CSCI 5410

Serverless Data Processing

Assignment 2

Name : Sagarkumar Pankajbhai Vaghasia

CSID : vaghasia

Banner ID : #B00878629

Gitlab Link :

https://git.cs.dal.ca/vaghasia/csci5410-f23-b00878629/-/tree/A2

PART-A Reading Task

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

In recent times, the development in IT industry have shown that the sector of cloud computing is both very demanding and rewarding. As the companies are growing, they are shifting towards cloud platforms due to many factors. Cloud services providers are continuously making effort to make transition smooth and robust still they have not reached to the point where they can make cloud automation. In this paper, the authors have described about making industrial automation by cloud computing and strategy for effective deployment and maintenance.

According to some researchers, they have predicted that more than 80% of the enterprises will move towards cloud infrastructure by 2025. For this, we need continuous integration(CI) and continuous delivery(CD) as a culture not as a practice in an organization. Continuous Integration(CI) is nothing but a software development process which helps the development teams to integrate their software components early and often which prevent them to get into large integration issues between the code updates. This practice helps the teams by faster issue discovery, reducing the possibilities of integration issues, boosts developer's productivity. Another term is continuous delivery(CD) which means software engineering policy that aims to reduce the time needed for the development of software by delivery small fragments of the working software and getting the feedback of the user(this practice is also known as agile development). The major advantages of CD are reduction in deployment risk, increased user feedback, quicker reaction time, more frequent releases and faster feature implementation. The drawback of CD is it needs lot of time and has a challenging and dynamic culture. Authors have proposed that Docker can be used to implement CI/CD and docker will verify the required information in the container to build the application. To implement CI/CD using docker the following components should be used : Docker host, Jenkins master, Jenkins slave and GitHub repository.

According to the author the increasing need for shorter development lifecycles, CI, CD, and cost savings in cloud computing led to the rise of containers. Docker containers provide agility and availability of dependencies. They are also very portable, and developers can build software in their local computer as it will be run identically on the host environment. Docker is implemented using a client-server architecture where Docker Client talks to Docker Daemon which runs on host machine. The best practices in container security are namespaces, control groups, Docker Daemon, Linux kernel capabilities and security. There are some points to consider while using docker containers. First one is containers are secure if the process is running inside non-root. Another thing is third-party containers are not safe, instead of it use private registry like Docker Trusted Registry. There are several industry recommended tools for docker

security namely Docker Bench for Security, CoreOSs Clair, Docker Security Scanning and App Armor/SELinux.

In the paper, the authors started by examining the industry trend toward moving traditional on-premises data centers to the cloud and the reasons why cloud infrastructure automation is essential for developer teams looking to create better, more rapid, and scalable products. Then authors discussed about Docker Security, CI/CD pipeline using Docker containers and the best practices that can be followed. The boilerplate code will be made accessible to the open source community in the future, which will help developers get started even faster.

References :

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.

URL: https://ieeexplore-ieee-org.ezproxy.library.dal.ca/document/8695332

PART-B Containerized Application using GCP.

Registration App (container 1) URL: https://container1-4hq67cpy3q-uc.a.run.app

Login App (container 2) URL: https://container2-4hq67cpy3q-uc.a.run.app

Online Users App (container 3) URL: https://container3-4hq67cpy3q-uc.a.run.app

- Firstly, I opened firebase[1] in the browser and clicked on console button to setup firebase account. Figure-1 shows the homepage of firebase.

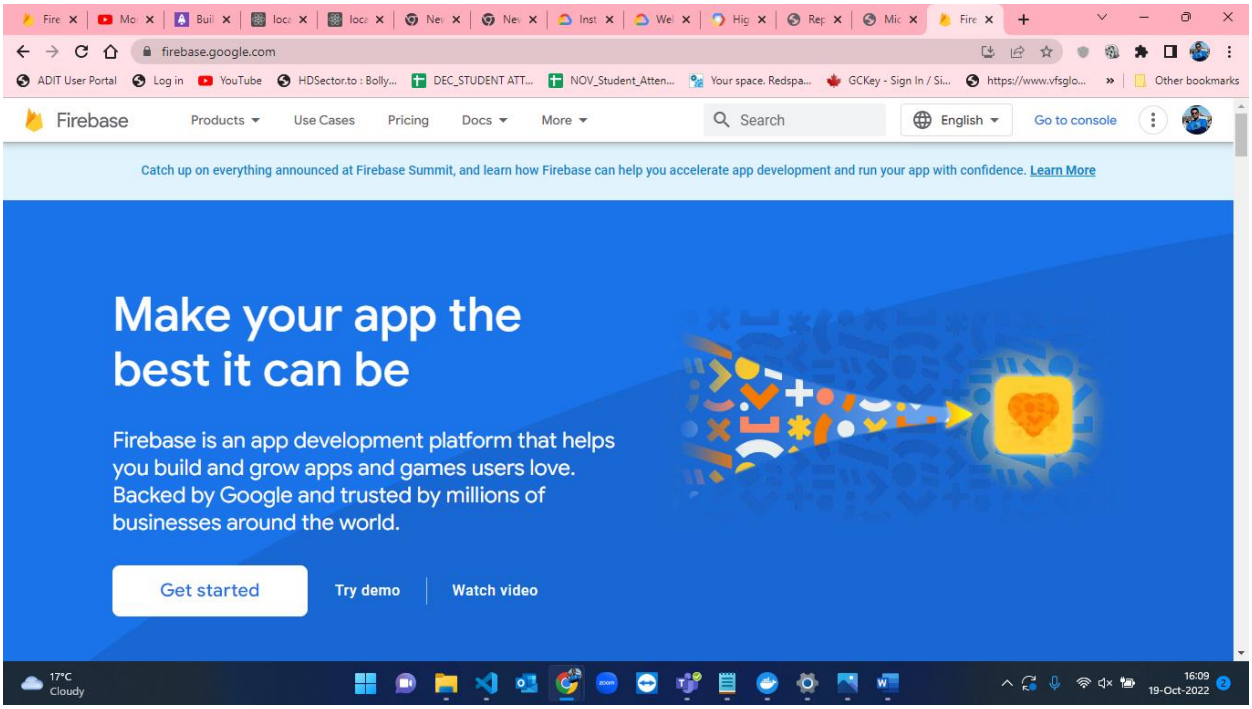

Figure-1: Firebase homepage.

- Figure 2 to Figure 6 represents the setup of a new project in firebase account. Firstly, I clicked on Add project, then we must input the project name where I input it as “CSCI5410-A2”. After that, it took some time to create the project. Once the project is created, I clicked on the continue button.

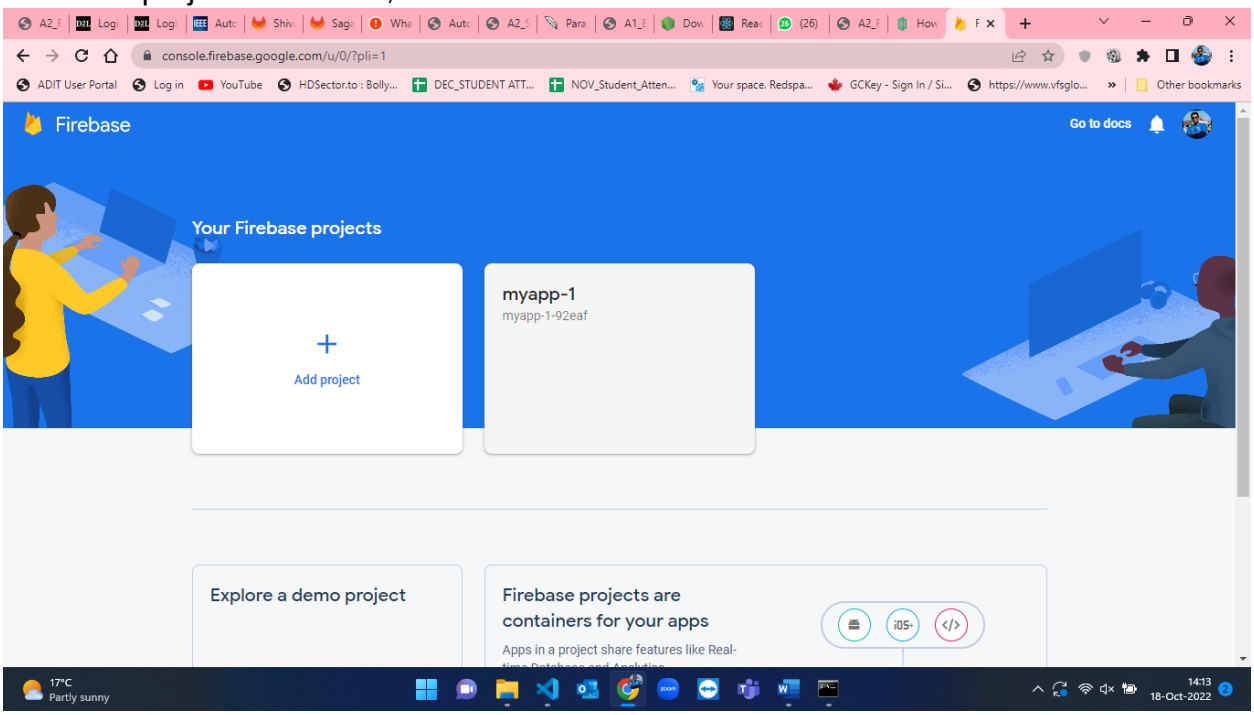

Figure-2: Console page of firebase.

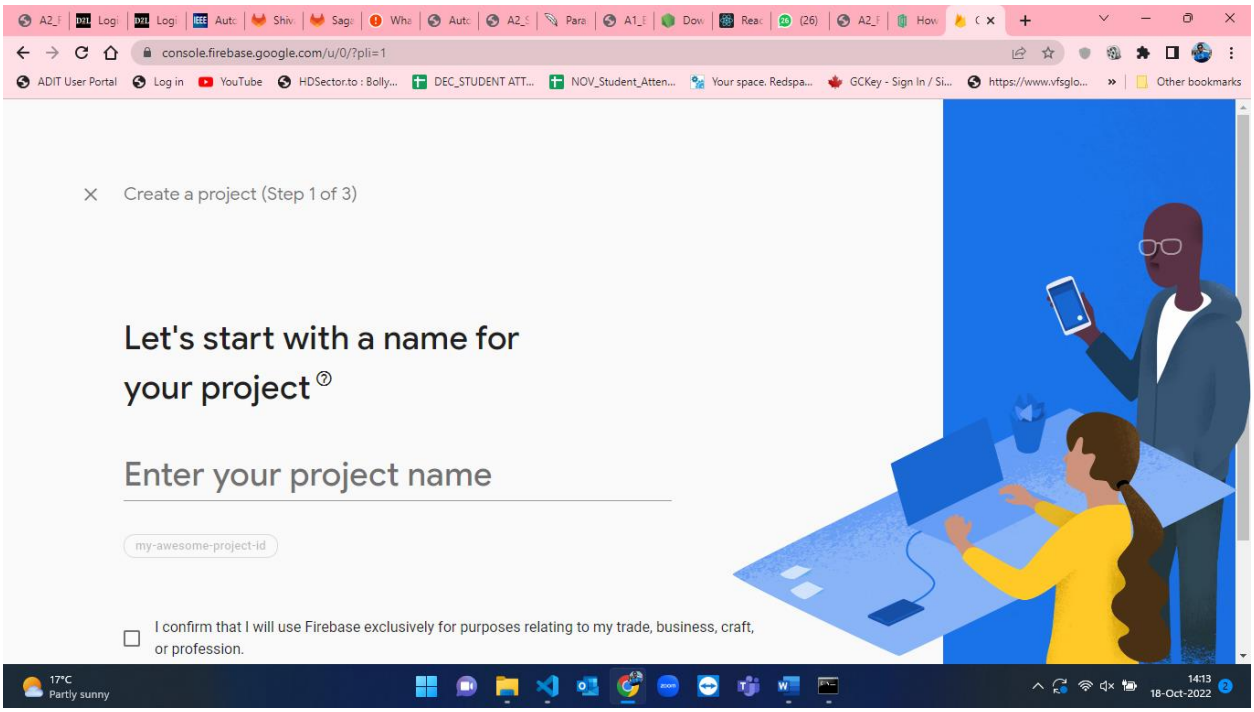

Figure-3: Blank project name page in firebase.

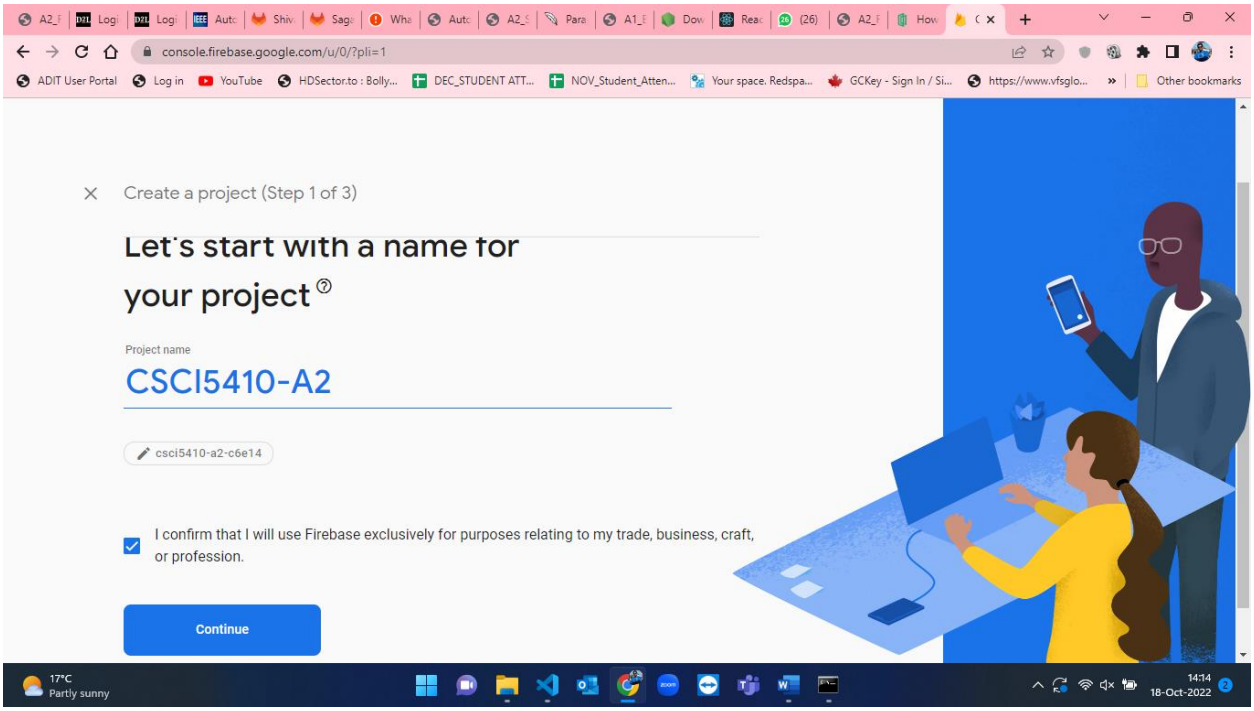

Figure-4: Create new project in firebase console.

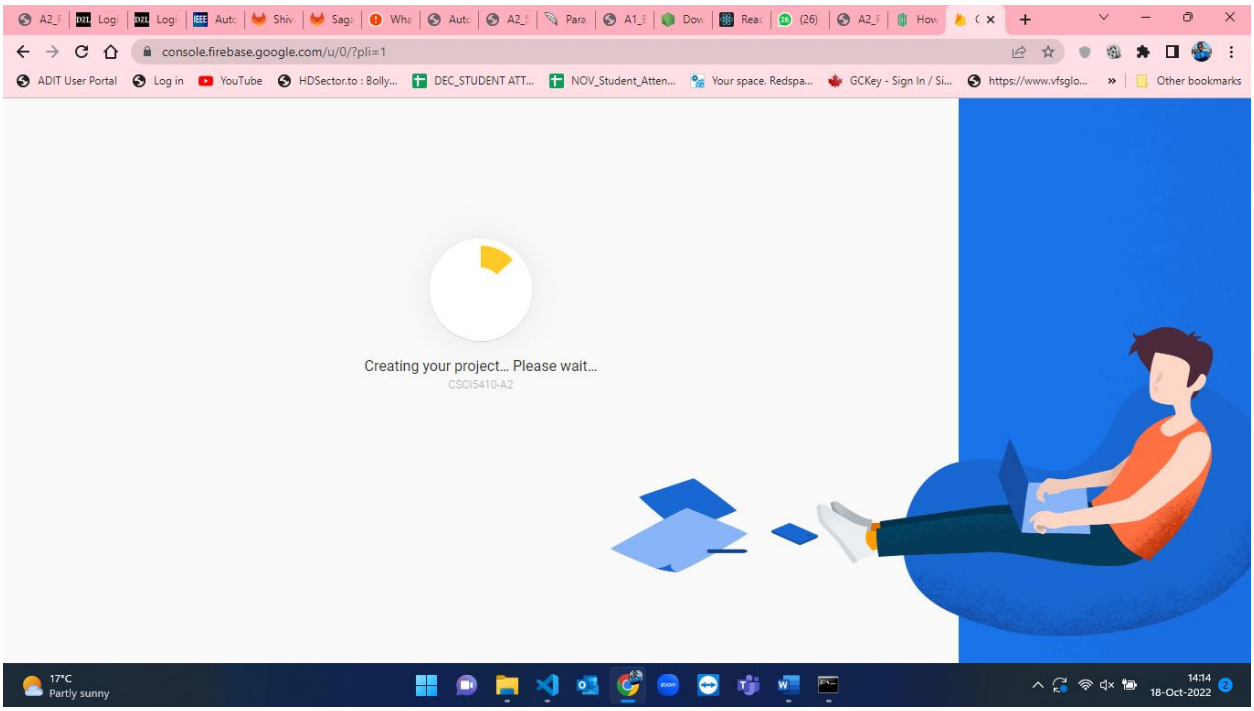

Figure-5: Creating new project page in firebase.

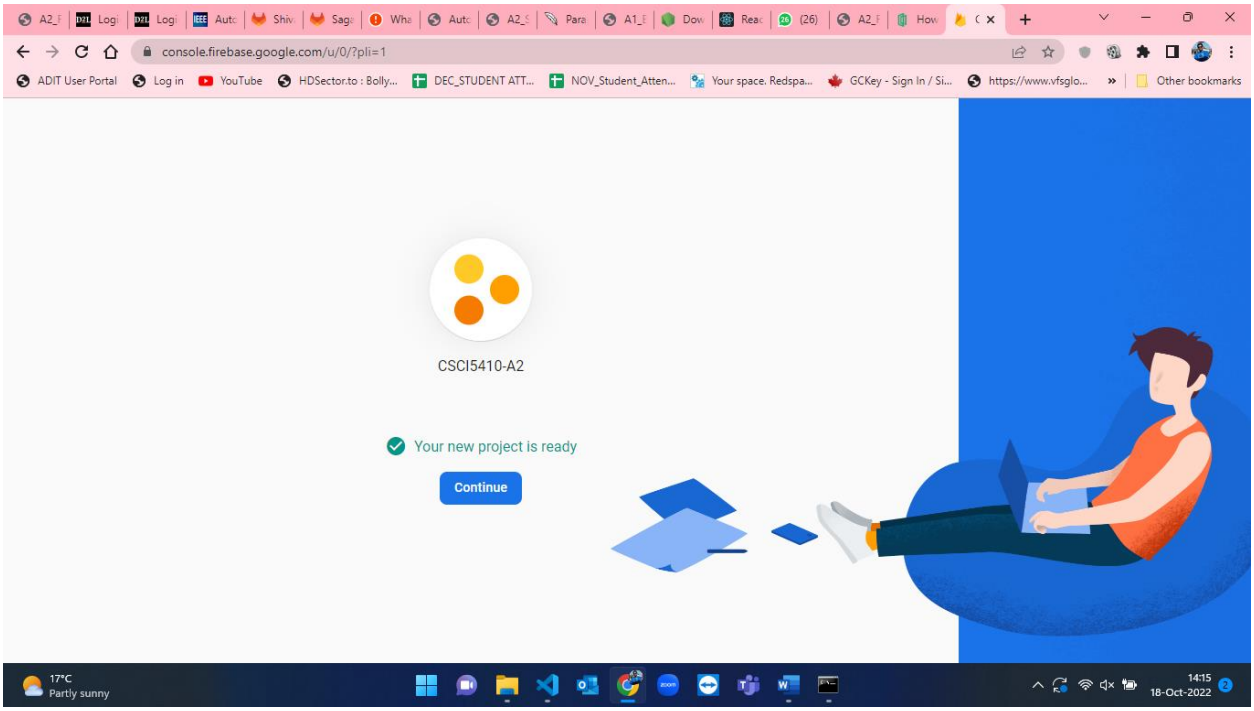

Figure-6: New project read to go in firebase.

- Once the project is created, after that I clicked on continue button which took me to the project dashboard page. The Figure-7 indicates the project dashboard in firebase.

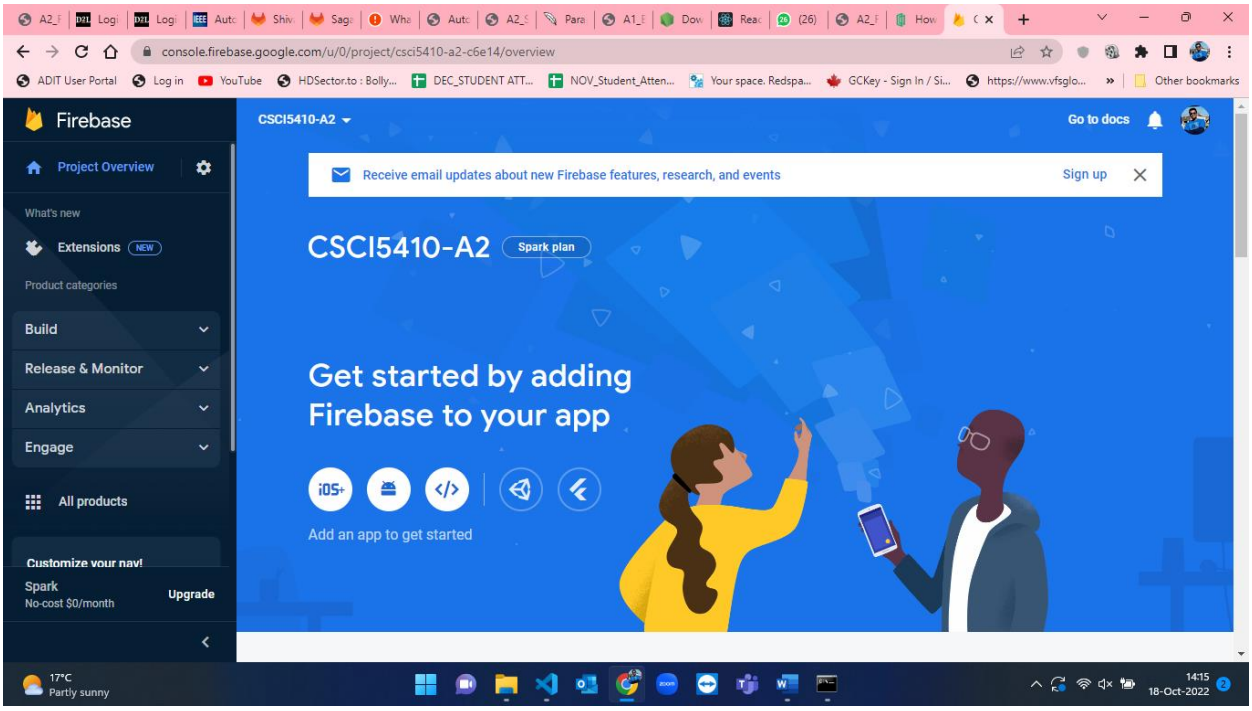

Figure-7: Project dashboard in firebase.

- From the project dashboard, I clicked on Build and from that I clicked on Firestore Database to setup firestore database. Figure-8 represents the homepage of firestore database.

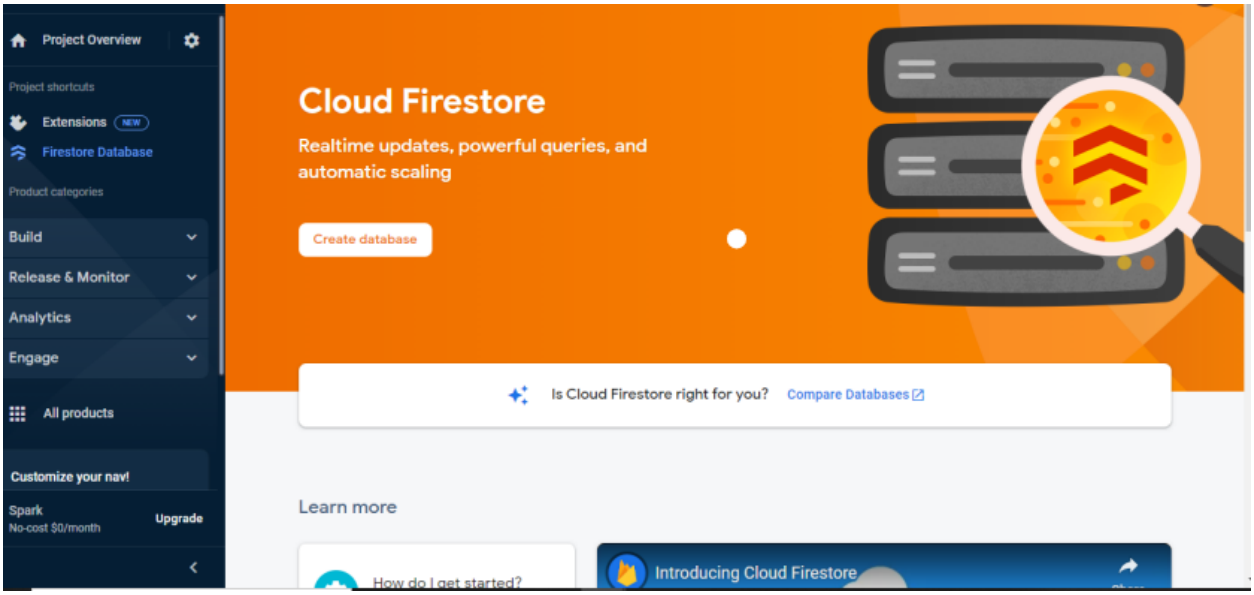

Figure-8: Firestore database homepage.

- After reaching dashboard in firestore database, click on “Create Database” to create a new database. Figure-9 to Figure-11 shows the setup of firestore database. Here, I have selected test mode for the current application.

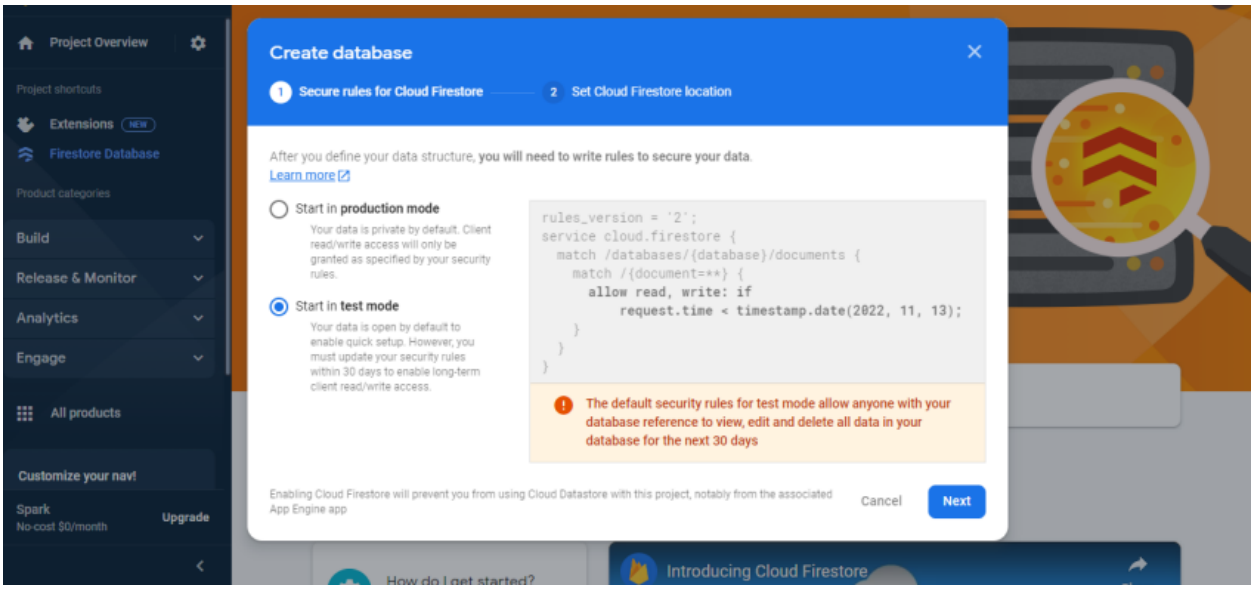

Figure-9: Selecting environment for database in firestore.

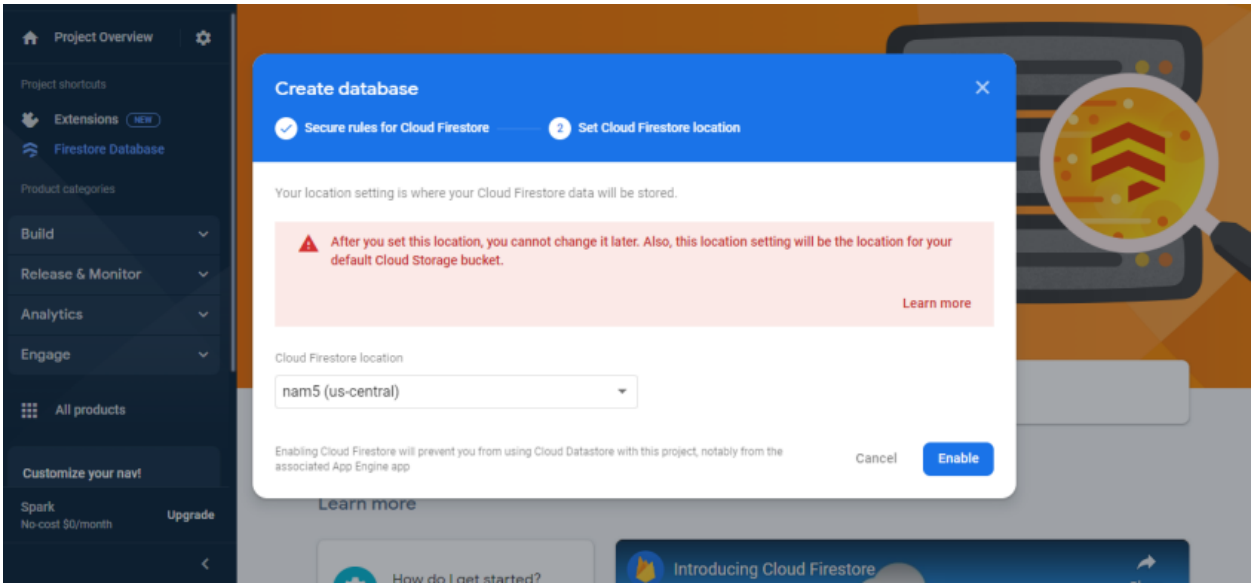

Figure-10: Setting cloud firestore location in firebase.

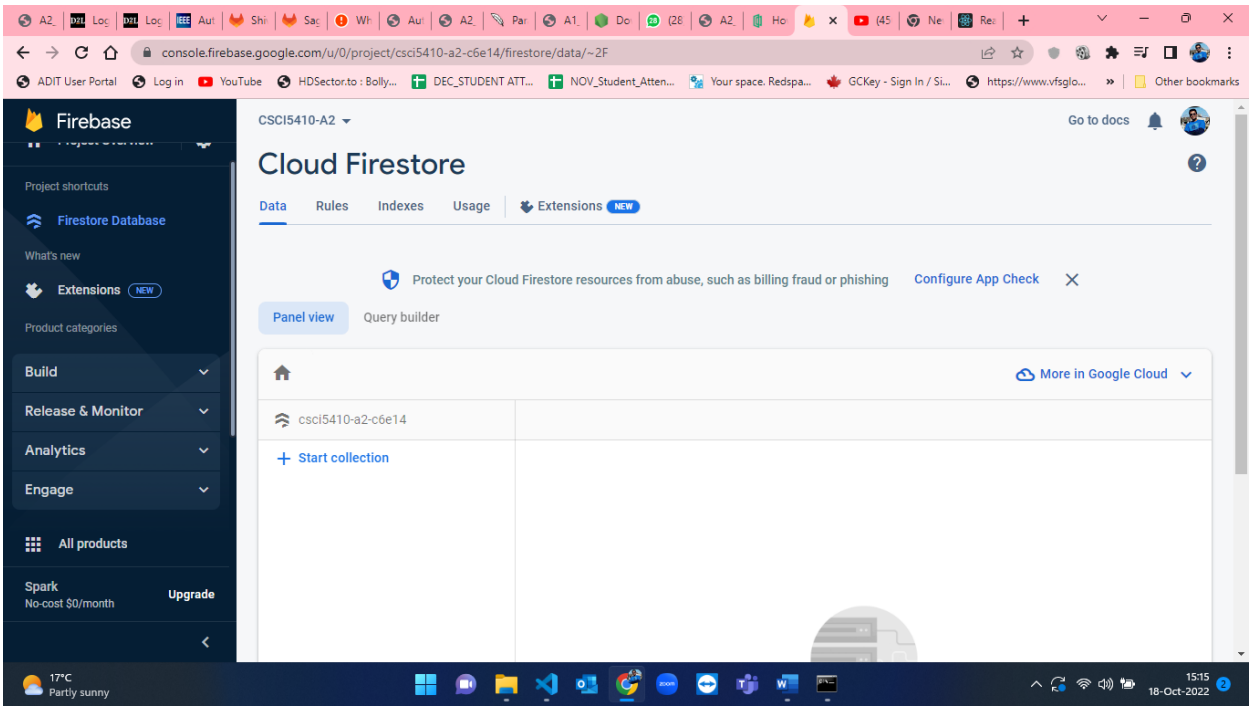

Figure-11: Default empty database din firestore.

- From the project dashboard, click on “Add an app to get started”. This is used to setup the app in firebase to get firebase configuration. Figure-12 to Figure-14 depicts the setup of an App in firebase for firebase configuration.

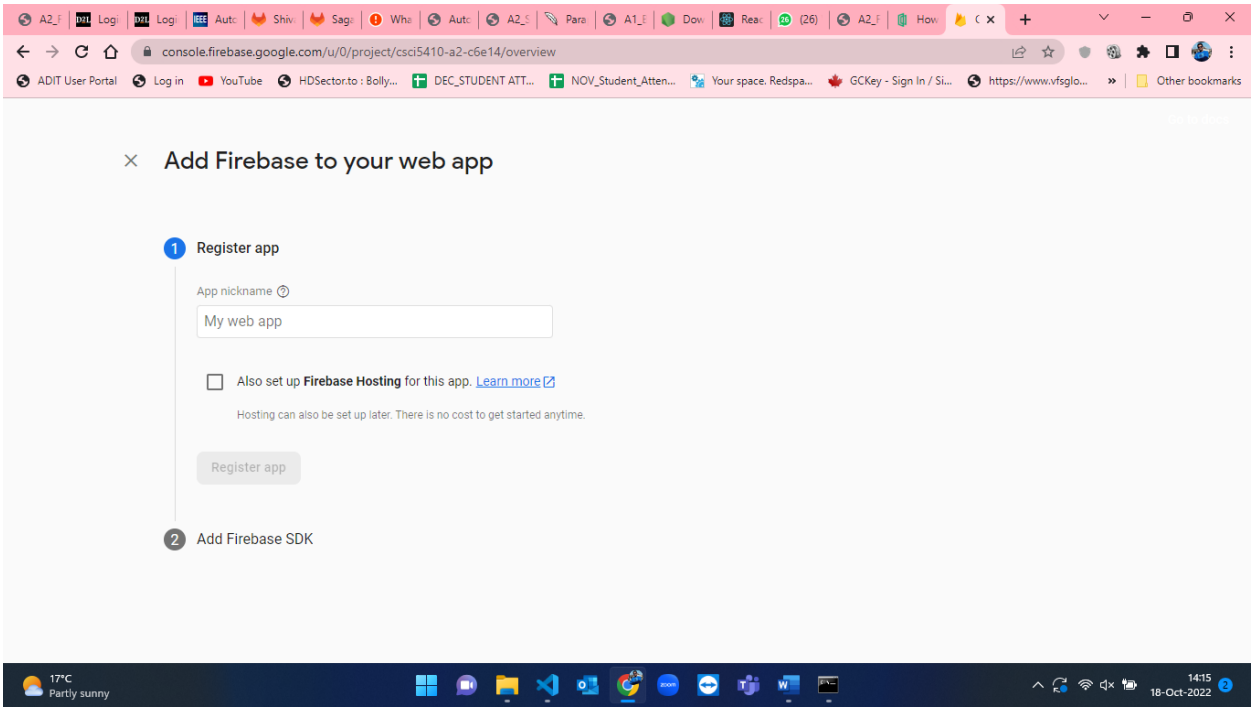

Figure-12: Web Application registration form.

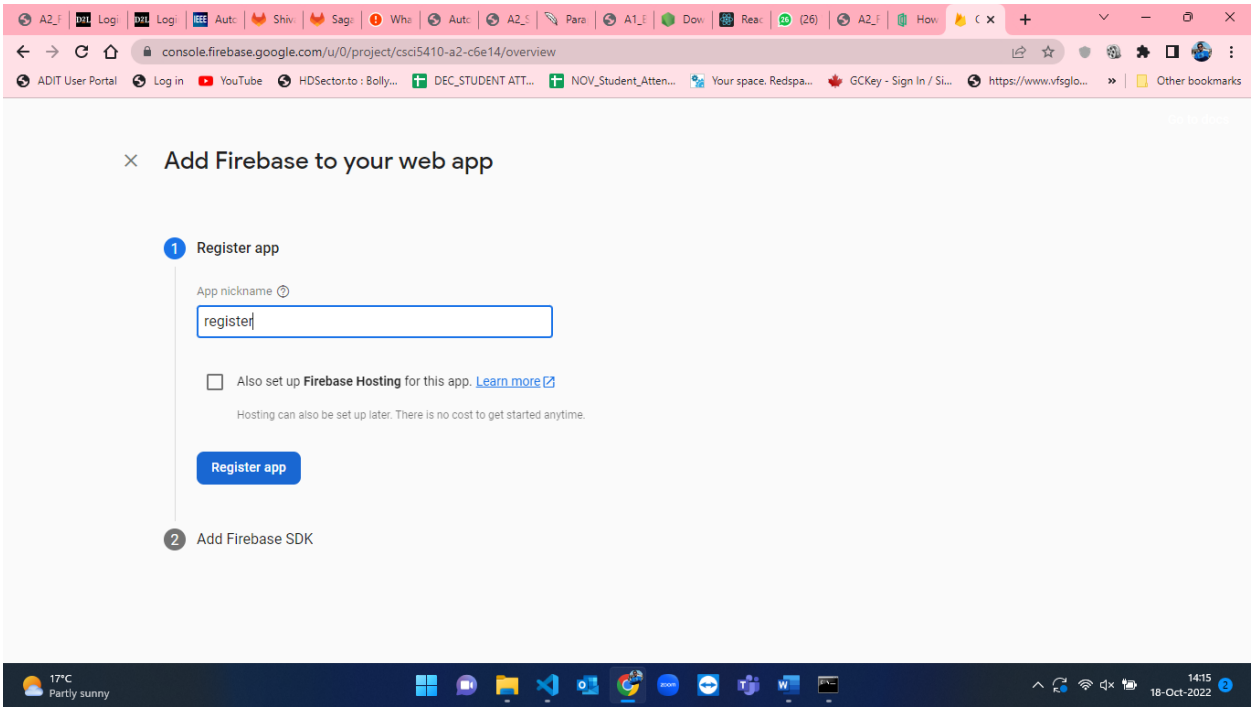

Figure-13: Named as “register” for app1 in Web Application registration form

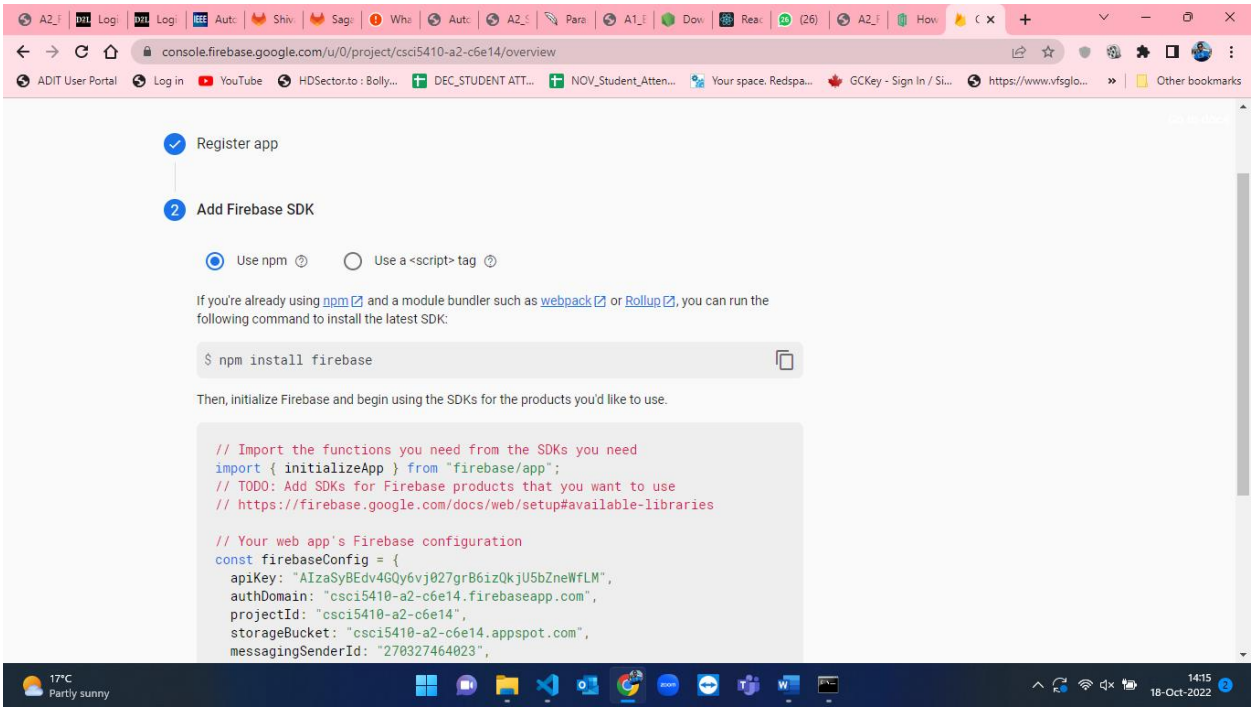

Figure-14: Firebase firestore configuration for using SDK.

- Once, the firebase database is setup, we have to configure the SDK into our react app container. For that I have used “npm install firebase” command in the directory where our container is located. Figure-15 and Figure-16 shows the installation of firebase database configuration in the local directory.

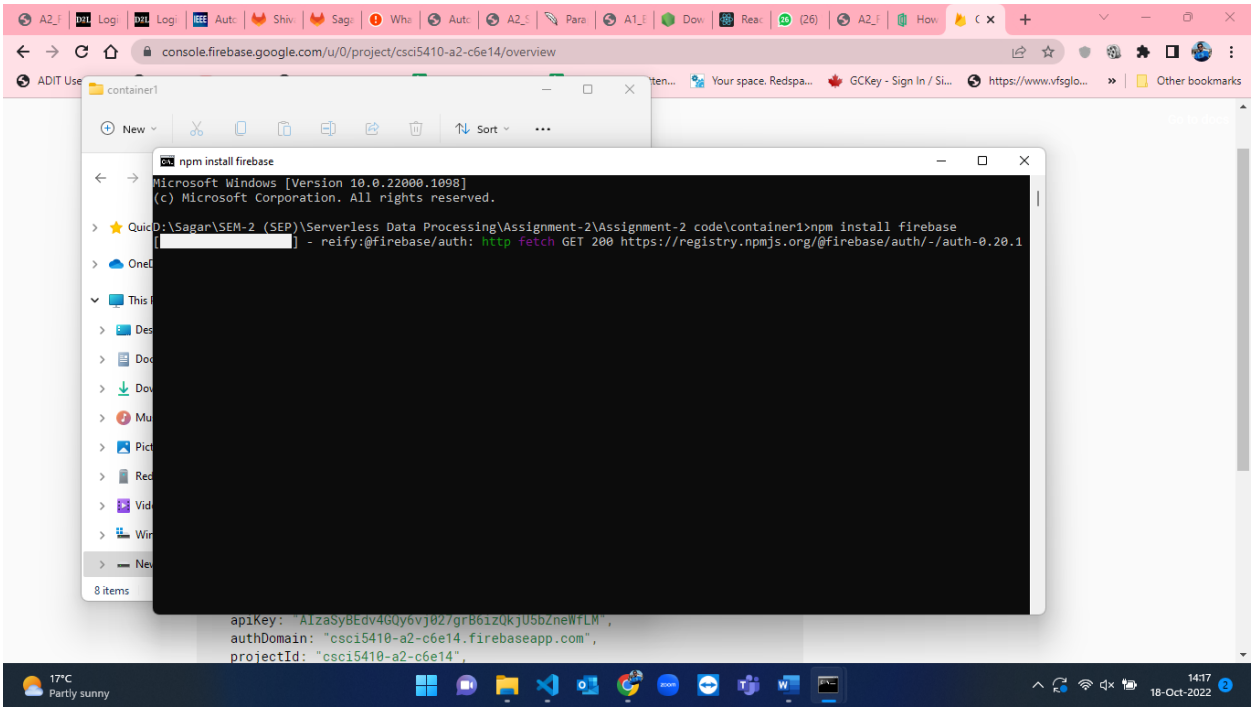

Figure-15: Installation of firebase in the directory named as container1 for app1 .

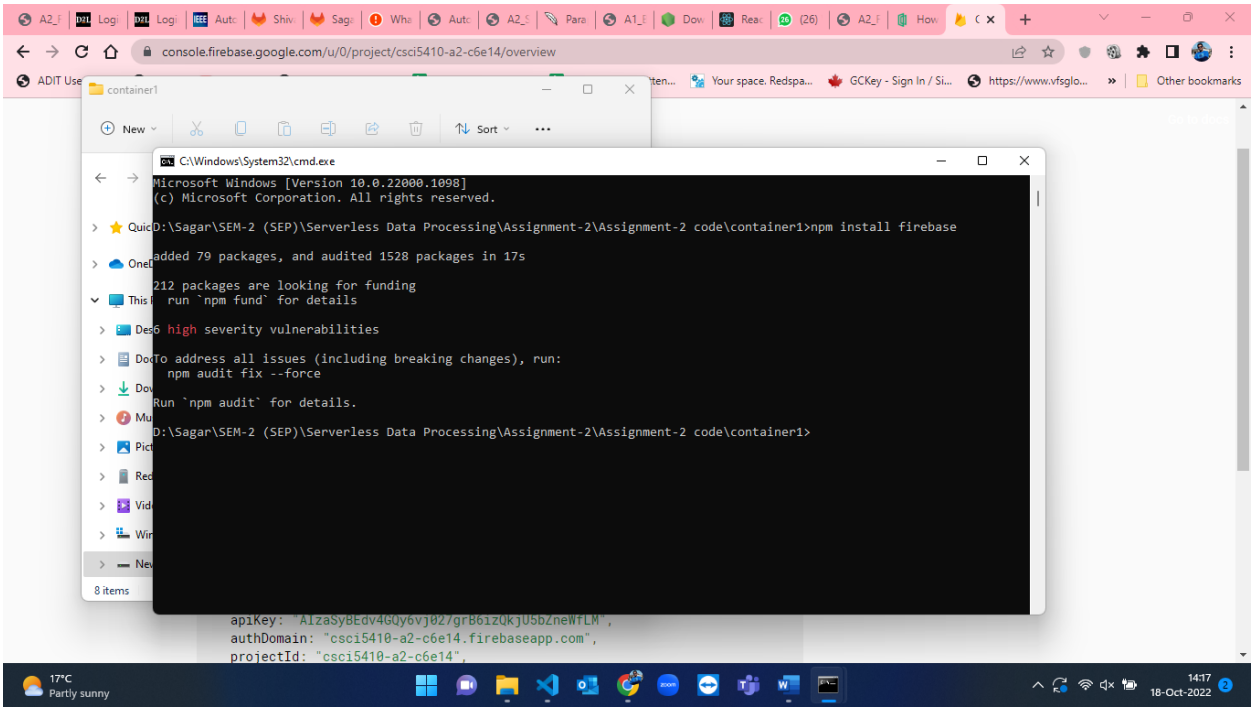

Figure-16: Firebase installed successfully in container1(app1).

- After installation of firebase, we have to create a firebase.js file to use firebase database from our react app. The below attached Figure-17 shows the script to initialize firebase and use SDK in our application. This script has been taken from Figure-14 when we are creating application.

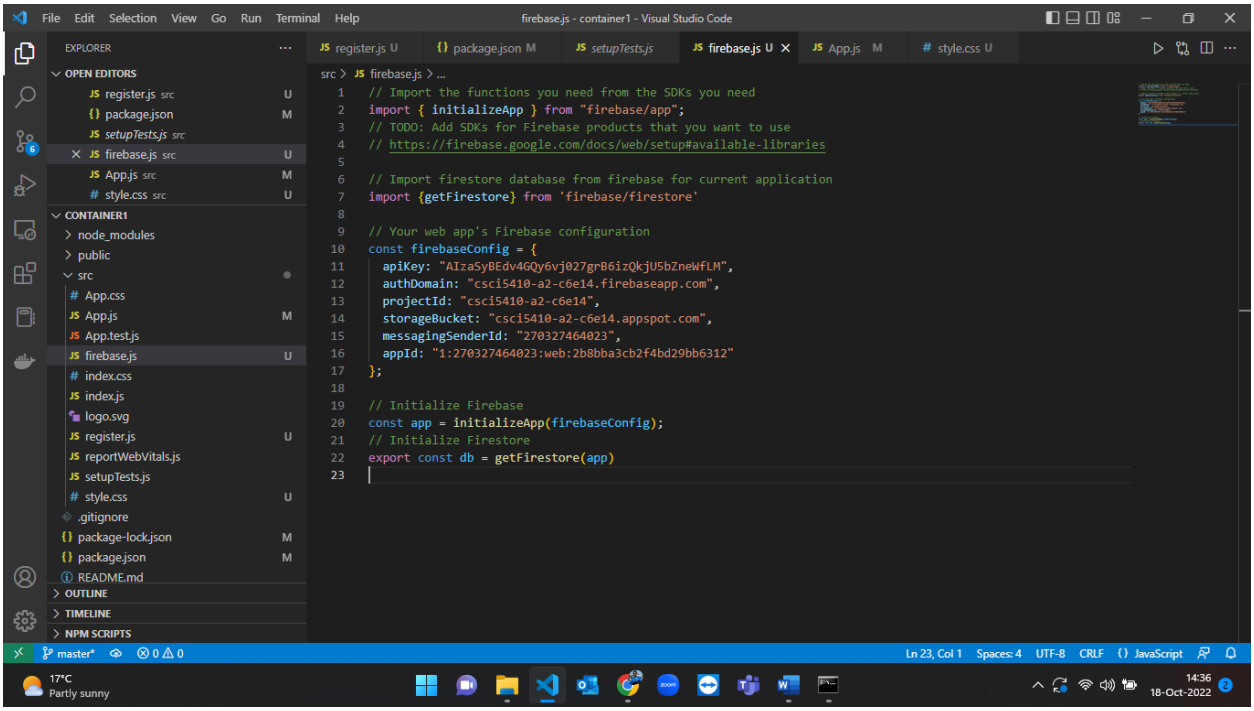

Figure-17: Script to initialize firebase in web application.

- After initialization of firestore, I created register.js file where front-end and back-end code for registration of a user is there. The below attached snapshots from Figure-18 shows the front-end of react application for registration page.

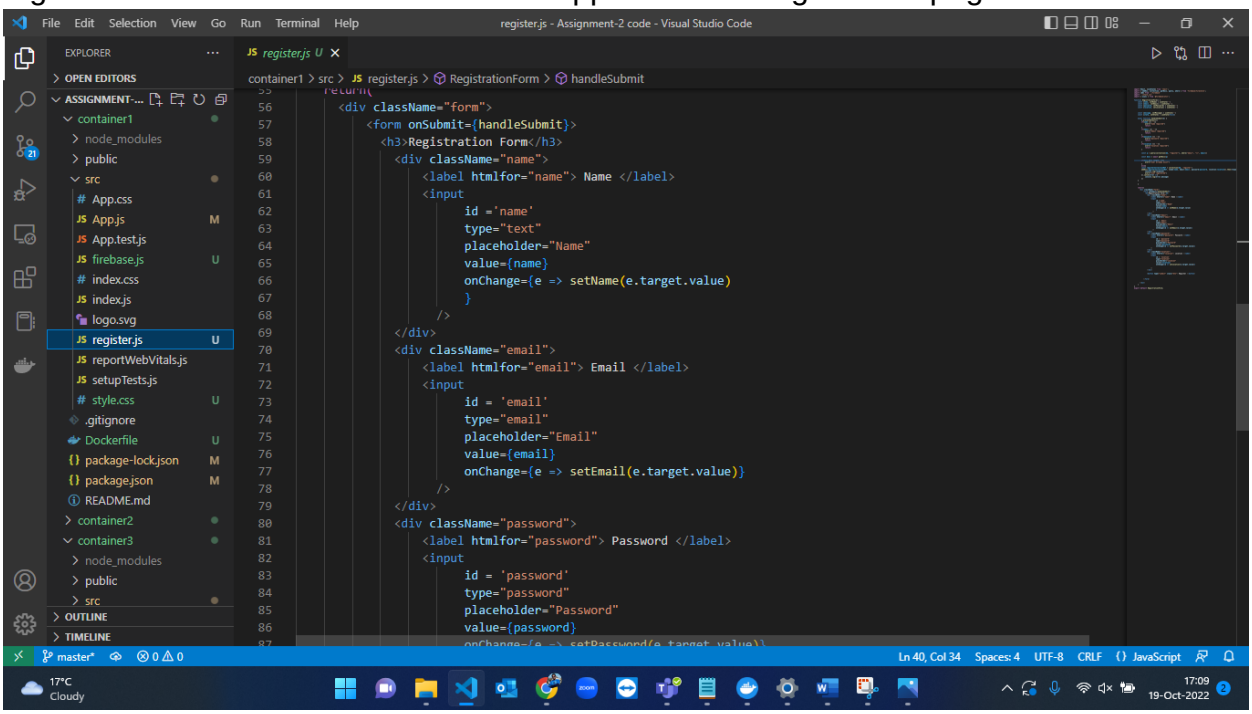

Figure-18: Front-end of registration form in react.

- After creating the front-end and backed of registration form, we have to start the web application. For that, command “npm install” is used. Figure 19 and Figure-20 represents the execution of the command to start the react app.

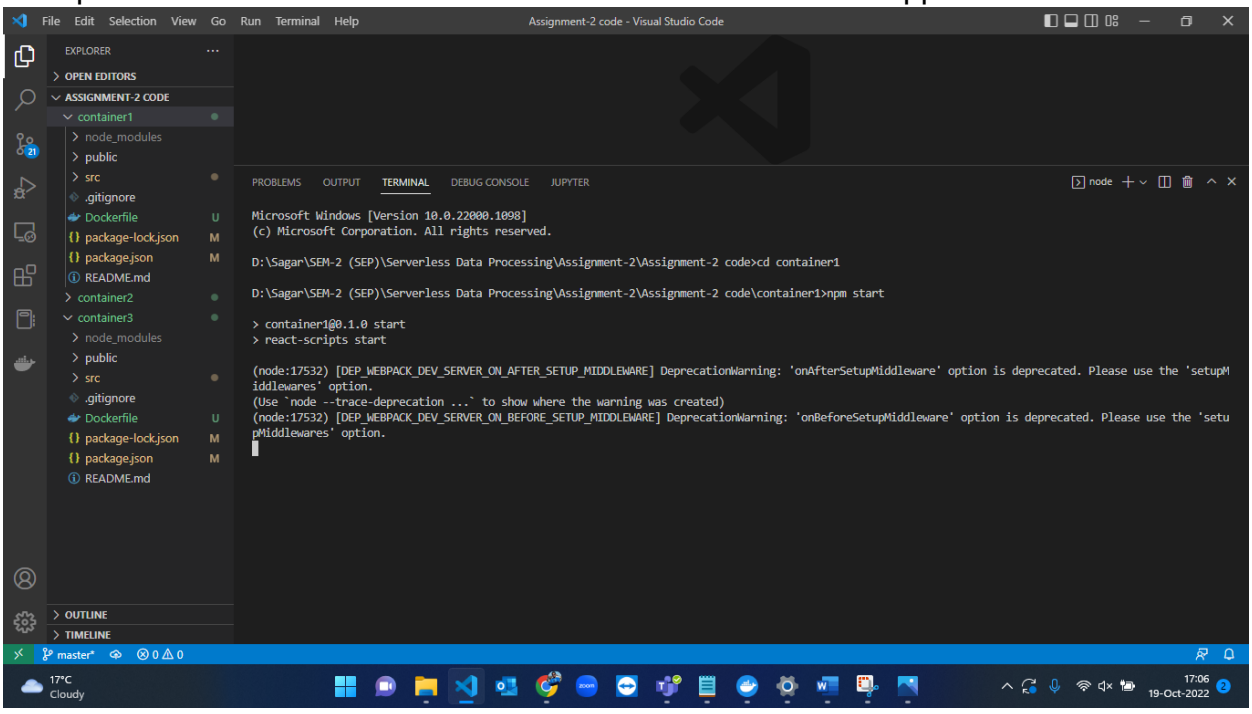

Figure-19 Command to run web app of conatiner1(app1) of registration page.

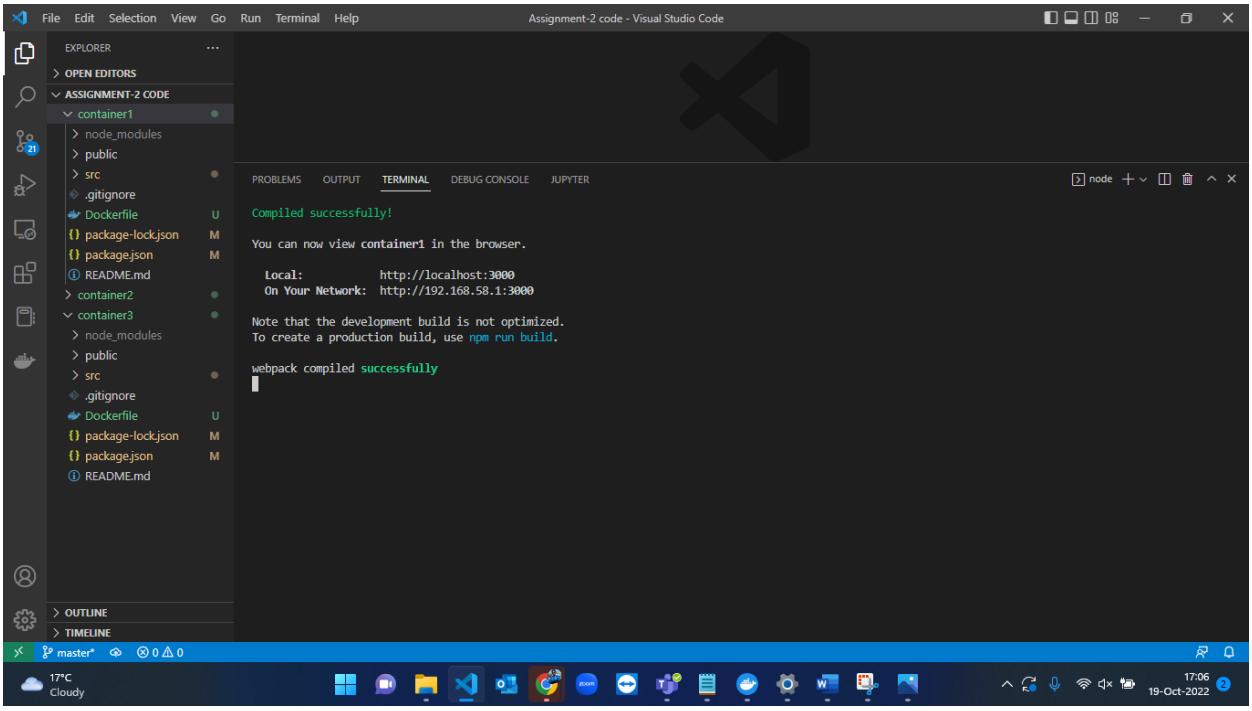

Figure-20 Successfully launch of registration page.

- The registration page is launched at localhost:3000. The user have to fill all the fields mandatory to register. The test cases are shown in the later section of the this document. Figure-21 to Figure-24 shows registration process by the user.

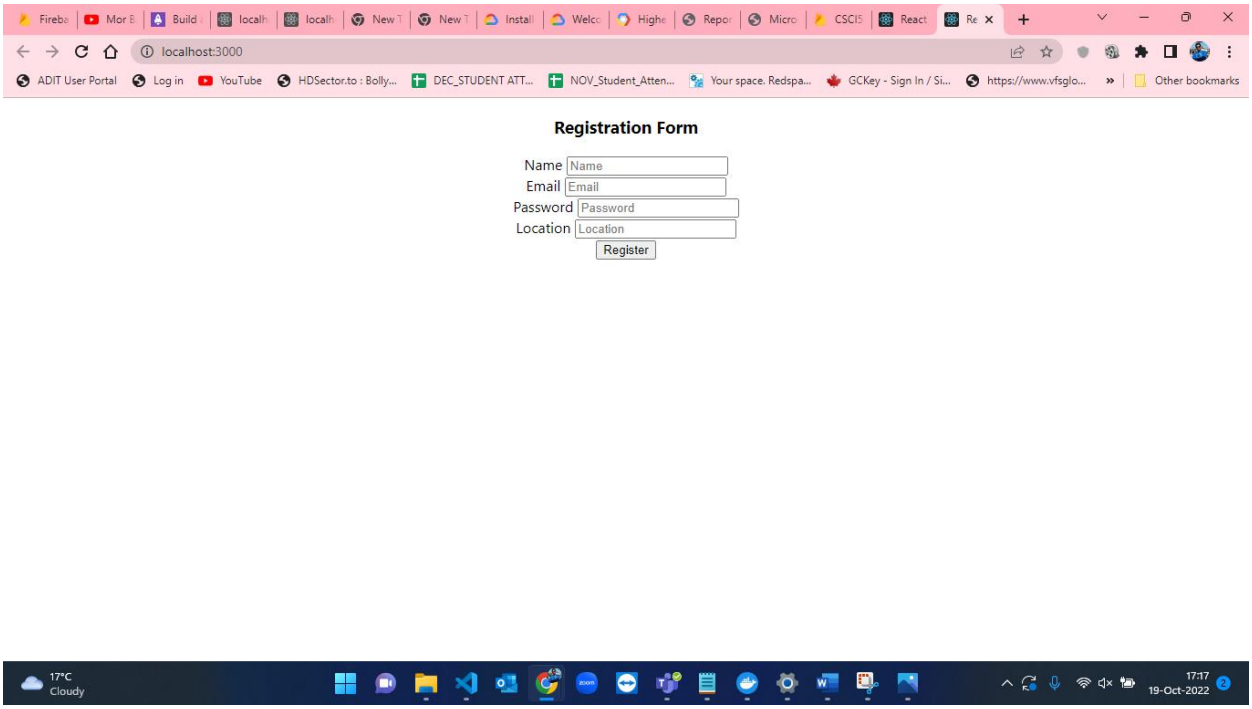

Figure-21: Blank page for registration.

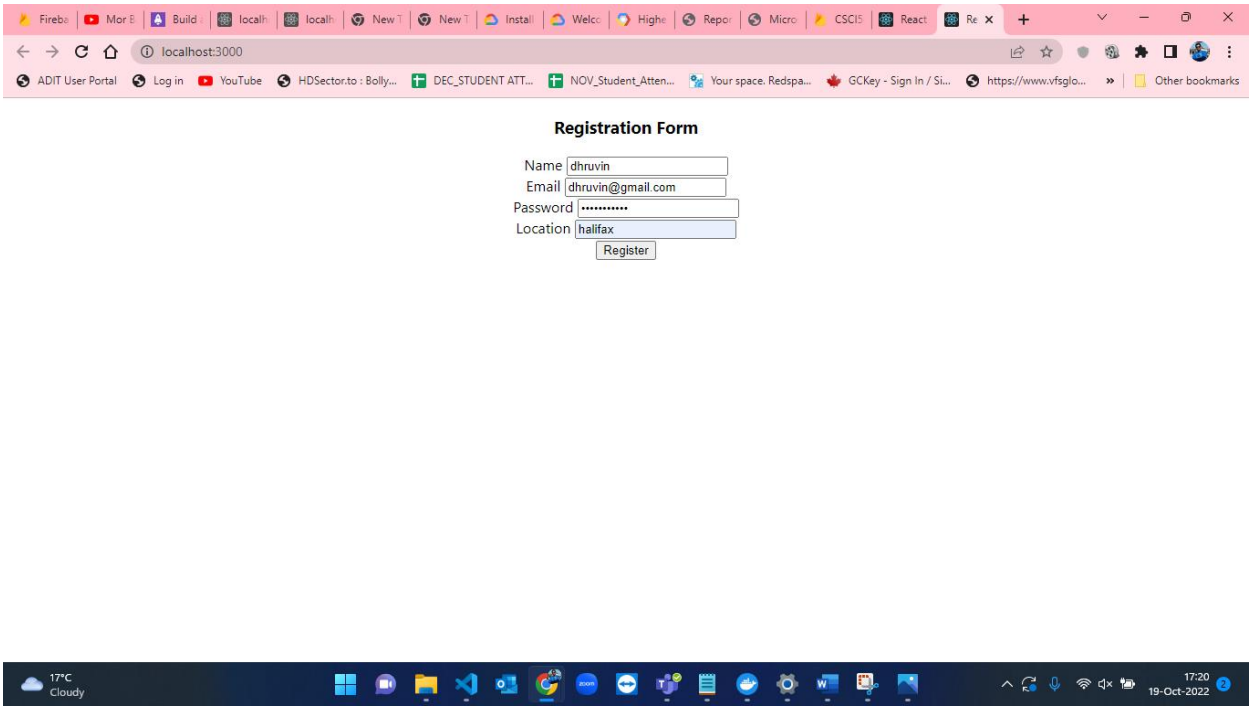

Figure-22: User fills the details in the registration form.

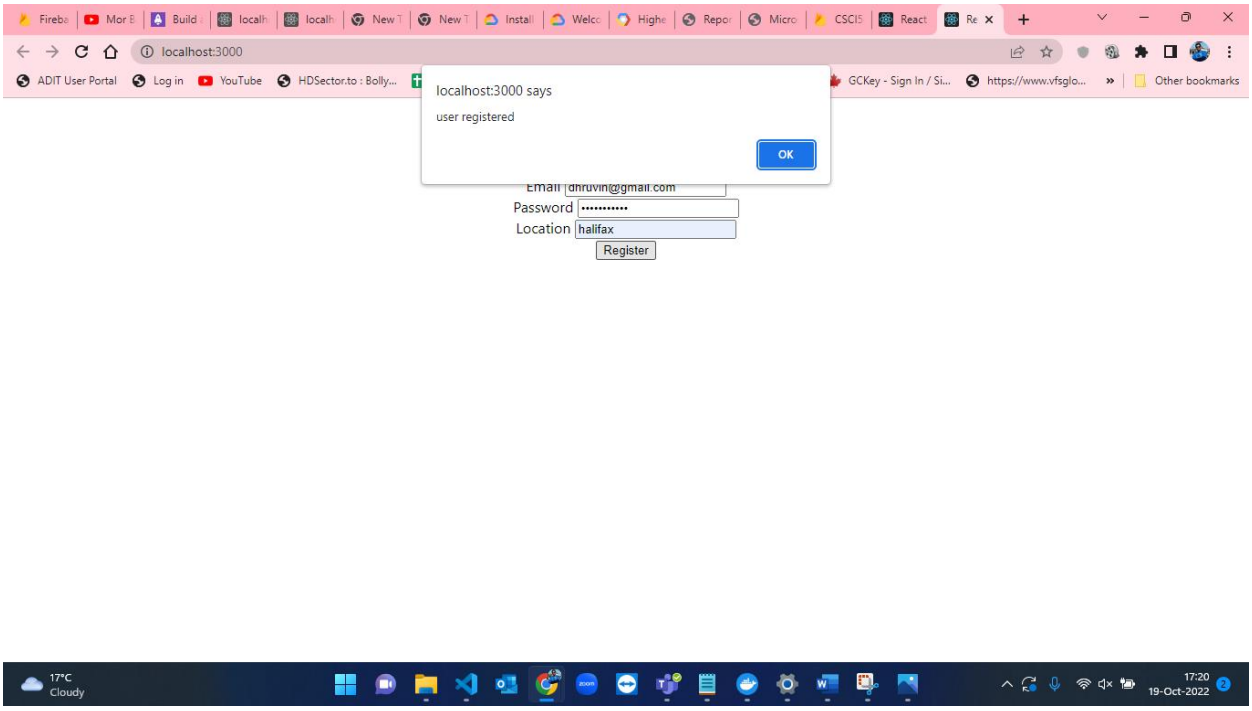

Figure-23: User registered successfully.

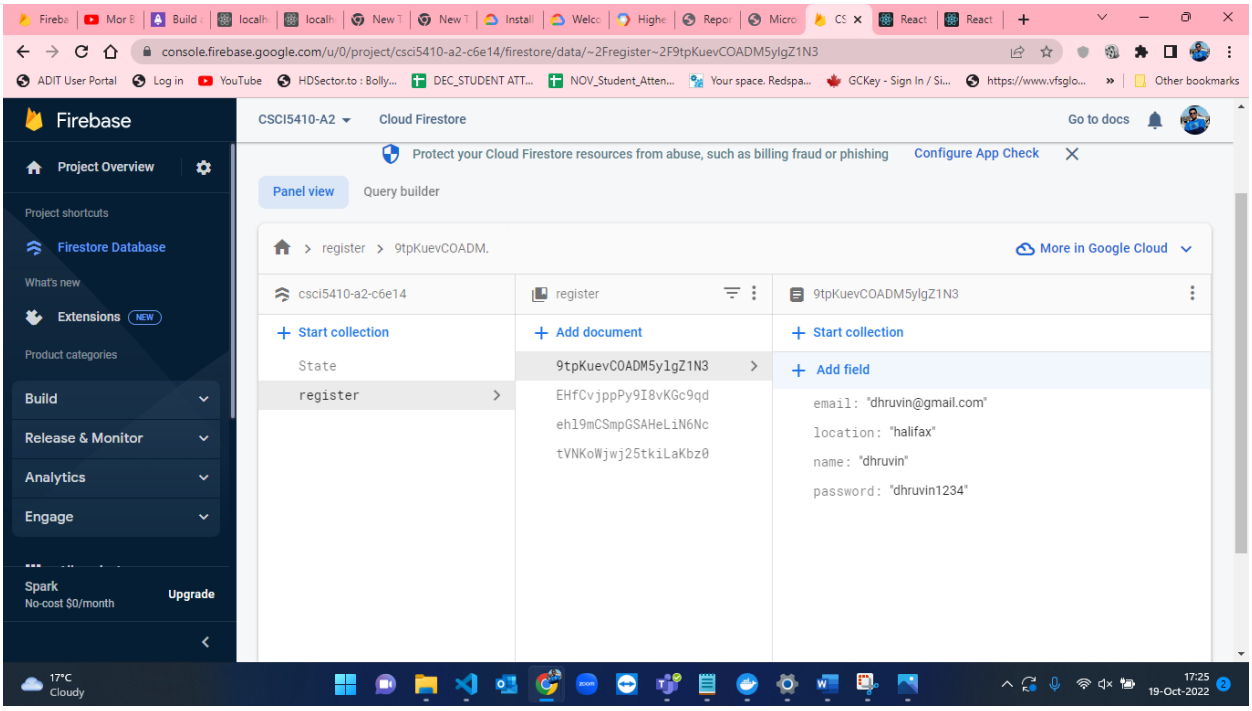

Figure-24: Database entry for the user registration.

- Figure-25 to Figure-32 shows the creation of new project in Google Cloud Console and enabling the Artifact Registry API for uploading docker image and hosting our containers on GCP.

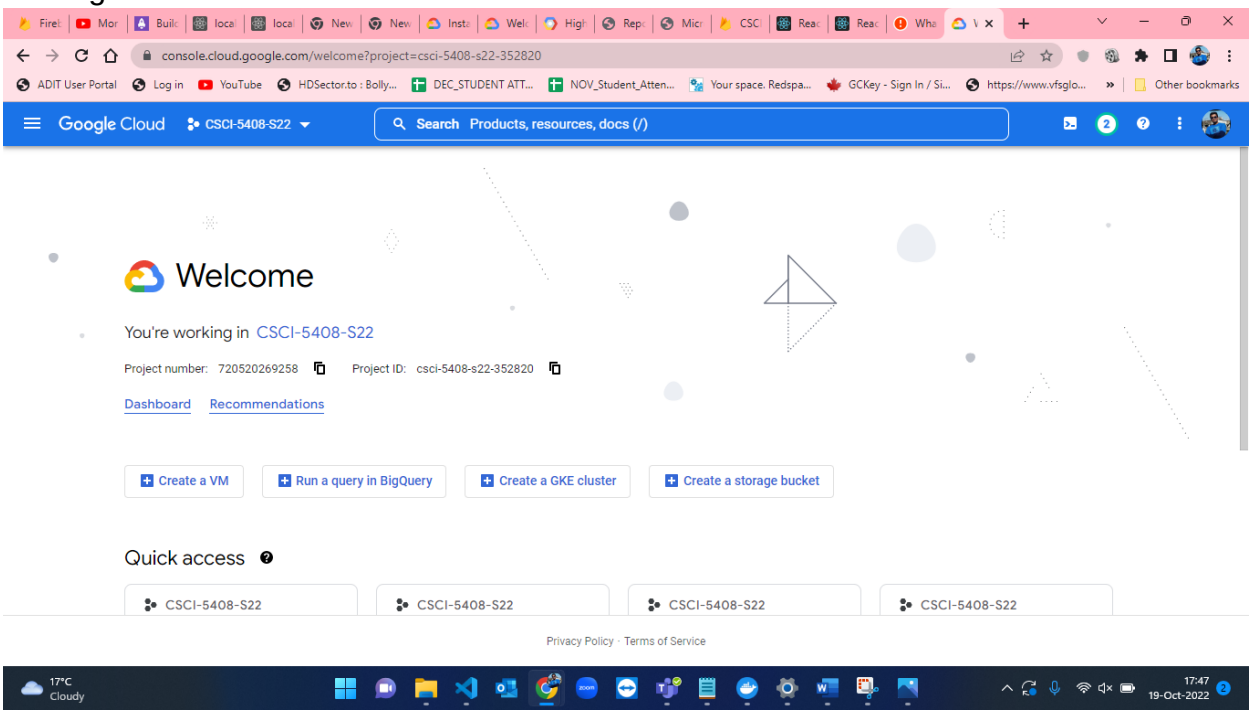

Figure-25: GCP Dashboard.

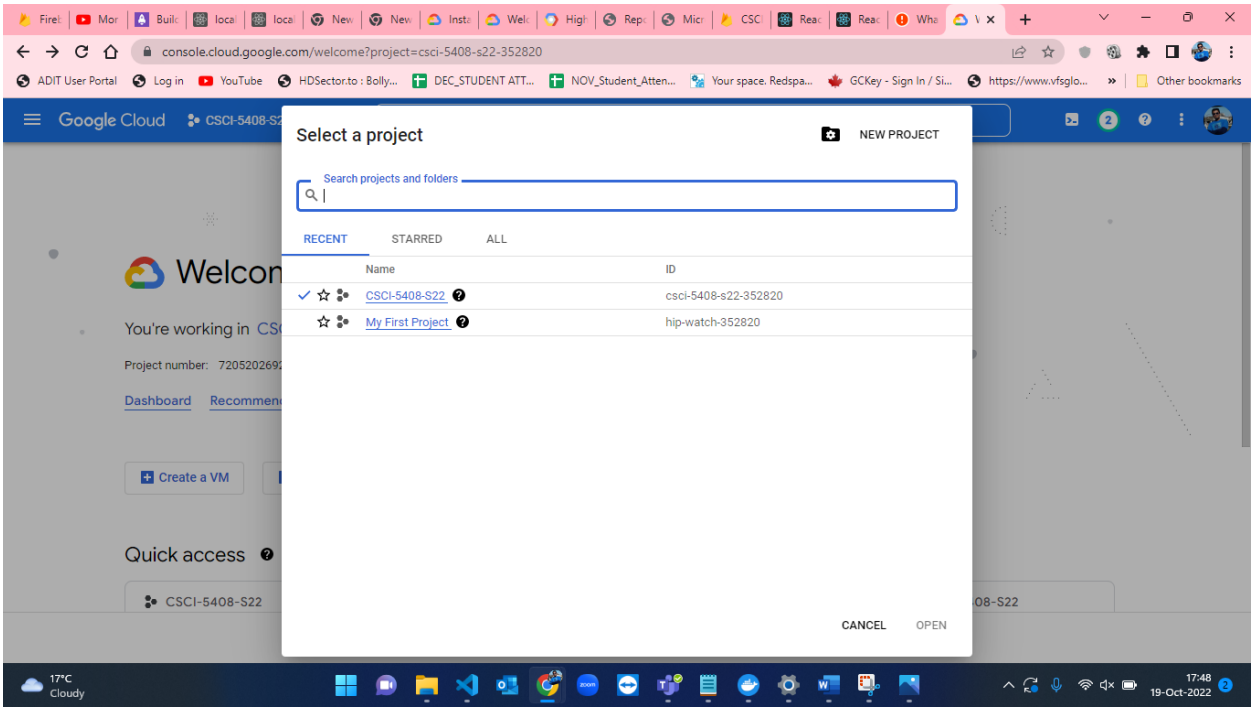

Figure-26: Google Cloud Platform new project creation window.

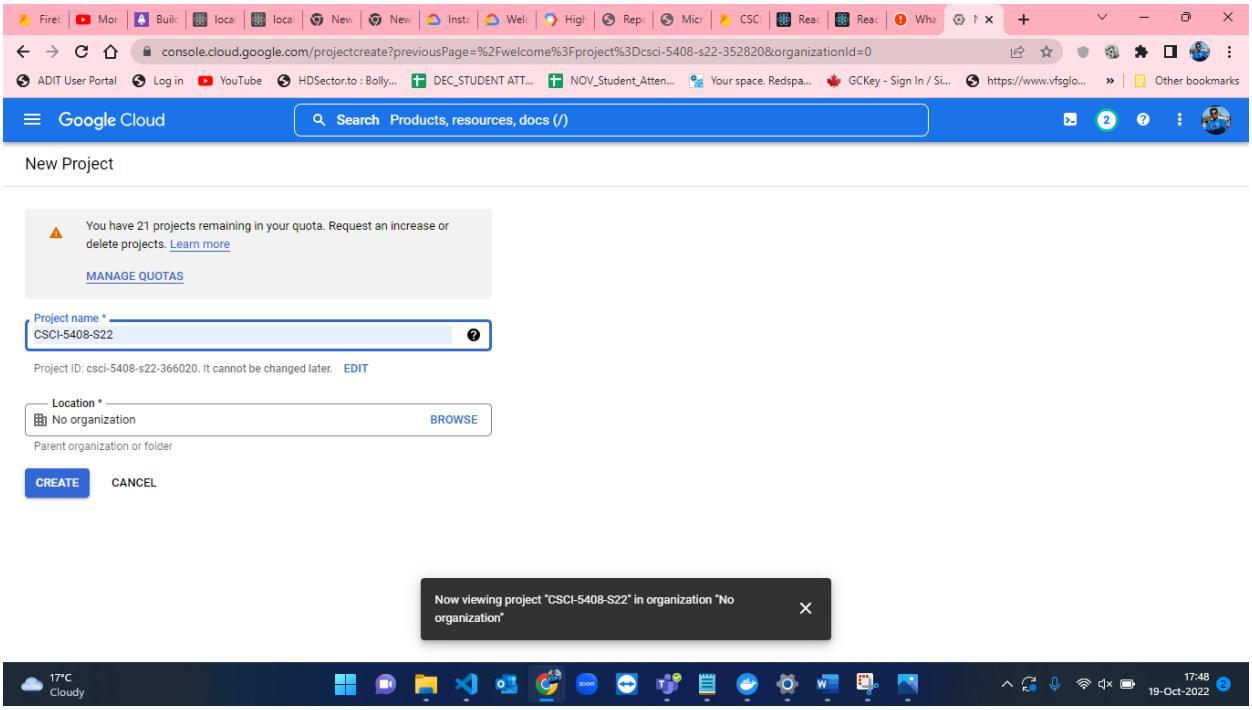

Figure-27: Setting project name for creating new project.

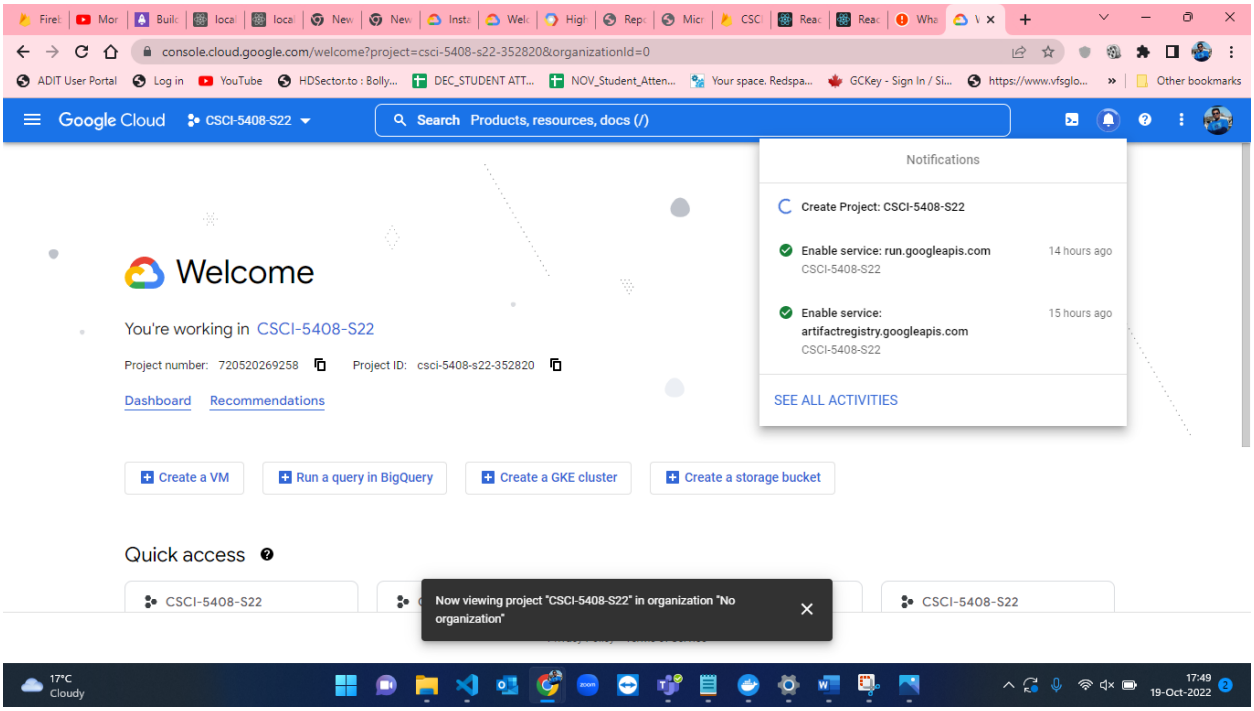

Figure-28: Successfully creation of new project in GCP.

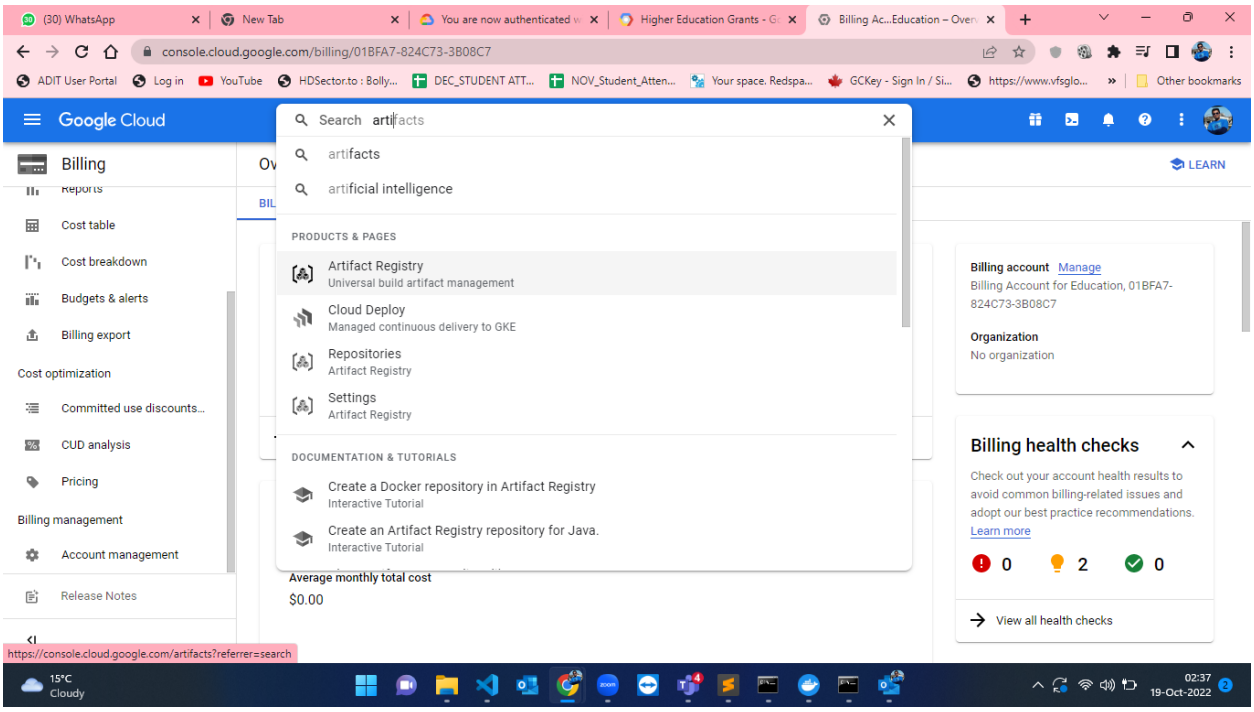

Figure-29: Artifact registry API search in GCP from searchbar.

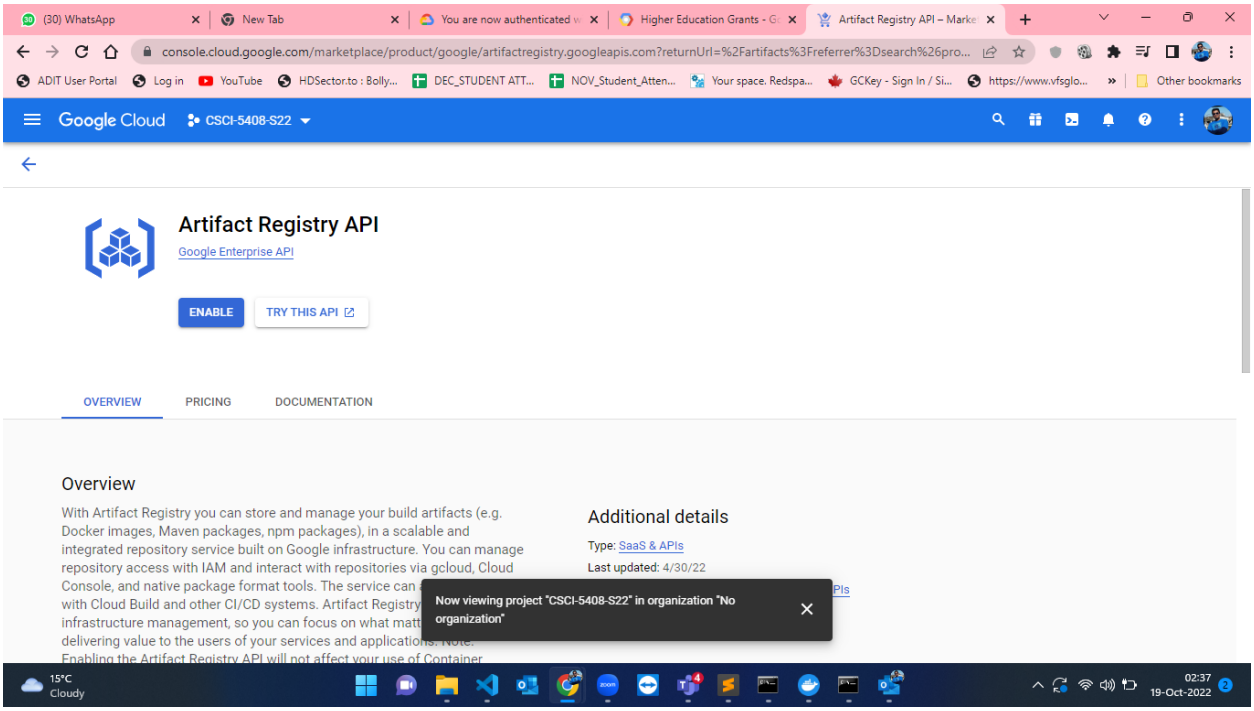

Figure-30: Artifact Registry API page in GCP.

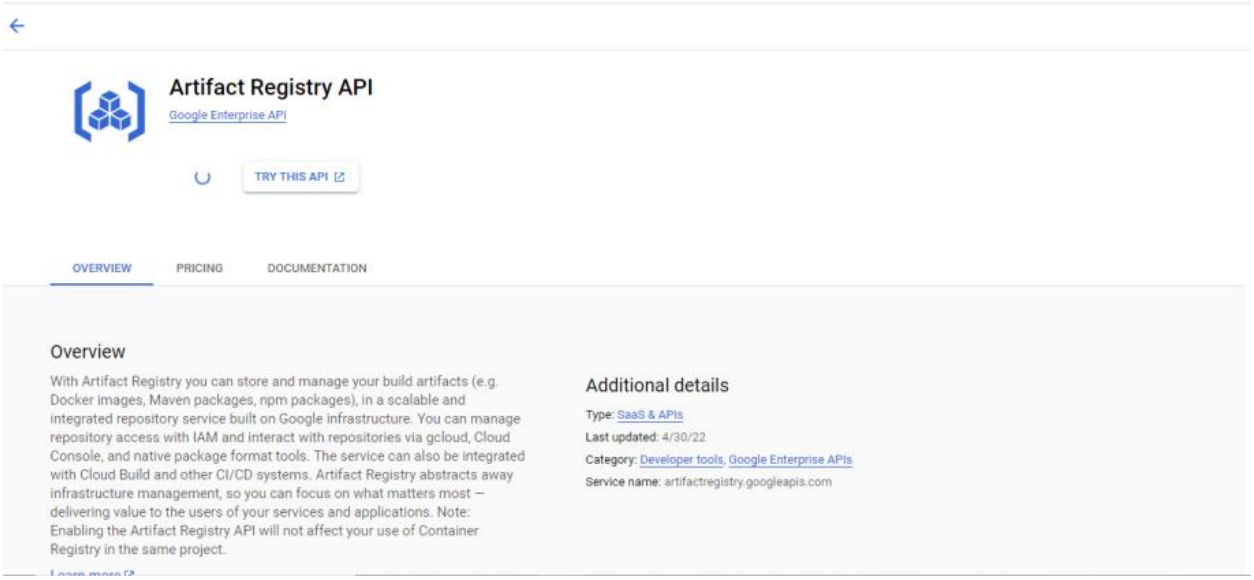

Figure-31: Enabling Artifact Registry API on GCP.

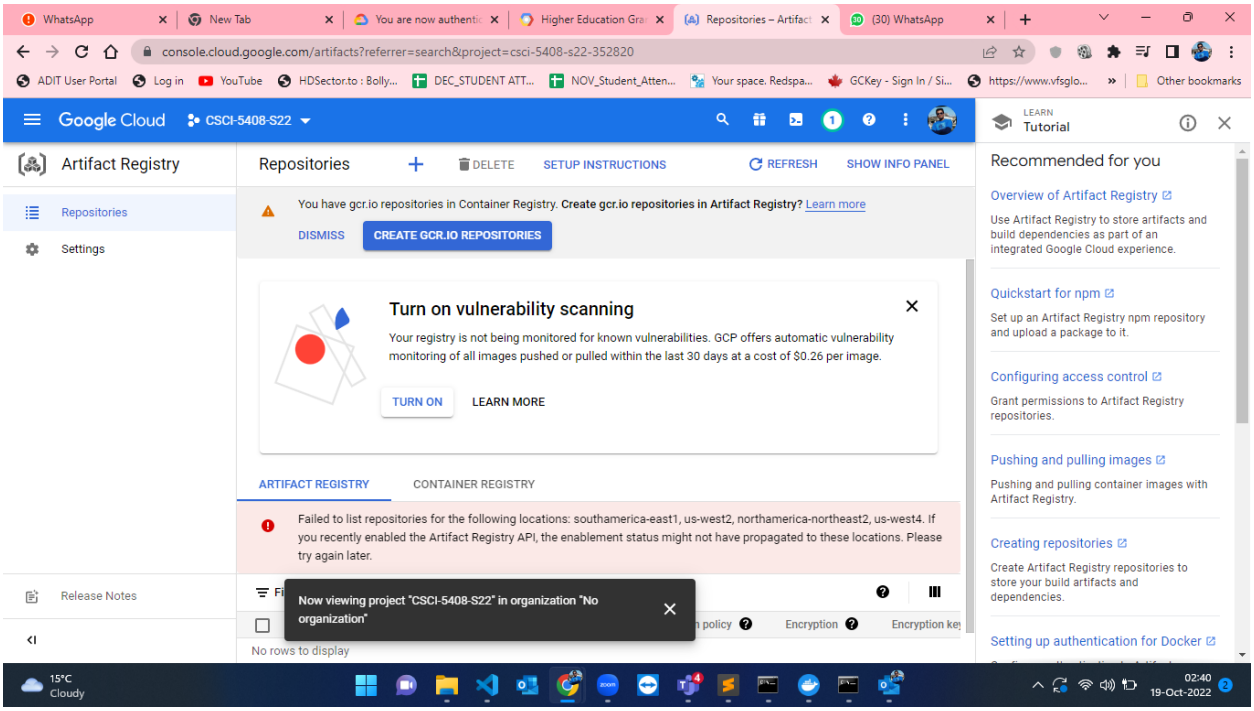

Figure-32: Artifact Registry API dashboard in GCP.

- As soon as the Artifact Registry API is enabled, we have to install Google Cloud CLI on the host to create and manage google cloud resources. I have downloaded setup for google cloud CLI[2]. The below attached snapshots from Figure-33 to Figure-39 shows the installation of google cloud CLI by setup.

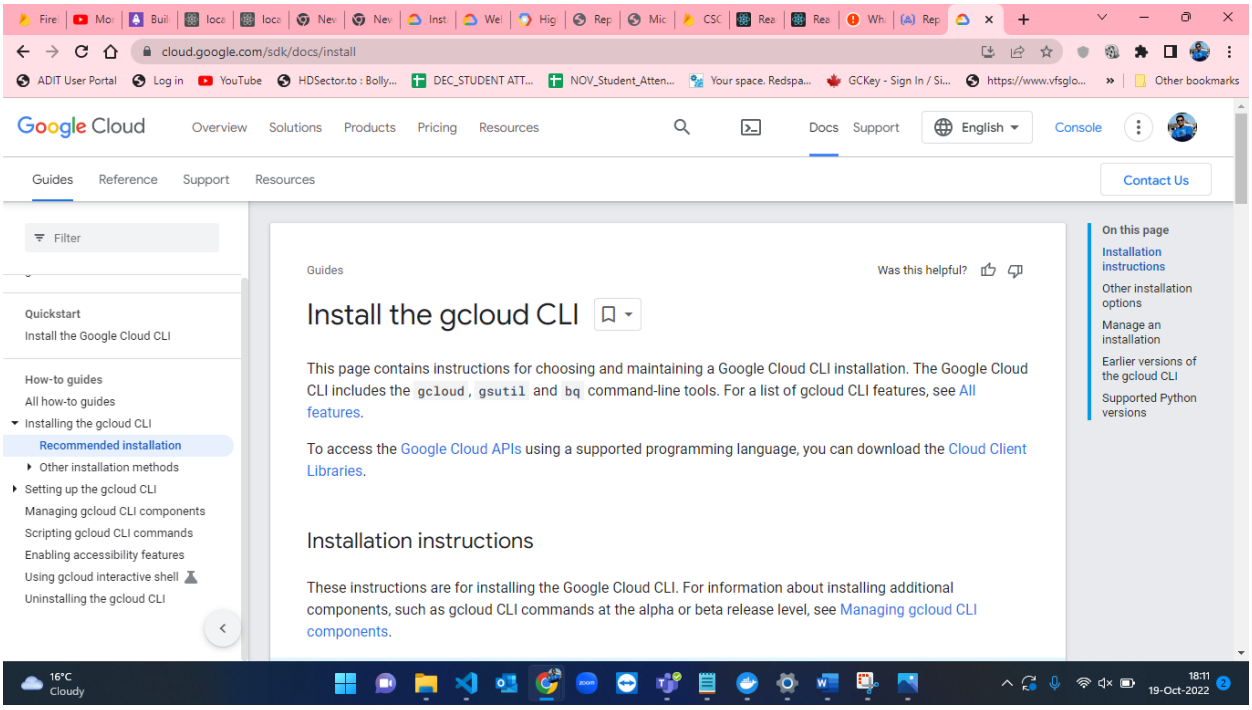

Figure-33: Gcloud CLI homepage.

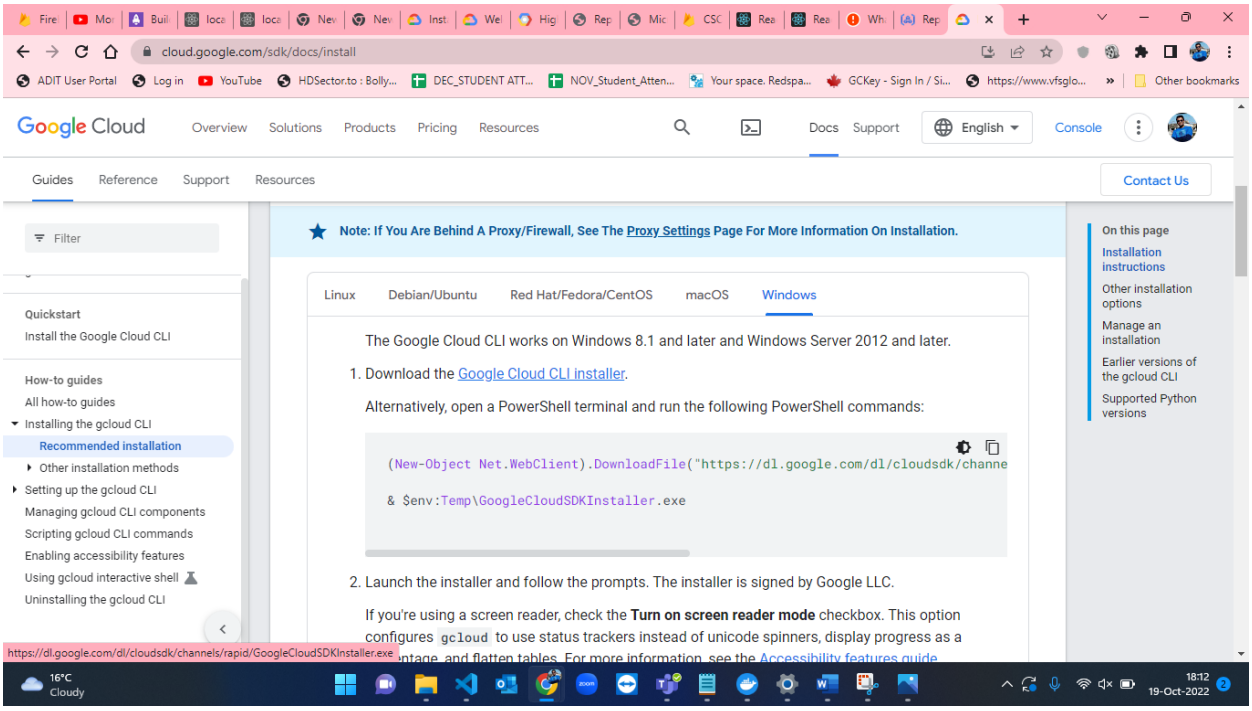

Figure-34: Gcloud CLI installer.

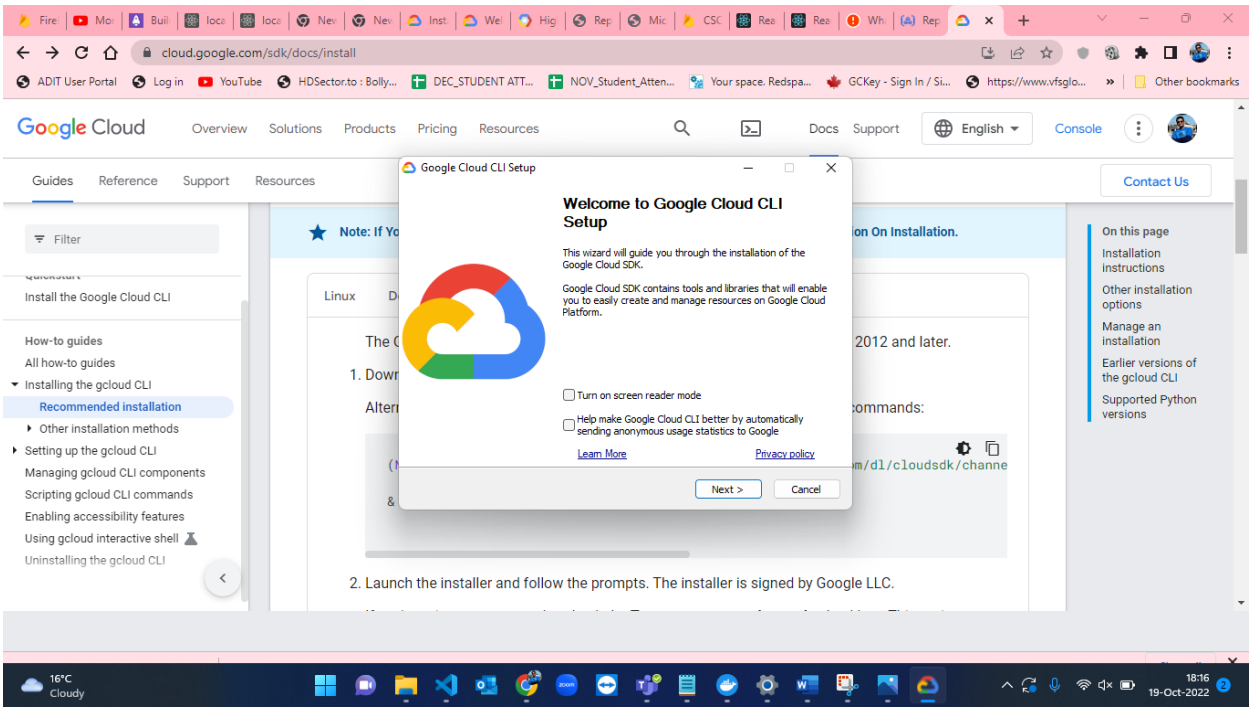

Figure-35: Gcloud installation Step-1.

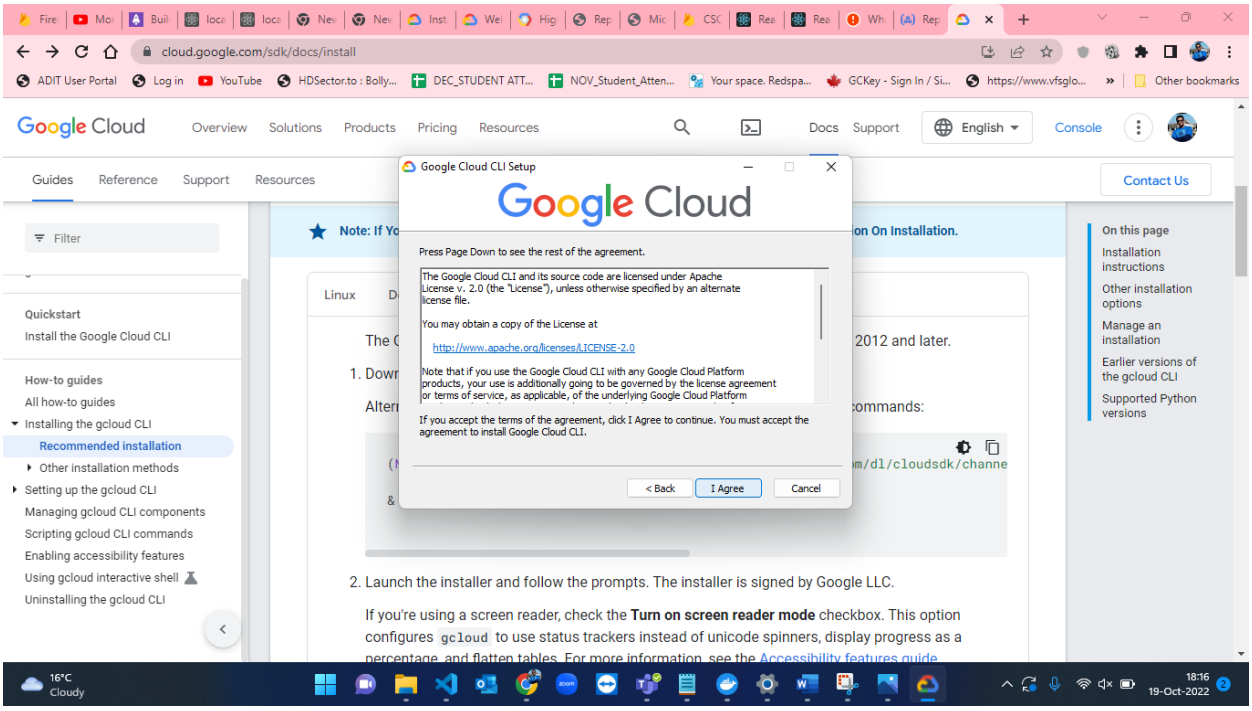

Figure-36: Gcloud installation Step-2.

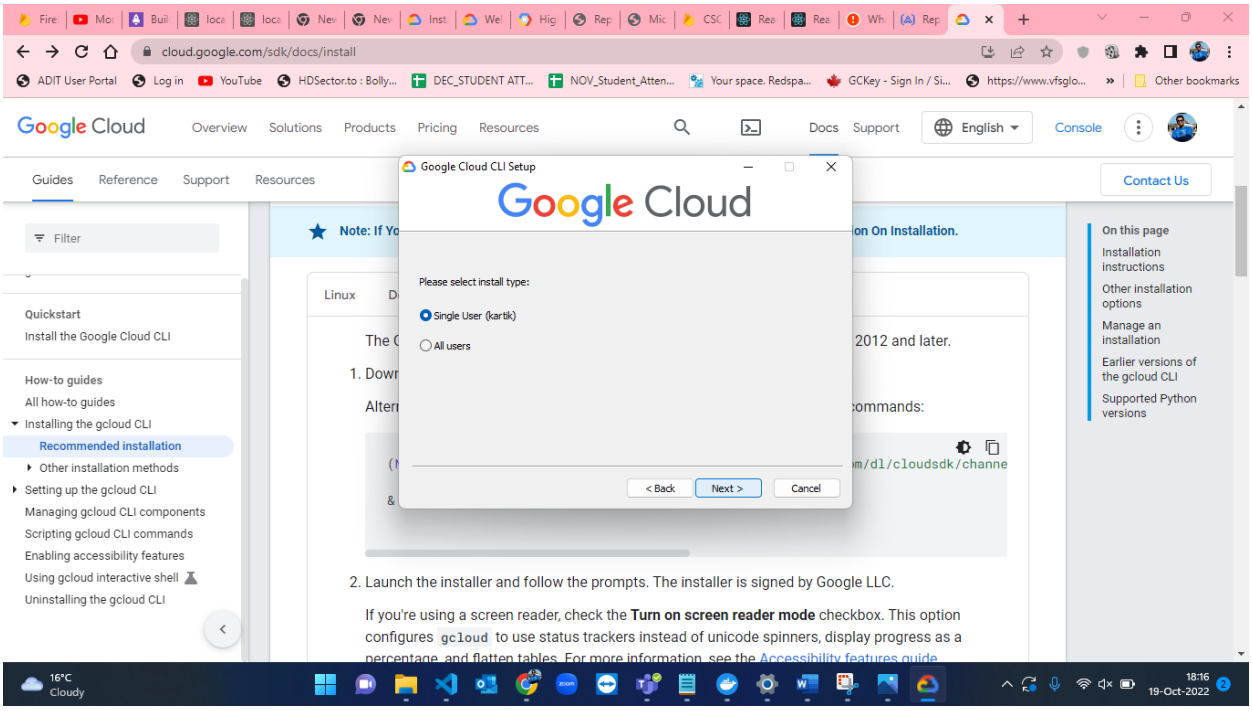

Figure-37: Gcloud installation Step-3.

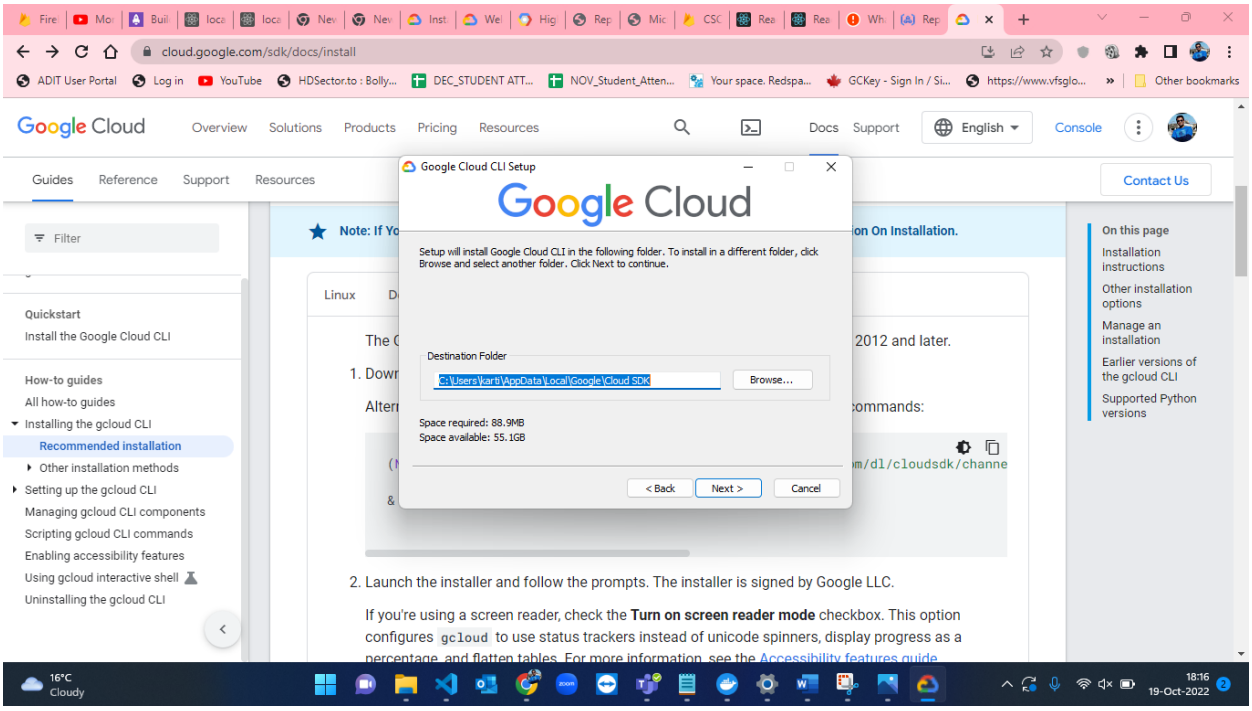

Figure-38: Gcloud installation Step-4.

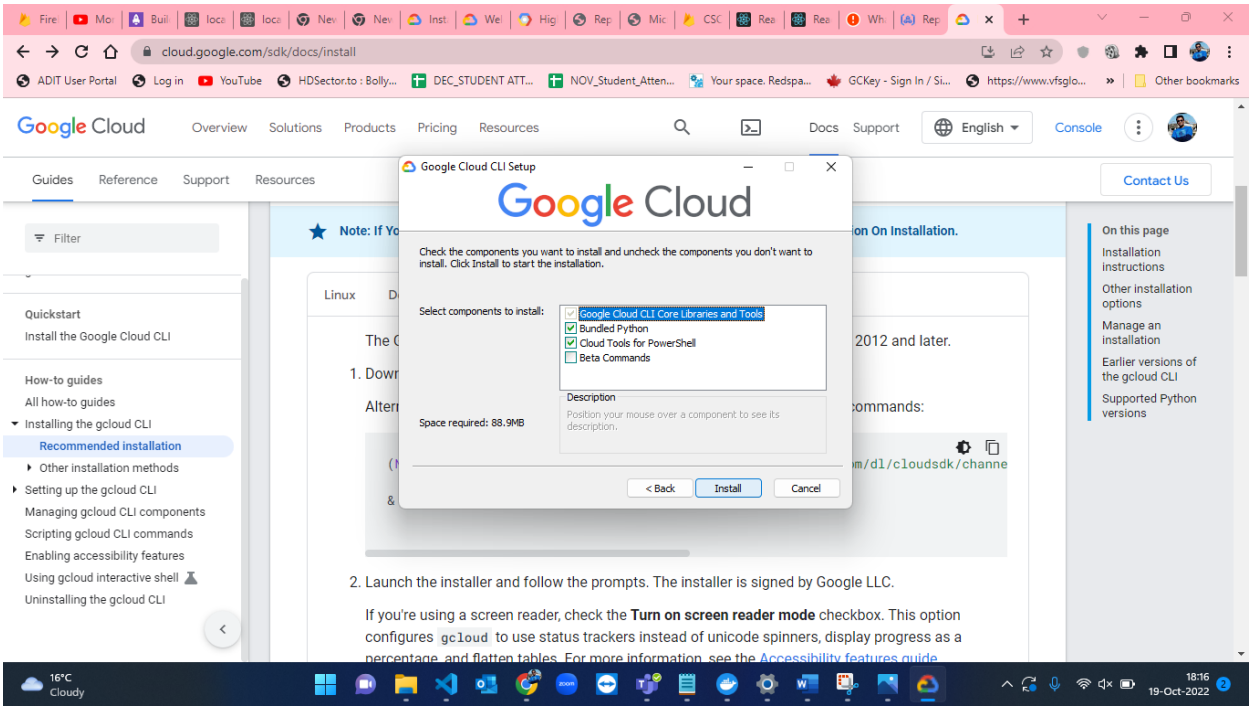

Figure-39: Gcloud installation Step-5.

- After installation of Gcloud CLI, the containers will have to be pushed from local machine to Artifact Repository and then we have create a service to host file. I have named the repository as “serverless-asignment” and selected the region as “us-central1 (lowa)”. Figure-40 and Figure-41 shows the creation of repository in Artifact registry.

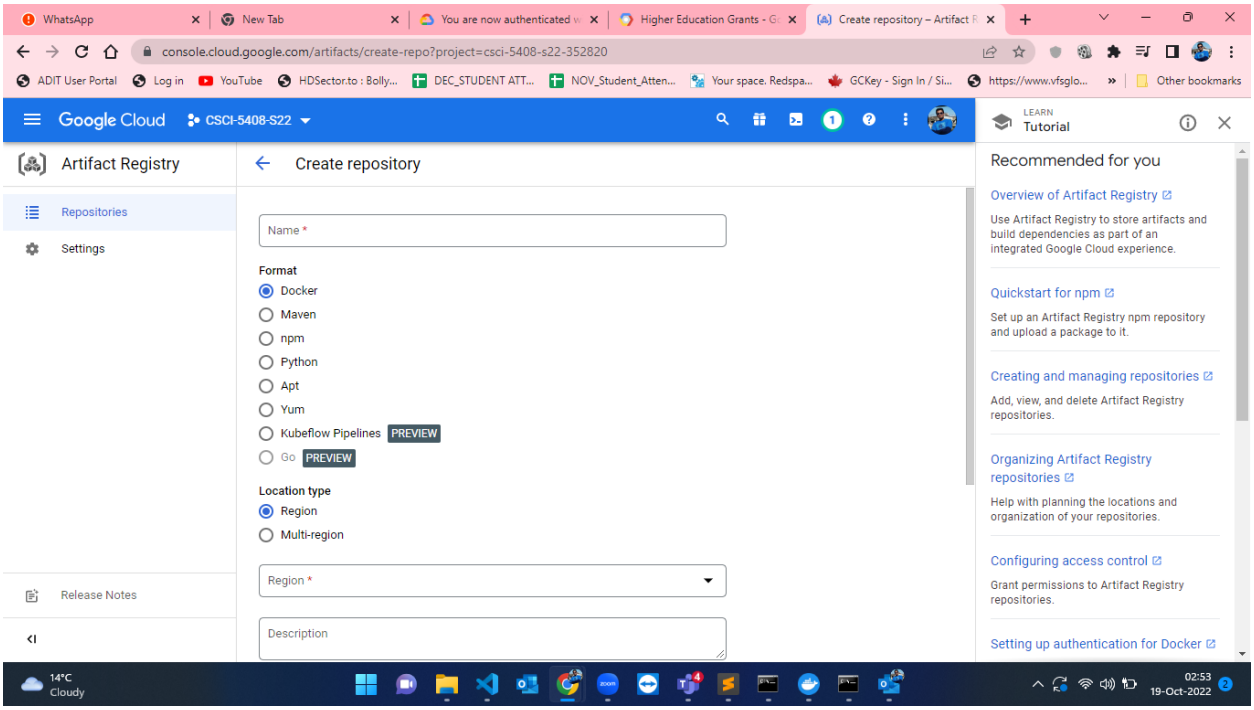

Figure-40: Repository creation in Artifact registry in GCP.

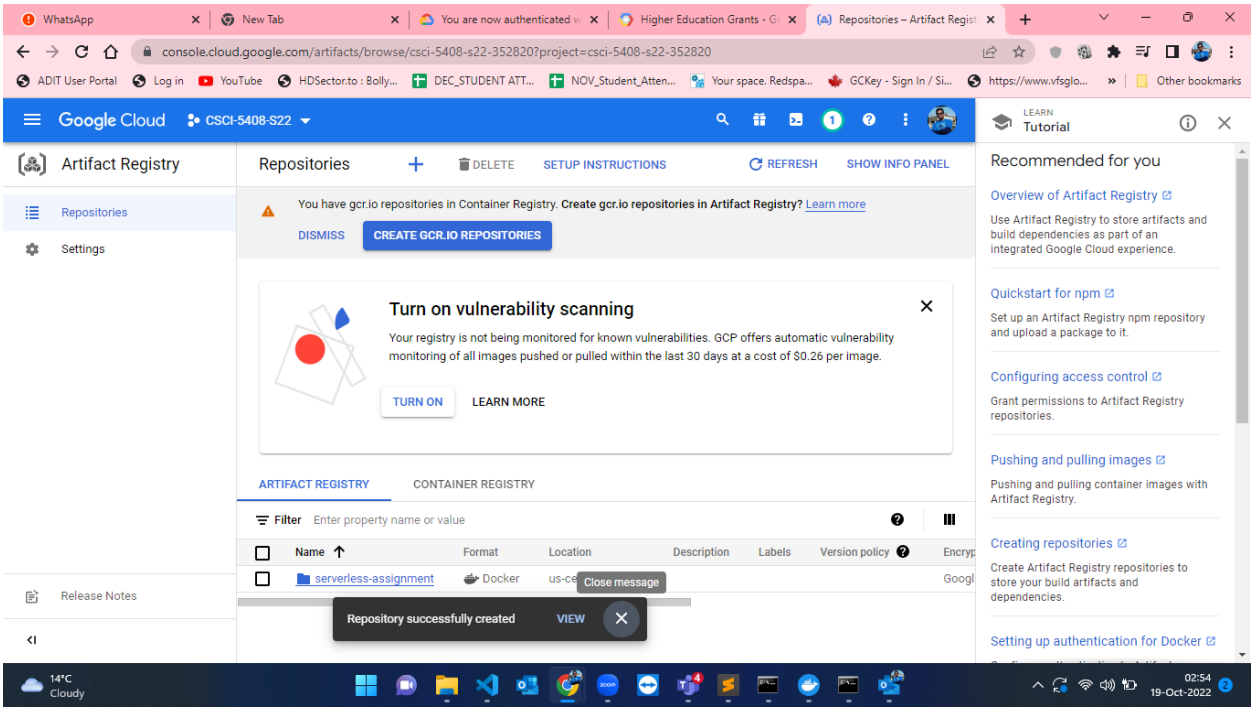

Figure-41: Created repository named as “serverless-assignment”.

- As the repository is created, we have to set the Gcloud account configuration. I have initialized it by using “gcloud init” and give my google account credentials to setup. The steps for this are shown in the below attached screenshots from Figure-42 to Figure-49.

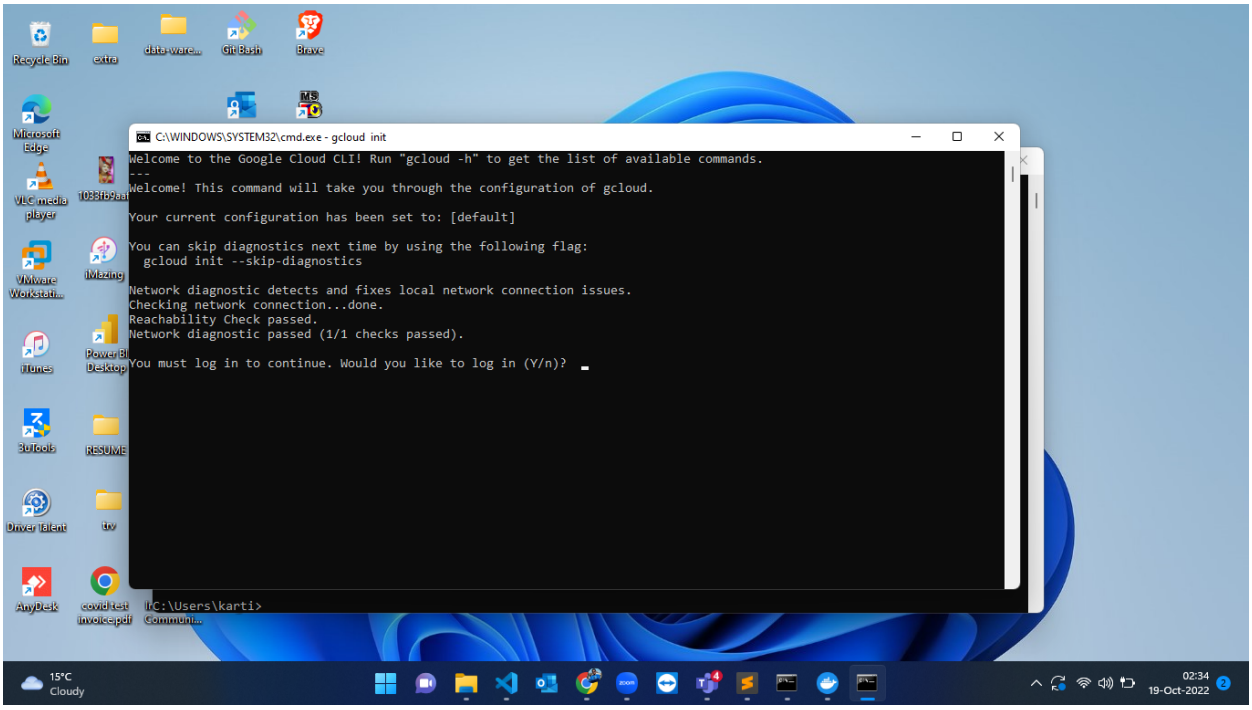

Figure-42: Configuration of Gcloud CLI Step-1.

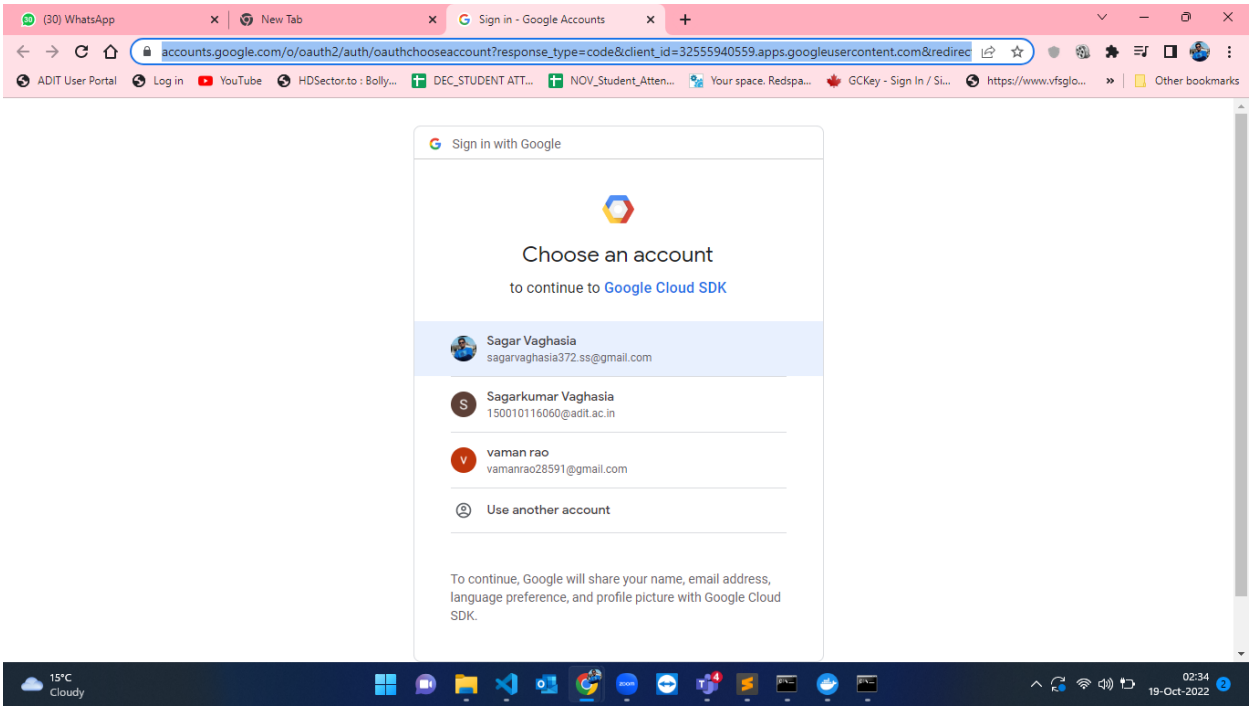

Figure-43: Configuration of Gcloud CLI Step-2.

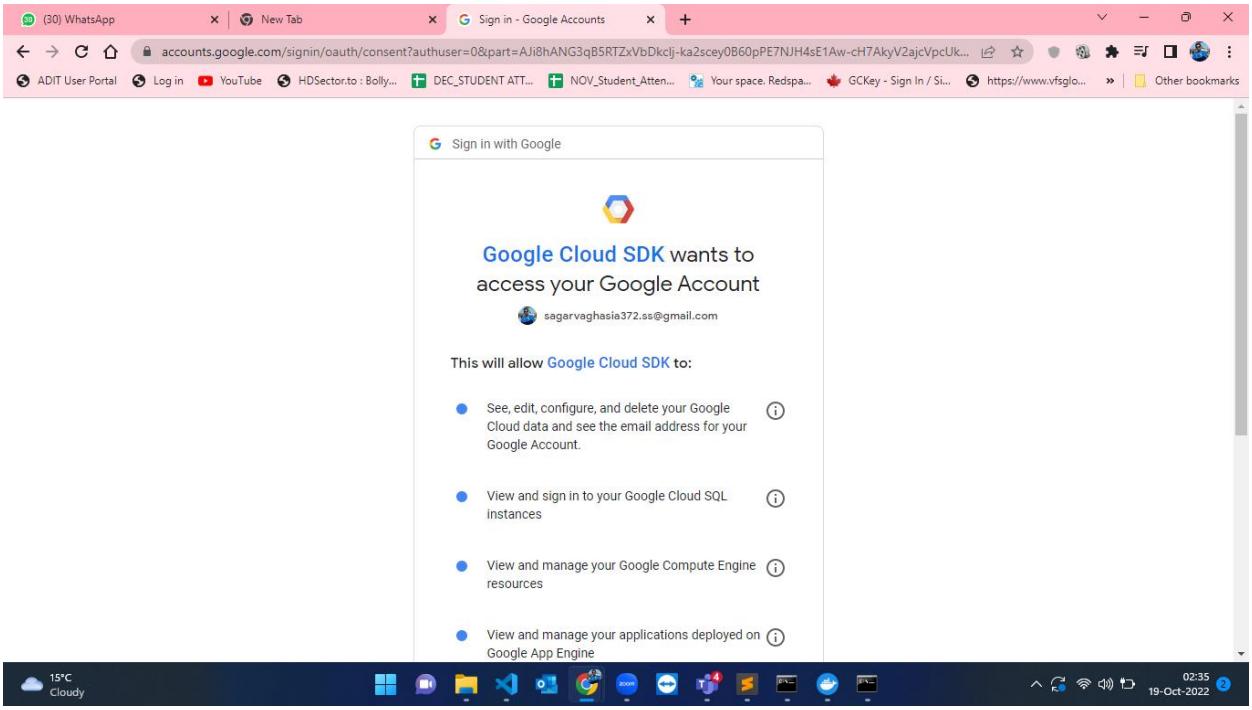

Figure-44: Configuration of Gcloud CLI Step-3.

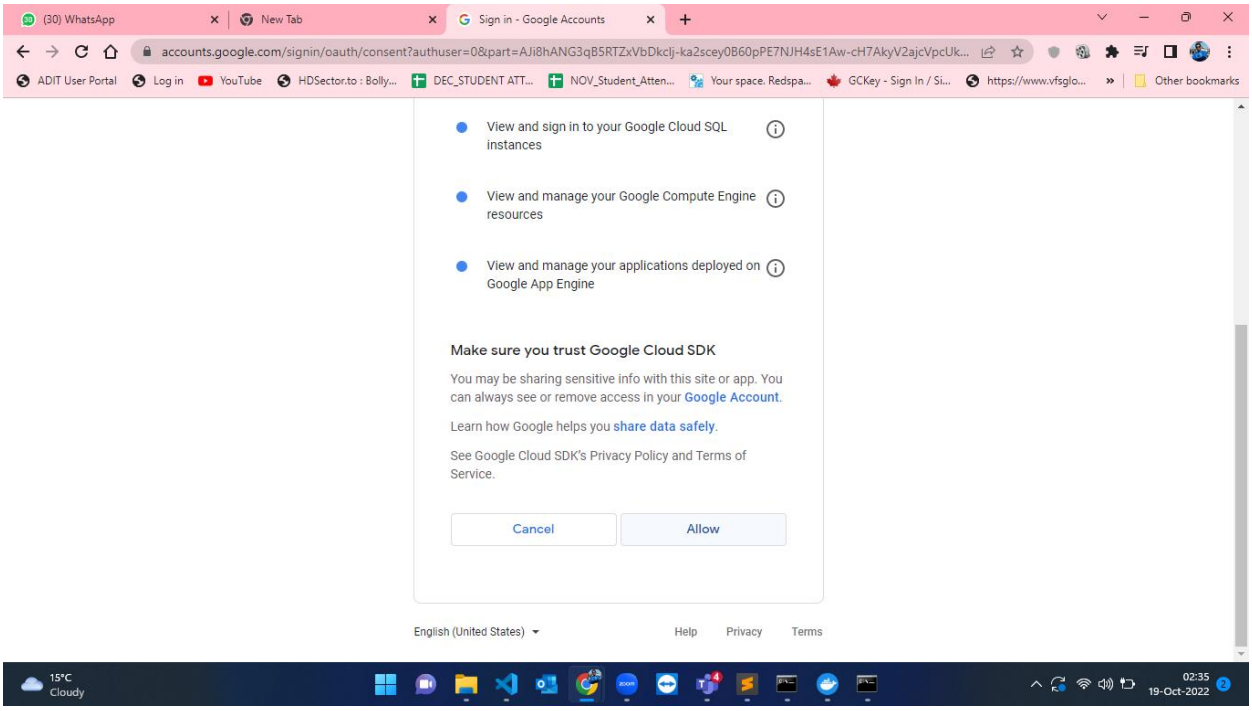

Figure-45: Configuration of Gcloud CLI Step-4.

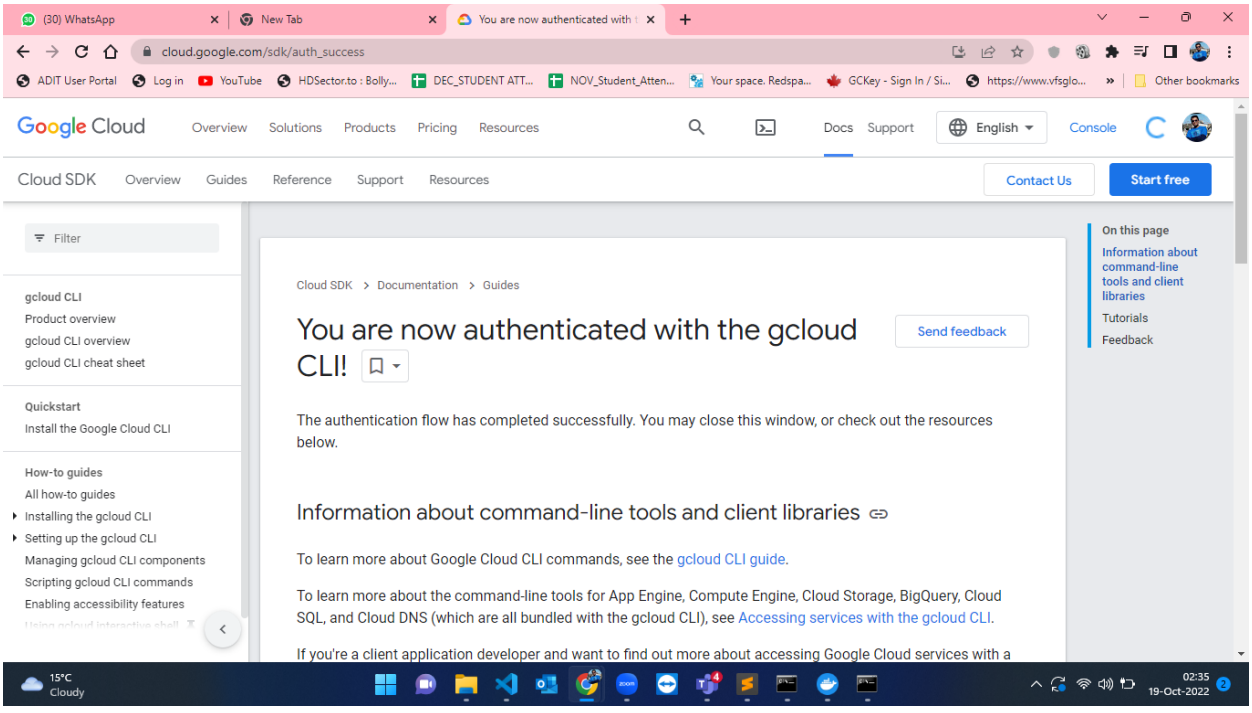

Figure-46: Configuration of Gcloud CLI Step-5.

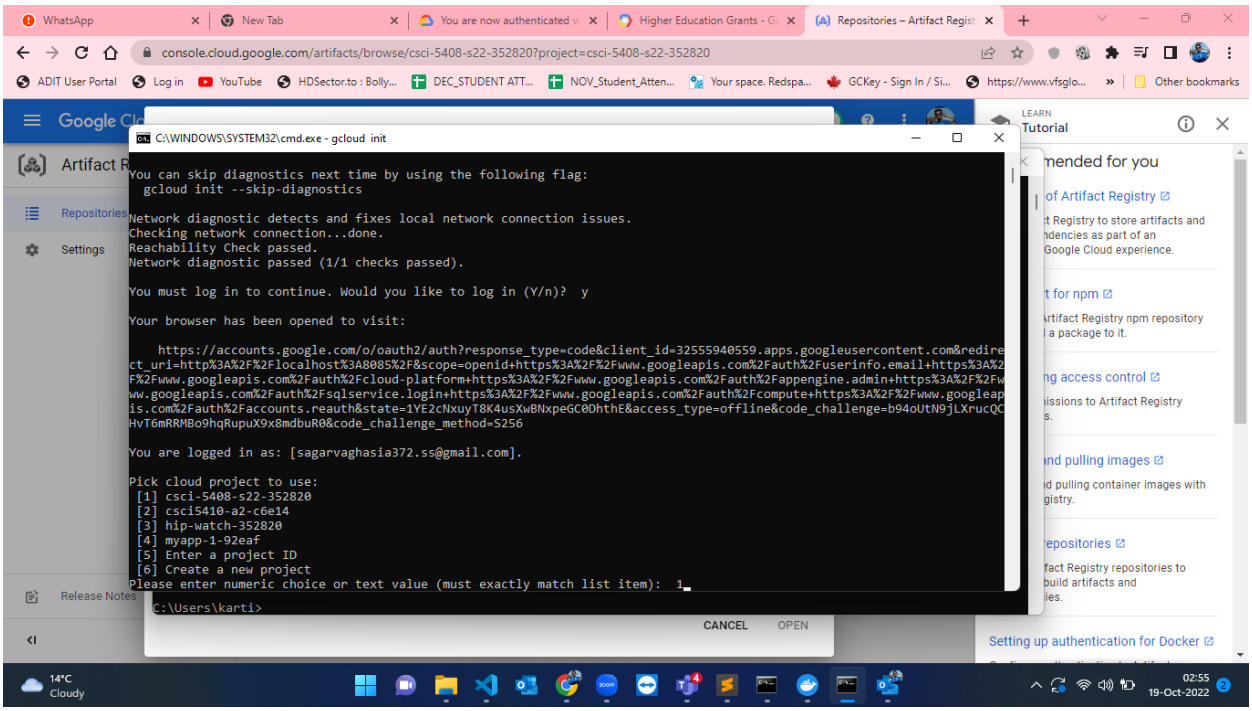

Figure-47: Configuration of Gcloud CLI Step-6.

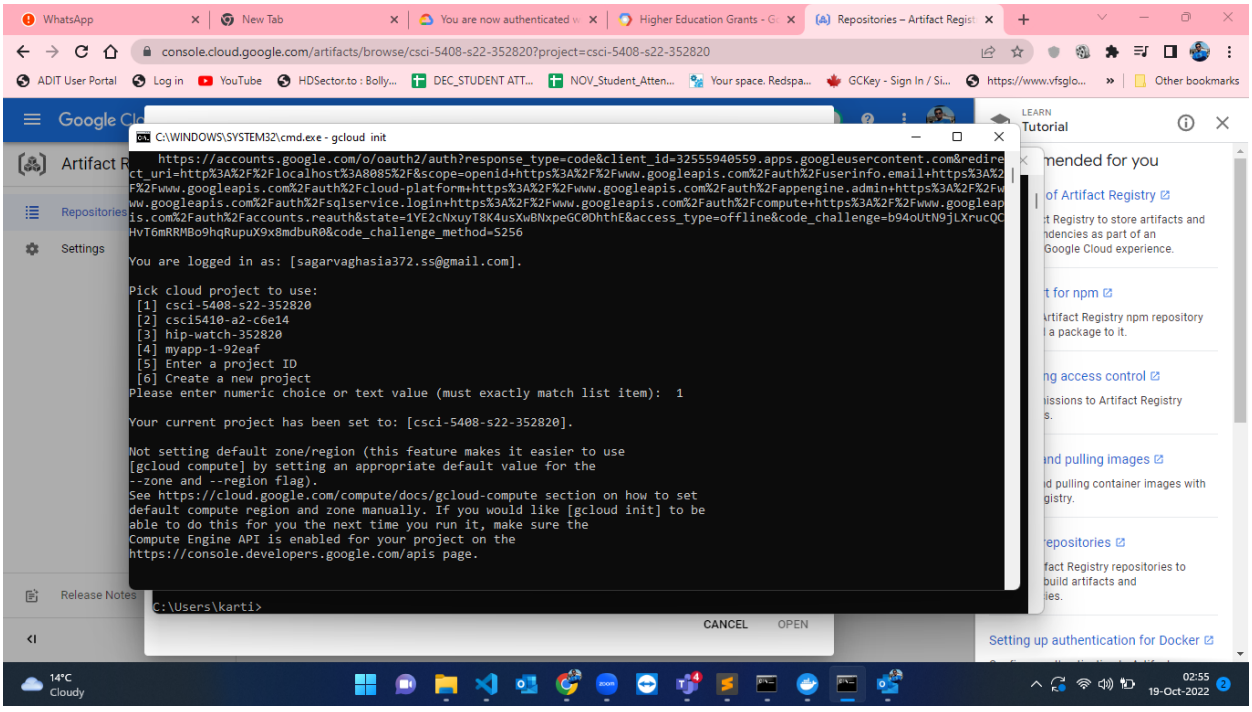

Figure-48: Configuration of Gcloud CLI Step-7.

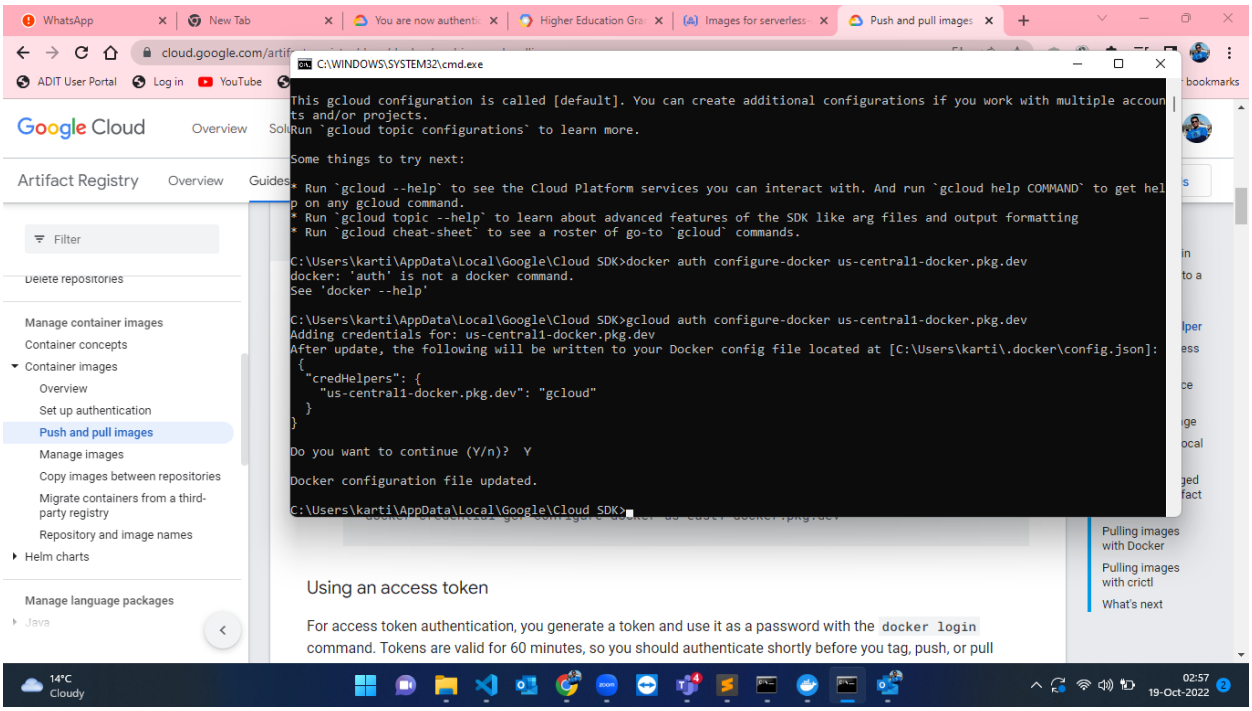

Figure-49: Configuration of Gcloud CLI Step-8.

- After completing the setup of Gcloud CLI in the machine, the tasks of creating docker image, pushing the code to repo and then running cloud run services for Registration have to be done. For building the docker image I have used the command “docker build -t container1 .”. After that for creating a docker image in the cloud repository I have used “docker tag conatiner1 us-central1-docker.pkg.dev/csci-5408-s22-352820/serverless-assignment/container1:latest”. After creating image for pushing the image to cloud repo I have used the command “docker push us-central1-docker.pkg.dev/csci-5408-s22-352820/serverless assignment/container1:latest”[6]. At, the end I have created cloud run service[7] and registered one user. The steps for the above process is shown below in the screenshots from Figure-50 to Figure-59

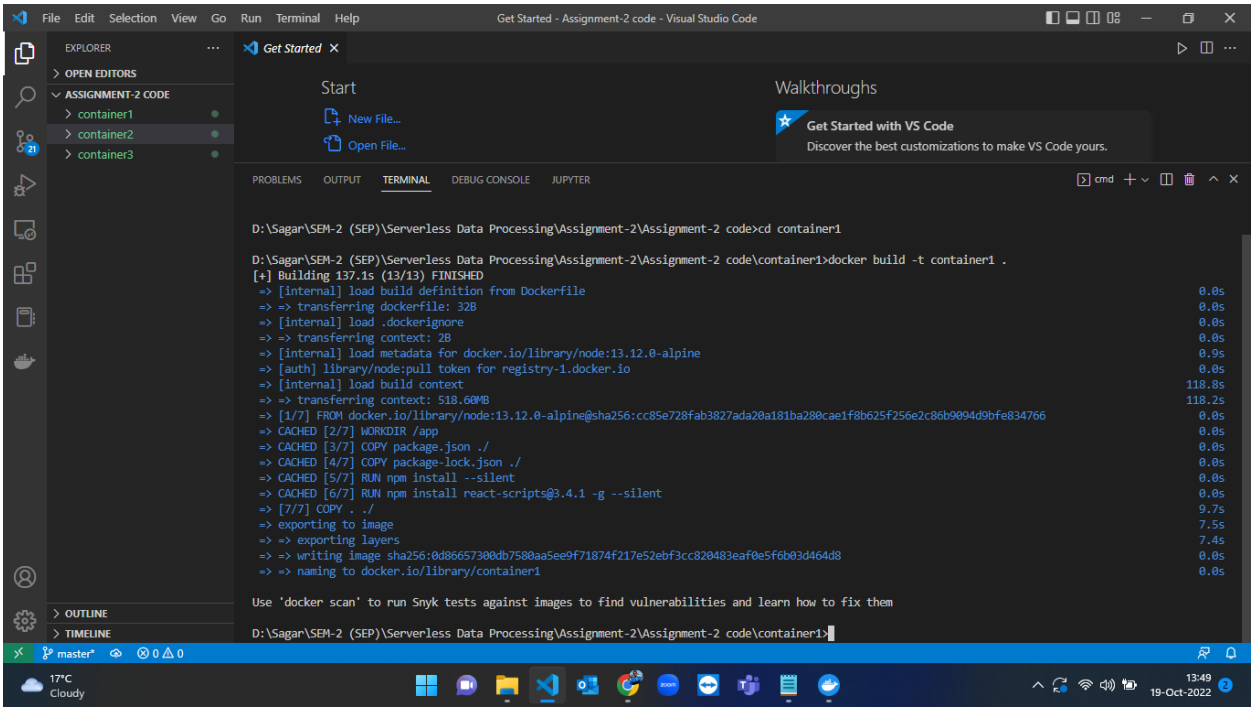

Figure-50: Building docker container1 for registration app.

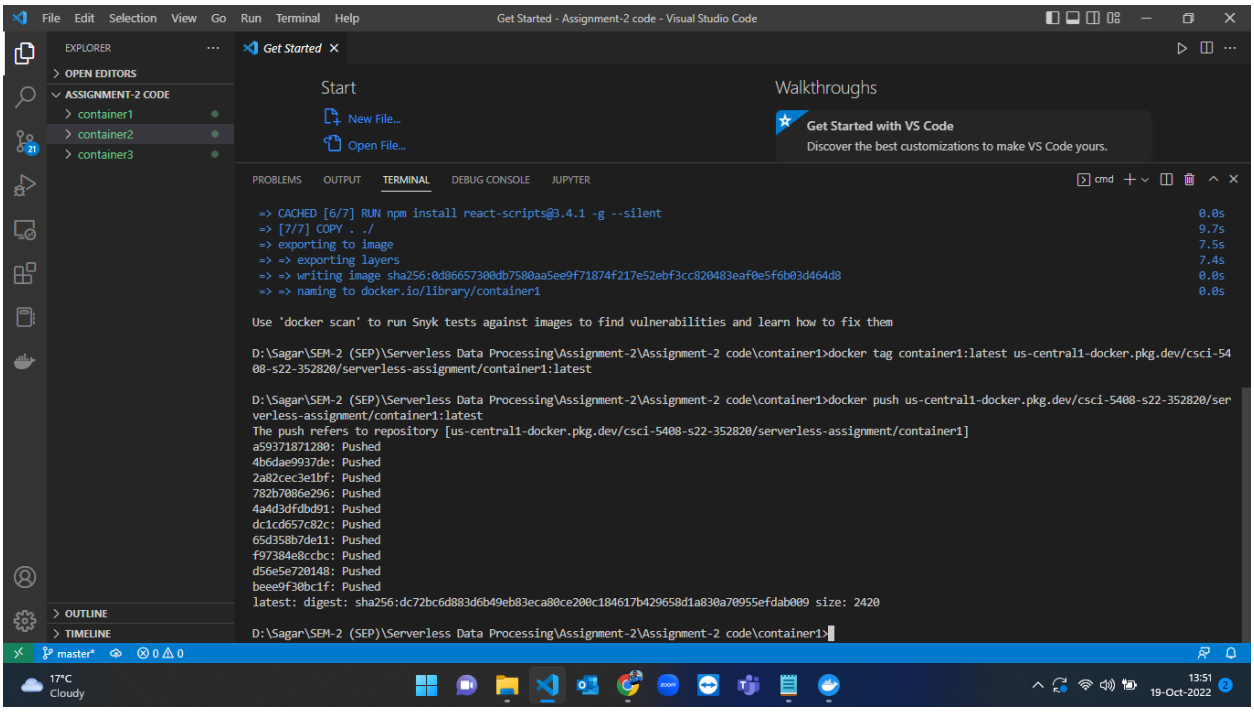

Figure-51: Registration application (conatiner1) docker image creation and pushing the code to GCP Artifact Registry Repository.

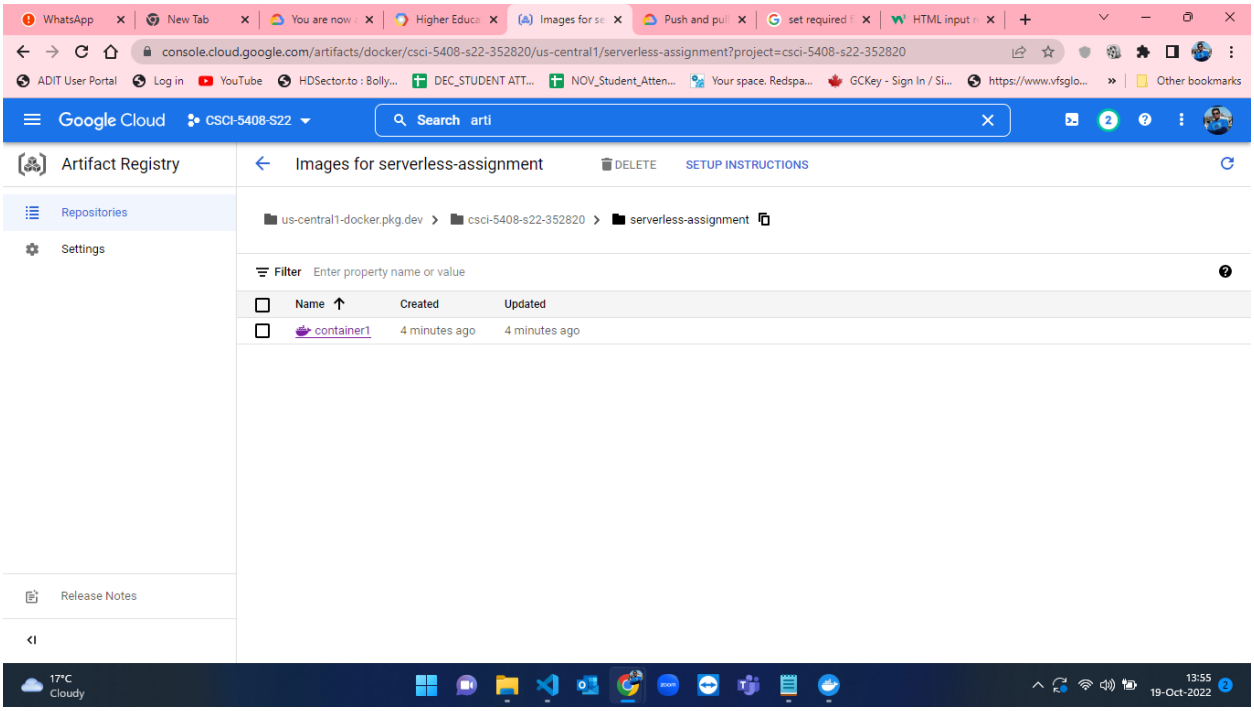

Figure-52: Artifact Registry Repository – container1 pushed in GCP.

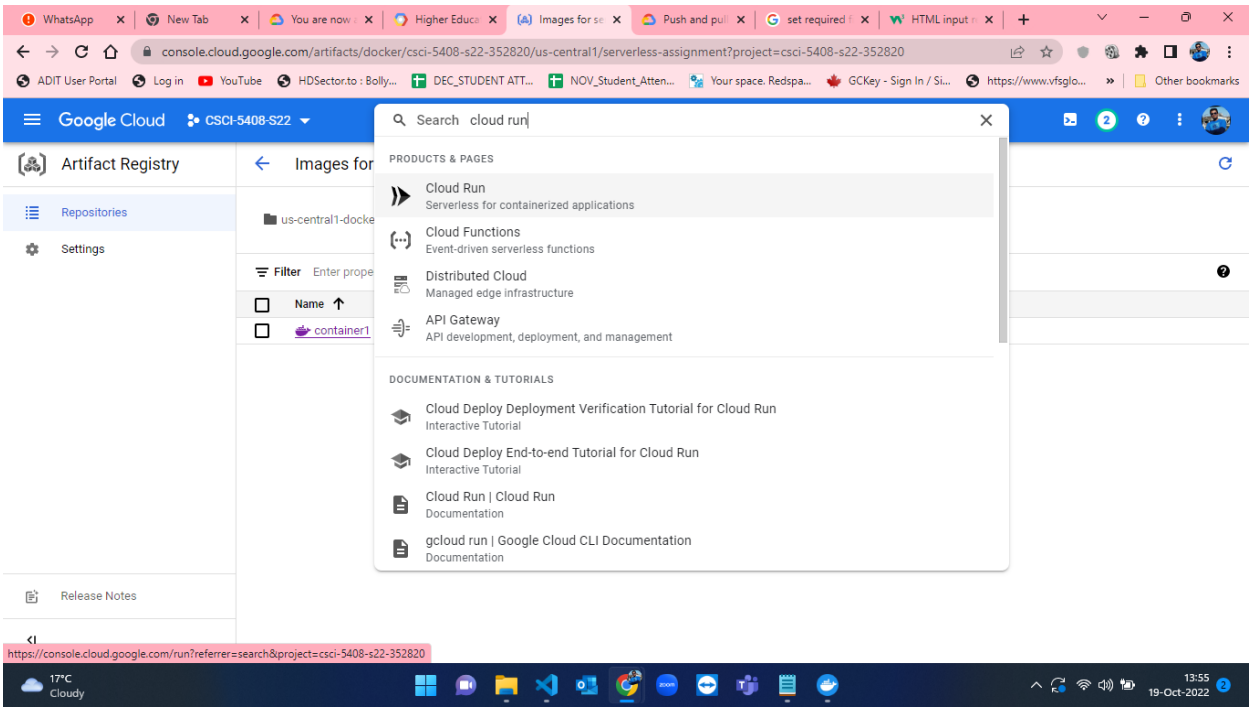

Figure-53: Cloud Run search in GCP.

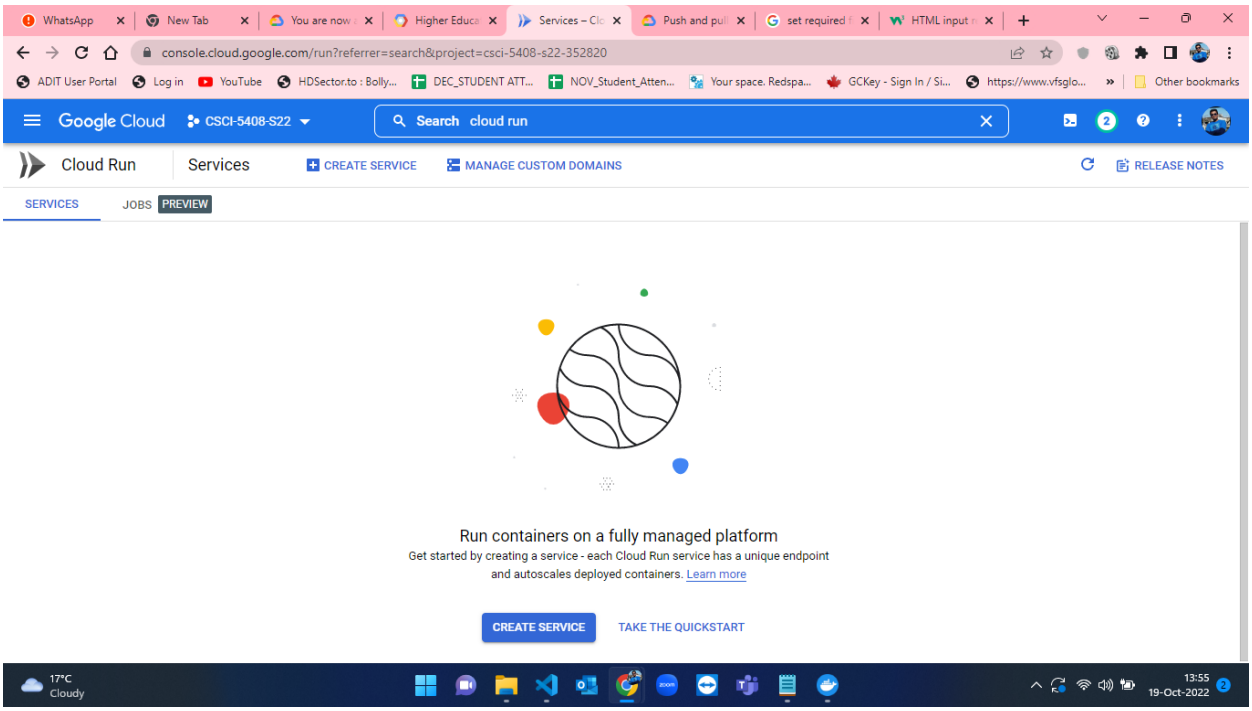

Figure-54: Cloud Run Homepage.

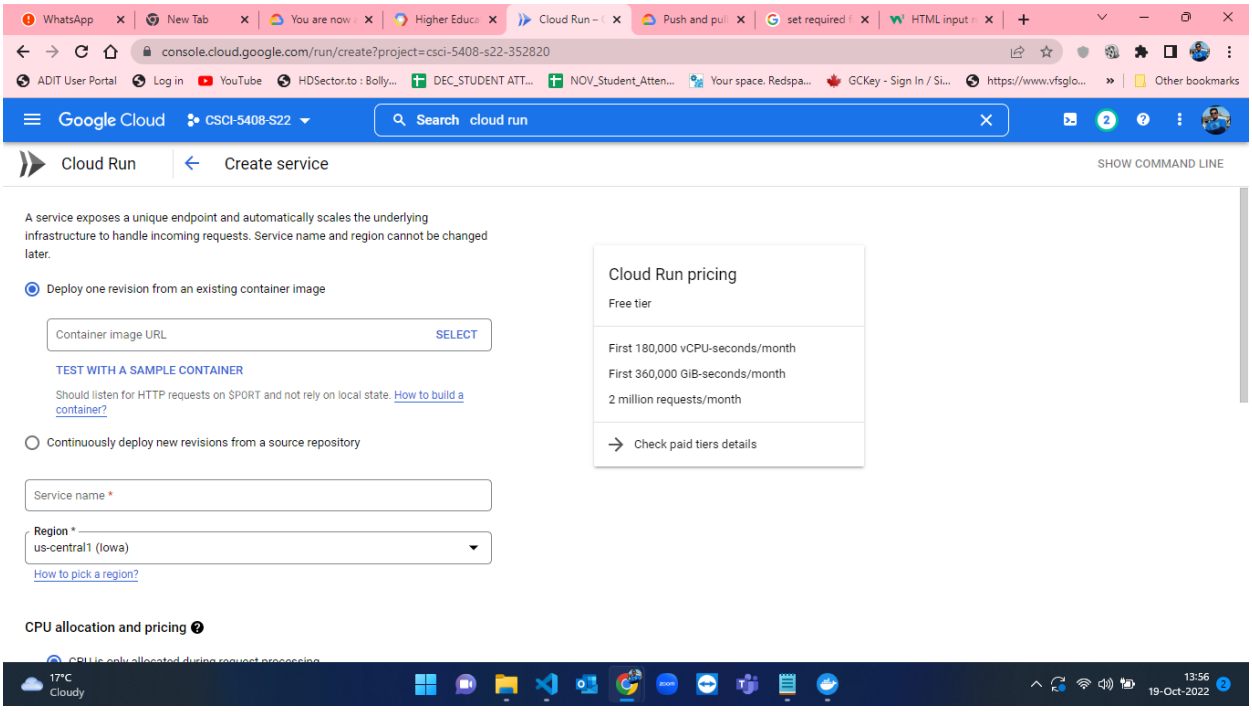

Figure-55: Cloud run form – Empty.

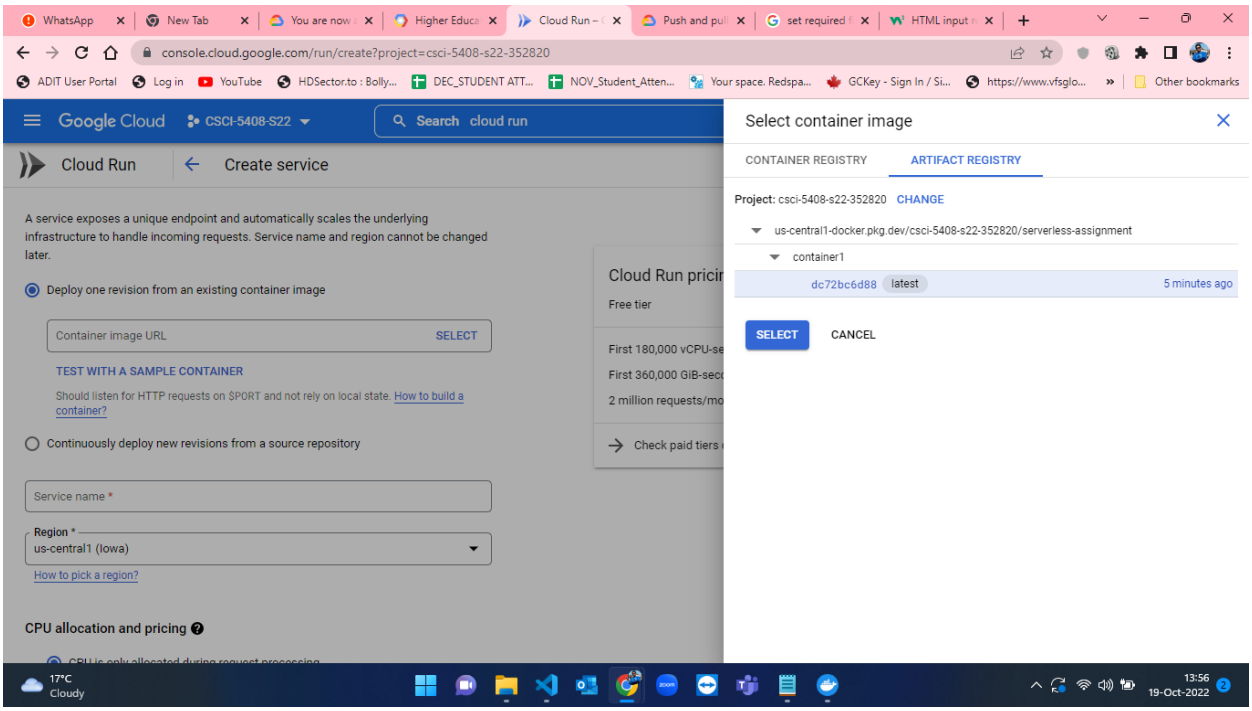

Figure-56: Selecting a container image from container1.

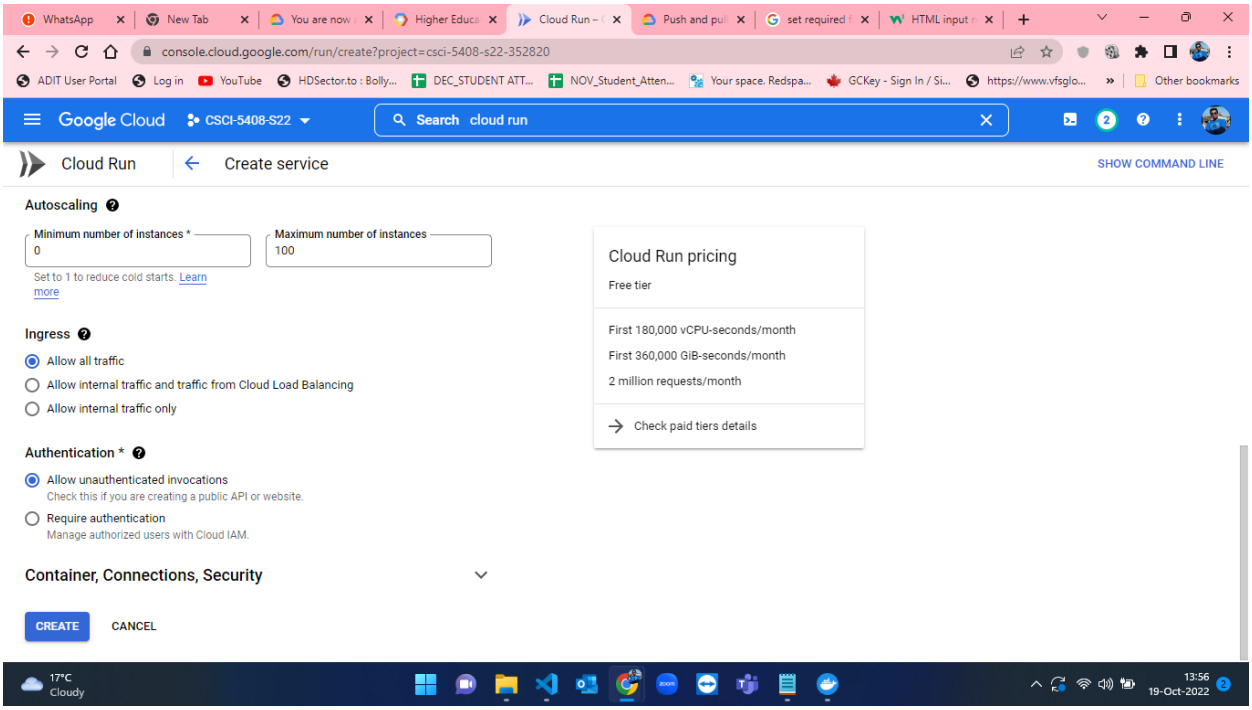

Figure-57: Selecting Allow unauthenticated invocations in cloud run.

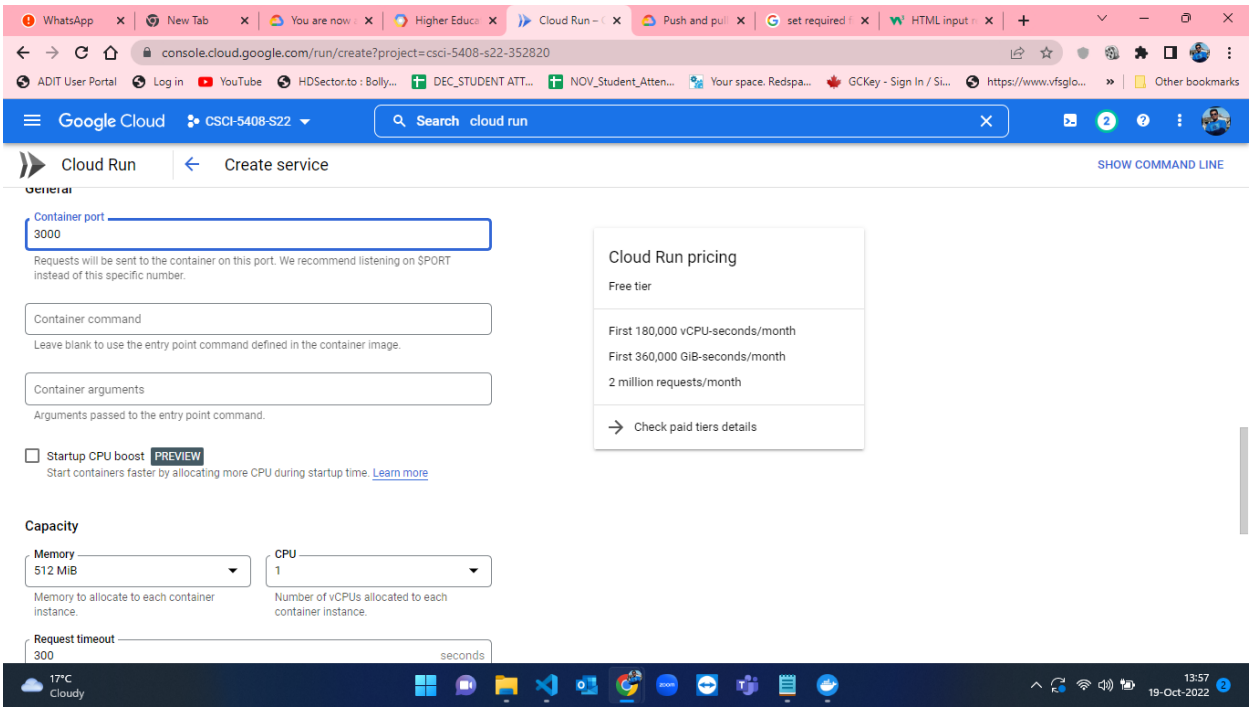

Figure-58: Changing port from 8080 to 3000 for port usage.

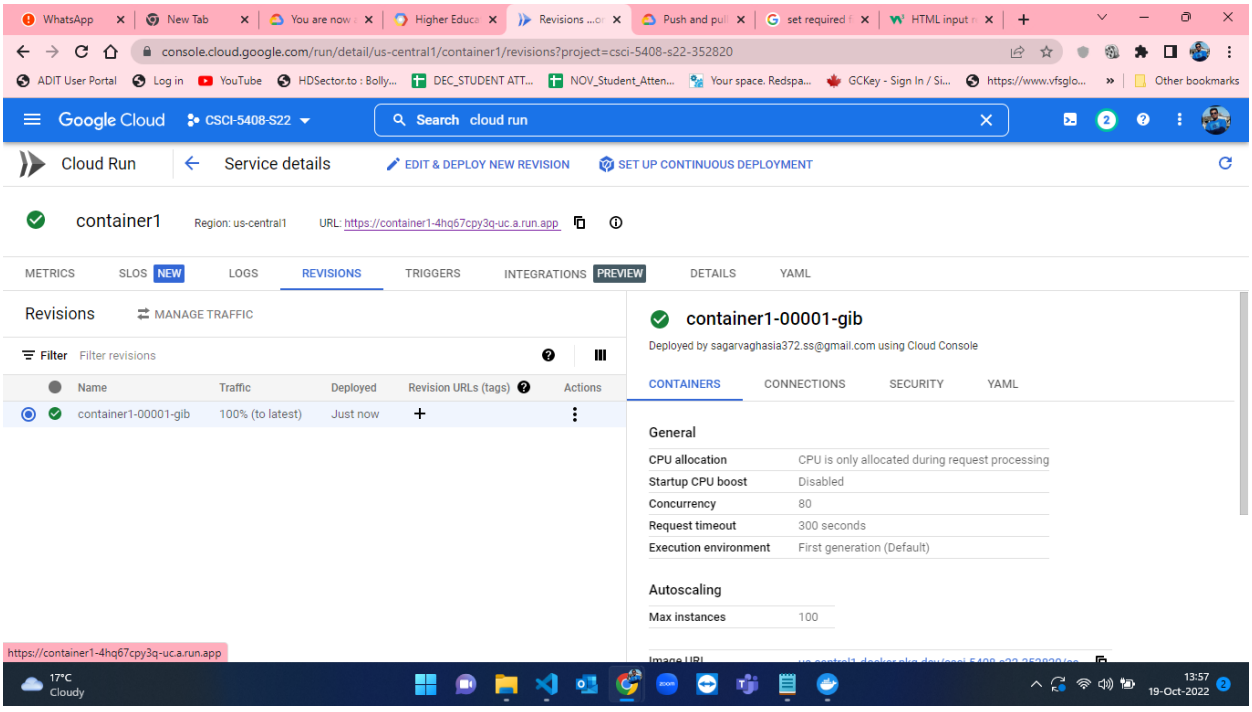

Figure-59: Registration app (conatiner1) creation completed.

- When the container is deployed, the web app is now available in the link mentioned at the starting of the document. The live application and user registration is shown in the figures from 60 to 62 .

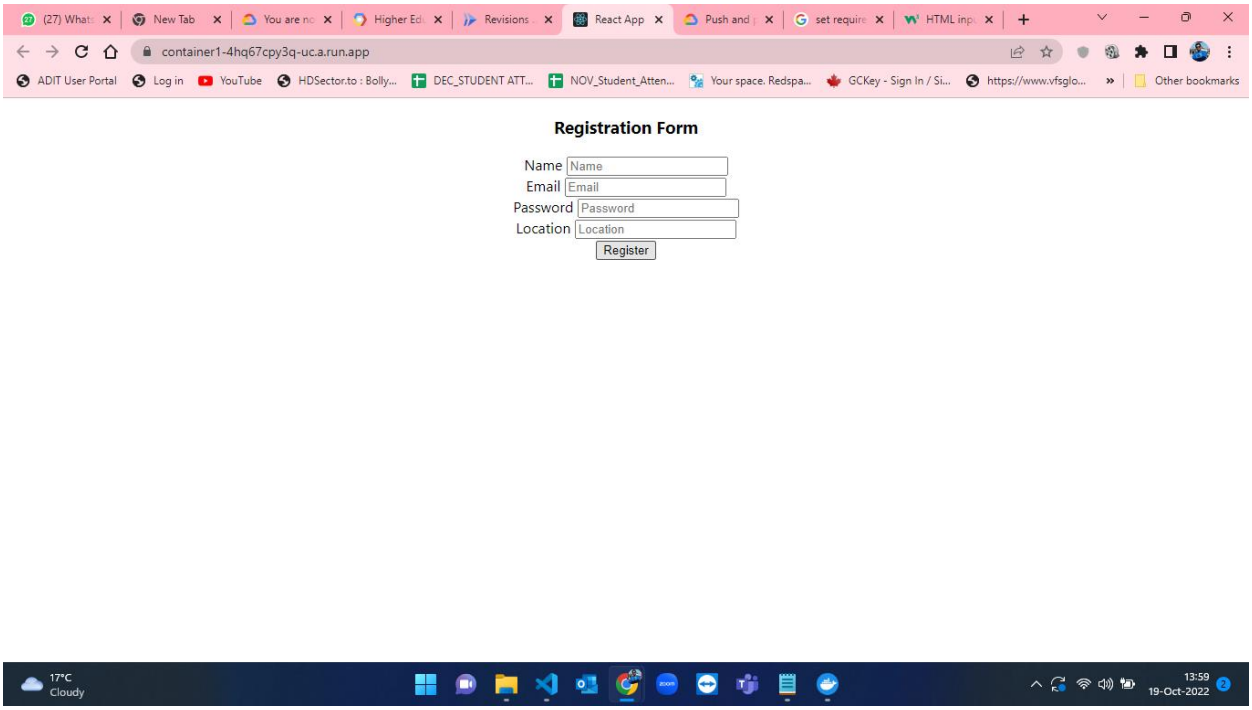

Figure-60: Registration Application from Cloud Run.

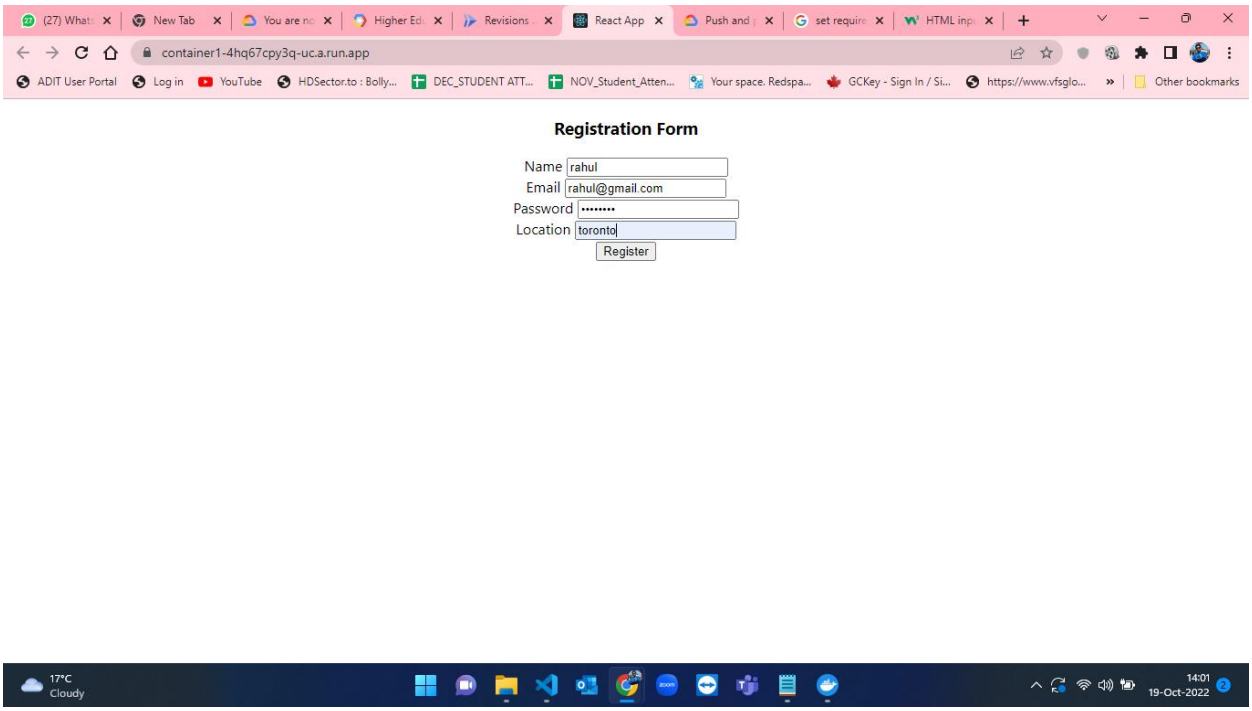

Figure-61: Registering user.

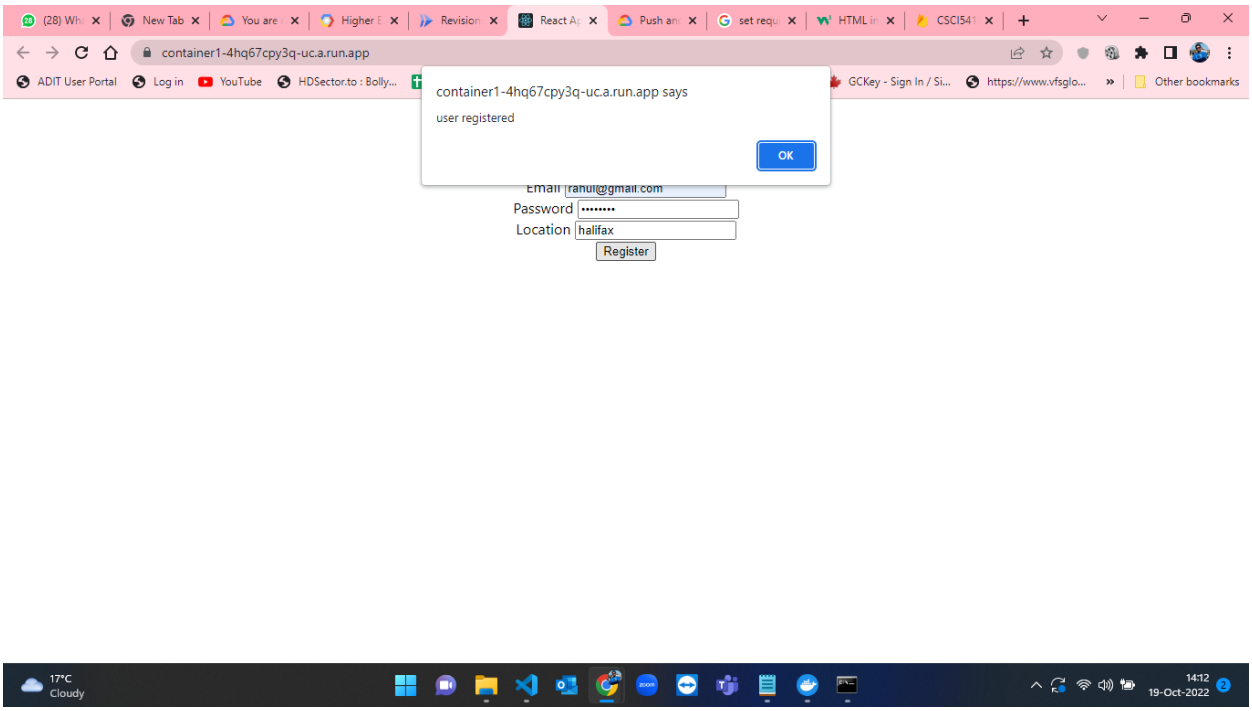

Figure-62: User registered successfully.

- After completing the registration process I developed the front-end and back-end for login app and the tasks of creating docker image, pushing the code to repo and then running cloud run services for Login have to be done. For building the docker image I have used the command “docker build -t container2 .”. After that for creating a docker image in the cloud repository I have used “docker tag conatiner2 us-central1-docker.pkg.dev/csci-5408-s22-352820/serverless-assignment/container2:latest”. After creating image for pushing the image to cloud repo I have used the command “docker push us-central1-docker.pkg.dev/csci-5408-s22-352820/serverless assignment/container2:latest”[6]. At, the end I have created cloud run service[7] and logged in for one user. The steps for the above process is shown below in the screenshots from Figure-63 to Figure-67.

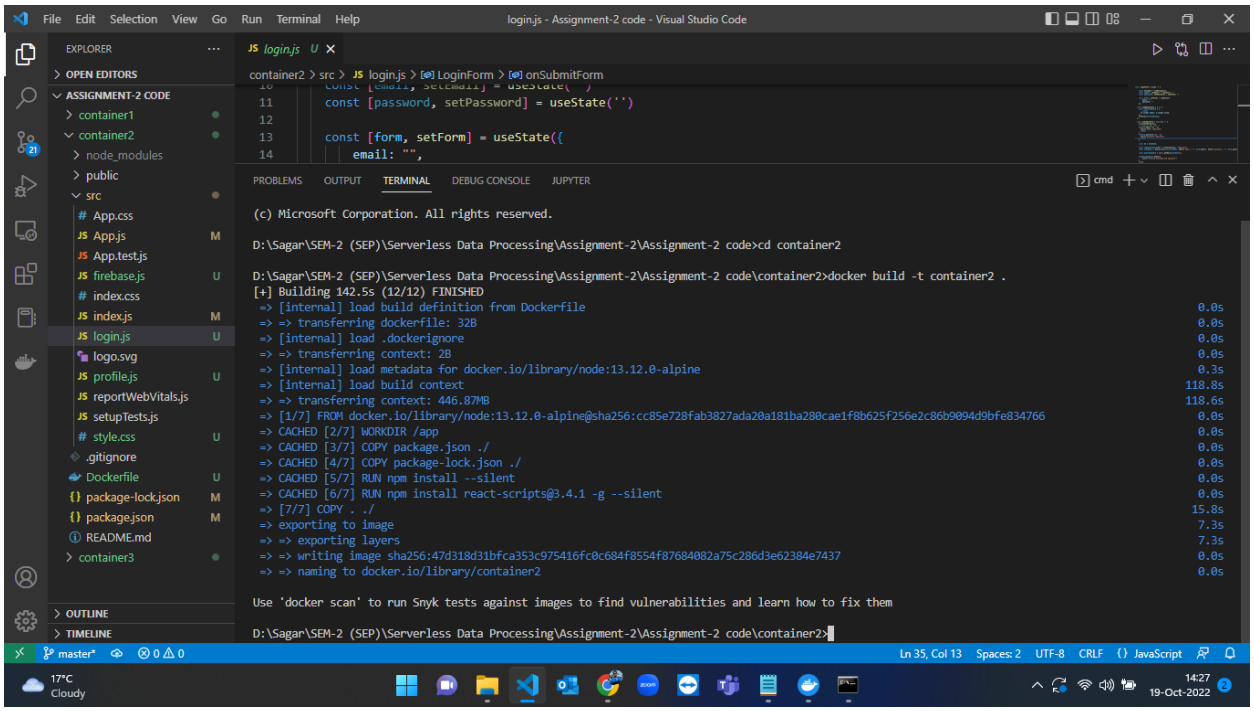

Figure-63: Building docker container2 for login app.

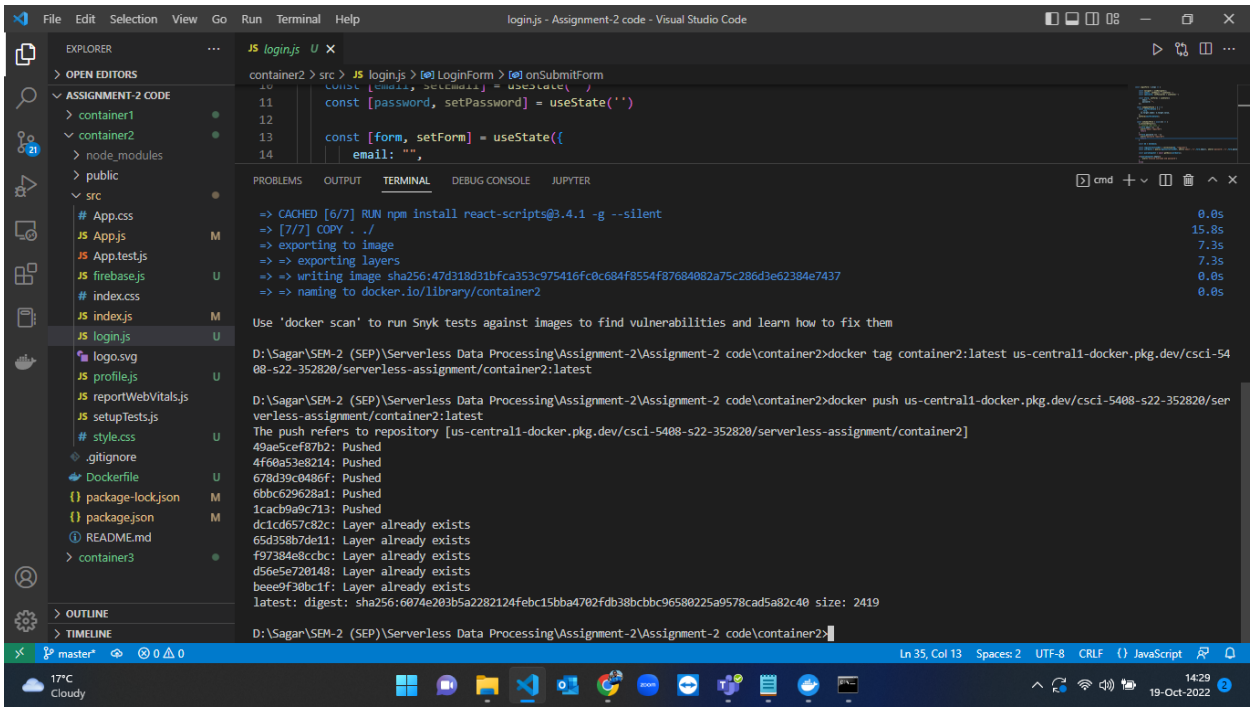

Figure-64: Login application (conatiner2) docker image creation and pushing the code to GCP Artifact Registry Repository.

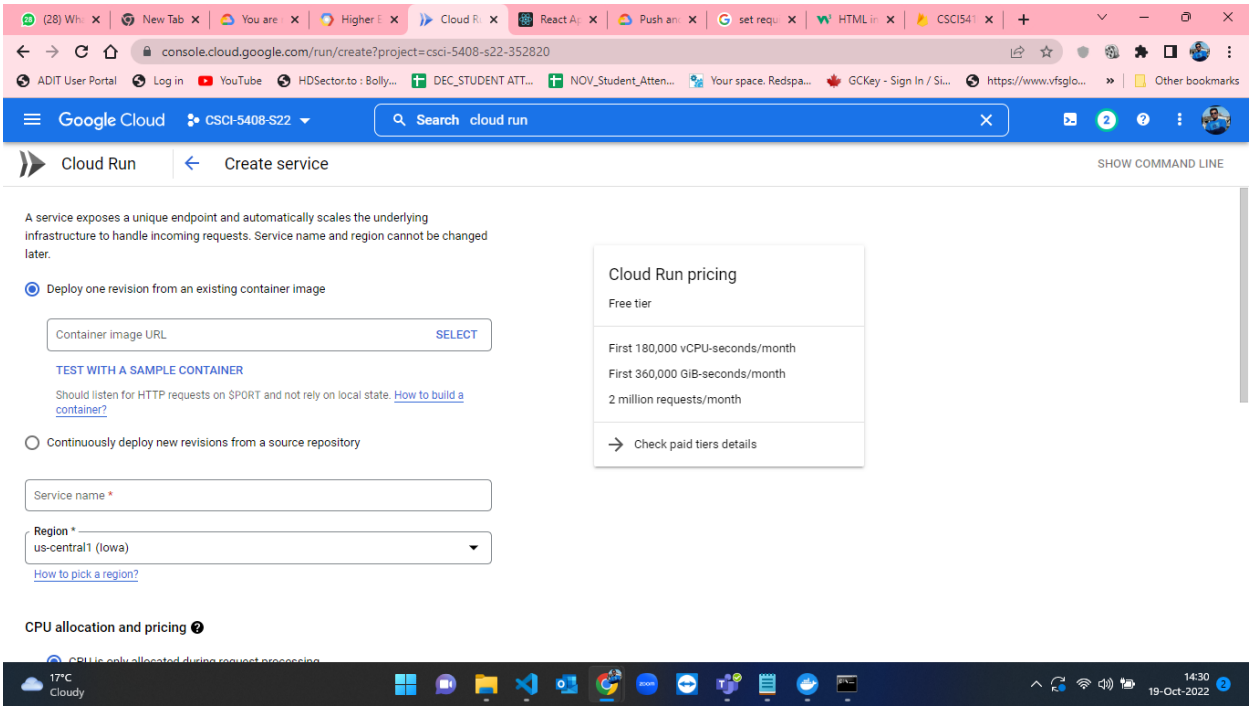

Figure-65: Cloud run form – Empty.

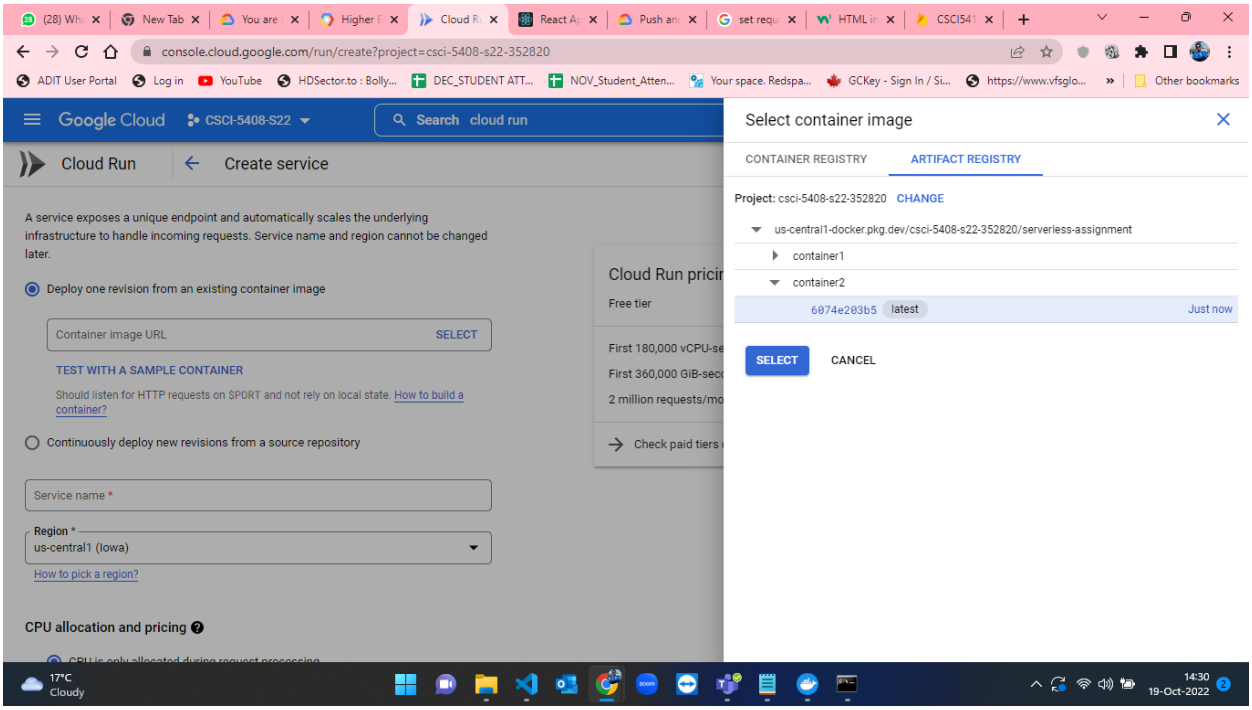

Figure-66: Selecting a container image from container2.

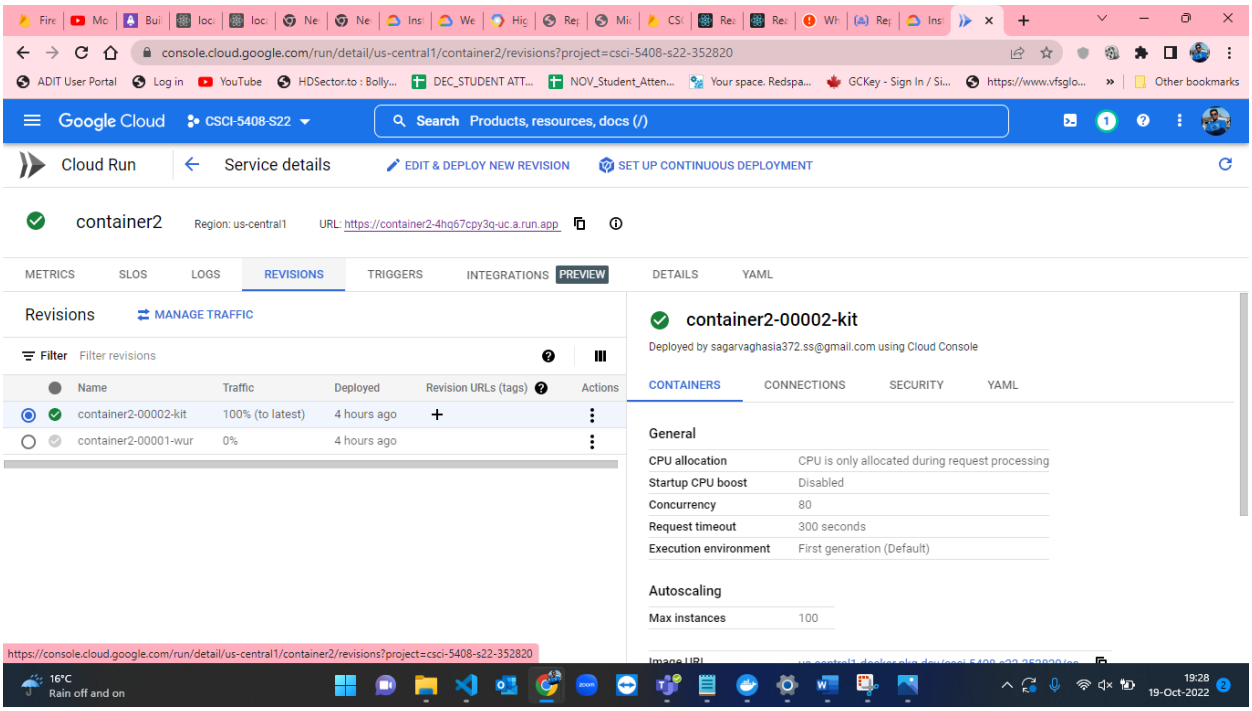

Figure-67: Login app (conatiner2) creation completed.

- When the container is deployed, the web app is now available in the link mentioned at the starting of the document. The live application and user login process is shown in the figures from 68 to 70.

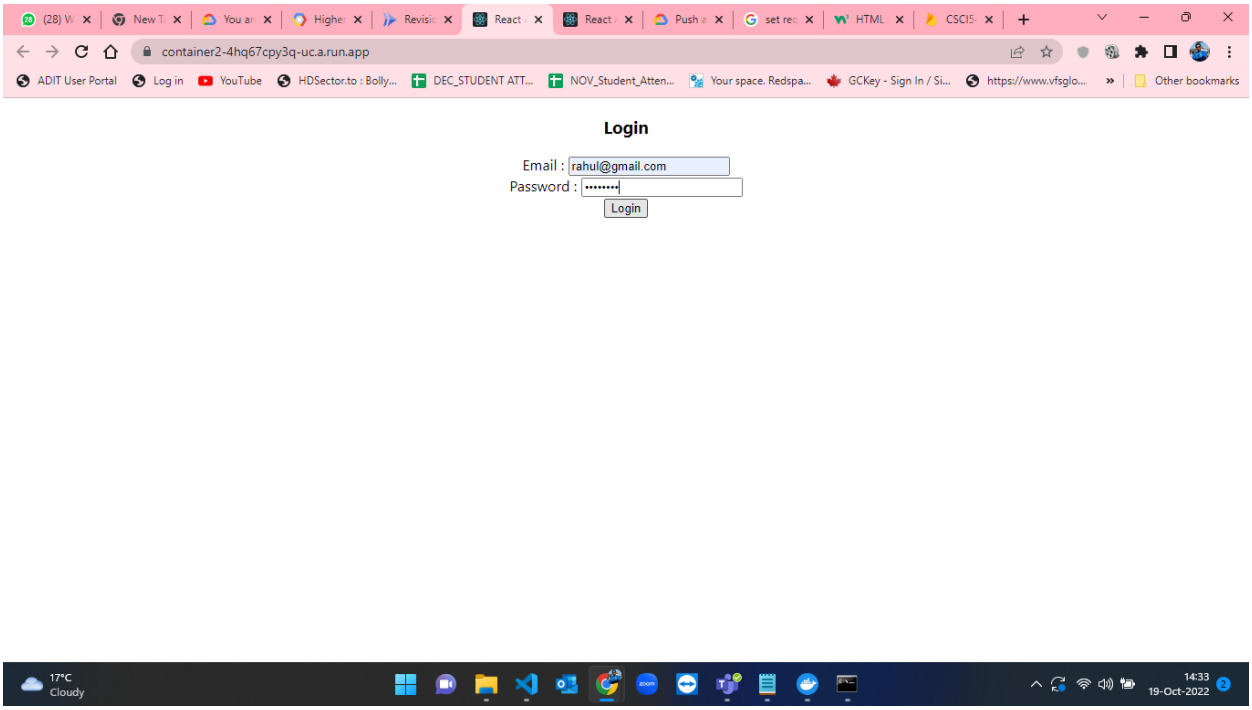

Figure-68: Login Page.

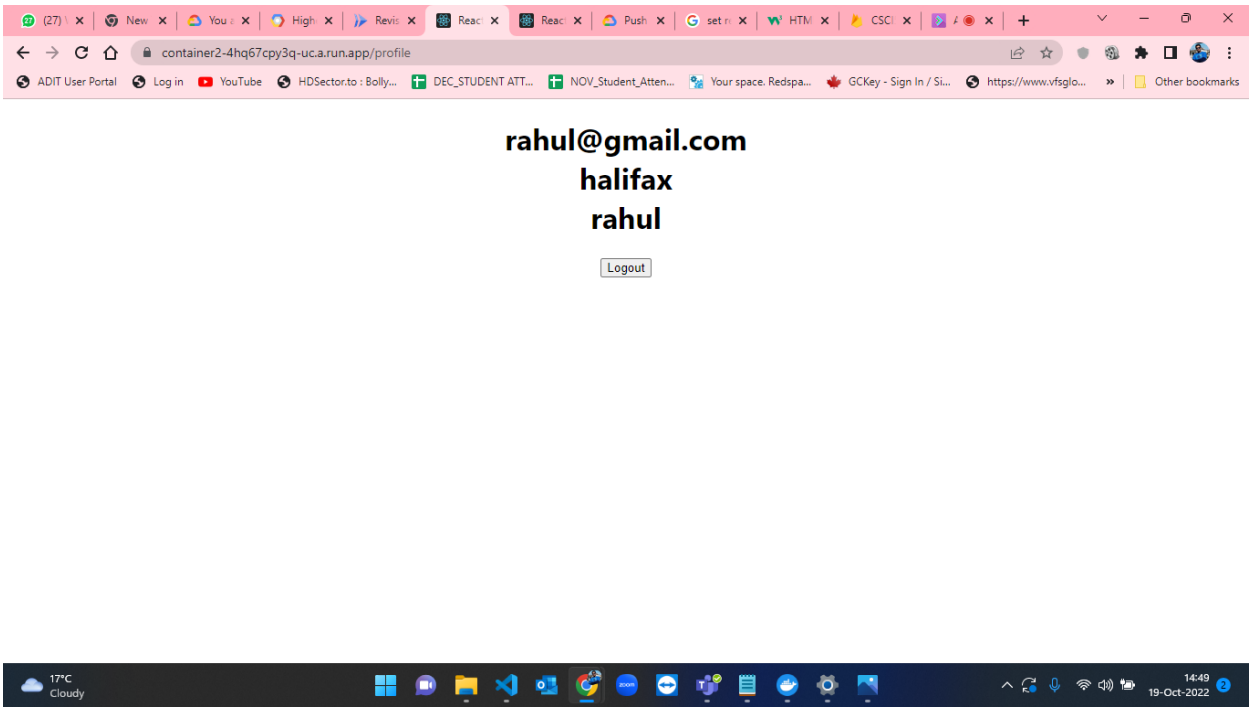

Figure-69: Redirection to profile page when user logged in.

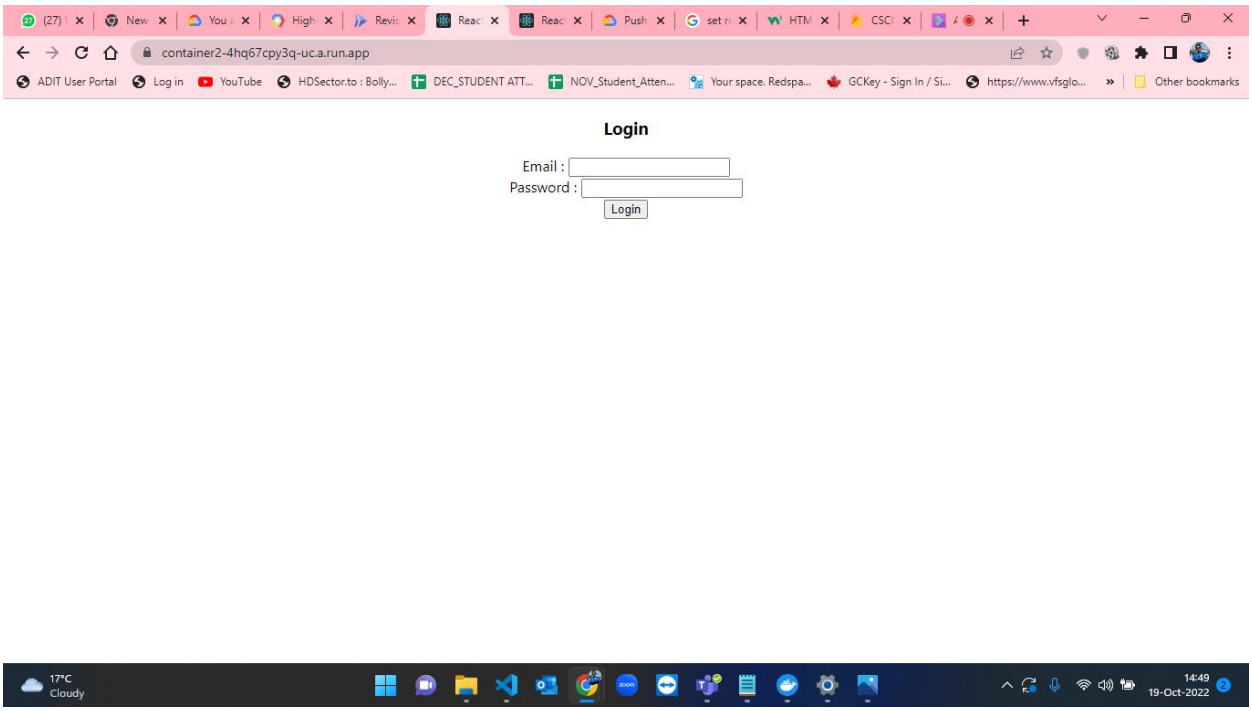

Figure-70: Redirection to Login page when user log out.

- Now, the user will be able to register and login in the web application. So, next task is to check for the users who are currently log in the application. After completing the login process, I developed the front-end and back-end for maintaining states and the tasks of creating docker image, pushing the code to repo and then running cloud run services for state have to be done. For building the docker image I have used the command “docker build -t container3 .”. After that for creating a docker image in the cloud repository I have used “docker tag conatiner3 us-central1-docker.pkg.dev/csci-5408-s22-352820/serverless-assignment/container3:latest”. After creating image for pushing the image to cloud repo I have used the command “docker push us-central1-docker.pkg.dev/csci-5408-s22-352820/serverless assignment/container3:latest”[6]. At, the end I have created cloud run service[7] and check the state of currently logged in users. The steps for the above process is shown below in the screenshots from Figure-71 to Figure-75.

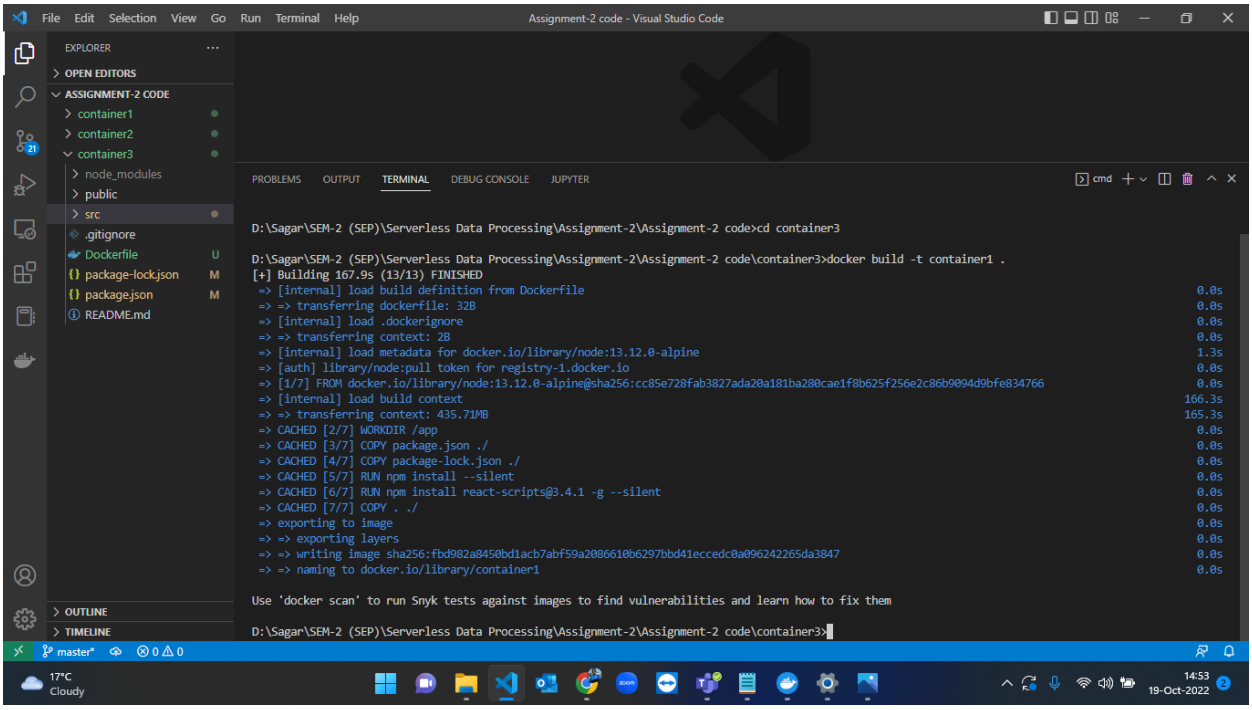

Figure-71 Building docker container3 for state app.

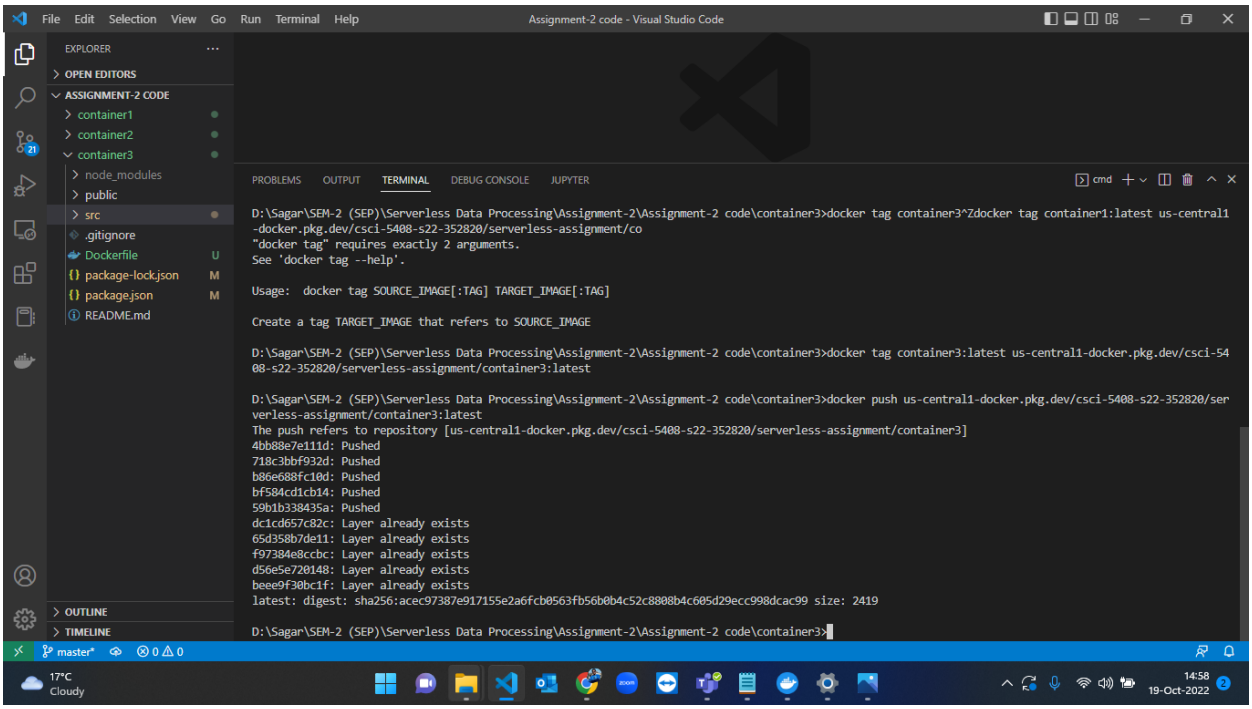

Figure-72: State application (conatiner3) docker image creation and pushing the code to GCP Artifact Registry Repository.

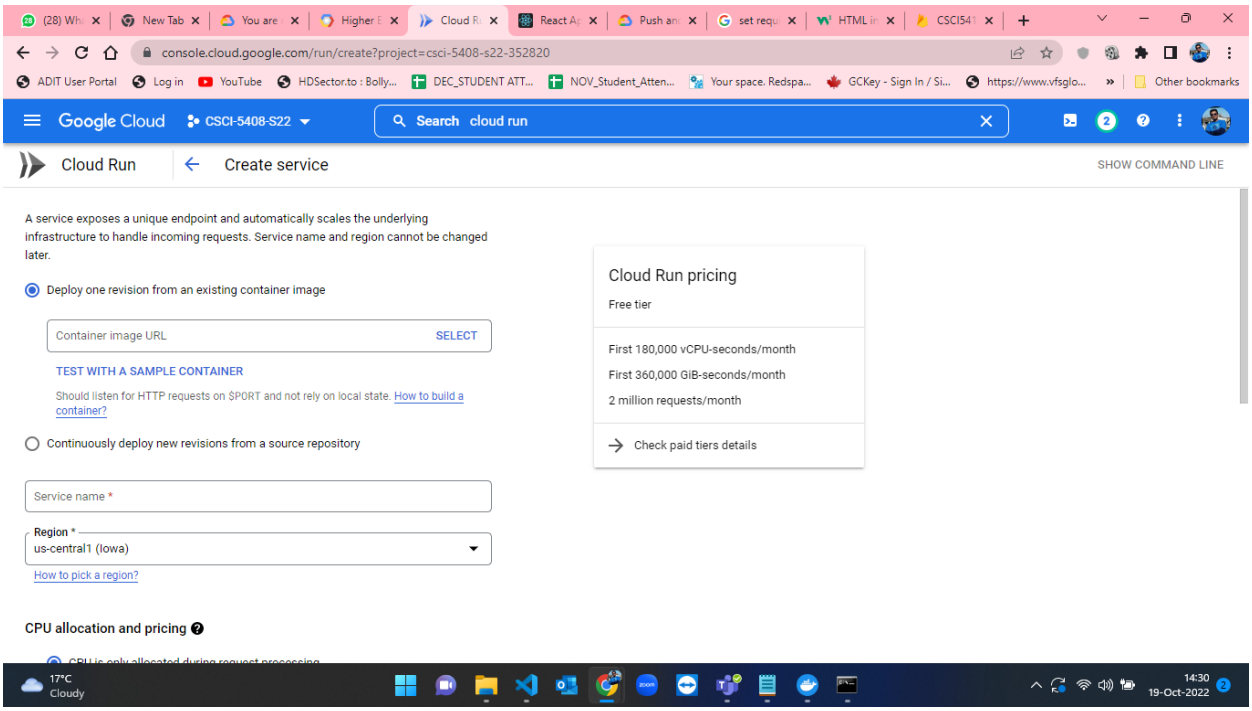

Figure-73 Cloud run form – Empty.

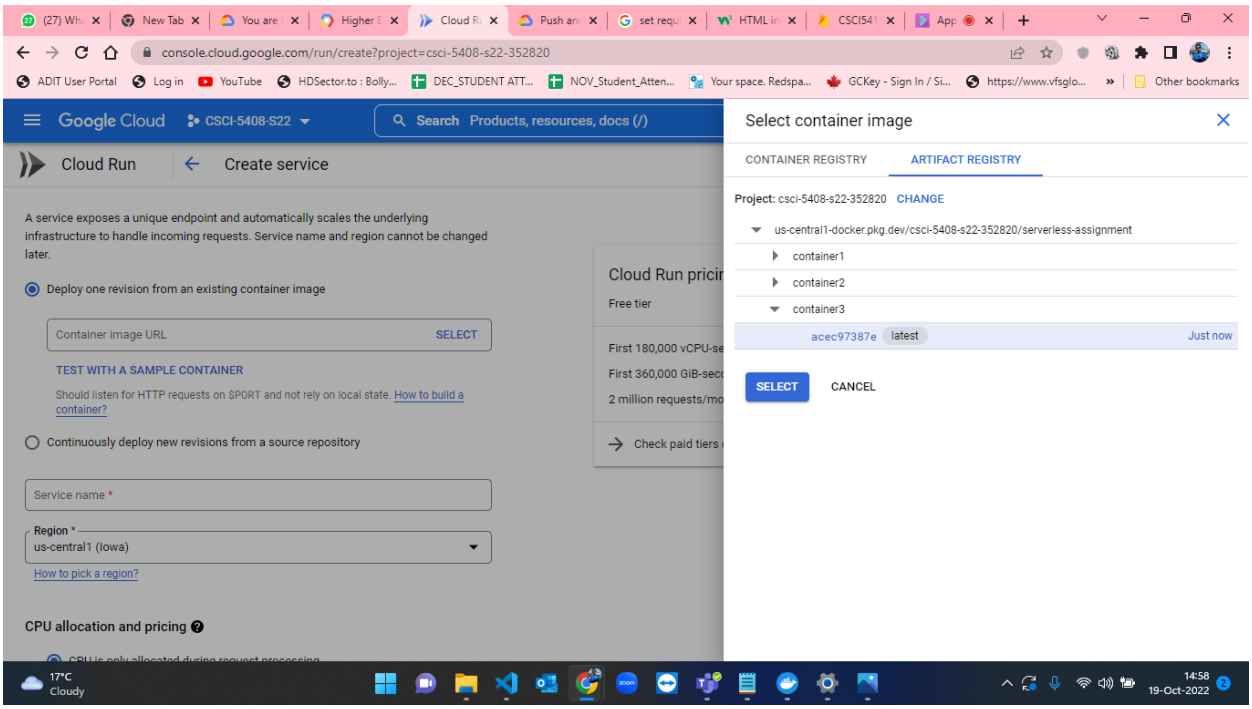

Figure-74: Selecting a container image from container3.

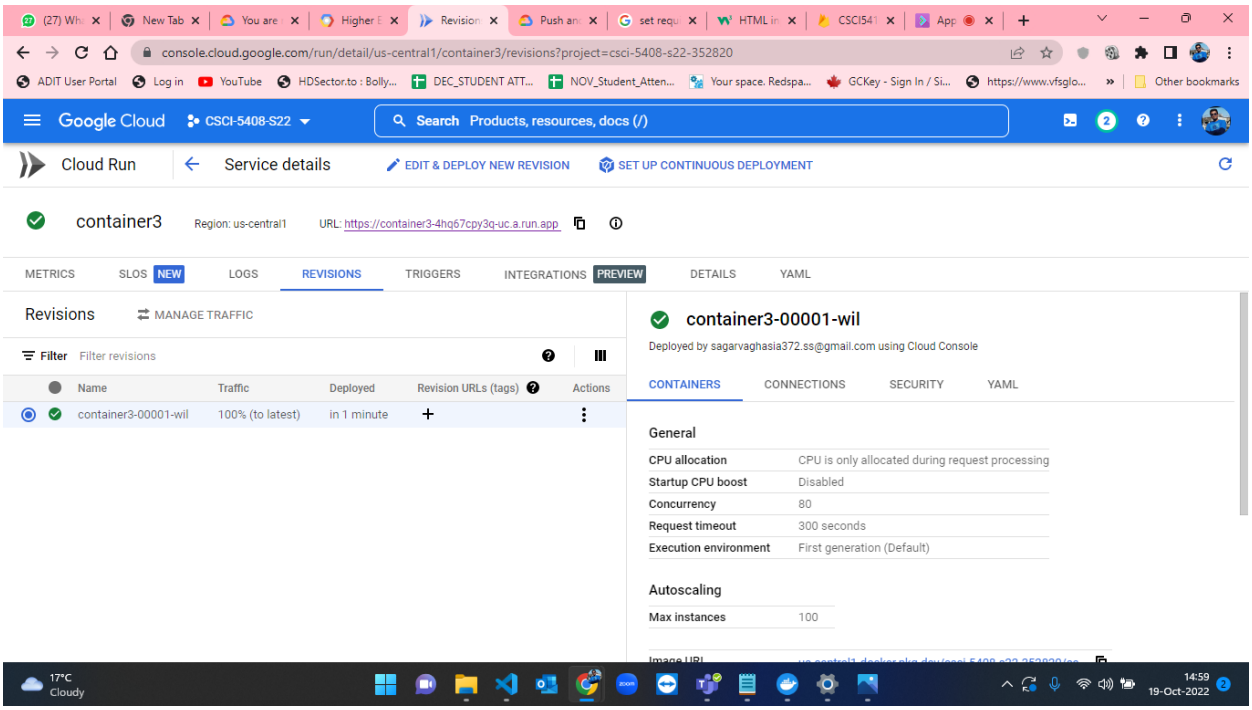

Figure-75: State app (conatiner3) creation completed.

- When the container is deployed, the web app is now available in the link mentioned at the starting of the document. The live application and state process is shown in the figures from 76 and 77.

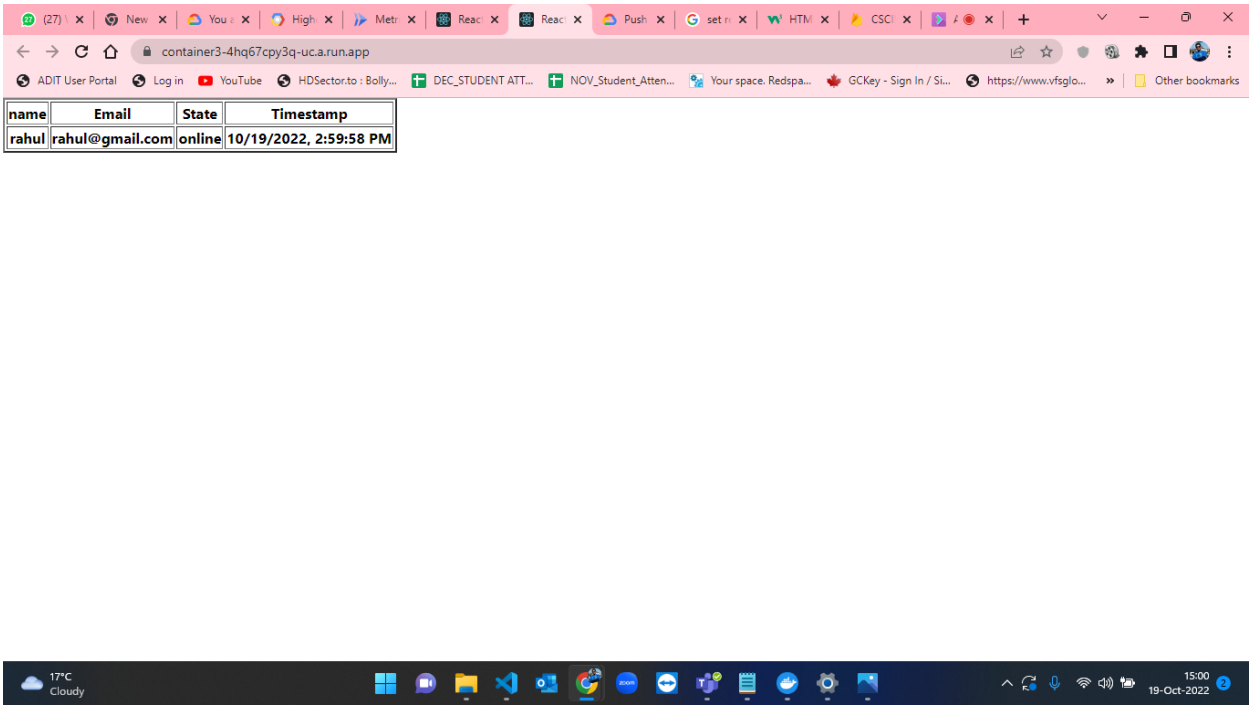

Figure-76: State application showing currently logged in users.

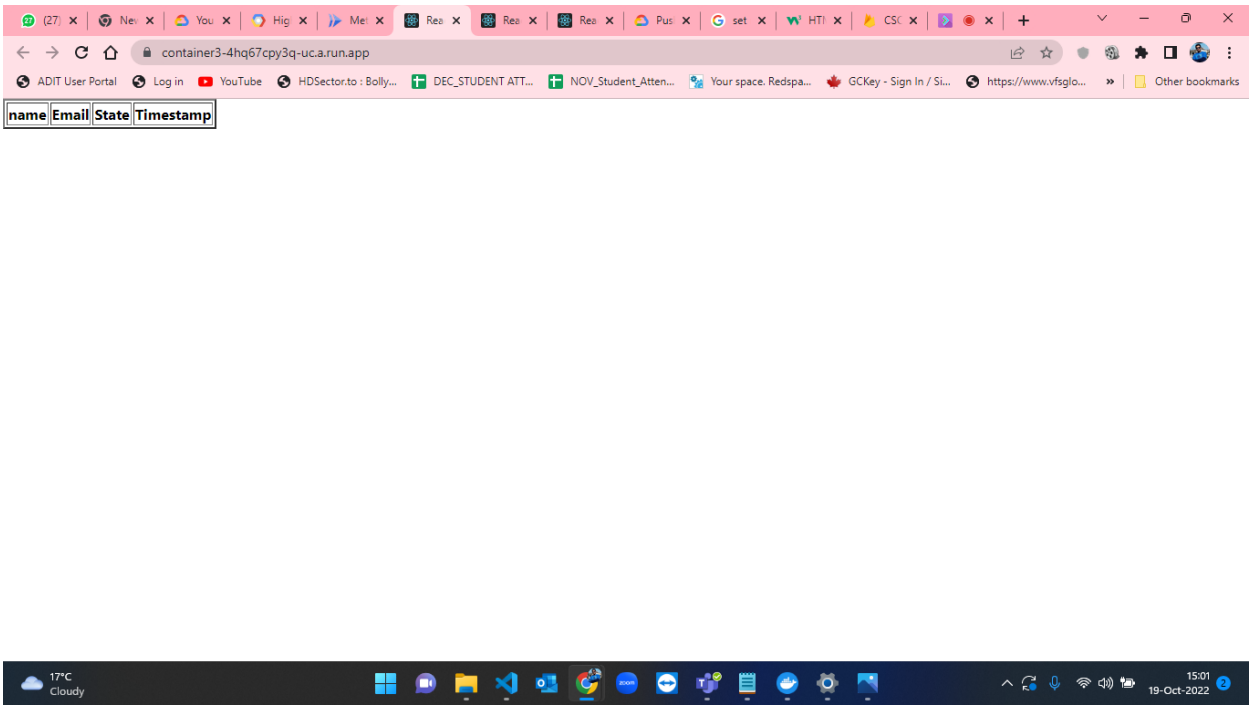

Figure-77: State application showing there are no active users currently.

Test Cases:

- Figures from 78 to 82 indicates the validations applied on the Registration Application.

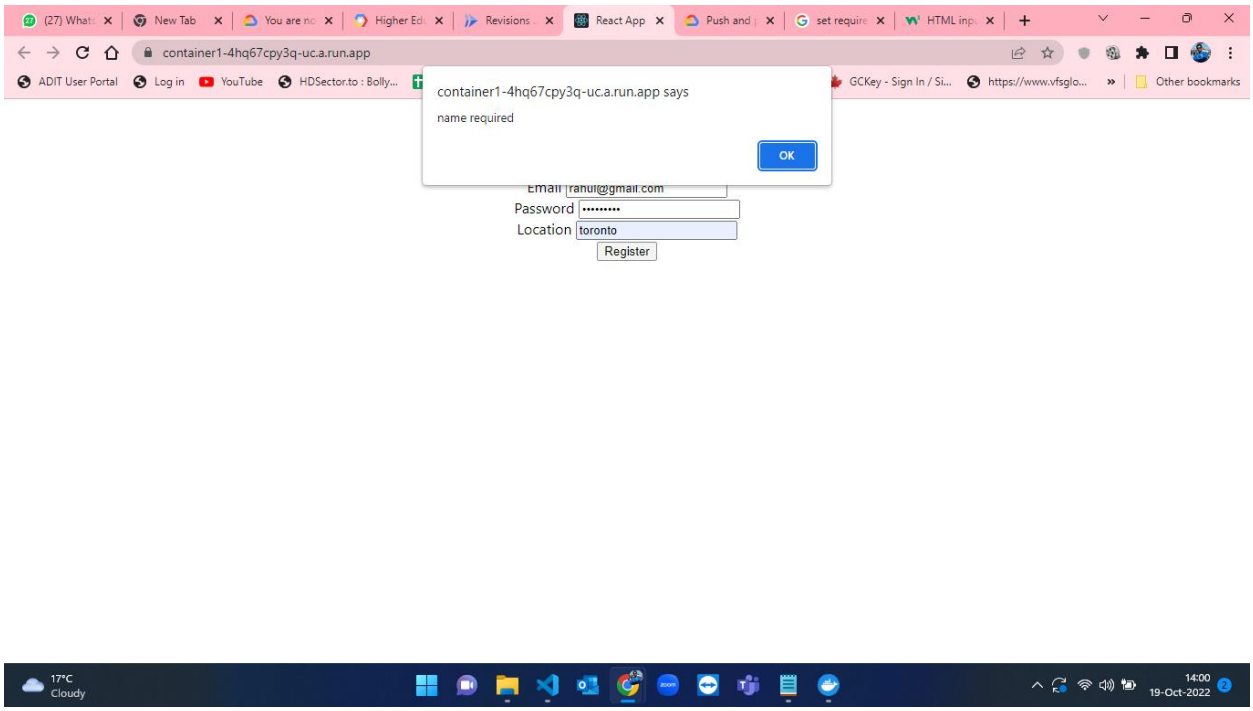

Figure-78: Name required validation in Registration App.

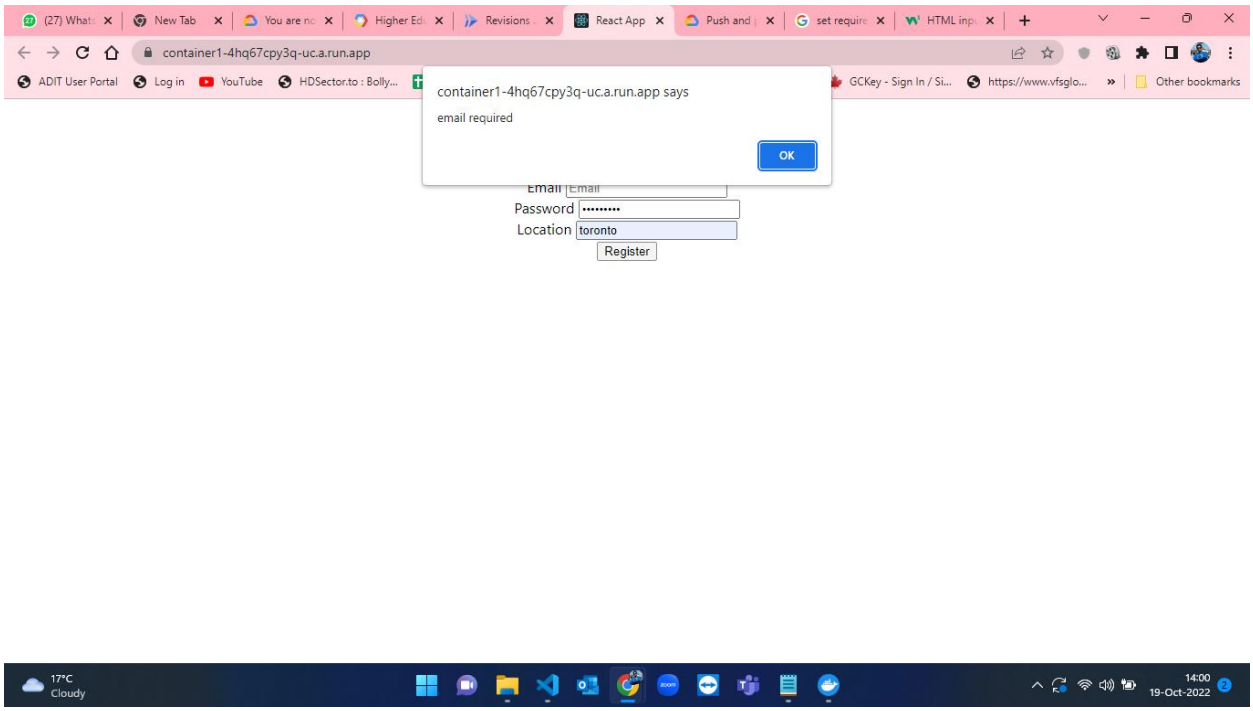

Figure-79: Email required validation in Registration App.

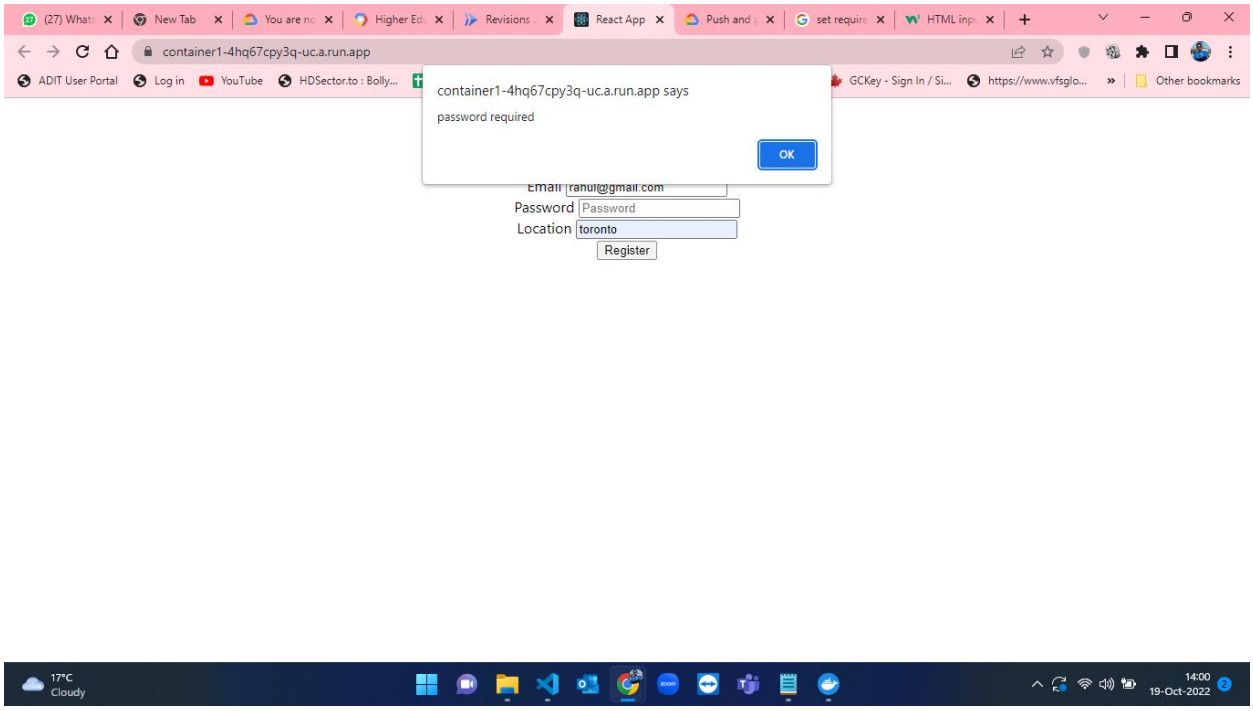

Figure-80: Password required validation in Registration App.

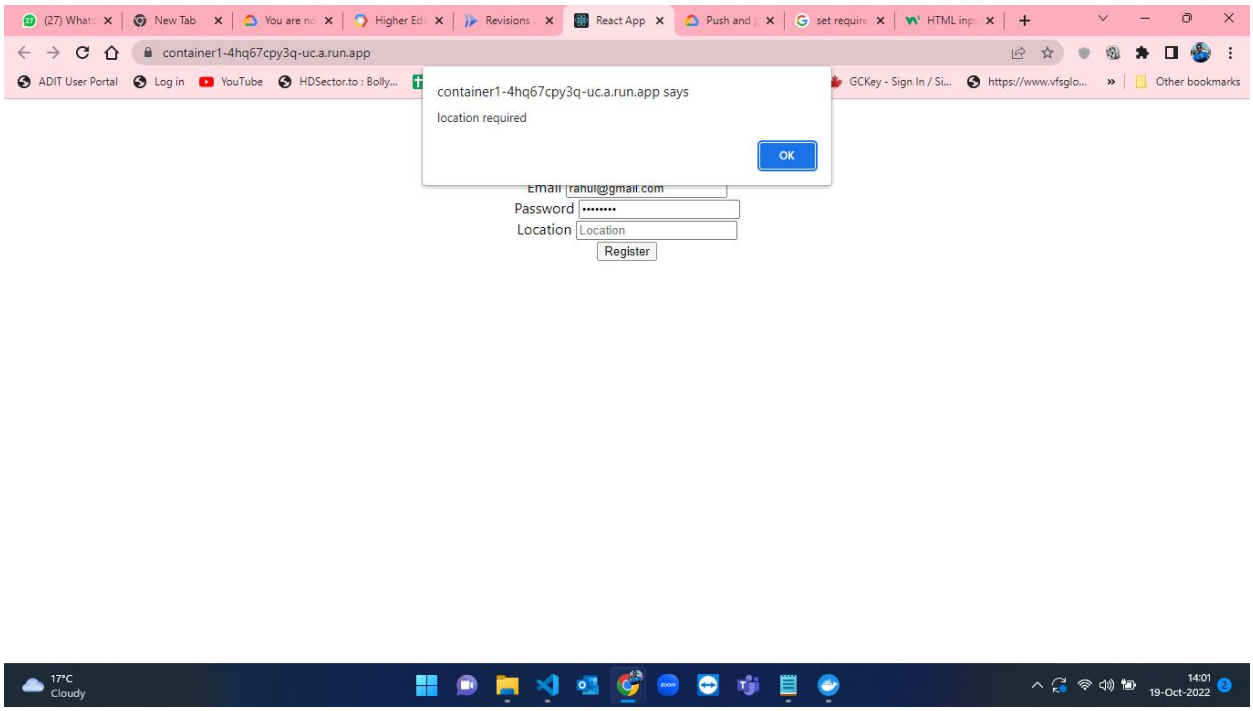

Figure-81: Location required validation in Registration App.

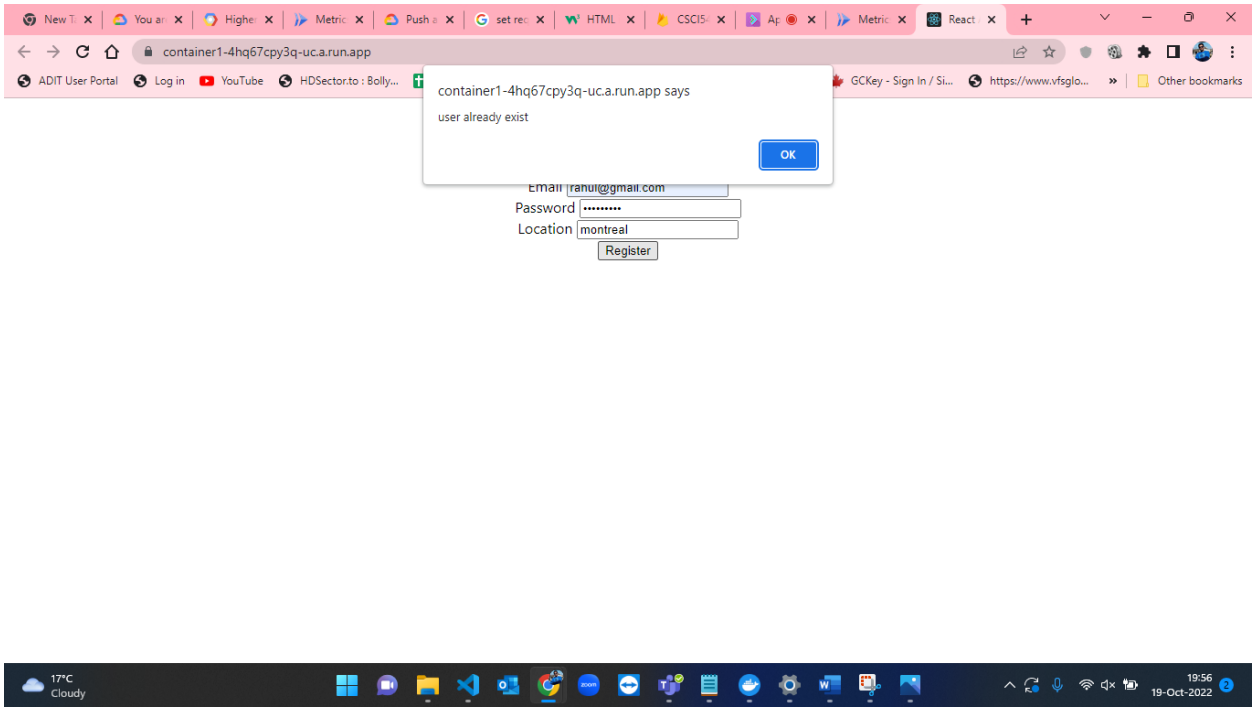

Figure-82: User already exists with the given email validation in Registration App.

- Figures from 83 to 85 indicates the validations applied on the Login Application.

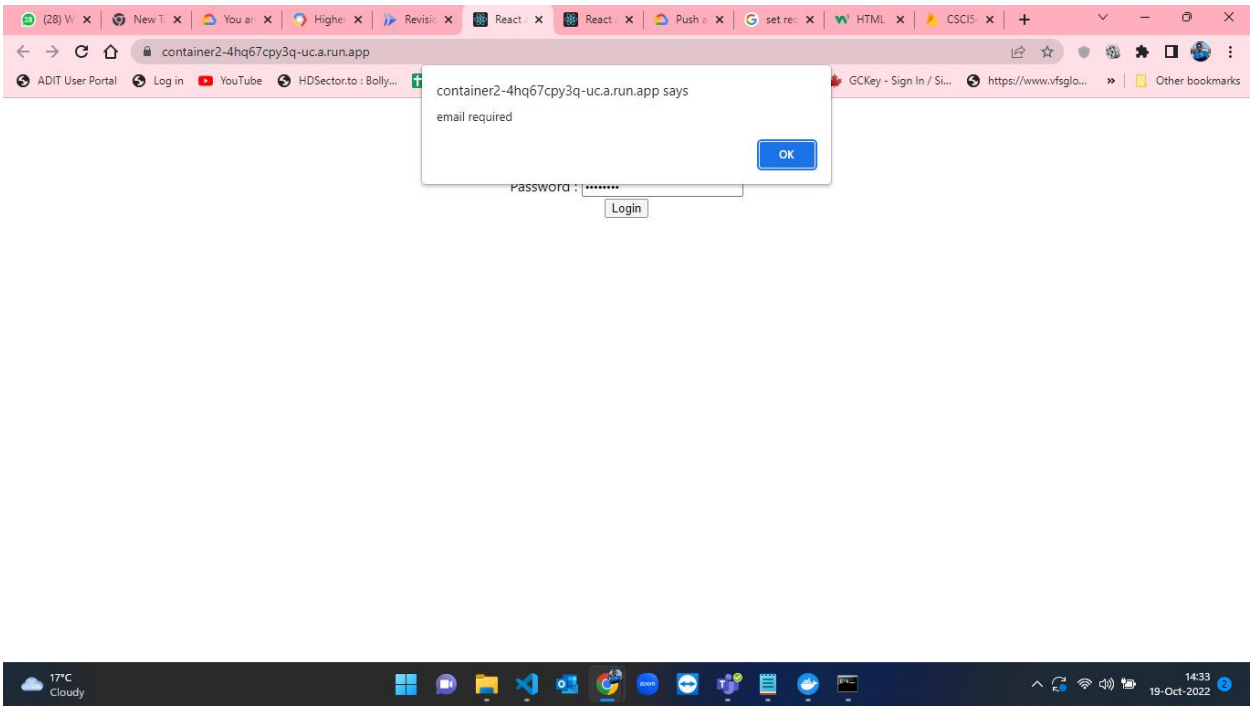

Figure-83: Email required validation in Login App.

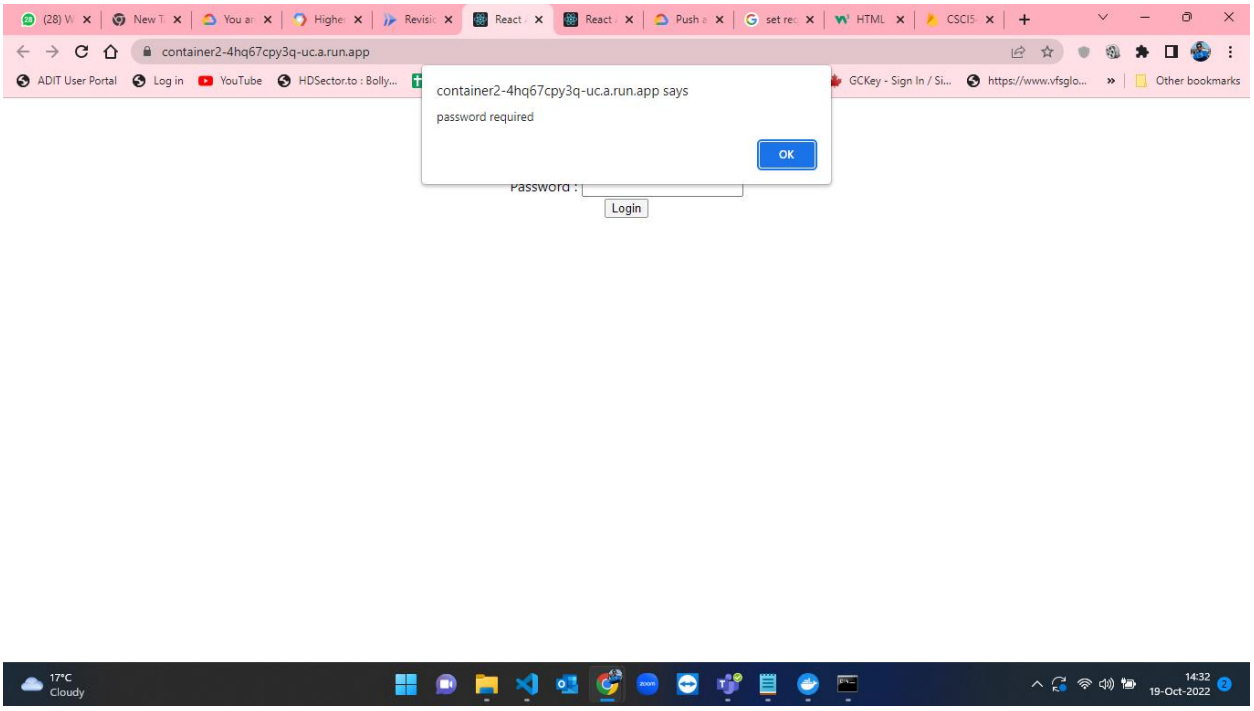

Figure-84: Password required validation in Login App.

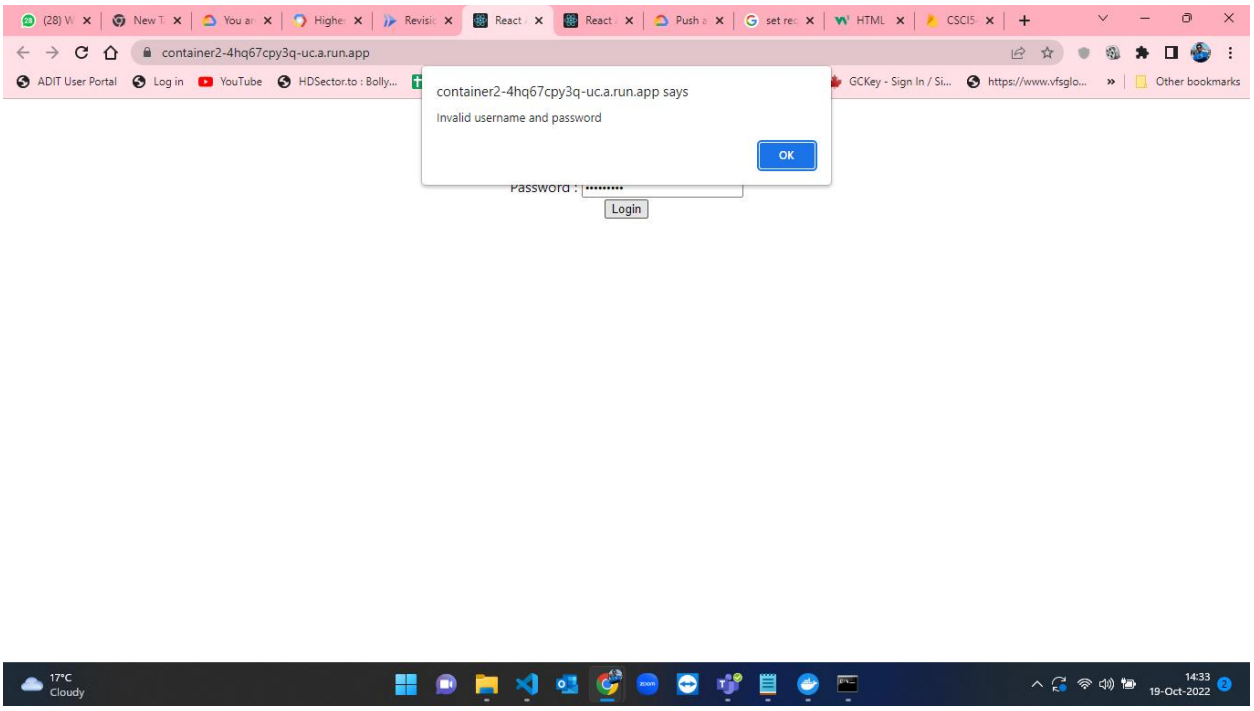

Figure-85: Invalid Username and Password validation with backend in Login App.

- Figures 86 and 87 indicates the table of active users in the application test cases which is applied on the State Application.

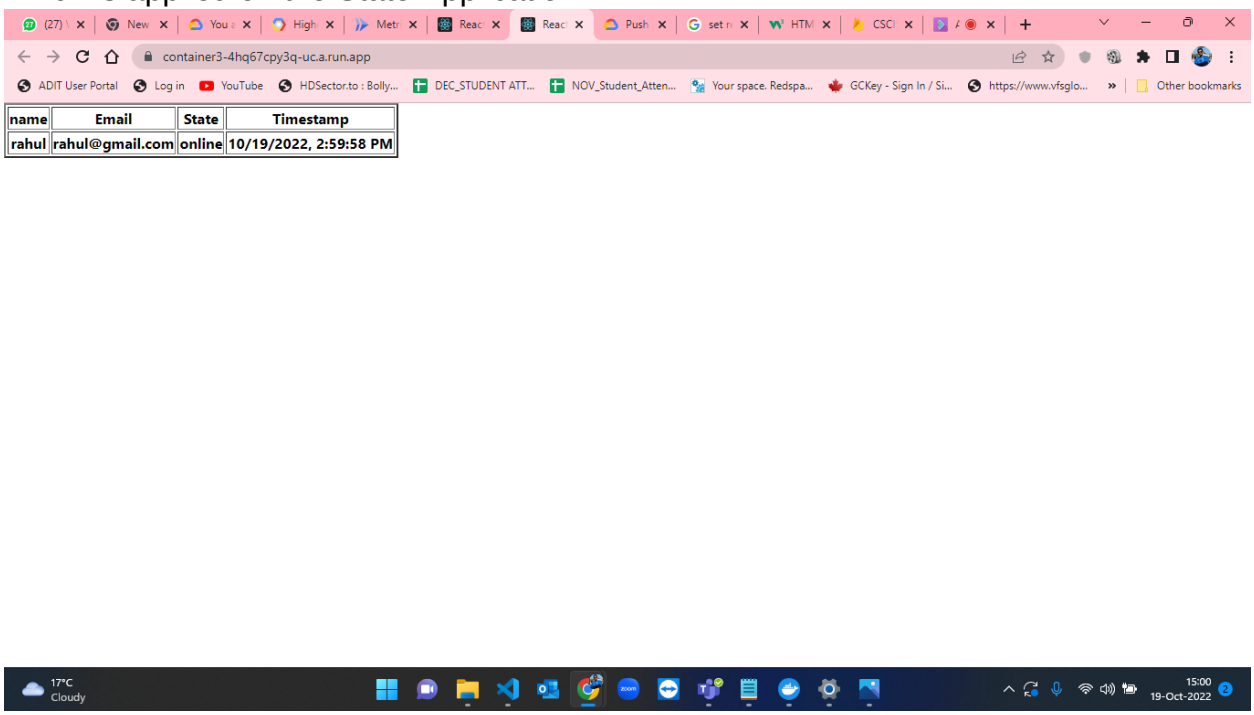

Figure-86: State application showing currently logged in users.

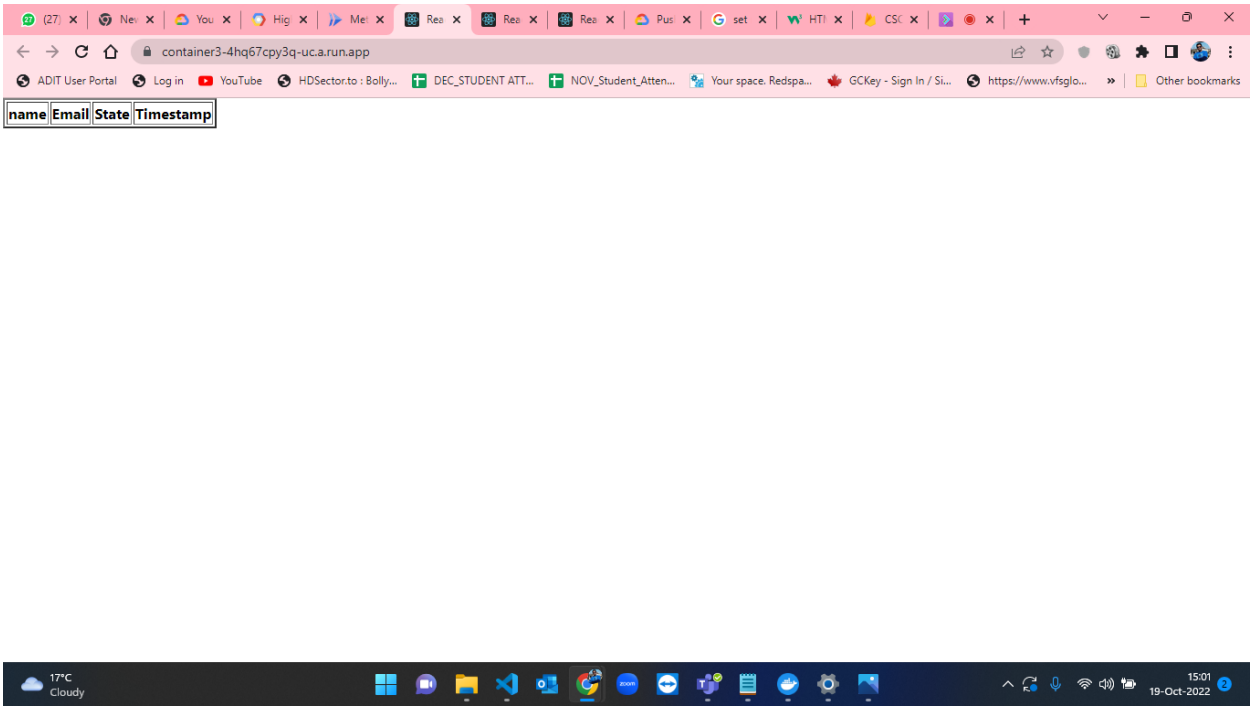

Figure-87: State application showing there are no active users currently.

Important Methods / pseudo code :

Container-1:

- **register.js**
  - function handleSubmit()

```
    async function handleSubmit(e) {
        e.preventDefault()
        if(name === ''){
            alert("name required")
            return
        }
        if(email === ''){
            alert("email required")
            return
        }
        if(password === ''){
            alert("password required")
            return
        }
        if(location === ''){
            alert("location required")
            return
        }

        const q = query(collection(db, "register"), where("email", "==", email))

        const docs = await getDocs(q)

        if(docs.docs.length > 0){
            alert("user already exist")
        }
        else{
        const registerCollectionRef = collection(db, 'register')
        addDoc(registerCollectionRef, {name:name, email:email, password:password,
location:location}).then(response => {
            console.log(response)
            alert("user registered")
        }).catch(error =>{
            console.log(error.message)
        })
    }

    }
```

Container-2:

- **login.js**
  - const onSubmitForm

```
  const onSubmitForm = async(e) => {
    e.preventDefault();
    if(form.email === ''){
      alert("email required")
      return
    }
    if(form.password === ''){
      alert("password required")
      return
    }

    const db = database;

    const loginCollectionRef = collection(db, "register")
    const userQuery = query(loginCollectionRef, where('email','==',form.email),
where('password','==',form.password))

    const querySnapshot = await getDocs(userQuery);

    if(querySnapshot.empty){
        alert('Invalid username and password')
    }
    else{
        const d = querySnapshot.docs[0]

        localStorage.setItem("authenticated", true)

        const q = query(collection(db, "State"), where("email", "==",
form.email))

        const docs = await getDocs(q);

        let stateId

        if(docs.docs.length > 0){
          console.log(docs.docs[0])
            await updateDoc(doc(db, "State", docs.docs[0].id), {
              "state": "online",
              "timestamp": Timestamp.now()
```

```
            })

            // data["stateId"] = docs.docs[0].id
            stateId = docs.docs[0].id
        }

        else{
          const docRef = await addDoc(collection(db, "State"), {
            "email": form.email,
            "name": d.data().name,
            "state": "online",
            "timestamp": Timestamp.now()
          })

          stateId = docRef.id
        }

        localStorage.setItem("stateId", stateId)
        navigate('/profile',{state: d.data()})
    }

};
```

- const onUpdateField

```
const onUpdateField = e => {
  const nextFormState = {
    ...form,
    [e.target.name]: e.target.value,
  };
  setForm(nextFormState);
};
```

- const [form, setForm] = useState

```
  const [form, setForm] = useState({
      email: "",
      password: "",
  });
```

- **profile.js**
  - const logout = async(e)

```
    const logout = async(e) => {

        e.preventDefault()
        console.log(localStorage.getItem("stateId"))

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

        navigate("/")
    }
```

Container-3:

- **App.js**

```
  const [states, setStates] = useState([])

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

    const docs = await getDocs(q);

    let data = []

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

    console.log(data)

    return data
  }


  useEffect(() => {

    getData().then((response) => {
      setStates(response)
    })
  }, [])

  console.log(states)
```

Summary

Docker is an open-source platform for building, deploying, and managing containerized applications[3]. Platform as a Service (PaaS) products from Docker are used to distribute software in packages known as containers using OS-level virtualization. There are free and premium tiers for the service. The Docker Engine is the name of the program that runs the containers[3]. A Docker image is **a file used to execute code in a Docker container**. Docker images act as a set of instructions to build a Docker container, like a template. Docker images also act as the starting point when using Docker. An image is comparable to a snapshot in virtual machine (VM) environments. Docker is installed on server side and it gives us the permissions to build, start, stop or deploy containers. Firstly, we write the commands and instructions in the DockerFile to install all the dependencies that our hosts need to install for the software. Docker executes all the commands and instructions inside the DockerFile. Then, it builds docker image by execution of all the commands in the DockerFile so that we can run the produced image in docker container[3].

Google Artifact Registry is a place to manage container images and language packages (such as Maven and npm)[4]. It is fully integrated with Google Cloud's tooling and runtimes and comes with support for native artifact protocols. This makes it simple to integrate it with your CI/CD tooling to set up automated pipelines[4].

Google Cloud Run is a managed compute platform that enables us to run containers that are invocable via requests or events[5]. Cloud Run is serverless: it abstracts away all infrastructure management, so you can focus on what matters most — building great applications.

In this assignment, I developed the front-end and back-end for registration app which is container1. For, development of this registration app I have used React JavaScript framework and I have used the firestore database to store the user information. Then, I have used Docker to create DockerFile for registration application and build that DockerFile to create docker images. After that, I pushed the docker images to the google artifact repository from terminal which is used to create cloud run services. Finally, I deployed that image from artifact repository by using cloud run service. I have done similar steps for other two apps which are container2 for Login app and container3 for State app. All the deployed links are available at the start of the report for part-b.

References :

[1]M. A. Sullivan, Firebase: A novel of wartime Vietnam suspense and romance. Mark Anthony Sullivan, 2017.

[2]“Install the gcloud CLI,” Google Cloud. [Online]. Available: https://cloud.google.com/sdk/docs/install. [Accessed: 18-Oct-2022].

[3]Wikipedia contributors, “Docker (software),” Wikipedia, The Free Encyclopedia, 15-Oct-2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=1116208227.

[4]“Artifact registry,” Google Cloud. [Online]. Available: https://cloud.google.com/artifact-registry. [Accessed: 18-Oct-2022].

[5]“Cloud run,” Google Cloud. [Online]. Available: https://cloud.google.com/run. [Accessed: 18-Oct-2022].

[6] “Store Docker container images in Artifact Registry,” Google Cloud. [Online]. Available:https://cloud.google.com/artifact-registry/docs/docker/store-docker-container-images. [Accessed: 18-Oct-2022].