

CSCI 5410: Assignment 2: Part B:

Waleed R. Alhindi – B00919848

GitLab A2 Repository:

The A2-Part B code has been pushed to a GitLab repository under the “A2” branch, which can be found here: <https://git.cs.dal.ca/alhindi/csci5410-summer-23-b00919848/-/tree/A2>

OR

Under the “A2” folder in the “main” branch, which can be found here:
<https://git.cs.dal.ca/alhindi/csci5410-summer-23-b00919848/-/tree/main>

Additionally, the professor and all TAs have been granted “Maintainer” access to this GitLab repository. This includes assigning “Maintainer” roles to the following GitLab accounts:

- *@saurabh (Dr. Saurabh Dey)*
- *@mudgal (Ankush Mudgal)*
- *@bharatwaaj (Bharatwaaj Shankanarayanan)*
- *@rmacwan (Rahul Ashokkumar Macwan)*

Operations Performed:

1. Creating Firestore Collections:

Before we can begin implementing our 3 containerized microservices, we first need to create the two required Firestore [1] collections: “Reg” to store registration information about users, and “state” which stores information about users’ online status. As such, these collections will store the following data:

- Reg Collection:
 - Name
 - Email
 - Password
 - Location

Note that Name must be unique as it will serve as the ID for corresponding documents in the “state” collection. This design decision was made since we need a way to uniquely identify each user. While the email field could be used instead of name, the name field was chosen due to its human-readability.

- State Collection:
 - OnlineStatus (Boolean value denoting whether a user is online)

- LastUpdated (Timestamp of when OnlineStatus was last updated)

Note that the document IDs in this collection are the names of the users associated with the documents.

Thus, we first create a Firestore database. Since we require our microservices to service requests coming from the public internet, we start the database in test mode, which allows public access to the database, as seen in Figure 1:

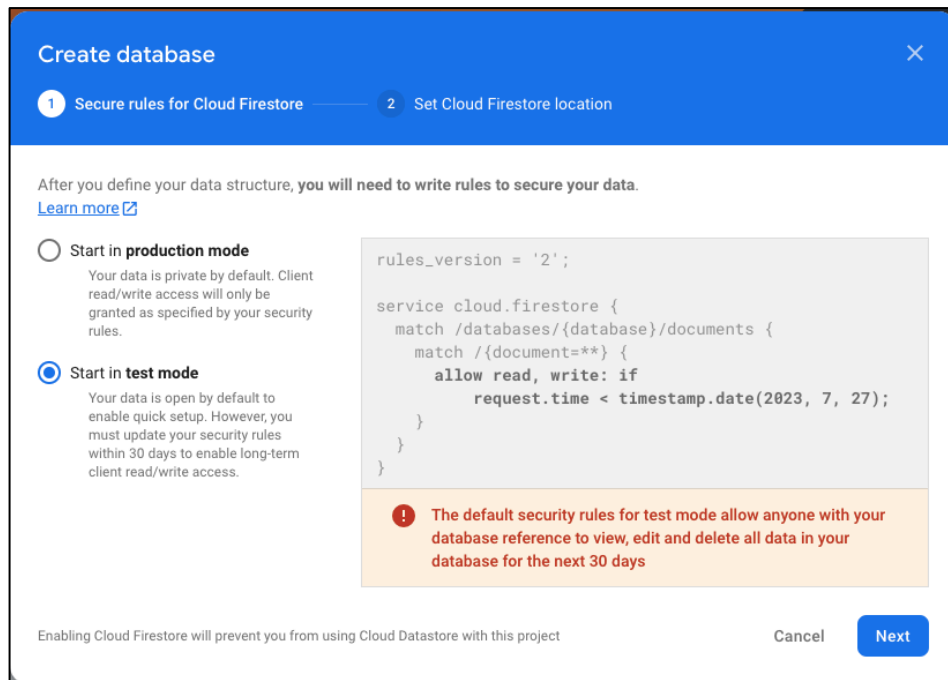


Fig 1. Creating Firestore Database in Test Mode [1]

We keep the database's location as the default "nam5 (United States)" as seen in Figure 2:

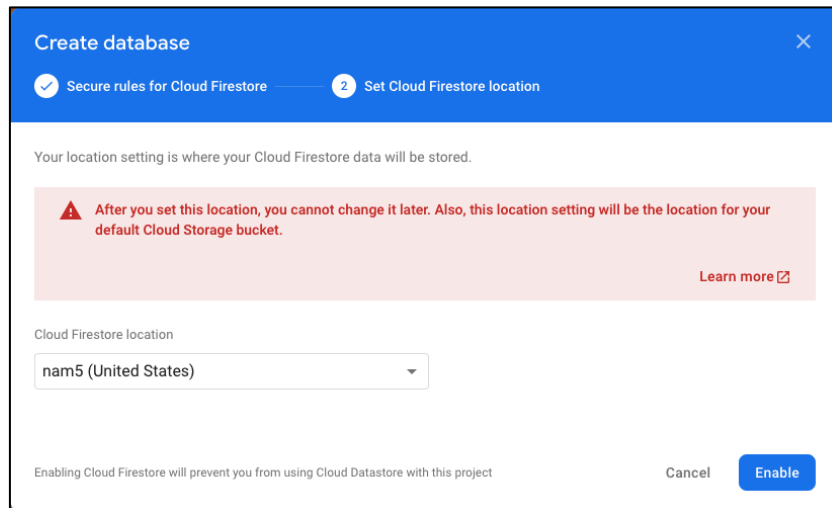


Fig 2. Firestore Database's Location [1]

Upon creating the database, we are greeted with an empty database with no collections as seen in Figure 3:

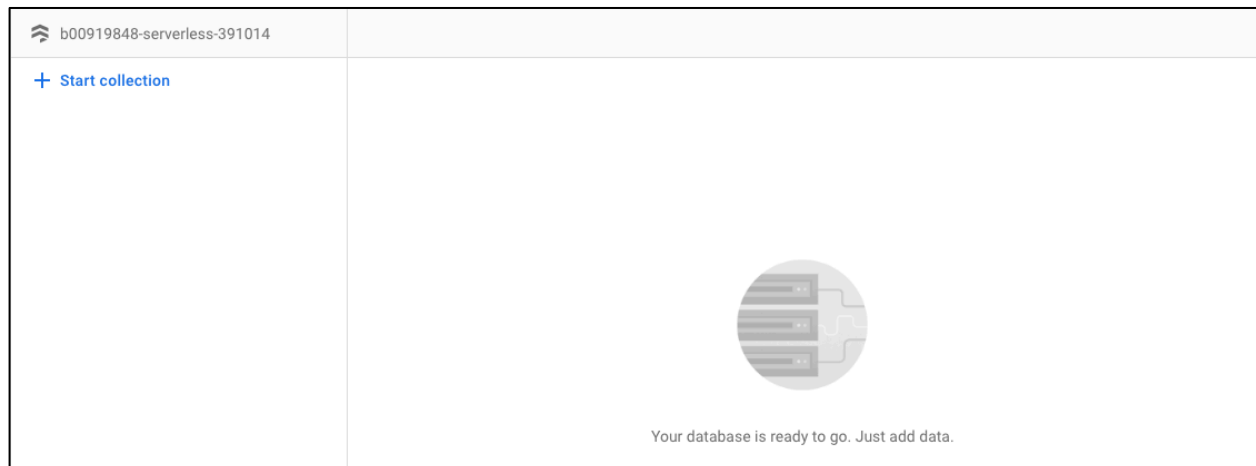


Fig 3. Newly Created Empty Firestore Database [1]

Then, we can go ahead and create the “Reg” and “state” collections. Starting with the “Reg” collection, we create the collection and add an initial document as seen in Figure 4:

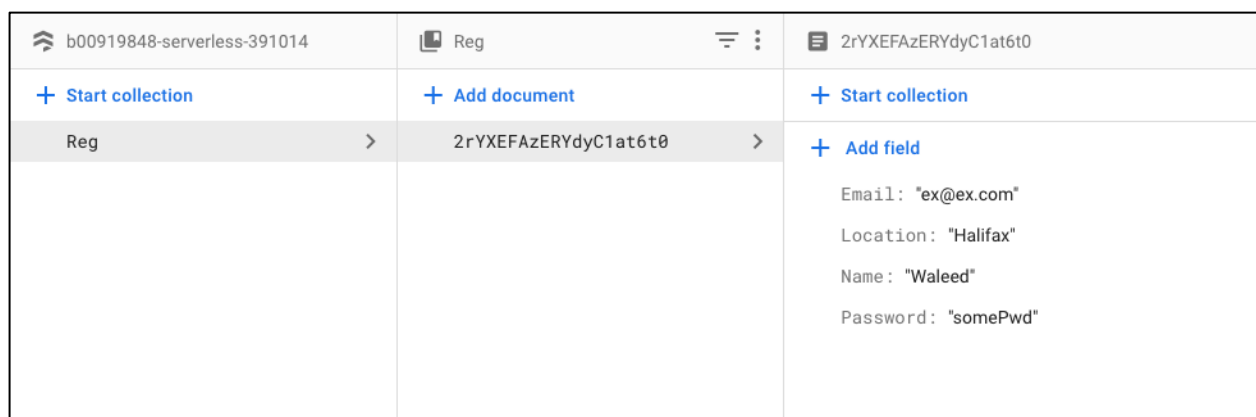


Fig 4. “Reg” Collection Created [1]

Similarly, the “state” collection was created, and an initial document is added as seen in Figure 5:

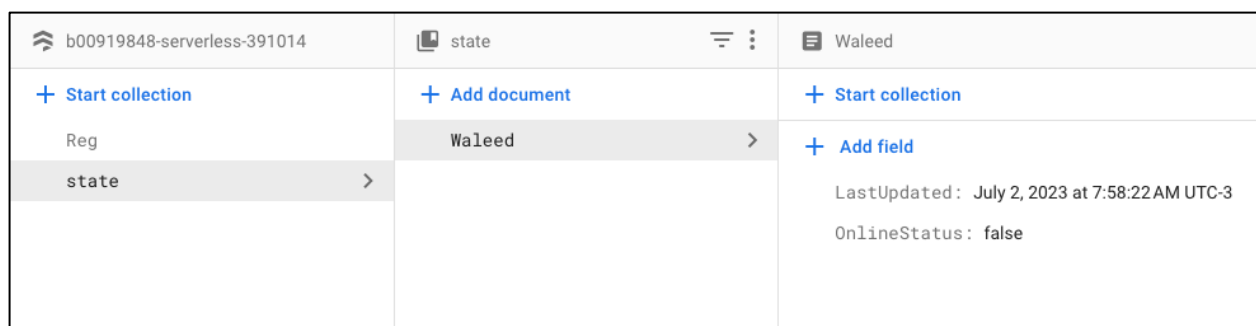


Fig 5. “state” Collection Created [1]

2. Composing Containers' Source Code:

Now that our Firestore collections have been configured, we need to implement each of the 3 microservices. The 3 microservices will be containerized React [2] applications that perform CRUD operations on the Firestore collections to simulate the process of a user registering, logging in, and logging out. Therefore, before we can begin implementing these microservices, we first need to register our app in our Firestore project and get our credentials. So, we navigate to “Project Overview”, click on “Add App”, then click on the web app symbol. We register our application as “SDP-A2”, as seen in Figure 6, which provides us with the code snippet needed to configure and initialize the Firestore SDK as seen in Figure 7:

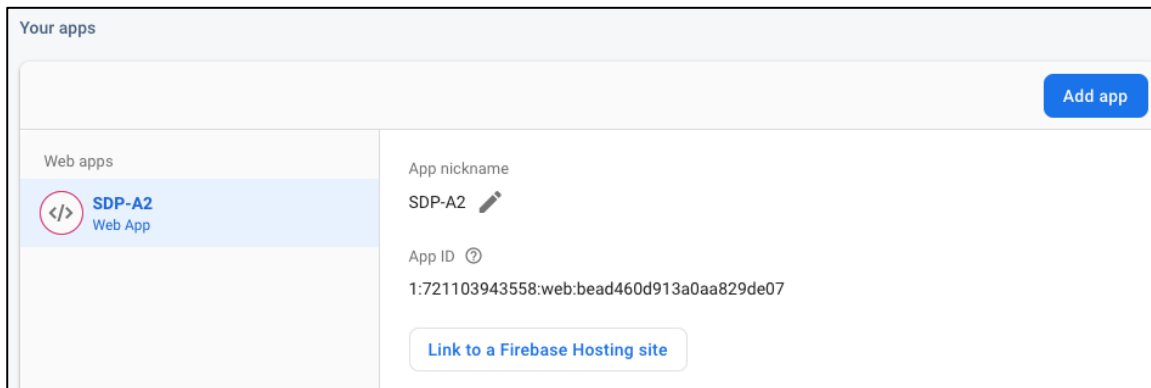


Fig 6. App “SDP-A2” Registered in Firestore Project [1]



Fig 7. Generated Firestore Configuration Code Snippet [1]

This code snippet will be used in all 3 of our microservices to facilitate access to our Firestore database. However, the credentials' values *will not be hard-coded* into our source code as that would expose our keys. Rather, we will store these credentials in a “credentials.json” file that will be added to our “.gitignore” file, and load in the values from there. Thus, the code snippet provided above was adapted to not expose credentials as seen in Figure 8:

```
const creds = require('./credentials.json');
let apiKey = creds['apiKey'];
let authDomain = creds['authDomain'];
let projectId = creds['projectId'];
let storageBucket = creds['storageBucket'];
let messagingSenderId = creds['messagingSenderId'];
let appId = creds['appId'];
let measurementId = creds['measurementId'];

const firebaseConfig = {
  apiKey: apiKey,
  authDomain: authDomain,
  projectId: projectId,
  storageBucket: storageBucket,
  messagingSenderId: messagingSenderId,
  appId: appId,
  measurementId: measurementId
};

const app = initializeApp(firebaseConfig);
const db = getFirestore(app);
```

Fig 8. Loading Credentials from a JSON File Instead of Hard-Coding

The code snippet in Figure 8 is common amongst the 3 containers, being present in container 1's Registration.js, container 2's Login.js, and container 3's LandingPage.js, since all 3 container microservices need to access our Firestore collections. Beyond that similarity, however, each container serves a different function. As such, the implementation of each container's key algorithms is outlined below:

A. Container 1:

Container 1 is responsible for registering users. Thus, it will display a registration form to the user, which requires them to provide their name, email, password, and location. Upon submitting the form, the application will validate the inputs and, if valid, will create a new document in the “Reg” collection to store the user's information as well as creating a

new document in the “state” collection to store information about the user’s online status, which is set as false upon creation. The user is then redirected to container 2’s log in page.

The pseudocode of the above process, which corresponds to the “ReadFromFS” method in container 1’s “Registration.js” file is as follows:

Get Name, Email, Password, and Location input values

Get all documents from “Reg” collection

Loop through “Reg” documents

If this document’s name matches the name entered by user

Notify user that name is already in use and must be unique

Return

Add new document in “Reg” containing user information

Add new document in “state” containing user online status

Redirect to container 2’s log in page

Additionally, container 1’s Registration.js’s “ReadFromFS” source code is outlined below:

```
const ReadFromFS = async(event) => {
  event.preventDefault();
  let name = event.target.name.value;
  let email = event.target.email.value;
  let pass = String(event.target.pass.value);
  let location = event.target.location.value;
  let data = null;
  /*CITATION NOTE:
  The following code used to retrieve and map all documents in a Firestore collection
  in React was adapted from the following source.
  URL: https://www.freecodecamp.org/news/how-to-use-the-firebase-database-in-react/
  */
  await getDocs(collection(db,"Reg")).then((snapshot)=>{
    data = snapshot.docs.map((doc) => ({id:doc.id, ...doc.data()}));
  })
  for(var i=0; i<data.length; i++){
    if(name.trim().toLowerCase()==data[i]['Name'].trim().toLowerCase()){
      alert("\nError: [Names must be unique]\n\nThe name you have chosen is already in
      use. Please try another name!");
      return;
    }
  }
  alert("Registration Successful!\nRedirecting to login page....");
  /*CITATION NOTE:
```

```

The following code used to create a document in a Firestore collection
in React was adapted from the following source.
URL: https://www.freecodecamp.org/news/how-to-use-the-firebase-database-in-react/
*/
try{
  await addDoc(collection(db, "Reg"), {
    Name: name,
    Email: email,
    Password: pass,
    Location: location
  });
  /*CITATION NOTE:
  The following code used to updated a specific, existing document in a
  Firestore collection in React was adapted from the following source.
  URL: https://stackoverflow.com/questions/56406406/javascript-date-to-firestore-timestamp
  */
  var now = Timestamp.fromDate(new Date());
  await setDoc(doc(db, "state", name), {
    OnlineStatus: false,
    LastUpdated: now
  });
}
catch(e){
  console.error(e);
  alert("Error: There was an issue uploading your data to Firstore: " + e);
}
window.location.href = 'https://container2-civ3vqnmnq-uc.a.run.app';
}

```

B. Container 2:

Container 2 serves as a log in microservice that accepts a user's name and password as inputs and validates them against the document in the "Reg" collection. If the name and/or password are incorrect, an error message is displayed to the user. However, upon successful sign in, the user's corresponding document in the "state" collection is updated to set OnlineStatus as true and set the LastUpdated timestamp to the current time. After which, the user is redirected to container 3's page.

The pseudocode of the above process, which corresponds to the "ValidateUser" method in container 2's "Login.js" file, is outlined below:

Get Name and Password input values

Get all documents from "Reg" collection

Loop through "Reg" documents

If this document's name matches the name input value

If this document's password matches the password input value

Update corresponding "state" document to indicate online

Redirect to container 3's page

Else

Notify user that password is incorrect

Return

Notify user that the name input is not registered

Additionally, container 2's Login.js's "ValidateUser" source code is outlined below:

```
const ValidateUser = async (event) => {
  event.preventDefault();
  let name = event.target.name.value;
  let pass = String(event.target.pass.value);
  let data = null;
  let status = null;
  var now = Timestamp.fromDate(new Date());
  /*CITATION NOTE:
  The following code used to retrieve and map all documents in a Firestore collection
  in React was adapted from the following source.
  URL: https://www.freecodecamp.org/news/how-to-use-the-firebase-database-in-react/
  */
  await getDocs(collection(db, "Reg")).then((snapshot) => {
    data = snapshot.docs.map((doc) => ({id: doc.id, ...doc.data()}));
  }).then(async () => {
    for (var i = 0; i < data.length; i++) {
      if (name.trim().toLowerCase() === data[i]['Name'].trim().toLowerCase()) {
        if (pass === data[i]['Password']) {
          /*CITATION NOTE:
          The following code used to update a specific, existing document in a
          Firestore collection in React was adapted from the following source.
          URL: https://stackoverflow.com/questions/56406406/javascript-date-to-
          firestore-timestamp
          */
          await updateDoc(doc(db, "state", name), {
            OnlineStatus: true,
            LastUpdated: now
          });
          await updateDoc(doc(db, "Session", "currSession"), {
            Name: name
          });
          alert("Log In Successful!\nRedirecting to Home Page....");
          window.location.href = 'https://container3-civ3vqnmnq-uc.a.run.app';
          return;
        }
      }
    }
  });
}
```



```

    }
    alert("Error: Incorrect password!\nPlease verify your password and try
again!");
    return;
  }
}
alert("Error: Username not registered!\nMake sure you are typing in your name
correctly!");
return;
})
}

```

C. Container 3:

Container 3 can be thought of as a landing page that users are redirected to after logging in, which displays a greeting message with the user's name as well as a list of all other users who are online by retrieving all the documents from the "state" collection that have their OnlineStatus as true. Additionally, the user can log out using the provided "Log Out" button. Upon clicking the log out button, the user's corresponding documents in the "state" collection is update to set OnlineStatus as false and update the LastUpdated field to the current time. The user is then redirected back to the registration page.

The pseudocode of the listing all online users, which corresponds to the "fetchData" method in container 3's "LandingPage.js" file, is as follows:

Get all documents from "state" collection

Initialize empty "onlineUsers" array

Iterate through all "state" documents

If this document's OnlineStatus is true

Add this document's ID (which is the user's name) to "onlineUsers" array

Store the "onlineUsers" array in the React Components state's "allUsers" field

Note that the "fetchData" method is invoked before rendering the page (i.e., inside ComponentDidMount{..}) and that the "onlineUsers" array is then rendered as a list by implementing the following code in the component's render:

```

<u>Other Users That Are Online:</u>
<ul>
  {this.state.allUsers.map((user) => (
    | <li><i>{user}</i></li>
  ))}
</ul>

```

Fig 9. Programmatically Listing Online Users

Additionally, container 3's "fetchData" source code is outlined below:

```
fetchData = async() => {
  let apiKey = creds['apiKey'];
  let authDomain = creds['authDomain'];
  let projectId = creds['projectId'];
  let storageBucket = creds['storageBucket'];
  let messagingSenderId = creds['messagingSenderId'];
  let appId = creds['appId'];
  let measurementId = creds['measurementId'];
  const firebaseConfig = {
    apiKey: apiKey,
    authDomain: authDomain,
    projectId: projectId,
    storageBucket: storageBucket,
    messagingSenderId: messagingSenderId,
    appId: appId,
    measurementId: measurementId
  };
  const app = initializeApp(firebaseConfig);
  const db = getFirestore(app);
  /*CITATION NOTE:
  The following code used to retrieve and map all documents in a Firestore
collection
  in React was adapted from the following source.
  URL: https://www.freecodecamp.org/news/how-to-use-the-firebase-database-in-react/
  */
  await getDocs(collection(db, "Session")).then((snapshot)=>{
    let data = snapshot.docs.map((doc) => ({id:doc.id, ...doc.data()}));
    this.setState({currentUser:data[0]['Name']});
  })
  await getDocs(collection(db,"state")).then((snapshot)=>{
    let onlineUsers = [];
    let states = snapshot.docs.map((doc) => ({id:doc.id, ...doc.data()}));
    for(var x=0; x<states.length; x++){
      if(states[x]['OnlineStatus']==true){
        onlineUsers.push(states[x]['id']);
      }
    }
    this.setState({allUsers: onlineUsers});
  });
}
```

3. Building and Publishing Container Images to Artifact Registry:

Now that the source code of each of the 3 containers has been implemented, we can move onto containerizing the applications and publishing them to Google Container Registry [3]. To do this, we first need to create a Dockerfile in the root folder of each container. The Dockerfiles of each

containers are identical since they will all have the same build process of installing dependencies, exposing port 6000, and running npm start as seen in Figure 10:

```
container1 > Dockerfile
1 FROM node:latest
2 WORKDIR /app
3 COPY package*.json ./
4 RUN npm install
5 COPY . .
6 EXPOSE 6000
7 CMD ["npm", "start"]
```

Fig 10. Container 1's Dockerfile

Thus, each container's folder structure is similar to container 1's folder structure, which is showcased in Figure 11:

```

  container1
  > node_modules
  > public
  > src
  .gitignore
  Dockerfile
  package-lock.json
  package.json
  README.md
```

Fig 11. Container 1's Folder Structure

Next, we build and publish an image of each of the 3 containers to Google Container Registry [3]. To do this, the folders of each container were uploaded to a Cloud Shell [4] instance as seen in Figure 12:

```
waleed woo alhindi@cloudshell:~ (b00919848-serverless-391014)$ ls
cont1 cont2 cont3 deployment.yaml initial_practice pvc.yaml README-cloudshell.txt
waleed woo alhindi@cloudshell:~ (b00919848-serverless-391014)$
```

Fig 12. Uploaded Container Folders to Cloud Shell [4]

Then, for each of the three container folders, we navigate into them and run the following commands to build and push an image to Google Container Registry [3]:

```
waleed_woo_alhindi@cloudshell:~ (b00919848-serverless-391014)$ ls
cont1 cont2 cont3 deployment.yaml initial_practice pvc.yaml README-cloudshell.txt
waleed_woo_alhindi@cloudshell:~ (b00919848-serverless-391014)$ cd cont3
waleed_woo_alhindi@cloudshell:~/cont3 (b00919848-serverless-391014)$ docker build -t gcr.io/b00919848-serverless-391014/container3:latest .
[+] Building 64.3s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 146B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/node:latest
=> [1/5] FROM docker.io/library/node:latest@sha256:2f0b0c15f97441defa812268ee943bbfaaf666ea6cf7cac62ee3f127906b35c6
=> => resolve docker.io/library/node:latest@sha256:2f0b0c15f97441defa812268ee943bbfaaf666ea6cf7cac62ee3f127906b35c6
```

Fig 13. Building Container 2's Image

```
waleed_woo_alhindi@cloudshell:~/cont3 (b00919848-serverless-391014)$ docker push gcr.io/b00919848-serverless-391014/container3:latest
The push refers to repository [gcr.io/b00919848-serverless-391014/container3]
ef832e1d9f33: Pushed
5a8c307b9717: Pushed
3feaf52434a0: Pushed
97ef523ed2a1: Pushed
```

Fig 14. Pushing Built Image to Google Container Registry

Once all 3 containers have been built and published, we can verify that the images were pushed successfully by inspecting the Google Container Registry console [3], which displays the following:

b00919848-serverless		
<div>Filter Enter property name or value</div>		
Name ↑	Hostname ?	Visibility ?
 container1	gcr.io	Private
 container2	gcr.io	Private
 container3	gcr.io	Private

Fig 15. Container Image Repositories Created Successfully


container1					
<div>gcr.io > b00919848-serverless-391014 > container1</div>					
<div>Filter Enter property name or value</div>					
<input type="checkbox"/>	Name	Tags	Virtual Size ?	Created	Uploaded ↓
<input type="checkbox"/>	 2b3bc2357ef8	latest	518.3 MB	3 hours ago	3 hours ago

Fig 16. Published Image in Container 1 Repository

As seen in Figures 15 and 16, an image repository is created for each of the 3 containers which each house the corresponding container's published image.

4. Deploying Containers Using Cloud Run:

Now that we have built and published images of our containers, we can move on to deploying each container as a service using Google Cloud Run [5]. To do this, we navigate to the Cloud Run console and click on "Create Service", where we need to select a container image to deploy and update the container port to 6000 as seen in Figure 17, 18, and 19:

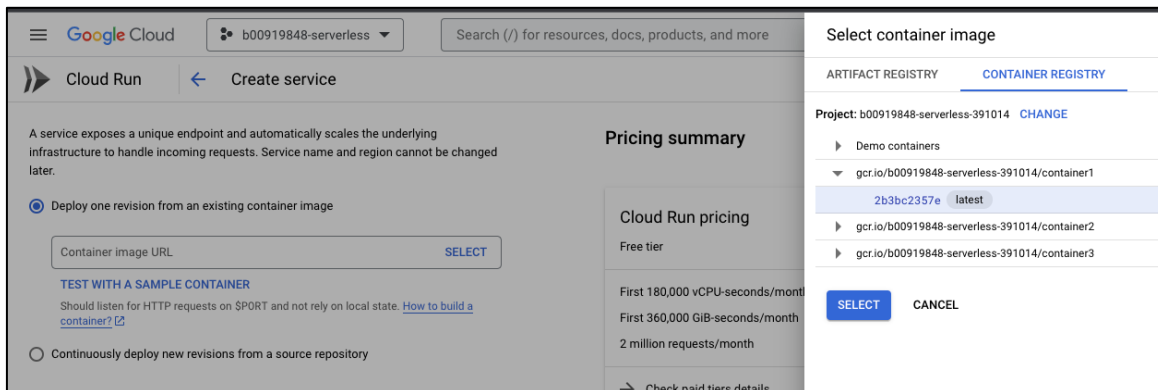


Fig 17. Selecting a Container Image from Container Registry

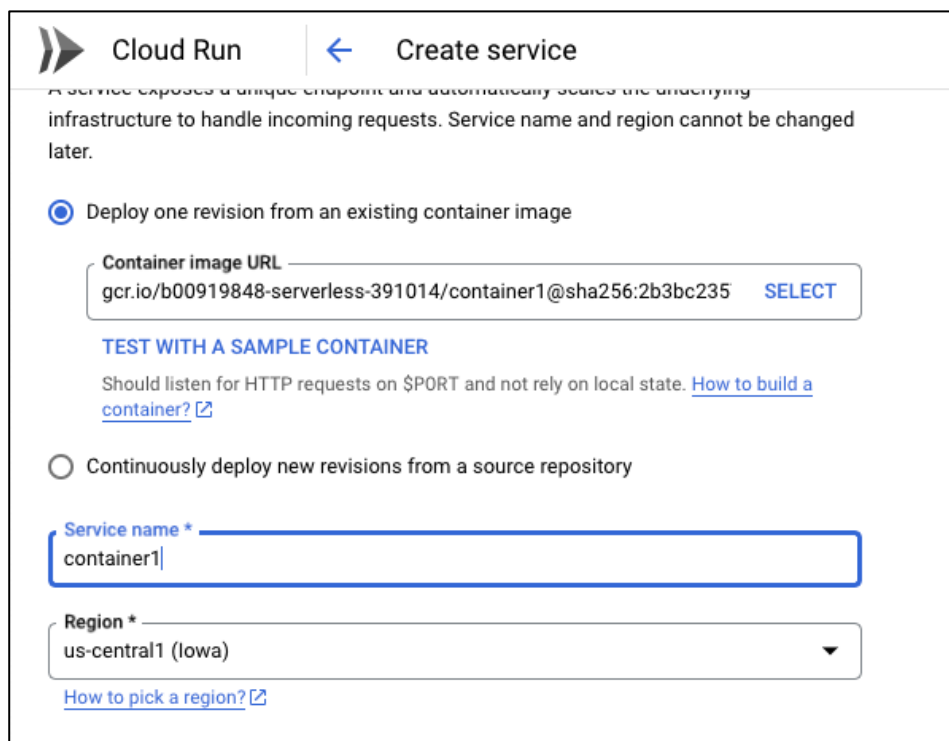


Fig 18. Selected Container Image

Container, Networking, Security

CONTAINER

NETWORKING

SECURITY

General

Container port

6000

Requests will be sent to the container on this port. We recommend listening on \$PORT instead of this specific number.

Fig 19. Setting Service’s Container Port to 6000

We then deploy each service as seen in Figure 20 and verify that all services are up and running by observing the Cloud Run console as seen in Figure 21:

Cloud Run

Service details

EDIT & DEPLOY NEW REVISION

SET UP CONTINUOUS DEPLOYMENT

container1

Region: us-central1

URL: <https://container1-clv3vqnmq-uc.a.run.app>

METRICS

SLOS

LOGS

REVISIONS

NETWORKING

SECURITY

TRIGGERS

INTEGRATIONS

PREVIEW

YAML

Revisions

MANAGE TRAFFIC

Filter

Filter revisions

Name	Traffic	Deployed	Revision URLs (tags)	Actions
container1-00003-8r6	100% (to latest)	1 minute ago	+	

container1-00003-8r6

Deployed by waleed.woo.alhindi@gmail.com using Cloud Console

CONTAINERS

VOLUMES

NETWORKING

SECURITY

YAML

General

CPU allocation

CPU is only allocated during request processing

Startup CPU boost

Disabled

Concurrency

80

Request timeout

300 seconds

Execution environment

First generation (Default)

Autoscaling

Max instances

100

Image URL

<gcr.io/b00919848-serverless-391014/container1@sha256-...>

Port

6000

Build

(no build information available)

Source

(no source information available)

Fig 20. Container 1 Service Successfully Deployed

SERVICES

JOBS

Filter

Filter services

<input type="checkbox"/>	<input checked="" type="checkbox"/>	Name ↑	Req/sec ?	Region	Authentication ?	Ingress ?
<input type="checkbox"/>	<input checked="" type="checkbox"/>	container1	0	us-central1	Allow unauthenticated	All
<input type="checkbox"/>	<input checked="" type="checkbox"/>	container2	0	us-central1	Allow unauthenticated	All
<input type="checkbox"/>	<input checked="" type="checkbox"/>	container3	0	us-central1	Allow unauthenticated	All

Fig 21. All Container Services Deployed Successfully

5. *Verifying Deployment:*

To verify that the three services are working correctly, we access container 1's URL and carry out the expected, normal workflow of registering a user, signing in, and logging out while verifying that our "Reg" and "state" collections are updated correctly. So, we access container 1's URL, which displays the following form:

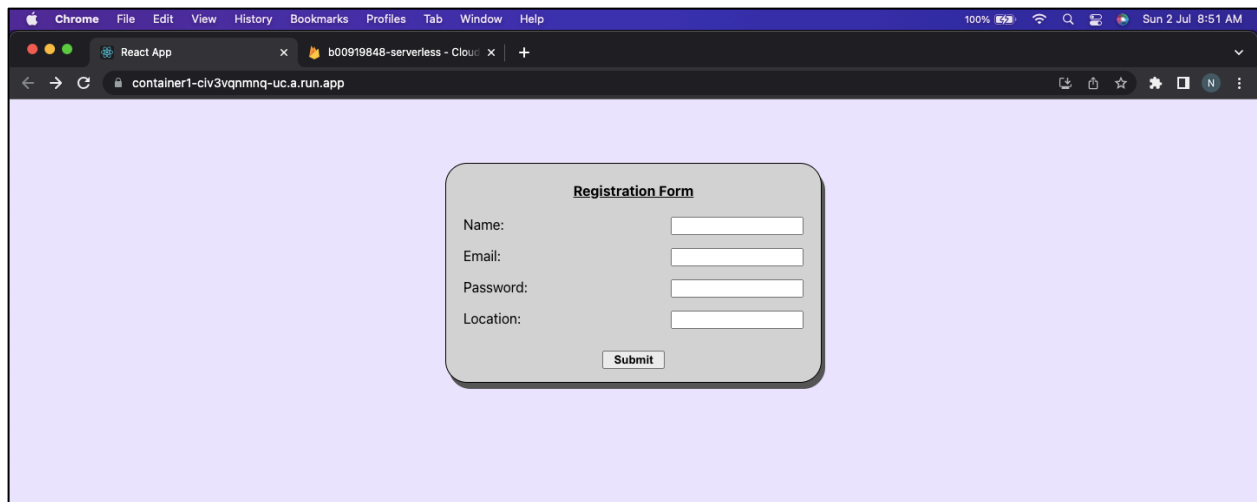
A screenshot of a web browser showing a registration form. The browser's address bar displays 'container1-civ3vqnmnq-uc.a.run.app'. The form is titled 'Registration Form' and contains four input fields: 'Name:', 'Email:', 'Password:', and 'Location:'. Each field has a corresponding text input box. Below the fields is a 'Submit' button. The entire form is centered on a light purple background.

Fig 22. Container 1 Registration Page

We enter "TEST" into all the input fields and hit "Submit" which displays the following registration success message and redirects the user to container 2's page:

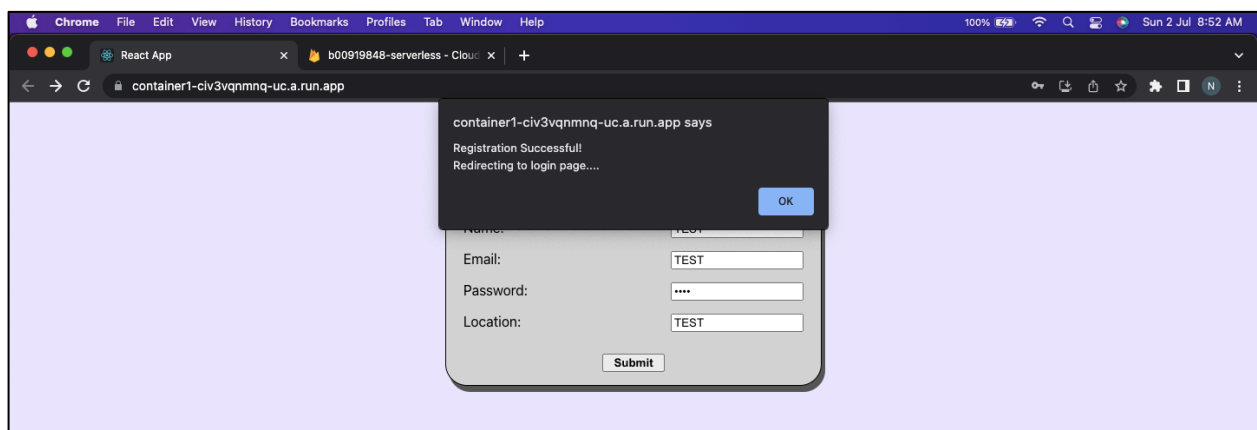
A screenshot of the same web browser showing the registration form. A dark modal dialog box is overlaid on top of the form. The dialog box contains the text: 'container1-civ3vqnmnq-uc.a.run.app says', 'Registration Successful', and 'Redirecting to login page...'. There is an 'OK' button in the top right corner of the dialog box. The registration form is visible behind the dialog box, with the input fields now containing the text 'TEST'.

Fig 23. Successful Registration in Container 1 Page

However, before we move onto container 2, let us verify that a document was created in the "Reg" and "state" collections that corresponds with the newly registered user:

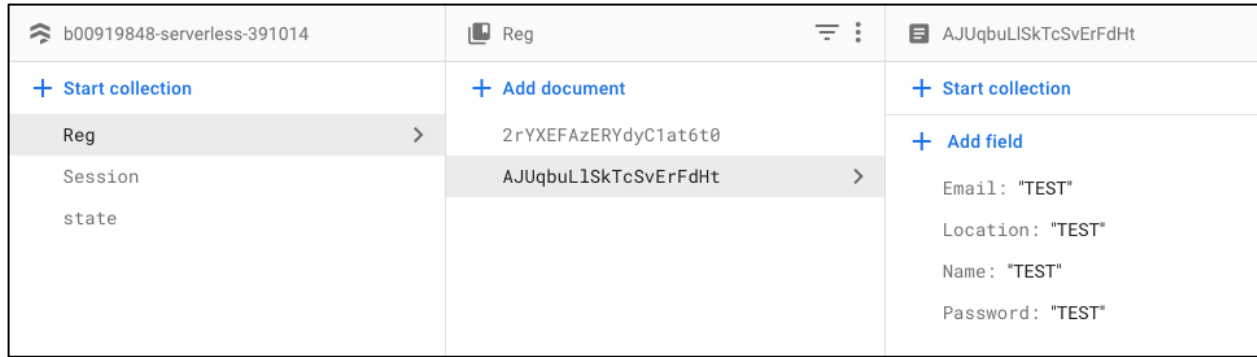


Fig 24. New Document Added to “Reg” Collection

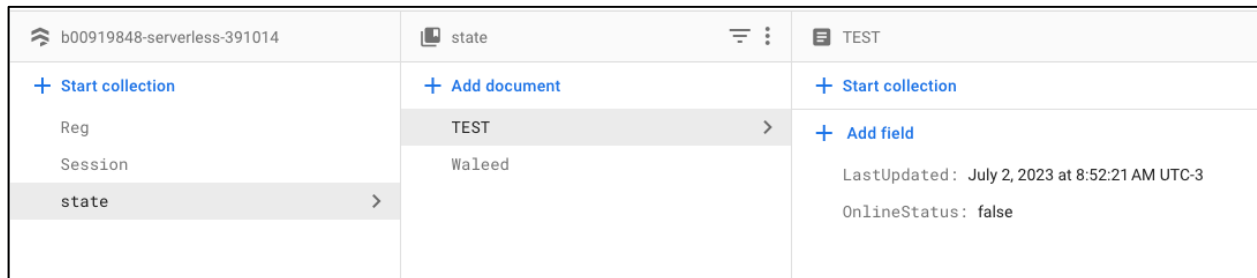


Fig 25. New Document Added to “state” Collection

Upon being redirected to container 2’s page (after clicking “Submit” on container 1’s page in Fig 23), a log in form is displayed as seen below:

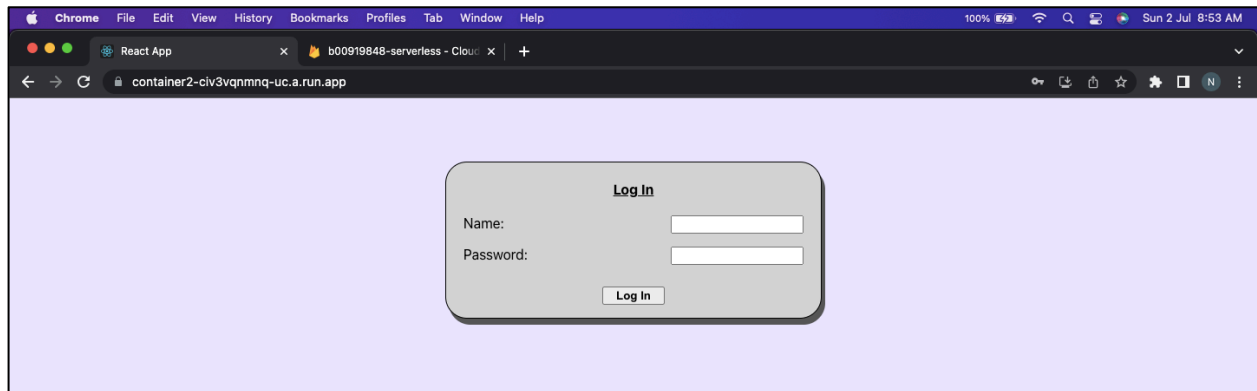


Fig 26. Container 2’s Log-In Page

We attempt to sign in using the user we just created by inputting “TEST” into the name and password inputs, then clicking log in – which displays a log in success message and redirects us to container 3’s page:

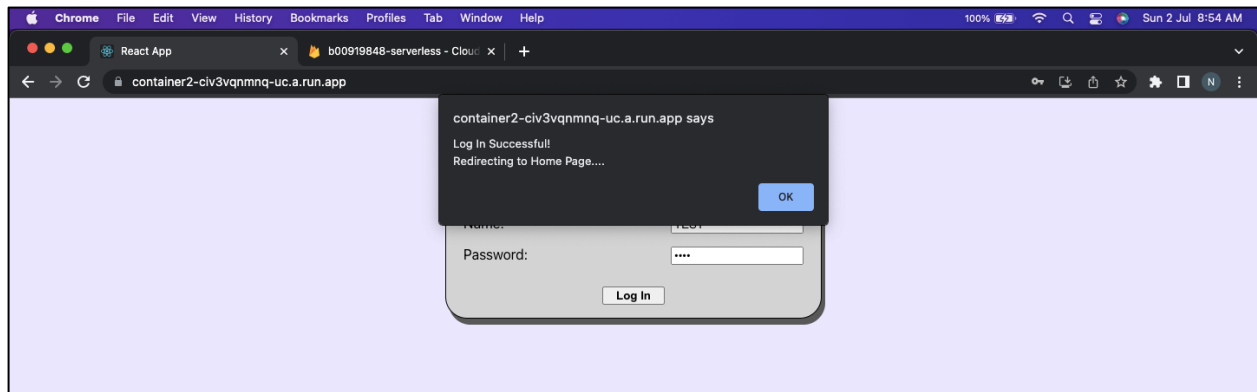


Fig 27. Successful Login Through Container 2

We must also verify that the document corresponding to the “TEST” user in the “state” collection has had its OnlineStatus updated to true since the user is now logged in. As seen in Figure 28, the corresponding document’s OnlineStatus has successfully been set to true:

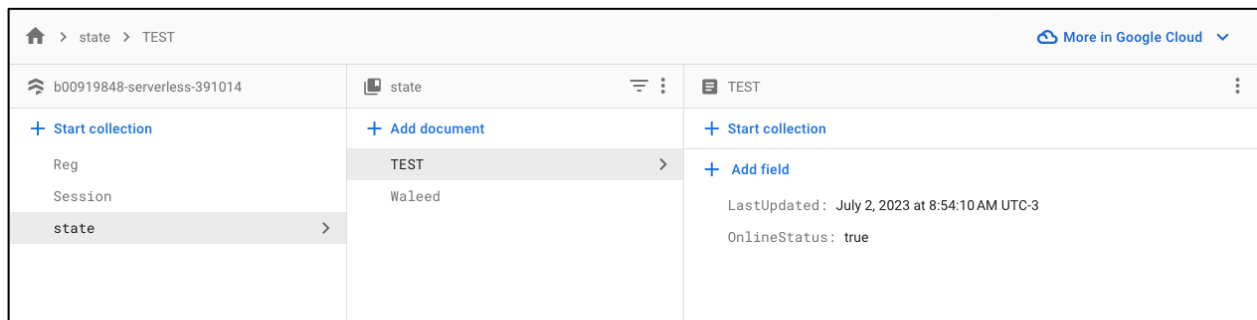


Fig 28. Successfully Updated OnlineStatus in the Corresponding “state” Collection’s Document

Lastly, upon being redirected to container 3’s page, a greeting message with the current user’s, “TEST”, name is displayed along with a “Log Out” button. Additionally, a list of all online users is displayed:

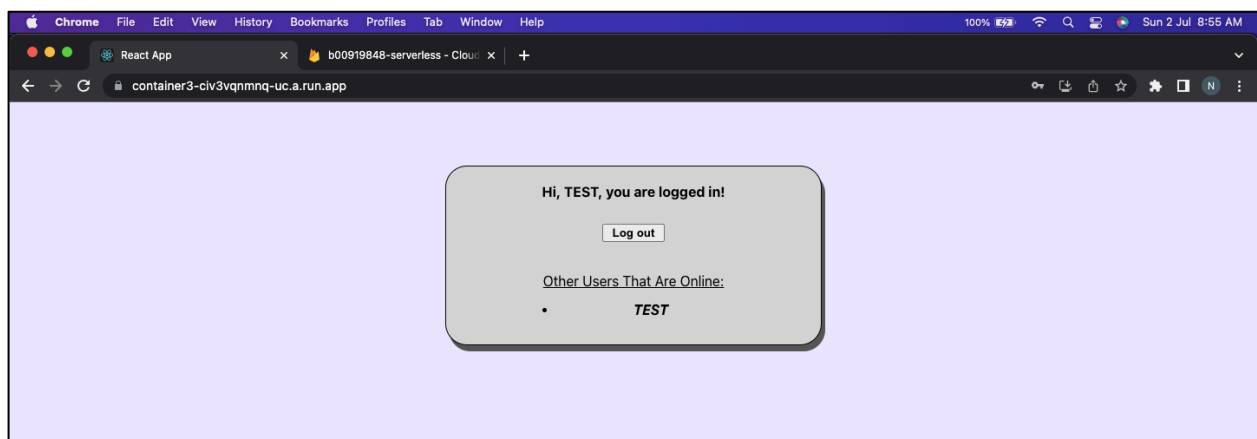


Fig 29. Container 3’s Page

However, since only one user, “TEST”, is online, the list of online users consists of only “TEST” even though another user exists in the collection. So, without logging out of “TEST”, let’s open a new tab and navigate to container 1’s page and register a new user, named “someNewUser” as seen in Figure 30, which creates a new document in the “Reg” and “state” collections as seen in Figure 31 and 32.

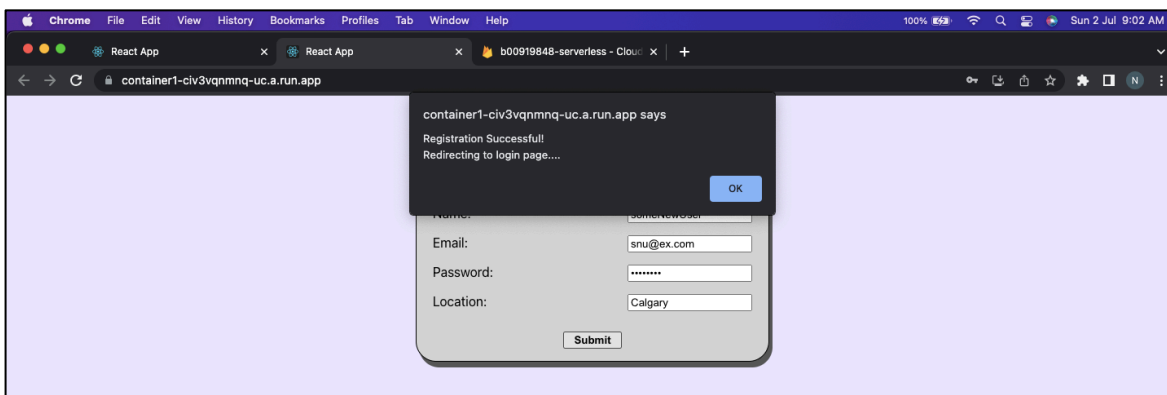


Fig 30. Registering New “someNewUser” User in Container 1’s Page

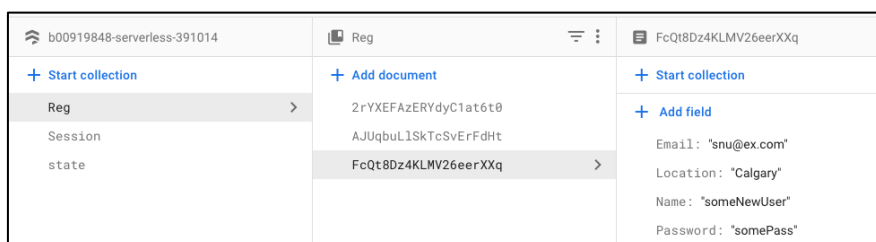


Fig 31. Corresponding “someNewUser” Document Created in “Reg” Collection

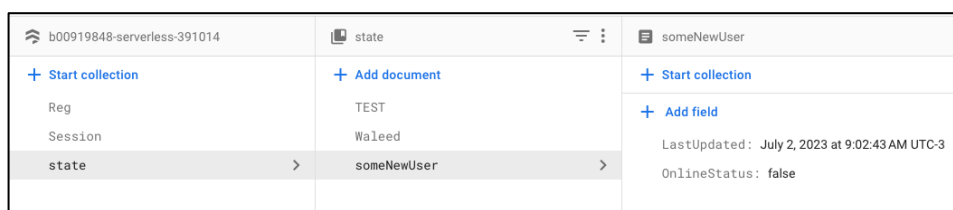


Fig 32. Corresponding “someNewUser” Document Created in “state” Collection

Then, we sign in using “someNewUser” credentials as seen in Figure 33:

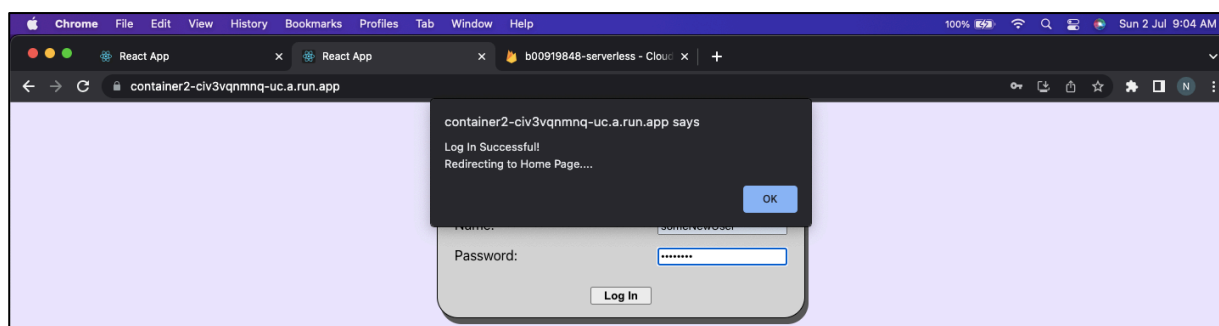


Fig 33. Successful “someNewUser” Log-In

When we are redirected to container 3's page, as depicted in Figure 34, we see that the list of online users now includes both "TEST" and "someNewUser".

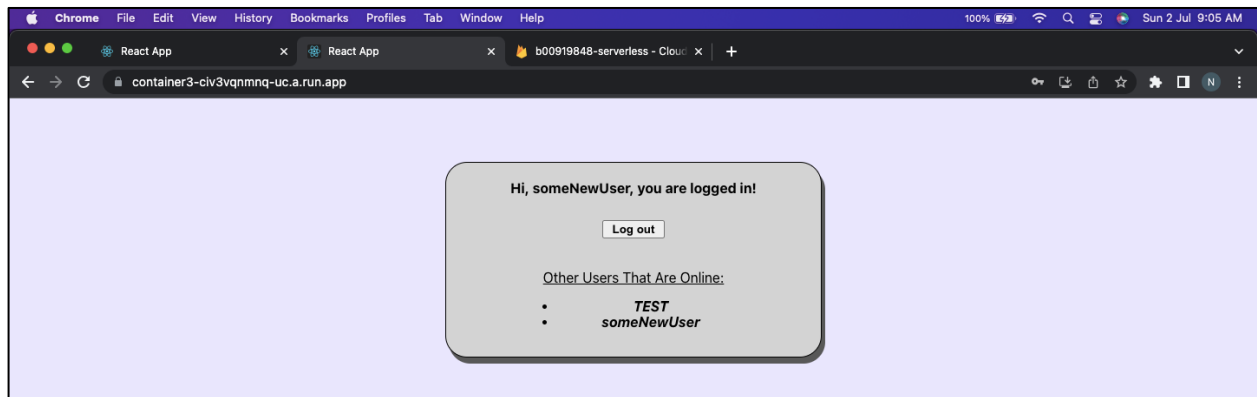


Fig 34. Container 3's List of Online Users when "TEST" and "someNewUser" are Logged In

Lastly, we can verify that the logout feature works correctly by logging out of the "someNewUser" account through clicking it the "Logout" button in Figure 35 and observing whether the corresponding document in the "state" collection is updated to have its OnlineStatus set to false, and whether we are successfully redirected to the registration page (container 1's page).

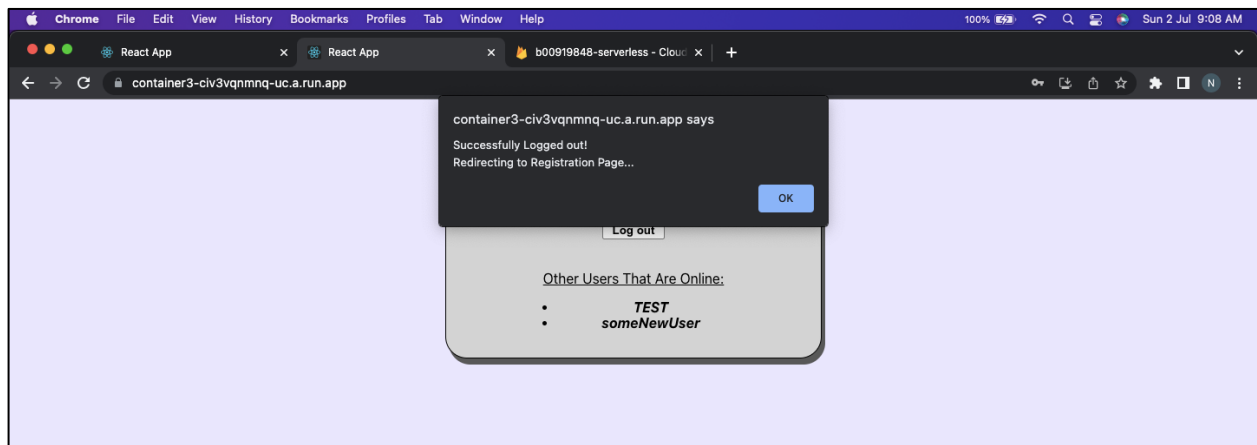


Fig 35. Successfully Logged Out of "someNewUser" Account

home > state > someNewUser More in Google Cloud		
b00919848-serverless-391014	state	someNewUser
+ Start collection	+ Add document	+ Start collection
Reg	TEST	+ Add field
Session	Waleed	LastUpdated: July 2, 2023 at 9:08:08 AM UTC-3
state >	someNewUser >	OnlineStatus: false

Fig 36. "someNewUser" OnlineStatus Set to False Upon Logging Out

To verify “someNewUser” has indeed logged out, let’s switch back to the tab where we were logged in as “TEST” (remember, we never logged out), where we can observe in Figure 37 that “someNewUser” is no longer in the list of online users since we just logged out of that account:

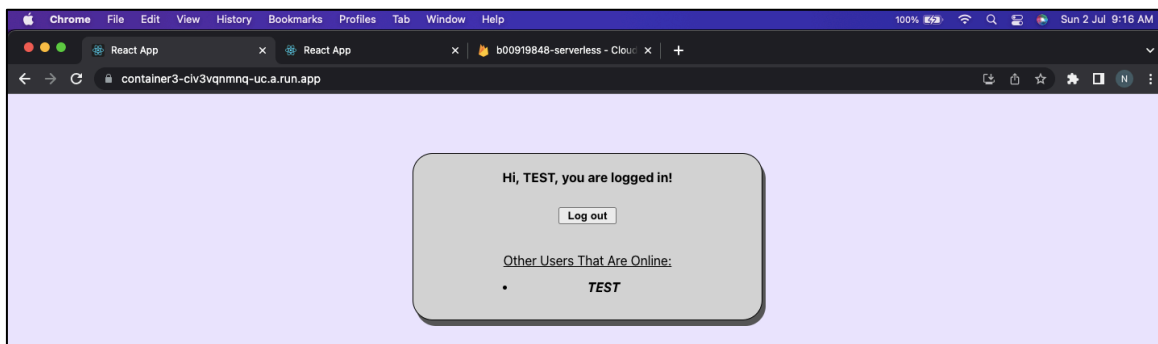


Fig 37. List of Online Users After “someNewUser” Has Logged Out

Thus, we have verified that our containerized microservices are rendering correctly, operating on Firestore collections as expected, and successfully redirecting to each other upon registration, log in, and log out.

6. Test Cases:

A. Registration Validation: Unique Name Input

As mentioned previously, the “Reg” collection documents’ “Name” fields need to be unique since they refer to the username/display name associated with each user and since this field is used to reference corresponding “state” collection documents. As such, in container 1’s registration page, an error message is displayed to the user rejecting their registration information if the name entered matches an existing “Reg” document’s name (case in-sensitive).

To test this, let us try to register a user named “test” while the user “TEST” (from the steps of the previous sections) already exists. As seen in Figure 38, the registration information is rejected with a message stating that the name field must be unique and that the name supplied is already in use.

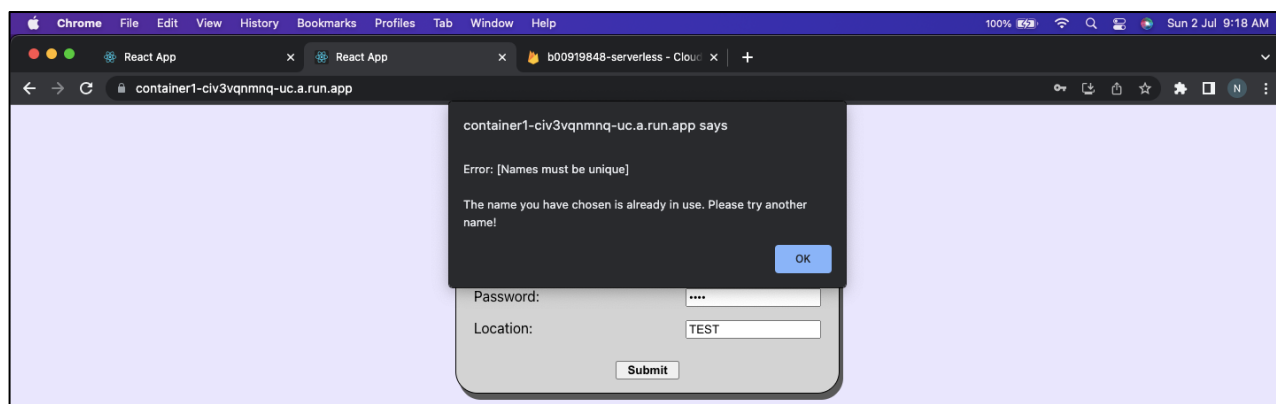


Fig 38. Registration Validation Rejects Name Already in Use

B. Log In Validation: Name Not Registered

Similarly, container 2's log in page validates that the name and password supplied match an existing "Reg" document. Thus, if the name entered does not match any "Reg" documents, the page alerts the user that the name entered has not been registered.

To test this, let's try to sign in using "UnkownUser" account as seen in Figure 40, which has never been registered. Figure 41 displays the error message informing the user that the name is not registered in the system.

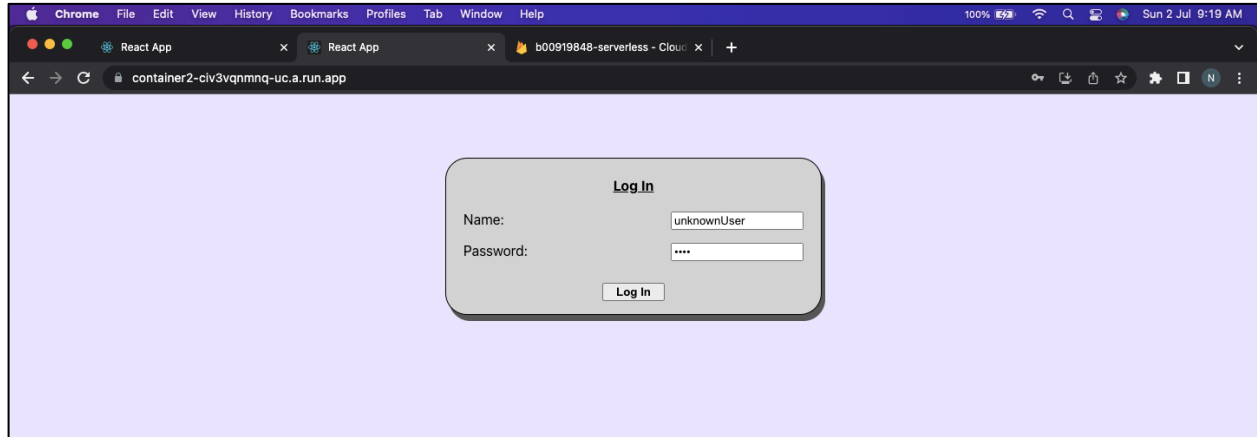


Fig 40. Logging In As unregistered "unknownUser"

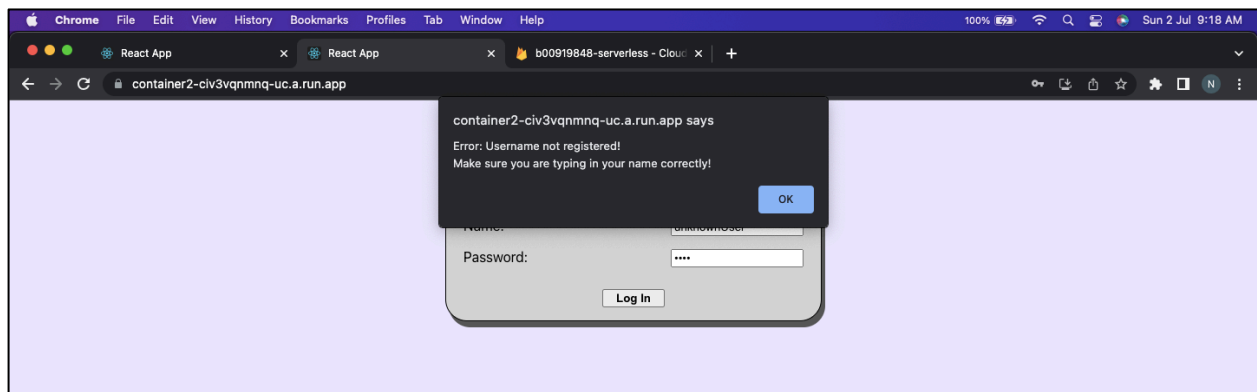


Fig 41. Login Validation Error Message Stating Name Not Registered

C. Log In Validation: Incorrect Password

Continuing off the last point, if the user enters a registered name, the program validates the user by checking whether the password supplied matches the password stored in the corresponding "Reg" document. If they do not match, the user is displayed with a generic incorrect password message.

Let us test this by attempting to log in as the existing "TEST" account, but supply an incorrect password. As seen in Figure 42 and 43, supplying an existing name and incorrect password alerts the user that the password they have entered is incorrect:

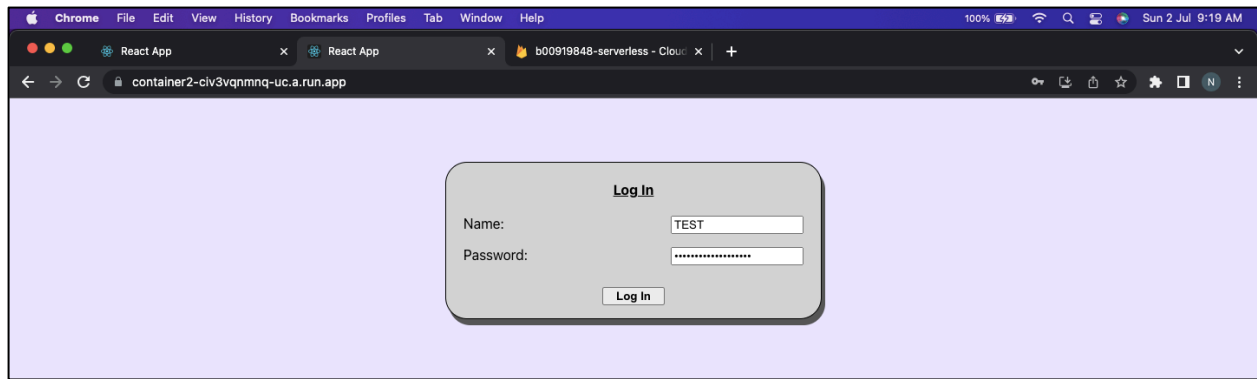


Fig 42. Logging in as “TEST” but with Incorrect Password

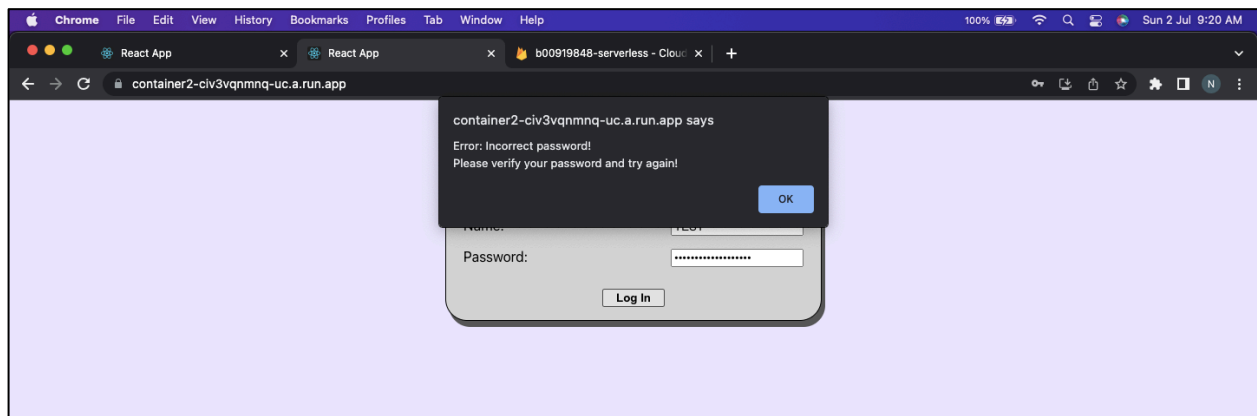


Fig 43. Incorrect Password Error Message Displayed

Observations About Utilized Technologies:

This assignment makes use of containerization, Firestore [1], Google Container Registry [3], and Google Cloud Run [5].

Beginning with containerization, as mentioned in previous sections, each microservice is built into a Docker container image that is pushed to Google Container Registry [3]. Containerization is a common approach to deploying applications, especially on Cloud Platforms, since it optimizes resource utilization by eliminating the need for a hypervisor and host OS, which reduces overhead. In addition to being more lightweight, containerization provides security and integrity since images are immutable and can be verified against their SHA digest. Moreover, containerization is well-suited for service-oriented architectures as they allow for a finer degree of modularity, which when utilized properly, leads to decreased coupling, easier time debugging, and increased flexibility and reusability since, like in this case, a system can house each of their underlying microservices in containers that can be efficiently revised and redeployed with minimal downtime.

However, after we build our container images, we need store them somewhere that can be accessed remotely since Cloud Run [5] will pull these images when deploying services. That's where Google Container Registry [3] comes in. It allows us to publish docker images to a private

container repository similarly to DockerHub. However, since we are deploying our microservices to Cloud Run [5], Container Registry has the additional advantage of already being integrated into the GCP environment, meaning it is easier to integrate with our other GCP services.

Lastly, once our container images were hosted in a repository, we still needed to deploy our containers as publicly accessible services that can handle incoming requests. In this case, Cloud Run [5] was used to deploy our images directly from Container Registry [3] with minimal configuration – yet another testament to the advantage of using Container Registry when deploying onto GCP rather than using external image repositories. While deploying onto Cloud Run is simple and straightforward, it still offers powerful and robust automated revisioning tools that can be used to build a complete CI/CD pipeline, where you can set a trigger to revise and redeploy services once a new image in the linked container repository is published.

Moreover, it is clear to see why these technologies are utilized commonly to develop service-oriented systems built upon underlying microservices, that are usually housed in some cloud environment. Although the technologies utilized in this assignment are offered by GCP, other cloud providers, like AWS, offer their own counterparts of similar services. Therefore, many of the concepts, steps, and procedures carry over to other cloud providers and, more generally, to how modern web services and online systems are deployed.

References:

- [1] Google for Developers, “Cloud Firestore,” *Google for Developers*. [Online], Available: <https://firebase.google.com/docs/firestore> [Accessed: June 27, 2023].
- [2] Meta Open Source, “React: The Library for Web and Native User Interfaces,” *Meta Open Source*. [Online], Available: <https://react.dev/> [Accessed: June 27, 2023].
- [3] Google Cloud, “Container Registry,” *Google Cloud*. [Online], Available: <https://cloud.google.com/container-registry/docs/> [Accessed: June 27, 2023].
- [4] Google Cloud, “Cloud Shell,” *Google Cloud*. [Online], Available: <https://cloud.google.com/shell> [Accessed: June 27, 2023].
- [5] Google Cloud, “Cloud Run,” *Google Cloud*. [Online], Available: <https://cloud.google.com/run/> [Accessed: June 27, 2023].