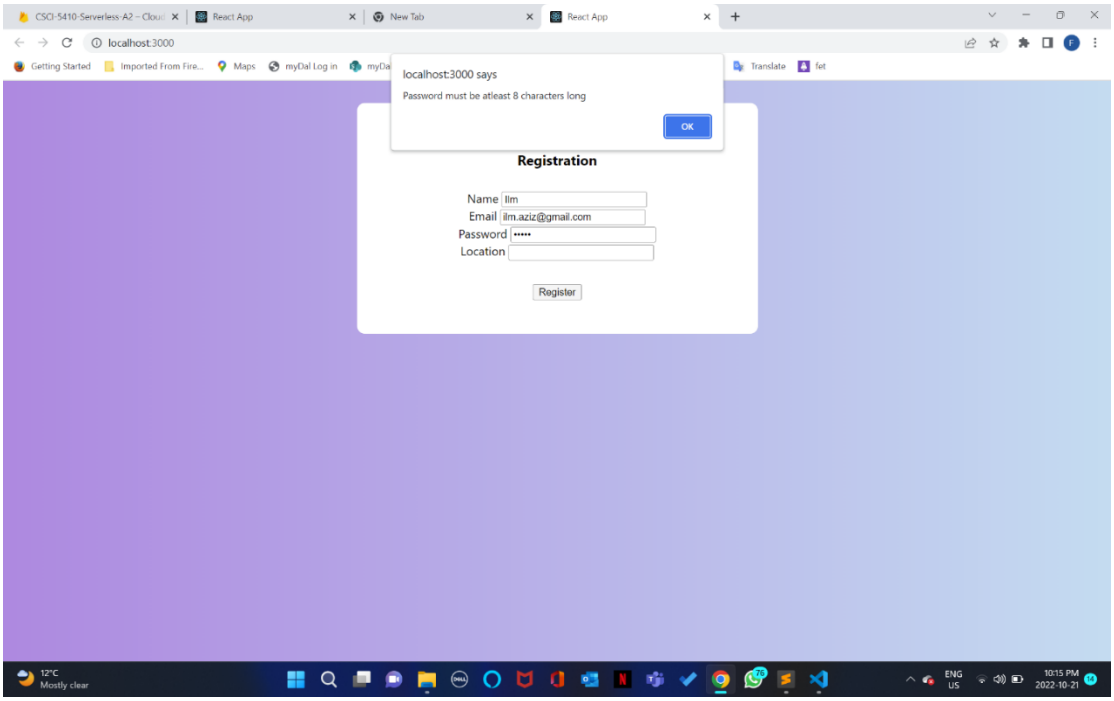


Figure 24: Alert message – Invalid email

- **Test case 4:** If the user clicks on the submit without entering any of the field, then the user will be prompted to enter values to mandatory fields. Without which user can't register themselves. In Fig 25, the user missed mentioning the location which showed "Location required" as an alert message.

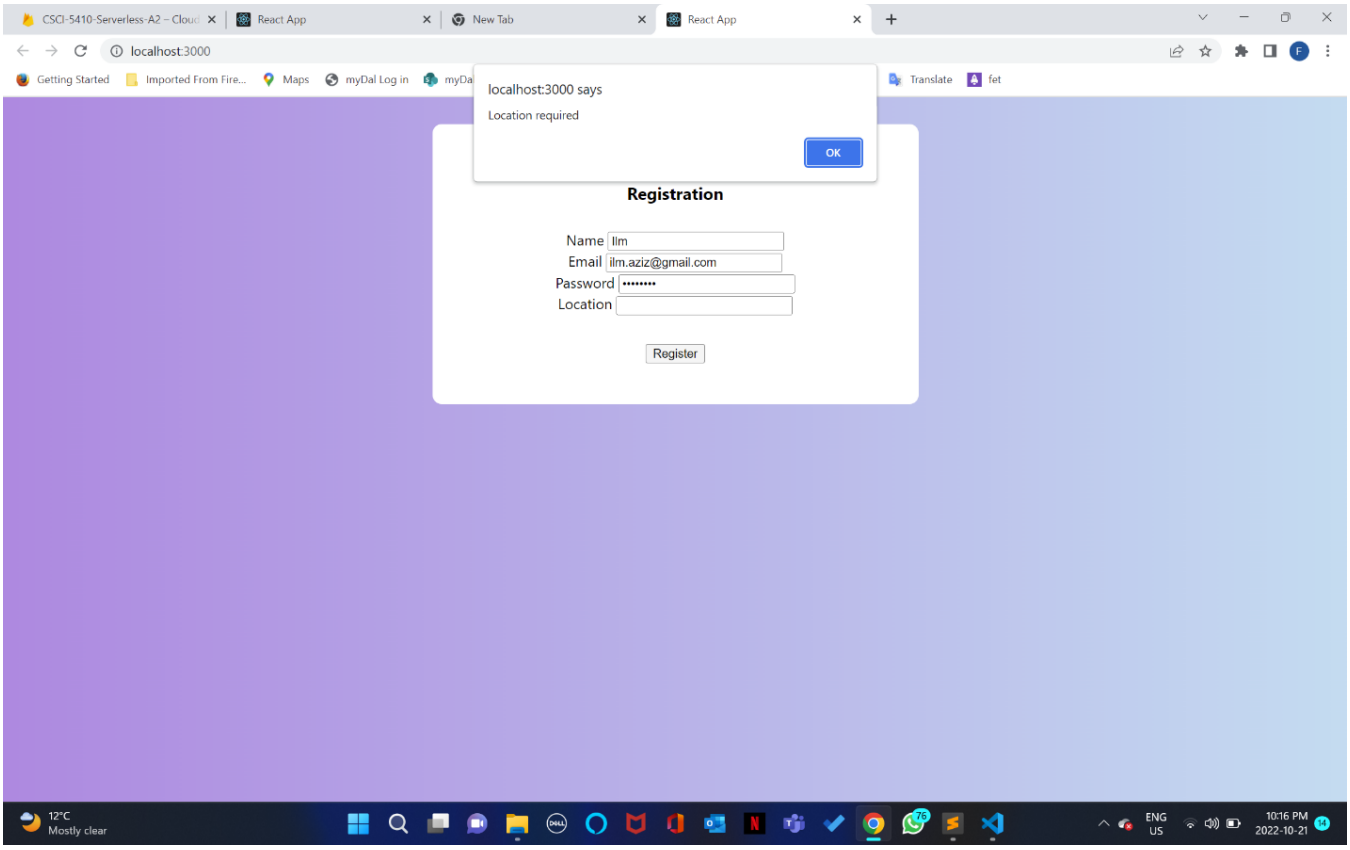


Figure 25: Alert message – Invalid email

- **Test case 5:** If the user entered the email which is already present in the database that means the user with that email id is already registered. So, any other user cannot register with the same email id because the email id is unique for all users. So the user will be prompted with the alert message "User with this email already exits" as shown in the Fig 26.

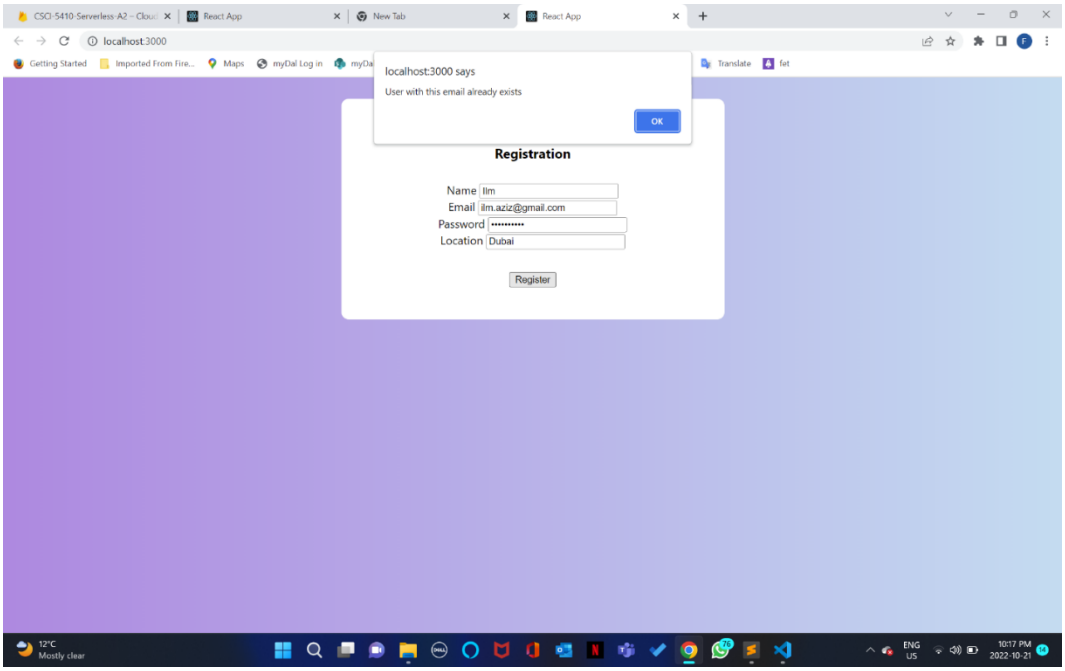


Figure 26: Alert message – User with this email already exists

- **<u>Test case 6:</u>** If the user entered all the details properly and have not entered registered email, then the user will be registered successfully as shown in the Fig. 27.

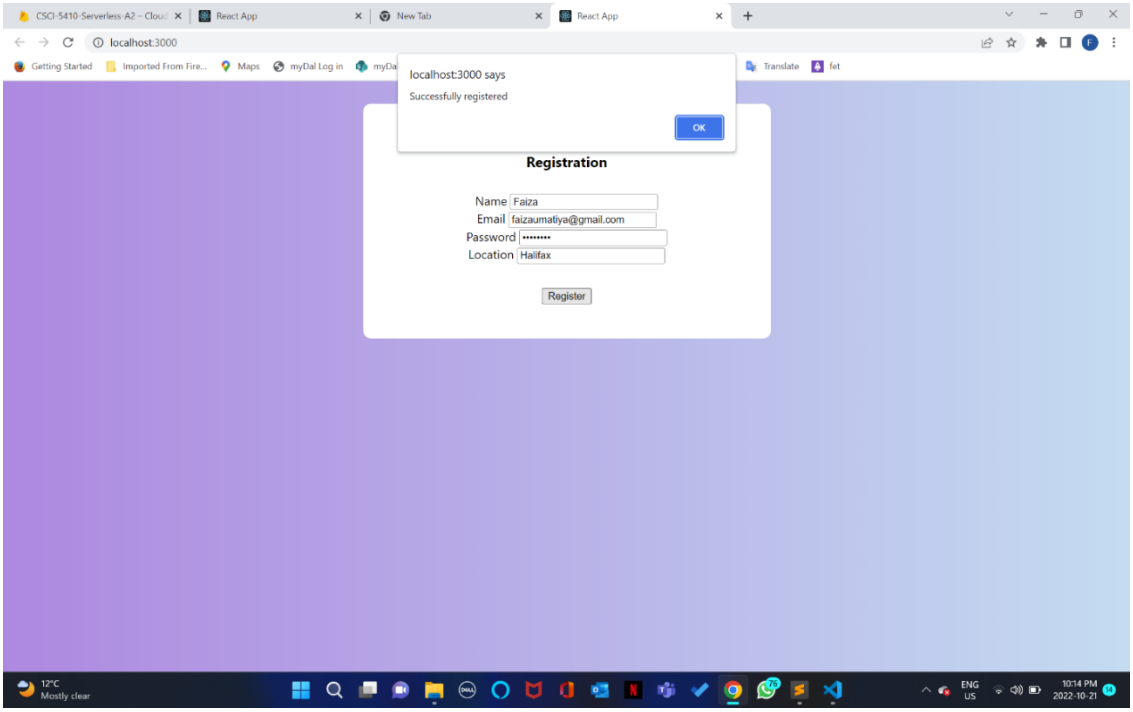


Figure 27: Alert message – User with this email already exists

9. After the user is registered successfully, the database is created in the Firestore having the name “Users” with the registration details as shown in the Fig. 28. And the values from the registration form is set to empty as shown in the Fig 29.

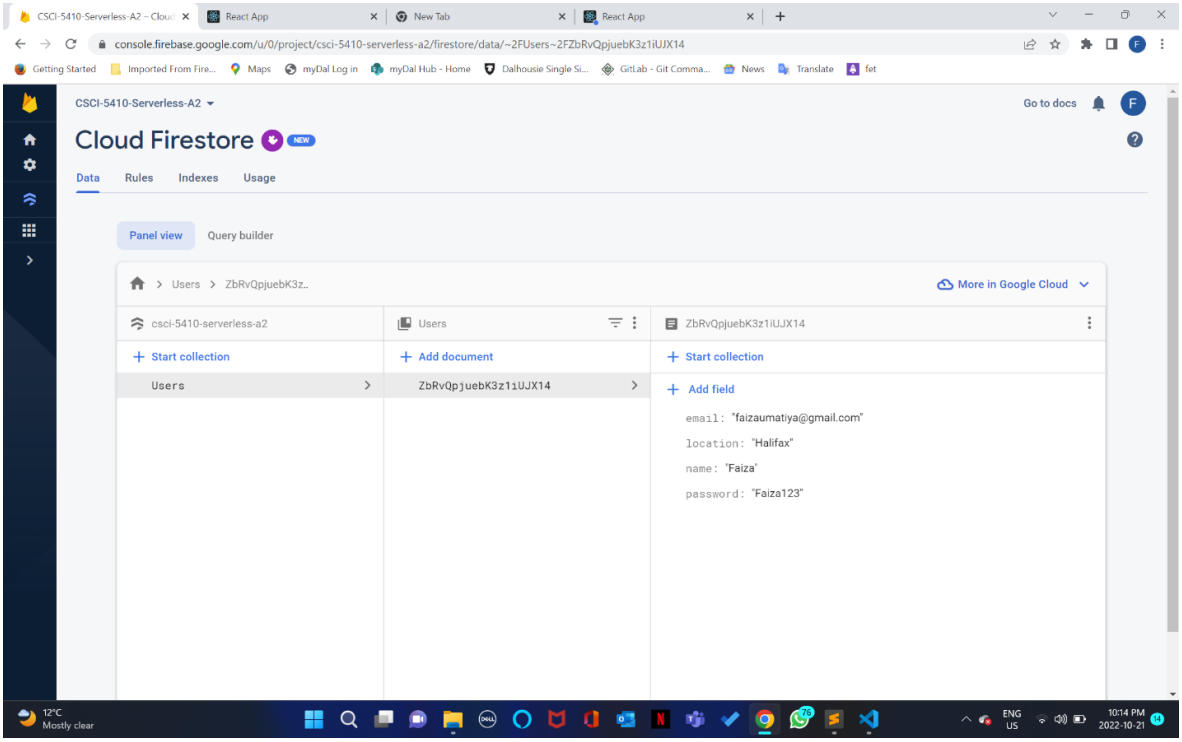


Figure 28: New “Users” database created in the Cloud Firestore

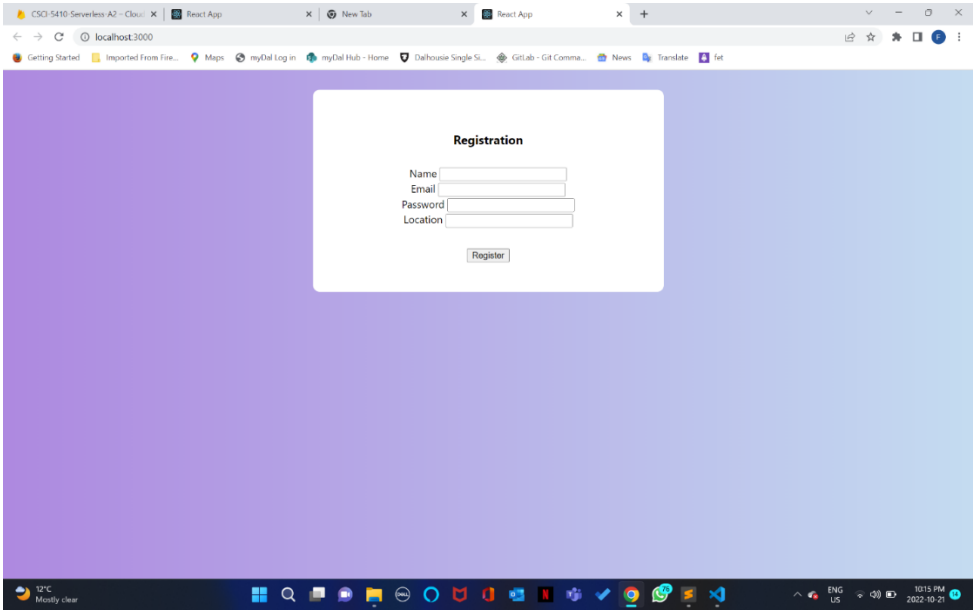


Figure 29: Values set empty after registering successfully

Other Test Cases: The user cant submit the registration if one of the details are missing, be it name, password, email or location. Refer Fig. 30, 31, 32 & 33.
Fig. 34 shows that all the users with their details are mentioned in the Firestore.

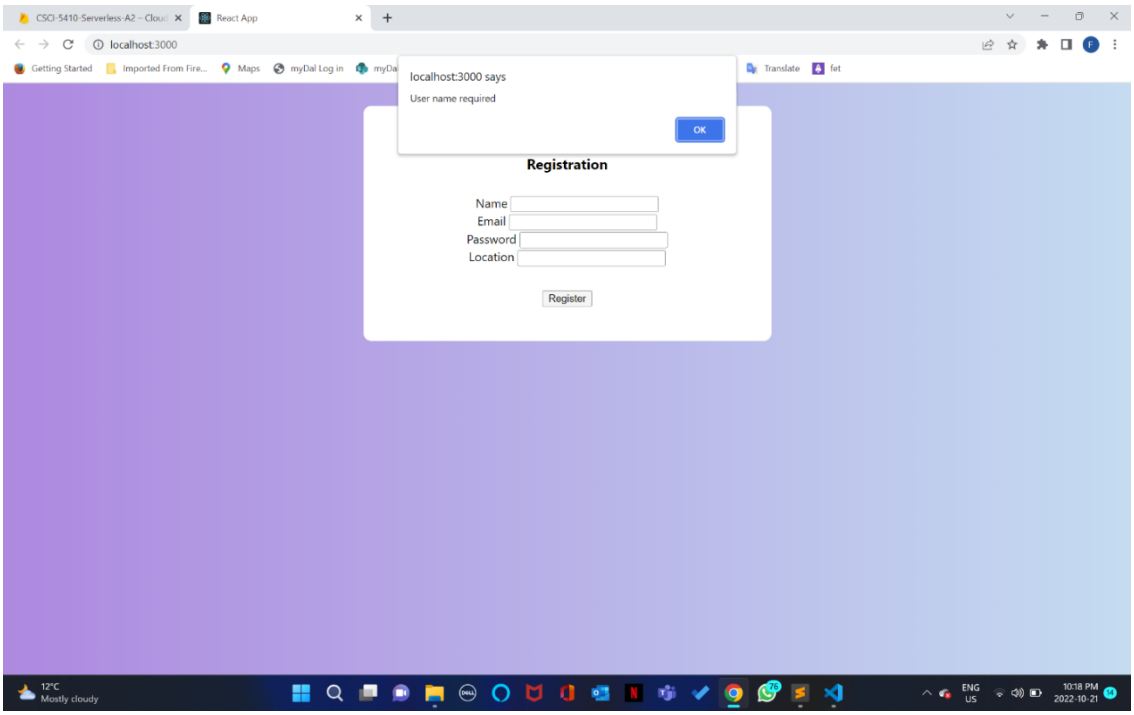


Figure 30: Alert message – User name required

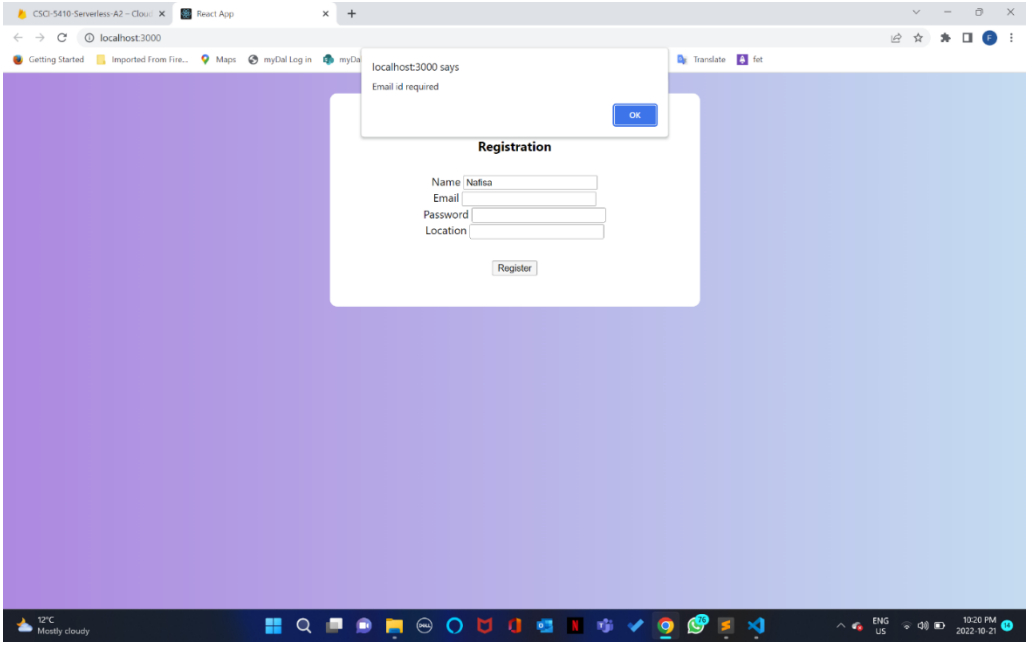


Figure 31: Alert message – Email required

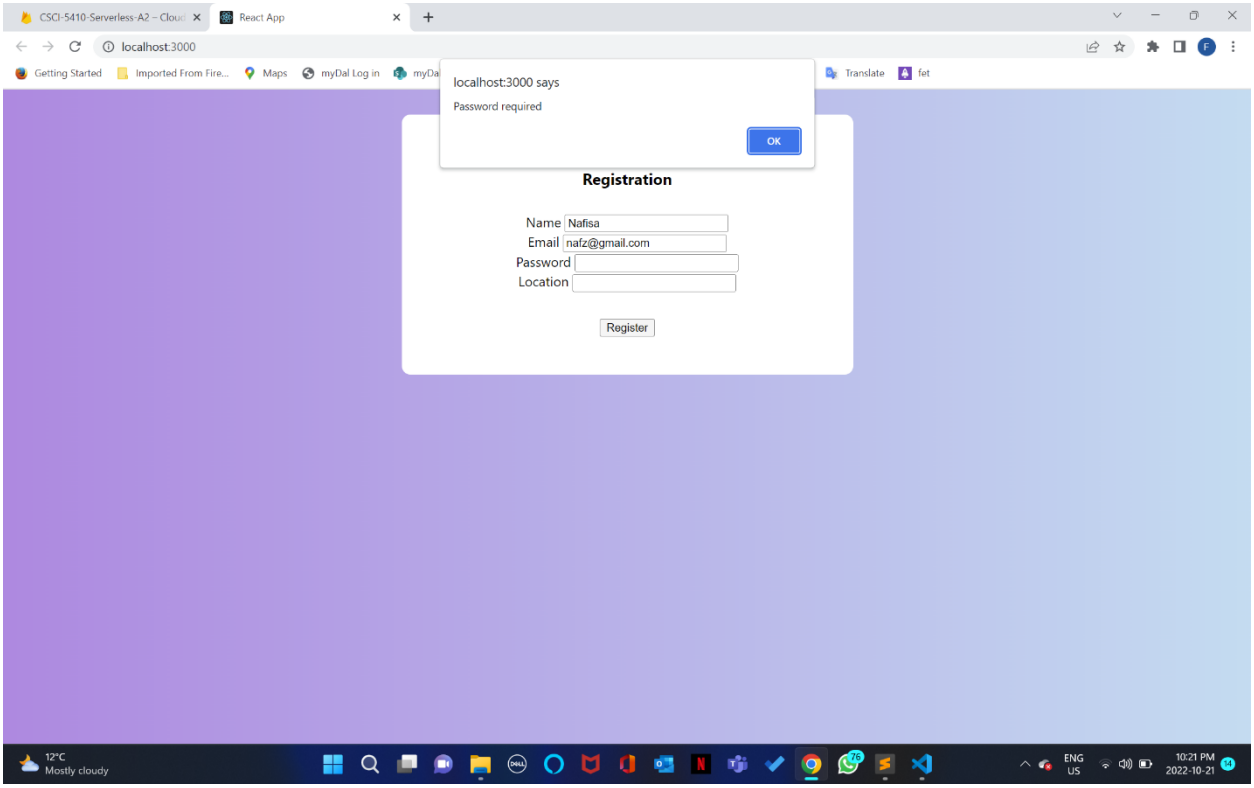


Figure 32: Alert message – Password required

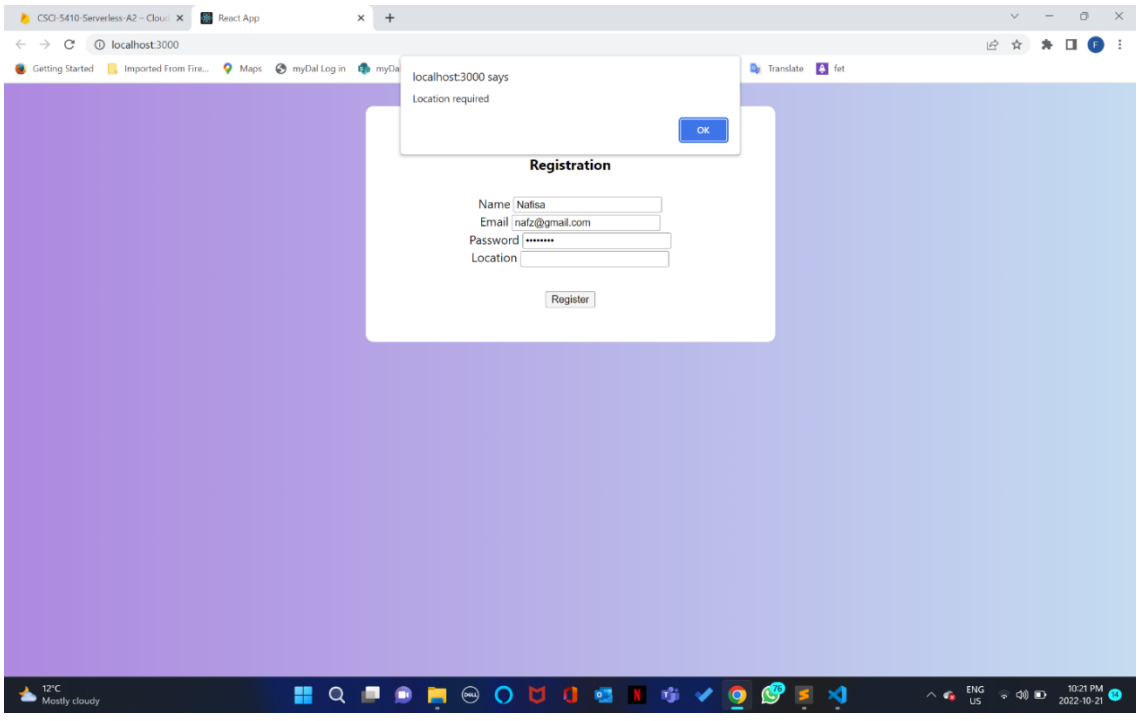


Figure 33: Alert message – Location required

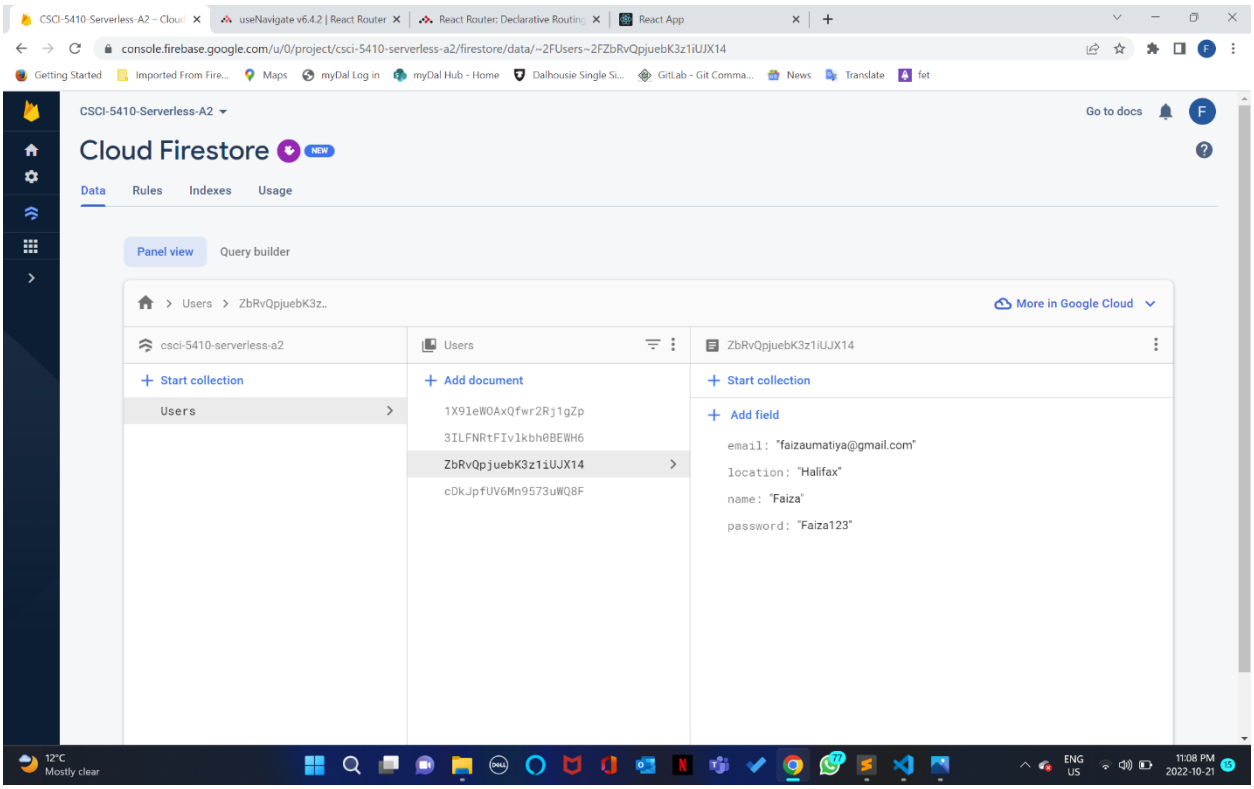


Figure 34: Users Database with user's registration details

Step C):

TO DO: Code and required dependencies in Container #2 are responsible for validating the Login information during a login process, and launching profile page with the registration information of the user, which will come from the Firestore (image 2).

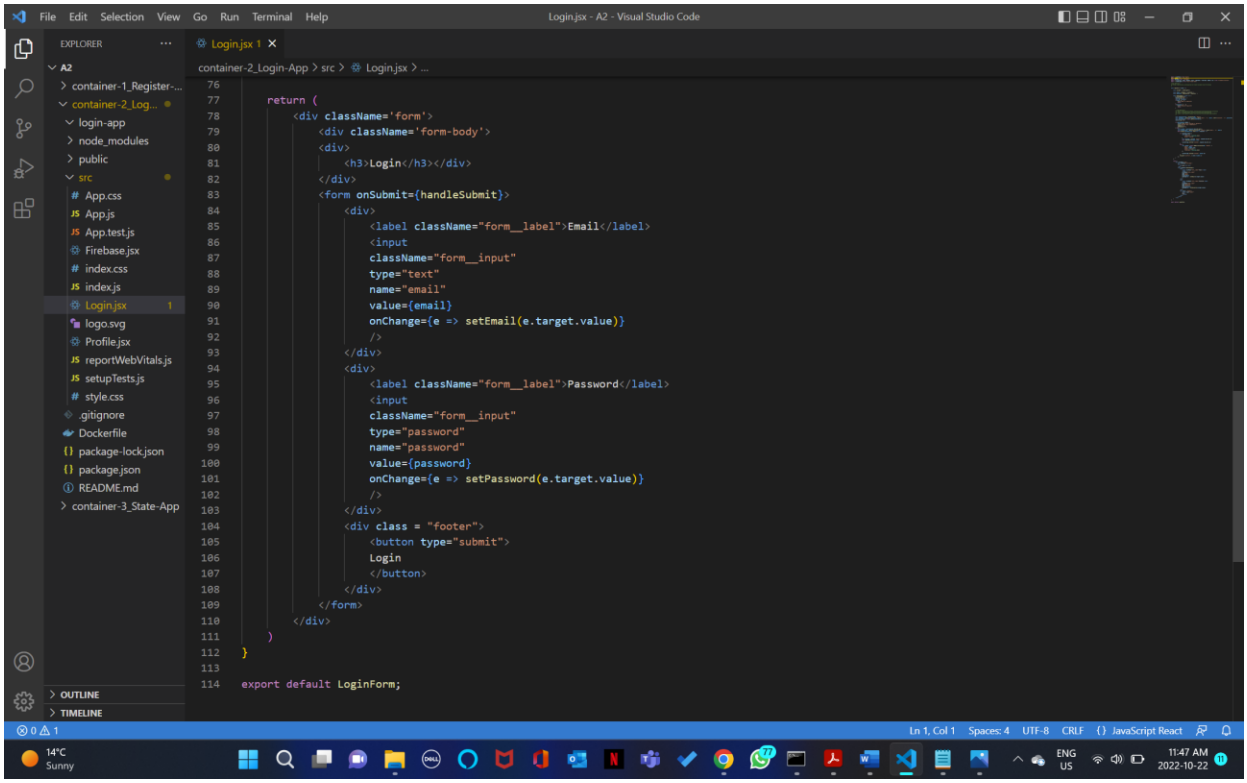
PROCEDURE:

I created Login form inside the "**container-2_Login-App**". The folder consists of 5 files namely:

1. Login.jsx
2. Profile.jsx
3. Firebase.jsx
4. App.js
5. Style.css

Steps I followed to create Login form:

1. I wrote the front-end code in **Login.jsx** (Refer Fig. 35) and **Profile.jsx** (Refer Fig. 36) for the Login and Profile page respectively. The front-end code for Login contains the User's Email and Password (Refer Fig. 35). Whereas the front-end code for the Profile page displays user's name, email and location as shown in the Fig. 36. Refer Fig. 37 & 38 for the CSS part which is present in the file **Style.css**.




```
76      return (
77          <div className='form'>
78              <div className='form-body'>
79              <div>
80                  <h3>Login</h3></div>
81              </div>
82              <form onSubmit={handleSubmit}>
83                  <div>
84                      <label className="form__label">Email</label>
85                      <input
86                      className="form__input"
87                      type="text"
88                      name="email"
89                      value={email}
90                      onChange={e => setEmail(e.target.value)}
91                      />
92                  </div>
93                  <div>
94                      <label className="form__label">Password</label>
95                      <input
96                      className="form__input"
97                      type="password"
98                      name="password"
99                      value={password}
100                     onChange={e => setPassword(e.target.value)}
101                     />
102                 </div>
103                 <div class = "footer">
104                     <button type="submit">
105                     Login
106                     </button>
107                 </div>
108             </form>
109         </div>
110     )
111 }
112
113 export default LoginForm;
114
```

Figure 35: Front-end code for Login.jsx

```
    return(
        <div className='form'>
            <div className='form-body'>
                <div>
                    <h3>Profile Page</h3>
                </div>
                <form>
                    <div>
                        <label className="form__label" for='name'>Name:</label>
                        {location.state.name}
                    </div>
                    <div>
                        <label className="form__label" for='email'>Email:</label>
                        {location.state.email}
                    </div>
                    <div>
                        <label className='form__label' for='location'>Location: </label>
                        {location.state.location}
                    </div>
                    <div class = "footer">
                        <button type="submit" onClick={logout}> Logout </button>
                    </div>
                </form>
            </div>
        </div>
    );
}

export default Profile;
```

Figure 36: Front-end code for Profile.jsx

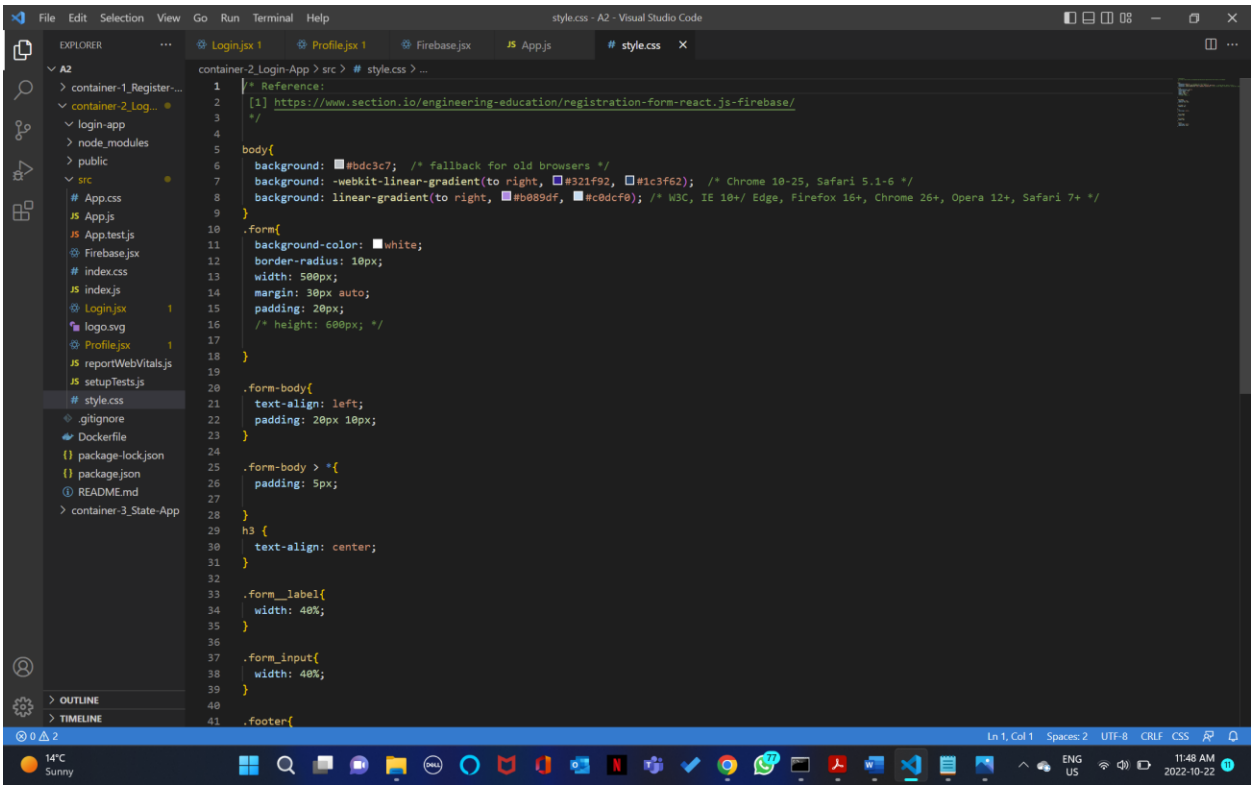


Figure 37: CSS Style code for Profile.jsx and Login.jsx

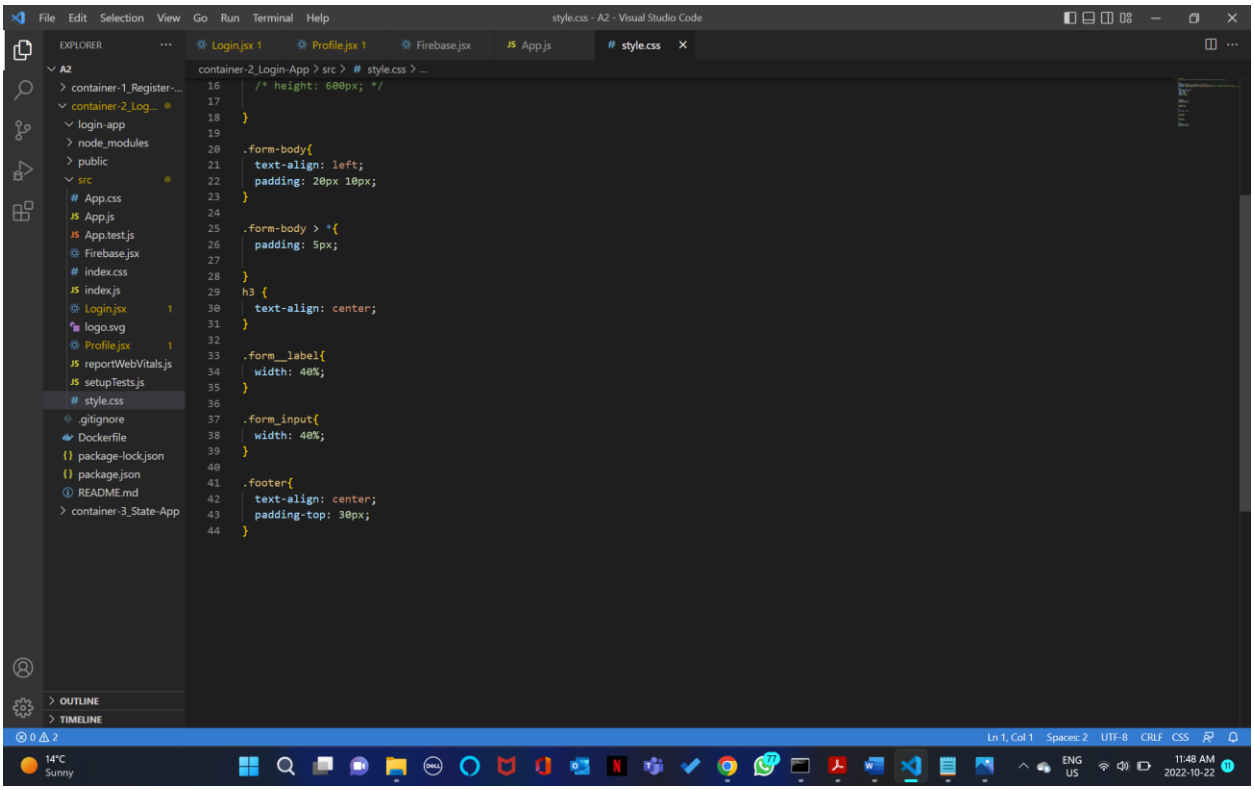


Figure 38: CSS Style code for Profile.jsx and Login.jsx

2. To display the content to the website we mention the class in the **App.js** file [1]. The file consists of the necessary imports like **LoginForm** and **Profile** from their absolute path i.e. "**./Login**" and "**./Profile**" respectively as shown in the Fig. 39. I added "router" to navigate between the login page and Profile page as shown in the Fig. 39.

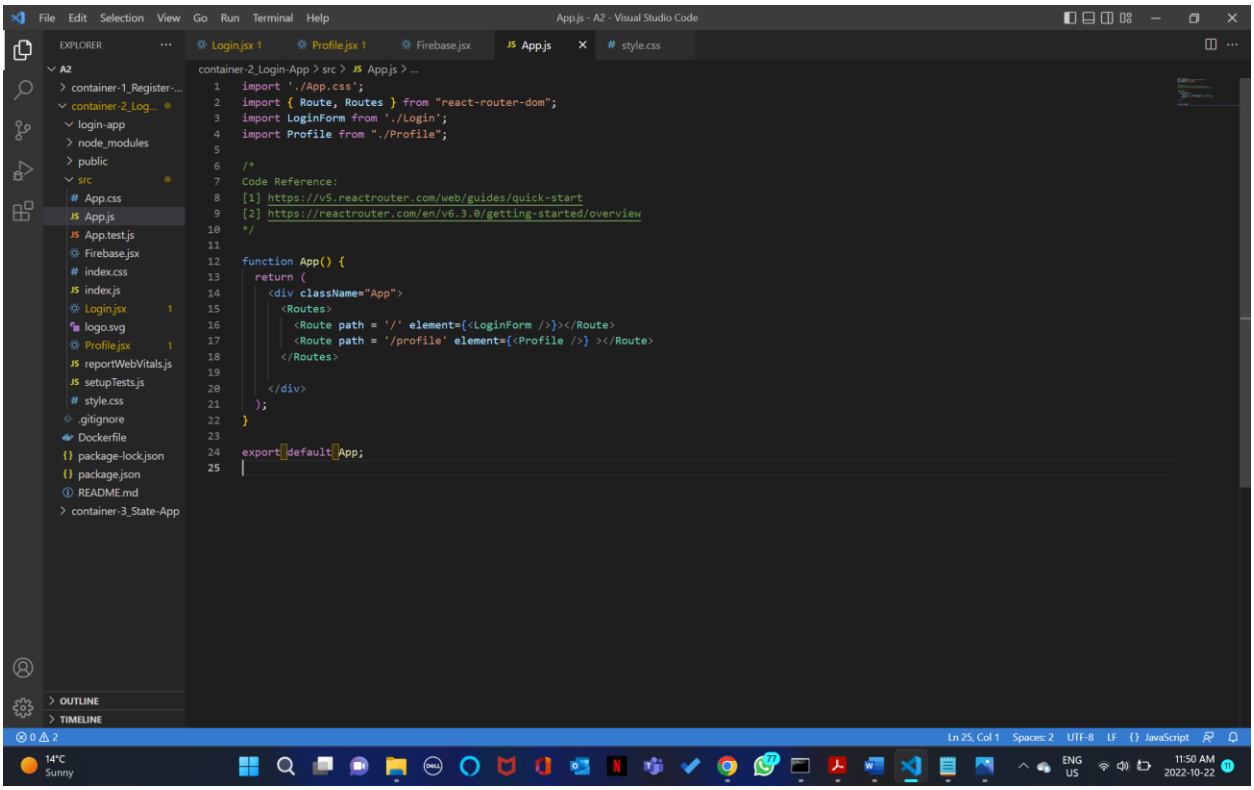


Figure 39: Necessary imports for Profile.jsx and Login.jsx

3. Login.jsx have 2 responsibilities:
    - It validates the user's login information and launch the user's Profile page.
    - It checks the state of the user (Whether user is online or offline) as shown in the **Step-D**.

4. I used handleSubmit function in which we'll get all the values that are filled in form [1] (Refer Fig. 40). This function is used to connect the app with the firebase [1]. I implemented all the test-cases like whether the password and email is valid or not (login credentials), whether the user entered empty string in any of the fields or not. This will prompt an error message like "Invalid login credentials", "Email required" etc (Refer Fig.).

    This test-cases will see in the following steps. If the user entered correct values then the user will login successfully (Refer Fig. 40). This condition is mentioned in the "else" part of the Fig. 41 which will also create a database named "States" in the firestore which will see in further steps.

```
const LoginForm = props => {

    const navigate = useNavigate();

    const [email, setEmail] = useState('')
    const [password, setPassword] = useState('')

    const handleSubmit = async e => {
        e.preventDefault();
        console.log('test   ')
        const db = database;
        if(email === ''){
            alert("Email id required")
            return
        }
        if(password === ''){
            alert("Password required")
            return
        }

        /* Code Reference:
        [2] https://react-query-firebase.invertase.dev/firestore/querying-collections
        [3] https://firebase.google.com/docs/firestore/query-data/queries
        [4] https://react-query-firebase.invertase.dev/firestore/querying-documents
        */

        const userCollection = collection(db, "Users")
        const userQuery = query(userCollection, where('email', '==', email), where('password', '==', password))
        const querySnapshot = await getDocs(userQuery);
        console.log('asd')

        if (querySnapshot.empty) {
            console.log('Invalid username or password')
            alert("Invalid login credentials")
            setEmail("");
            setPassword("");
```

Figure 40: If part of handleSubmit function for the Login.jsx

```
        } else {
            const docData = querySnapshot.docs[0].data();
            const docReference = query(collection(db, "State"), where('email', '==', email))
            await getDocs(docReference).then(async (snapshot) => {

                if (!snapshot.empty) {
                    const data = {
                        "timestamp": Timestamp.now(),
                        "state": "online"
                    }
                    const document = doc(db, "State", snapshot.docs[0].id)
                    await updateDoc(document, data)

                    localStorage.setItem("stateId", snapshot.docs[0].id)
                }
                else {
                    const docRef = await addDoc(collection(db, "State"), {
                        "email": email,
                        "name": docData.name,
                        "state": "online",
                        "timestamp": Timestamp.now()
                    })

                    localStorage.setItem("stateId", docRef.id)
                }
                navigate('/profile', { state: docData })
            });
        }
    };
```

Figure 41: Else part of handleSubmit function for the Login.jsx

5. I created the **Profile.jsx** which display the information when the user logs-in from the Login-page as shown in the Fig. 36.
6. To connect to the firebase, follow the **Step-B) 5)**. Copy paste the firebase connection code to the **Firebase.jsk** as shown in the Fig. 42.

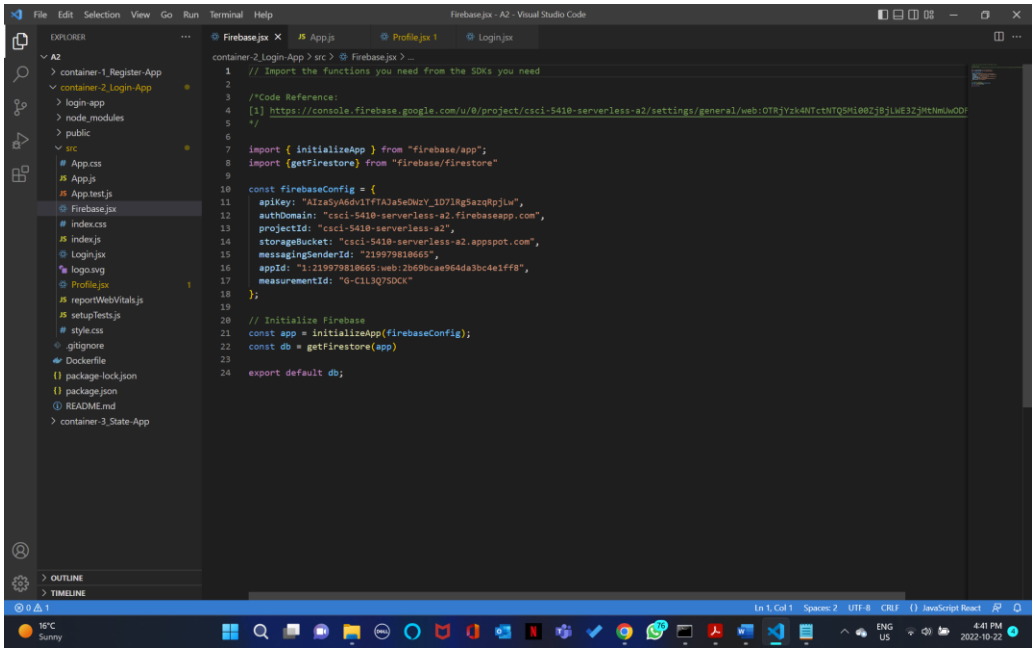


Figure 42: Firebase.jsk

7. Before running the command we see that the Firestore is empty, there is no database named “State ”as shown in the Fig. 43.

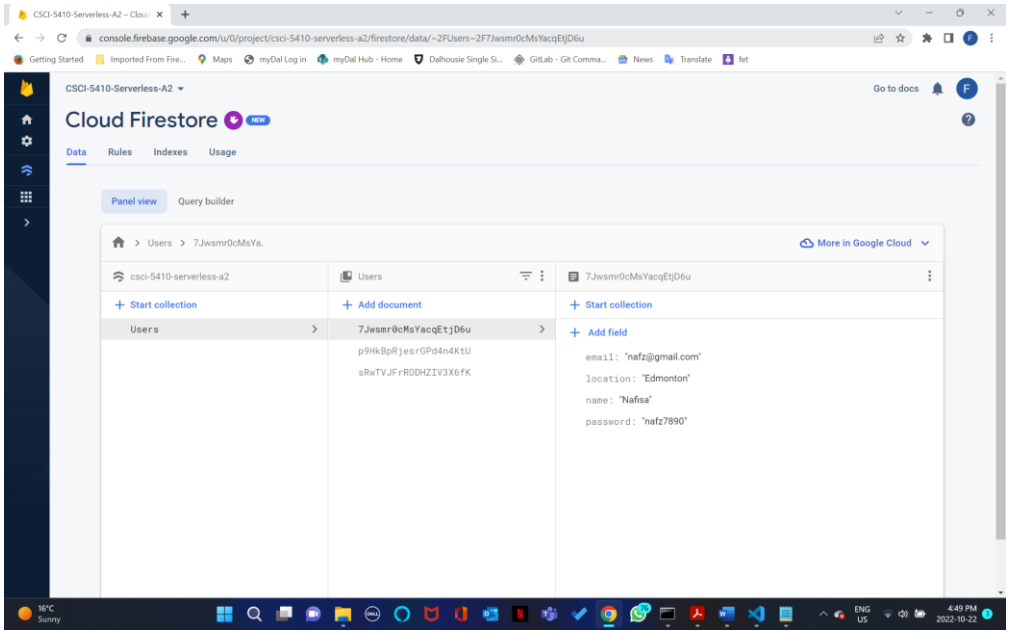


Figure 43: Cloud Firestore without the collection “State”

8. Now Run the command by providing the path of the container and write “npm start” to launch the login app as shown in the Fig. 44 & 45. The LoginForm is launched successfully which is shown in the Fig. 46 .

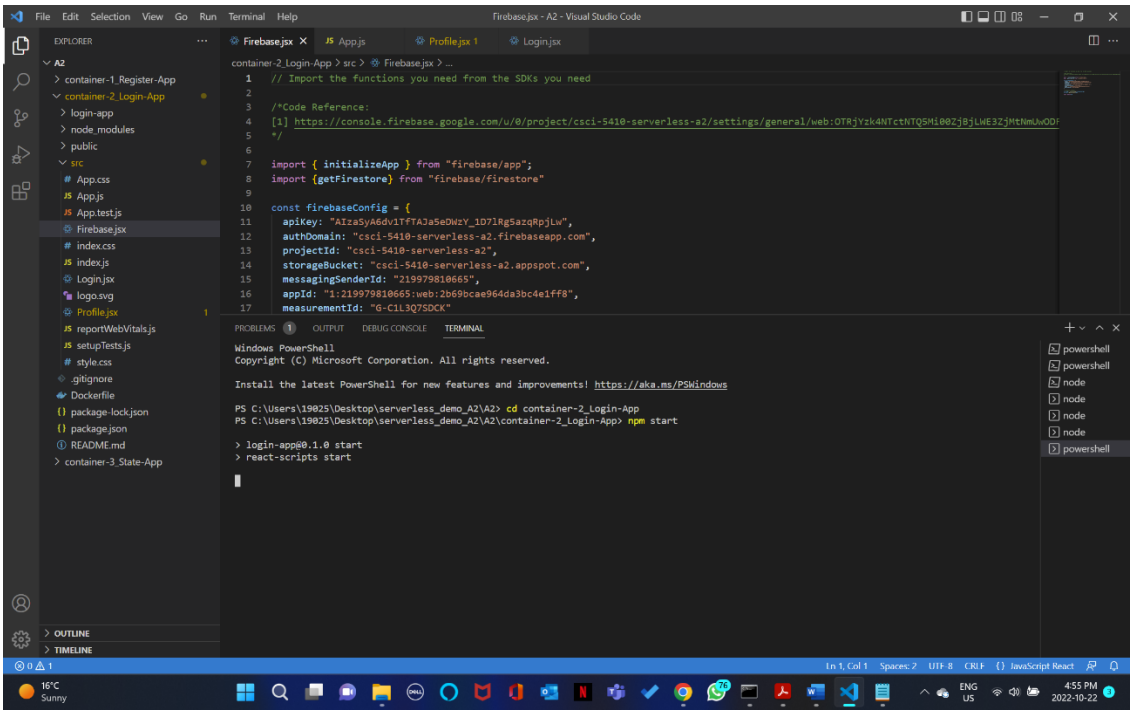


Figure 44: Launching the Login app by running “npm start” command

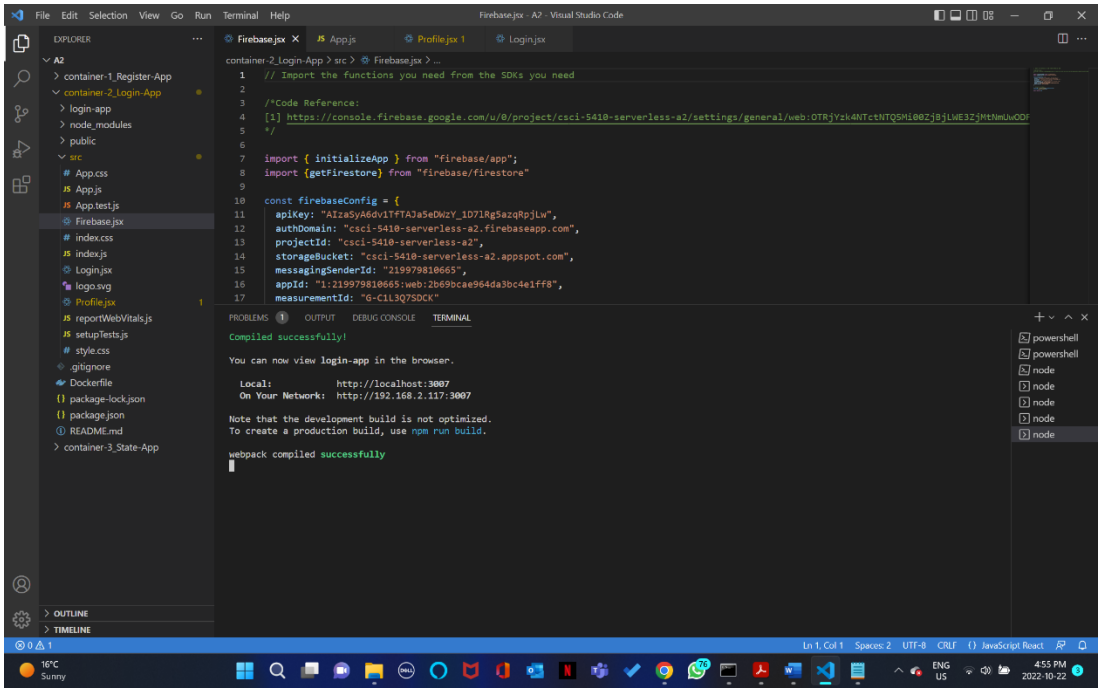


Figure 45: App launched successfully

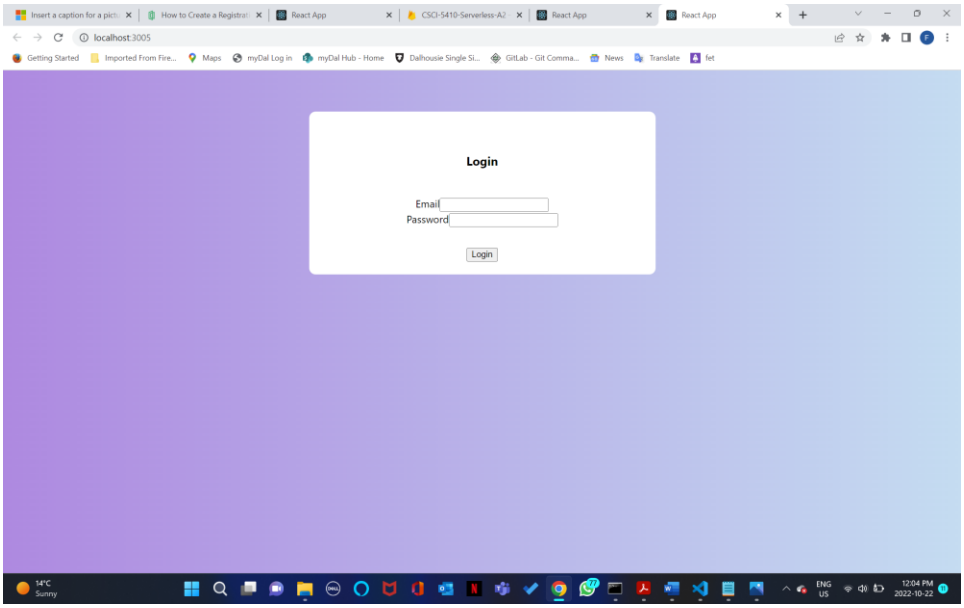


Figure 46 : Login Page

9. After launching the app, the user filled the details as shown in the Figures below.

I provided different Test Cases while Login the form which is shown below:

- **Test case 1:** When the user provides valid login credentials i.e the user is already registered means the user's registration details is mentioned in the cloud firestore (Refer Fig. 34 & Fig. 47). After the user has logged-in successfully, the user will be directed to the profile page which displays user's details (Refer Fig. 48).

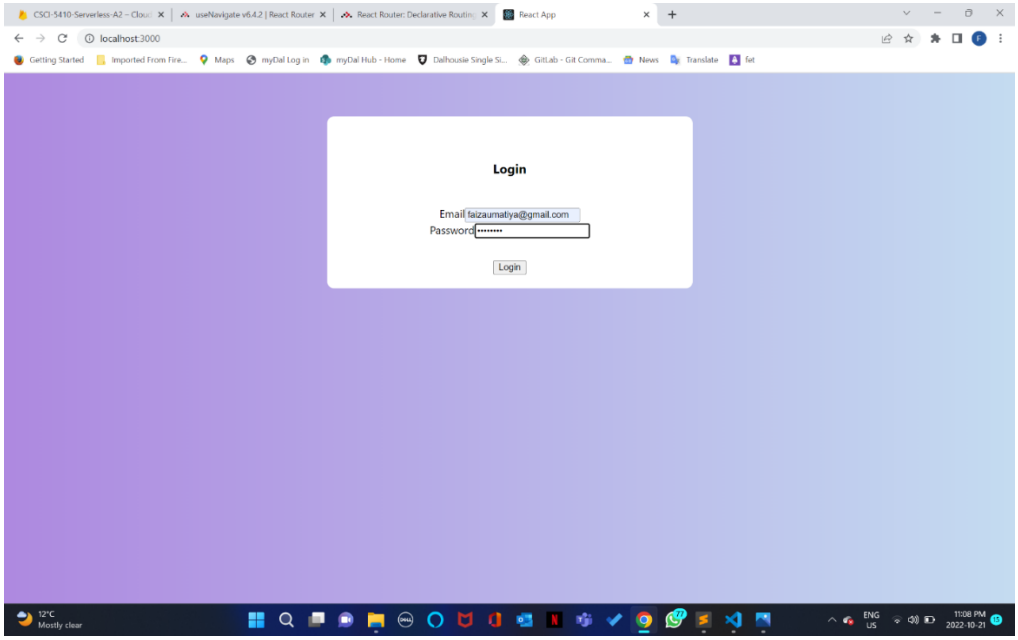


Figure 47 : Login Page

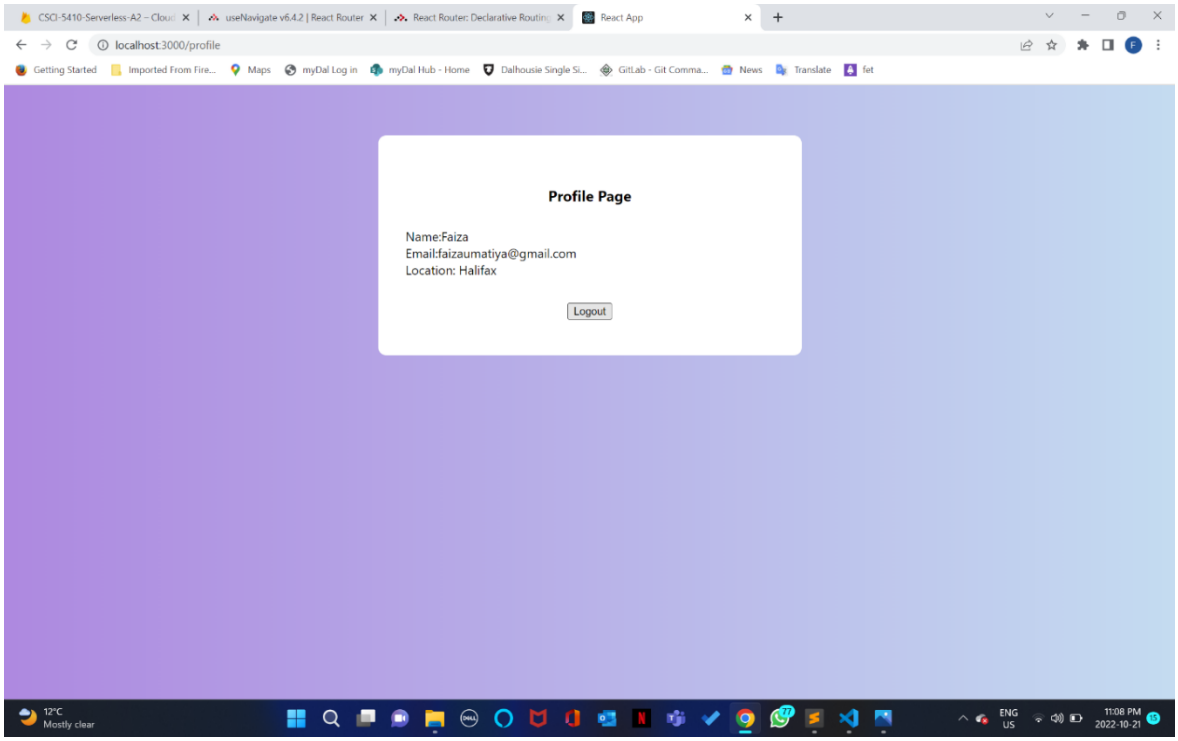


Figure 48 : Profile Page

- **<u>Test case 2:</u>** Fig 49. Prompts the alert message “Email id required” when user skipped entering the “email”. This will not allow user to login without entering the email id. Fig. 50. If the user skipped entering the “password”, an alert message will be prompted to the user “Password required”. When there is any error occurred while login, the fields gets empty (Refer Fig. 51)

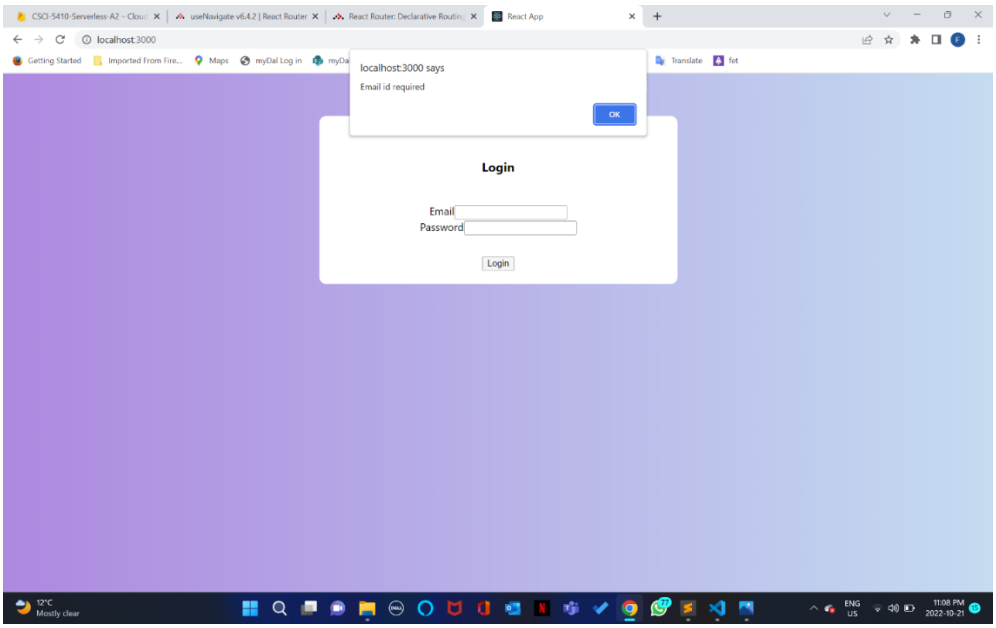


Figure 49 : Alert message – Email required

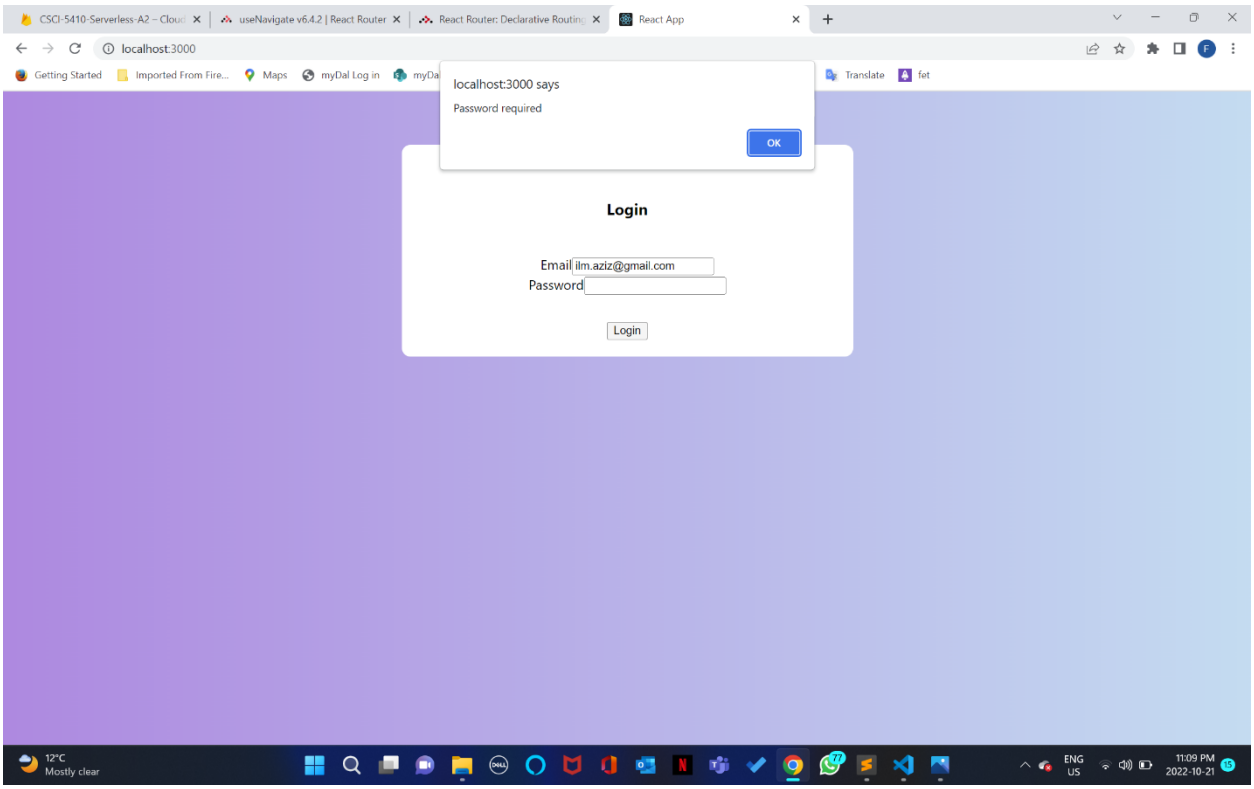


Figure 50 : Alert message – Password required

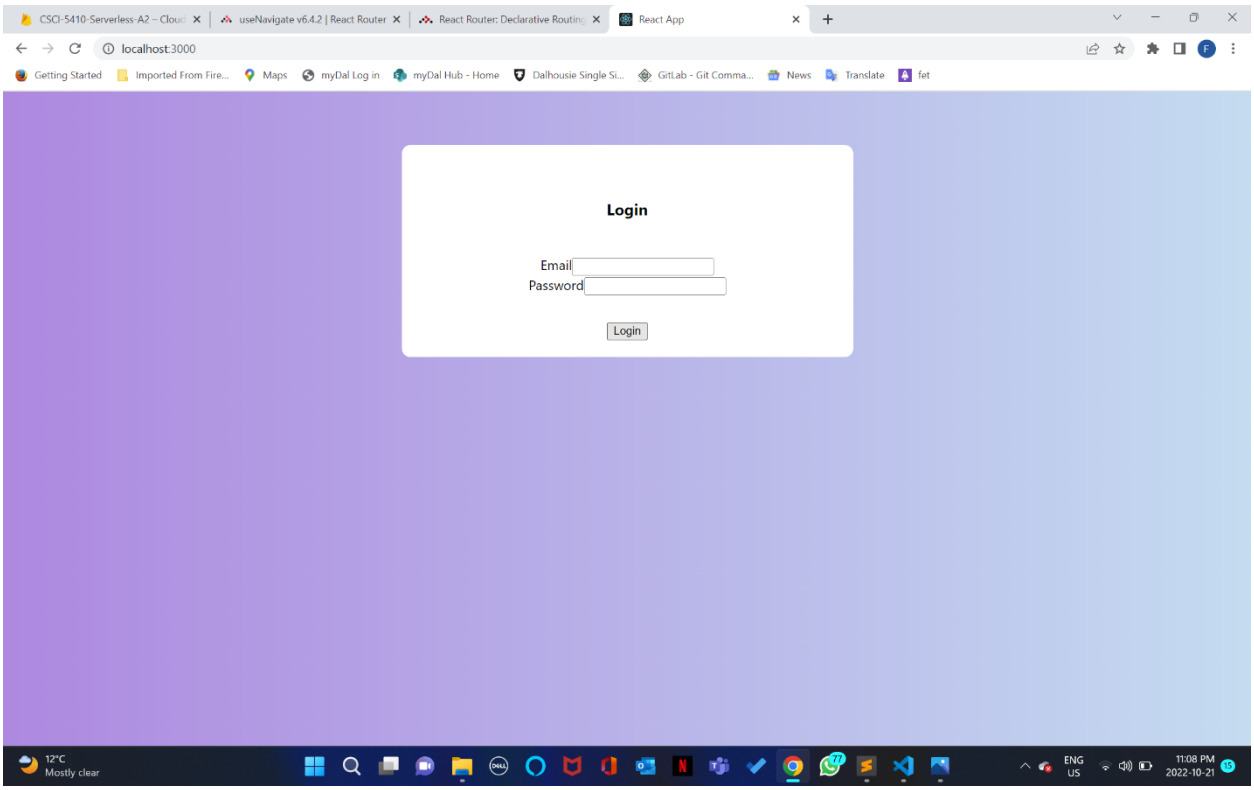


Figure 51: Login unsuccessful – Fields get empty

- **<u>Test case 3:</u>** Fig 52. Prompts the alert message “Invalid Login credentials” when user entered the credentials which are invalid or not present in the Cloud Firestore. This will not allow user to login without entering correct credentials. When there is any error occurred while login, the fields gets empty (Refer Fig. 53)

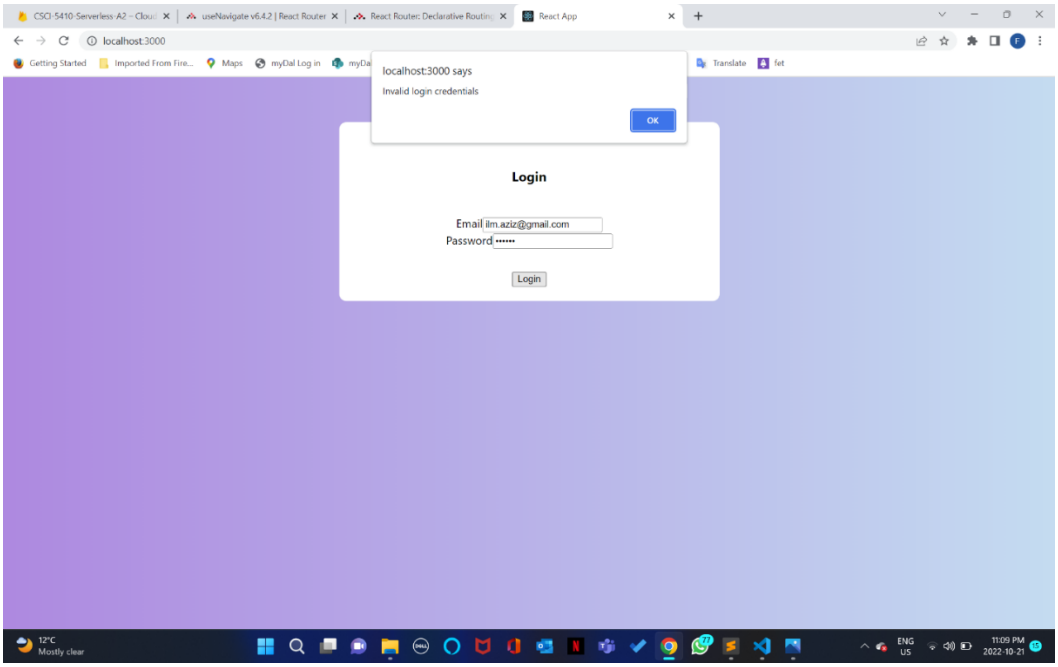


Figure 52: Alert message – Invalid Login credentials

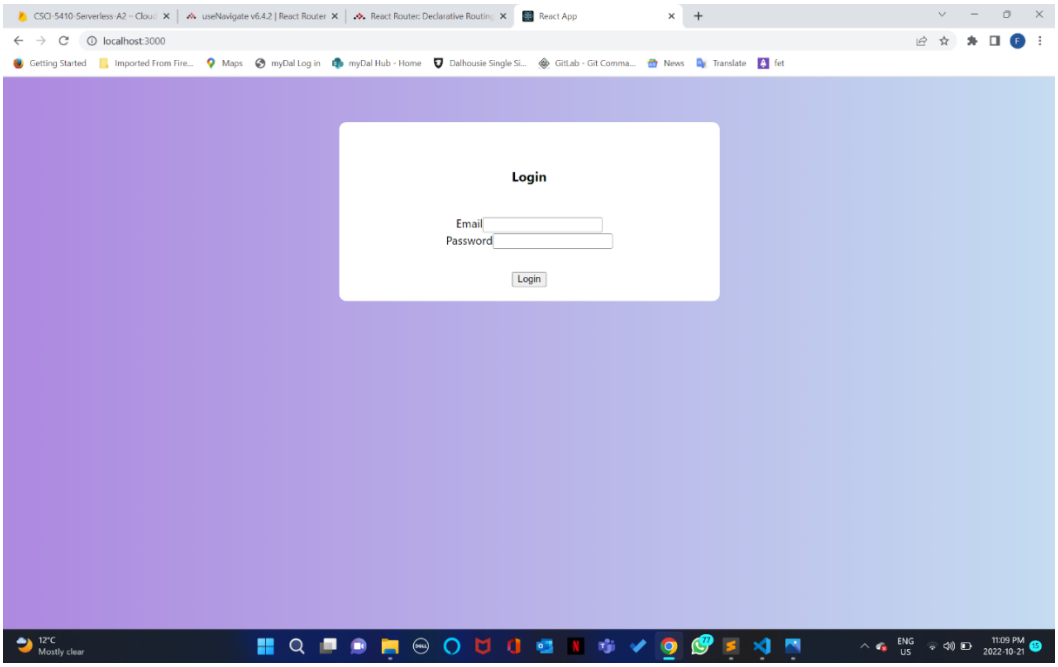


Figure 53: Login unsuccessful – Fields get empty

Step – D):

TO DO: Once a user is logged in- the state of the user changes to online in the Firestore database.

PROCEDURE:

I created Login form inside the "**container-2_Login-App**". The folder consists of 5 files namely:

1. Login.jsx
2. Profile.jsx
3. Firebase.jsx
4. App.js
5. Style.css

Refer **Step – C)** we saw how user logs in and gets directed to the Profile page. In this Step we'll check the status of the user in the firestore.

1. To check the status of the user, Refer Fig. 41 to see the else part of the handleSubmit function that was present in the "Login.jsx" file.
2. In this handleSubmit function, I used a query to refer to the database, if the user is logged in then the user details will be checked in the database and the status will be changed to "online". This update is done in the new collection that's created called "state" (Refer Fig. 54)
3. First let the user login (Refer Fig. 47)
4. The "state" database is created which shows which user is online or offline. The person logged in. (Refer Fig. 54)

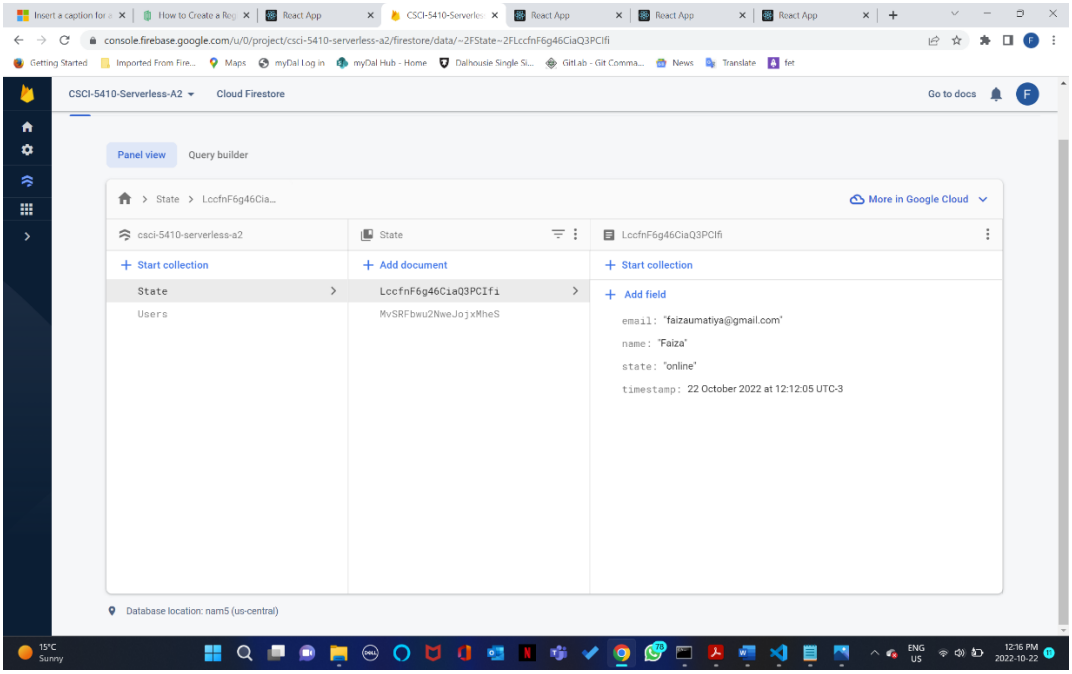


Figure 54: State collection created

```
function Profile(props){

    const location = useLocation()
    const navigate = useNavigate()

    const logout = async(e) => {
        e.preventDefault()

        console.log(localStorage.getItem("stateId"))
        const docRef = await updateDoc(doc(db, "State", localStorage.getItem("stateId")), {
            "state": "offline"
        })

        navigate("/")

    }
```

Figure 55: User State changes to offline on logging out

Step – E):

TO DO: Your database should contain only 2 documents. One document to contain registration data (Name, Password, Email, Location), another document to contain user state (online, offline, timestamp etc.) information with the name and email of the users.

PROCEDURE:

1. The database contains 2 collection “states” and “users” as shown in the Fig. 54.
2. To check the timestamp I created “container-3_State-App” that displays the timestamp of the user’s status.
3. User logs in by entering the login credentials as shown in the Fig. 47.
4. User enters to the Profile page which means the user is online (Refer Fig. 56).
5. The Profile.jsx allows navigation when user logouts. It redirects user to the login page (Refer Fig. 55).
6. To check the timestamp and session go to Step – G).

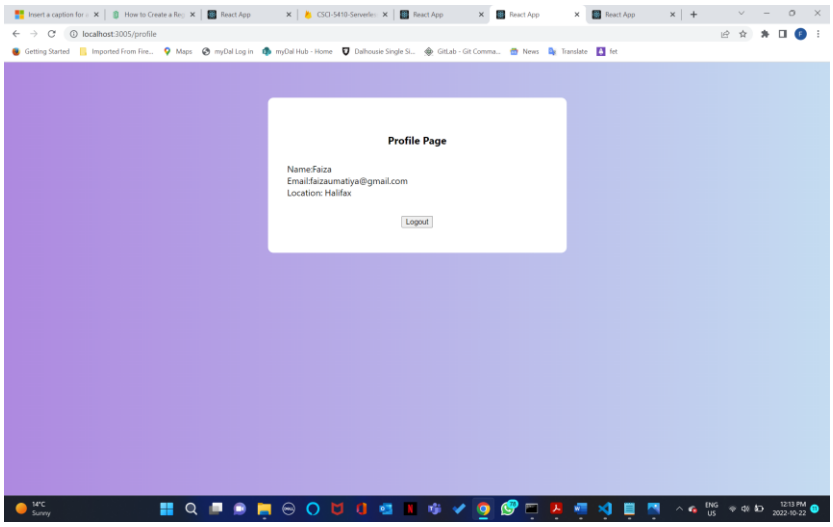


Figure 56: User logged in

Step – F & G):

TO DO: Once the user signs out from the profile page, it load code and dependencies of Container #3.

Code and required dependencies in Container #3 are responsible for extracting state information from database. Eg. Who was online at what time. You need to maintain the session from login to logout. The session must expire after clicking the logout, which should update the state in the Firestore database.

PROCEDURE:

I created the container-3_State-App to check the timestamp of the user when they login and logout. I created the logic in App.js that displays the table of the session of the user (Refer Fig. 57). The table is displayed using the front-end which is shown in the Fig. 58.

```
function App() {

  //[1] https://www.pluralsight.com/guides/consume-data-from-firebase-firestore-in-a-react-app
  const [states, setStates] = useState([])

  const getData = async () => {
    const q = query(collection(db, "State"), where("state", "==", "online"))

    return await getDocs(q).then((snapshot) => {
      let data = []

      snapshot.docs.forEach((d) => {
        data.push(d.data())
      })

      console.log(data)
      return data
    });


  }

  //[2] https://reactjs.org/docs/hooks-effect.html
  useEffect(() => {
    getData().then((response) => {
      setStates(response)
    })
  }, [])
```

Figure 57: App.js

```
//[3] https://www.geeksforgeeks.org/how-to-create-a-table-in-reactjs/
return (
  <div className="App">
    <table className="table" border="2px">
      <thead>
        <tr>
          <th>Name</th>
          <th>Email</th>
          <th>State</th>
          <th>Timestamp</th>
        </tr>
      </thead>
      <tbody>
        {states ? states.map((data) => {
          return (
            <tr>
              <th>{data["name"]}</th>
              <th>{data["email"]}</th>
              <th>{data["state"]}</th>
              {/* https://stackoverflow.com/questions/48689876/how-to-convert-timestamp-in-react-js */}
              {/* <th>{new Intl.DateTimeFormat('en-US', {year: 'numeric', month: '2-digit',day: '2-digit', hour: '2-digit', minute: '2-digit', second: '2-digit'}).fo
              <th>{new Date(data["timestamp"].seconds * 1000).toLocaleString()}</th>
            </tr>
          )
        }) : <tr></tr>}
      </tbody>
    </table>

  </div>
);
}

export default App;
```

Figure 58: Front-end part of App.js

We connected firebase and run the application as shown in the above steps.

- Before the user logs in, the timestamp is null which means the user is offline (Refer Fig. 59).
- User logs in which means user is online (Refer Fig. 60)
- When we ran the container-3_State-App, the timestamp showed as user online (Refer Fig. 61).
- User logs out and redirects to the login page as shown in the Fig. 62. That means user is offline.
- The timestamp is null now because the useris offline (Refer Fig. 63).

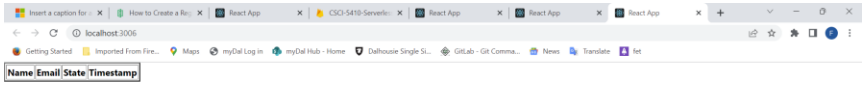


Figure 59: User not logged in yet

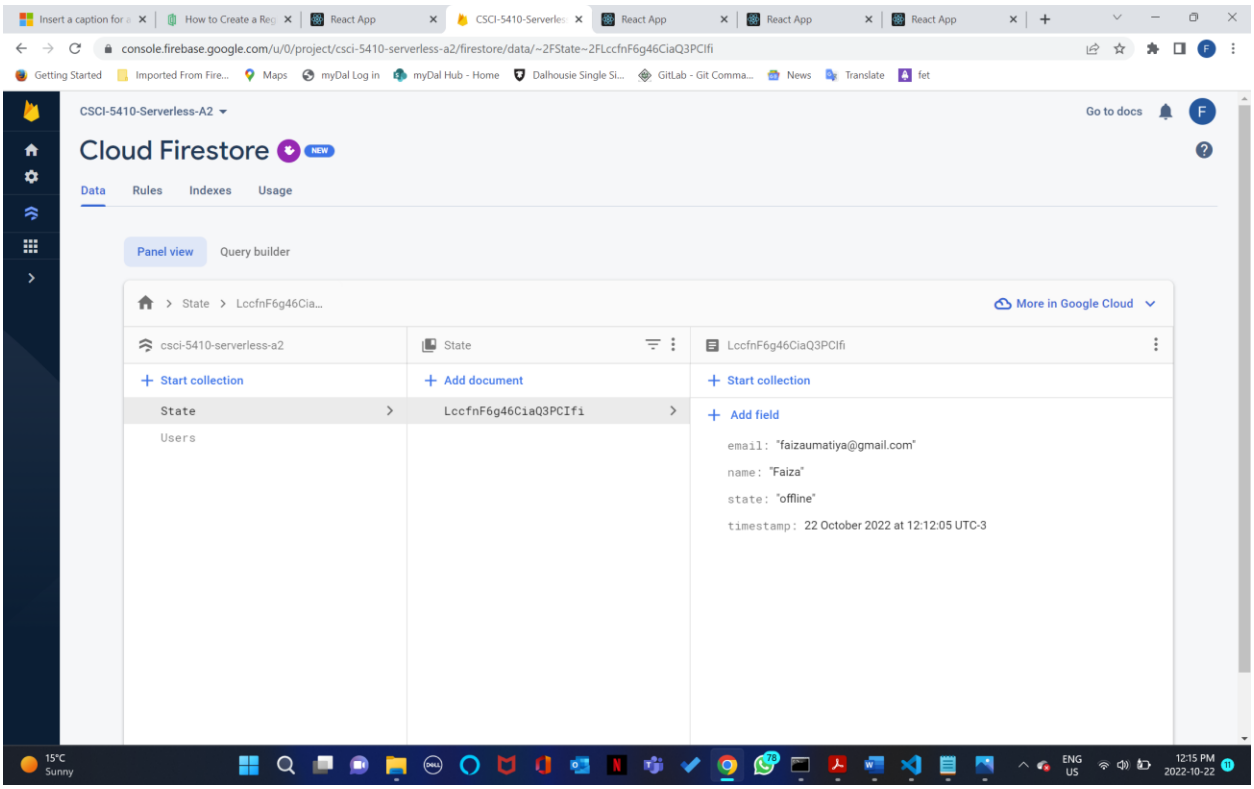


Figure 64: User is offline in Cloud Firestore

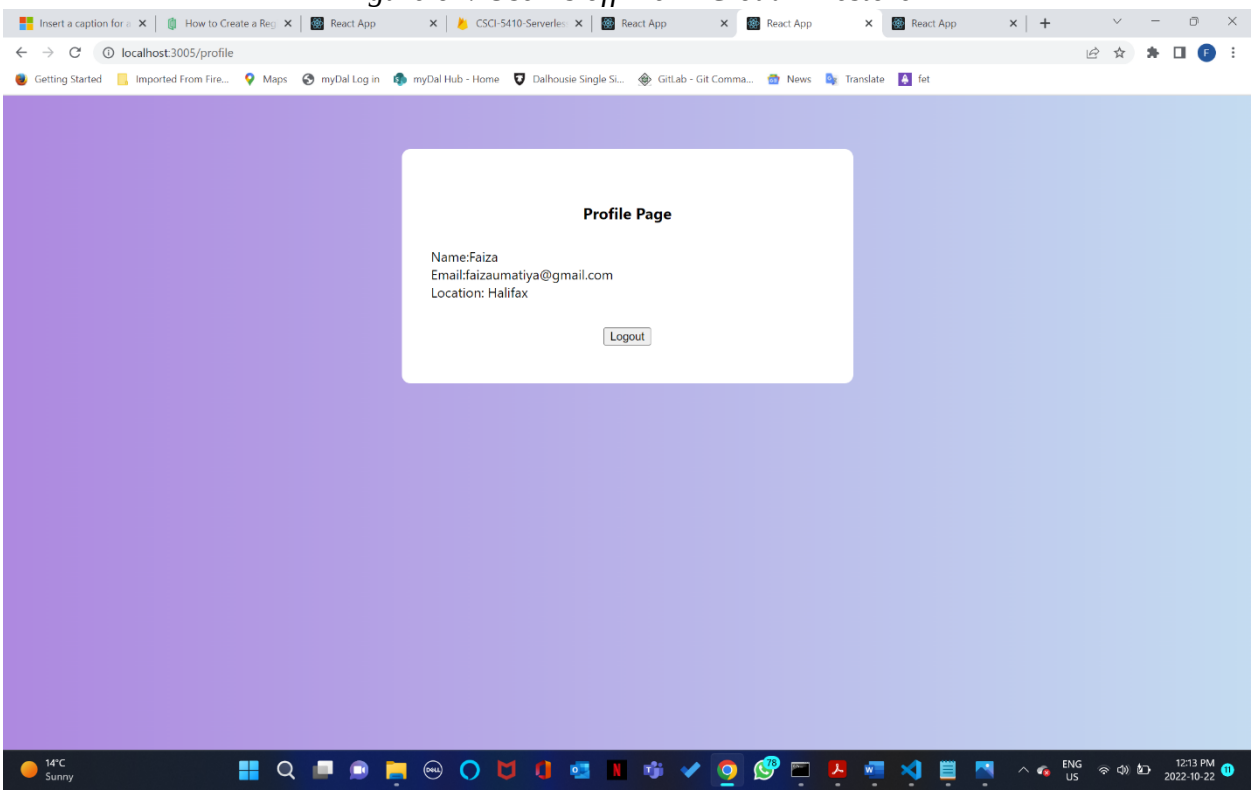


Figure 60: User not logged in

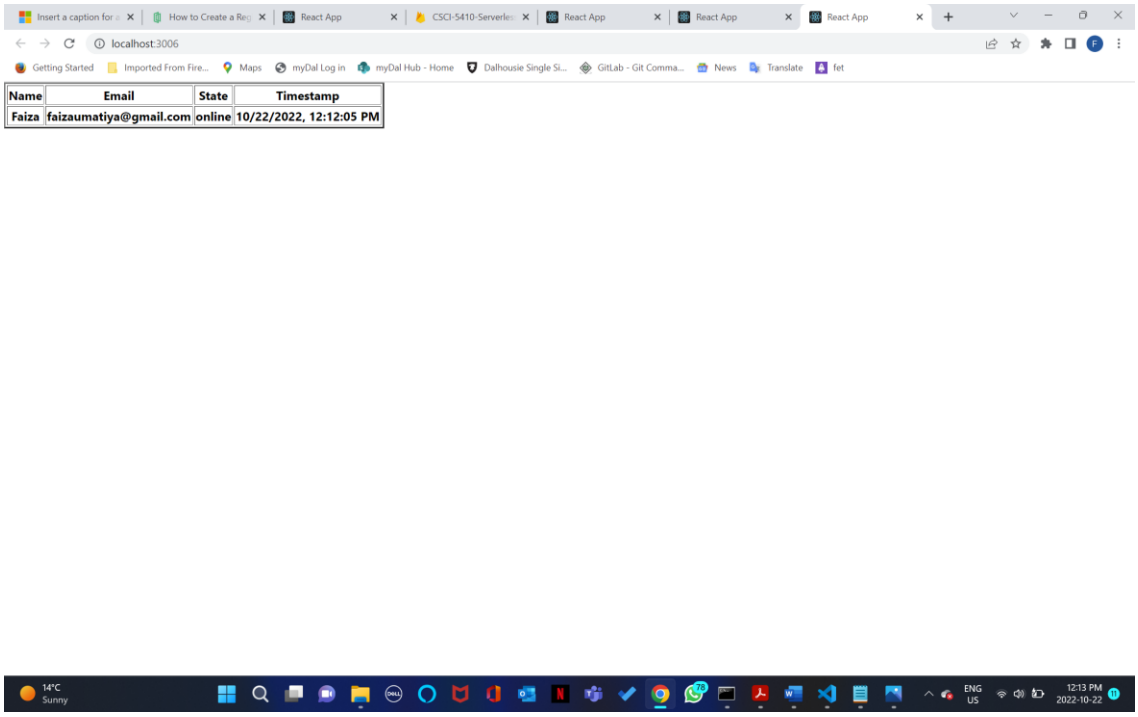


Figure 61: User status "online" with timestamp

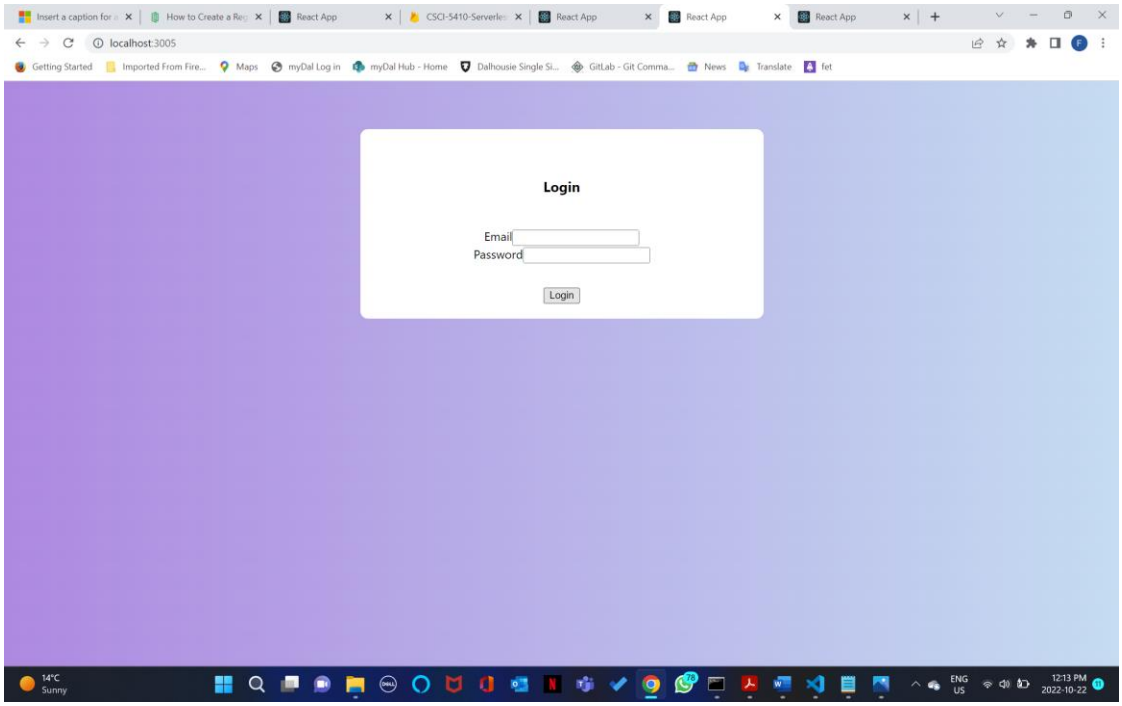


Figure 62: User logs out and redirects to login page

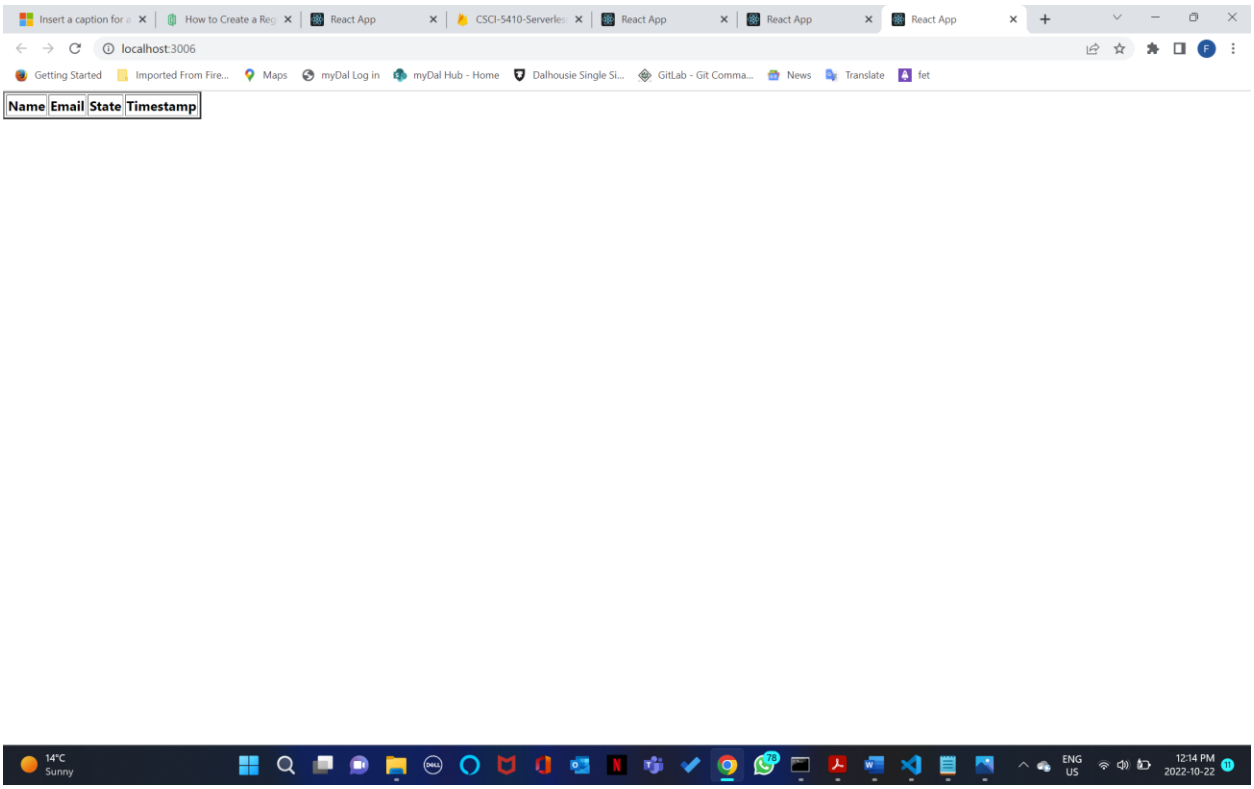


Figure 63: User timestamp empty (Status is offline)

Step-H)

DOCKER SETUP:

TO DO:

After creating the 3 microservices which will come from the Firestore as 3 docker images. We need to then push those container images to artifact registry repository (GCR is deprecated now). Once it is done, we need to deploy those in Cloud Run.

STEPS I FOLLOWED TO DEPLOY:

1. Installed Docker Desktop.
2. Installed Google CLI.
3. Open GCP Console and created new VM instance (Refer Fig. 64).
4. Go to artifact registry as shown in the Fig. 65.
5. Enable Artifact Registry API as shown in the Fig. 66.
6. Create Repository as shown in the Fig. 67, 68, 69 & 70.
7. Repository "**serverless-a2**" created successfully as shown in the Fig. 71.
8. Authorize google cloud with the service account as shown in the Fig 72, 73, 74, 75 & 76.
9. Now push the container images to the artifact registry by running the command as follows:
    - Docker build -t container-1
    - Docker run -p -d 3000:3000 container-1
    - Docker tag container-1: latest us-central1-docker.pkg.dev/cdi-serverless-a2/container-1:latest
    - Docker push us-central1-docker.pkg.dev/cdi-serverless-a2/container-1:latest
10. Similarly this commands are run for container-2 and 3 respectively (Refer 77 to see the docker container-2 push).
    Note: Port 3000, 3001 and 3002 is taken respectively for 3 containers.
11. After pushing the the docker container, we need to create new service as shown in the Fig. 78, 79, 80, 81, 82, 83 & 84.
12. Follow the same steps for other 2 containers too.
13. We'll see all 3 containers pushed in the artifact registry as shown in the Fig 85.
14. Lastly we see that the docker images are formed in the docker desktop as shown in the Fig. 86.

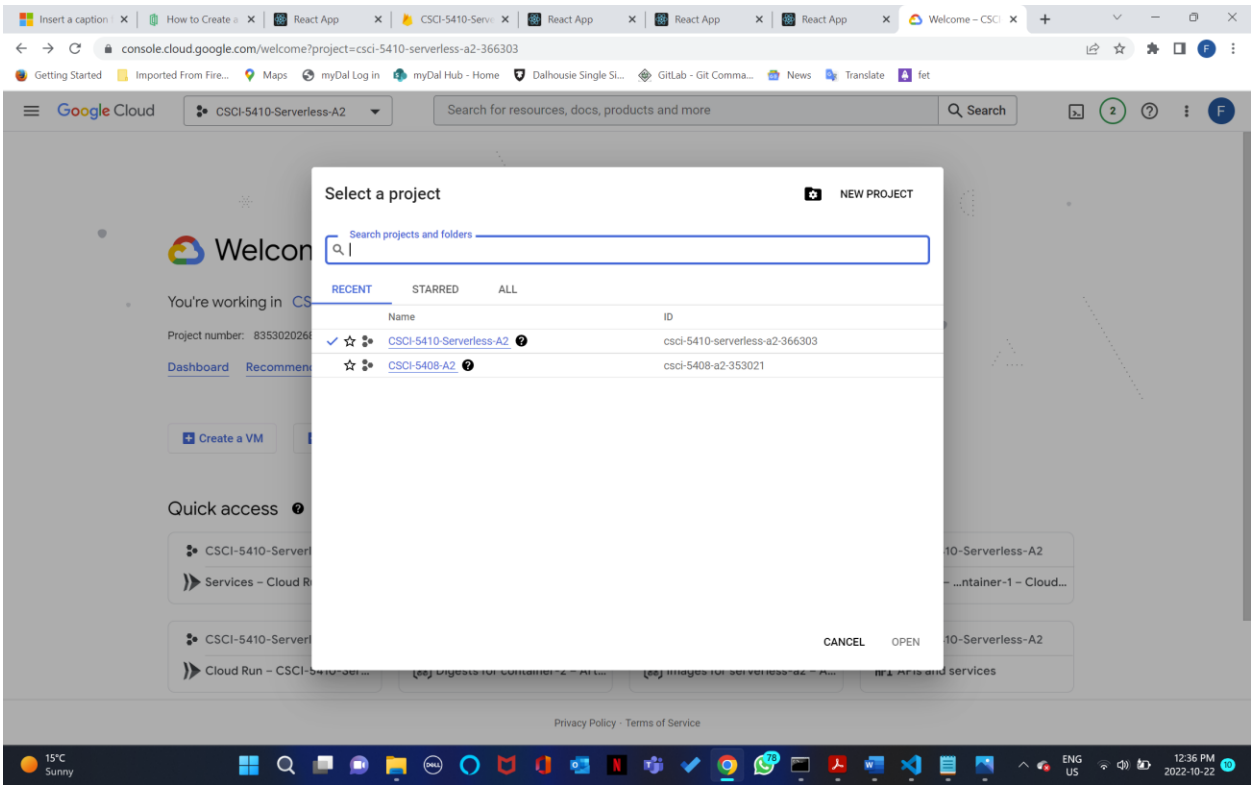


Figure 64: GCP Console – VM instance

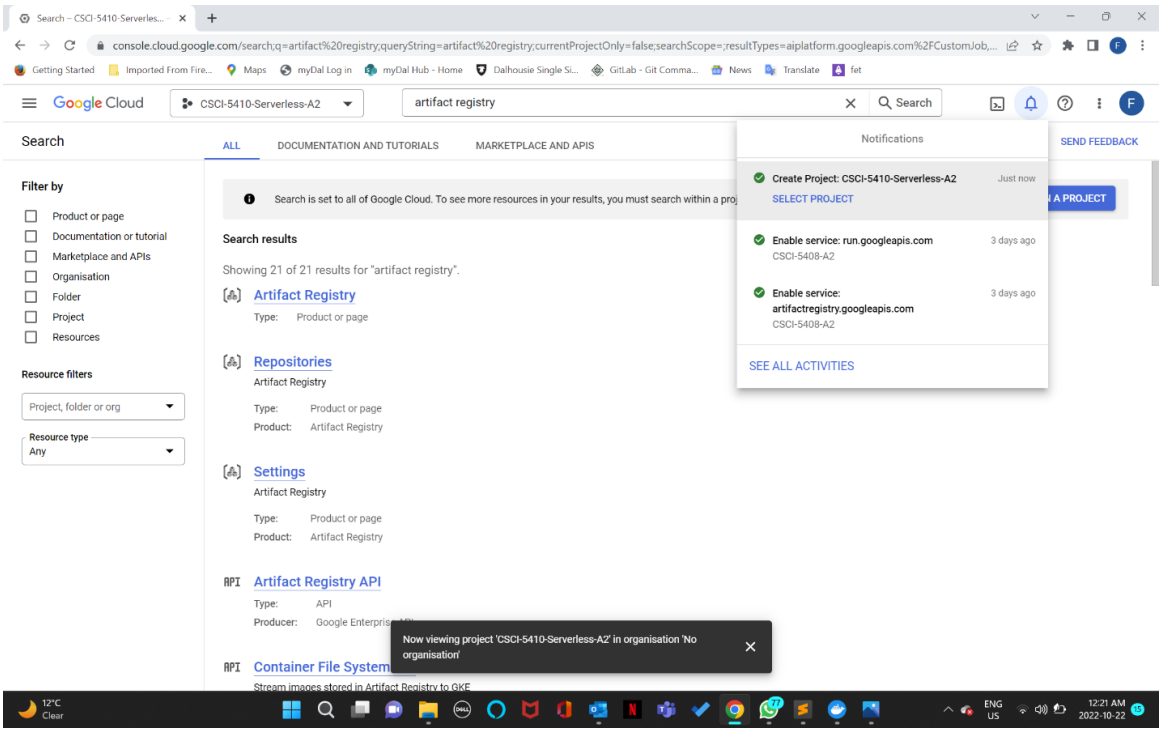


Figure 65: GCP Console – Artifact Registry

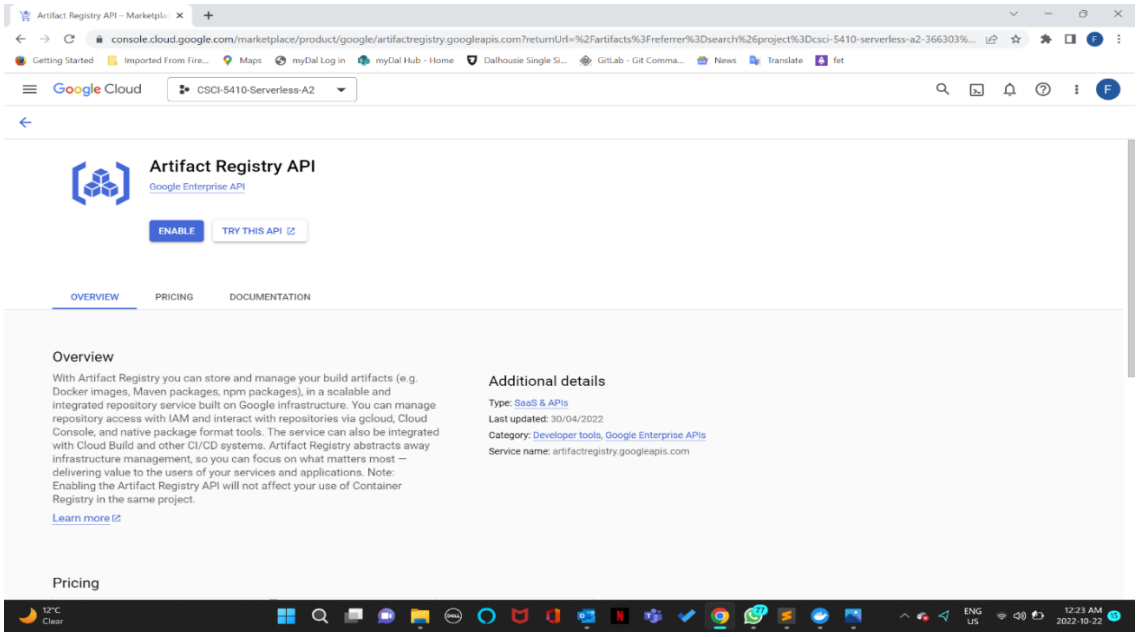


Figure 66: Artifact Registry – Enable API

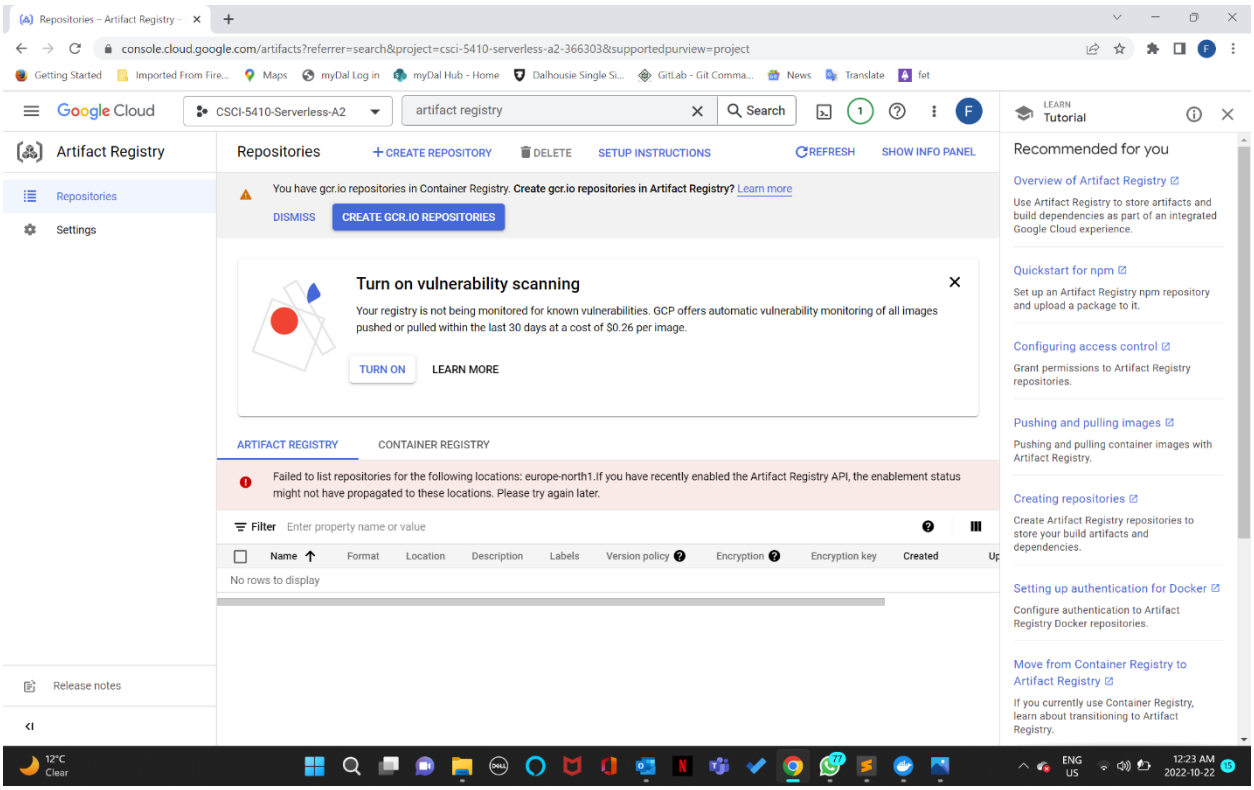


Figure 67: Create Repository

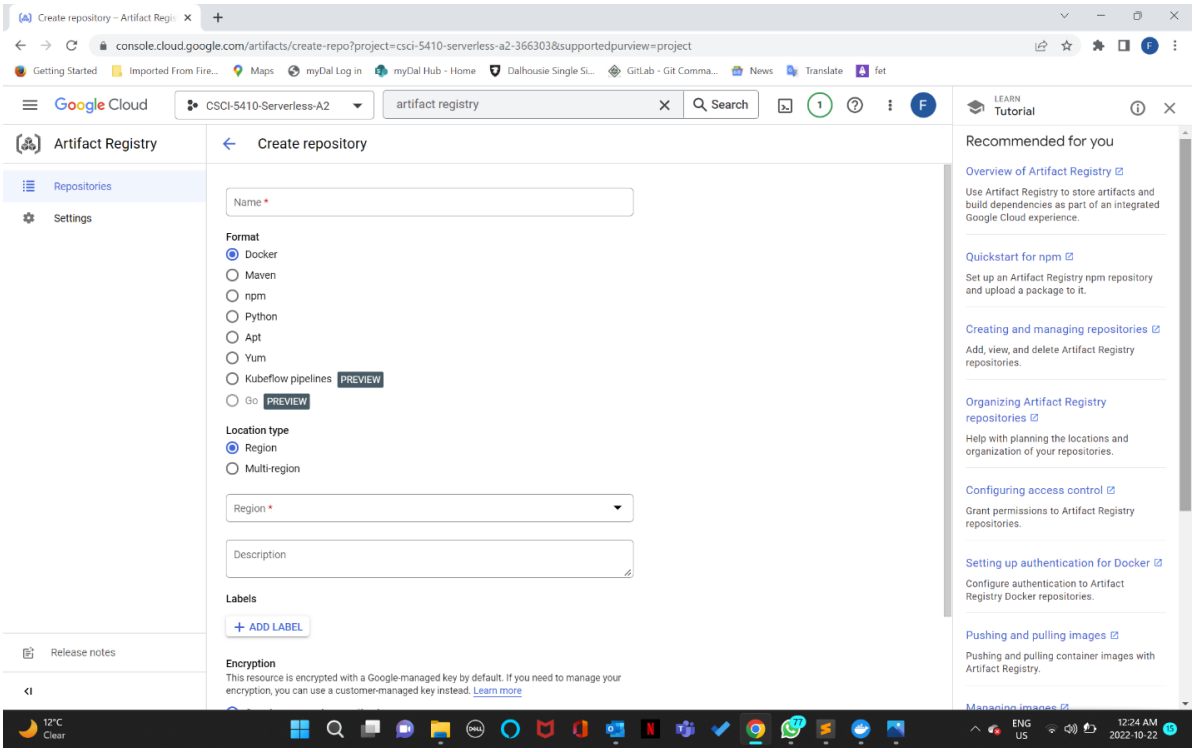


Figure 68: Create Repository – mark the fields

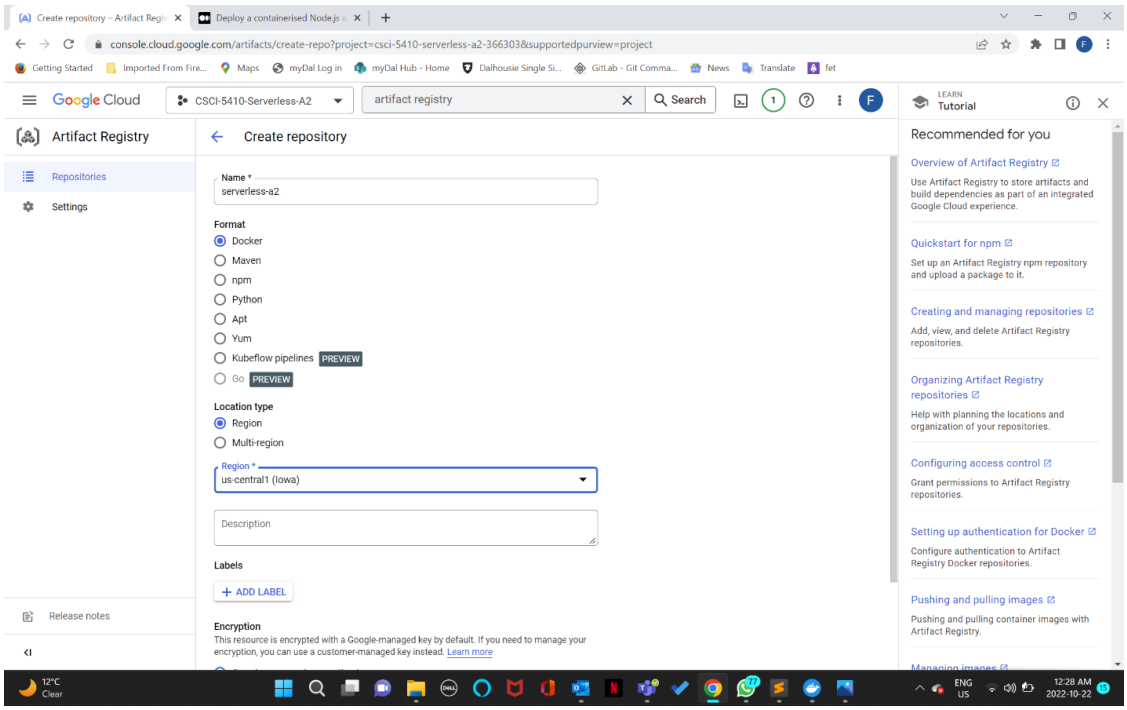


Figure 69: Create Repository – mark the fields

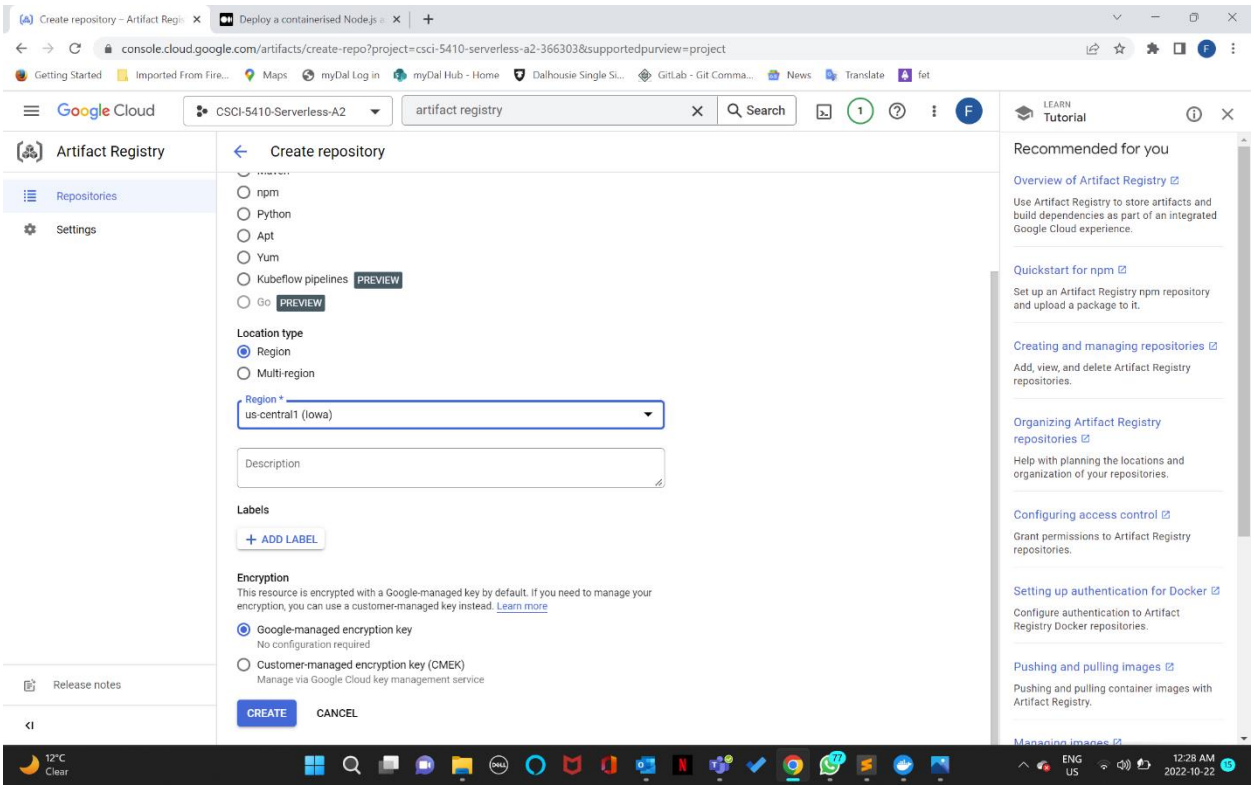


Figure 70: Create Repository – mark the fields

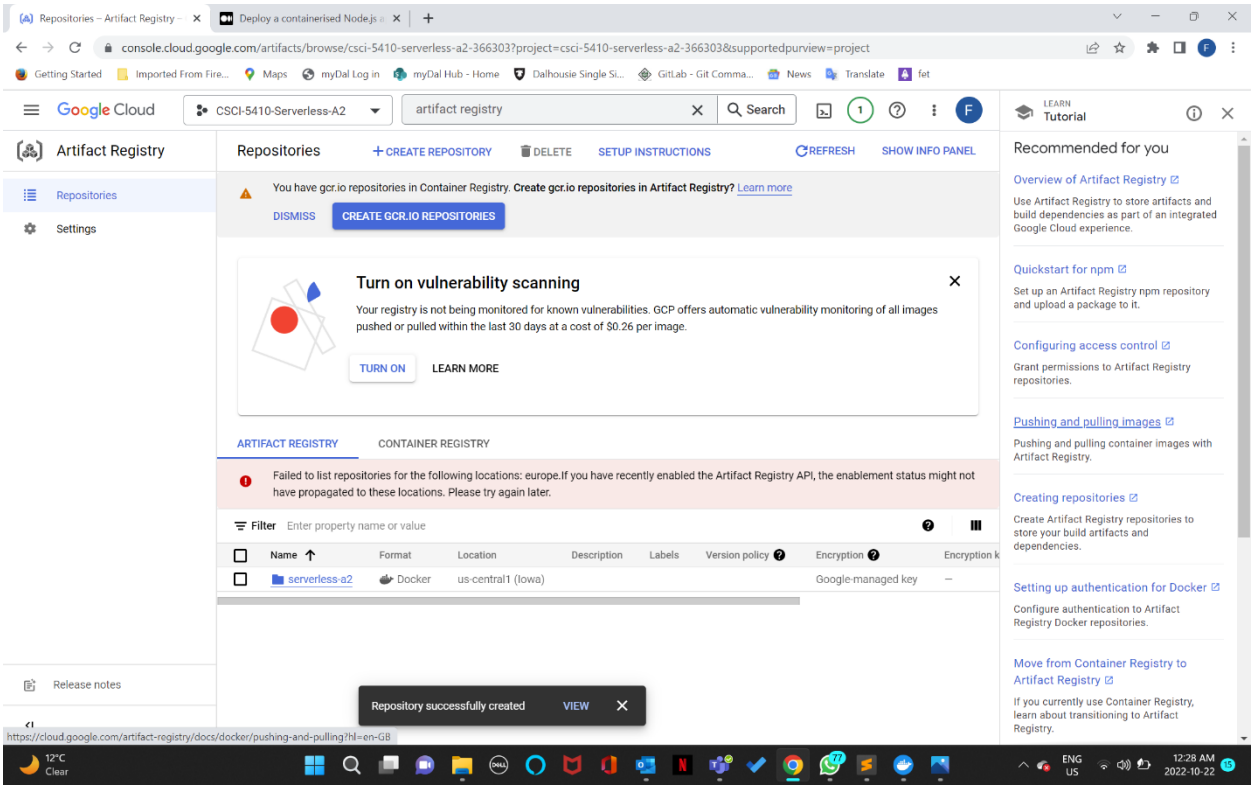


Figure 71: Repository created

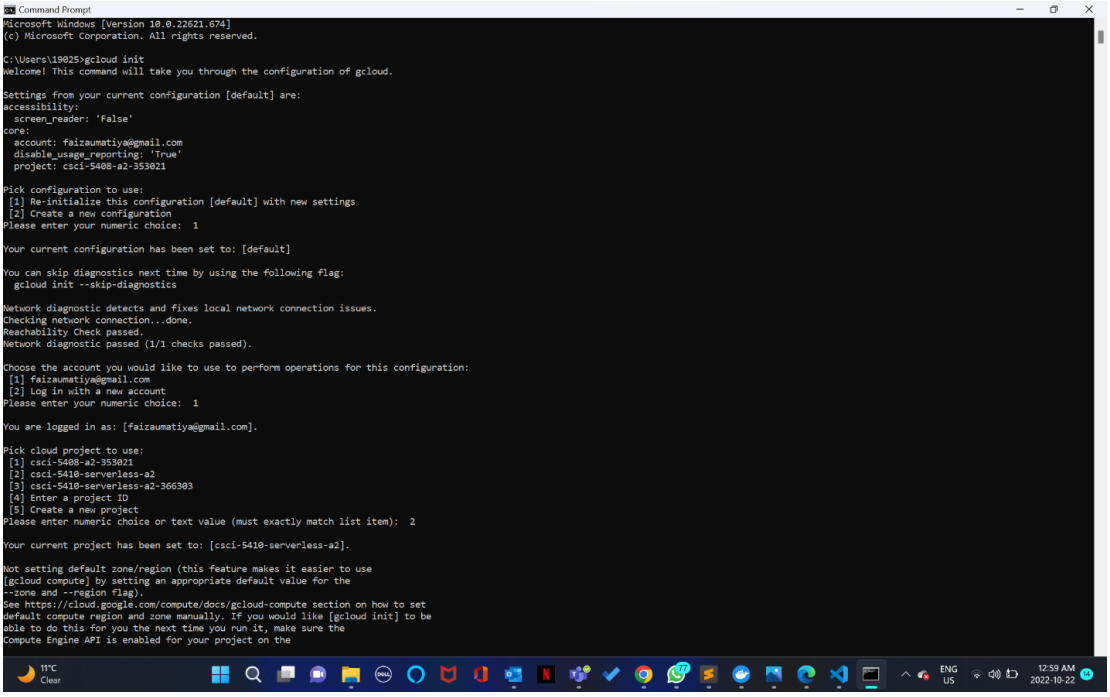


Figure 72: Google Cloud Authorization

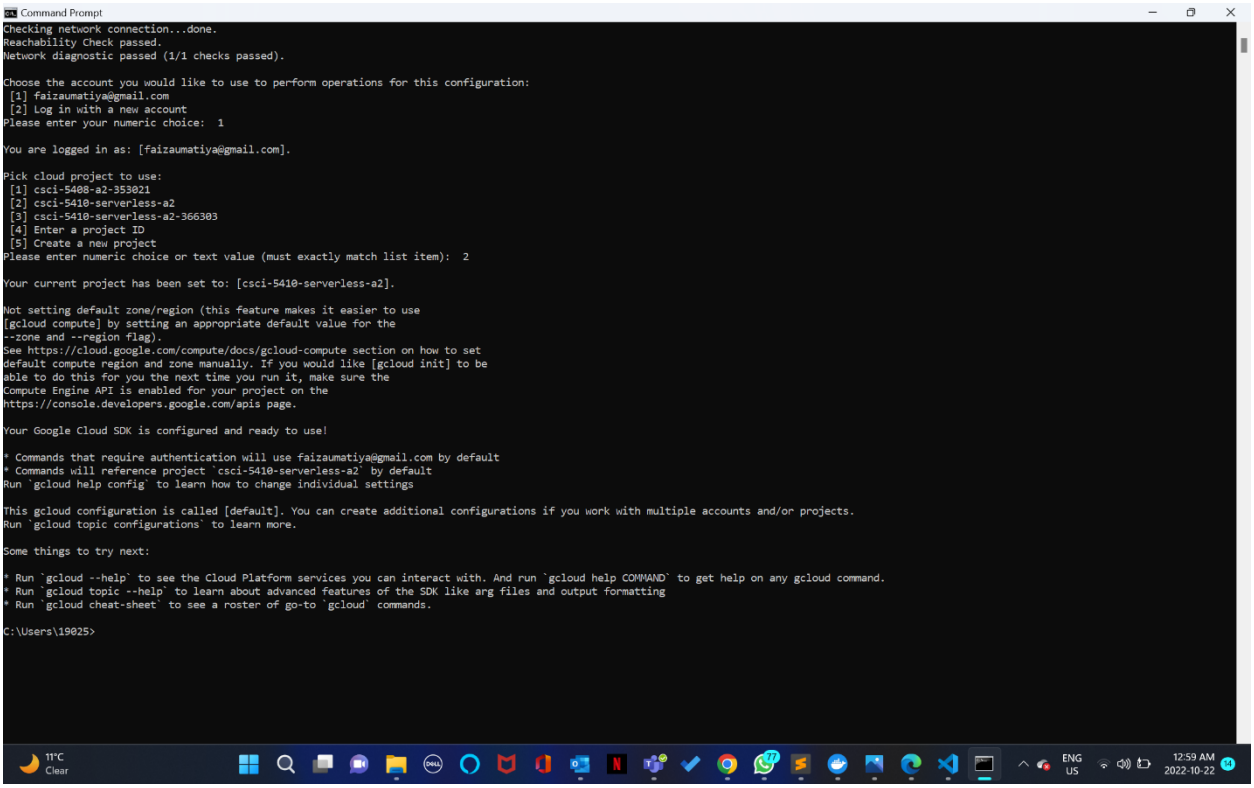


Figure 73: Google Cloud Authorization

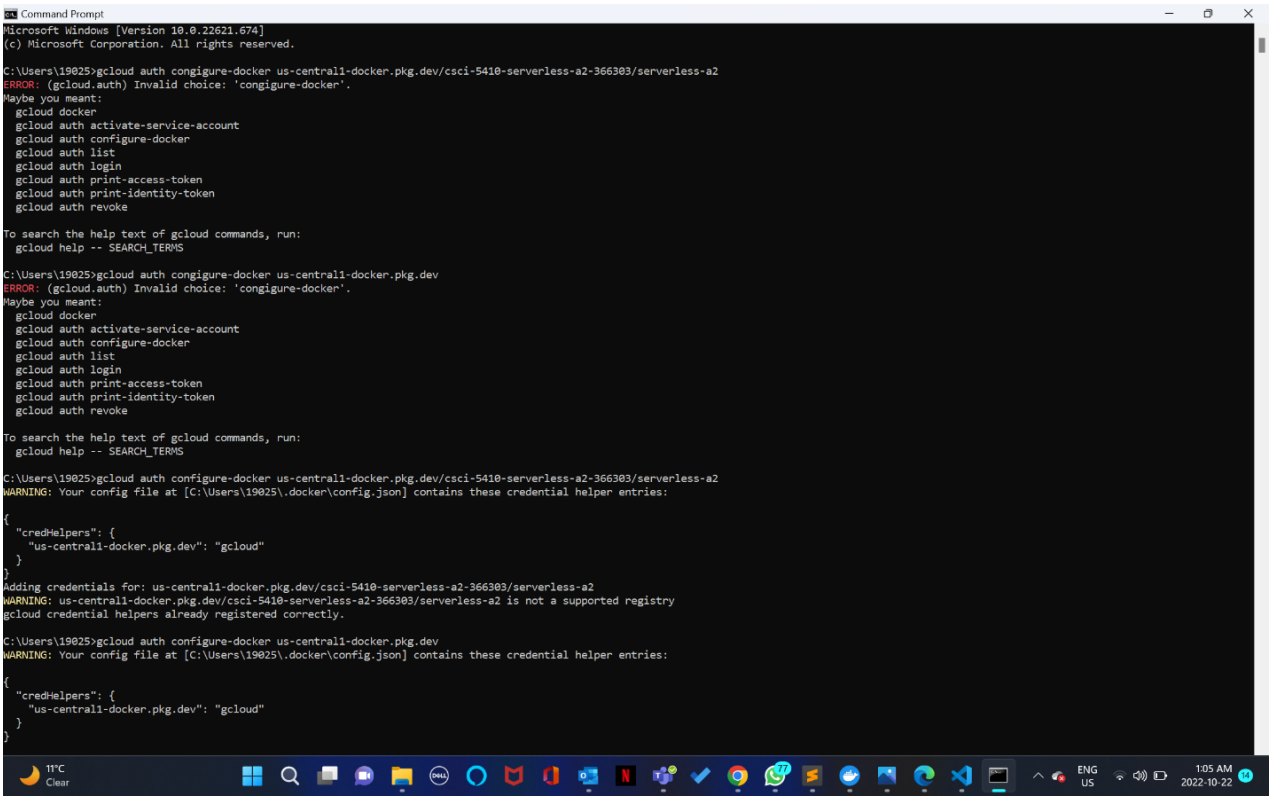


Figure 74: Google Cloud Authorization

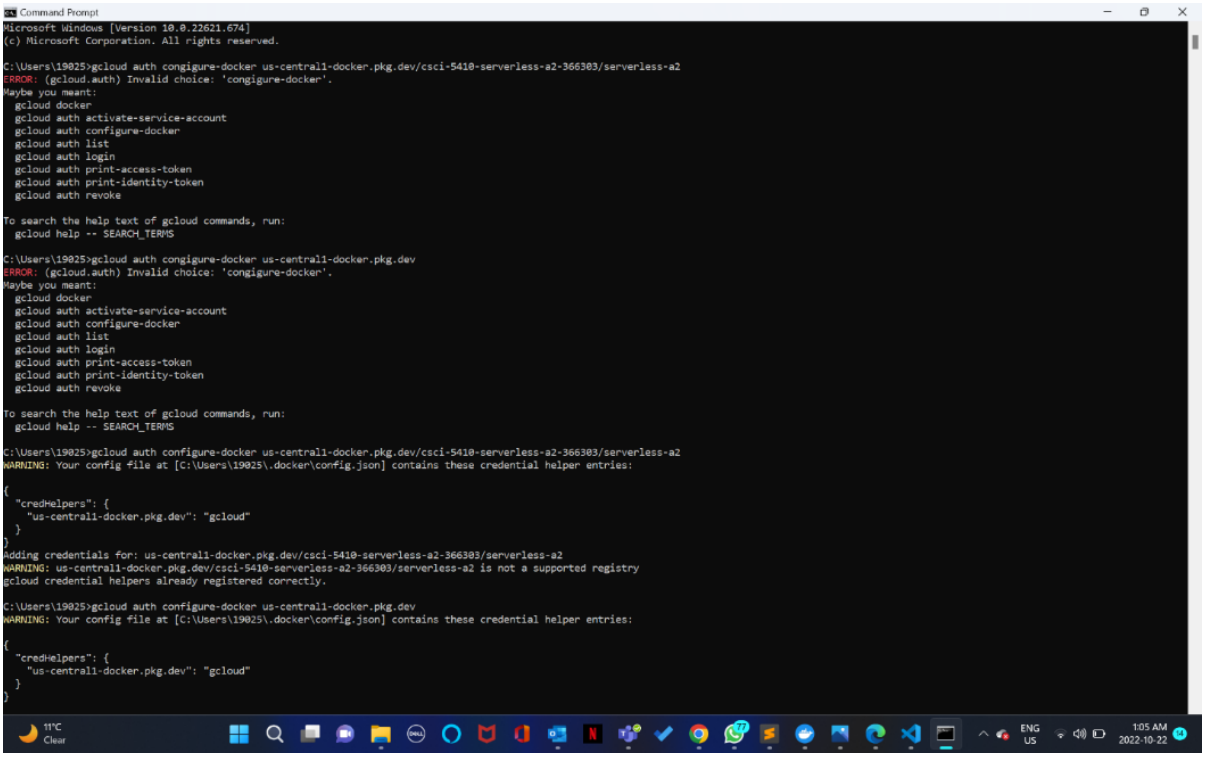


Figure 75: Google Cloud Authorization

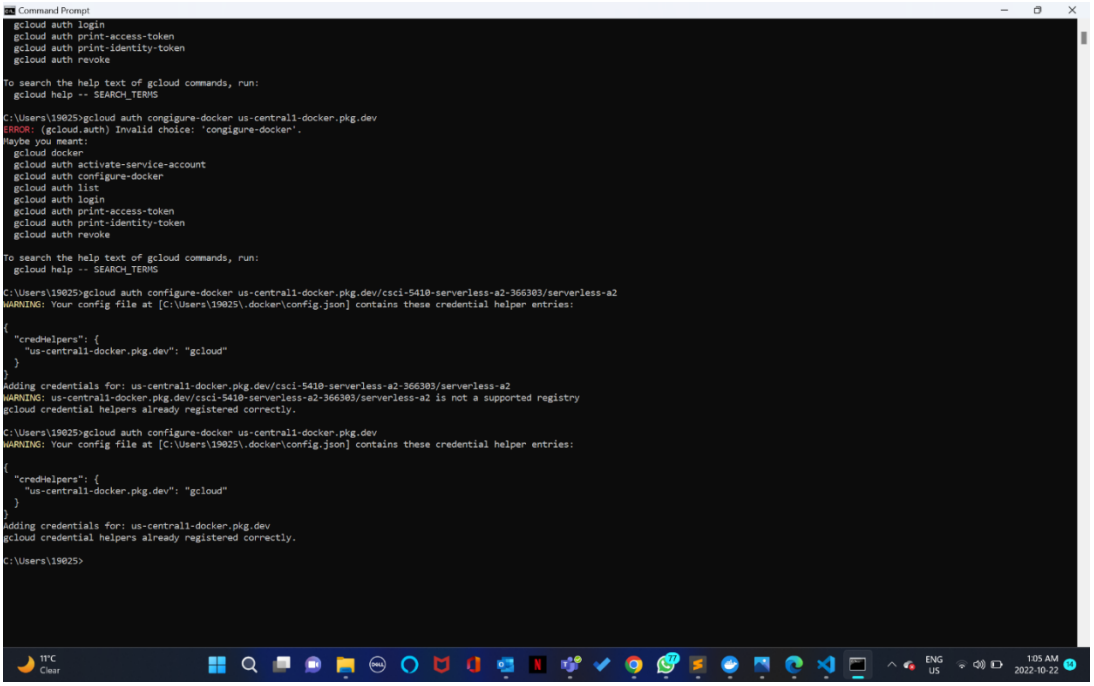


Figure 76: Google Cloud Authorization

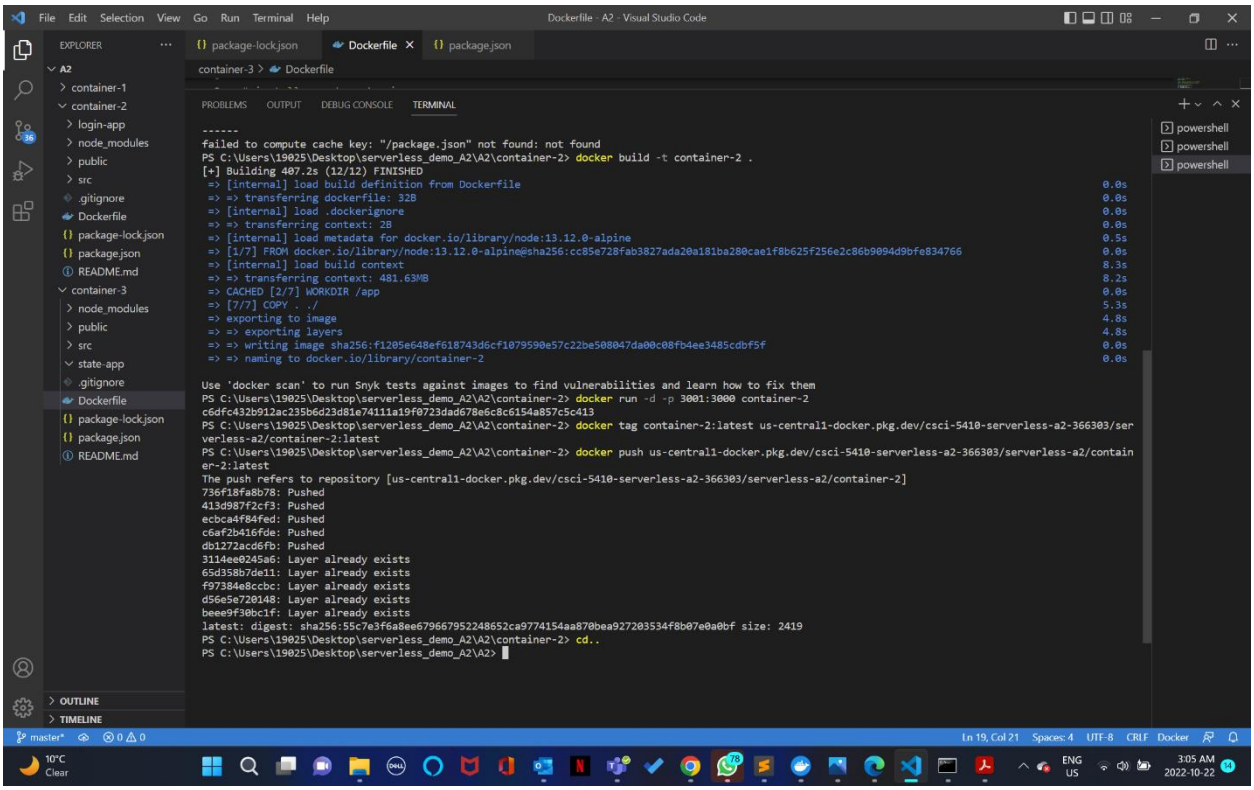


Figure 77: Docker container push commands to artifact registry

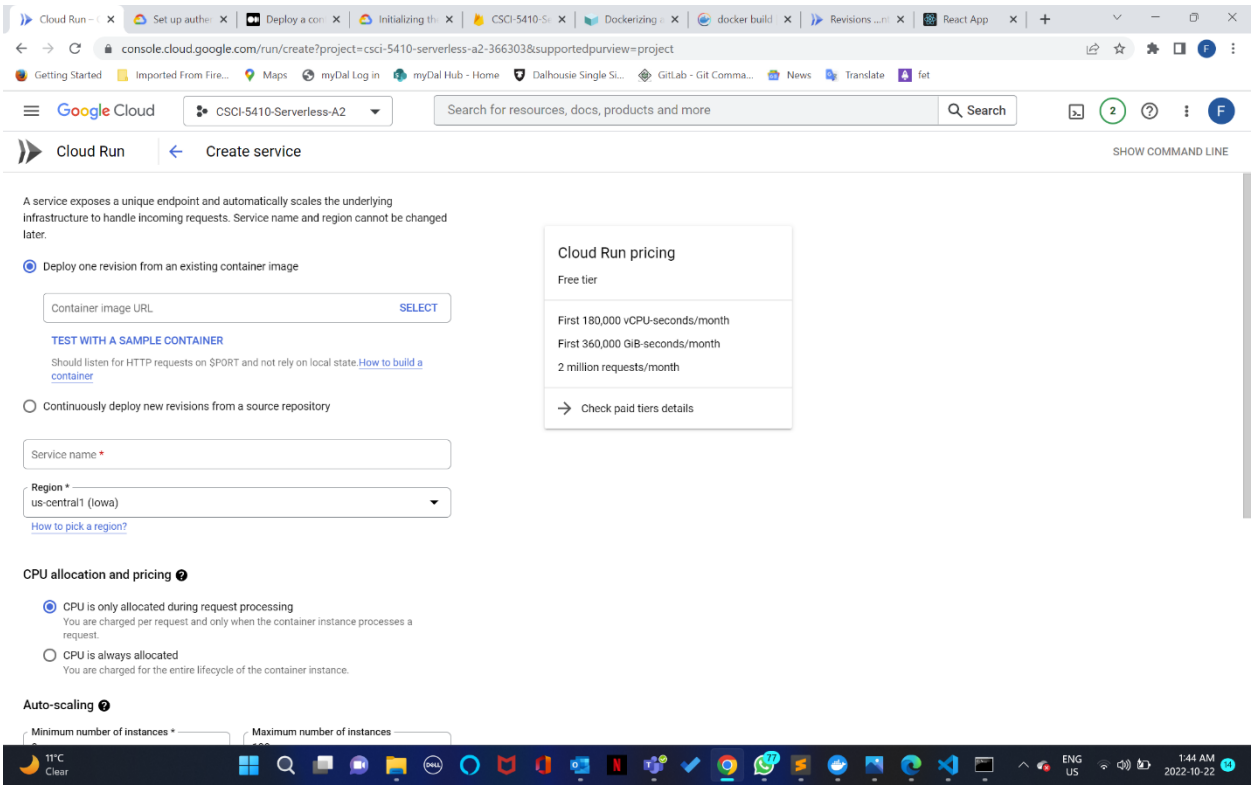


Figure 78: Creating Service

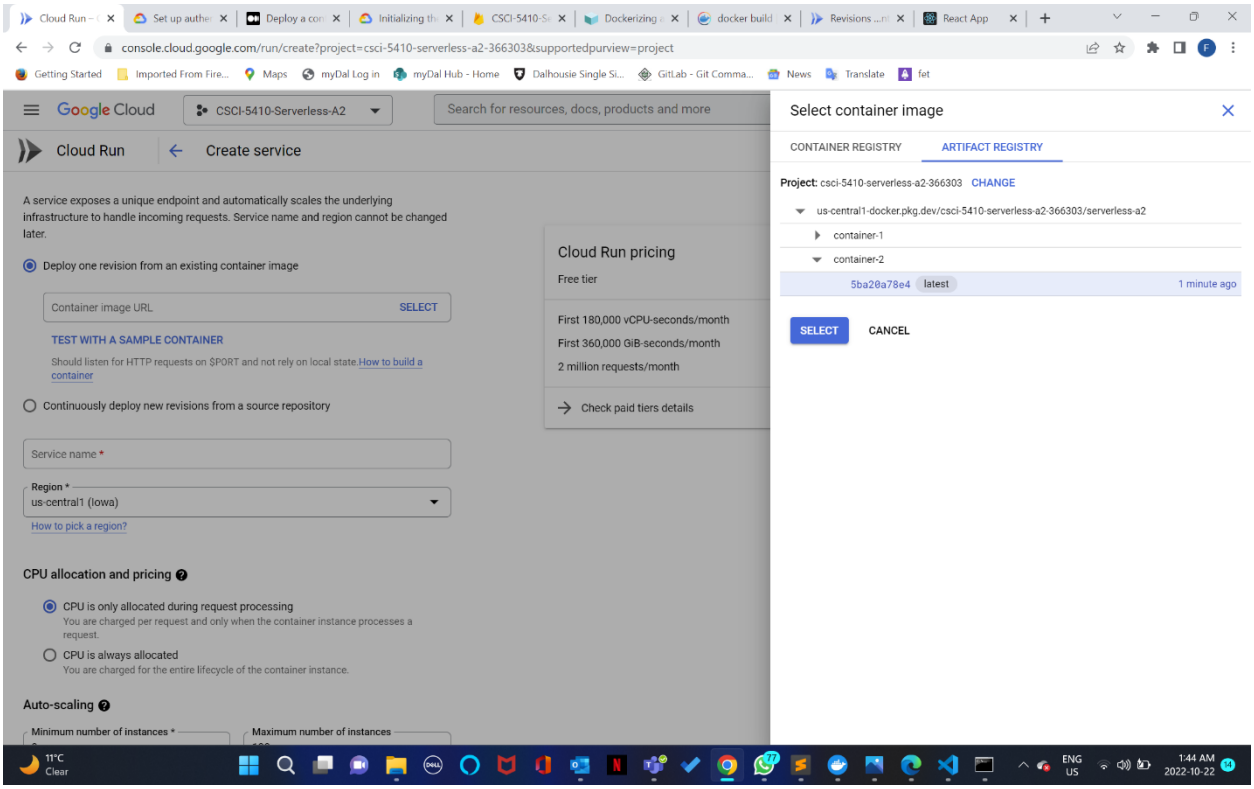


Figure 79: Creating Service

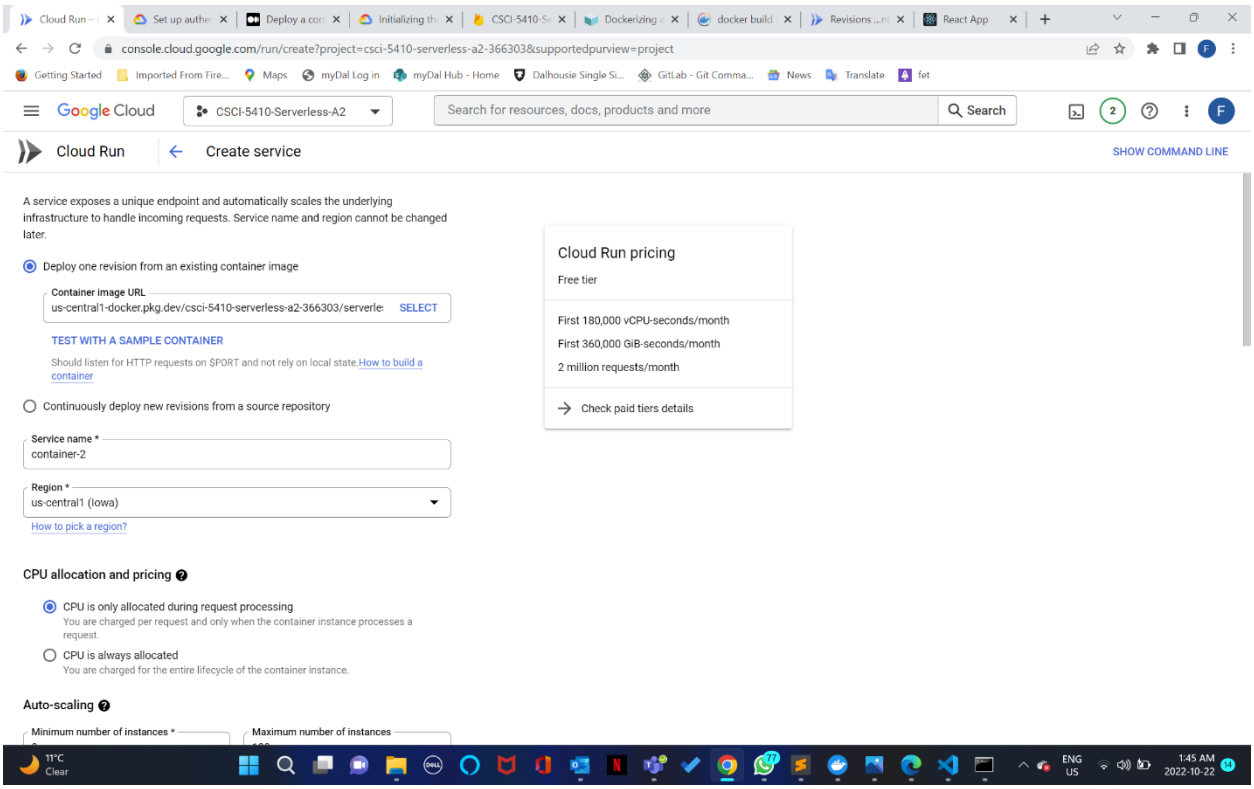


Figure 80: Creating Service

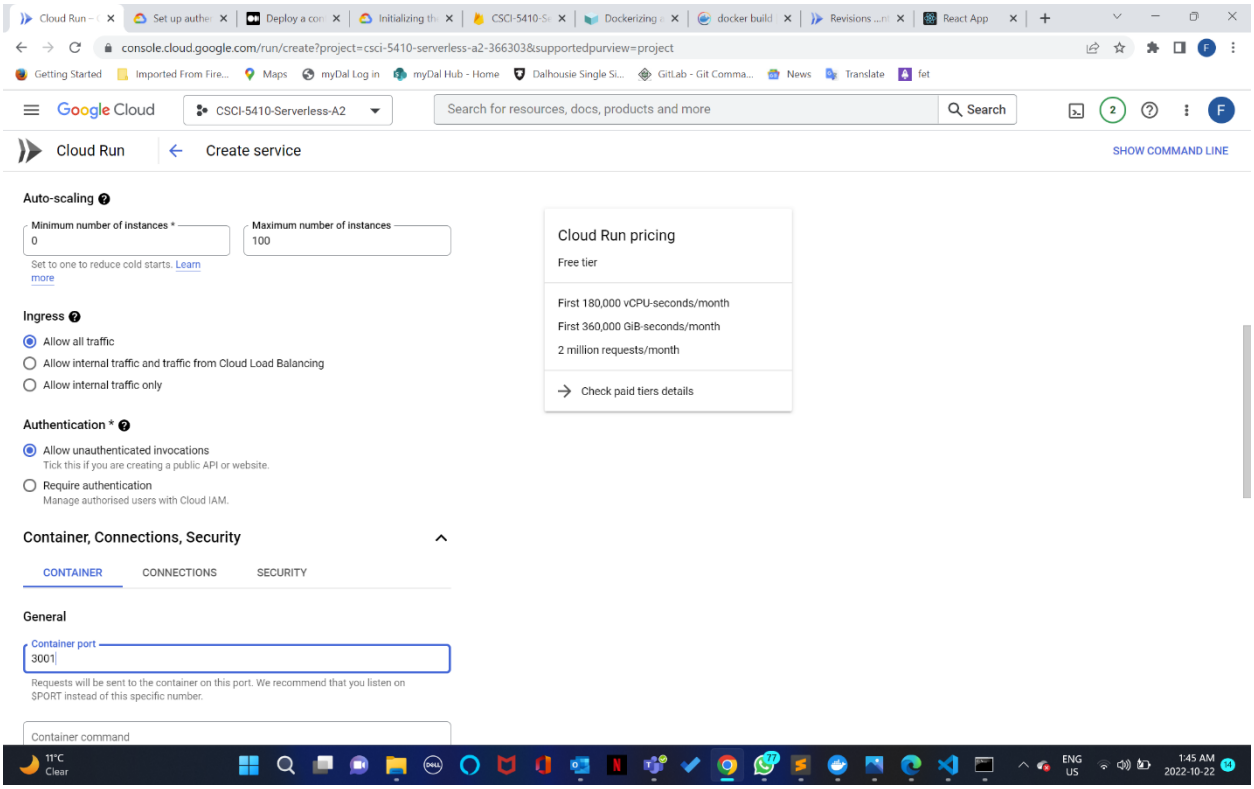


Figure 81: Creating Service

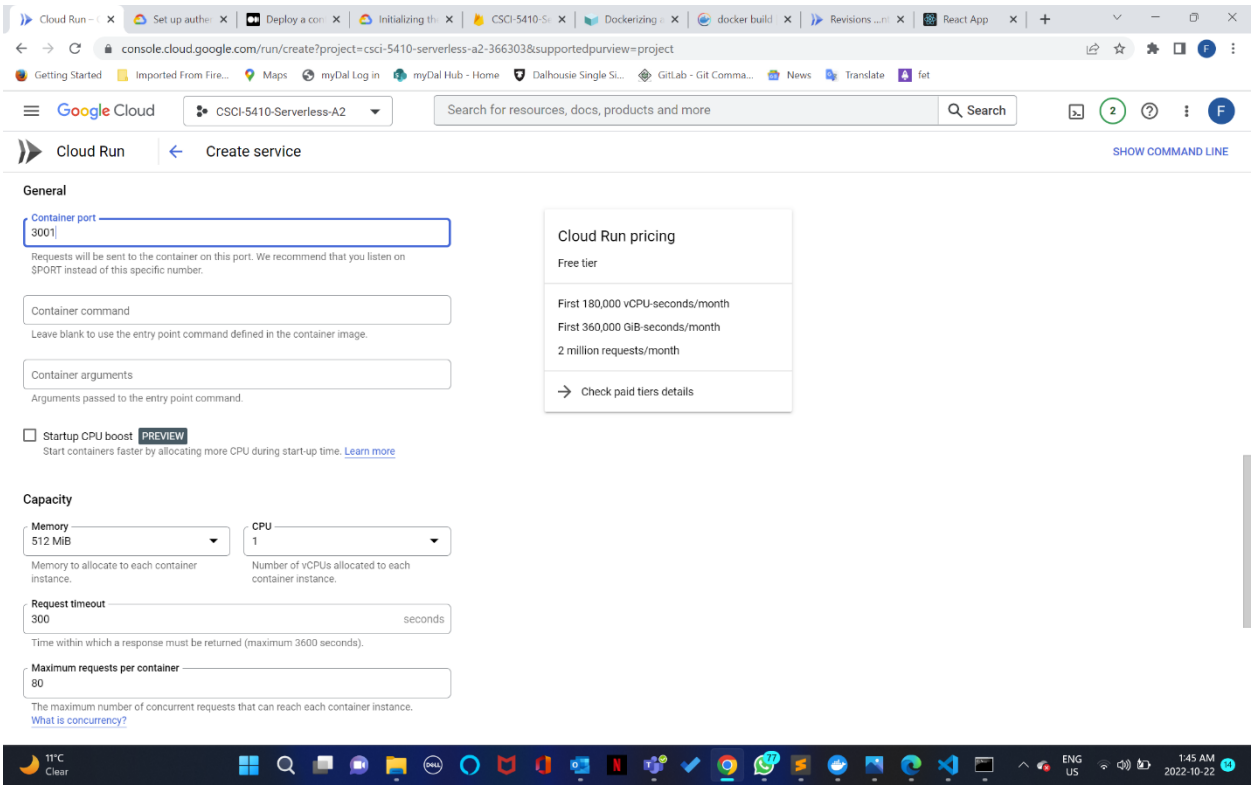


Figure 82: Creating Service

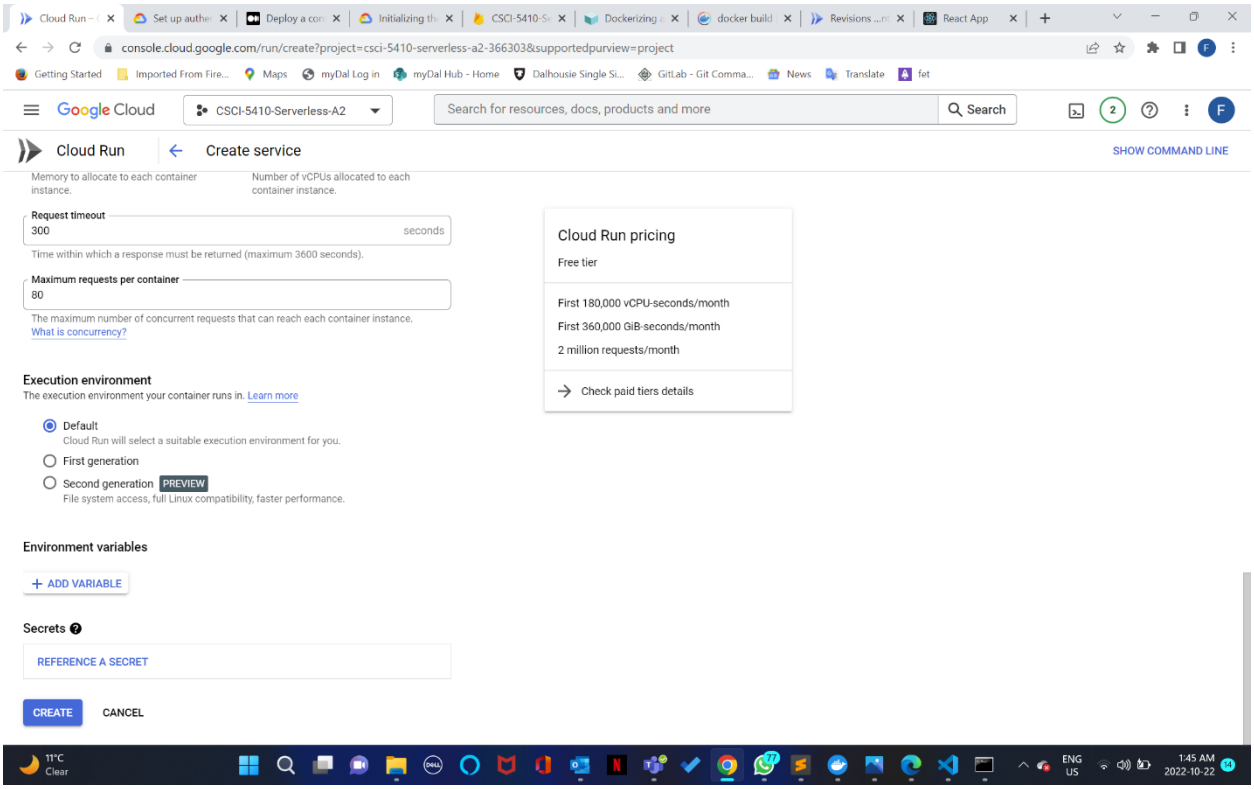


Figure 83: Creating Service

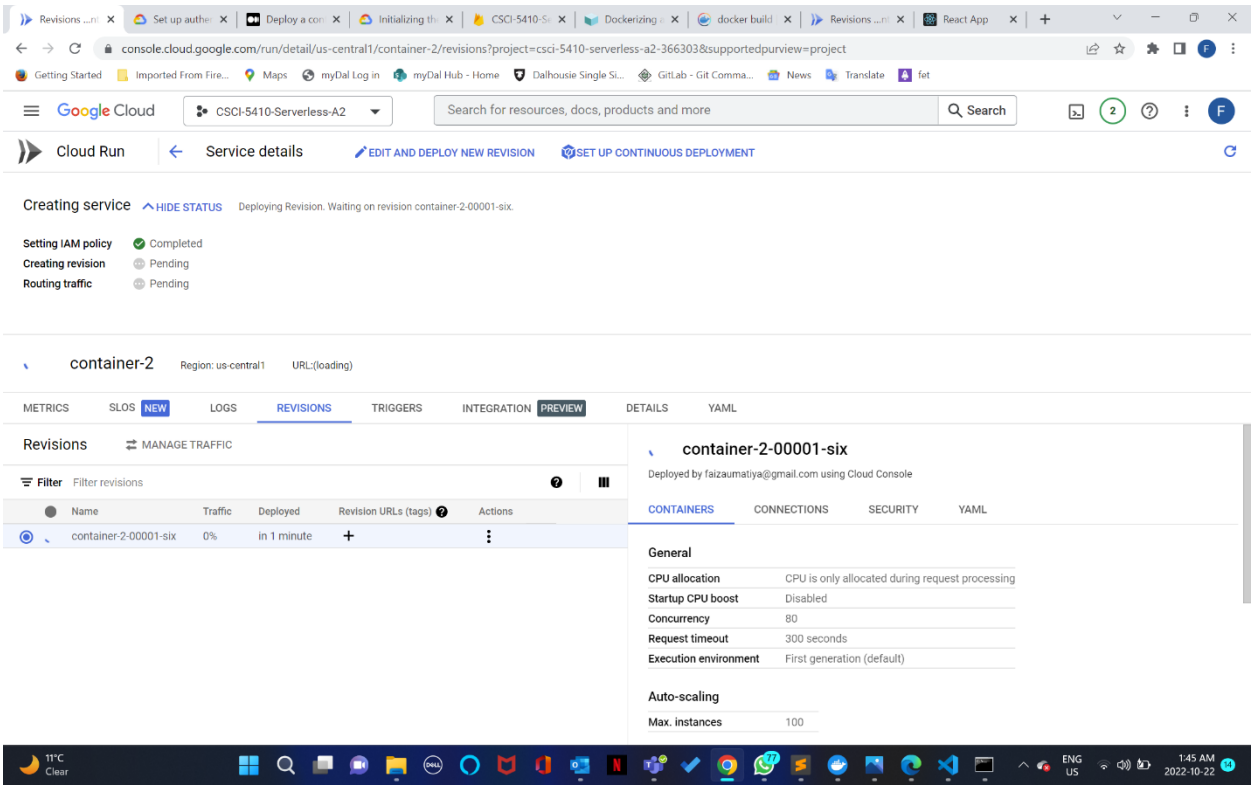


Figure 84: Creating Service

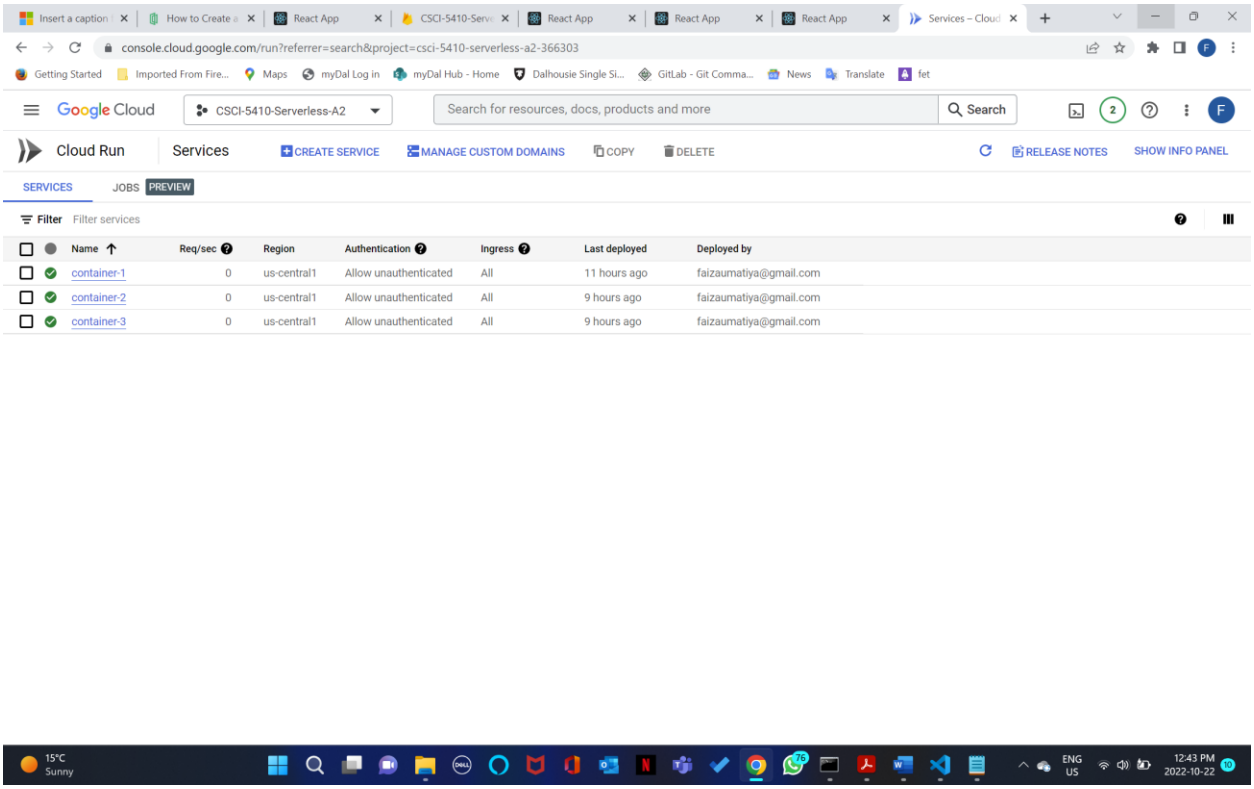


Figure 85: Containers pushed to artifact registry

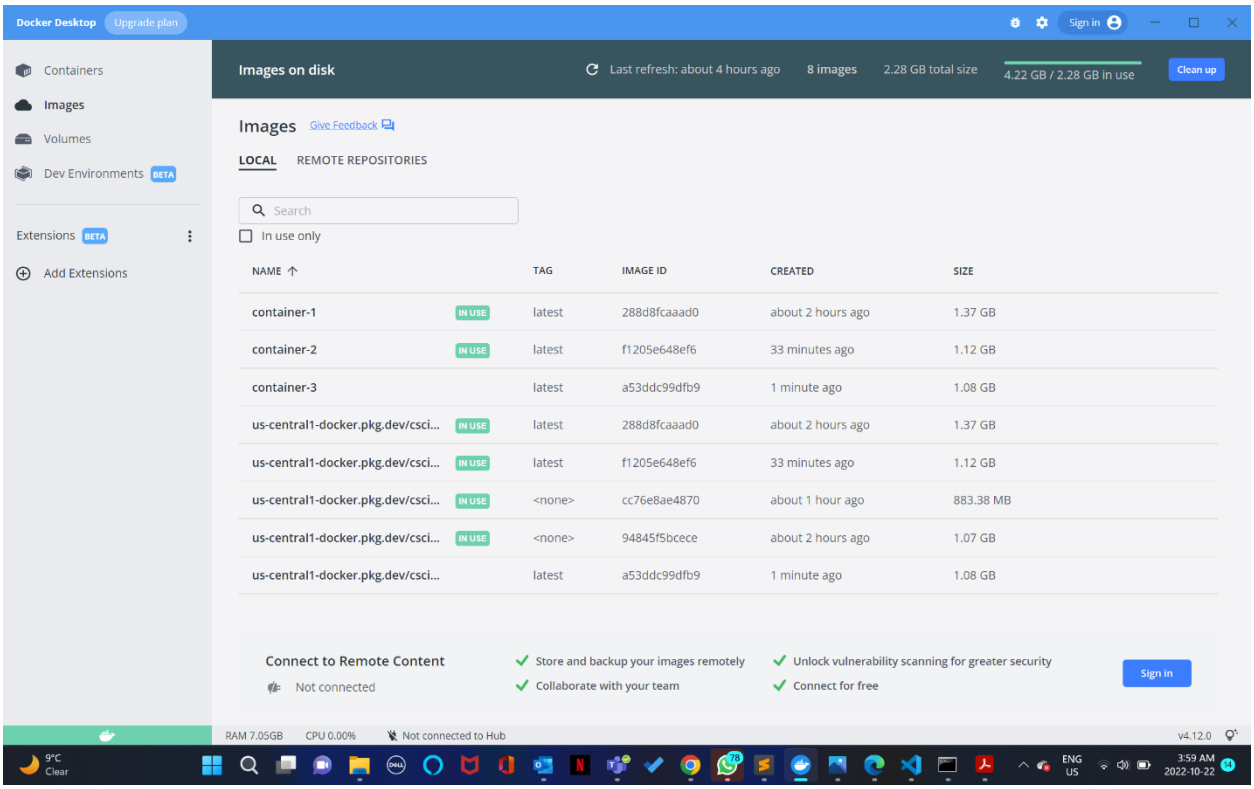


Figure 86: Containers pushed to artifact registry

Summary:

I gained knowledge of several technologies throughout the assignment, including Google Cloud Run, GCR/Artifact Registry, Docker Container, and React. The first step is to create a container for each microservice. If a container image is created by Google Cloud Run, it can be quickly deployed and help run the containers that I develop [2]. It also helps to integrate effectively with other Google Cloud services so that robust applications can be created [2]. A cloud-based service can be classified as either private or public, which aids in upholding security [2]. However, I created containers with Docker that allow for local software development regardless of the host system [3]. In addition to portability, they also offer agility [3]. System administration and environmental settings are both provided by Docker [3]. It also contains a conceptual Docker architecture with a Docker client and a Docker daemon, in which a user interacts with the Docker client rather than a daemon directly [3]. Docker implements a client-server design [3]. Applications can function without a platform.

The Docker images are produced and then pushed into the Artifact registry. The artefact registry offers a central location for managing packages and Docker container images, in addition to aiding in the storage of Docker images [4]. To save the packages from your builds, the Artifact Registry interfaces with Cloud Build and other continuous delivery and continuous integration systems [4]. Additionally, trusted dependencies that are used for builds and deployments can be stored [4]

IEEE Reference:

[1] A. Bansal, “How to Create a Registration Form using React.js and Connect it to Firebase,” *Engineering Education (EngEd) Program | Section*. [Online]. Available: https://www.section.io/engineering-education/registration-form-react.js-firebase/. [Accessed: 23-Oct-2022].

[2] “What is cloud run,” *Google Cloud*. [Online]. Available: https://cloud.google.com/run/docs/overview/what-is-cloud-run. [Accessed: 23-Oct-2022].

ACM Reference:

[3] S. Garg and S. Garg, "Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security," 2019 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR), 2019, pp. 467-470, doi: 10.1109/MIPR.2019.00094.

IEEE Reference:

[4] F. Martinez, “Artifact Registry: the new way to keep your App artifacts and Docker Images on GCP,” *Google Cloud - Community*, 18-Apr-2020. [Online]. Available: https://medium.com/google-cloud/artifact-registry-the-new-way-to-keep-your-app-artifacts-and-docker-images-on-gcp-d1a72da09ff9. [Accessed: 23-Oct-2022].

[5] Google, "Cloud Firestore," Google, [Online]. Available: https://firebase.google.com/products/firestore. [Accessed 20 October 2022].

[6] Docker, "Developers Love Docker," Docker, [Online]. Available: https://www.docker.com/. [Accessed 20 October 2022]