CSCI 5410 – Serverless Data Processing
Final Project Report
Project: Serverless B&B

Course Instructor: Dr. Saurabh Dey

Submitted by: Group 29

Group Member Name	Banner ID
Aditya Deepak Mahale	B00867619
Jayasree Kulothungan	B00894354
Rishika Bajaj	B00902713
Sai Chand Kolloju	B00897214
Sourav Malik	B00839958
Udit Gandhi	B00889579

Table of Contents

Project Links

GitLab repository link: https://git.cs.dal.ca/kolloju/csci5410_group29
Application link: http://csci5410-group29.s3-website-us-east-1.amazonaws.com/

Project and Architecture Overview

The Serverless Bed & Breakfast is a cloud-based hotel reservation system. This system provides the facility to reserve hotel accommodation, order food from the restaurant and allows customers to request a personalized trip package from the tour operator. Customers, Hotel Management, Kitchen, and Tour Operators are the four primary components of this system. Customers must register or login to use the other services provided by the application. Furthermore, customer oversees communicating with other services to register feedback, book tours, and place food orders etc. Users may also get online help with website navigation, room availability, booking management, and ordering food from the restaurant. Customers may use the Hotel Management service to book rooms or beds and engage with the kitchen to acknowledge their service. Kitchen service oversees taking customer orders, sending bills to hotel management, and preparing meals (only breakfast). Meal preparation necessitates inventory access, cooking, scheduling, and order management. The application also provides tour operator services which offers customers with a customized tour package. It also sends out created tour package information to customers through email [1].

Serverless B & B follows multi-cloud architecture paradigm by using Google Cloud Platform (GCP) and Amazon Web Services (AWS) (See Figure 1) [2]. This ensures application's high availability. The static assets generated by the React [3] frontend of our application are hosted in an S3 bucket [4] on AWS, where the application is deployed. This makes it possible for users to access the application online. The user data and application must be kept on Cloud FireStore [5] and Amazon DynamoDB [6], two services that, among other things, offer user registration and multi-factor user authentication. Using Amazon Cognito [7], which is supported by the DynamoDB and FireStore databases, users are authorised and managed [2].

The virtual help system for the application is run by Amazon Lex [8], which uses AWS Lambda [9] functions to fulfil user requests while utilising the DynamoDB data. Most of the application's logical processing is handled by AWS Lambda Functions and Google Cloud Functions [10], which replace the traditional server-oriented design. The Google Cloud Pub/Sub [11] service, which facilitates communication and the concurrent processing of requests throughout the application, supports this processing. Through the API Gateway service offered by both the GCP and AWS cloud providers, these cloud services are safely made available to the application. To create personalised tour packages and analyse client input, the application employs machine learning methods offered by GCP, more notably the Vertex AI [12] and NLP API [13] services. Reports on user login statistics are produced using the cloud services provided by Amazon CloudTrail [14] and Cloudwatch [15].

Core Components

The application will be developed using the below mentioned AWS and GCP services:

Module	Services
User Management	AWS: Cognito, Lambda, DynamoDB, API Gateway GCP: Firestore
Authentication	AWS: Cognito, DynamoDB, Lambda GCP: Cloud Functions, Firestore
Online Support	AWS: S3, Lex, Lambda, DynamoDB
Message Passing	GCP: Cloud functions, Pub/Sub, Firestore
Machine Learning	GCP: NLP API, Vertex AI
Web Application Building & Hosting	React and GitLab CI/CD AWS: S3
Other Essential Modules: 1. Report Generation Module 2. Visualization	AWS: CloudTrail, CloudWatch Google Data Studio

Serverless B&B Application Architecture

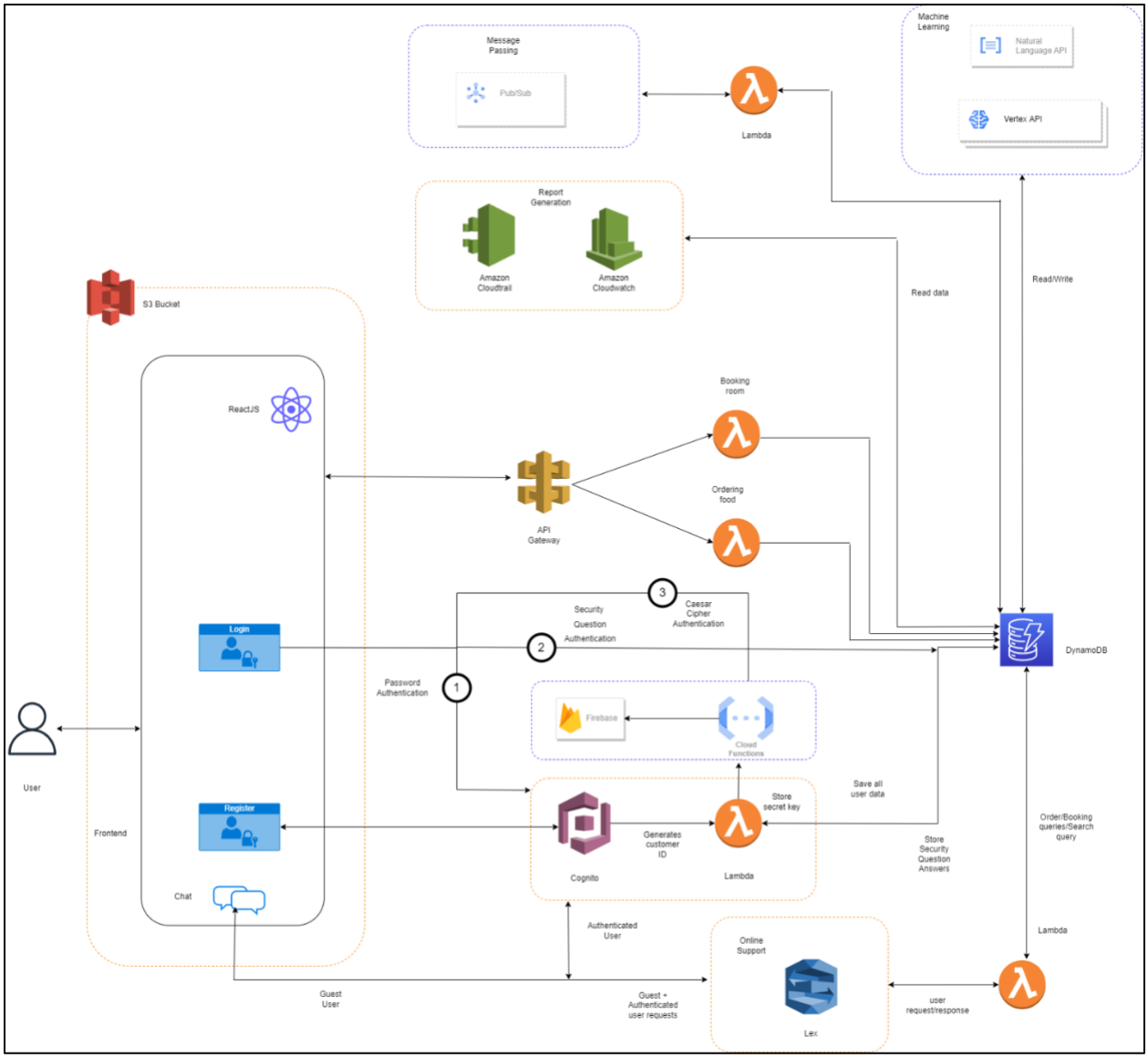


Figure 1: Architecture Diagram [2]

Modules and work distribution (Individual)

Group Member Name	**Contribution**
Aditya Deepak Mahale	16.67%
Jayasree Kulothungan	16.67%
Rishika Bajaj	16.67%
Sai Chand Kolloju	16.67%

Sourav Malik	16.67%
Udit Gandhi	16.67%

Modules and work distribution (Group)

Module Name	**Group Member Name**
User Management	Jayasree Kulothungan
Authentication	Sourav Malik
Online Support	Aditya Mahale
Message Passing	Udit Gandhi, Sai Chand Kolloju
Machine Learning	Rishika Bajaj
Web Application Building Hosting	Udit Gandhi, Sai Chand Kolloju, Aditya Mahale, Sourav Malik, Jayasree Kulothungan, Rishika bajaj Sai Chand Kolloju
Other Essential Modules: 1. Report Generation Module 2. Visualization	Jayasree Kulothungan Sai Chand Kolloju

Note: Every individual in our team worked responsibly and worked on their respective modules end to end, i.e., from creating the Front-End in React, to building it back in Node.JS and deploying its own cloud services. Our team worked in an agile environment, supporting each other in a time of difficulty.

Modules and their details

User Registration Service

High-level Implementation

User Registration service is implemented using the following AWS and GCP cloud services:

- **Amazon Cognito [7]:** This service is used to store user credentials like email, first name , last name and password. It is also used to verify if email exists.

- **Amazon Lambda [9]:** The lambda function is used for 3 purposes – to generate customer Id and to store the credentials in DynamoDB.

- **Amazon DynamoDB [6]:** This database service is used to store user credentials like first name, Last name, email, security questions and answers.

- **Amazon API Gateway [16]:** This service is used to call the lambda functions that create the customer ID and store data to DynamoDB directly from React.js using REST API.

- **GCP Firestore [5]**: This service is used is used to store the email of the user and the secret key provided by them for the purpose of Caesar Cipher.

Flowchart

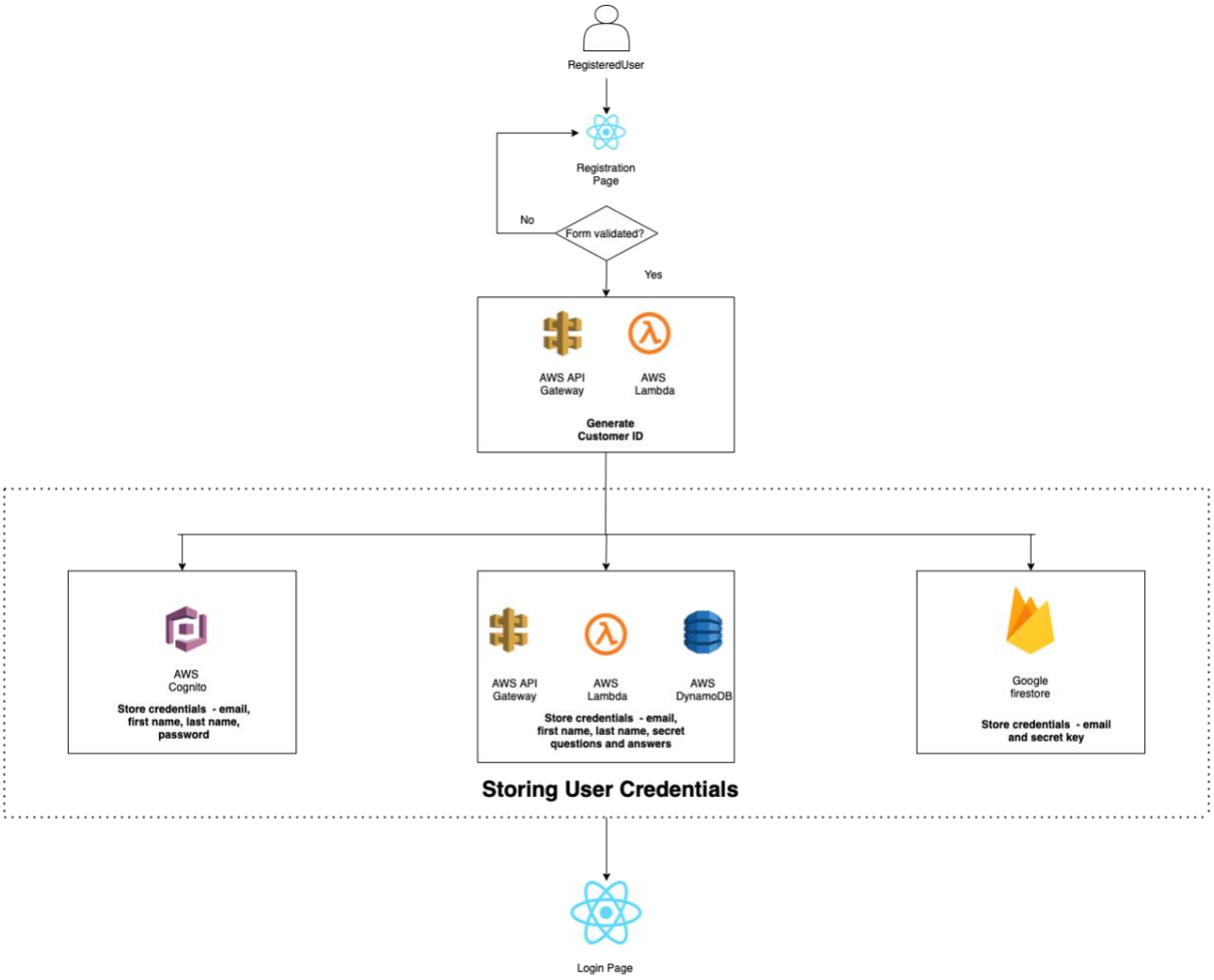


Figure 2: User Registration Flow Diagram

Pseudo Code

1. **Validating a new user**
    a. Registration page with all the fields is displayed to the user
    b. User enters the details and submits it
    c. Validate first name
        i. if first name is not present
            i. Display “First name is required”
        ii. If first name has numbers and special characters
            i. Display “First name can only have alphabets”
    d. Validate last name
        i. if last name is not present
            i. Display “Last name is required”
        ii. If first name has numbers and special characters
            i. Display “Last name can only have alphabets”
    e. Validate email
        i. If email is not present
            i. Display “Email is required”
        ii. If email is not of correct format
            i. Display “Email is invalid”
    f. Validate password
        i. If password is not present
            i. Display “Password is required”
        ii. If password is not of correct format
            i. Display “Password is invalid”
    g. Validate Secret answers
        i. If answer is not present
            i. Display “Answer required”
        ii. If answer has special characters
            i. Display “Answers can only be alphanumeric”
    h. Validate Secret Key
        i. If key is not present
            i. Display “key is required”
        ii. If key is greater than 3 numbers
            i. Display “Key should be less than 3 numbers”
        iii. If key has alphabets or special characters
            i. Display “Key should be numeric”

2. **Generating customer ID for a user**
    a. The form is validated
    b. An API Gateway [16] is created to trigger a lambda function
    c. The first name of the user and the room type booked is sent to the post method of API Gateway
    d. The information is sent to lambda [9]
    e. The customer ID is generated as a string combining first name, room type and timestamp making it unique.

f. The ID is returned to API gateway and is got as response.

3. Registering a user

a. The first name, last name, email and password is sent to AWS Cognito [7] using the pool ID and client ID
b. On successfully creating user in AWS Cognito [7], an post method of API gateway is triggered by passing customer ID, first name, last name, email, security questions and answers as parameters
c. The API gateway [16] passes the parameters to the lambda function
d. The lambda function updates the information to the table UserCredentials in AWS DynamoDB [6]
e. On successfully storing the user details on AWS DynamoDB, the email and secret key is stored In the Firestore database of Google Firebase.
f. The webpage navigates to login page for authentication.

Front-end Code Implementation

The code to validate the registration form is shown below

registration.jsx

frontend > src > components > registration > registration.jsx > SignUp

```
const validate = (data) => {
    const errors = {};
    if (data.get("firstName") === "") {
        errors.firstname = "First name Required!";
    } else if (!/^[A-Za-z]+$/.test(data.get("firstName"))) {
        errors.firstname = "First name can only have characters!";
    }
    if (data.get("lastName") === "") {
        errors.lastname = "Last name Required!";
    } else if (!/^[A-Za-z]+$/.test(data.get("lastName"))) {
        errors.lastname = "First name can only have characters!";
    }
    if (data.get("email") === "") {
        errors.email = "Email Required! ";
    }
    if (data.get("password") === "") {
        errors.password = "password is required";
    } else if (data.get("password").length < 8) {
        errors.password = "password length is less then 8";
    } else if (!/^[ A-Za-z0-9_@./#&+-]*$/.test(data.get("password"))) {
        errors.password = "Only special characters and alphanumeric is allowed!";
    }
    if (data.get("answer1") === "") {
        errors.answer1 = "Security answer is required";
    } else if (!/^[A-Za-z0-9]+$/.test(data.get("answer1"))) {
        errors.answer1 = "Only Alphanumeric is allowed";
    }
    if (data.get("answer2") === "") {
        errors.answer2 = "Security answer is required";
    } else if (!/^[A-Za-z0-9]+$/.test(data.get("answer2"))) {
        errors.answer2 = "Only Alphanumeric is allowed";
    }
    if (data.get("answer3") === "") {
        errors.answer3 = "Security answer is required";
    } else if (!/^[A-Za-z0-9]+$/.test(data.get("answer3"))) {
        errors.answer3 = "Only Alphanumeric is allowed";
    }
    if (data.get("secretKey") === "") {
        errors.secretKey = "Secret key is required";
    } else if (data.get("secretKey").length > 3) {
        errors.secretKey = "Secret key should be 3 digits";
    } else if (!/^[0-9]+$/.test(data.get("secretKey"))) {
        errors.secretKey = "Only Numeric is allowed";
    }
    setError(errors);
};
```

Figure 3: Form validation

On clicking on submit the values are set to state variables as shown below

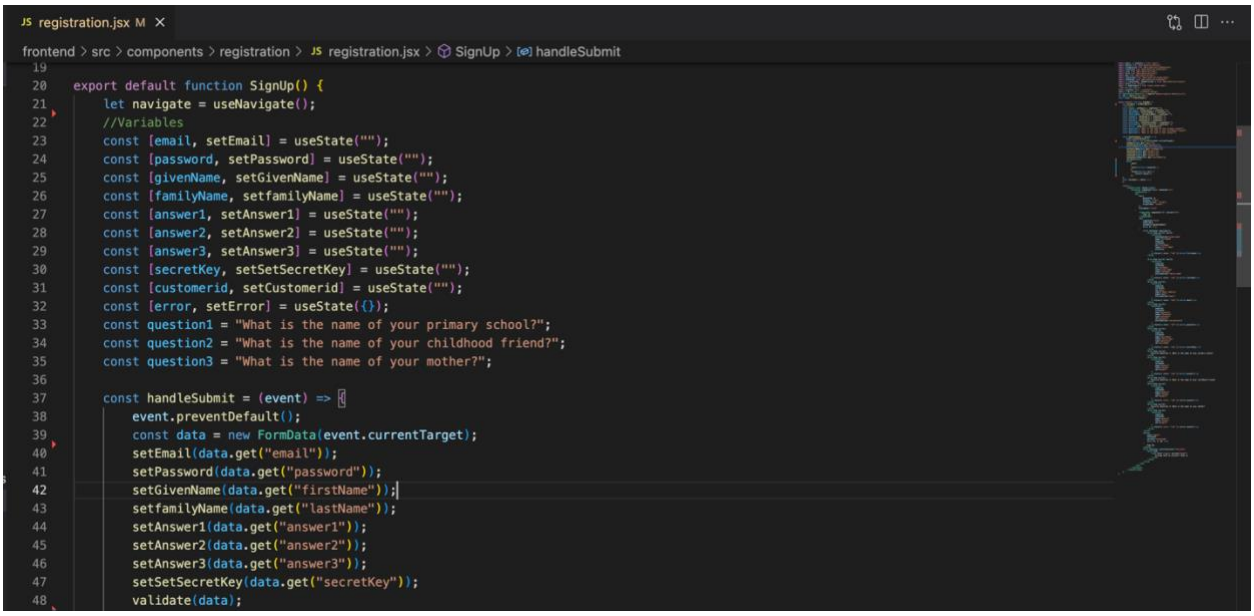


Figure 4: Setting variables

The code to call the API gateway to create customer id is shown below

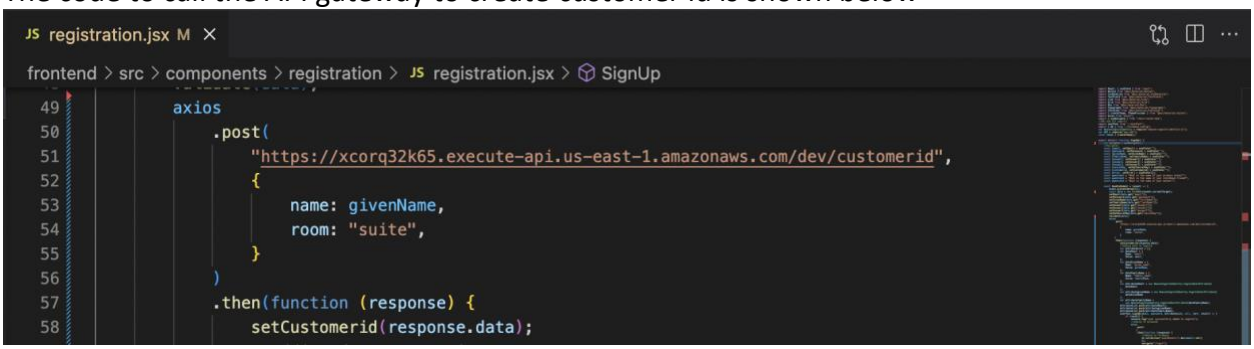


Figure 5: API gateway call to lambda function to create customer ID

The code to send data such as first name, last name, email and password to AWS Cognito is shown below

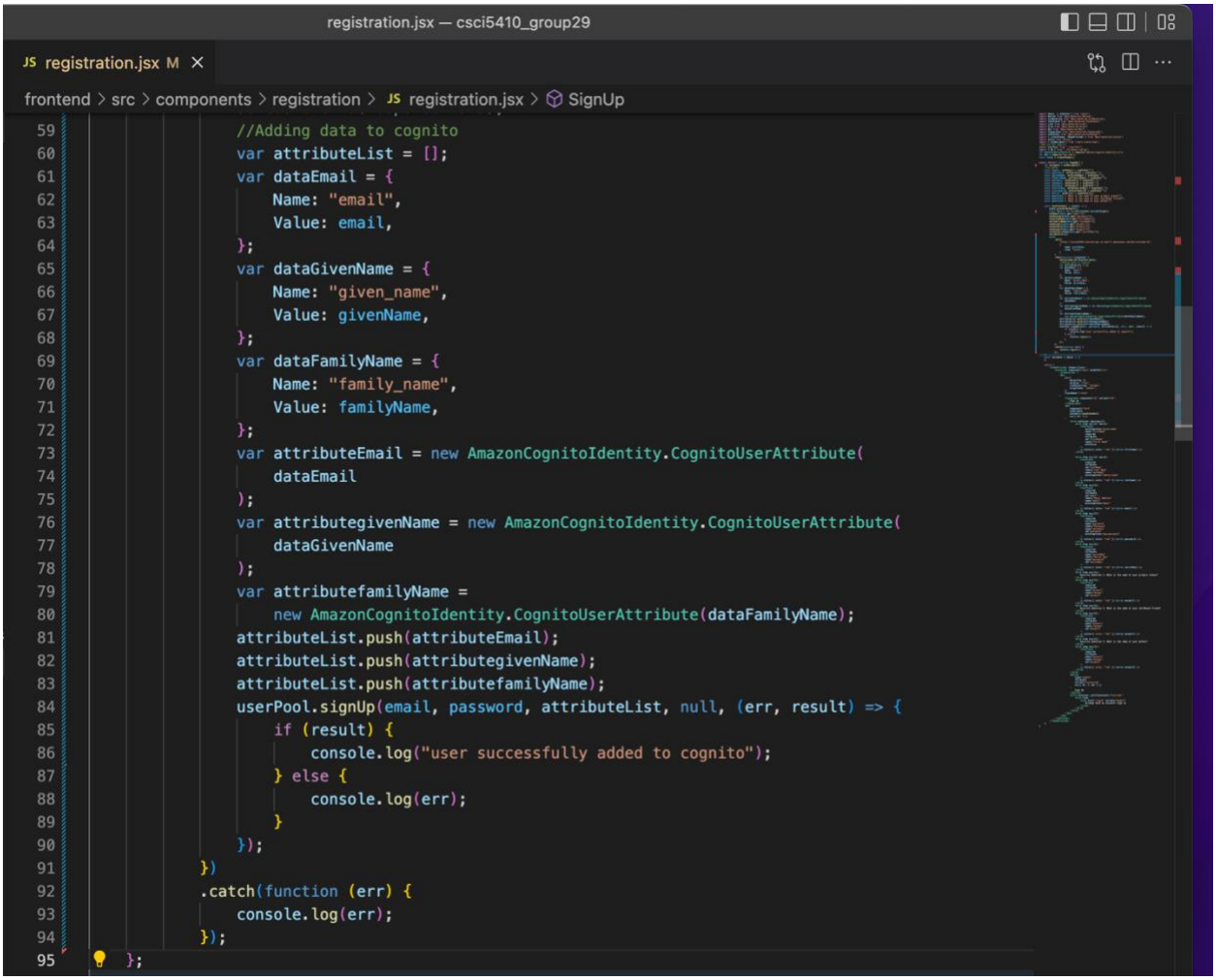


Figure 6: Sending user data to AWS Cognito

The code to send user data such as first name, last name, email, secret questions, and answers in DynamoDB through API Gateway is shown below

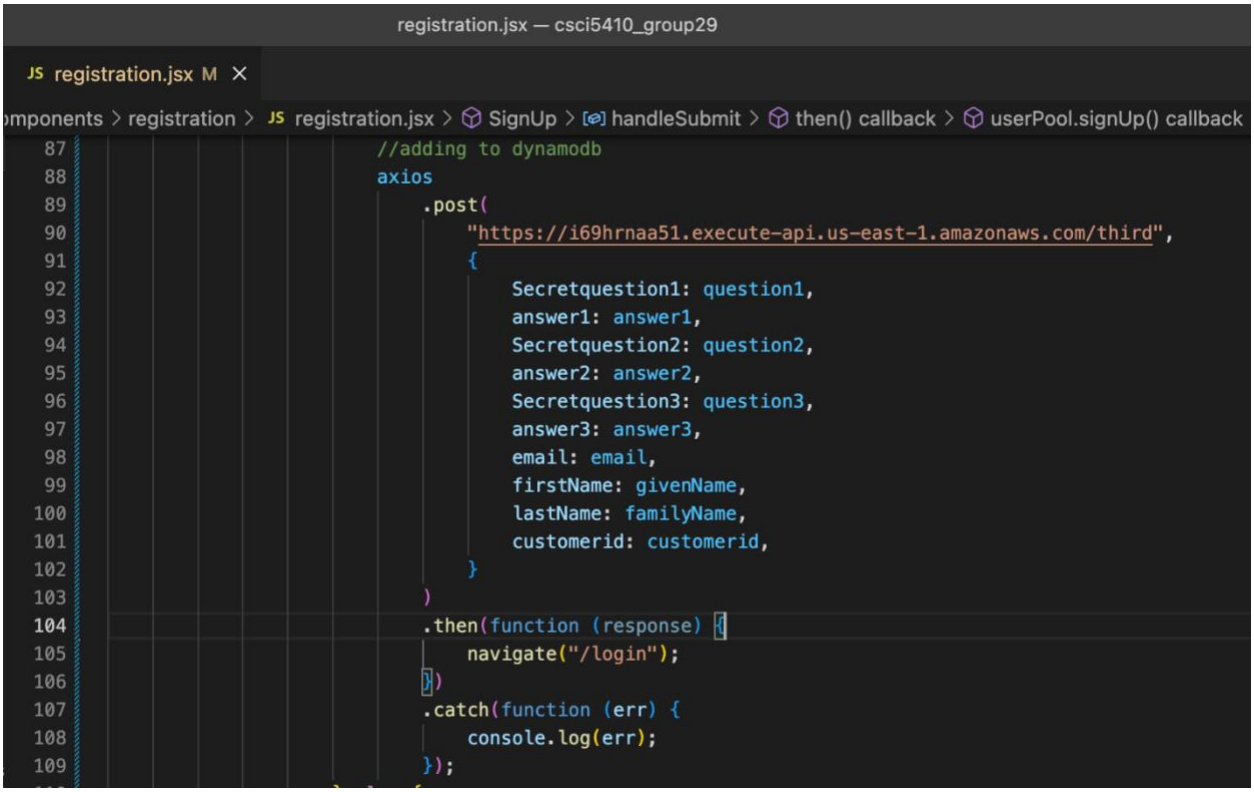


Figure 7: Sending user data to DynamoDB via API Gateway

The code to send user data such as email and secret key to firestore database of Google Firebase is shown below

Figure 8: Registering data inside database

Back-end Code Implementation

The backend code implementation includes all the lambda functions and API gateway.

The API gateway setup for triggering the lambda function to create customer ID is shown below

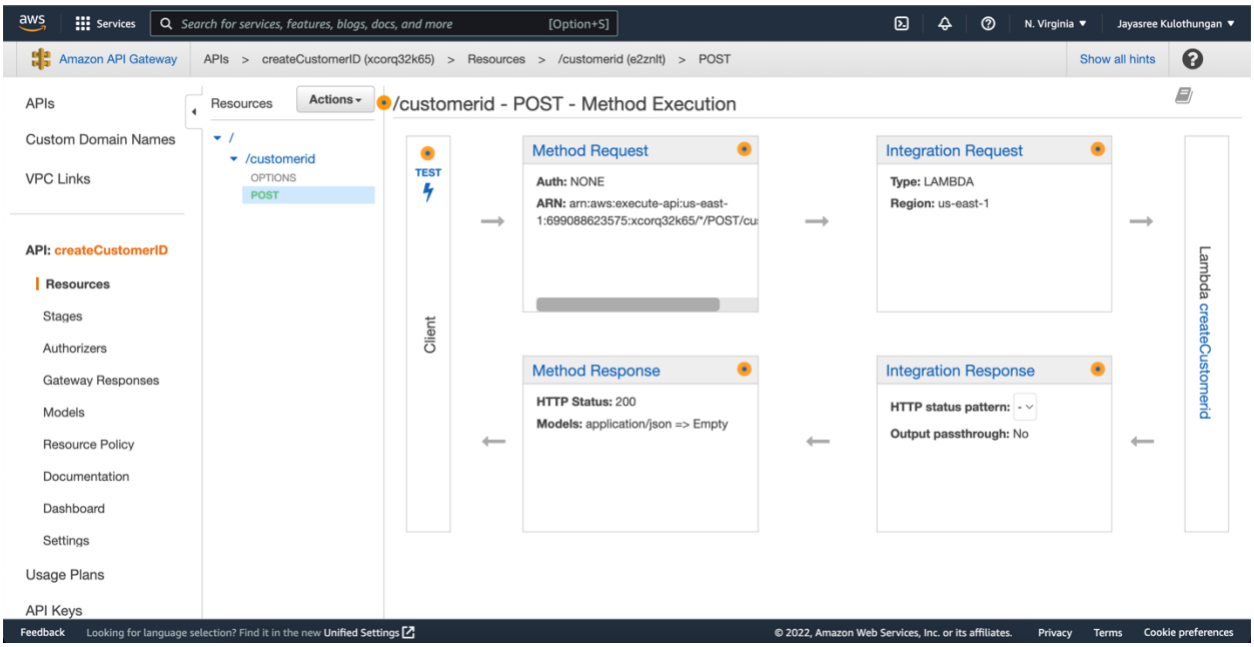


Figure 8: API gateway to trigger createCustomerID lambda function

The createCustomerID lambda function that creates the customer Id is shown below

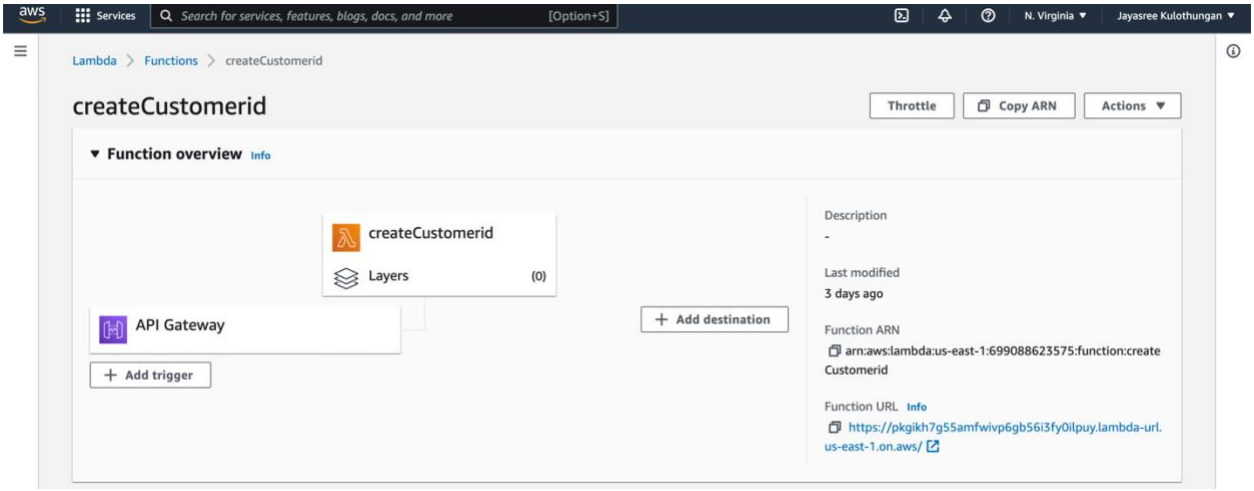


Figure 9: createCustomerId lambda function with API gateway trigger

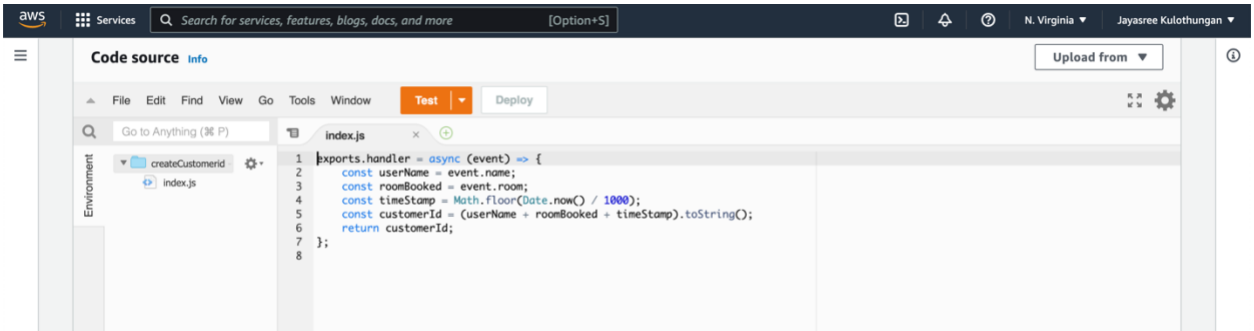


Figure 10: createCustomerID lambda function

The API gateway to call the lambda function that updates the dynamo DB is shown below

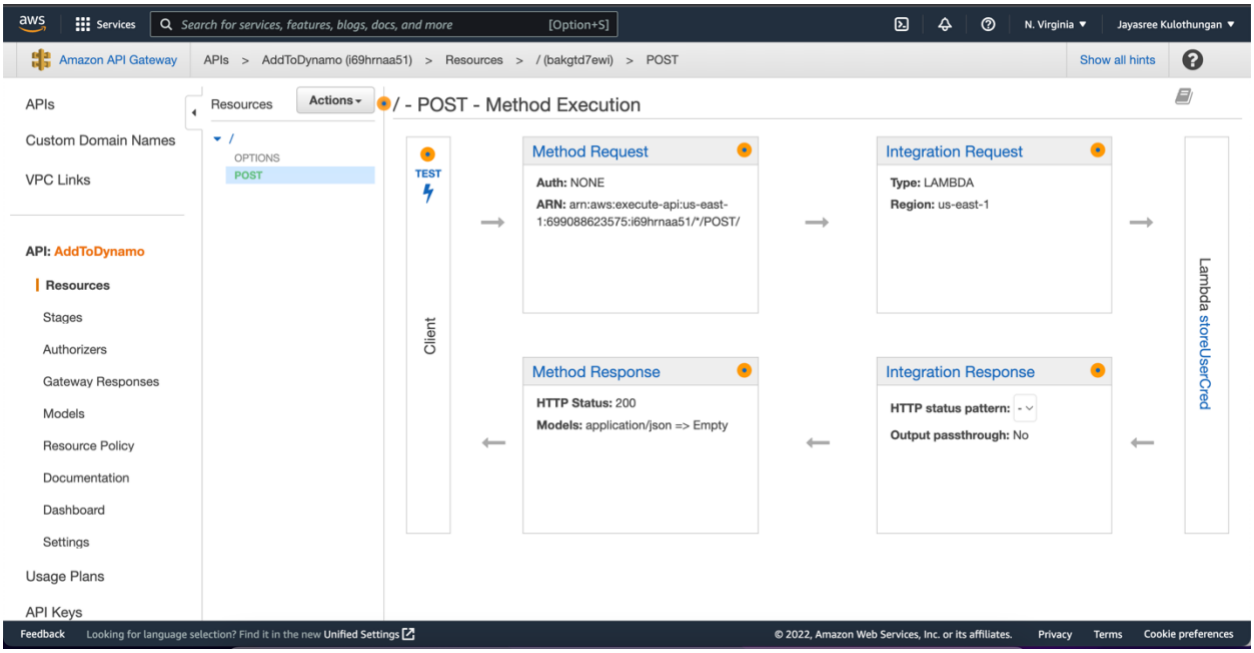


Figure 11: API gateway to trigger storeUserCred lambda function

The lambda function to store user credentials on dynamo DB is given below

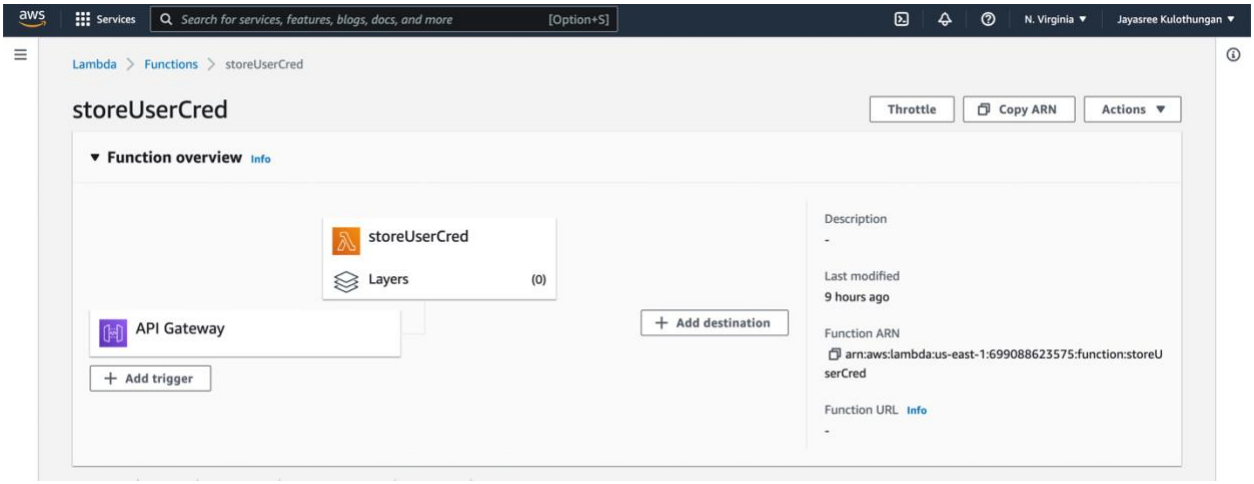


Figure 12: storeUserCred lambda function with API Gateway trigger

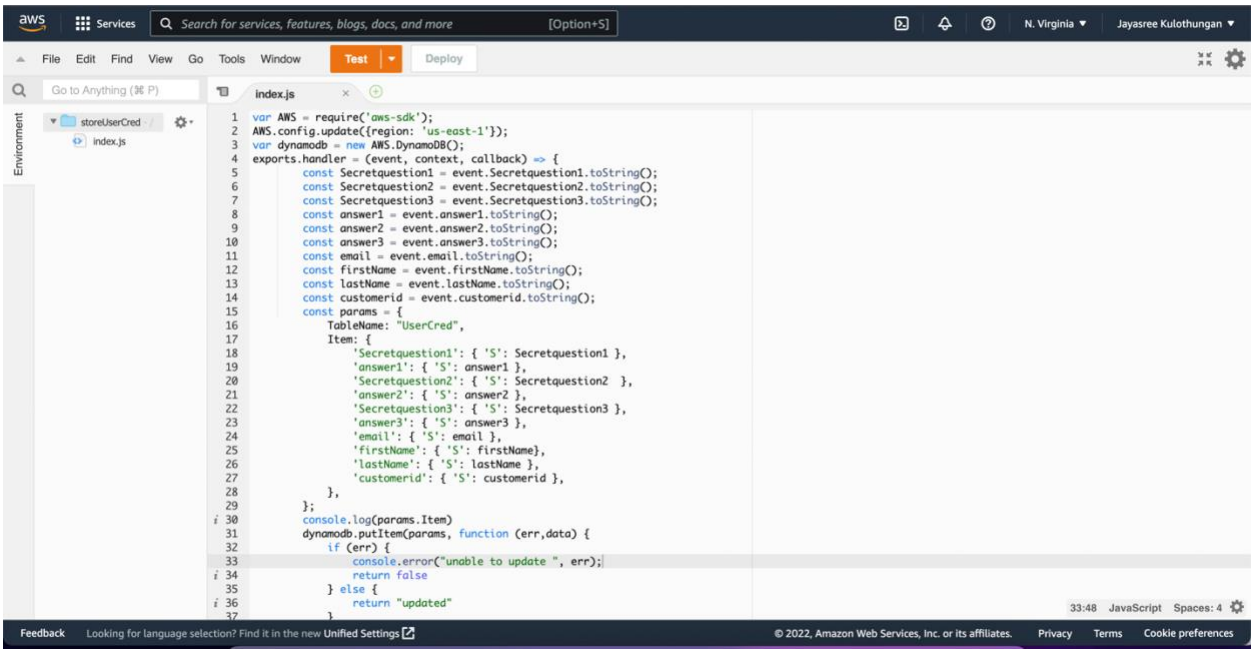


Figure 13: storeUserCred lambda function

Testing

Testing for validations if fields are empty

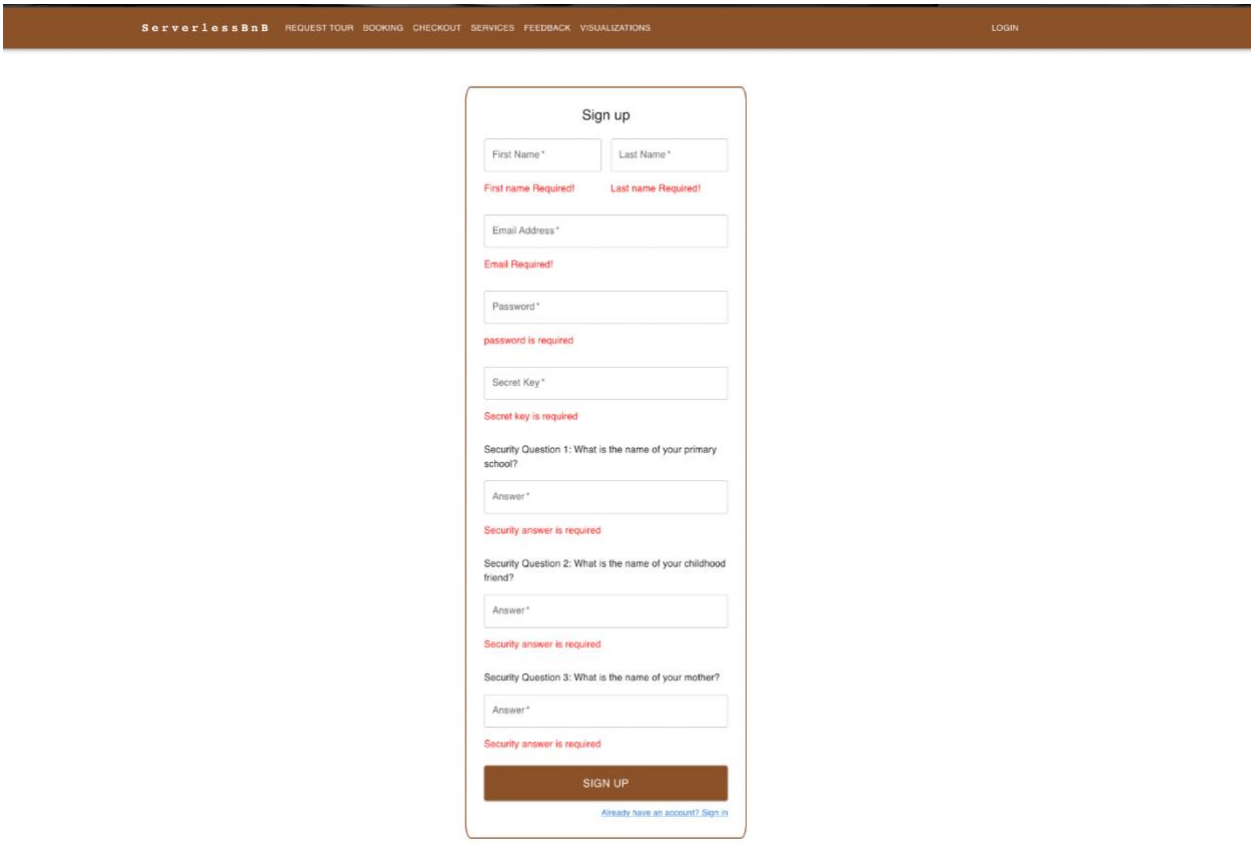


Figure 14: Validation for missing fields

Testing for validation when input format is incorrect

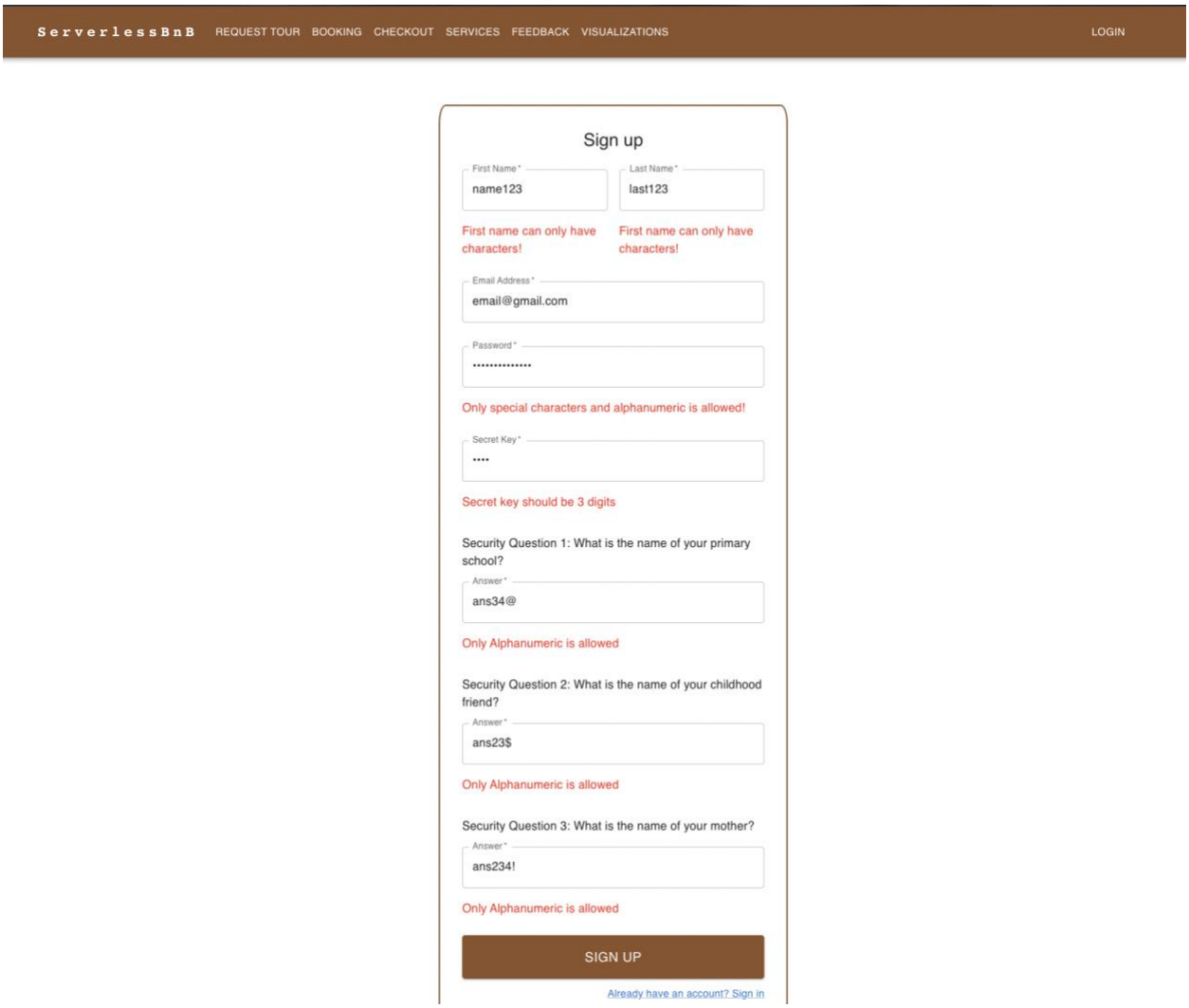


Figure 15: Validation for incorrect input format

Testing by registering a user

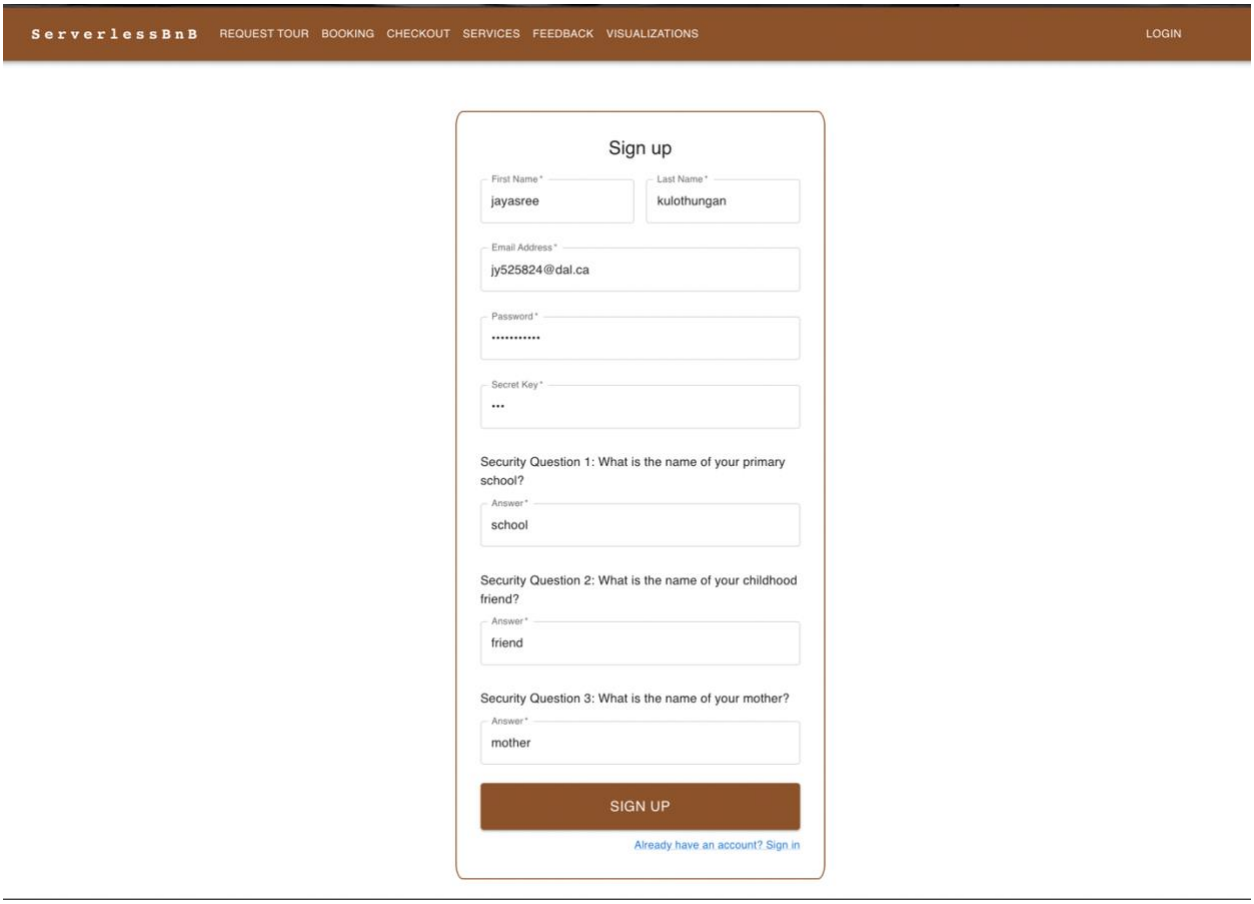


Figure 16: Registering a user

a) Creating a user in AWS Cognito

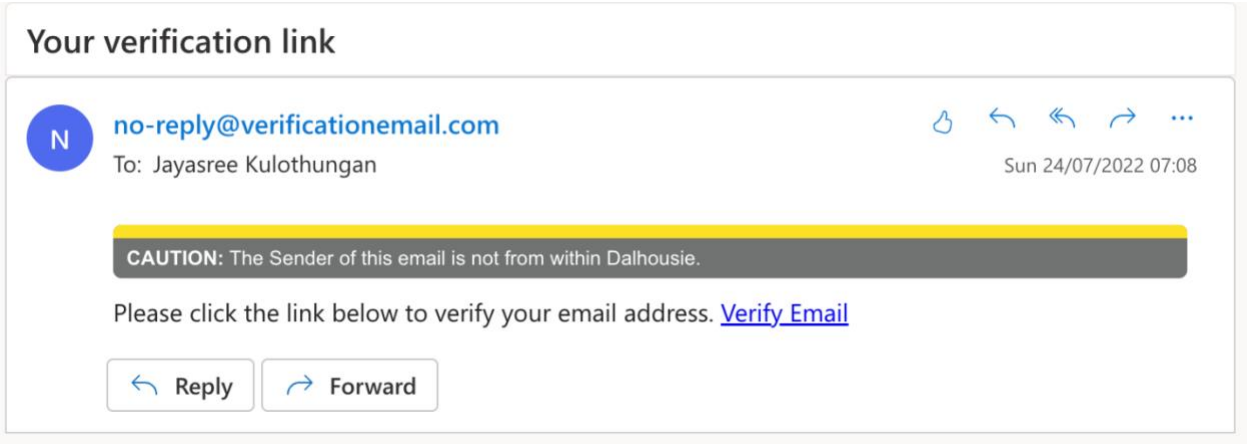


Figure 17: Verification link sent by AWS Cognito

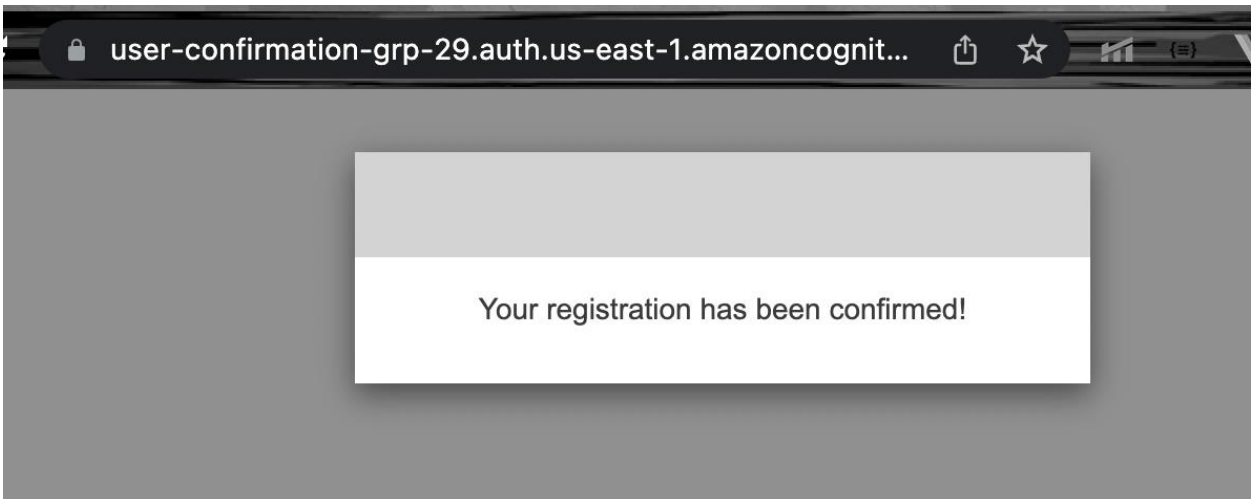


Figure 18: User confirmed

The user is created in the user pool

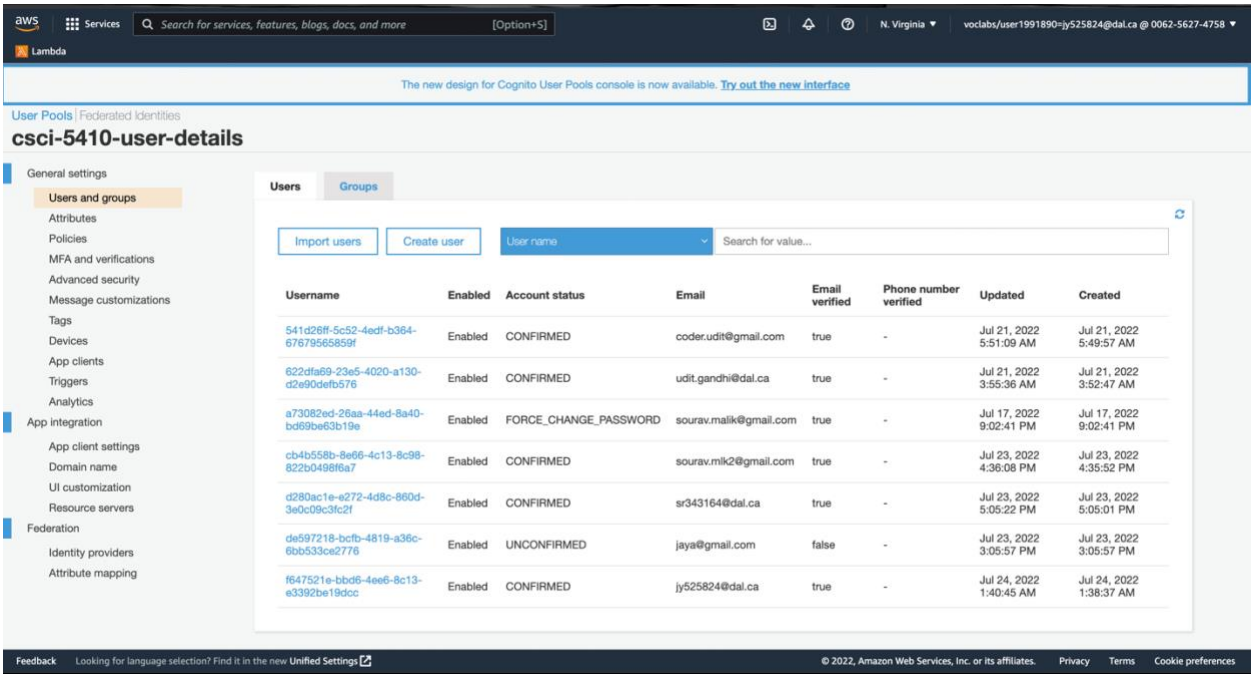


Figure 19: user created in AWS Cognito user pool

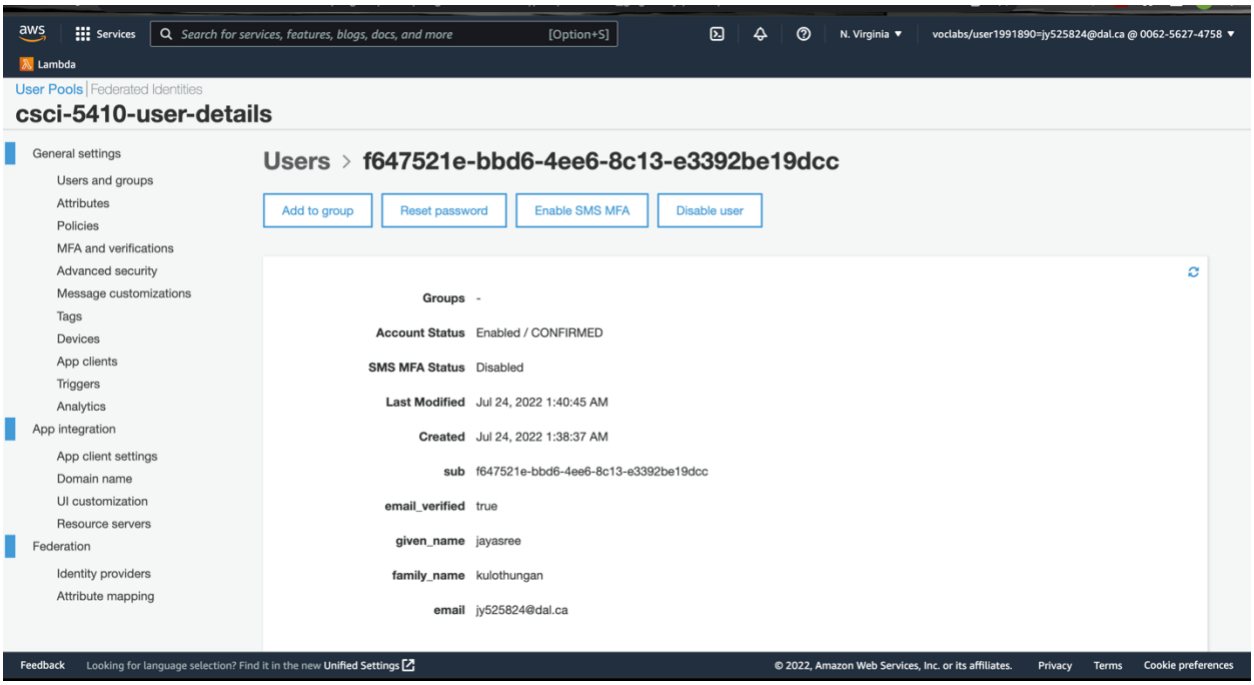


Figure 20: User created in AWS Cognito

b) Storing user credentials in DynamoDB

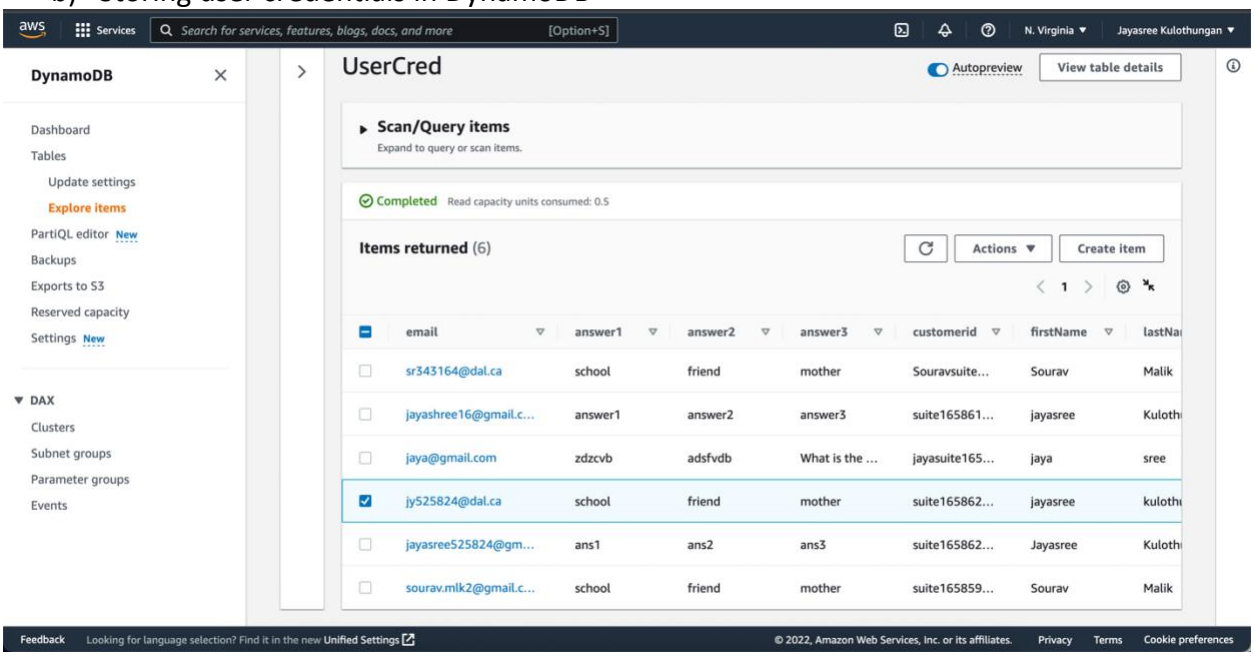


Figure 21: Stored user credentials in DynamoDB

c) Storing secret key in Firestore

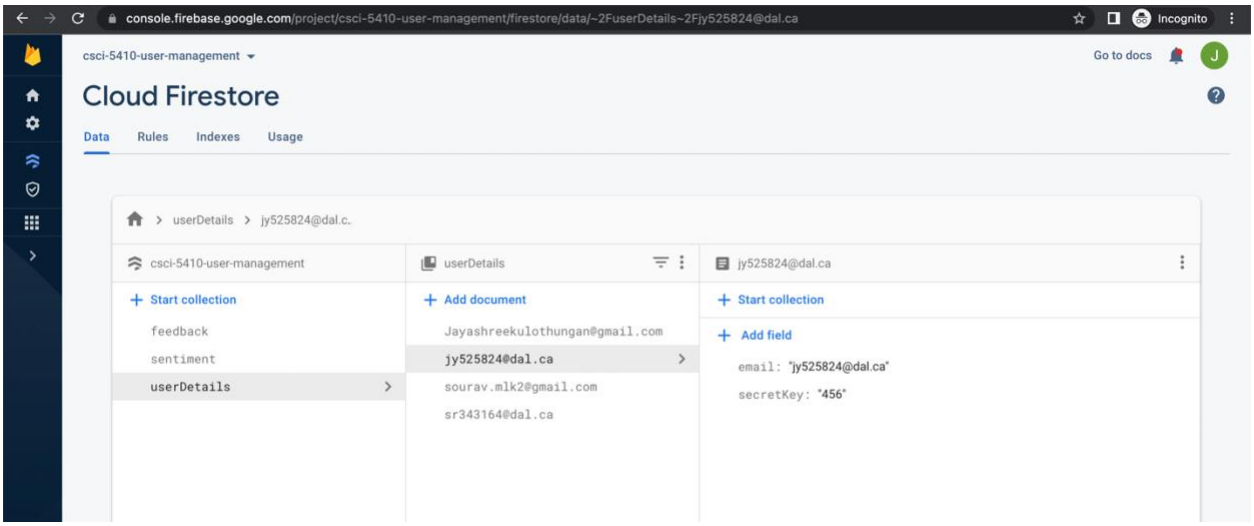


Figure 22: Stored email and secret key in Firestore

Authentication Service

High-level Implementation

User Authentication service is implemented using the following AWS and GCP cloud services:

- **Amazon Cognito [7]:** This service is used to validate user ID and password stored in the user pool of Cognito.

- **AWS Lambda [9]**: Lambda function is triggered to validate the security question answers provided by the user in the step 2 of authentication.

- **Amazon DynamoDB [6]**: User details are stored in the DynamoDB after user has successfully registered to the application. Therefore, lambda triggered in the step 2 gets user specific data including security questions and answers from DynamoDB to validate it with user answers.

- **GCP Cloud Function [10]**: Cloud Function is used for the 3rd step of authentication i.e., Caesar Cipher. Cloud function is triggered to validate the user encrypted Caesar cipher string with the system encryption of Caesar cipher.

- **GCP FireStore [5]**: The NoSQL database is used for fetching Caesar cipher secret key to perform the encryption at the system level and then validate it with user encrypted cipher for last step of authentication.

Flow Chart

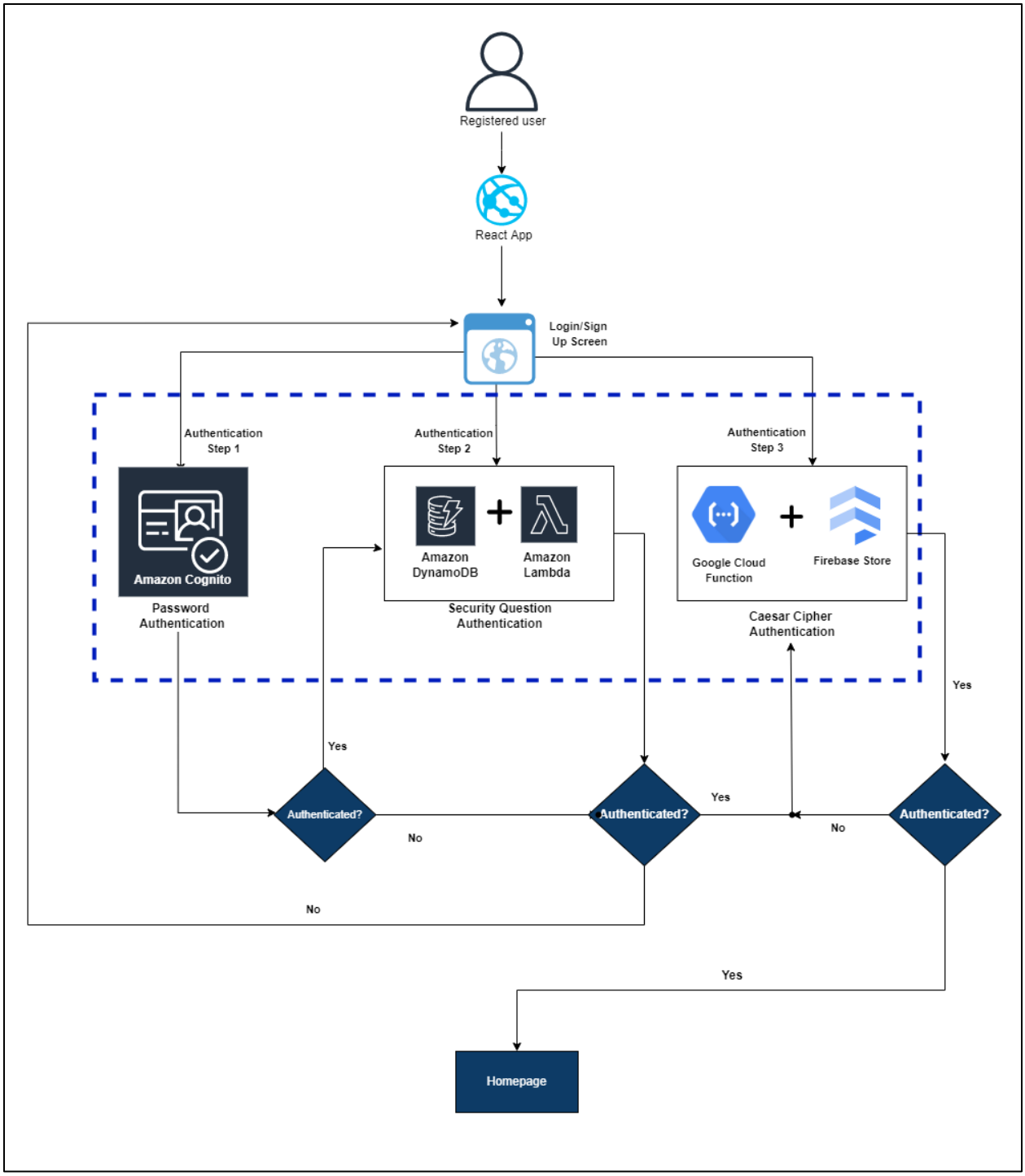


Figure 23: User Authentication Flow Diagram

Pseudo Code

Any user registered with the application goes through a 3-factor authentication to log-in to the application. The steps involve:

1. **ID-Password Validation**
    a. Login screen with username and password fields are displayed to the user.
    b. User enters the credentials and submits it.
    c. User credentials are sent to AWS Cognito [7] using the User Pool ID and the Client ID.
    d. User credentials are validated with the credentials stored in the User Pool.
    e. Upon successful validation of credentials, user is navigated to the security question answer screen for next step of authentication.

2. **Security Question Answer**
    a. Security Question validation screen is displayed to the user.
    b. Lambda function is triggered to get the list of security questions.
    c. Lambda function connects with DynamoDB to get the user data from userDetails table.
    d. One random security question is selected from the list of 3 security questions.
    e. Selected security question is displayed to the user on Security Question validation screen.
    f. User inputs their answer to the provided security question.
    g. User's answer to the question is taken and validated with the answer stored in the DynamoDB using the triggered lambda function.
    h. If the answer matches, user is navigated to Caesar cipher validation screen for the last step of authentication.

3. **Caesar Cipher**
    a. Caesar cipher validation screen is displayed to the user for 3rd-factor authentication.
    b. A random 4-character string is displayed to the user for encryption.
    c. User encrypts the provided string with the secret key that they shared with the system at the time of registration.
    d. Google Cloud Function is triggered with the random generated 4-character string and user encrypted string as the parameters.
    e. Cloud function connects with FireStore and fetches the secret key shared by the user at the time of registration.
    f. System performs the encryption on its end on the 4-character string using the fetched secret key from the FireStore.
    g. System's encrypted cipher is matched with the user's encrypted cipher.
    h. If the encrypted ciphers match, user is authenticated and is allowed to log in to the application.

Front-end Code Implementation

- Below code is for validating user credentials from Cognito user pool where username and password and sent and validated.

```
const onLoginFormSubmit = (event) => {
    event.preventDefault();
    if (loginForm.email == '' || loginForm.password == '') {
        toast.error("Please enter credentials!!");
        return;
    }

    const userAuthDetails = new AuthenticationDetails({
        Username: loginForm.email,
        Password: loginForm.password
    });

    const loginUser = new CognitoUser({
        Username: loginForm.email,
        Pool: UserPool
    });


    loginUser.authenticateUser(userAuthDetails, {
        onSuccess: data => {
            console.log("Successfully", data);
            localStorage.setItem("username", loginForm.email);
            navigate("../question-validate");
        },

        onFailure: err => {
            console.error("onFailure:", err);
            toast.error("Invalid Username or Password. Please try again!!");
        }
    });
}
```

Figure 24: Code for validating credentials using Amazon Cognito

- Below code is for displaying security question to the user using lambda function URL

```
function QuestionValidation() {


    const [securityQuestion, setSecurityQuestion] = useState("");
    const [sQuestionAnswerForm, setValues] = useState({
        answer: ''
    });
    let navigate = useNavigate();
    let preventCall = false;
    useEffect(() => {
        if (!preventCall) {
            preventCall = true;
            axios.get('https://j34bg3jr2hbeiktrfabgz5ypgu0ustay.lambda-url.us-east-1.on.aws/authenticateQuestion/question?username=' + localStorage.getItem("username"))
                .then(res => {
                    console.log(res);
                    setSecurityQuestion(res.data.question);
                }).catch(err => {
                    console.log(err);
                })
        }

    }, []);


    const handleQuestionnaireForm = (event) => {
        event.preventDefault();
        if (sQuestionAnswerForm['answer'] == '') {
            toast.error('Please enter something.');
            return;
        }

        let obj = {
            [securityQuestion]: sQuestionAnswerForm.answer,
            "username": localStorage.getItem('username')
        }
        const requestBody = {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: obj
        };
```

Figure 25: Display Security Question

- Below code is for calling the lambda function URL for validating user response to security question.

```
        axios.post("https://j34bg3jr2hbeiktrfabgz5ypgu0ustay.lambda-url.us-east-1.on.aws/authenticateQuestion/validateAns", obj)
            .then(res => {
                console.log(res.data);
                navigate("../caesarcipher-validate");

            })
            .catch(error => {
                console.log(error + " questionnaire response");
                toast.error('Incorrect Answer. Please try again!!');
            });
    }


    const handleFormValueChange = (event) => {
        const { name, value } = event.target;
        setValues({ ...sQuestionAnswerForm, [name]: value });
    };


    return (
        <>
            <form onSubmit={handleQuestionnaireForm}>
                <div className="f-body">
                    <div >
                        <label>Security Question:</label>
                    </div>
                    <div>
                        <label>{securityQuestion}</label>
                        <input type="text" value={sQuestionAnswerForm['answer']} name="answer" onChange={handleFormValueChange}></input>
                    </div>
                    <button type="submit">Submit</button>
                </div>
            </form>
        </>
```

Figure 26: Call Lambda for validating user answer

- Below code is for generating random 4-character string for Caesar cipher encryption.

```
function CaesarCipherValidation() {

    const [createdCipherString, setCreatedCipherString] = useState();

    const [encryptedStrForm, setValues] = useState({
        encryptedStr: ''
    });

    let navigate = useNavigate();

    useEffect(() => {
        setCreatedCipherString(createCipherString())
    }, []);

    const createCipherString = () => {
        var cipherString = '';
        var alphabets = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';
        for (var i = 0; i < 4; i++) {
            cipherString = cipherString + alphabets.charAt(Math.floor(Math.random() * alphabets.length));
        }
        return cipherString;
    }
```

Figure 27: Generate Caesar Cipher String

- Below code is for calling Google Cloud Function for Caesar cipher validation.

```
const oSubmitCaesarCipherForm = (event) => {
    event.preventDefault();
    if (encryptedStrForm.encryptedStr == '') {
        toast.error('Encrypted String can not be empty!!');
        return;
    }

    let obj = {
        'username': localStorage.getItem('username'),
        'caesarString': createdCipherString,
        'userInput': encryptedStrForm.encryptedStr
    };
    axios.post('https://us-central1-csci-5410-user-management.cloudfunctions.net/validateCaesar', obj)
        .then(response => {
            if (response.status === 401) {
                toast.error('Incorrect cipher. Please try again!!');
            }
            else if (response.status === 200) {
                localStorage.setItem("group29_logged_in", true);
                toast.info("Login successful")
                navigate("../home");
            }
            else {
                toast.error('Something went wrong!!');
            }
        })
        .catch(error => {
            console.log(error);
            toast.error('Incorrect cipher. Please try again!!');
        });
}
```

Figure 28: Call Google Cloud Function for Caesar Cipher Validation

Back-end Code Implementation

- Below code is from the lambda function index.js file which defines the GET and POST paths for fetching security questions from DynamoDB and validating user response to provided security question.

Figure 29: Lambda function index.js file

- Below code is from the lambda function authenticateUsingQuestions.js file which connects with the DynamoDB, fetches the user details including security questions and answers to those questions along with randomly selecting a question from the received security questions list to display it to user on Security question validation screen.

Figure 30: Get Security Questions

- Below code is from the lambda function authenticateUsingQuestions.js file which validates the user's answer to the provided security question.

Figure 31: Validate user's response to security question

- Below code is from Google Cloud Function in which system performs Caesar cipher encryption on its end to validate it with user encrypted cipher.

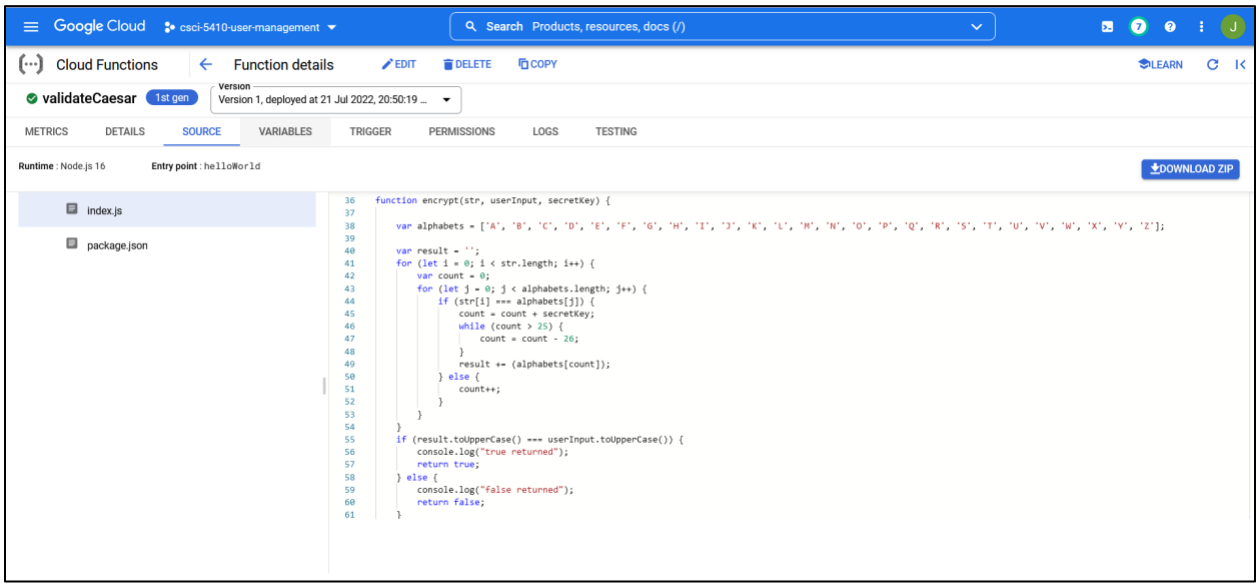


Figure 32: Google Cloud Function Caesar cipher encryption by system

- Below code is from the Google Cloud Function in which system matches the user's encrypted Caesar Cipher with the system encrypted Caesar Cipher string to authenticate the user in the final step of authentication.

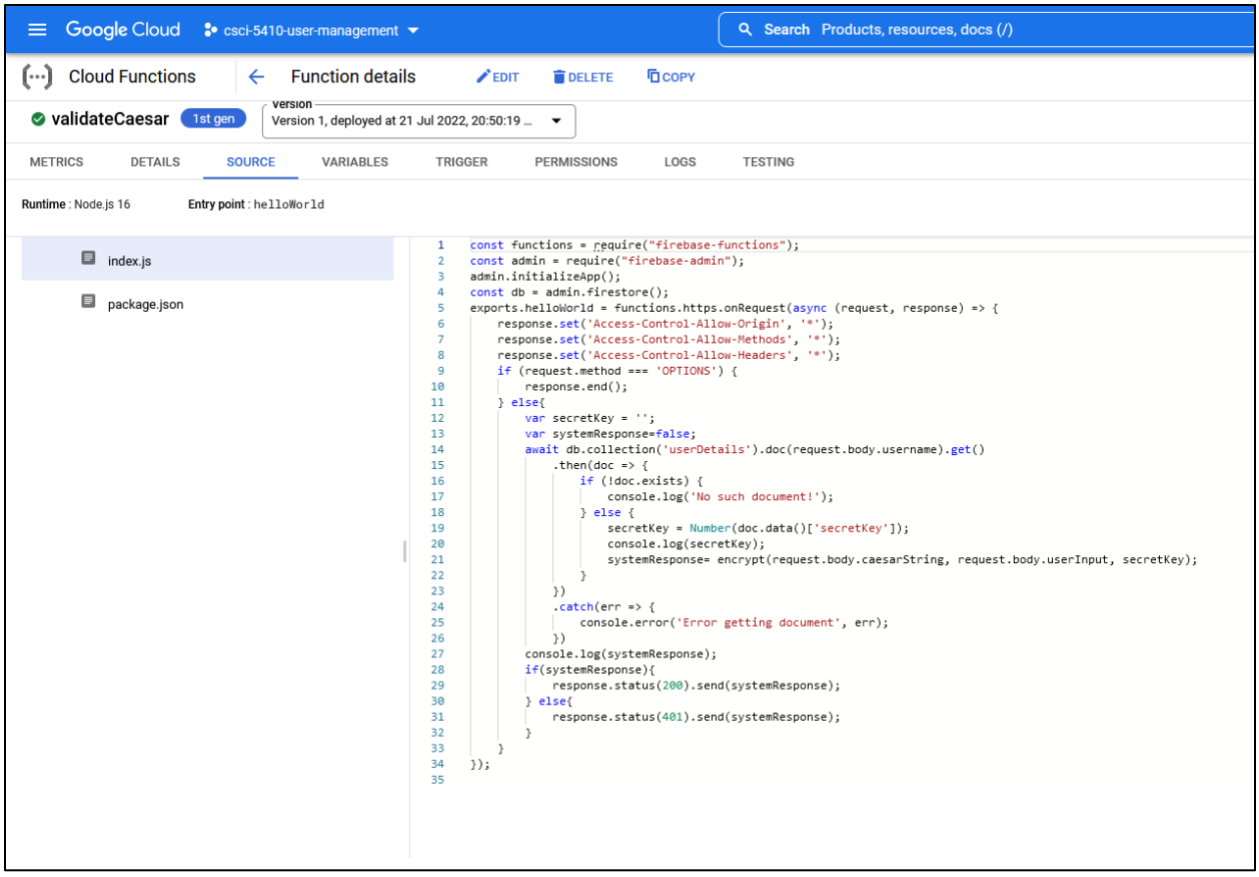


Figure 33: Google Cloud Function Caesar Cipher Validation

Testing

Scenario 1: User tries to log in without entering the credentials. Error toast message is displayed.

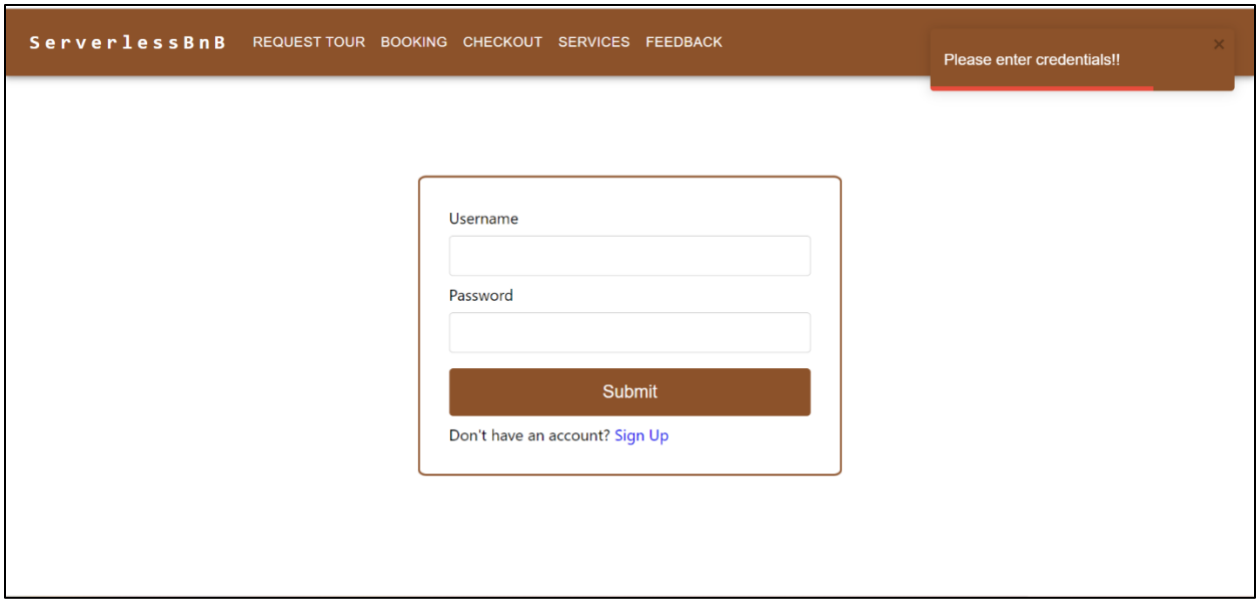


Figure 34: Login without credentials

Scenario 2: User tries to log in with invalid credentials. Error toast message is displayed.

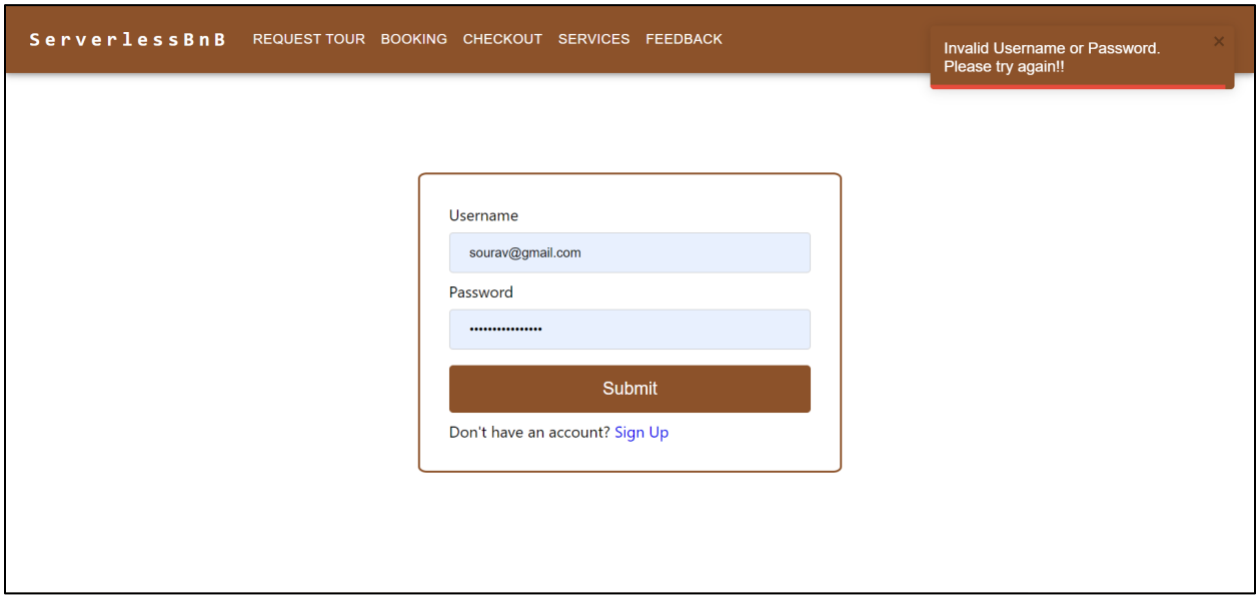


Figure 35: Login with Invalid Credentials

Scenario 3: User tries to log in with valid credentials. User is navigated to next step of Authentication i.e., Security question validation screen.

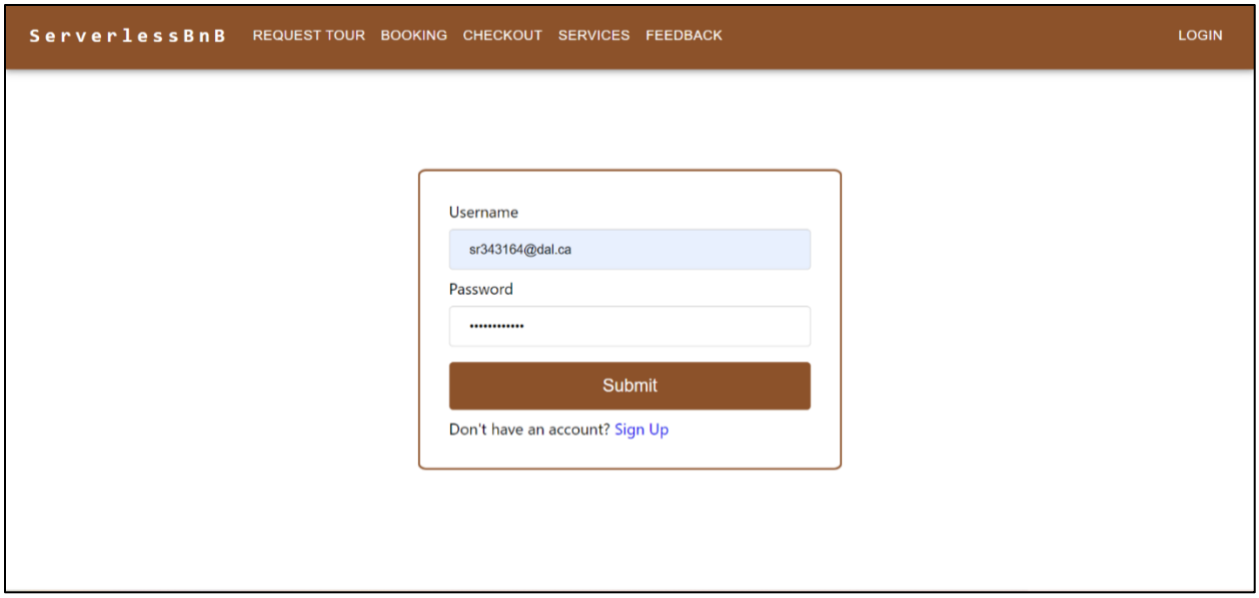


Figure 36: Login with valid credentials

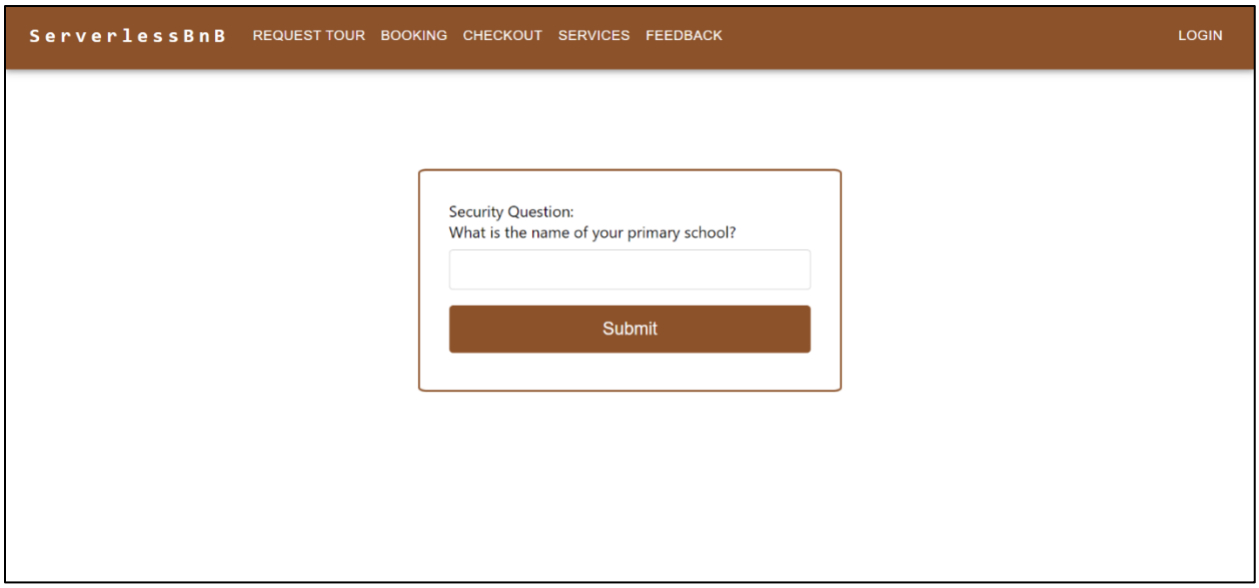


Figure 37: Screen displayed after successful validation from Cognito

Scenario 4: User tries to log in without providing answer to the security question. An error toast message is displayed.

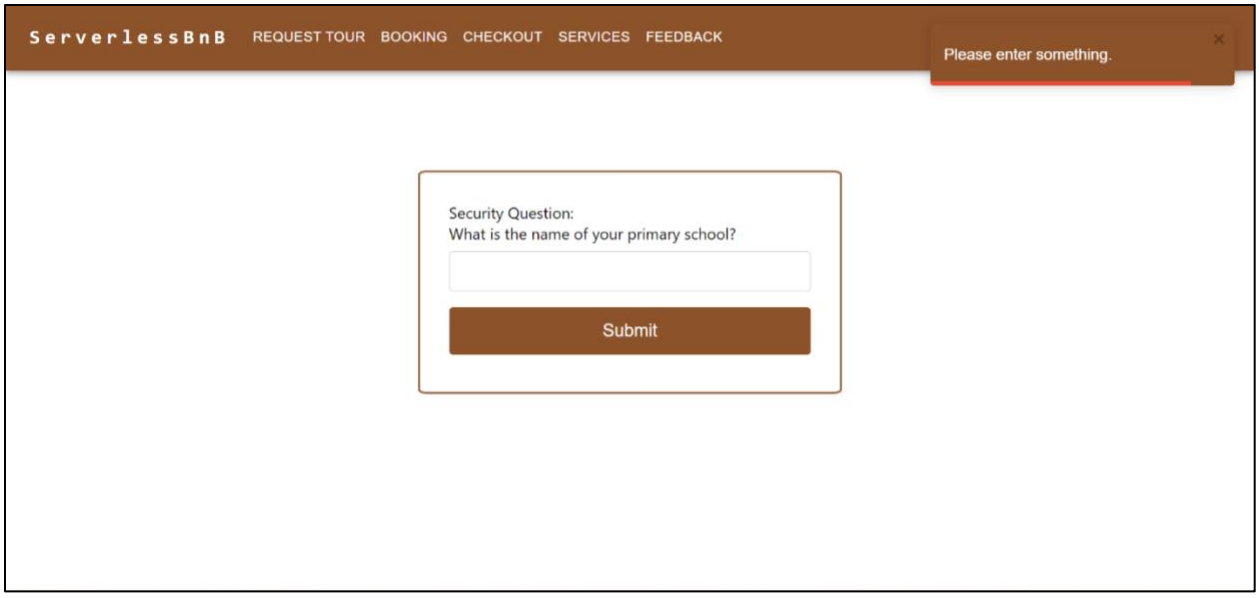


Figure 38: Log in without providing answer to security question

Scenario 5: User provides incorrect answer to the security question. Lambda function is triggered to validate the answer and an error toast message is displayed.

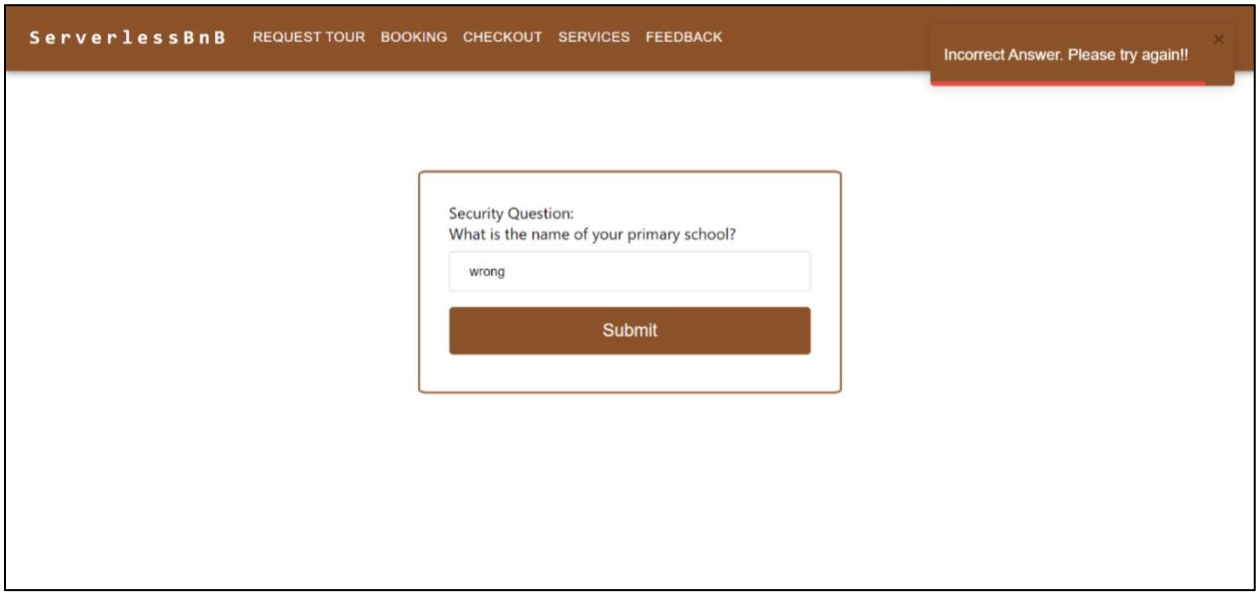


Figure 39: Incorrect answer error toast message

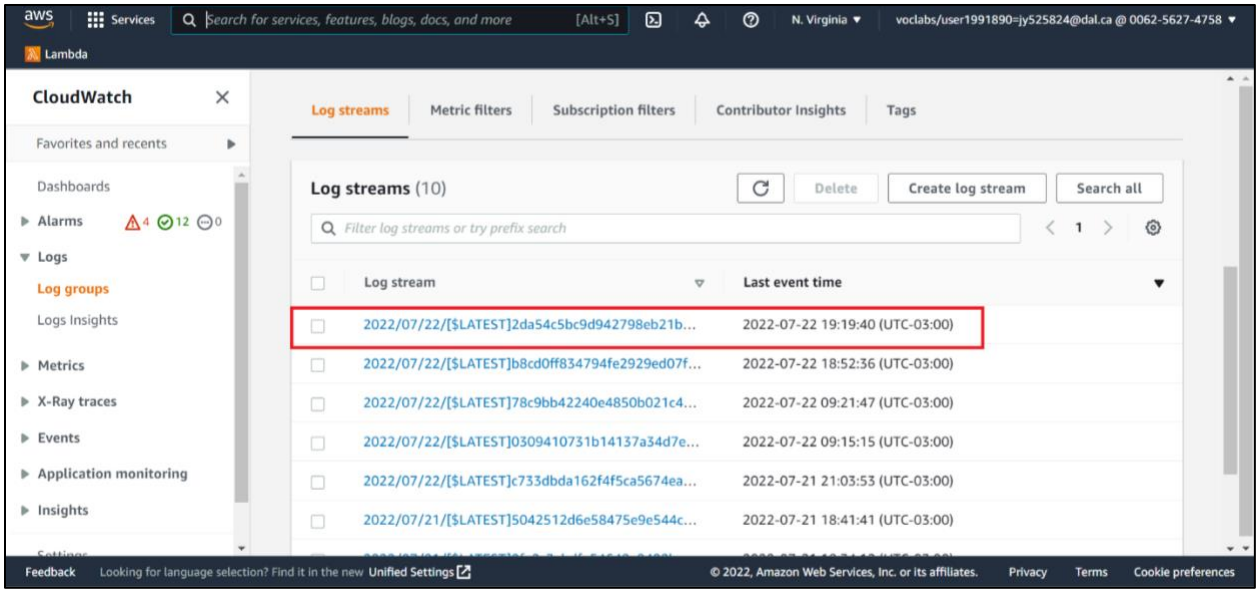


Figure 40: Lambda function is triggered for validating answer

Scenario 6: User provides correct answer to the security question. Lambda function is triggered to validate the answer. After successful validation, user is navigated to Caesar Cipher validation screen.

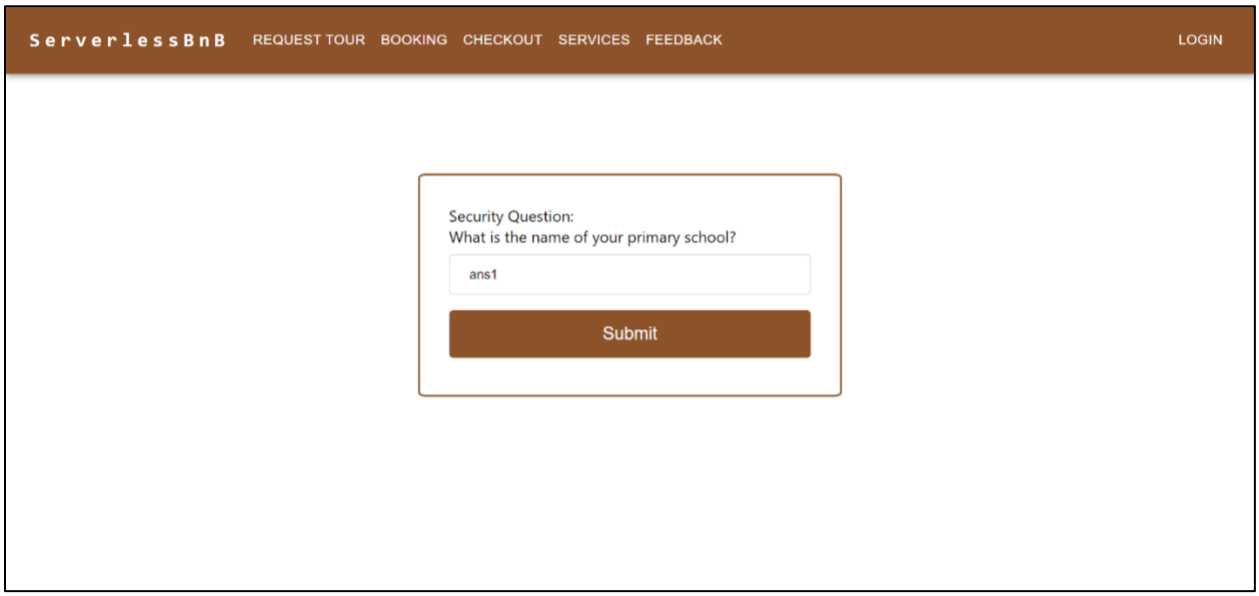


Figure 41: Correct answer to security question

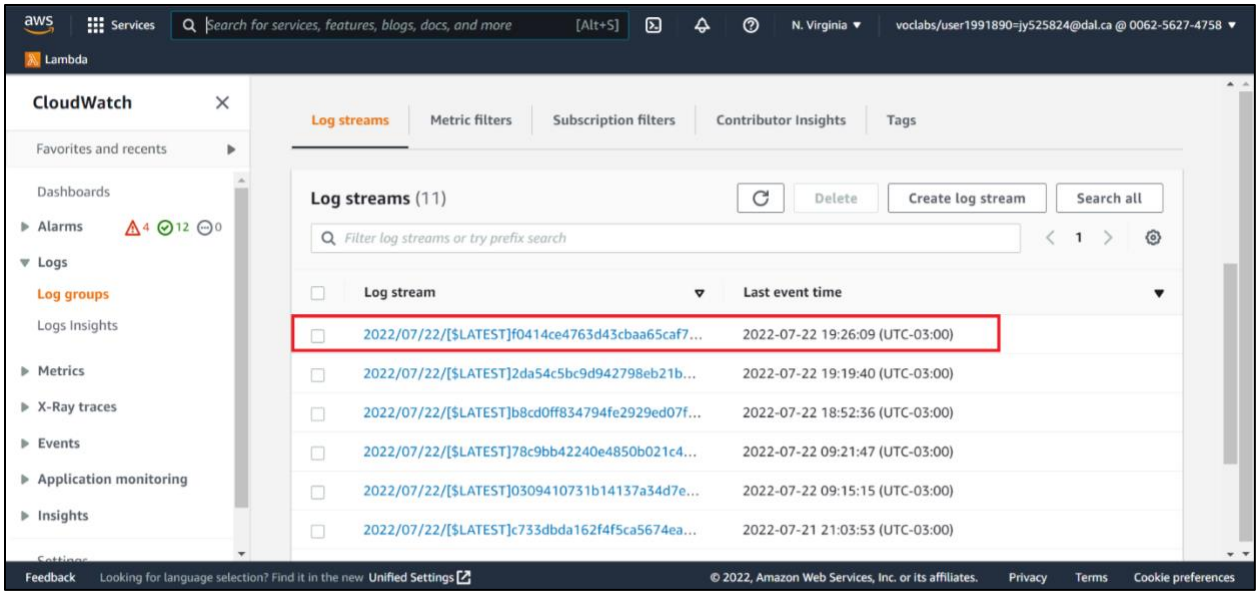


Figure 42: Lambda function is triggered for validation

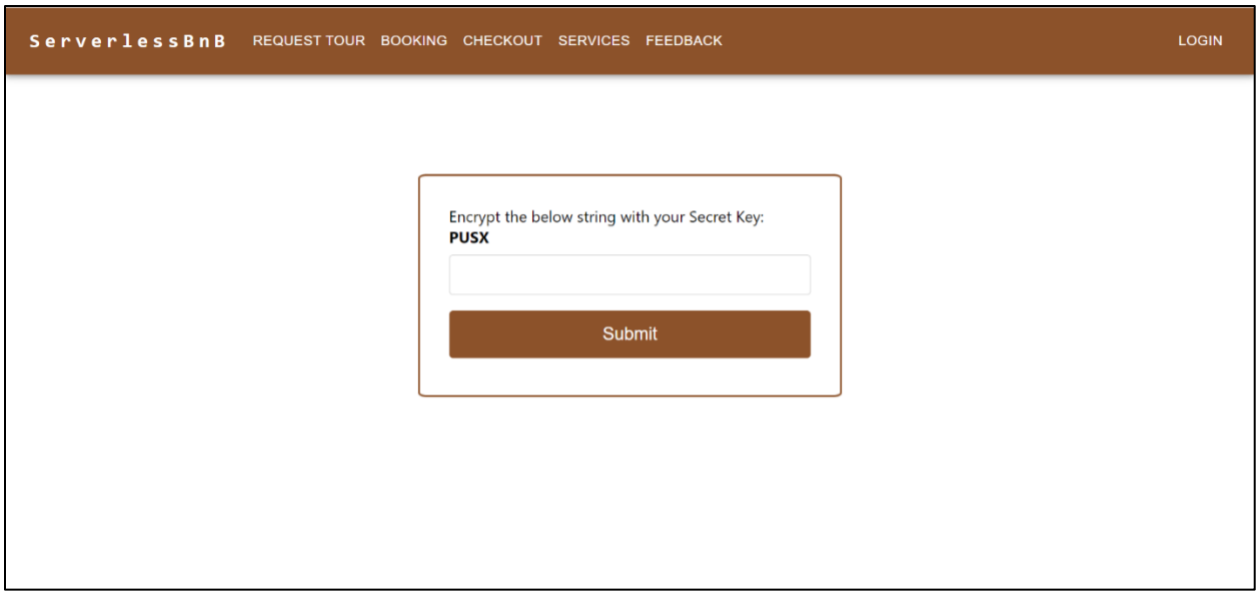


Figure 43: Caesar Cipher validation screen

Scenario 7: User tries to log in without providing encrypted Caesar cipher string. An error toast message is displayed.

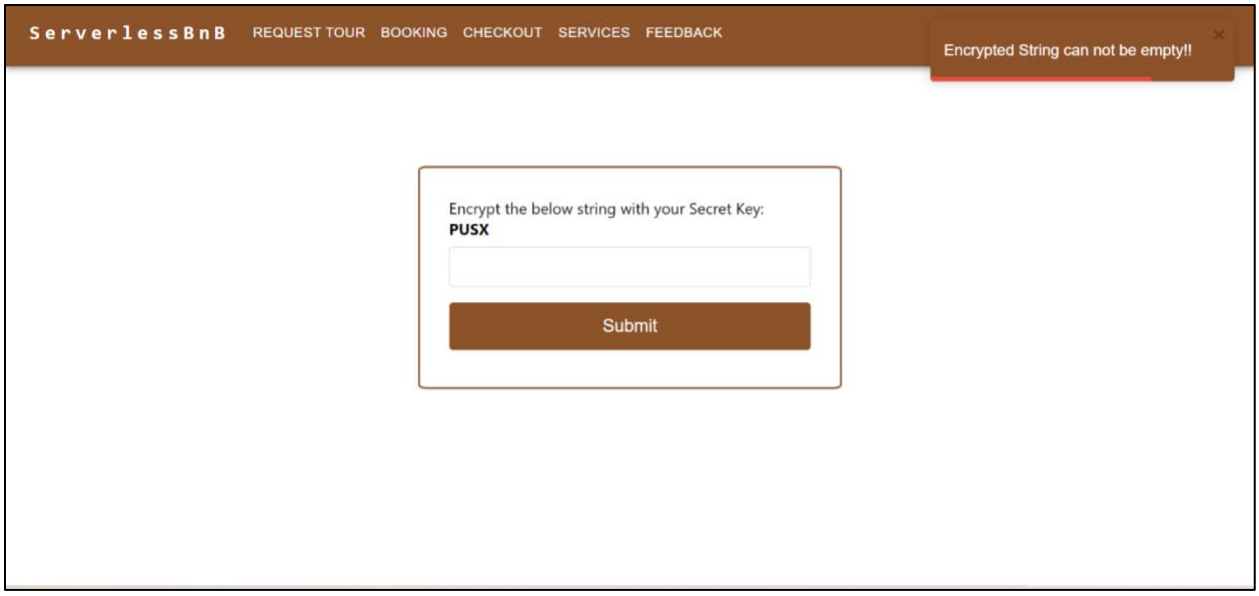


Figure 44: Log in without encrypting Caesar Cipher string

Scenario 8: User provides incorrect encrypted cipher. Google Cloud Function is triggered to validate the encrypted string and an error toast message is displayed.

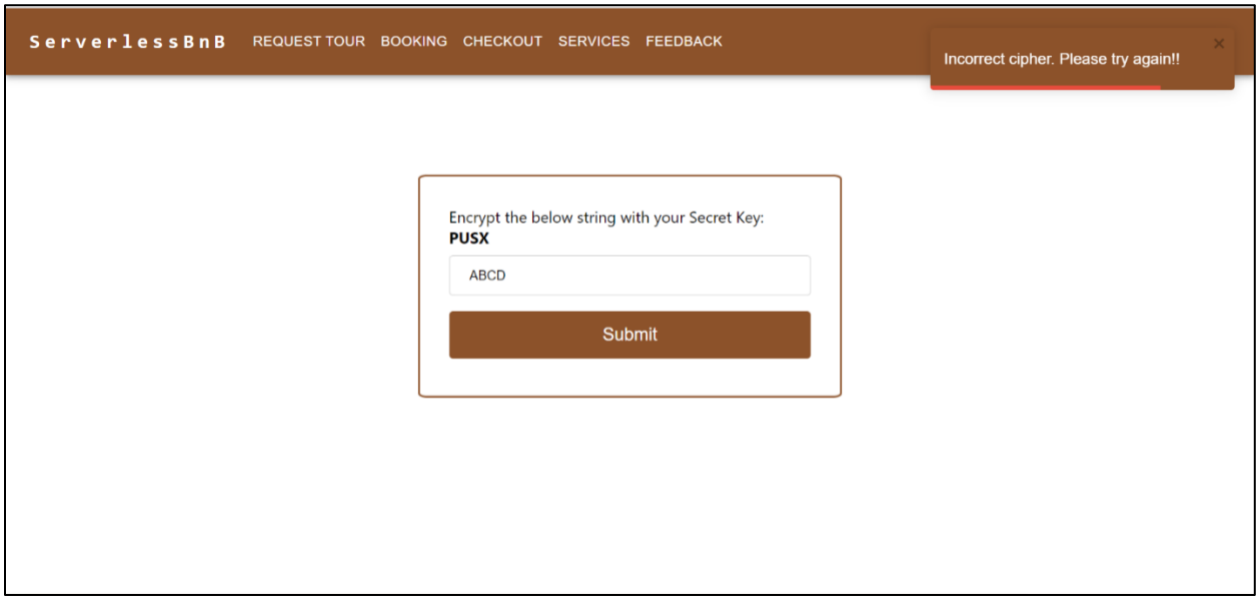


Figure 45: Invalid Cipher error toast message

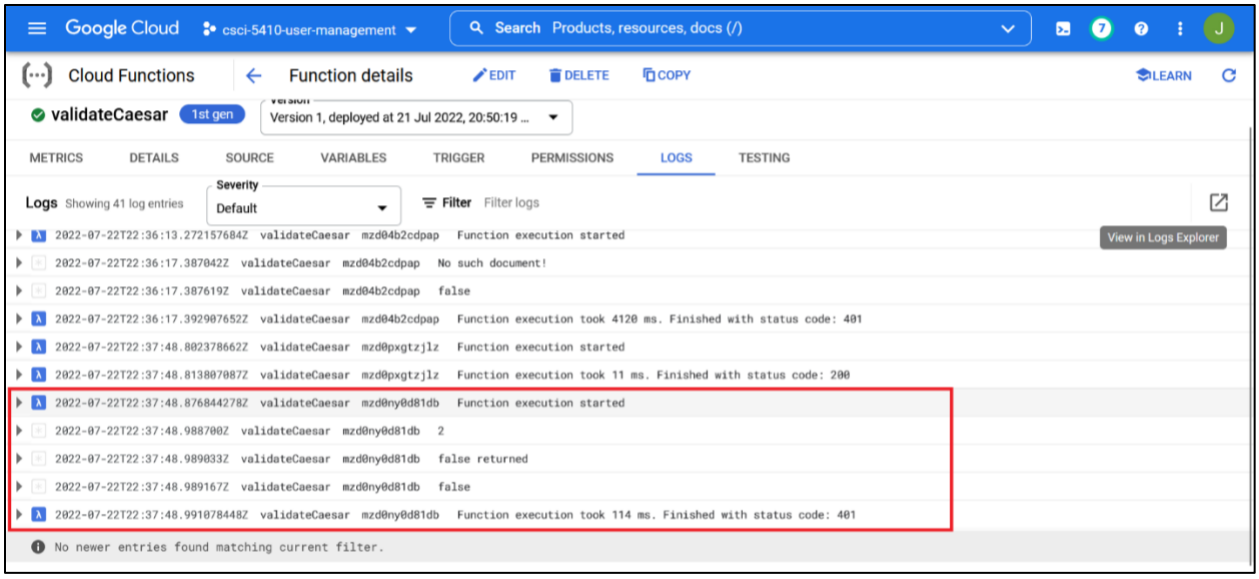


Figure 46: Triggered Google Cloud Function Logs

Scenario 9: User provides correct encrypted cipher with secret key "2". Google Cloud Function is triggered to validate the encrypted string. After successful validation of Caesar cipher, successful log in message is displayed and user is logged in to the application.

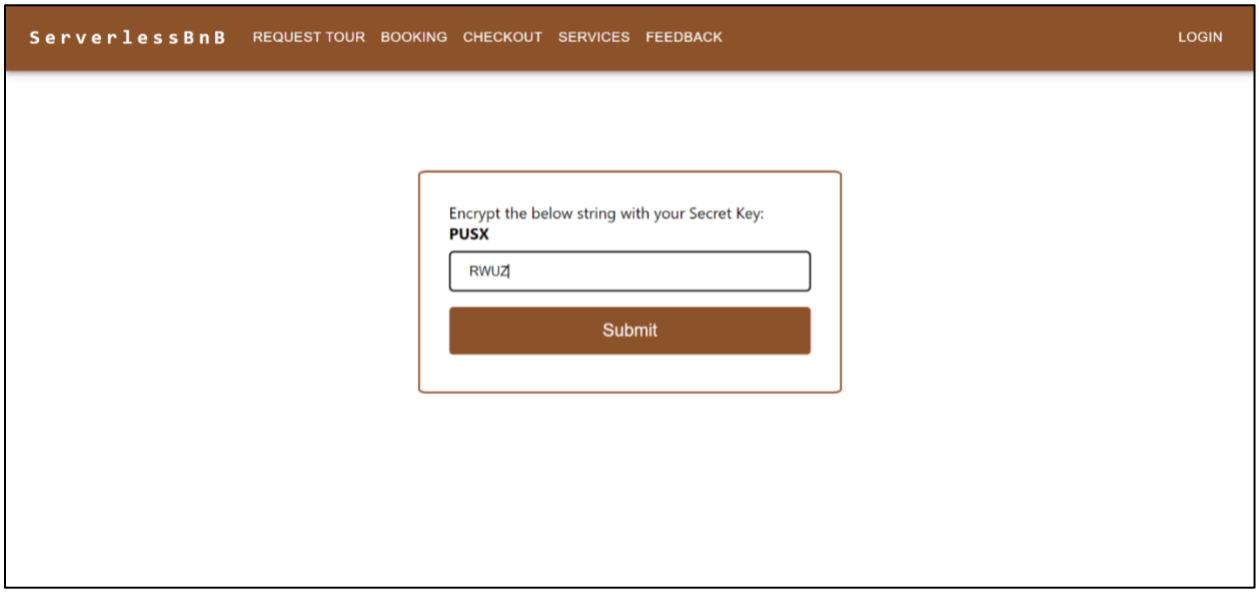


Figure 47: Log in with correctly encrypted Caesar Cipher string with secret key 2

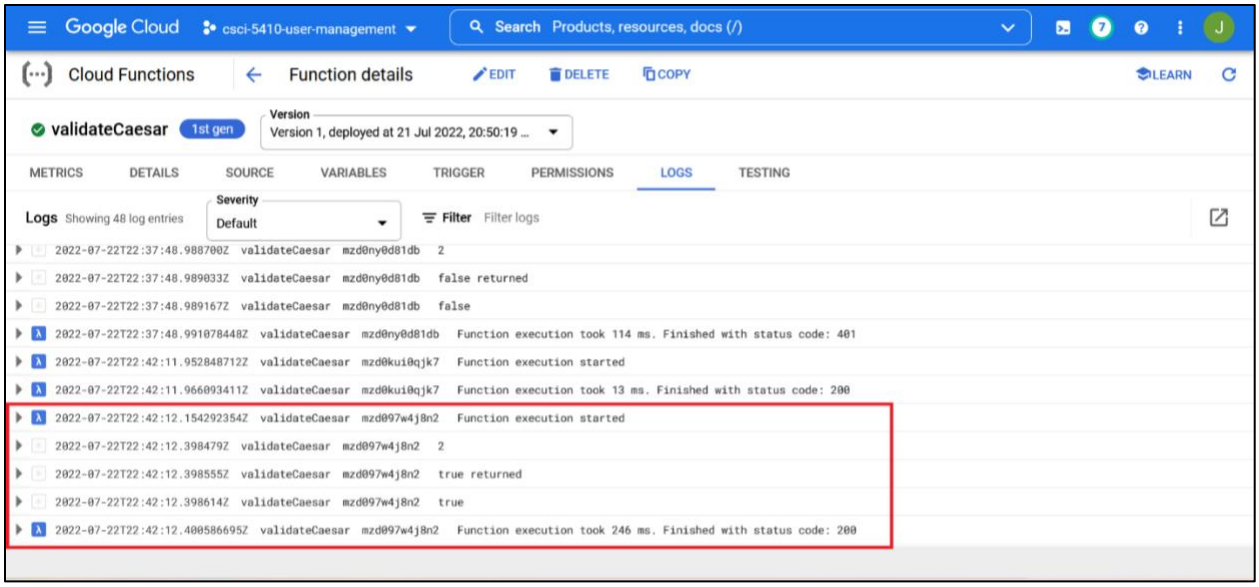


Figure 48: Triggered Cloud Function logs

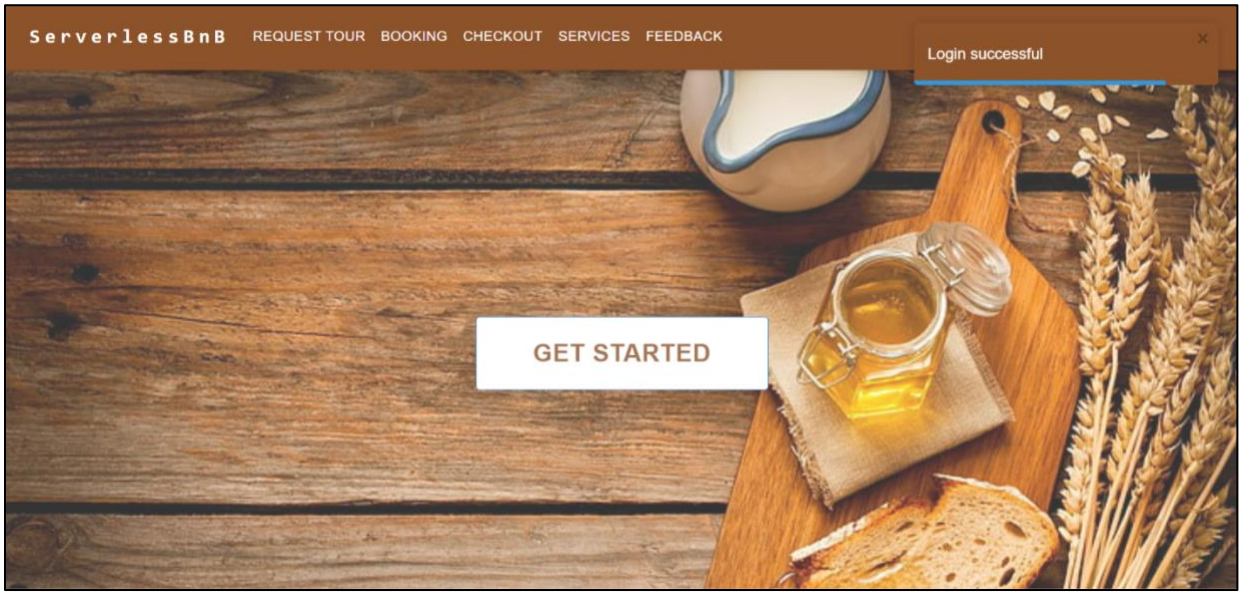


Figure 49: Home page with log in successful message

Kitchen Service

Implementation details

Front-End: React application deployed using S3

- The static assets generated by the React [3] are hosted in an S3 bucket [4] on AWS, where the application is deployed.

Back-End: Node.JS deployed using GCP Cloud Run

- **Node.JS** [17] is a modern programming language which helps in building scalable applications due to its inherent property of handling concurrent requests using Event Loop.

- **GCP Cloud Run [18]** offers a serverless platform to host back-end services. Node.JS code was containerized using Docker and the image was published directly to cloud run. With the help of Cloud Run, application can scale on demand.

Data Store: GCP Firestore

- **Firestore** [5] offers a NoSQL database which follows the Document DB model. Our application needs to be scalable, and we do not have any complex relationships to store. Therefore, choosing Firestore became the preferred choice.

Cloud Services: GCP Cloud Functions, Pub/Sub

- **GCP Cloud Functions [10]**: Cloud functions follow a pay-as-you-go model. With the increase in traffic, it scales up automatically, and we are charged accordingly. Therefore, it follows a serverless architecture.

- **GCP Pub/Sub:** Pub/Sub [11] is an asynchronous message delivery system where messages get transferred between publishers and subscribers. Publisher publishes their messages through topics which are subscribed as subscriptions by subscribers. The subscribers who subscribe to a particular topic will receive messages from the publisher of that topic. Messages can be transferred either through pull or push where pull is done by the subscribers when required and push messages are automatically delivered to a subscriber. Until and unless all the subscribers give an acknowledgement after receiving a message, message stays in the message queue. This ensures robustness and reliability of the communication.

Pseudo-Code

- **Getting food items from the Firestore to display to the customer as breakfast options.**

  When the customer logs in to the web app, he/she has the option to order breakfast. To give various options for the breakfast, this method fetches all the food items from the Firestore.

```
13  const [selectedFoodOption, setSelectedFoodOption] = useState("Poha");
14  const [selectedFoodPrice, setSelectedFoodPrice] = useState(10);
15  //useEffect(() => {}, [foodOrdered]);
16  useEffect(() => {
17    axios("https://kitchen-service-kc2rqvhqga-uc.a.run.app/getFoodItems").then(
18      (res) => {
19        console.log(res.data);
20        setFoodOptions(res.data);
21      }
22    );
23    setSelectedFoodOption(foodOptions[0].foodItem);
24    setSelectedFoodPrice(foodOptions[0].price);
25  }, []);
```

Figure 50: Front-End code for calling backend API to get breakfast options

```
11  const getFoodItems = async (req, res) => {
12    const foodItemsRef = firestore.collection("foodItems");
13    const snapshot = await foodItemsRef.get();
14    let foodItems = [];
15    if (snapshot.empty) {
16      console.log("No matching documents.");
17      return res.status(404);
18    }
19    snapshot.forEach((foodItem) => {
20      console.log(foodItem.data());
21      foodItems.push(foodItem.data());
22    });
23    return res.status(200).json(foodItems);
24  };
```

Figure 51: Back-End code for getting breakfast options from the Firestore

Code Explanation
Line 12: Gets the foodItems collection reference from the Firestore.
Line 13: By awaiting, gets the snapshot from the collection reference.
Line 15-17: If the snapshot is empty return not found as status.
Line 19-22: For each food item found, adds it into the foodItems arrays and returns it.

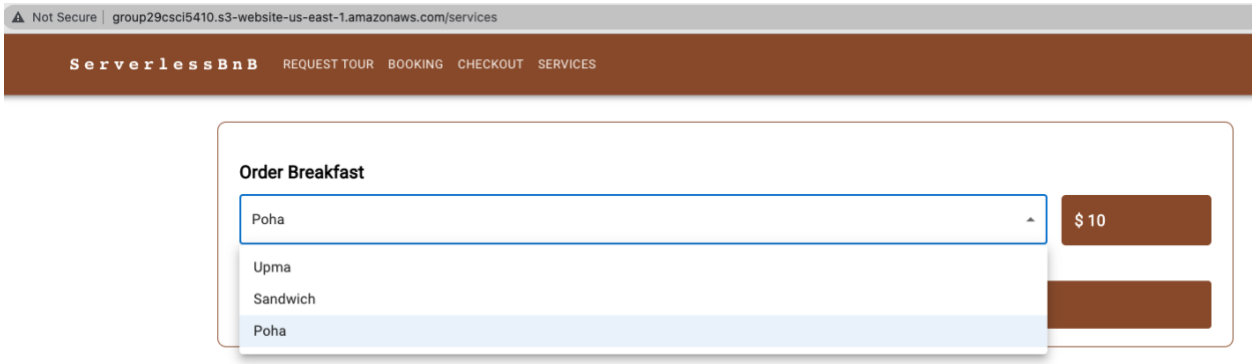


Figure 52: Breakfast options after being fetched.

- **Place order for the selected food item by a customer.**
  Once the user has decided which breakfast option, he/she wants to order for the breakfast. User clicks on Order Breakfast and a request with customer id, order and price is placed to the backend.

```
27  function placeOrder() {
28    const options = {
29      method: "post",
30      url: "https://kitchen-service-kc2rqvhqga-uc.a.run.app/placeorder",
31      headers: {
32        "Access-Control-Allow-Origin": true,
33        "Content-Type": "application/json",
34      },
35      data: {
36        customerId: "56789",
37        order: selectedFoodOption,
38        price: selectedFoodPrice,
39      },
40    };
41    axios(options).then((res) => {
42      console.log(res);
43      setFoodOrdered(true);
44    });
45  }
```

Figure 53: Front-End code for calling backend API to place order

```
26  const placeOrder = async (req, res) => {
27    console.log(req.body);
28    const { customerId, order, price } = req.body;
29    const message = JSON.stringify({
30      customerID: customerId,
31      order: order,
32      mealPrice: price,
33      createdDate: new Date(),
34    });
35    const messageId = await sendMessage(topicID, message);
36    return res.status(200).json({
37      success: true,
38      message: `Message ${messageId} published :)`,
39    });
40  };
41
```

Figure 54: Back-End code for sending order information to pub/sub

Code Explanation

Line 28: Gets the customer id and the order that customer has place and the price of that order from the request.
Line 29-34: Stringify the JSON object to send it as a message.
Line 35: Sends a message to GCP Pub/Sub and waits for the deliver response.
Line 36-39: Return success status on correct delivery of the message.

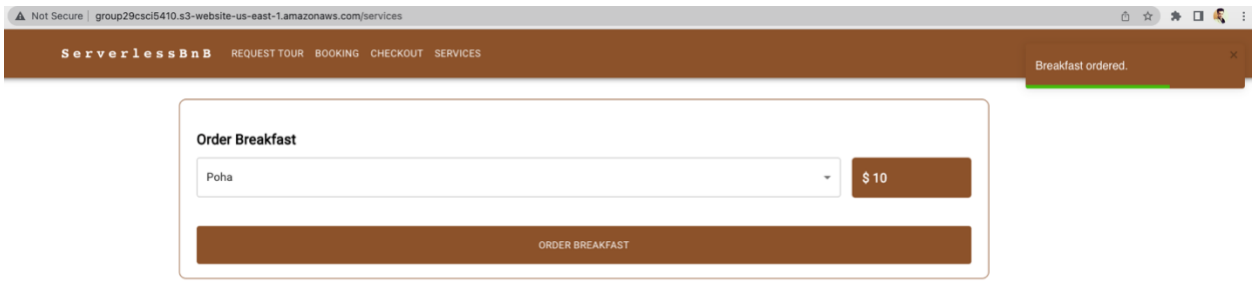


Figure 55: After successfully placing breakfast order.

- **Create a common send message function to send any message dynamically based on topic id and message.**

  While placing an order, order is sent to GCP Pub/Sub. With the use of Pub/Sub. Kitchen service follows Event driven architecture and can scale independently without waiting for the message to be processed. Also, at the time of extensive load, orders get queued and get processed as the kitchen is available to process.

```
You, 4 days ago | 1 author (You)
1  // Imports the Google Cloud client library      You, 4 days ago • Updated fro
2  const { PubSub } = require("@google-cloud/pubsub");
3  const pubSubClient = new PubSub();
4
5  const sendMessage = async (topicID, message) => {
6    const messageBuffer = Buffer.from(message);
7    try {
8      const messageId = await pubSubClient.topic(topicID).publish(messageBuffer
9      console.log(`Message ${messageId} published.`);
10     return messageId;
11   } catch (error) {
12     console.error(`Received error while publishing: ${error.message}`);
13   }
14 };
15
16 module.exports = { sendMessage };
17
```

Figure 56: Back-End code for sending order information to pub/sub

Code Explanation

Line 1-2: Gets the GCP Pub/Sub from SDK.
Line 6: Creates a buffer from the message.
Line 8-13: Sends the message to the publisher, and if its success returns the message it otherwise prints and error message to the console.

- **Cloud function code for processing GCP Pub Sub message.**

  Cloud function is triggered whenever there is a message inside the Pub/Sub. It acts as a consumer for the Pub/Sub. While processing the message it stores the details inside the Firestore.

```
1  /**
2   * Triggered from a message on a Cloud Pub/Sub topic.
3   *
4   * @param {!Object} event Event payload.
5   * @param {!Object} context Metadata for the event.
6   */
7  exports.helloPubSub = (event, context) => {
8    const message = event.data
9      ? Buffer.from(event.data, "base64").toString()
10     : "Hello, World";
11   console.log(message);
12
13   const retrievedMessage = JSON.parse(message);
14   console.log("New message: " + retrievedMessage);
15   const Firestore = require("@google-cloud/firestore");
16   // Use your project ID here
17   const PROJECTID = "hotel-management-serverless";
18   const COLLECTION_NAME = "orders";
19   const firestore = new Firestore({
20       (property) timestampsInSnapshots: boolean
21     timestampsInSnapshots: true,
22   });
23
24   return firestore
25     .collection(COLLECTION_NAME)
26     .add(retrievedMessage)
27     .then((doc) => {
28       console.info("stored new doc id#", doc.id);
29     })
30     .catch((err) => {
31       console.error(err);
32     });
33 };
```

Figure 57: Cloud function code for process order message

Code Explanation

Line 8-11: Gets the message from the buffer.
Line 13-22: Parses the message back to JSON and creates a connection with Firestore.
Line 24-33: Adds a new document to the Firestore collection and logs the document id and error message if error encountered.

- **Cloud function code for sending email to customer on processing food order.**

Whenever the order is saved inside the Firestore after processing order, cloud function gets triggered with Firestore Create trigger. This function uses the send grid email API to send email to the user who ordered the food.

Figure 58: Cloud function code for sending email to customer on processing order

Code Explanation

Line 1-5: Configures the sendgrid API [19].
Line 7-14: Creates a function to send email to user with the given message.
Line 16-27: Creates a dynamic message based on user id to send to the user email.

- **Cloud function code for getting orders for a customer within a date range.**

  During the checkout, we need to get all the orders that customer placed in between a certain time range. This is directly fetched from the Firestore using Node.JS API in the backend.

```
42  const getOrdersByCustomer = async (req, res) => {
43    console.log(req.query);
44    let { customerId, startDate, endDate } = req.query;
45    startDate = new Date(startDate);
46    endDate = new Date(endDate);
47    const ordersRef = firestore.collection(COLLECTION_NAME);
48    const snapshot = await ordersRef.get();
49    if (snapshot.empty) {
50      console.log("No matching documents.");
51      return res.status(404);
52    }
53    let orders = [];
54    snapshot.forEach((doc) => {
55      console.log(doc.id, "=>", doc.data());
56      const createdDate = new Date(doc.data().createdDate);
57      console.log(createdDate.getFullYear());
58      if (
59        createdDate.getDate() >= startDate.getDate() &&
60        createdDate.getDate() <= endDate.getDate() &&
61        createdDate.getFullYear() >= startDate.getFullYear() &&
62        createdDate.getFullYear() <= endDate.getFullYear()
63      ) {
64        console.log(doc.id + " selected.");
65        orders.push(doc.data());
66      }
67    });
68    return res.status(200).json({
69      success: true,
70      orders: orders,
71      message: `Total orders by the customer ${customerId} : ${orders.length}`,
72    });
73  };
```

Figure 59: Back-End code for getting orders of a customer

Code Explanation

Line 42-46: Gets the customer id, and the date-range for which orders needs to be fetched.
Line 47-48: By awaiting, gets the snapshot from the collection reference.
Line 49-53: If the snapshot is empty return not found as status.
Line 54-67: For each food order found, adds it into the orders arrays and filter it out with the start and end date.

- **Get all the orders for visualizing orders preference.**

To visualize what kind of orders is preferred, get orders endpoint gets all the orders from the Firestore.

```
81  const getOrders = async (req, res) => {
82    const ordersRef = firestore.collection(COLLECTION_NAME);
83    const snapshot = await ordersRef.get();
84    if (snapshot.empty) {
85      console.log("No matching documents.");
86      return res.status(404);
87    }
88    let orders = [];
89    snapshot.forEach((doc) => {
90      console.log(doc.id, "=>", doc.data());
91      orders.push(doc.data());
92    });
93    return res.status(200).json({
94      success: true,
95      orders: orders,
96      message: `Total orders by the customers: ${orders.length}`,
97    });
98  };
```

(property) getOrders

Figure 60 Code for getting orders

Code Explanation

Line 83: By awaiting, gets the snapshot from the collection reference.
Line 84-88: If the snapshot is empty return not found as status.
Line 89-92: For each food order found, adds it into the orders arrays.

Deployed GCP Services

Backend on GCP Cloud Run

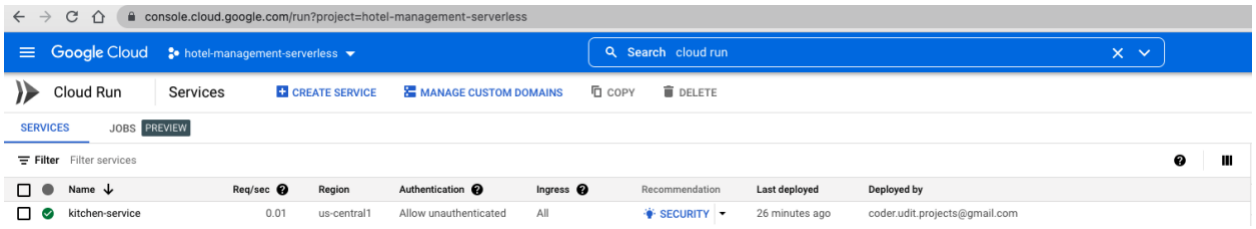


Figure 61: Deployed backend as a service on Cloud run

Cloud functions for processing message and Firestore triggers

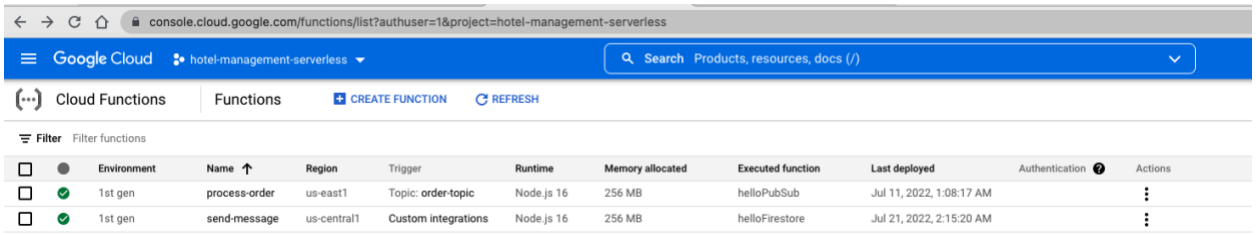


Figure 62: Deployed cloud functions to process message and Firestore triggers.

Pub/Sub for message passing

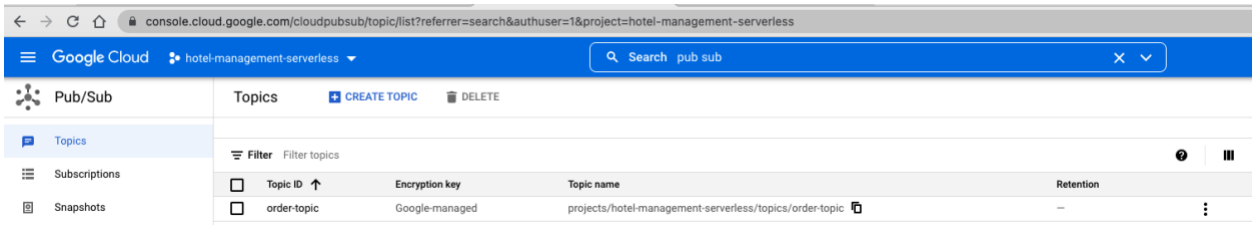


Figure 63: Pub/Sub for passing order messages.

Firestore for managing inventory and orders

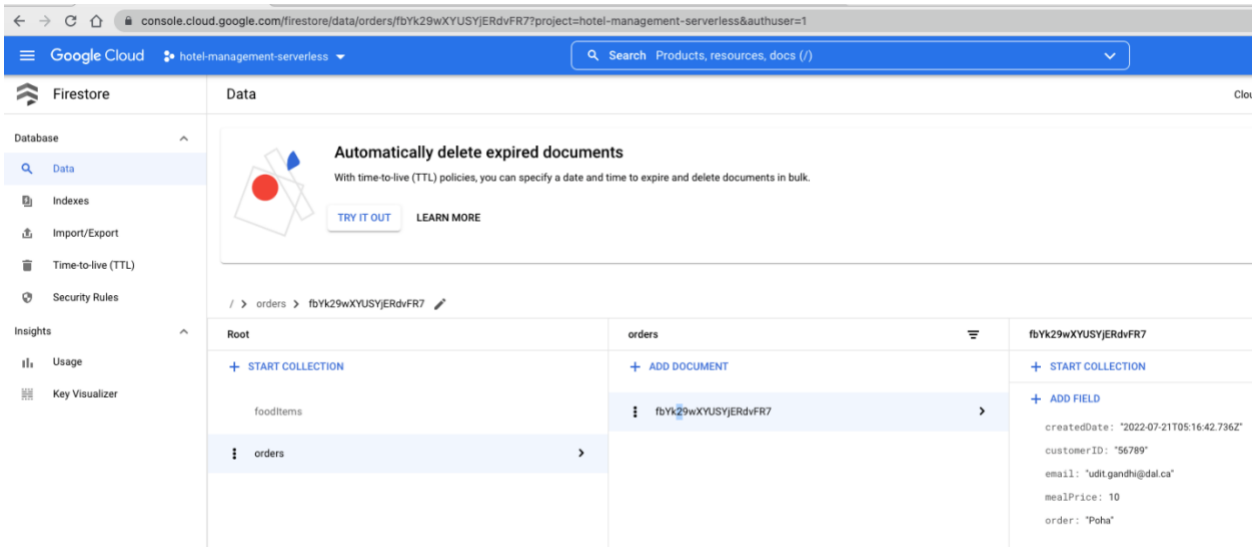


Figure 64: Firestore for storing inventory and orders

Flowchart

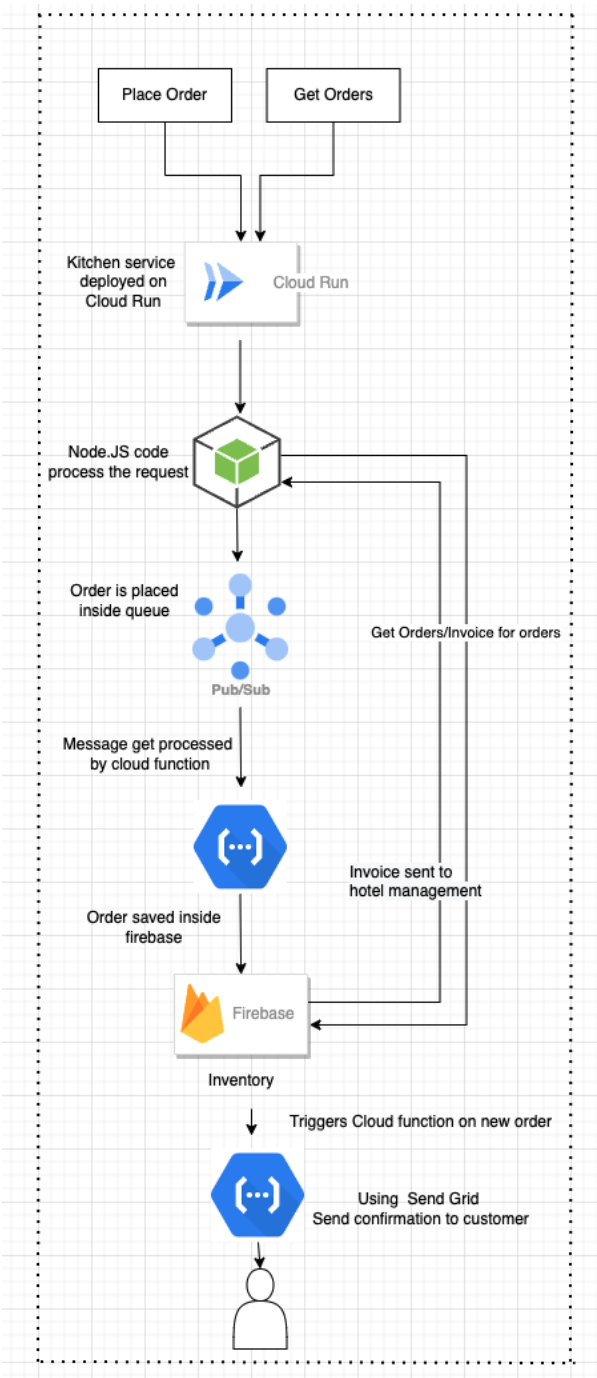


Figure 65: Interaction diagram for Kitchen Service

Kitchen service test cases with testing

1. **Customer can order breakfast only between 6 AM-11AM. If user tries to order before or after the time, then an error message will be shown.**

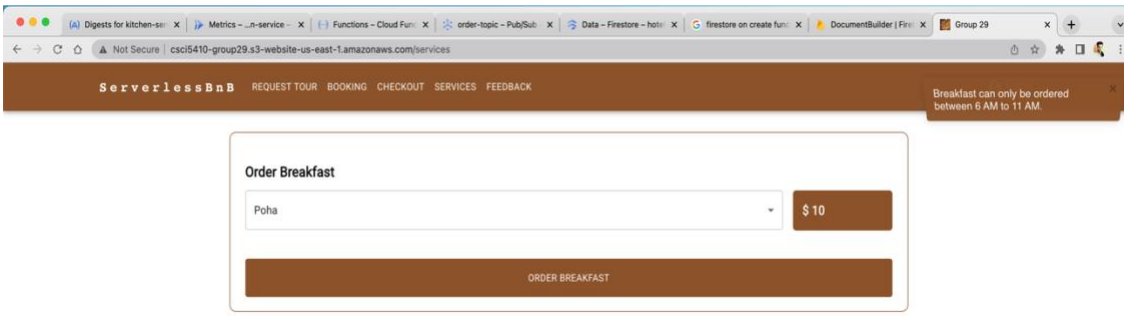


Figure 66: Application showing already ordered message when I tried to order food not between 6 AM and 11 AM.

2. Order should get placed when customer clicks on the order breakfast.

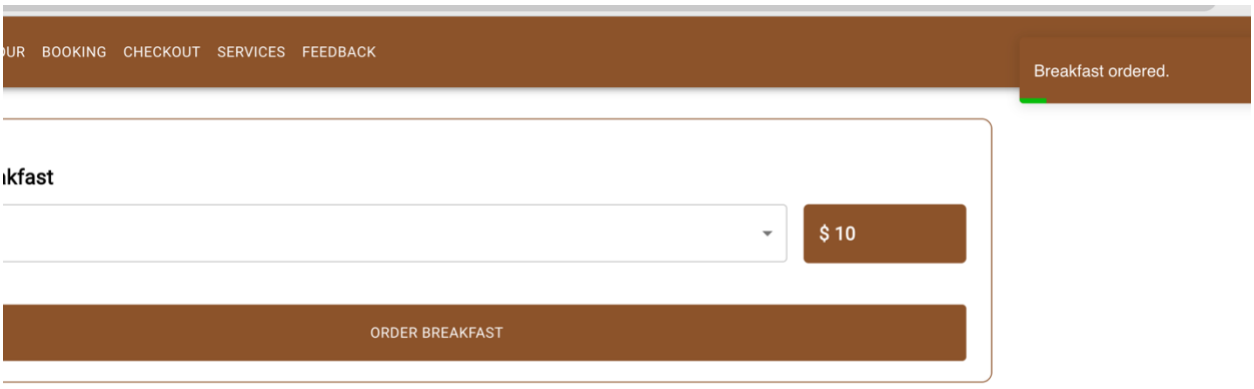


Figure 67: Breakfast ordered on clicking order breakfast.

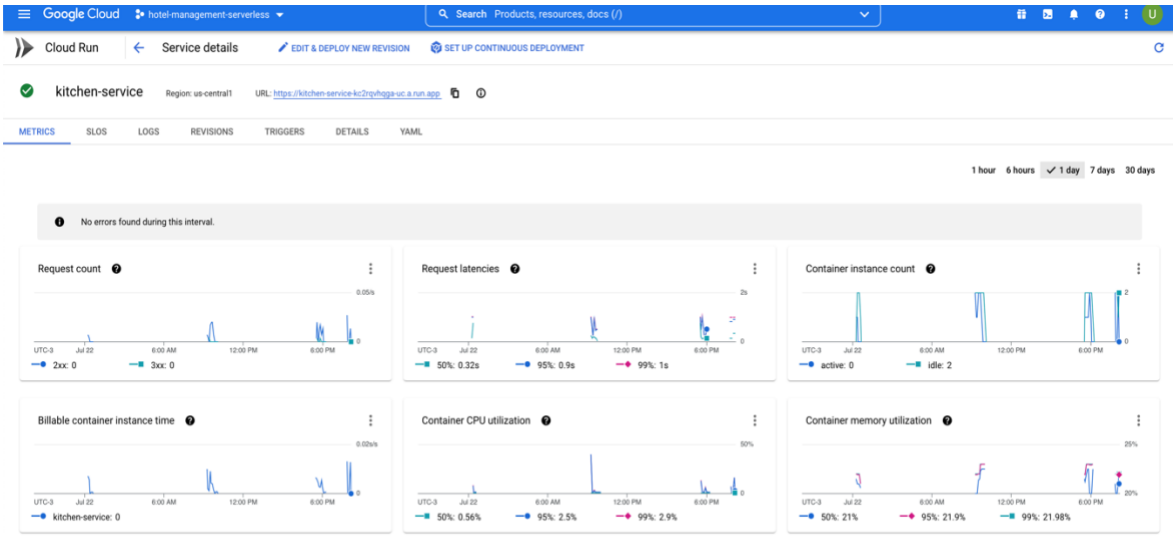


Figure 68: Metrics for receiving of order by the backend kitchen service.

3. **Customer can order breakfast only once per day.**

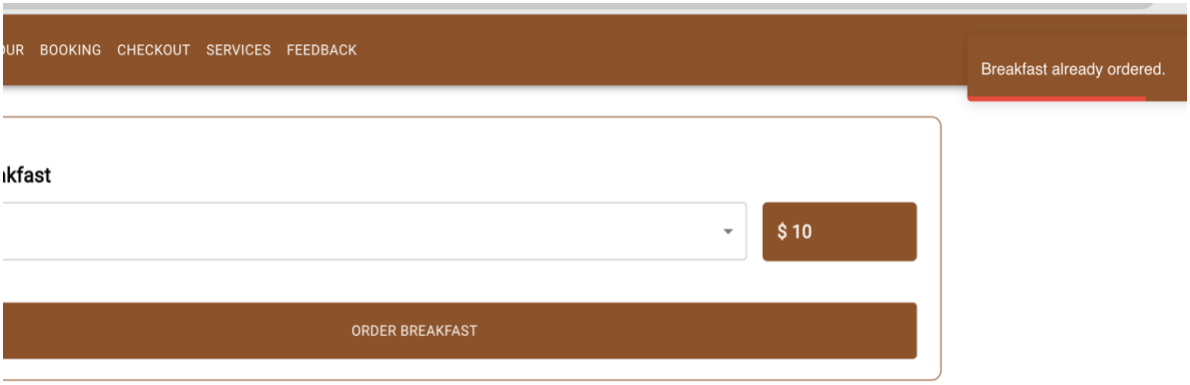


Figure 69: Breakfast already ordered error message on ordering again for the same day.

4. **All the food items stored inside the Firestore should get displayed to the user.**

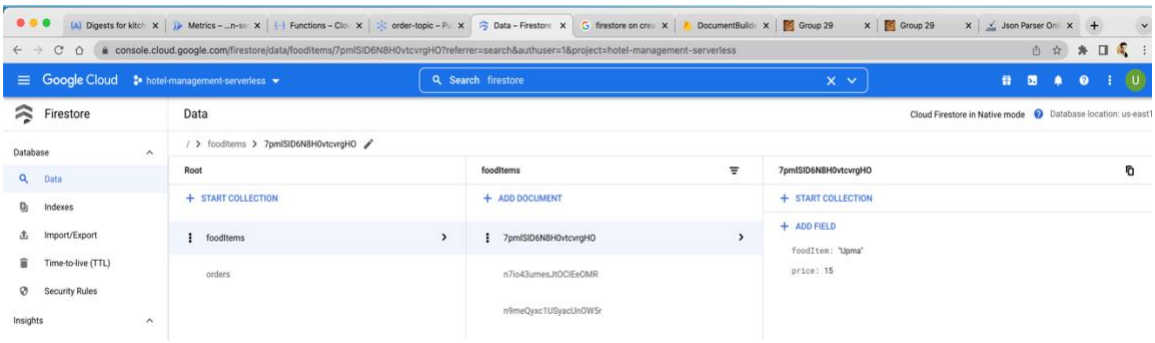


Figure 70: Firestore has 3 food options to be displayed.

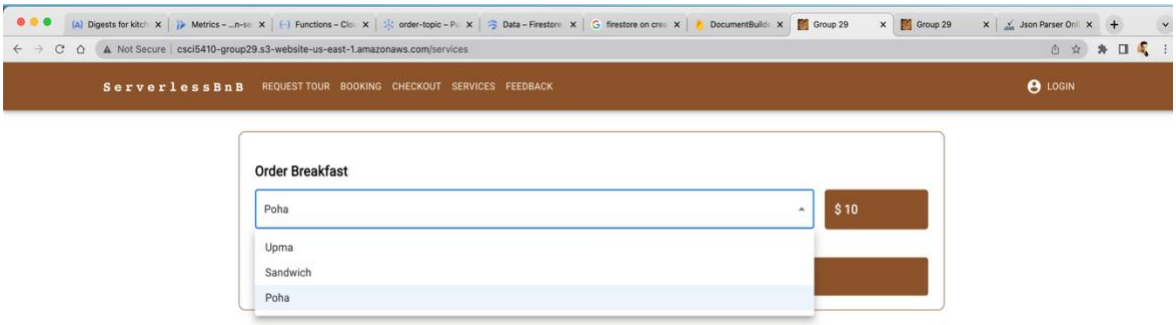


Figure 71: 3 food options get displayed on the application.

5. Order should be sent as a message on “order-topic” after receiving it from the backend.

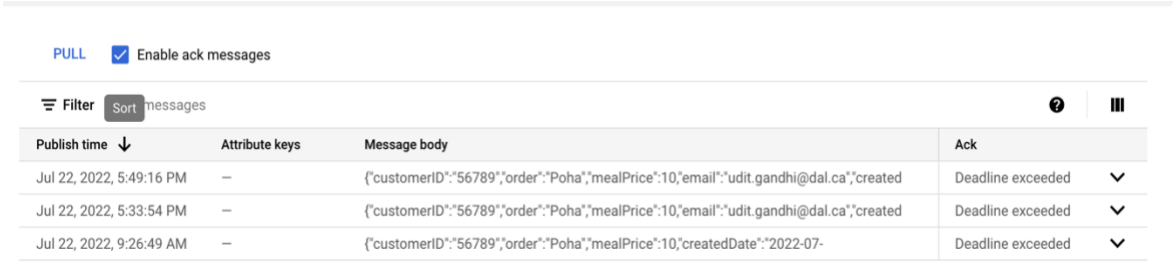


Figure 72: Order messages on “order-topic”.

6. Order Message should get processed by the cloud function “process-order”.

Figure 73: Logs for processing of order message by “process-order” cloud function

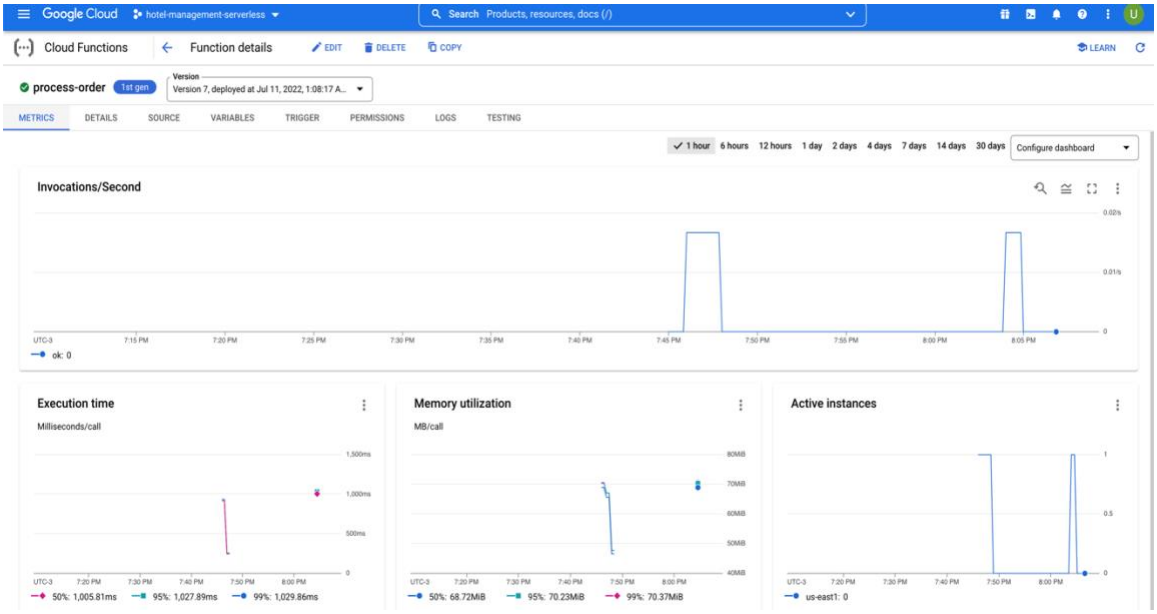


Figure 74: Metrics for "process-order" cloud function

7. "process-order" cloud function should store the data inside the Firestore correctly.

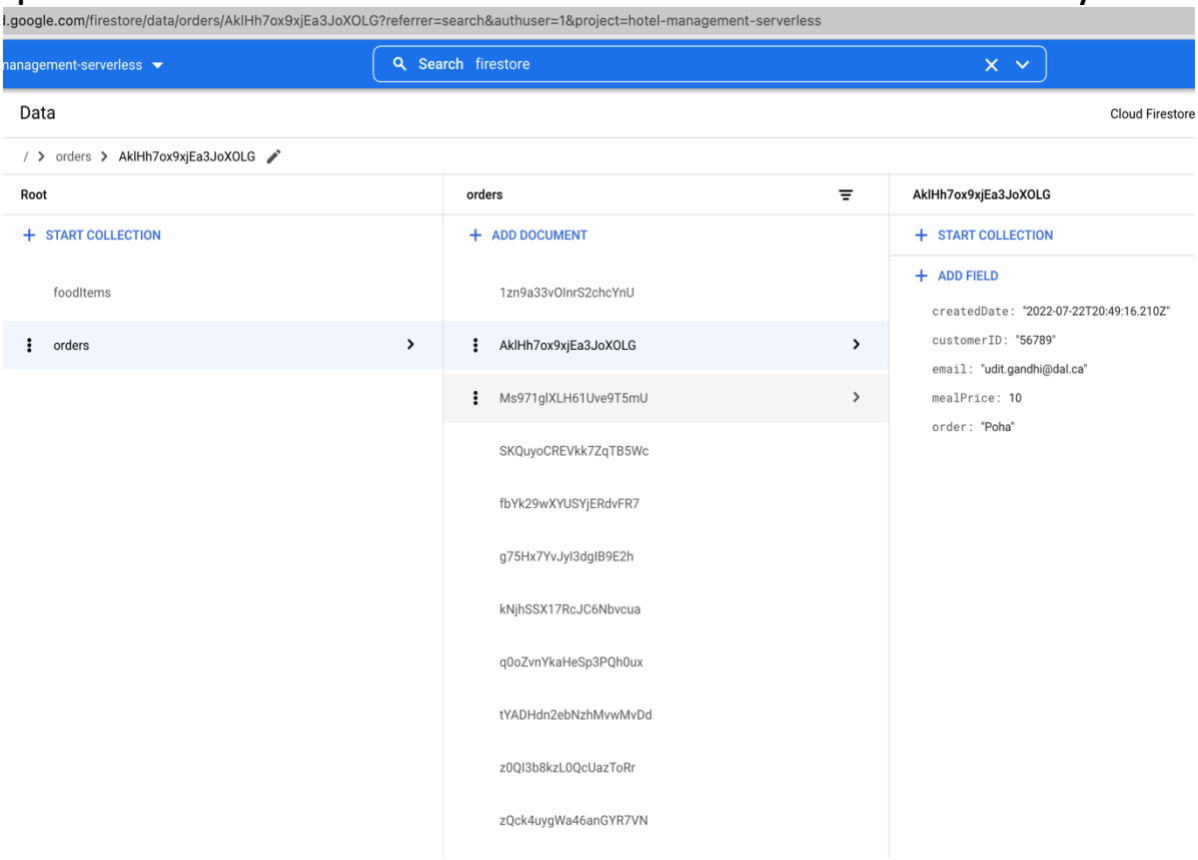


Figure 75: Stored order inside Firestore collection "orders".

8. **On storage of order inside Firestore, it should trigger another cloud function "send-message".**

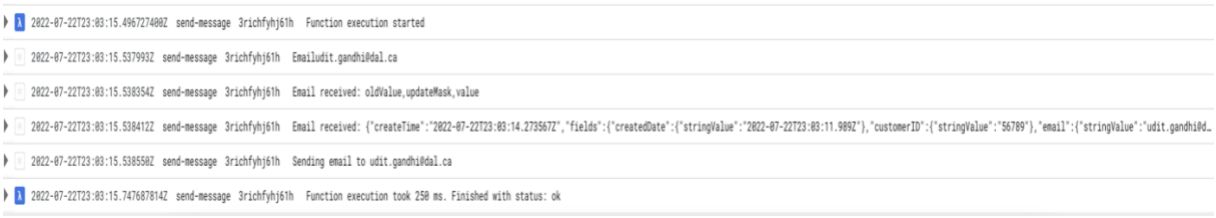


Figure 76: Logs for triggered "send-message" cloud function.

9. **"send-message" cloud function should send email to the customer with the help of send grid API.**

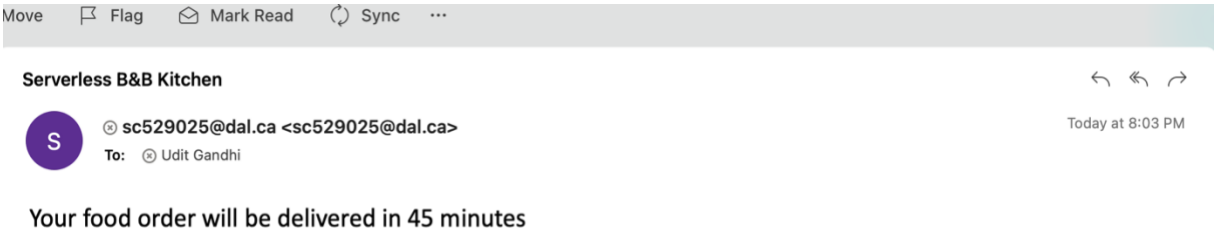


Figure 77: Received email from send grid api.

Limitations

1. Currently the customer can book only 1 meal for a day.
2. Customer is allowed to book meal for only 1 day. We can add future date range for ordering food.

Booking Service

Searching Rooms

When an unregistered user accesses the website, they can search for the available room on the homepage.

1. The React [3] app deployed on S3 bucket [4] sends the static content to the user's browser.
2. The website displays all the available rooms on the home page by calling the GET method for the /rooms endpoint.
3. The request gets passed to the API Gateway [16].

4. The API gateway passes this request to the lambda function for processing this information.
5. The lambda function cleans and processes this information and then queries the DynamoDB database [6].
6. DynamoDB returns the list of available rooms to the lambda function.
7. The lambda function returns this information to the API gateway.
8. The API gateway sends the response back to the React frontend.

Booking Room

When registered user accesses the website, they can book an available room on the website.

1. The React app deployed on S3 sends the static content to the user's browser.
2. The website displays all the available rooms on the home page by calling the GET method for the /rooms endpoint.
3. The user clicks on the book button on one of the available rooms.
4. The request gets passed to the lambda URL.
5. The lambda function cleans and processes this information and then updates the DynamoDB [6].
6. The lambda function returns this success status information to the API gateway [16].
7. The API gateway sends the response back to the React frontend.

Figure below shows the flow diagram for searching and booking room flows.

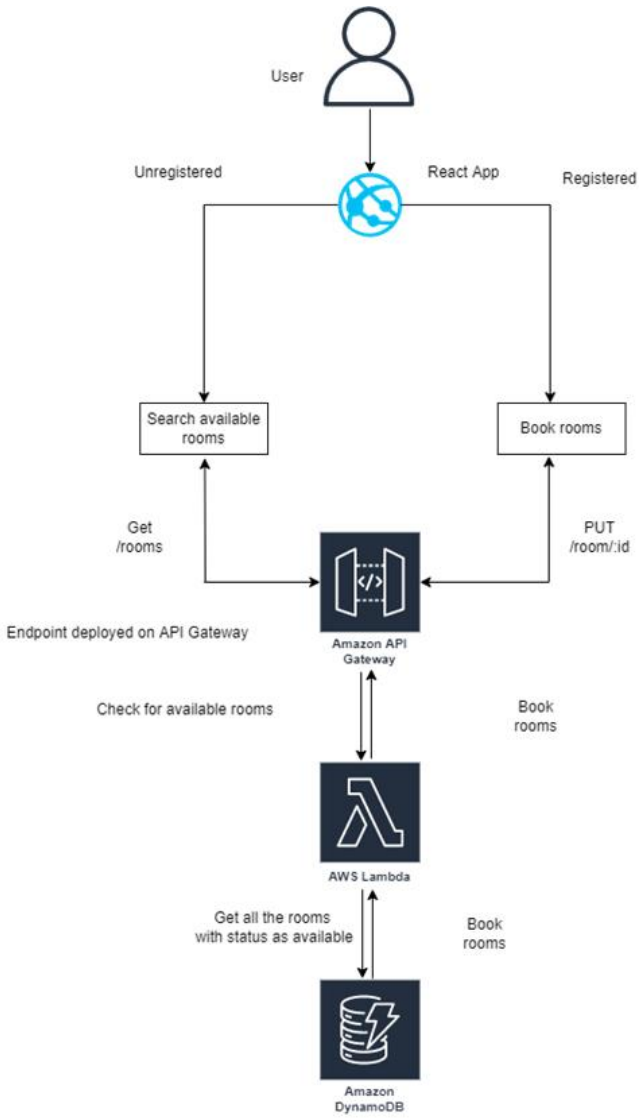


Figure 78 Booking and Searching Rooms User Flow Diagram

High Level Implementation

On a high level, the booking room feature utilizes three services: S3, Lambda Function, and DynamoDB table.

S3: The S3 bucket [4] hosts the static content of the website for booking rooms.

Lambda Function: The lambda function [9] has the logic for booking rooms, adding rooms, and checking out.

DynamoDB: The DynamoDB [6] table has Booking and Room tables.

Front-End Code Details:
The front-end in React code utilized four modules for interacting with the backend service deployed on the lambda function.

1. HTTP Booking Service

   The HTTP booking service has the API URL of the lambda function [9]. It also has all the interceptors for handling errors. The module is a wrapper for sending all the request to the backend.

```
import axios from "axios";
import { toast } from "react-toastify";

axios.defaults.baseURL =
  "https://xqet4nlofotprm2yiiz6rvrcm40giliq.lambda-url.us-east-1.on.aws/";

axios.interceptors.response.use(null, (error) => {
  const expectedError =
    error.response &&
    error.response.status >= 400 &&
    error.response.status < 500;

  if (!expectedError) {
    toast.error("Unexpected error occurred");
  }

  return Promise.reject(error);
});

export default {
  get: axios.get,
  put: axios.put,
  post: axios.post,
  delete: axios.delete,
};
```

2. Booking Service

   The booking service has the code for fetching booking for a particular customer. It also has the function for adding new booking for a specific customer.

```
import http from "./httpBookingService";
```

```
const bookingsAPIEndpoint = "/bookings/";

export const getBookings = (id) => {
  return http.get(`${bookingsAPIEndpoint}${id}`);
};

export const addBooking = (booking) => {
  return http.post(`${bookingsAPIEndpoint}`, {
    customerID: booking.customerID,
    type: booking.type,
    active: true,
  });
};
```

3. Room Service

   The room service has the code for fetching all the available rooms from the DynamoDB table.

```
import http from "./httpBookingService";

const roomsAPIEndpoint = "/rooms/";

export const getRooms = () => {
  return http.get(roomsAPIEndpoint);
};
```

4. Checkout Service

   The checkout service has the code for checking out a customer by setting the active status of the booking to false.

```
import http from "./httpBookingService";

const checkoutAPIEndpoint = "/checkout/";

export const checkout = (checkoutBody) => {
  return http.post(`${checkoutAPIEndpoint}`, {
    type: checkoutBody.type,
    customerID: checkoutBody.customerID,
  });
};
```

Lambda Function Configuration

Created a lambda function [9] with lambda URLs enabled.

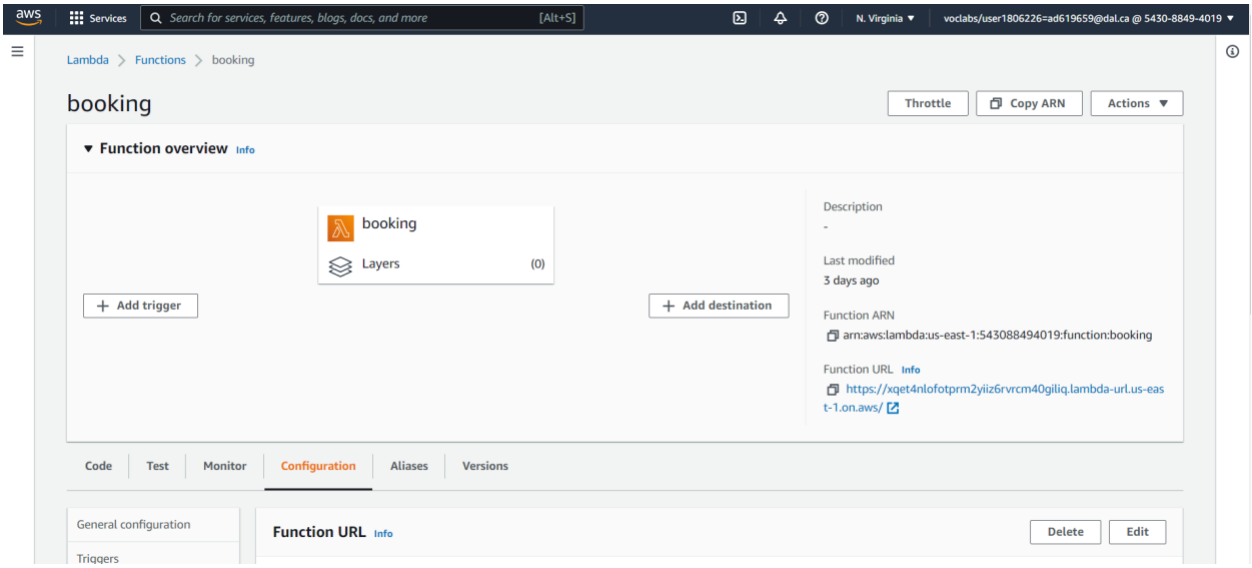


Figure 79 Booking Lambda Function

The function has CORS enabled so that the frontend application can communicate with it properly.

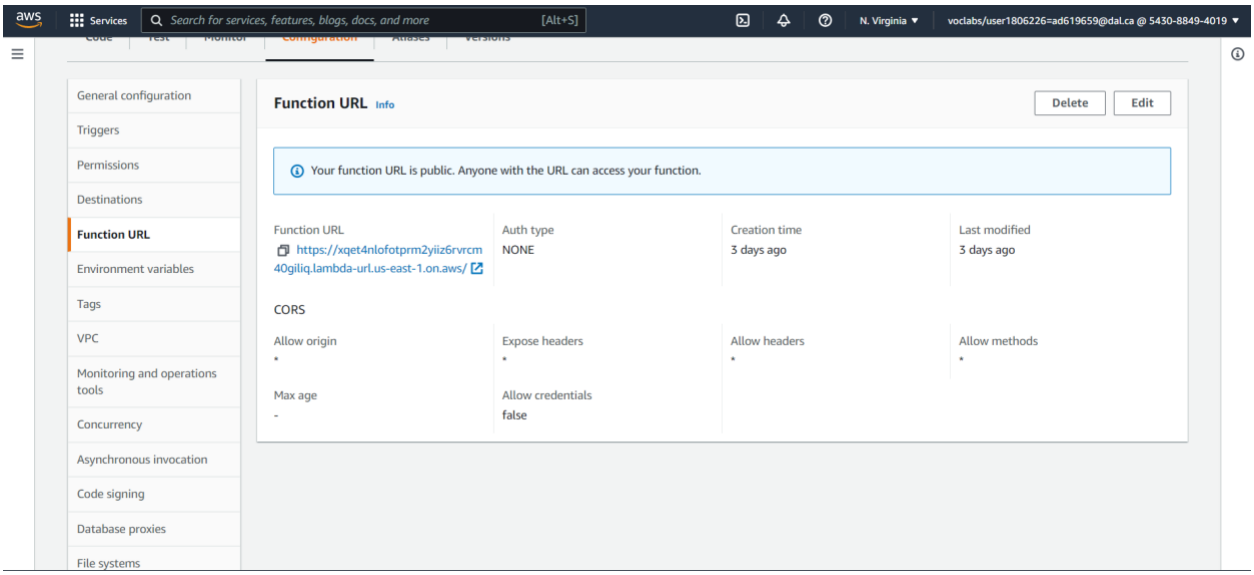


Figure 80 Booking Lambda Function Configuration

The lambda function has a lab role attached to it so that it has necessary permission to access other services on the AWS.

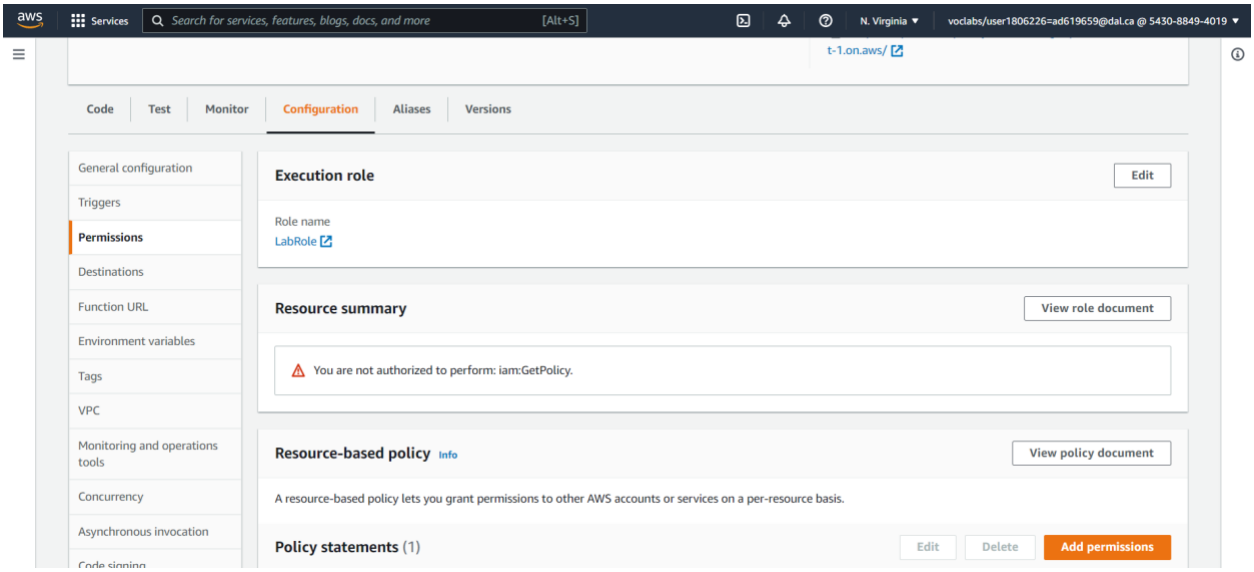


Figure 81 IAM Policy Attached to Lambda Function

Booking Lambda Function Code

We have used NodeJS/ExpressJS for the backend. We have used a serverless-express wrapper library for making it entirely serverless.

```
const serverlessExpress = require("@vendia/serverless-express");
const app = require("./app");
exports.handler = serverlessExpress({ app });
```

There are three routes for handling the booking service:

1. Booking
2. Rooms
3. Checkout

```
const express = require("express");
const bookings = require("./routes/bookings");
const rooms = require("./routes/rooms");
const checkout = require("./routes/checkout");
const cors = require("cors");
const app = express();
app.use(express.json());
app.use(cors());

// require("./startup/initializeDB")();

app.use("/bookings", bookings);
```

```
app.use("/rooms", rooms);
app.use("/checkout", checkout);

const port = process.env.PORT || 5000;

module.exports = app;
```

We have defined two models for interacting with the backend DynamoDB tables.

1. Booking Model

```
const dynamoose = require("dynamoose");
const Joi = require("joi");

const Booking = dynamoose.model(
  "Booking",
  new dynamoose.Schema({
    customerID: {
      type: String,
      required: true,
    },
    type: {
      type: String,
      required: true,
    },
    active: {
      type: Boolean,
      required: true,
    },
  })
);

module.exports.validate = (object) => {
  const schema = Joi.object({
    customerID: Joi.string().required(),
    type: Joi.string().min(3).max(100).required(),
    active: Joi.boolean().required(),
  });

  return schema.validate(object);
};

module.exports.Booking = Booking;
```

2. Room Model

```
const dynamoose = require("dynamoose");
const Joi = require("joi");

const Room = dynamoose.model(
  "Room",
  new dynamoose.Schema({
    type: {
      type: String,
      required: true,
    },
    available: {
      type: Number,
      required: true,
    },
    price: {
      type: Number,
      required: true,
    },
    AirConditioning: {
      type: Boolean,
      required: true,
    },
    DailyHousekeeping: {
      type: Boolean,
      required: true,
    },
    TV: {
      type: Boolean,
      required: true,
    },
    image: {
      type: String,
      required: true,
    },
    DailyHousekeeping: {
      type: Boolean,
      required: true,
    },
    FreeWiFi: {
      type: Boolean,
      required: true,
```

```
  },
  Bathroom: {
   type: Boolean,
   required: true,
  },
  Intercom: {
   type: Boolean,
   required: true,
  },
 })
);

module.exports.validateRoom = (object) => {
 const schema = Joi.object({
  type: Joi.string().required(),
  available: Joi.number().required(),
  price: Joi.number().required(),
 });

 return schema.validate(object);
};

module.exports.Room = Room;
```

There are three routes for managing the overall booking.

1. Booking

   The booking route has two endpoints for interacting with the backend DynamoDB table: Booking

   a. GET /bookings/:id
      This endpoint fetches and returns the booking for a specific customer.

   b. POST /bookings/
      This endpoint is for adding new booking for a customer.

2. Room

   The room route has two endpoints for interacting with the backend DynamoDB table: Room

   a. GET /rooms/
      This endpoint fetches and returns all the available rooms.

   b. POST /rooms/

This endpoint is for adding new type of rooms.

3. Checkout

   The checkout route has an endpoint for interacting with the backend DynamoDB tables: Room, Booking

   a. POST /checkout/
      This endpoint increases the room availability by one. It also marks the active status of booking to false.

DynamoDB Tables

There are two tables in the DynamoDB [6] table: Booking and Room

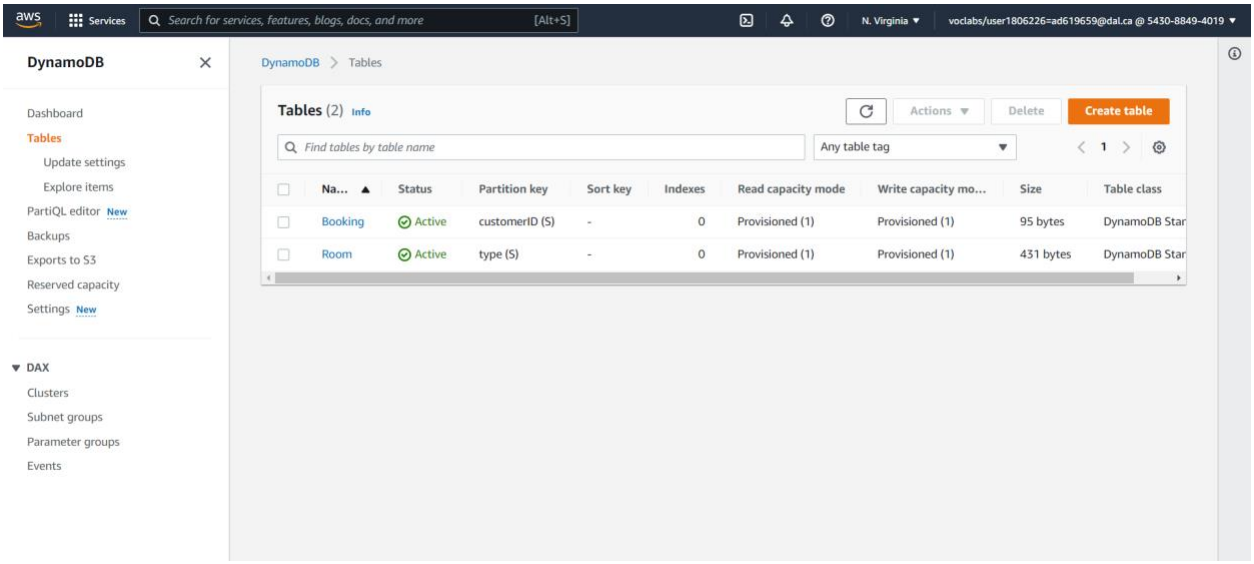


Figure 82 Booking and Room Table in DynamoDB

The booking table has three attributes: customerID, active and type. The customerID uniquely identifies the customer booking. The active attribute shows the status of booking. The type attribute shows the type of room booked by the customer.

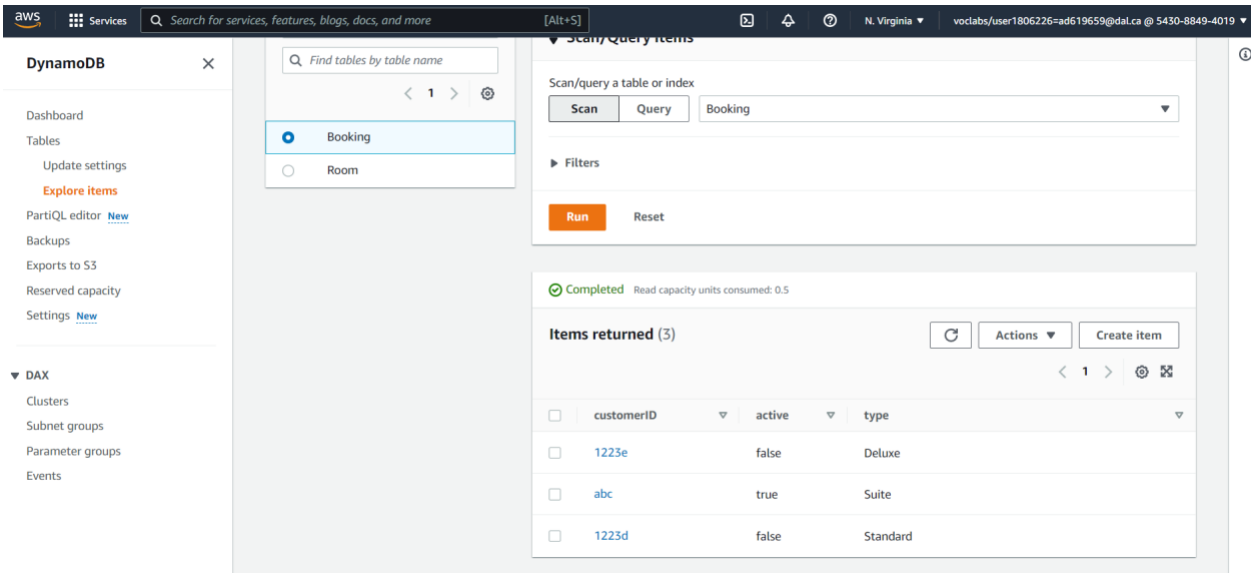


Figure 83 Booking Table Entries

The rooms table has the information about the available rooms.

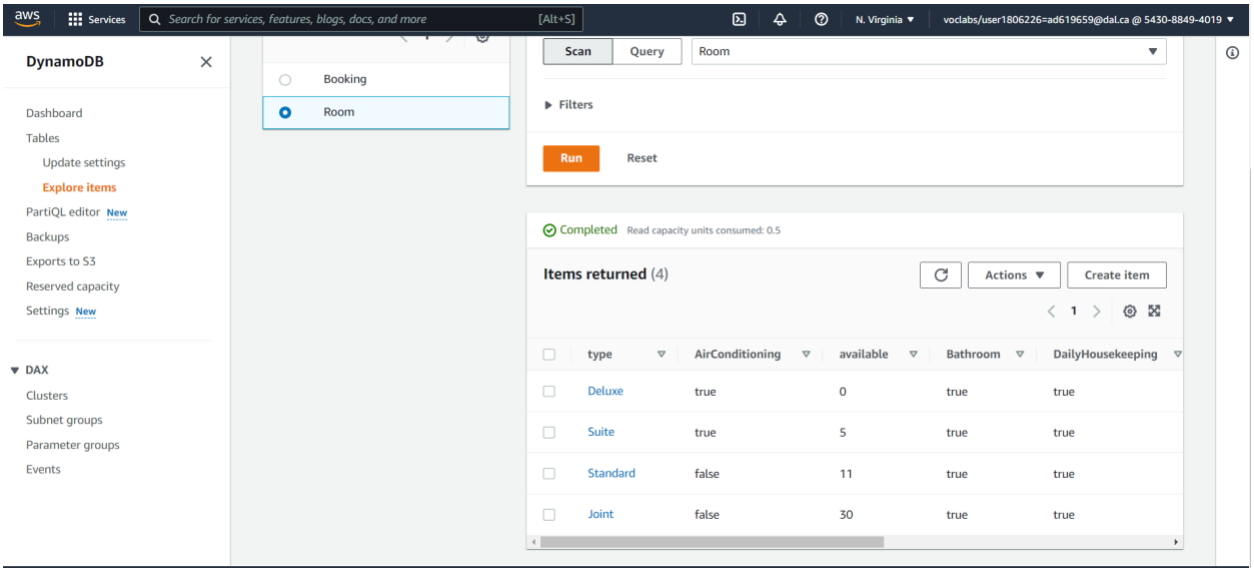


Figure 84 Room Table Entries

Testing

1. The availability of the rooms matches with what is displayed on the frontend page.

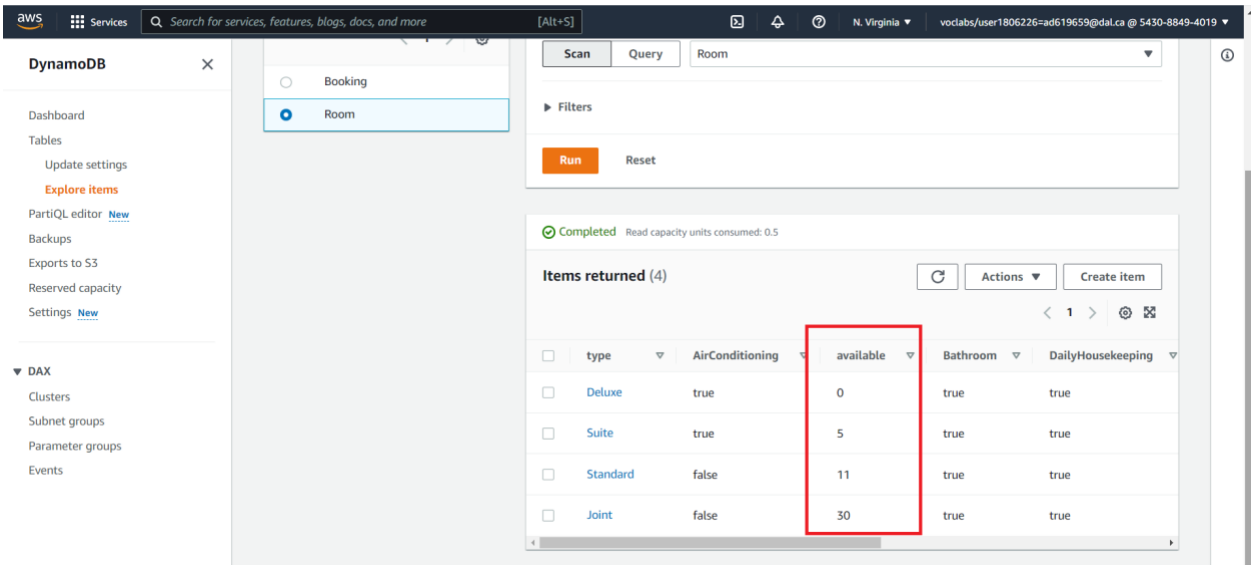


Figure 85 Room Availability

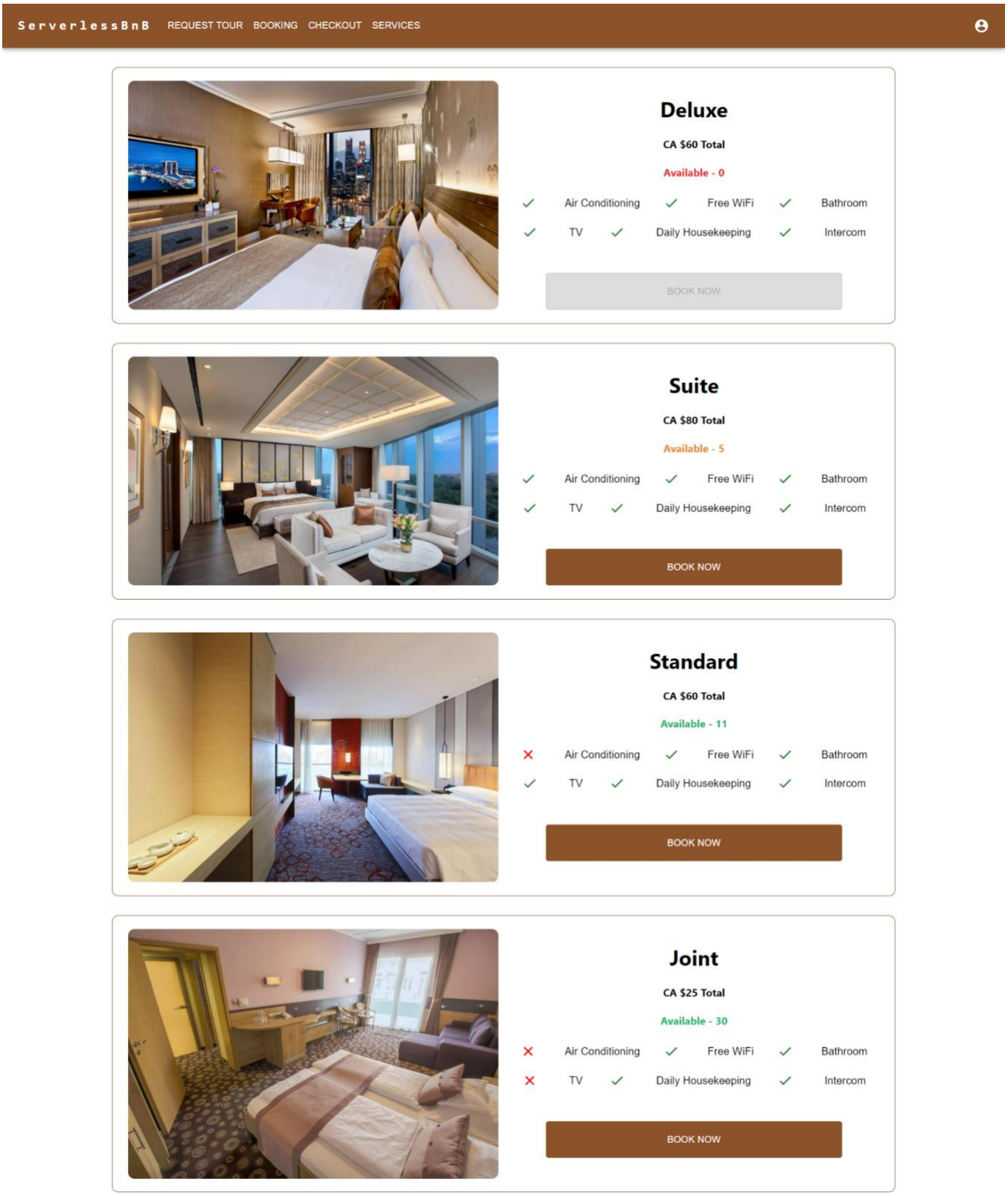


Figure 86 UI: Room Availability matching backend

2. After clicking on the “BOOK NOW” button, the availability decreases by one on the frontend.

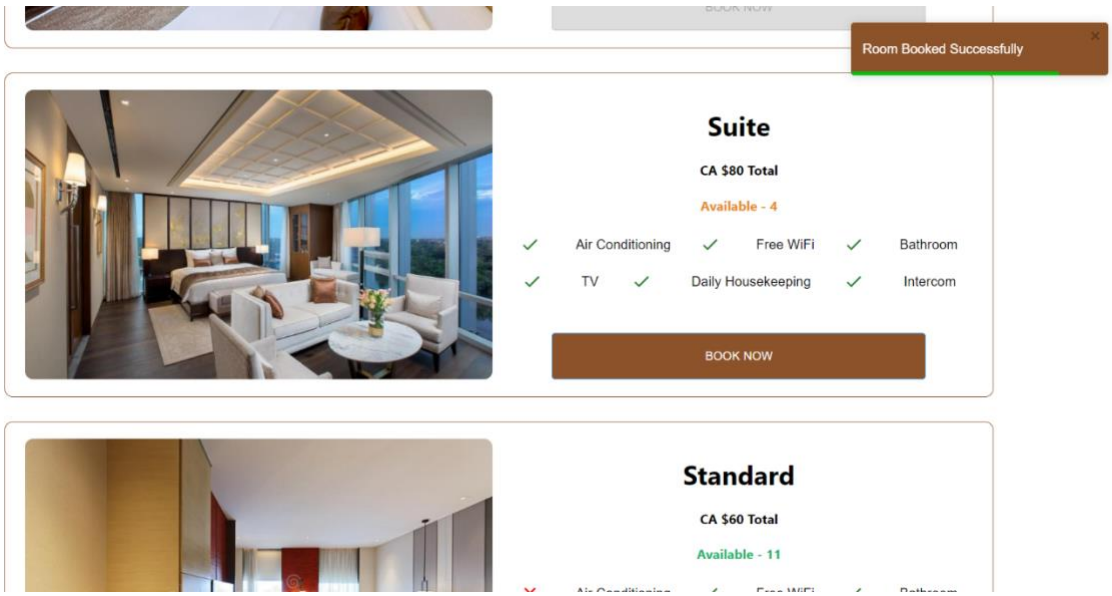


Figure 87 Room Booking

3. The lambda function triggers to handle the incoming request

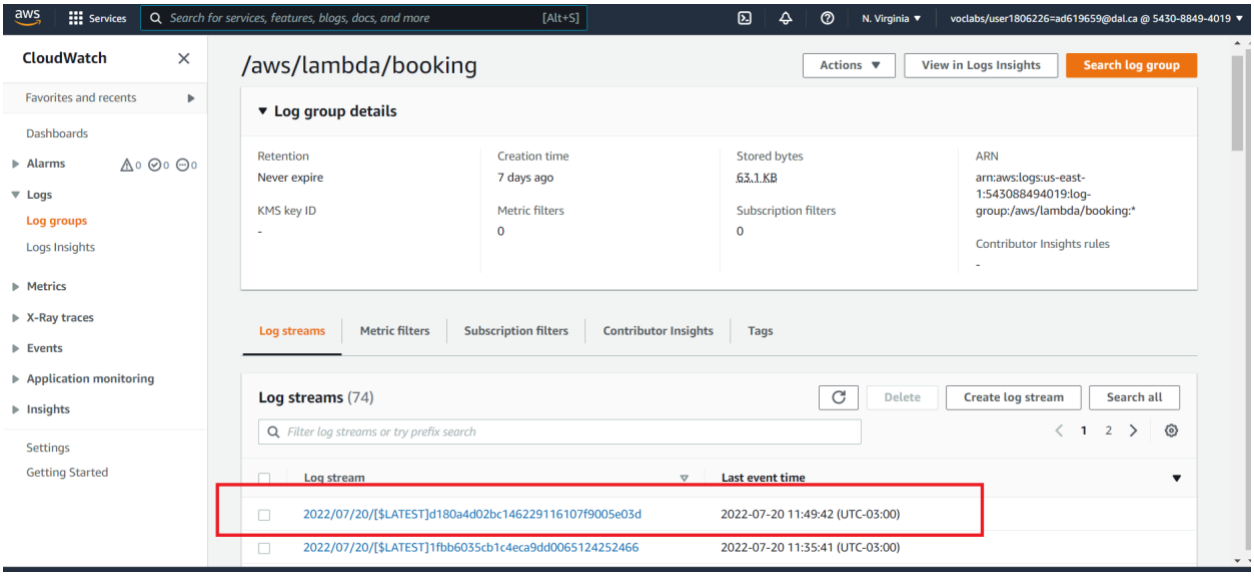


Figure 88 Lambda Triggered on booking

4. The availability also decreases by one in the DynamoDB table

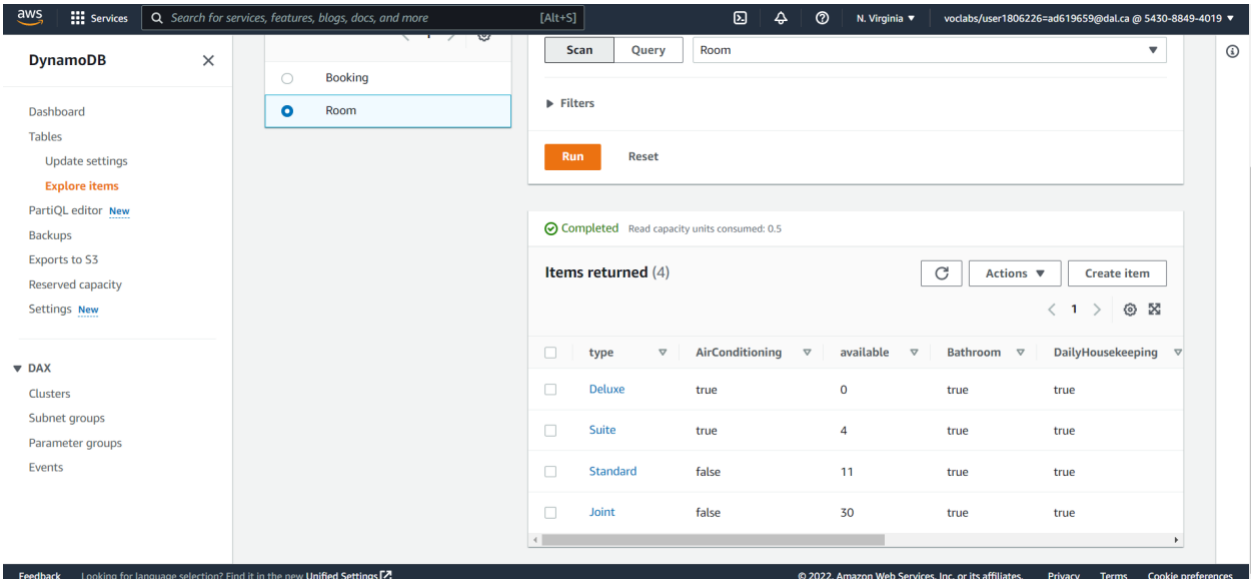


Figure 89 Room Availability Updated in Table

5. The booking table also gets updated for the user.

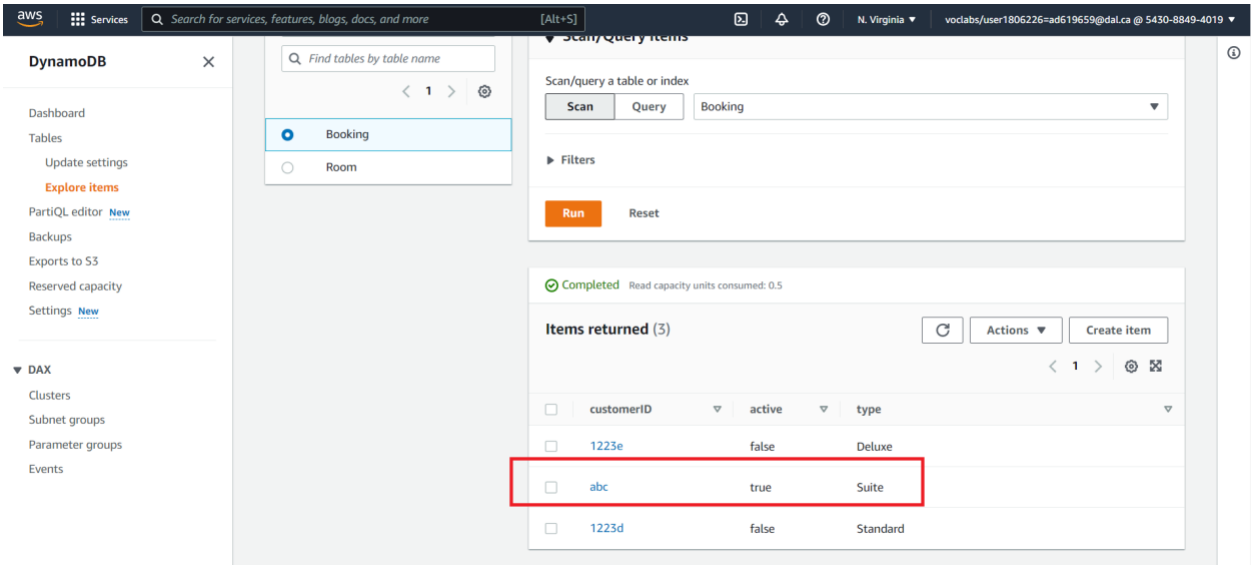


Figure 90 Booking Table Updated

Online Support - Chatbot

A user can utilize the chatbot on the website both as a guest user and after logging in. A guest user can only search for available rooms, while a logged-in user can book a room and order food using the chatbot.

Figure below shows the flow diagram for Lex [8] chatbot interactions.

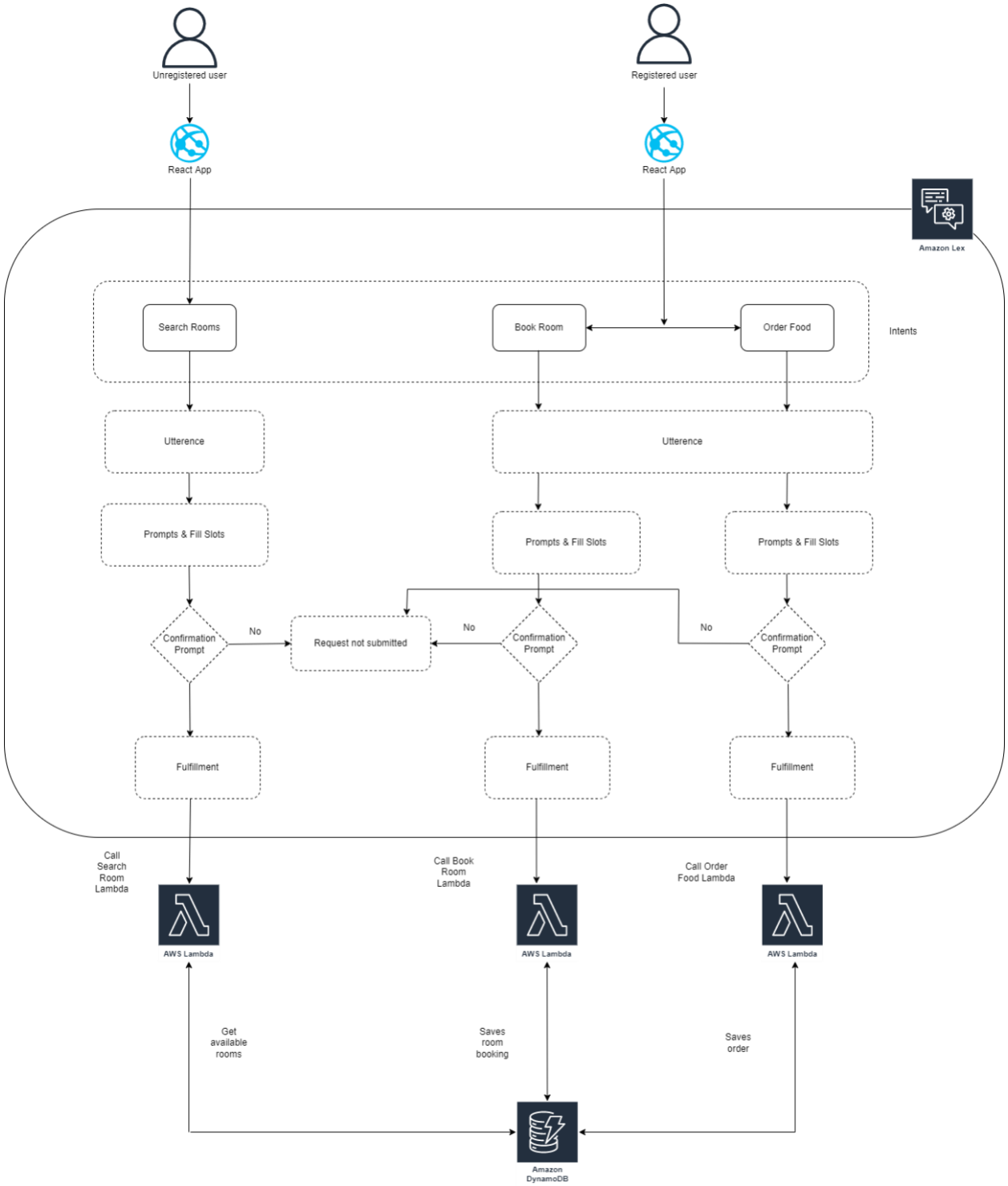


Figure 91 Online Support User Flow Diagram

Chatbot - Unregistered User Flow (Search Rooms)

1. When guest user accesses the website, they can access the chatbot by clicking on the chat icon at the bottom right corner.
2. After clicking on the chat icon, a chat window appears on the website.
3. A user can type in the chat window to search for the available rooms.
4. The user can type one of the following texts to get a valid response from the backend.

   I want to search the available rooms
   Search the available rooms
   I want to know if the room is available between ___ and ___ dates.
   Are there any deluxe rooms available?

5. Based on the text typed by the user, the chatbot will respond by asking a few more questions to get the entire information from the user.
6. The user is prompted with two questions.

   Which type of room? (Basic/Premium)
   What are the start date and end date of your stay?

7. Once the user answers all the questions, the system will have all the information to process this information.
8. The information entered by the user is then passed to a lambda function. The lambda function cleans and processes the information.
9. The lambda function [9] then calls the DynamoDB [6] to get the list of available rooms in that category.
10. The lambda function receives this information and sends the response back to the lex chatbot.
11. The chatbot passes this information to the front-end to fulfill the user's request of searching the available rooms.

Chatbot - Registered User Flow (Book a Room)

1. When registered user accesses the website, they can access the chatbot by clicking on the chat icon at the bottom right corner.
2. After clicking on the chat icon, a chat window appears on the website.
3. A user can type in the chat window to book a room.
4. The user can type one of the following texts to get a valid response from the backend.

   I want to book a room
   Book a room

I want to book a basic room.
I want to book a deluxe room from ____ to _____.

5. Based on the text typed by the user, the chatbot will respond by asking a few more questions to get the entire information from the user.
6. The user is prompted with two questions.

Which type of room? (Basic/Premium)
What are the start date and end date of your stay?

7. Once the user answers all the questions, the system will have all the information to process this information.
8. The chatbot asks for confirmation.
9. The information entered by the user is then passed to a lambda function. The lambda function cleans and processes the information.
10. The lambda function then updates the DynamoDB to book the room.
11. The lambda function receives this information and sends the response back to the lex chatbot.
12. The chatbot passes this information to the front-end to fulfill the user's request of booking the room.

Chatbot - Registered User Flow (Order Food):

1. When registered user accesses the website, they can access the chatbot by clicking on the chat icon at the bottom right corner.
2. After clicking on the chat icon, a chat window appears on the website.
3. A user (who has already booked a room) can type in the chat window to order food.
4. The user can type one of the following texts to get a valid response from the backend.

I want to order breakfast
Please serve breakfast

5. The chatbot asks for confirmation.
6. The information entered by the user is then passed to a lambda function. The lambda function cleans and processes the information.
7. The lambda function then updates the DynamoDB to order food.
8. The lambda function receives this information and sends the response back to the lex chatbot.
9. The chatbot passes this information to the front-end to fulfill the user's request of ordering food.

High Level Implementation

On a high level, the chatbot feature utilizes four services: S3, Lex, Lambda Function, and DynamoDB table.

S3: The S3 bucket [4] hosts the static content of the website for displaying chat UI.

Lex: The Lex [8] bot uses Natural Language Processing for parsing the input data and then generating responses.

Lambda Function: The lambda function [9] has the logic for booking rooms, searching rooms and ordering food.

DynamoDB: The DynamoDB [6] table has Booking and Room tables.

Lex Configuration

We have created a chatbot called Lex for managing hotel bookings, and ordering food.

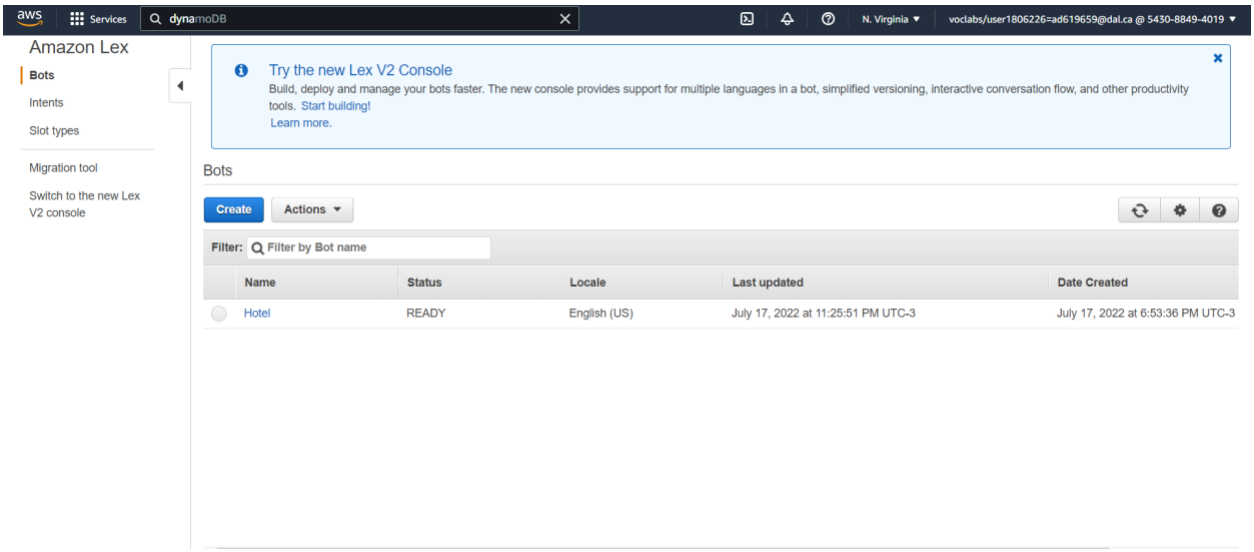


Figure 92 Hotel Chatbot

Creation of Custom Slot Type

We have created a custom slot type for defining type of rooms:

1. Deluxe
2. Suite
3. Joint
4. Standard

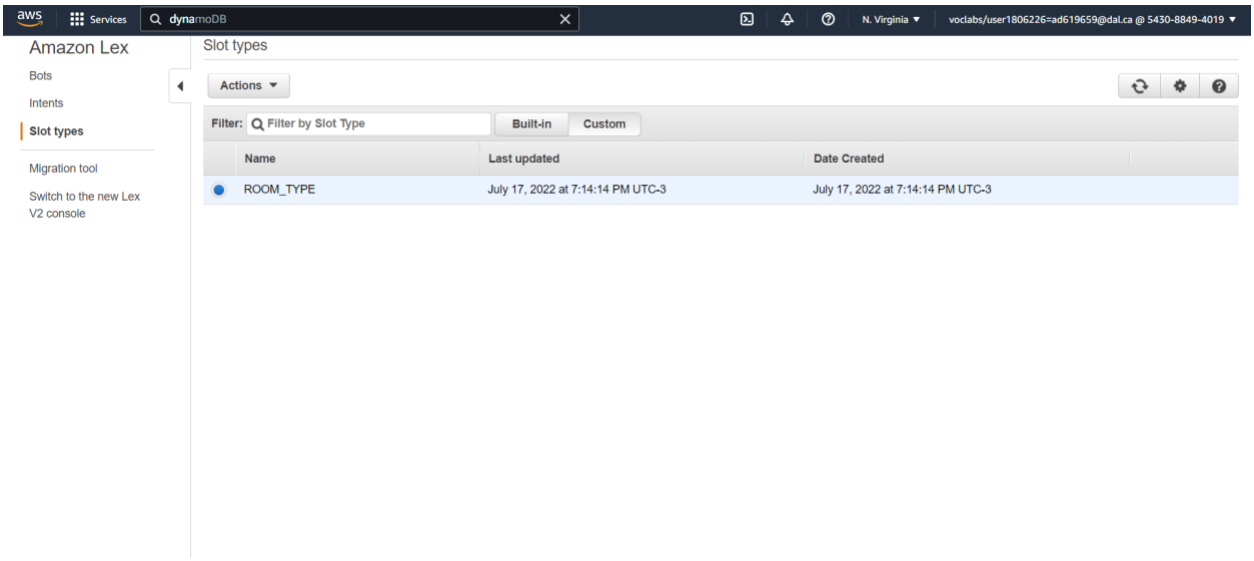


Figure 93 Custom Room Type Slot

Intents

We have created two intents for managing bookings

1. BookRoom

   The book room intent books a room

   Figure below shows the sample utterances for booking a room.

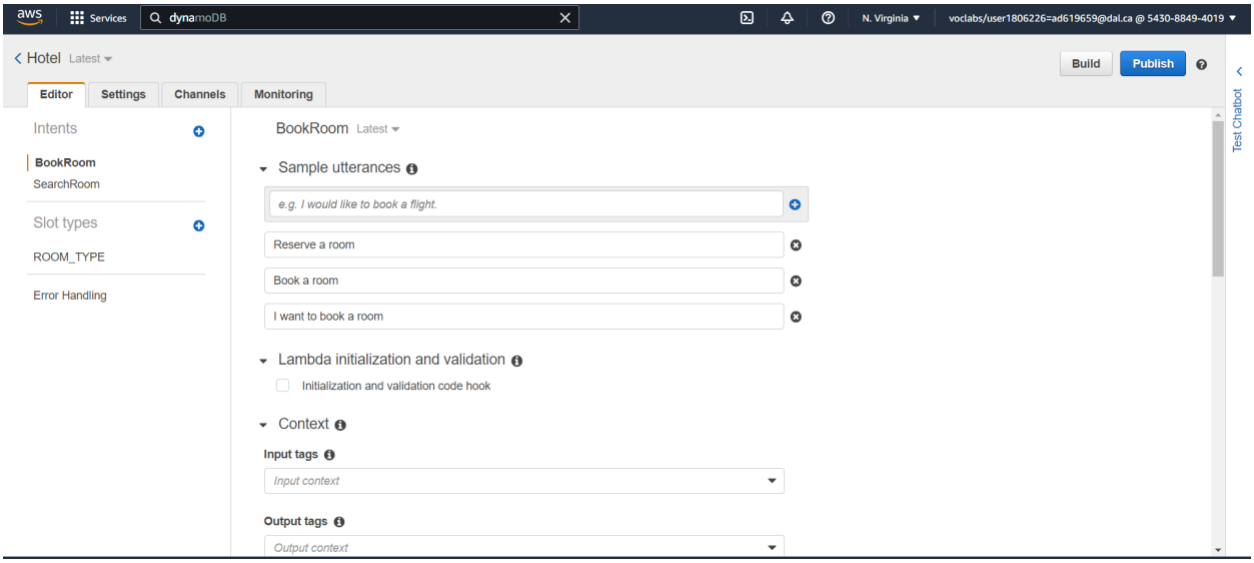


Figure 94 Book Room Utterances

As shown in Figure below, we have created a slot for asking the type of room.

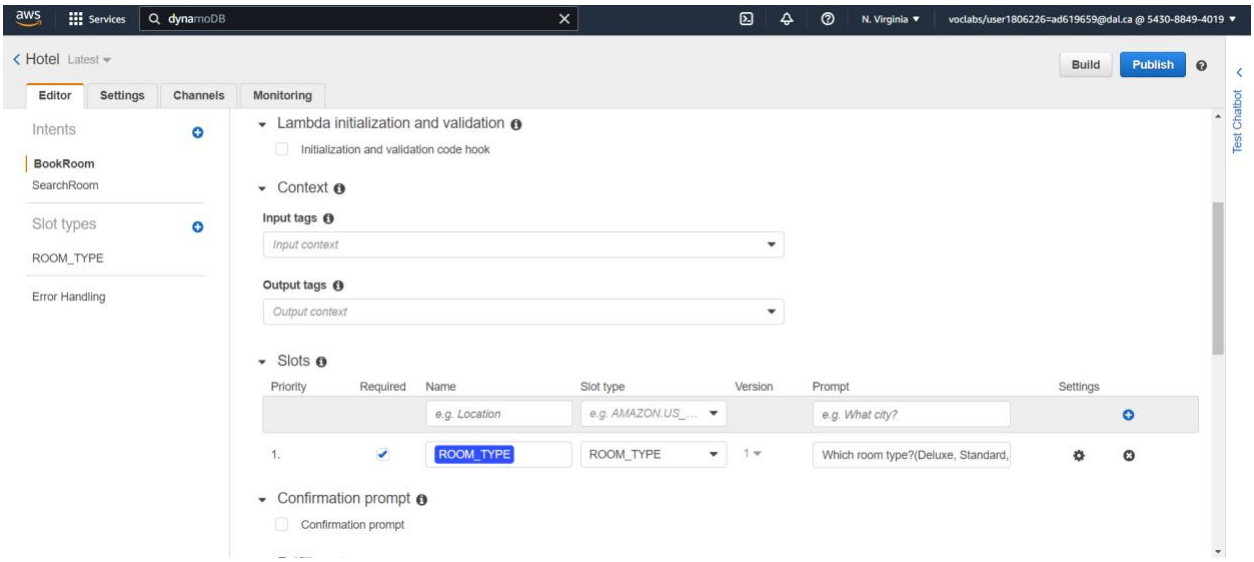


Figure 95 Book Room Slot

Attached a lambda function called “BookingRoom” for handling the fulfillment request.

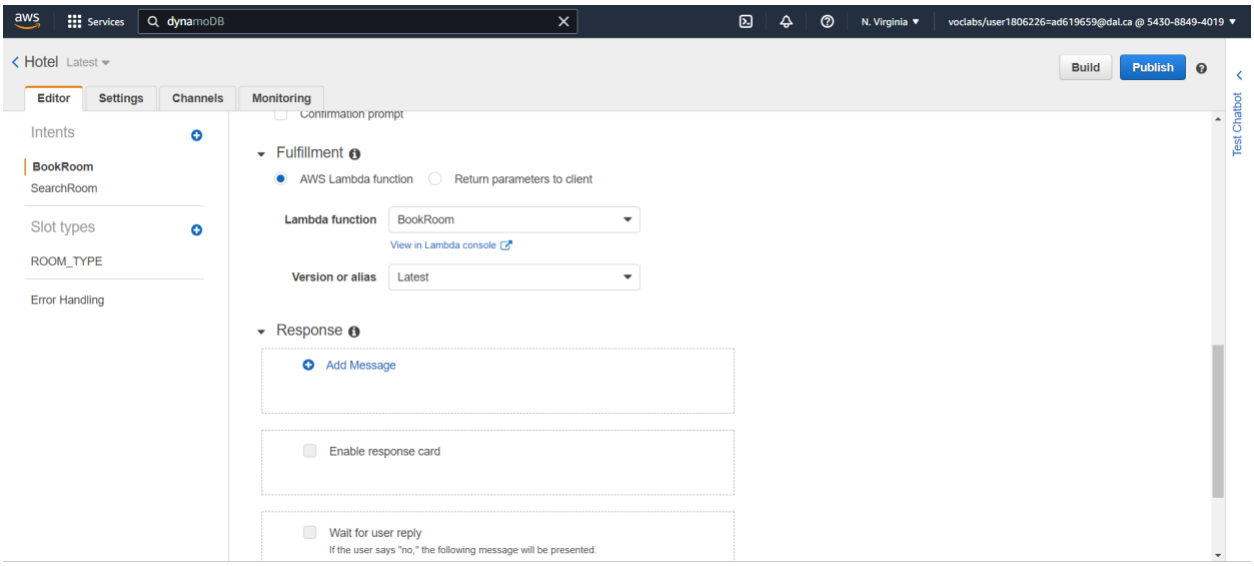


Figure 96 Booking Room Fulfillment

BookingRoom Lambda Function

The booking room lambda function has the code for communicating with the DynamoDB table.

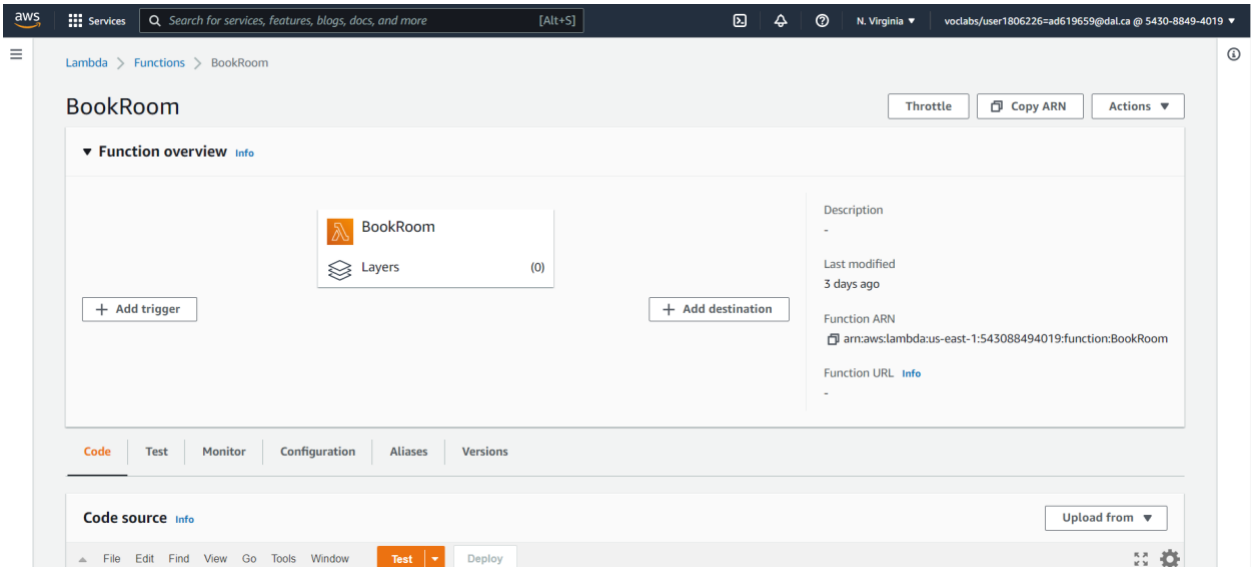


Figure 97 Book Room Lambda Function

Lambda Function Code

The lambda function uses the DynamoDB client for interacting with the booking and rooms table. Based on whether the room is available or not, it sends a response to the Lex chatbot. If the room is available, it creates a new item in the booking table. If the room is not available, then it sends a message that the room is not available.

```
            const AWS = require('aws-sdk');
const docClient = new AWS.DynamoDB.DocumentClient();


exports.handler = async (event) => {
   // TODO implement
   const roomType = event['inputTranscript'];
   const params = {
    TableName : 'Room',
    Key: {
      type: roomType
    }
   };


   try {
      const data = await docClient.get(params).promise();
      const availability = data['Item']['available'];

      if (availability === 0) {
         return {dialogAction: {
```

```
            type: "Close",
            fulfillmentState: "Failed",
            message: {
               contentType: "PlainText",
               content: "Room is not available"
            }
         }};
    }

    // Decrease room availability by 1
    const putParams ={
     TableName : 'Room',
     Item: {
       ...data['Item'],
       available: availability - 1
     }
    };
    await docClient.put(putParams).promise();

    return {
       dialogAction: {
          type: "Close",
          fulfillmentState: "Fulfilled",
          message: {
             contentType: "PlainText",
             content:  roomType + " room booked"
          }
       }};
  } catch (err) {
    return err;
  }
};
```

2. Search Room

   The search room intent searches for the given type of room and returns a response.

   Figure below shows sample utterances of the “SearchRoom” intent.

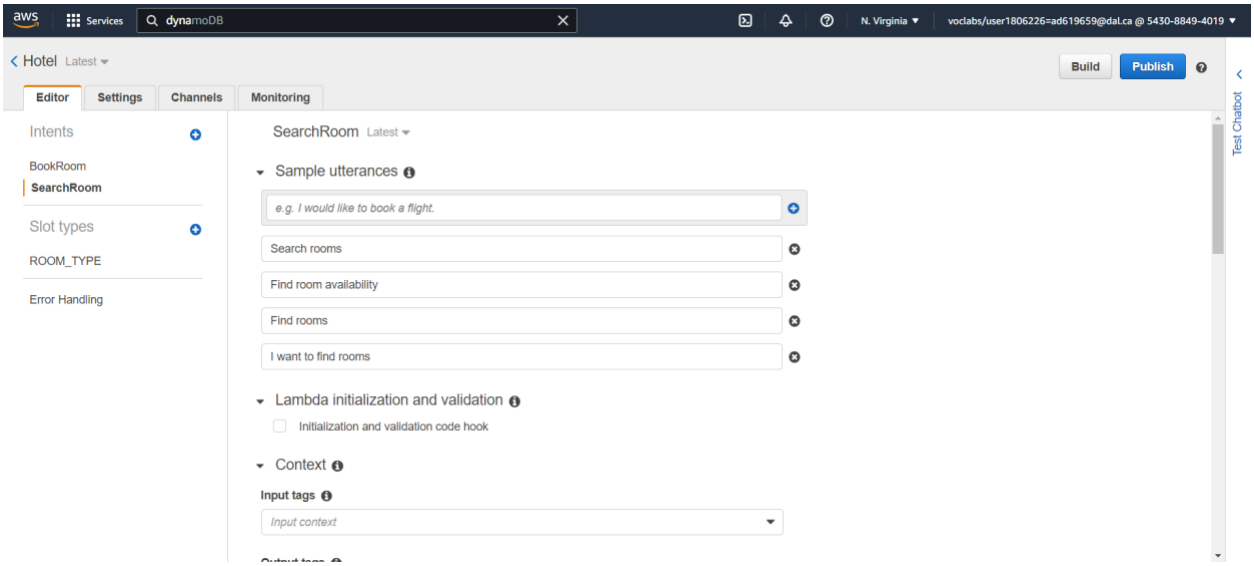


Figure 98 Search Room Utterances

As shown in Figure, we have created a slot for asking the type of room.

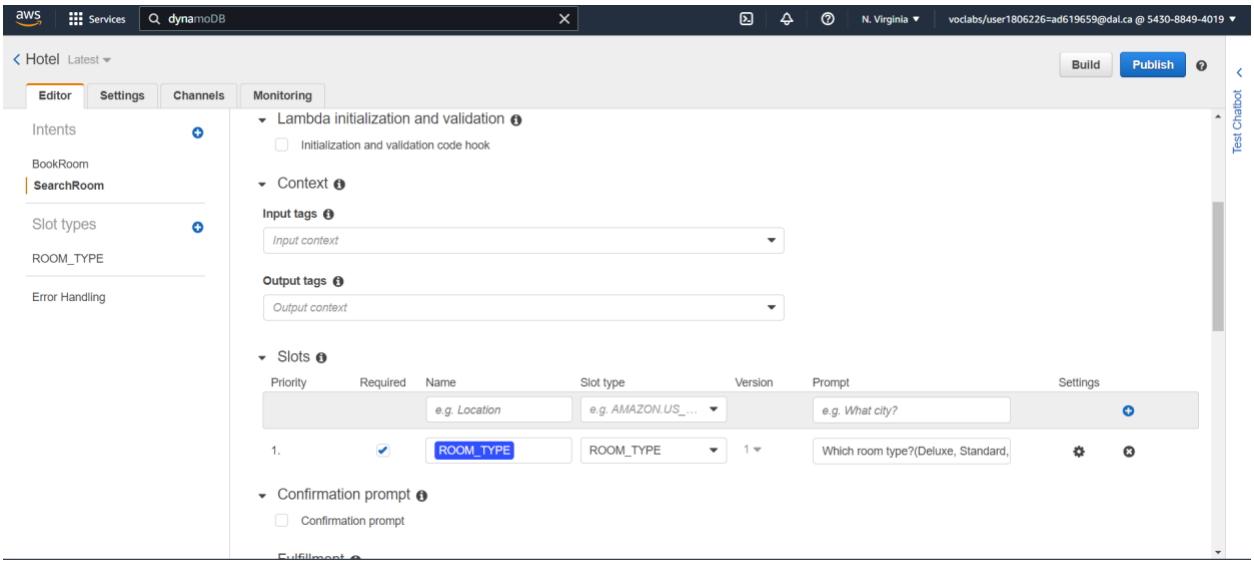


Figure 99 Search Room Slot

Attached a lambda function called “SearchRooms” for handling the fulfillment request.

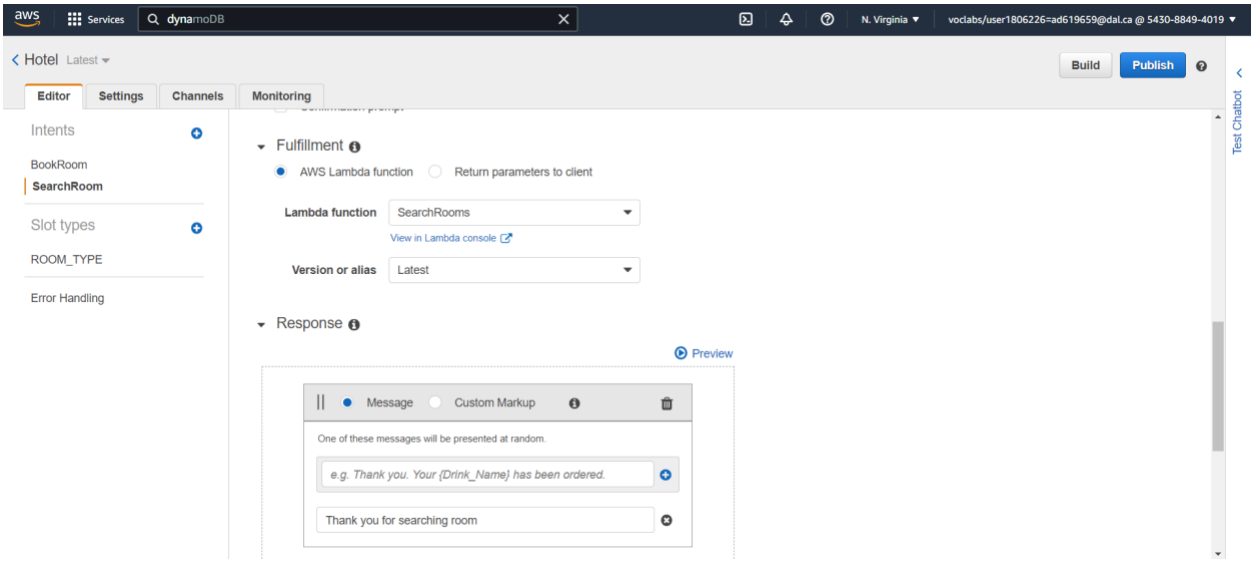


Figure 100 Search Room Fulfilment

SearchRooms Lambda Function

The search room lambda function has the code for communicating with the DynamoDB table.

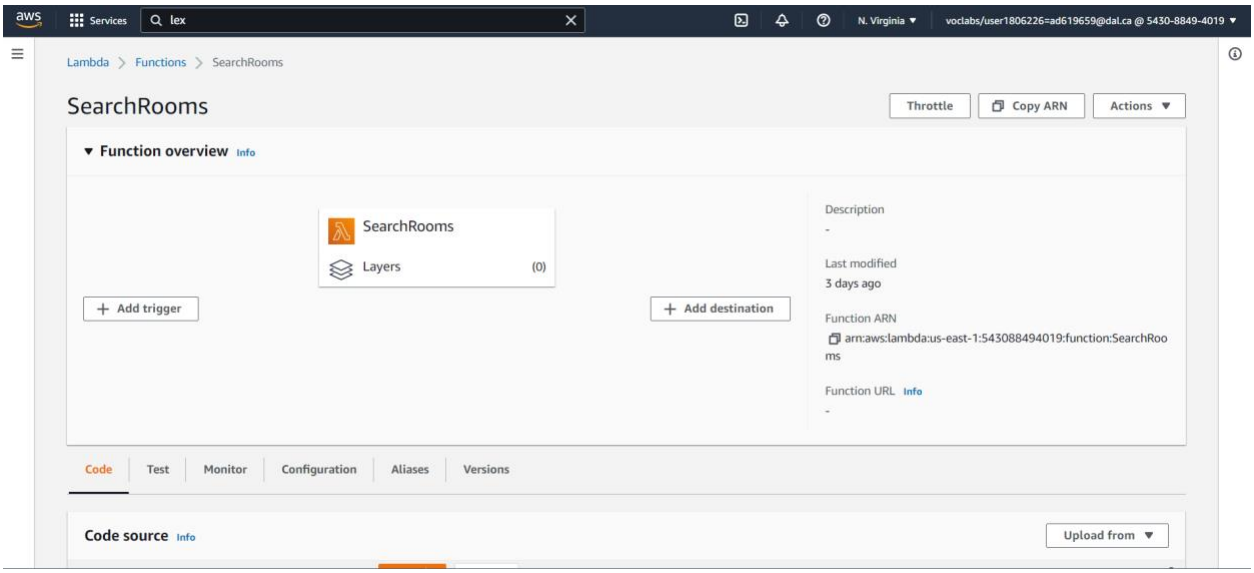


Figure 101 Search Room Lambda Function

Lambda Function Code

The lambda function uses the DynamoDB client for interacting with the rooms table. Based on whether the room is available or not, it sends a response to the Lex chatbot.

```
const AWS = require('aws-sdk');
const docClient = new AWS.DynamoDB.DocumentClient();
```

```
exports.handler = async (event) => {
   // TODO implement
   const roomType = event['inputTranscript'];
   const params = {
    TableName : 'Room',
    Key: {
     type: roomType
    }
   };


   try {
      const data = await docClient.get(params).promise();
      return {
         dialogAction: {
            type: "Close",
            fulfillmentState: "Fulfilled",
            message: {
               contentType: "PlainText",
               content: data["Item"]["available"] + " " + roomType + " rooms are available - Price: $" +
data["Item"]["price"]
            }
         }};
   } catch (err) {
      return err;
   }
};
```

Lex UI Integration

Created a lambda function called “LexMediator” for integrating with the frontend. The lambda function is exposed as a URL. It communicates with the Lex bot “Hotel” and return valid responses.

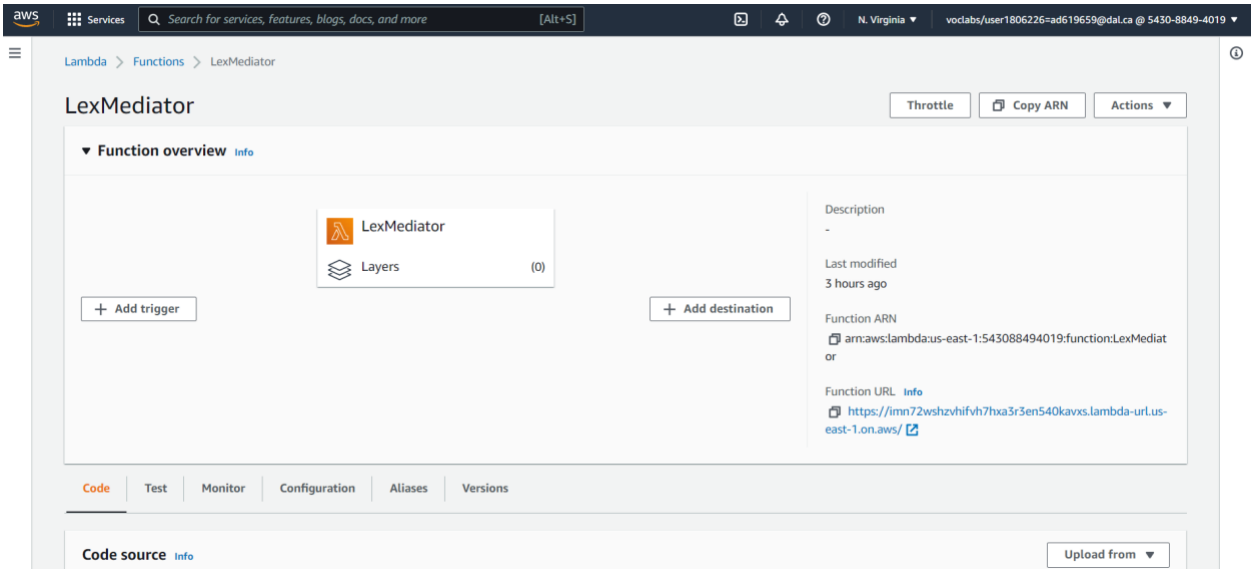


Figure 102 LexMediator Lambda Function

```
const AWS = require('aws-sdk');
var lexruntime = new AWS.LexRuntime();

exports.handler = async (event) => {
  try {
    const body = JSON.parse(event['body']);
    const message = body['message'];

    var params = {
     botAlias: 'Hotel',
     botName: 'Hotel',
     inputText: message,
     userId: 'AWSServiceRoleForLexBots',
    };

    let res = await lexruntime.postText(params).promise();
    return res;
  } catch (err) {
    return {
      statusCode: 500,
      body: "Something went wrong",
    };
  }
};
```

Testing

1. Figure below shows initial state of the DynamoDB table

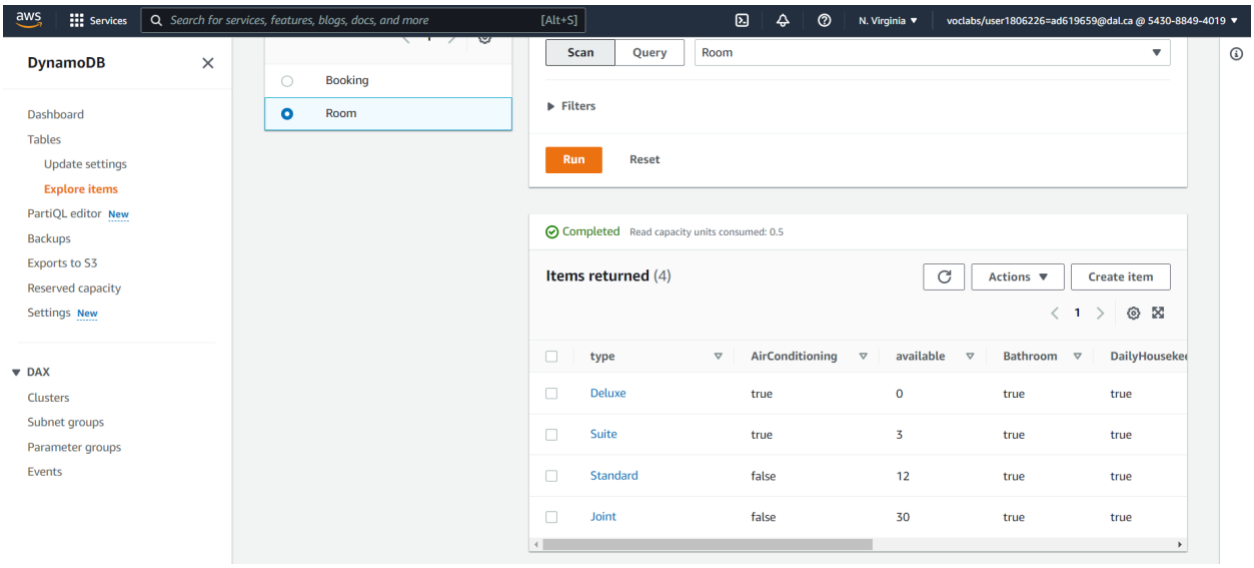


Figure 103 DynamoDB Initial State

2. Booking a room type which is not available

   The chat bot says that the room is not available.

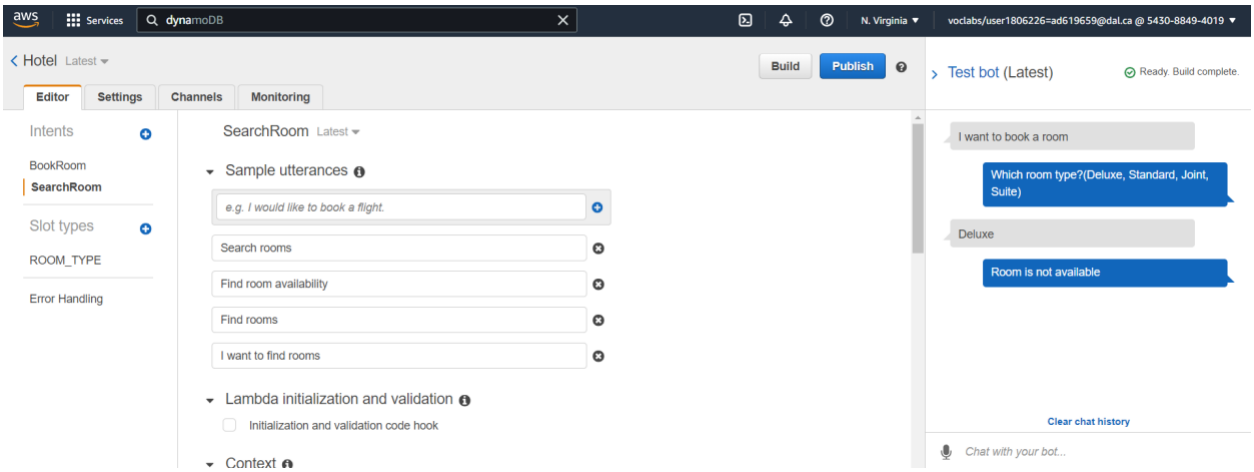


Figure 104 Testing Chatbot: Room not available

3. Booking a room type which is available

   The chat bot sends the request to the lambda function. The lambda function queries the DynamoDB table to check if the room is available. When the room is available, it says that the room is booked successfully.

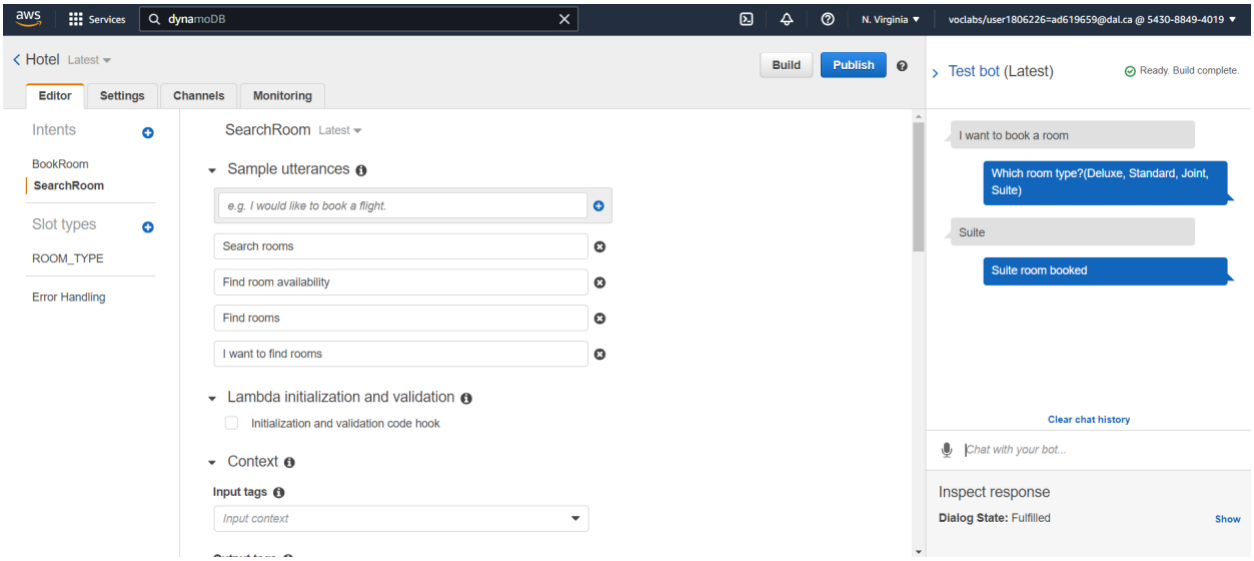


Figure 105 Testing Chatbot: Booking Room

The Figure below shows the updated DynamoDB table.

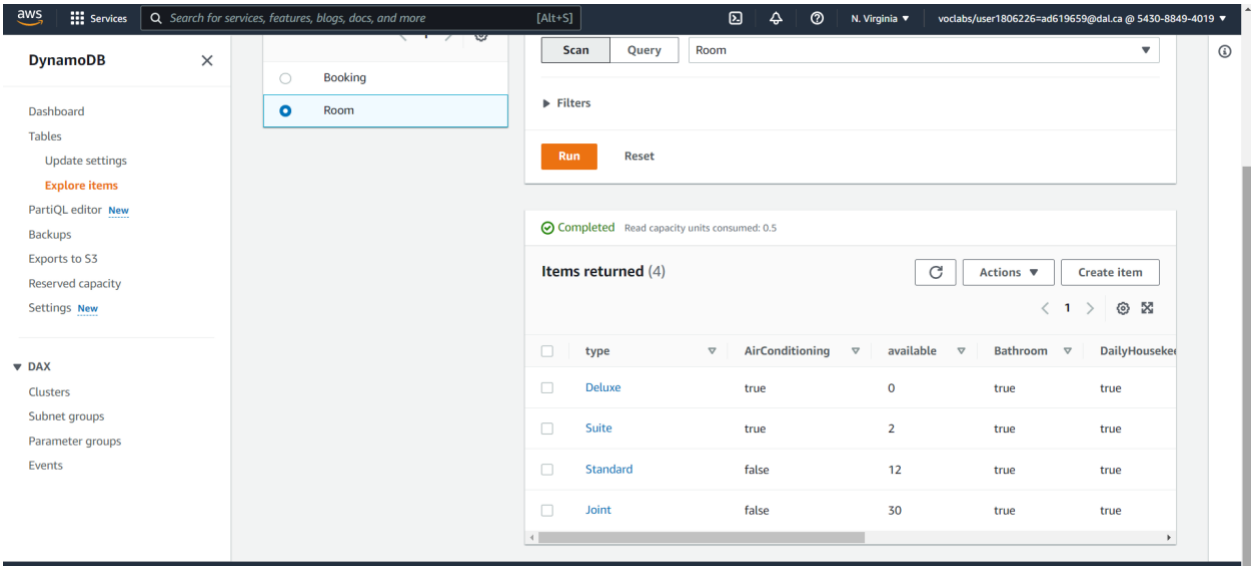


Figure 106 Updated DynamoDB Table

4. Searching rooms

The Lex chatbot triggers the lambda function. The lambda function scans the rooms table in the DynamoDB and return the response to the chatbot.

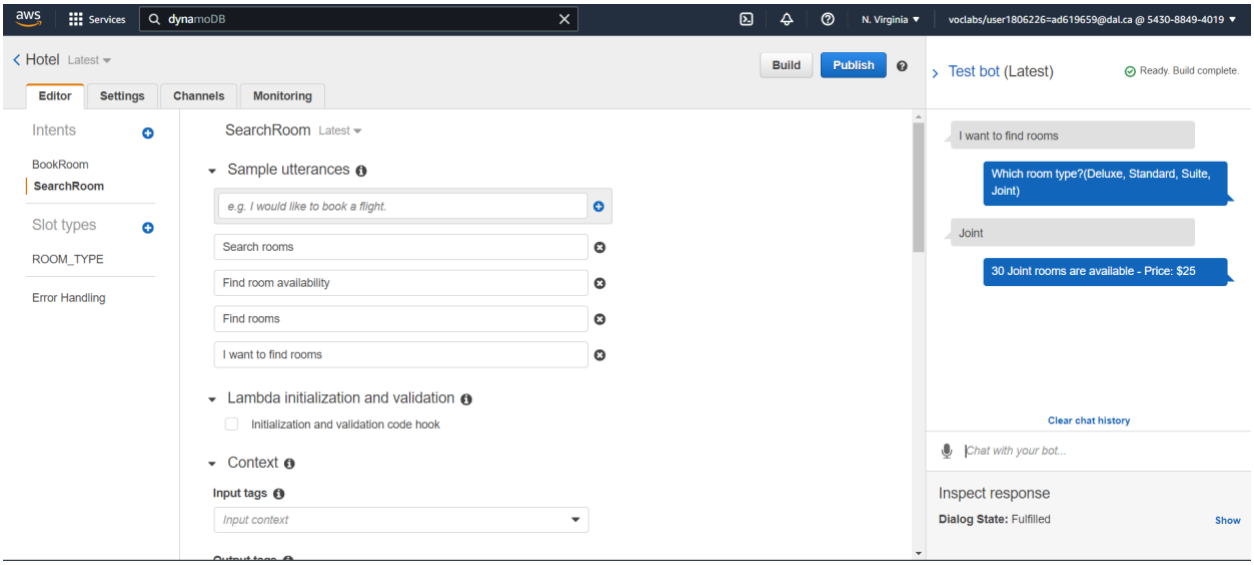


Figure 107 Testing Chatbot: Searching Rooms

5. Testing with real chat widget. The Figure below shows the current state of the DynamoDB table.

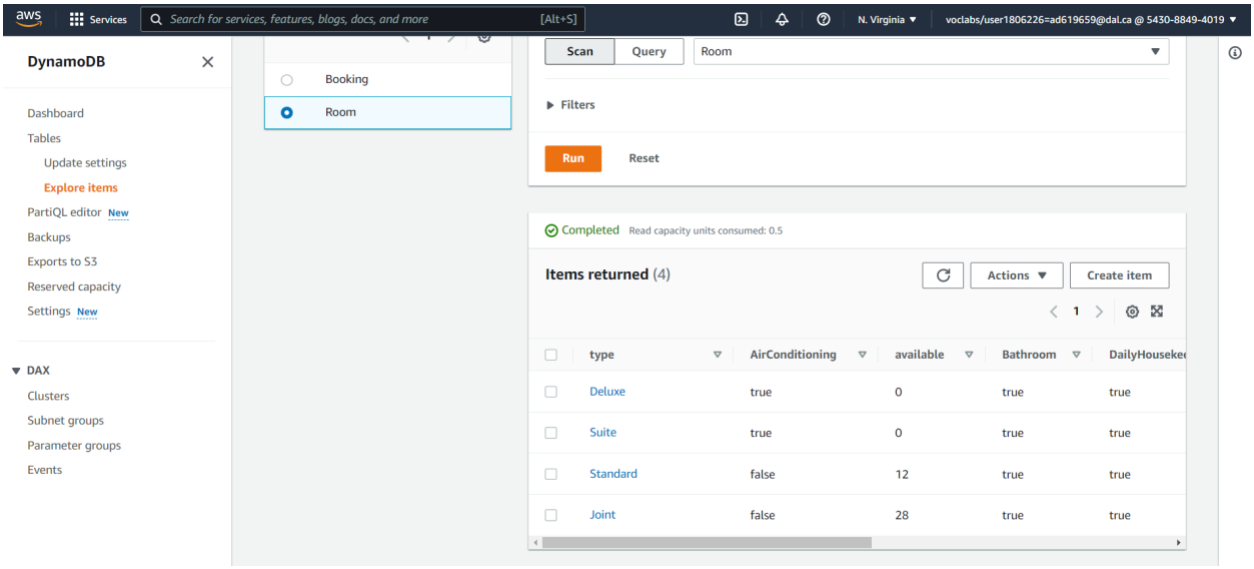


Figure 108 Testing Chatbot: DynamoDB Initial State

The Deluxe rooms are unavailable in the DynamoDB table. Hence, the chatbot shows 0 Deluxe rooms.

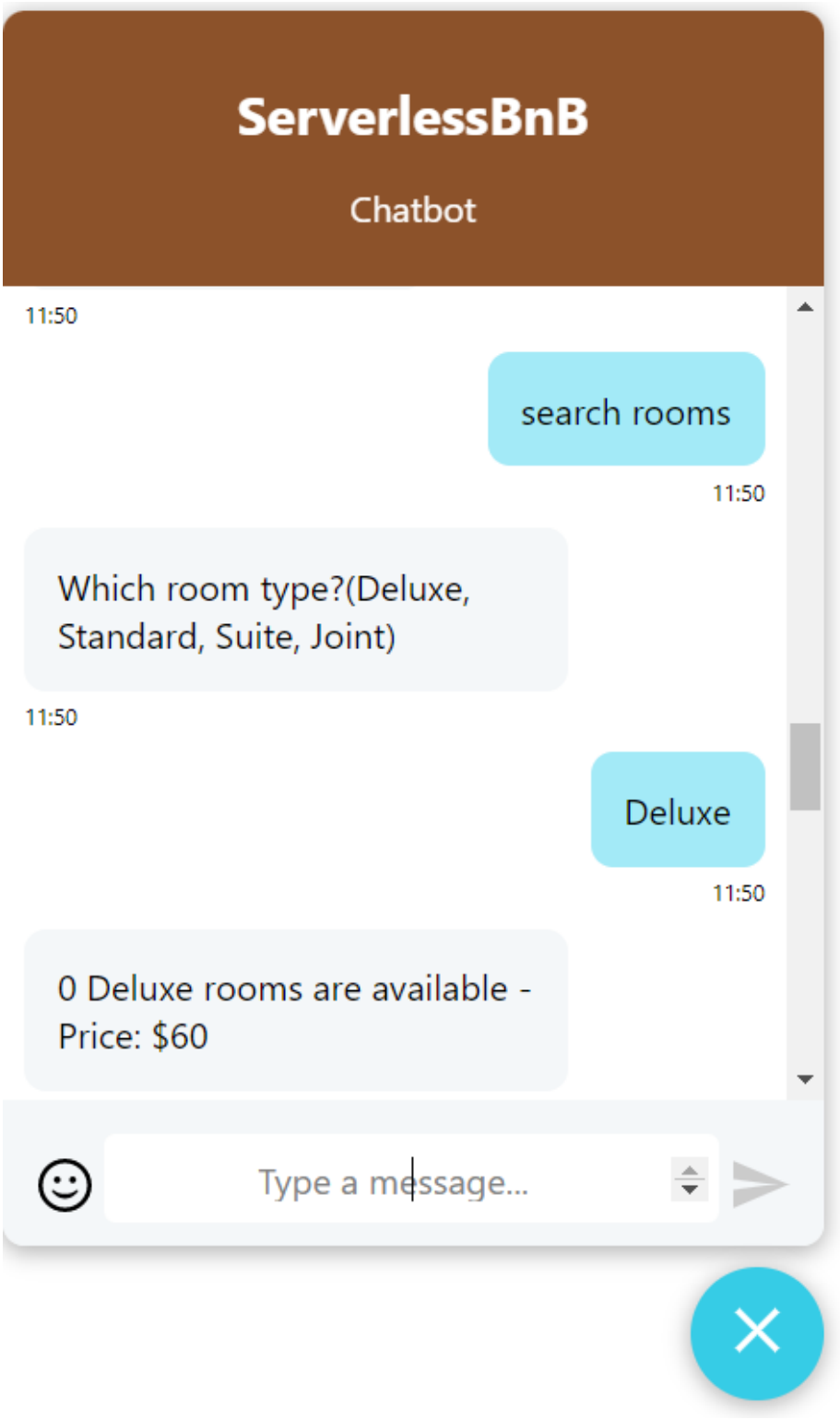


Figure 109 Chatbot UI :Search Unavailable Room

The Standard rooms are available in the DynamoDB table. Hence, the chatbot shows 12 Standard rooms.

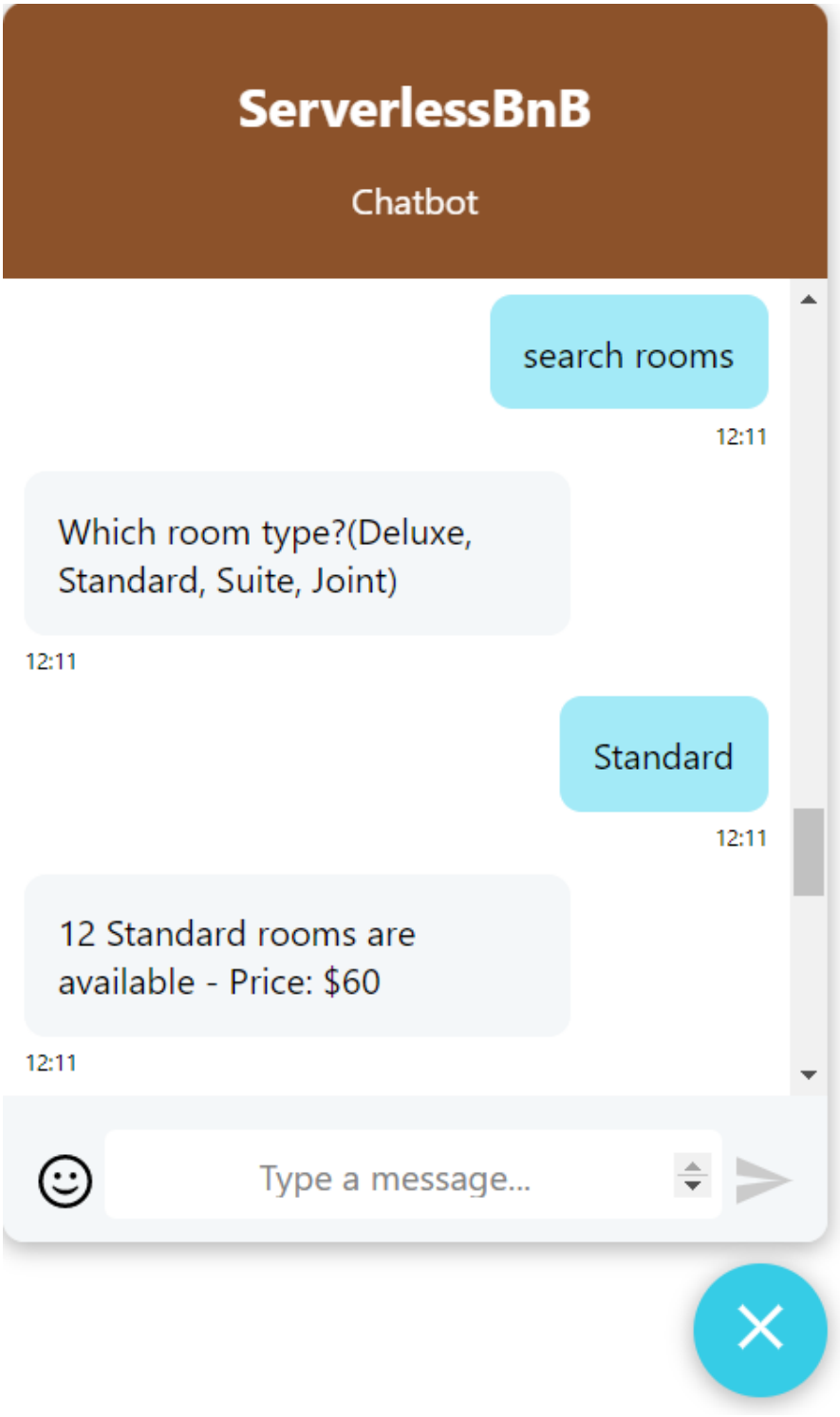


Figure 110 Chatbot UI: Search available Room

The Deluxe rooms are available in the DynamoDB table. Hence, the chatbot does not book the room.

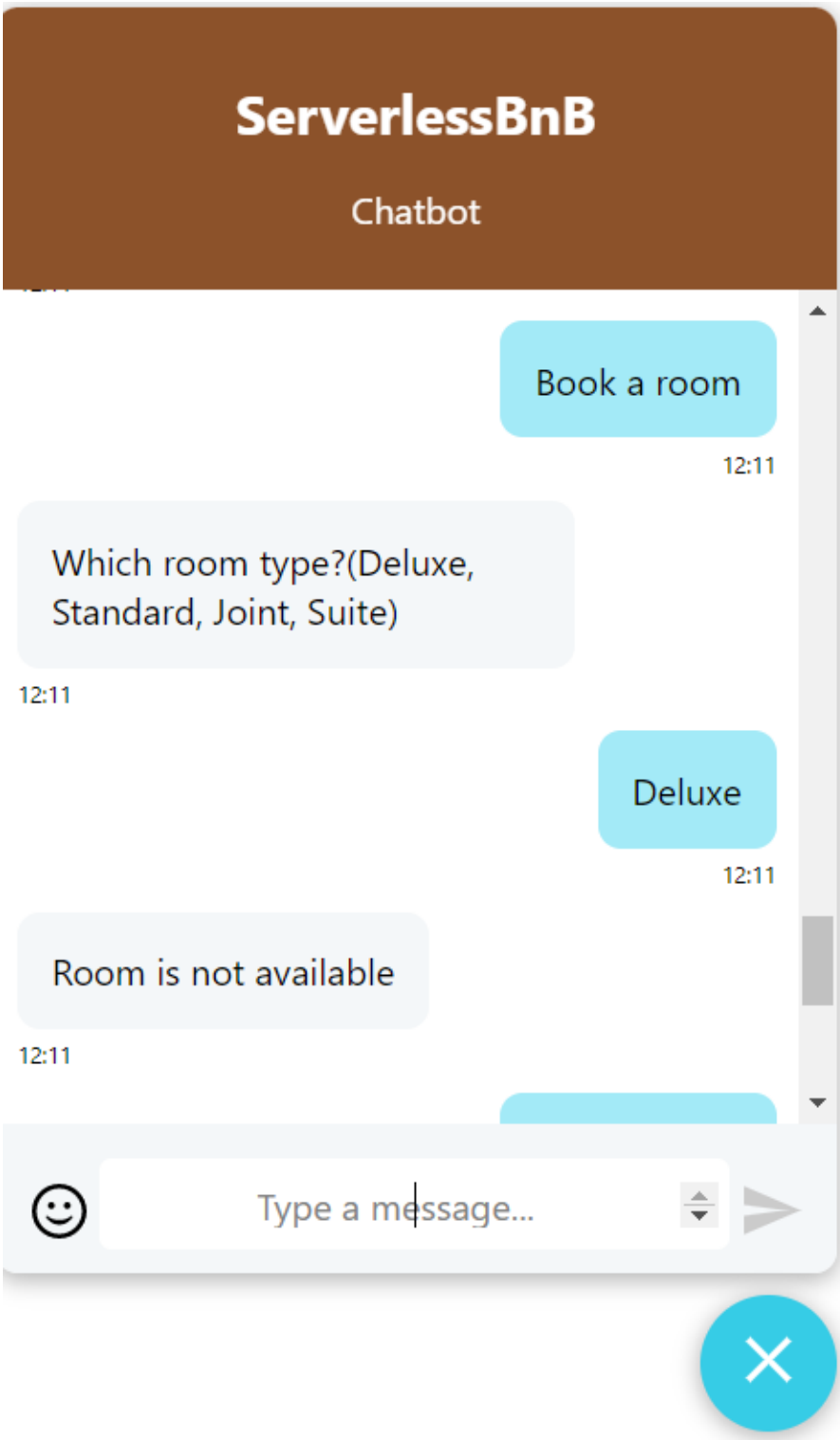


Figure 111 Chatbot UI : Book Unavailable Room

The Joint rooms are available in the DynamoDB table. Hence, the chatbot successfully books a room.

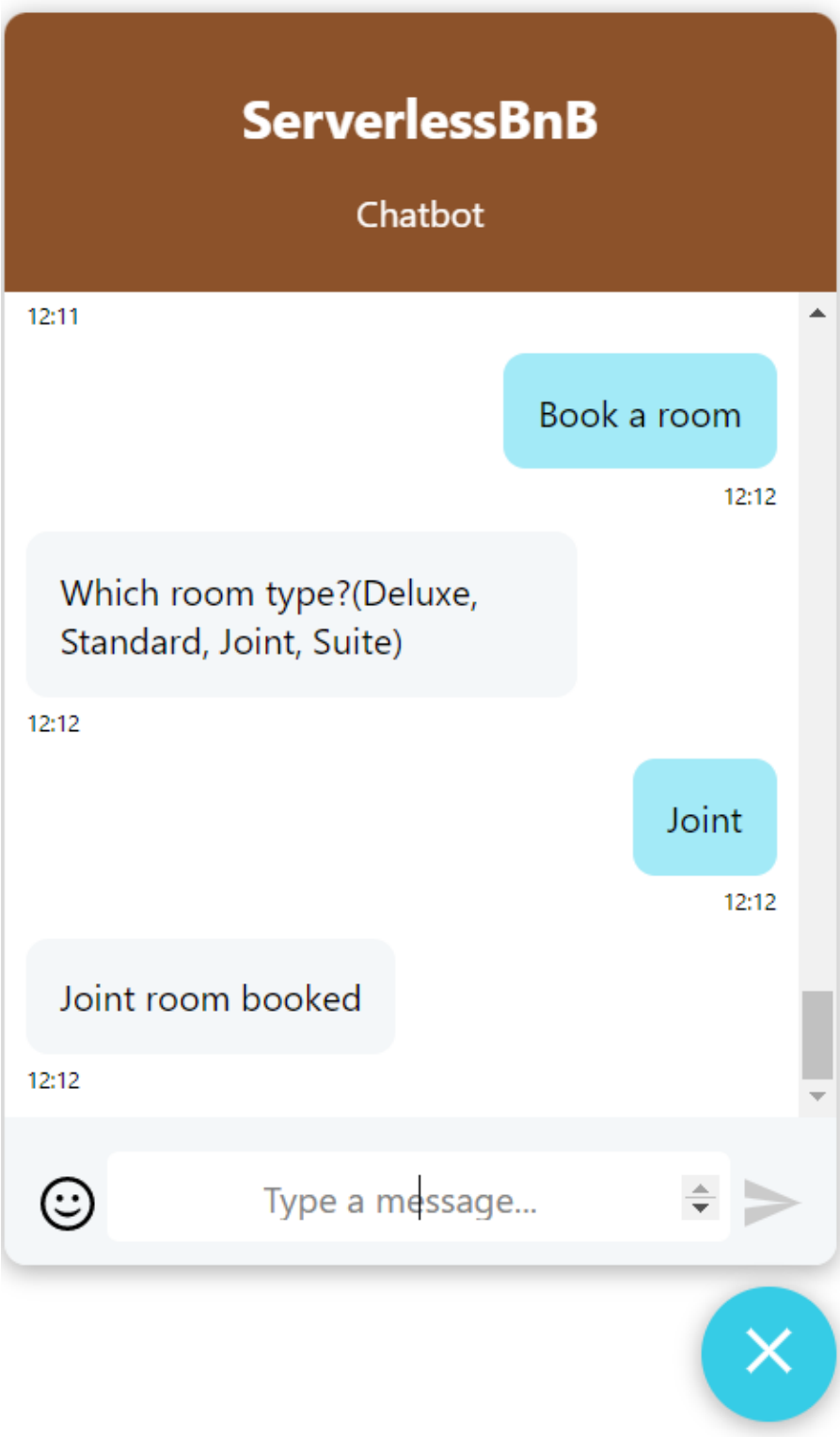


Figure 112 Chatbot UI :Book Available Room

Tour operator Service

The tour operator service is responsible for accepting tour requests from registered customers and suggest a tour package that best suits the requirements specified by the customers.

Implementation Details

The tour operator service makes use of the following cloud services from the Amazon Web Services (AWS) and Google Cloud Platform (GCP) cloud providers:

- **Cloud Functions**: Cloud functions [10] are used in the tour operator service to receive and process the tour request and suggest an ideal tour package based on the tour requirements of the customers. In addition, they send out emails with tour package details to the customers.
- **Pub/Sub [11]**: This service is used to carry the tour package details which are picked up by the cloud functions to send out emails.
- **API Gateway [20]**: This service is used to secure the cloud function endpoints by adding a security layer of its own. The cloud functions are accessed by the API Gateway's endpoint rather than the endpoint of the cloud functions themselves.
- **DynamoDB [6]**: This service is used to store the tour packages that are supported by ServerlessBnB. The cloud functions interact with this service to retrieve information about the tour packages.

Following is the sequence of actions that take place from the initial tour request to the tour recommendation made:

1. Authorized users would be able to request a tour package by specifying their required length of the tour from the user interface of the website.
2. The request is received by the **requestTour** cloud function, that is behind a GCP API Gateway [20], which processes the tour request and suggests an ideal tour package to the customers.
3. Synthetic Dataset of 2500 users is created which includes names, stay duration and tour packages named Alpha, Beta and Gamma. Alpha is a 1 day tour, Beta is a 3 day tour and Gamma is a 5 day tour.
4. A model named Train-TourPrediction is made using Vertex AI [12]. Auto ML [21] is used to train the model and give results on the basis of the dataset entered.
5. Then the cloud function is created to call the model and predict and test the new values being entered.
6. The **requestTour** cloud function sends an HTTP request to above cloud function that interacts with the machine learning model that predicts tour packages and assigns them a score based on the customers' tour length specification.
7. The **requestTour** cloud function receives the list of tour package recommendations by their id along with their assigned scores and picks the tour package id with the highest score.

8. The cloud function fetches the package details from the **tourPackages** table on DynamoDB [6] containing a list of tour packages, based on the tour package id with the highest score.
9. It constructs an email body with the package details and places the message on the **EmailService** topic of the Pub/Sub service.
10. The **emailSender** cloud function watches the **EmailService** topic and is invoked when a message is published to the topic.
11. The **emailSender** cloud function receives the message from the **EmailService** topic and sends out an email to the customer.
12. The tour package details are also shown to the customer on the website.

Flow Chart

Figure 113 shows the flow chart of the actions that place in the tour operator service

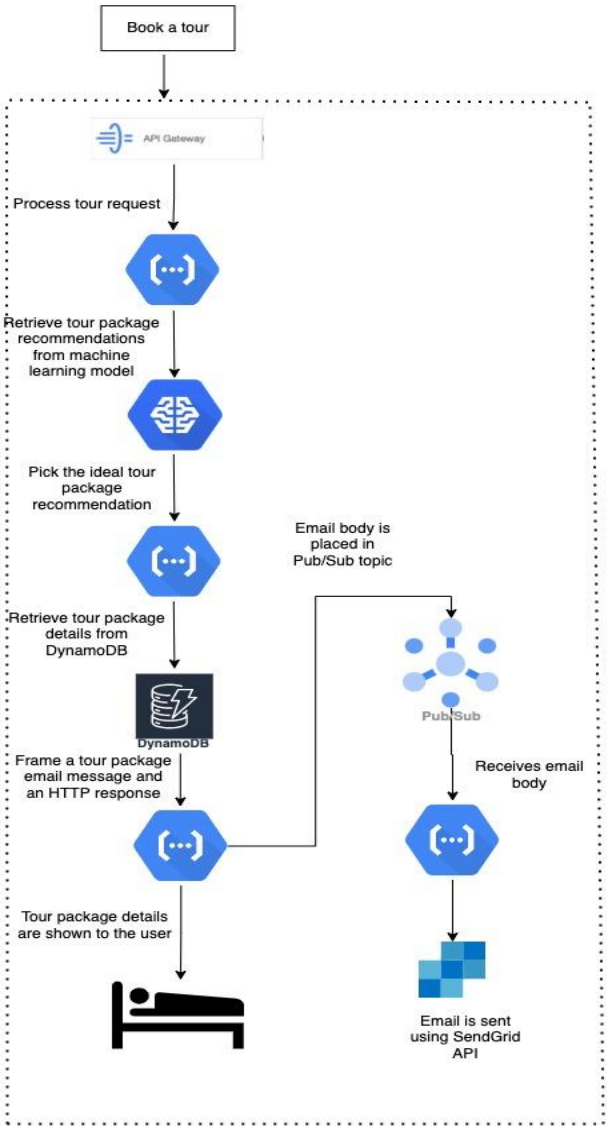


Figure 113: Tour operator service flow chart

Cloud services screenshots

Following are the screenshots of the cloud services that enable the operation of the tour operator service:

Figure 114 shows the EmailService Pub/Sub topic [11] that carries emails that are sent to the customers by the emailSender cloud function.

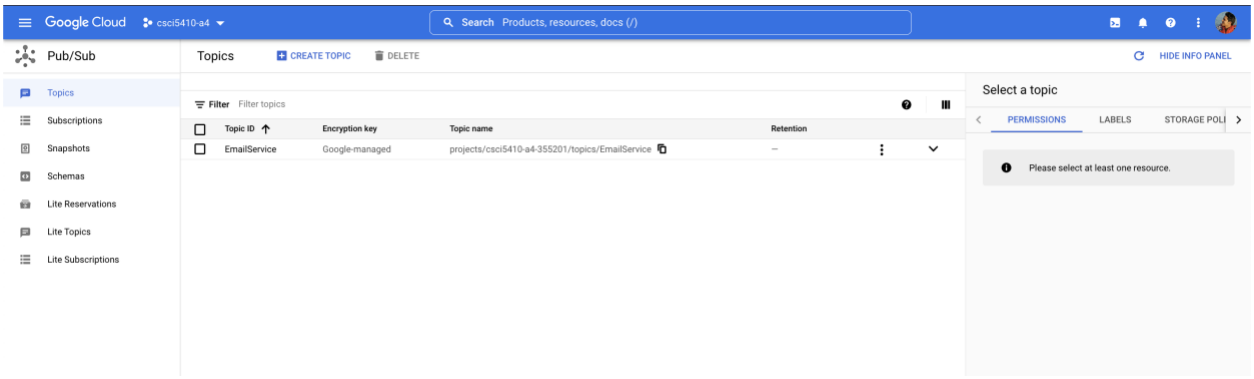


Figure 114: EmailService Pub/Sub topic that carries emails to be sent to the customers

Figure 115 shows the EmailService Pub/Sub topic subscribed by the emailSender cloud function.

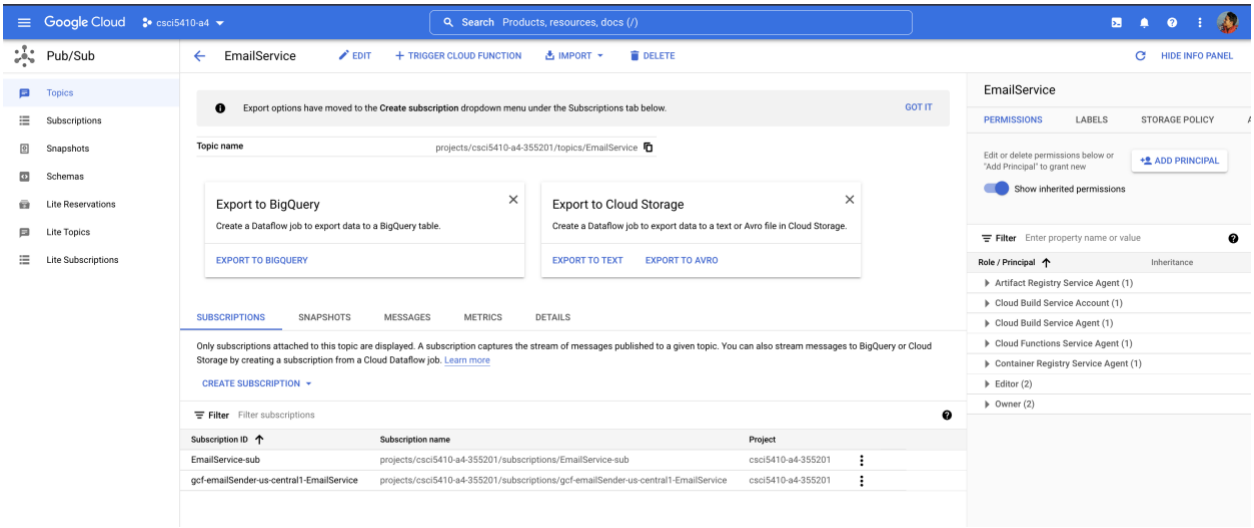


Figure 115: EmailService Pub/Sub topic details showing a subscription by emailSender Cloud Function

Figures 116 through 118 shows the API Gateway and its configuration that secures the cloud functions

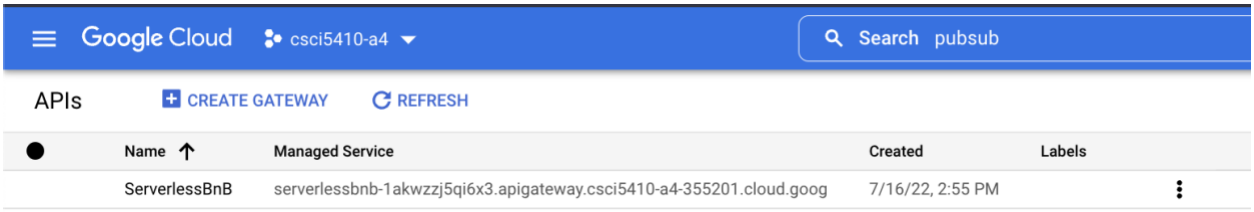


Figure 116: API created to host the API Gateway

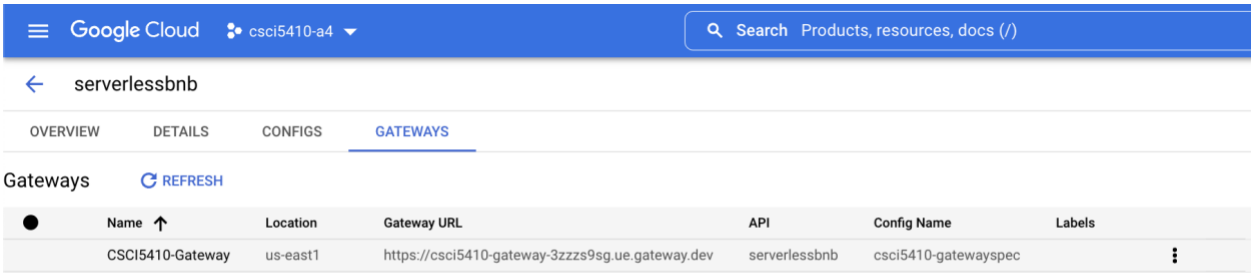


Figure 117: API Gateway that secures the cloud functions used in tour operator service

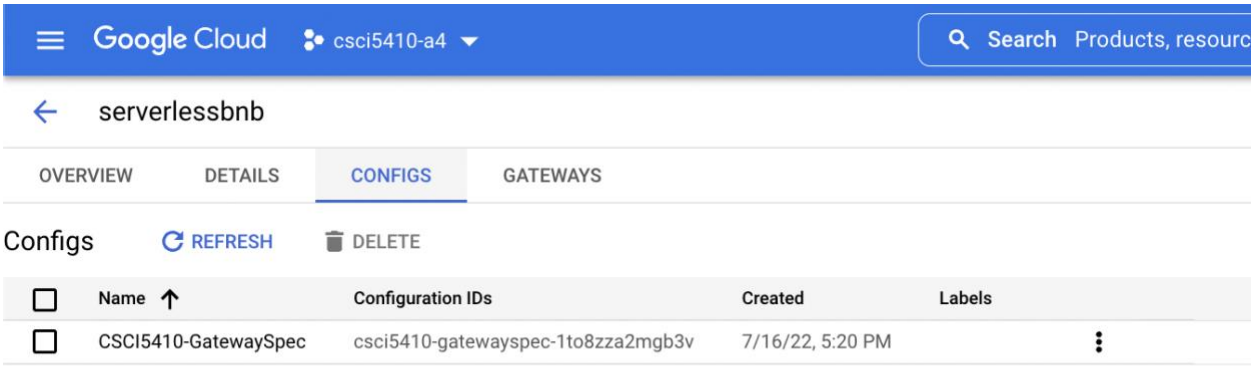


Figure 118: API Gateway open spec configuration for the created gateway

Figure 119 shows the cloud functions that supports the functioning of the tour operator service

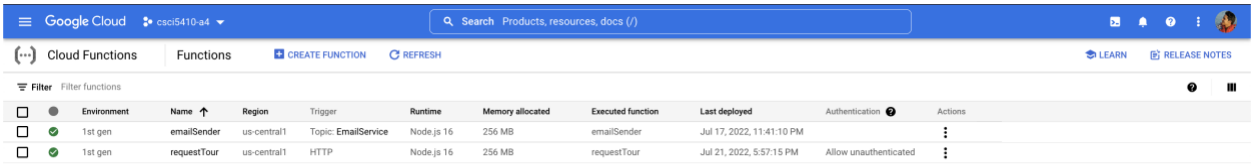


Figure 119: Cloud functions that process the tour requests and send out emails to the customers

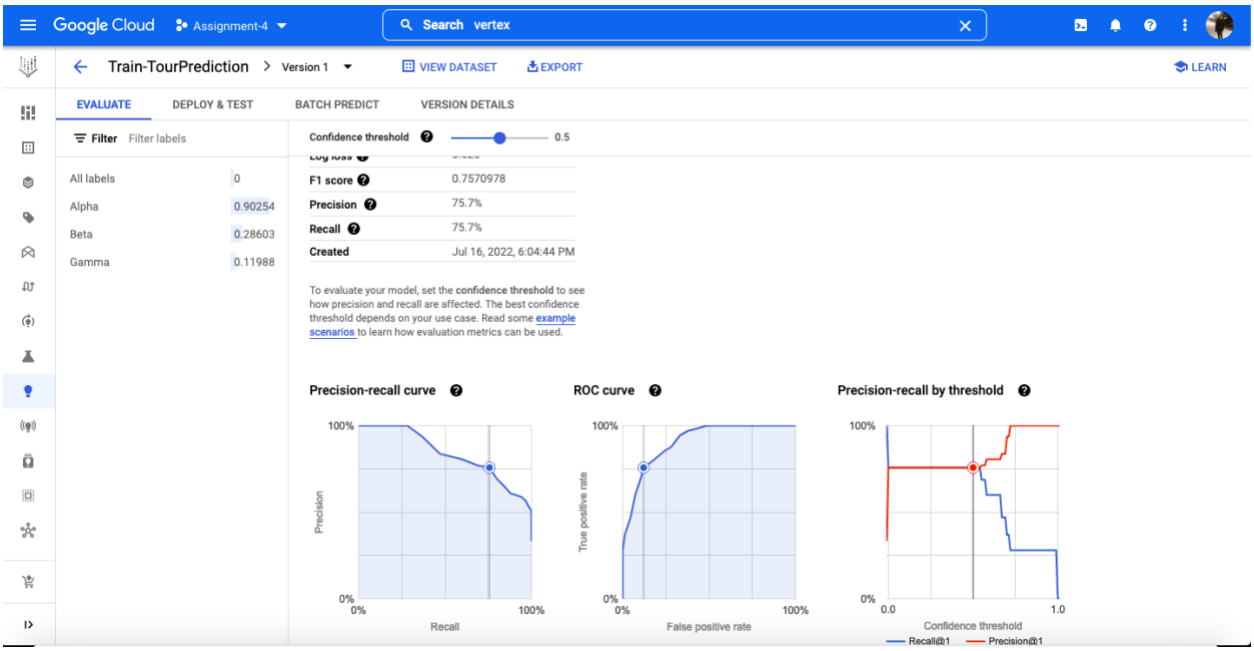


Figure 120: Model Created and its features

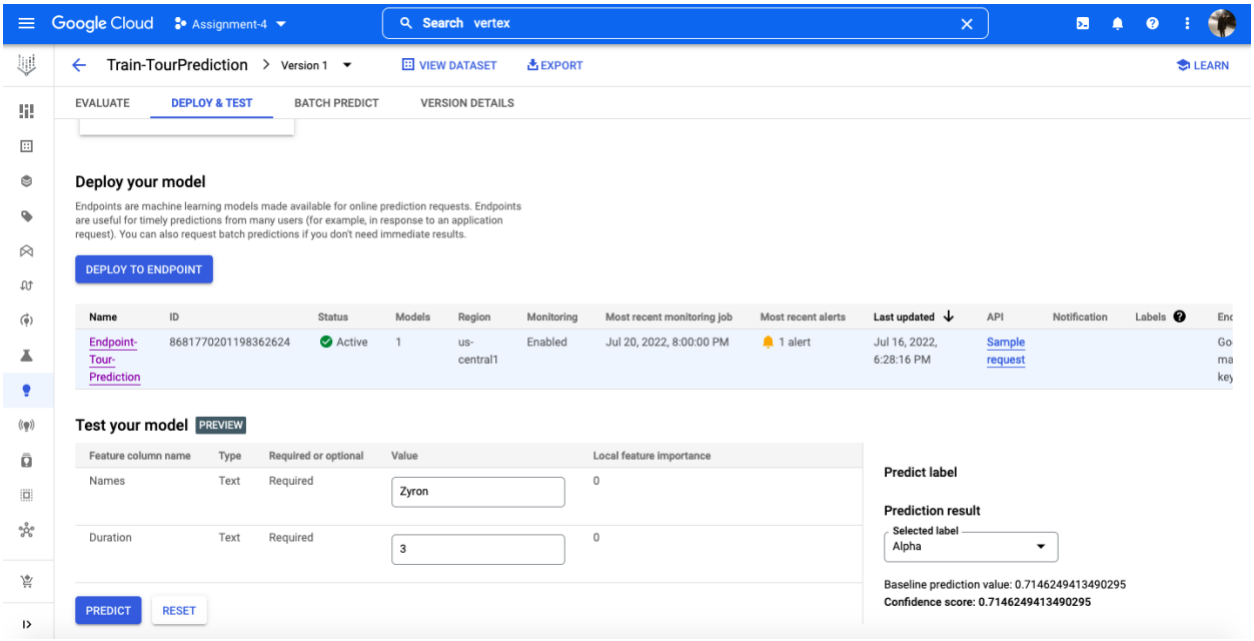


Figure 121: Deployment and Testing of Vertex AI Model

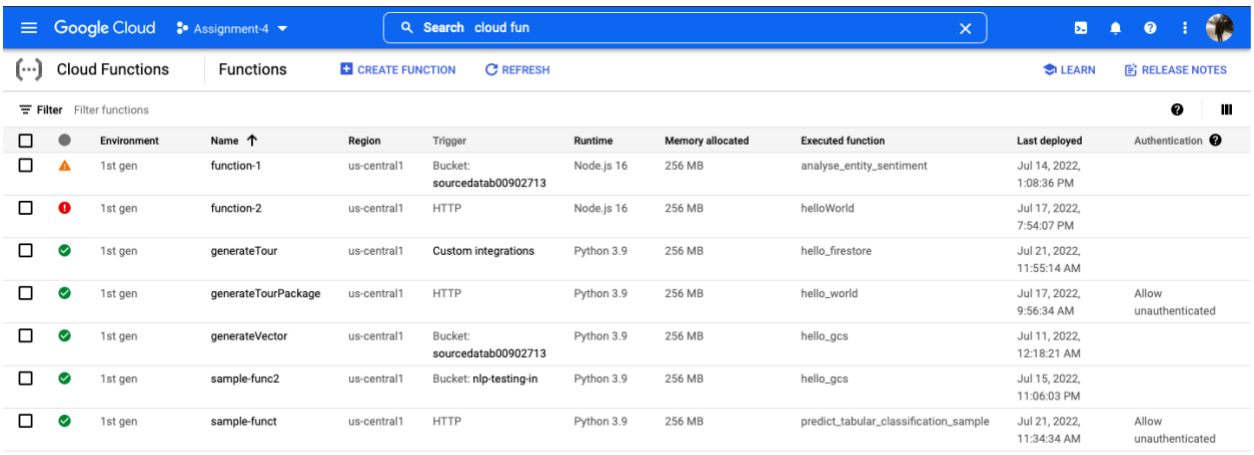


Figure 122: sample-funct Cloud Function to fetch the ML model

Figures 123 and 124 show the tour packages on DynamoDB

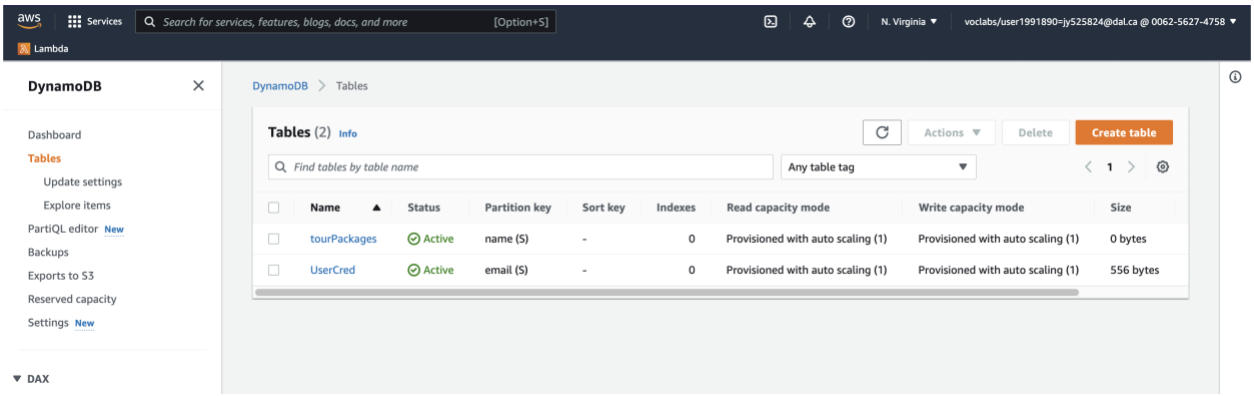


Figure 123: tourPackages table on DynamoDB that stores tour package details supported by ServerlessBnB

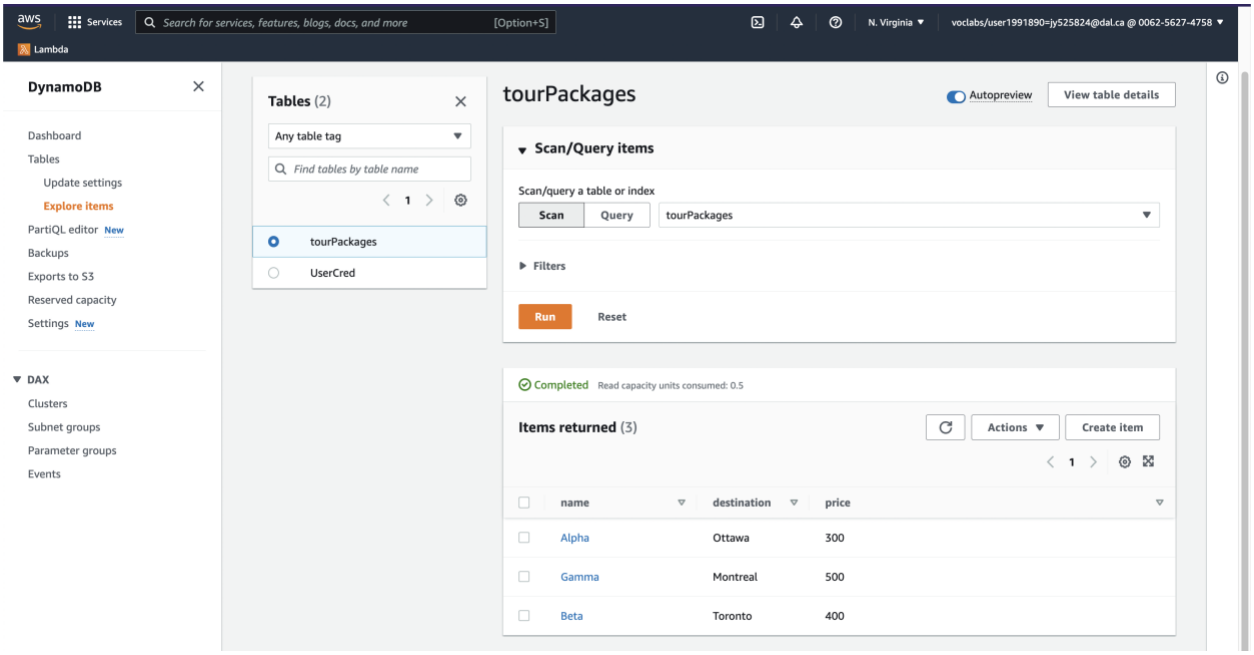


Figure 124: tourPackages DynamoDB table showing the available tour packages

Code snippets

requestTour.js: This file contains the code for the **requestTour** cloud function that processes the incoming tour request from the customers and returns the ideal tour package based on the requirements specified. It retrieves the list of recommended packages from the cloud function that interacts with the machine learning model and determines the package with the highest recommendation score. It retrieves the details of the package with the highest score from the **tourPackages** table on DynamoDB and places the details on the EmailService Pub/Sub topic. Following are the contents of this file:

```
const { PubSub } = require('@google-cloud/pubsub')
const aws = require('aws-sdk')
const axios = require('axios')

aws.config.loadFromPath('./config.json')

const pubsub = new PubSub()
const dynamodb = new aws.DynamoDB.DocumentClient()

const EMAIL_SERVICE_TOPIC = 'EmailService'

const formatTourPackages = ({ name, price, destination }) => {
 return `

   <h3>Here is the recommended tour package for you</h3>
```

```
    <p>
      <strong>Name:  ${name}
    </p>
    <p>
      <strong>Destination: ${destination}
    </p>
    <p>
      <strong>Price: CAD ${price}
    </p>
  `
}

const getTourPackageDetails = async (id) => {
  const params = {
    Key: {
      name: id,
    },
    TableName: 'tourPackages',
  }
  const res = await dynamodb.get(params).promise()
  return res
}

const getMostPreferredTour = (tourPackages) => {
  let maxScore = -1
  let mostPreferredTourPackageID = null
  for (const [packageId, score] of Object.entries(tourPackages)) {
    if (score > maxScore) {
      maxScore = score
      mostPreferredTourPackageID = packageId
    }
  }
  return mostPreferredTourPackageID
}

exports.requestTour = async (req, res) => {
  res.set('Access-Control-Allow-Origin', '*')
  res.set('Access-Control-Allow-Methods', '*')
  res.set('Access-Control-Allow-Headers', '*')
  const { stayDuration, recipientEmail } = req.body

  if (!stayDuration)
    return res.json({ success: false, message: 'Stay duration is required' })

  if (!recipientEmail)
    return res.json({ success: false, message: 'Recipient email is required' })

  const { data } = await axios.post(
```

```
  'https://us-central1-tidy-interface-355301.cloudfunctions.net/sample-funct',
  {
    Names: 'test',
    Duration: String(stayDuration),
  }
 )
 const { success, tourPackages, message: tourPredictionMessage } = data

 if (success) {
  try {
    const mostPreferredTourID = getMostPreferredTour(tourPackages)
    const { Item } = await getTourPackageDetails(mostPreferredTourID)

    const message = {
      recipient: recipientEmail,
      body: formatTourPackages(Item),
    }

    const messageBuffer = Buffer.from(JSON.stringify(message))

    await pubsub
      .topic(EMAIL_SERVICE_TOPIC)
      .publishMessage({ data: messageBuffer })

    return res.json({ success: true, tourPackage: Item })
  } catch (err) {
    console.error(err)
    return res.json({ success: false, message: err.message })
  }
 } else res.json({ success, message: tourPredictionMessage })
}
```

generateTourPackage.py : This file contains the code for generateTourPackage cloud function that gives data to Vertex AI which in turn generates the prediction score and prediction magnitude of the entered data.
Following are the contents of the file:

```
from typing import Dict
import json
from google.cloud import aiplatform
from google.protobuf import json_format
from google.protobuf.struct_pb2 import Value


# def hello_world(request):
#     request_json = request.get_json()
```

```
#    if request_json:
#        Dict={
#            "Names": request_json["Names"],
#            "Duration": request_json["Duration"],
#            }
#    return Dict


def predict_tabular_classification_sample(request):
    request_json = request.get_json()
    myDict={
        "Names": request_json["Names"],
        "Duration": request_json["Duration"],
        }

    # myDict={
    #     "Names": "Rishika",
    #     "Duration": "7",
    #     }
    final_response = {}
    project ="388819083887"
    endpoint_id ="8681770201198362624"
    instance_dict = myDict
    location = "us-central1"
    api_endpoint = "us-central1-aiplatform.googleapis.com"
    # The AI Platform services require regional API endpoints.
    client_options = {"api_endpoint": api_endpoint}
    # Initialize client that will be used to create and send requests.
    # This client only needs to be created once, and can be reused for multiple requests.
    client = aiplatform.gapic.PredictionServiceClient(client_options=client_options)
    # for more info on the instance schema, please use get_model_sample.py
    # and look at the yaml found in instance_schema_uri
    instance = json_format.ParseDict(instance_dict, Value())
    instances = [instance]
    parameters_dict = {}
    parameters = json_format.ParseDict(parameters_dict, Value())
    endpoint = client.endpoint_path(
        project=project, location=location, endpoint=endpoint_id
    )
    response = client.predict(
        endpoint=endpoint, instances=instances, parameters=parameters
    )
    print("response")
    print(" deployed_model_id:", response.deployed_model_id)
    # See gs://google-cloud-aiplatform/schema/predict/prediction/tabular_classification_1.0.0.yaml for the format of the
predictions.
    prediction_array = []
    count = 1
```

```
    predictions = response.predictions

    if(len(predictions) > 0):
        tour_prediction = dict(predictions[0])

        prediction_classes = tour_prediction['classes']
        prediction_scores = tour_prediction['scores']

        tour_packages = {key: value for (key,value) in zip(prediction_classes, prediction_scores)}

        return {'success': True, 'tourPackages': tour_packages}

    return {'success': False, 'message': 'No tour packages could be generated'}
```

emailSender.js: This file contains the code for the **emailSender** cloud function that listens to the EmailService Pub/Sub topic and sends out emails to the customers upon finding a message in the Pub/Sub topic. It uses the SendGrid email client to send out emails to the customers. Following are the contents of this file:

```
// This file is responsible for emailing tour packages to the customers

const sgMail = require('@sendgrid/mail')

sgMail.setApiKey(
  '<SENDGRID API KEY>'
)

async function sendMail(msg) {
  try {
    console.log('Sending email to ', msg.to)
    await sgMail.send(msg)
  } catch (err) {
    console.log(err.toString())
  }
}

exports.emailSender = async (message, context) => {
  const emailDetails = message.data
    ? JSON.parse(Buffer.from(message.data, 'base64').toString())
    : {}

  const msg = {
    to: emailDetails?.recipient,
    from: 'sc529025@dal.ca',
    subject: 'Tour Package Details',
    text: 'This is your tailored tour package',
```

```
  html: emailDetails?.body,
 }

 sendMail(msg)
}
```

apiGatewaySpec.yaml: This file contains the open spec API configuration used to setup the API Gateway service. It defines the endpoints for underlying cloud functions. Following are the contents of this file:

```
swagger: '2.0'
info:
 title: serverlessbnb API
 description: API gateway for securing the serverless API resources of the Serverless BnB application
 version: 1.0.0
schemes:
 - https
produces:
 - application/json

host: serverlessbnb-1akwzzj5qi6x3.apigateway.csci5410-a4-355201.cloud.goog
x-google-endpoints:
 - name: serverlessbnb-1akwzzj5qi6x3.apigateway.csci5410-a4-355201.cloud.goog
   allowCors: True

paths:
 /requestTour:
   post:
    summary: Request a tour package
    operationId: requestTour
    x-google-backend:
     address: https://us-central1-csci5410-a4-355201.cloudfunctions.net/requestTour
    responses:
     '200':
      description: A successful response
      schema:
       type: string
```

httpClient.js: This file is present in the utils folder inside the frontend code of the project. It sets up a base url that sends the requests to the API gateway created above. Following are the contents of this file:

```
import axios from 'axios'

const httpClient = axios.create({
```

```
  baseURL: 'https://csci5410-gateway-3zzzs9sg.ue.gateway.dev',
  headers: {
    'Access-Control-Allow-Origin': '*',
  },
})

export default httpClient
```

Testing screenshots

Figure 125 shows the interface through which the customer can specify their required length of tour package

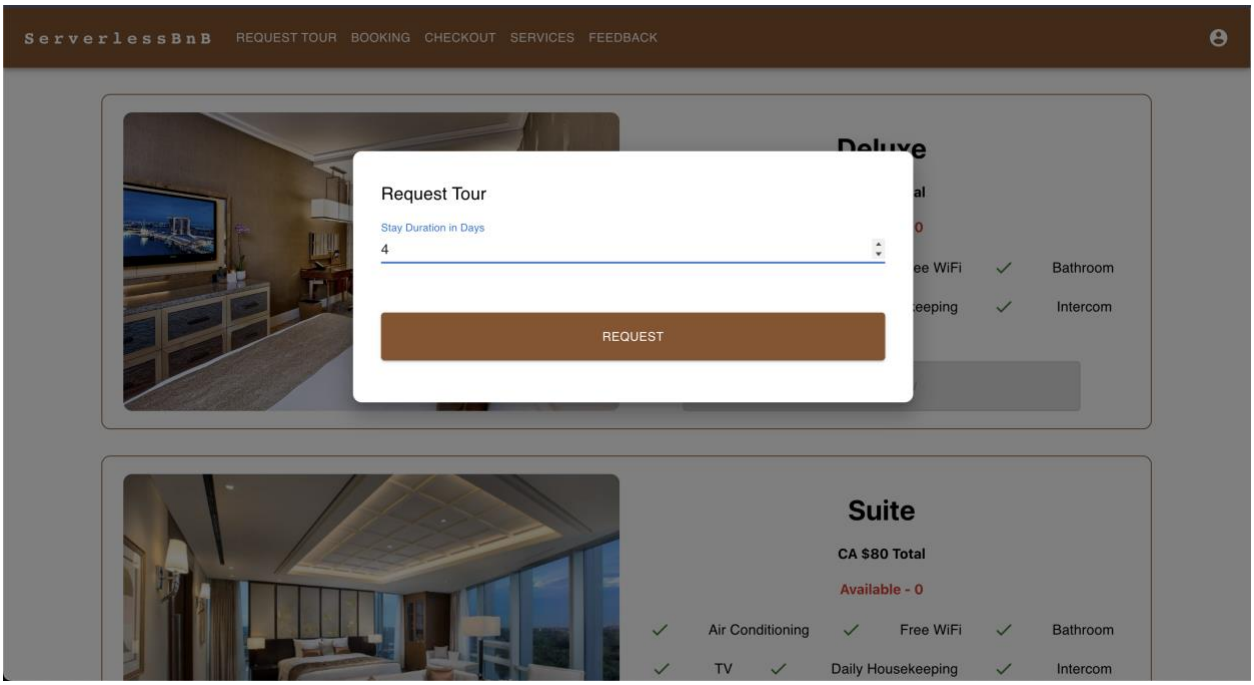


Figure 125: Request tour user interface that specifies the length of the required tour package

Figure 126 shows the recommended package details to the customer on the website

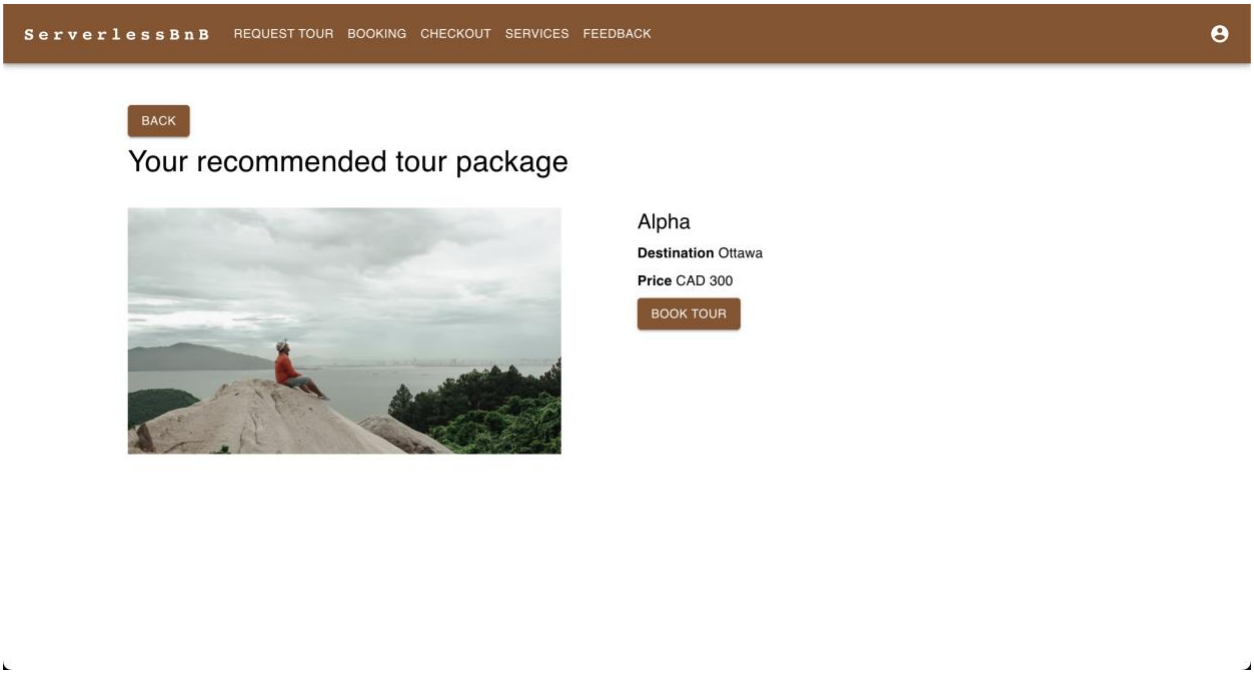


Figure 126: Recommended tour package details shown to the customer

Figure 127 shows the API Gateway endpoint the tour request was sent to.

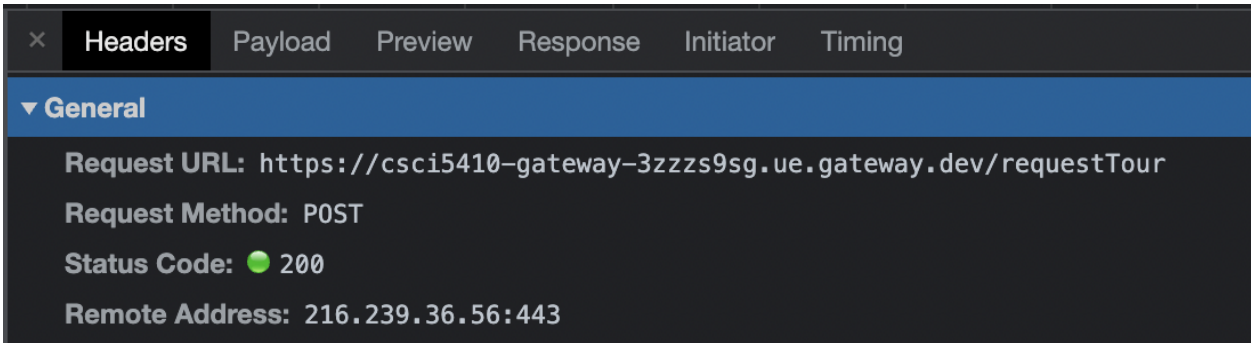


Figure 127: Tour package request sent to the requestTour cloud function hidden behind the API Gateway

Figure 128 shows the requestTour cloud function logs for the request made above

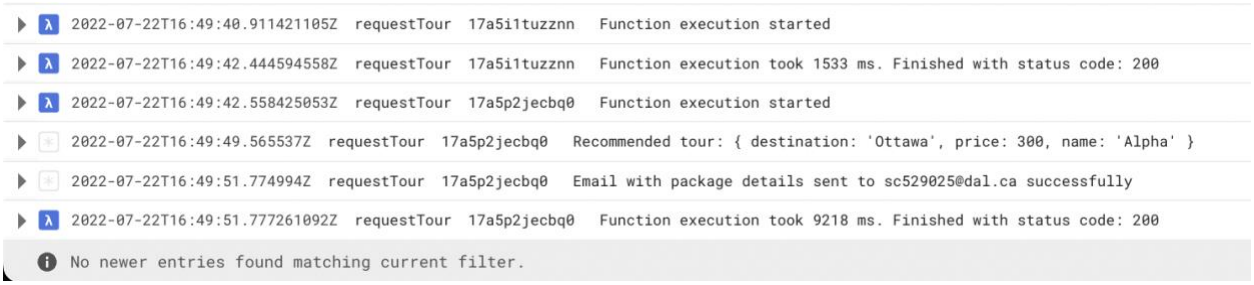


Figure 128: requestTour cloud function logs pertaining to the request made above

Figure 129 the email with tour package details sent to the customer

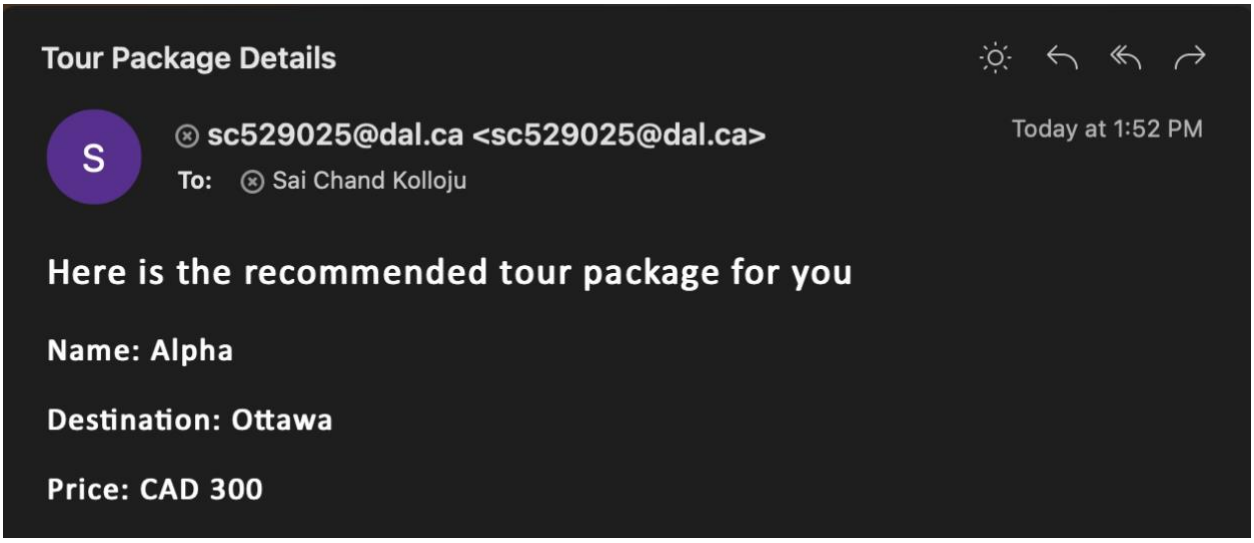


Figure 129: Email with tour package details sent to the logged in user

Customer Feedback and Feedback Analysis

The customer feedback and feedback analysis service is responsible for taking feedback from the customers and generating the feedback results.

Implementation Details

The feedback service makes use of the following cloud services from the Google Cloud Platform (GCP) cloud providers:

- **Firestore:** Firestore [5] is used to store names and feedbacks from the customer in a collection. It is also used to store feedback sentiment analysis in another collection.
- **Cloud Functions**: Cloud functions [10] are used to fetch data from firestore and calculates feedback sentiment analysis and add the same to firestore.
- **Cloud Natural Language API**: Cloud Natural Language API [13] is used for the sentiment analysis of the feedbacks added by the users.

Following is the sequence of actions that take place from the feedback initialization to the sentiment analysis of the feedback:

1. Authorized users would be able to enter the feedback. The user has to add the name and the feedback of the user.
2. The request is saved in the firestore in the collection called **feedback.**
3. The cloud function named **feedback_Score_analysis** fetches data from the collection and uses Natural Language API to do the sentiment analysis of the feedback.

4. After doing the sentiment analysis of the feedback, it stores the data back in the firestore with sentiment score, sentiment magnitude and polarity.
5. The feedback gets displayed on the website.

Step by Step:

1. Entering Feedback from the user interface.

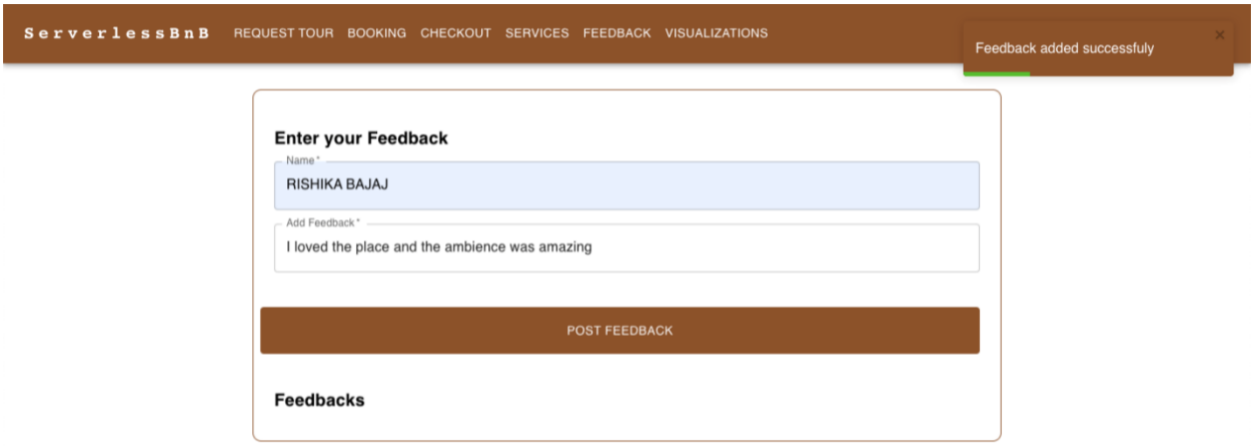


Figure 130: Inserting feedback through our UI

2. Feedback gets stored in the Firestore in a collection.

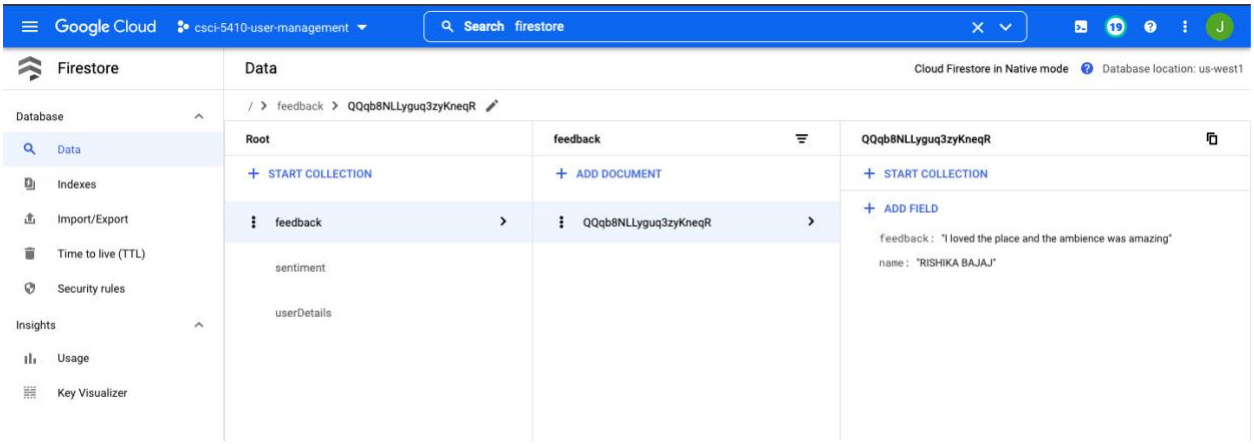


Figure 131: Feedback gets stored in the feedback collection

3. Cloud Function triggers when a new value is added in the collection. It takes the latest value and hits the language api to predict sentiment score.

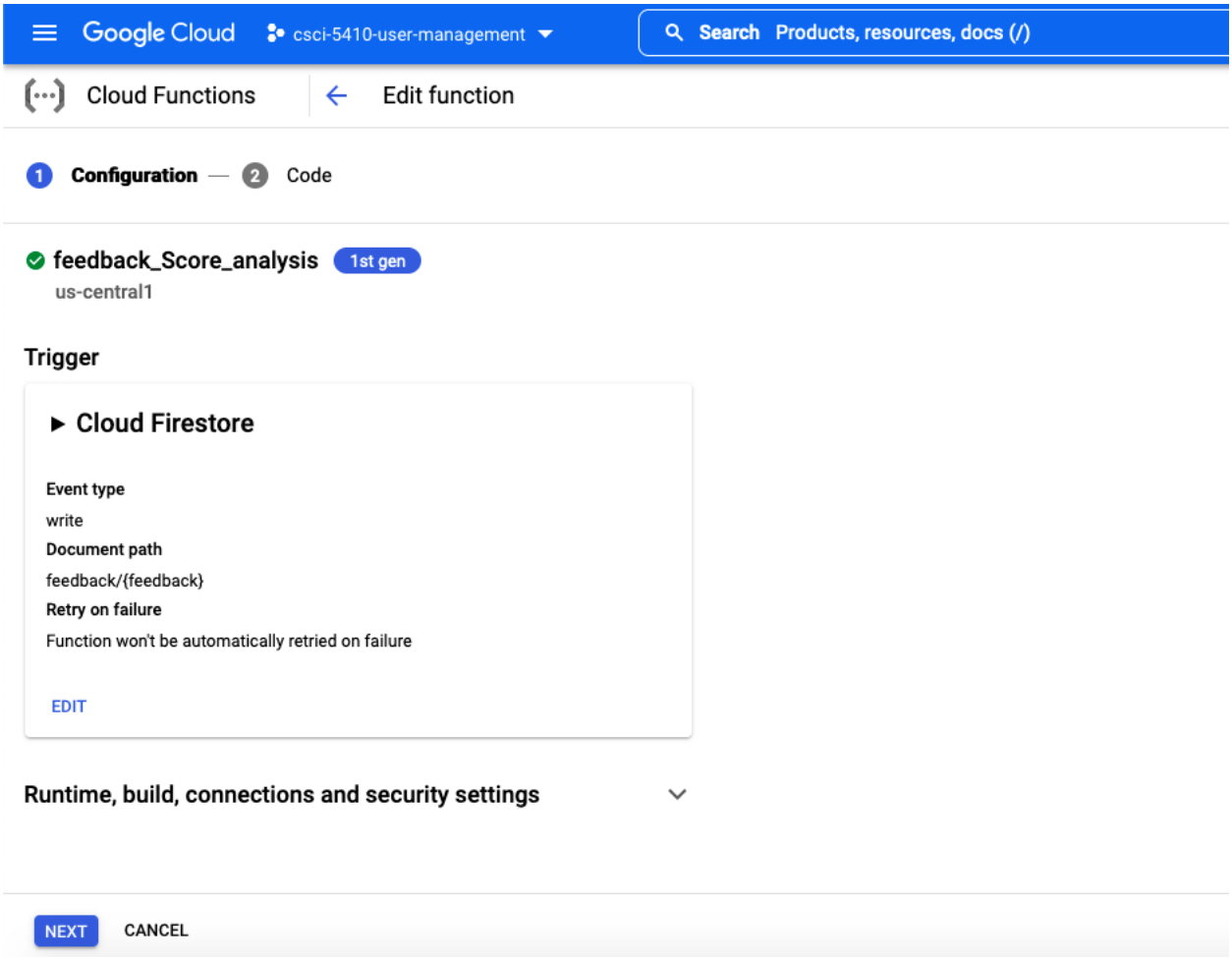


Figure 132: Triggering the cloud function when a document gets added in feedback collection

Code Snippets

feedback_Score_Analysis.py : This file is basically used to analyse the feedback received from the firestore and analyze it. After sentiment analysis of the feedback, the polarity, sentiment score and magnitude is stored in the new collection in the firestore.

```
from google.cloud import firestore
from google.cloud import language_v1
import json
import io

# Add a new document
db = firestore.Client()
client = language_v1.LanguageServiceClient()

def hello_firestore(event, context):
  # Then query for documents
```

```
  path_parts = context.resource.split('/documents/')[1].split('/')
  collection_path = path_parts[0]
  document_path = '/'.join(path_parts[1:])

  doc = db.collection(collection_path).document(document_path)
  cur_value = event["value"]["fields"]["feedback"]["stringValue"]
  cur_name = event["value"]["fields"]["name"]["stringValue"]
  name = str(cur_name)
  text = str(cur_value)
  document = language_v1.Document(content=text, type_=language_v1.Document.Type.PLAIN_TEXT)
  sentiment = client.analyze_sentiment(request={"document": document}).document_sentiment
  print(sentiment)
  if (sentiment.score < 0):
    polarity = "NEGATIVE"
  else:
    polarity = "POSITIVE"
  feed = {}
  feed['name'] = name
  feed['feedback'] = text
  feed['sentimentscore'] = sentiment.score
  feed['sentimentmagnitude'] = sentiment.magnitude
  feed['polarity'] = polarity
  jsonText = ""
  db.collection(u'sentiment').add(feed)
  jsonText +="Text: {}, Sentiment Score: {}, Sentiment Magnitude: {}, Polarity : {}".format(text, sentiment.score,
sentiment.magnitude, polarity)
  print (jsonText)
```

4. Sentiment analysis of the feedback gets stored in the Firestore in a collection called sentiment.

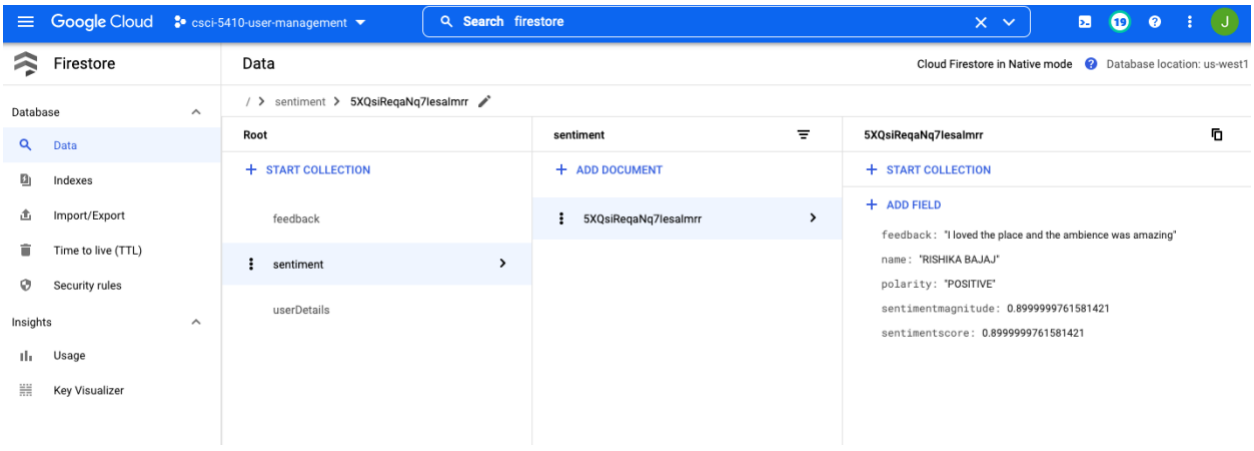


Figure 133: Sentiment Collection in Firestore

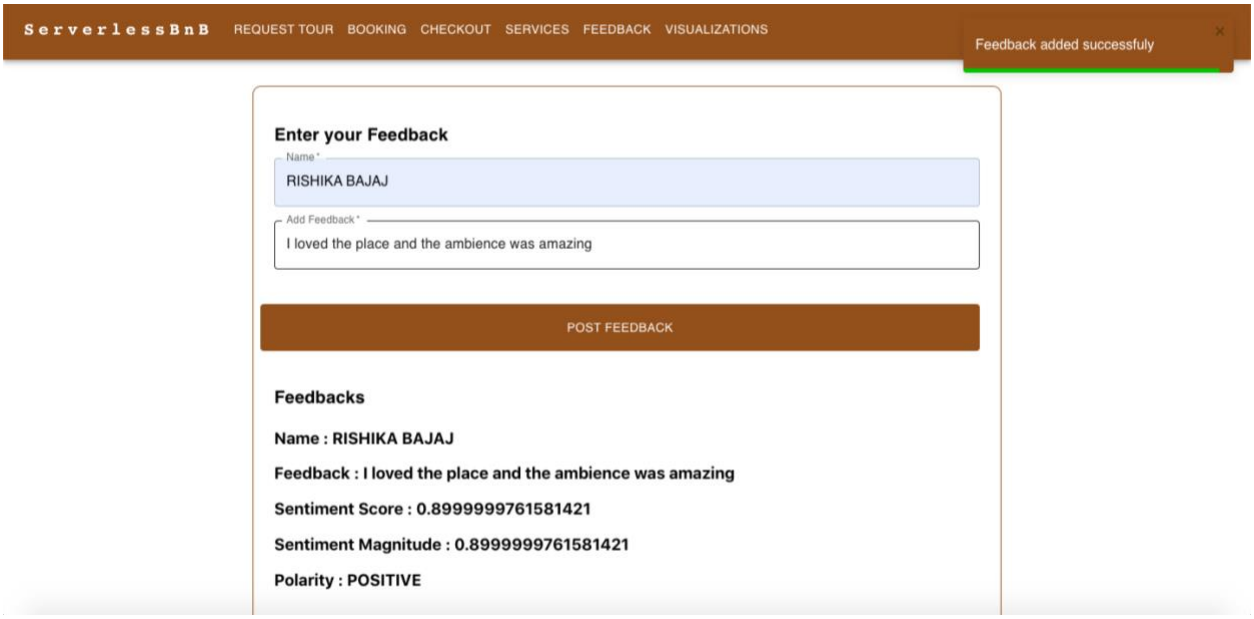


Figure 134: Feedback Sentiment Analysis being displayed in the UI

Report Generation

There are two main services used for generating the report – AWS CloudTrail [14] and AWS CloudWatch [15].

The user logs are maintained the number of logins, registrations and other login details are maintained by integrating AWS CloudTrail [14]. A trail is created that monitors the sign-in and sign-up activities that take place in AWS Cognito [7].

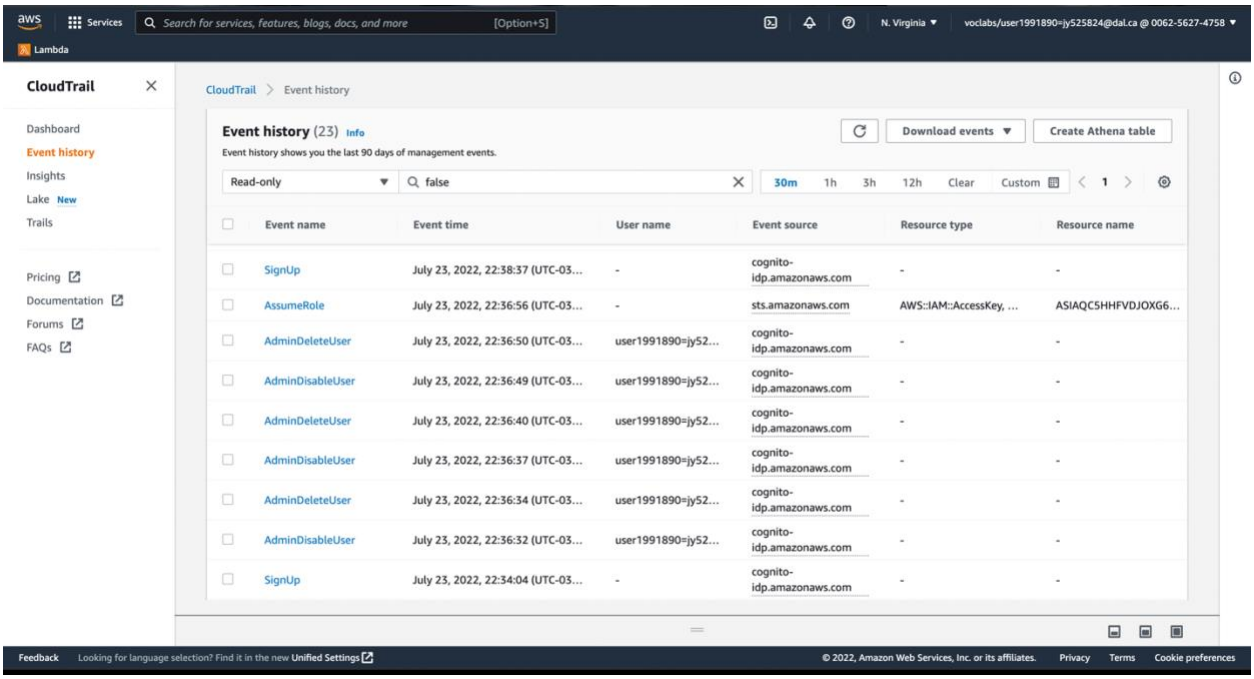
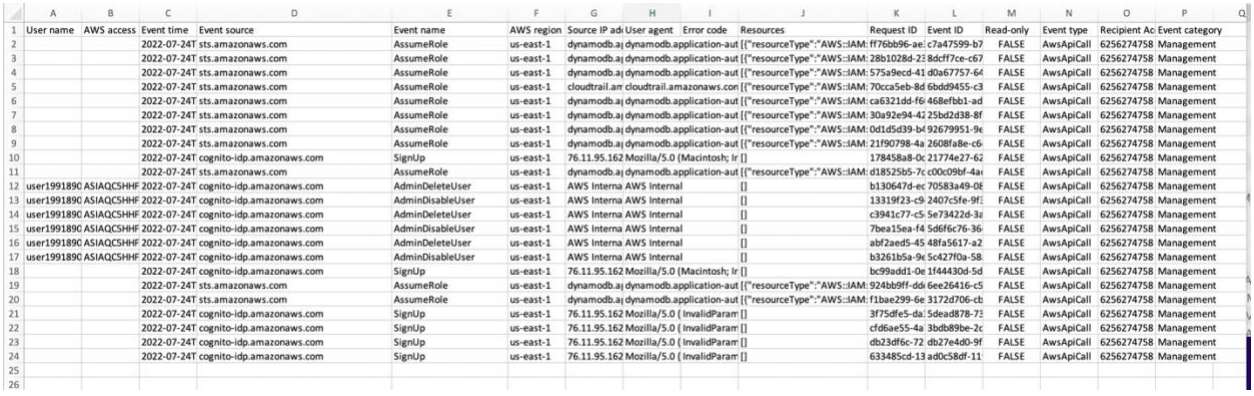


Figure 135: Cloud trail event history dashboard showing AWS Cognito logs

The report can also be downloaded in a csv format as shown below



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	User name	AWS access	Event time	Event source	Event name	AWS region	Source IP ad	User agent	Error code	Resources	Request ID	Event ID	Read-only	Event type	Recipient Ac	Event category	
2			2022-07-24T	sts.amazonaws.com	AssumeRole	us-east-1	dynamodb.a	dynamodb.application-aut		[{"resourceType":"AWS::IAM:	ff76bb96-ae	c7a47599-b7	FALSE	AwsApiCall	6256274758	Management	
3			2022-07-24T	sts.amazonaws.com	AssumeRole	us-east-1	dynamodb.a	dynamodb.application-aut		[{"resourceType":"AWS::IAM:	28b1028d-23	8dcff7ce-c67	FALSE	AwsApiCall	6256274758	Management	
4			2022-07-24T	sts.amazonaws.com	AssumeRole	us-east-1	dynamodb.a	dynamodb.application-aut		[{"resourceType":"AWS::IAM:	575a9ecd-41	d0a67757-64	FALSE	AwsApiCall	6256274758	Management	
5			2022-07-24T	sts.amazonaws.com	AssumeRole	us-east-1	cloudtrail.am	cloudtrail.amazonaws.com		[{"resourceType":"AWS::IAM:	70cca5eb-8d	6bdd9455-c3	FALSE	AwsApiCall	6256274758	Management	
6			2022-07-24T	sts.amazonaws.com	AssumeRole	us-east-1	dynamodb.a	dynamodb.application-aut		[{"resourceType":"AWS::IAM:	ca6321dd-f6	468efbb1-ad	FALSE	AwsApiCall	6256274758	Management	
7			2022-07-24T	sts.amazonaws.com	AssumeRole	us-east-1	dynamodb.a	dynamodb.application-aut		[{"resourceType":"AWS::IAM:	30a92e94-42	25bd2d38-8f	FALSE	AwsApiCall	6256274758	Management	
8			2022-07-24T	sts.amazonaws.com	AssumeRole	us-east-1	dynamodb.a	dynamodb.application-aut		[{"resourceType":"AWS::IAM:	0d1d5d39-b4	92679951-9e	FALSE	AwsApiCall	6256274758	Management	
9			2022-07-24T	sts.amazonaws.com	AssumeRole	us-east-1	dynamodb.a	dynamodb.application-aut		[{"resourceType":"AWS::IAM:	21f90798-4a	2608fa8e-c6	FALSE	AwsApiCall	6256274758	Management	
10			2022-07-24T	cognito-idp.amazonaws.com	SignUp	us-east-1	76.11.95.162	Mozilla/5.0 (Macintosh; Ir		[]	178458a8-0c	21774e27-62	FALSE	AwsApiCall	6256274758	Management	
11			2022-07-24T	sts.amazonaws.com	AssumeRole	us-east-1	dynamodb.a	dynamodb.application-aut		[{"resourceType":"AWS::IAM:	d18525b5-7c	c00c09bf-4a	FALSE	AwsApiCall	6256274758	Management	
12	user1991890	ASIAQC5HHF	2022-07-24T	cognito-idp.amazonaws.com	AdminDeleteUser	us-east-1	AWS Interna	AWS Internal		[]	b130647d-ec	70583a49-08	FALSE	AwsApiCall	6256274758	Management	
13	user1991890	ASIAQC5HHF	2022-07-24T	cognito-idp.amazonaws.com	AdminDisableUser	us-east-1	AWS Interna	AWS Internal		[]	13319f23-c9	2407c5fe-9f	FALSE	AwsApiCall	6256274758	Management	
14	user1991890	ASIAQC5HHF	2022-07-24T	cognito-idp.amazonaws.com	AdminDeleteUser	us-east-1	AWS Interna	AWS Internal		[]	c3941c77-c5	5e73422d-3a	FALSE	AwsApiCall	6256274758	Management	
15	user1991890	ASIAQC5HHF	2022-07-24T	cognito-idp.amazonaws.com	AdminDisableUser	us-east-1	AWS Interna	AWS Internal		[]	7bea15ea-f4	5d6f6c76-36	FALSE	AwsApiCall	6256274758	Management	
16	user1991890	ASIAQC5HHF	2022-07-24T	cognito-idp.amazonaws.com	AdminDeleteUser	us-east-1	AWS Interna	AWS Internal		[]	abf2aed5-45	48fa5617-a2	FALSE	AwsApiCall	6256274758	Management	
17	user1991890	ASIAQC5HHF	2022-07-24T	cognito-idp.amazonaws.com	AdminDisableUser	us-east-1	AWS Interna	AWS Internal		[]	b3261b5a-9e	5c427f0a-58	FALSE	AwsApiCall	6256274758	Management	
18			2022-07-24T	cognito-idp.amazonaws.com	SignUp	us-east-1	76.11.95.162	Mozilla/5.0 (Macintosh; Ir		[]	bc99add1-0e	1f44430d-5d	FALSE	AwsApiCall	6256274758	Management	
19			2022-07-24T	sts.amazonaws.com	AssumeRole	us-east-1	dynamodb.a	dynamodb.application-aut		[{"resourceType":"AWS::IAM:	924bb9ff-dd	6ee26416-c5	FALSE	AwsApiCall	6256274758	Management	
20			2022-07-24T	sts.amazonaws.com	AssumeRole	us-east-1	dynamodb.a	dynamodb.application-aut		[{"resourceType":"AWS::IAM:	f1bae299-6e	3172d706-cb	FALSE	AwsApiCall	6256274758	Management	
21			2022-07-24T	cognito-idp.amazonaws.com	SignUp	us-east-1	76.11.95.162	Mozilla/5.0 (InvalidParam		[]	3f75dfe5-da	5dead878-73	FALSE	AwsApiCall	6256274758	Management	
22			2022-07-24T	cognito-idp.amazonaws.com	SignUp	us-east-1	76.11.95.162	Mozilla/5.0 (InvalidParam		[]	cfd6ae55-4a	3bdb89be-2c	FALSE	AwsApiCall	6256274758	Management	
23			2022-07-24T	cognito-idp.amazonaws.com	SignUp	us-east-1	76.11.95.162	Mozilla/5.0 (InvalidParam		[]	db23df6c-72	db27e4d0-9f	FALSE	AwsApiCall	6256274758	Management	
24			2022-07-24T	cognito-idp.amazonaws.com	SignUp	us-east-1	76.11.95.162	Mozilla/5.0 (InvalidParam		[]	633485cd-13	ad0c58df-11	FALSE	AwsApiCall	6256274758	Management	
25																	
26																	

Figure 136: CSV file with event history from AWS CloudTrail

Reports of logs from storing information on DynamoDB can be got from AWS CloudWatch as shown below

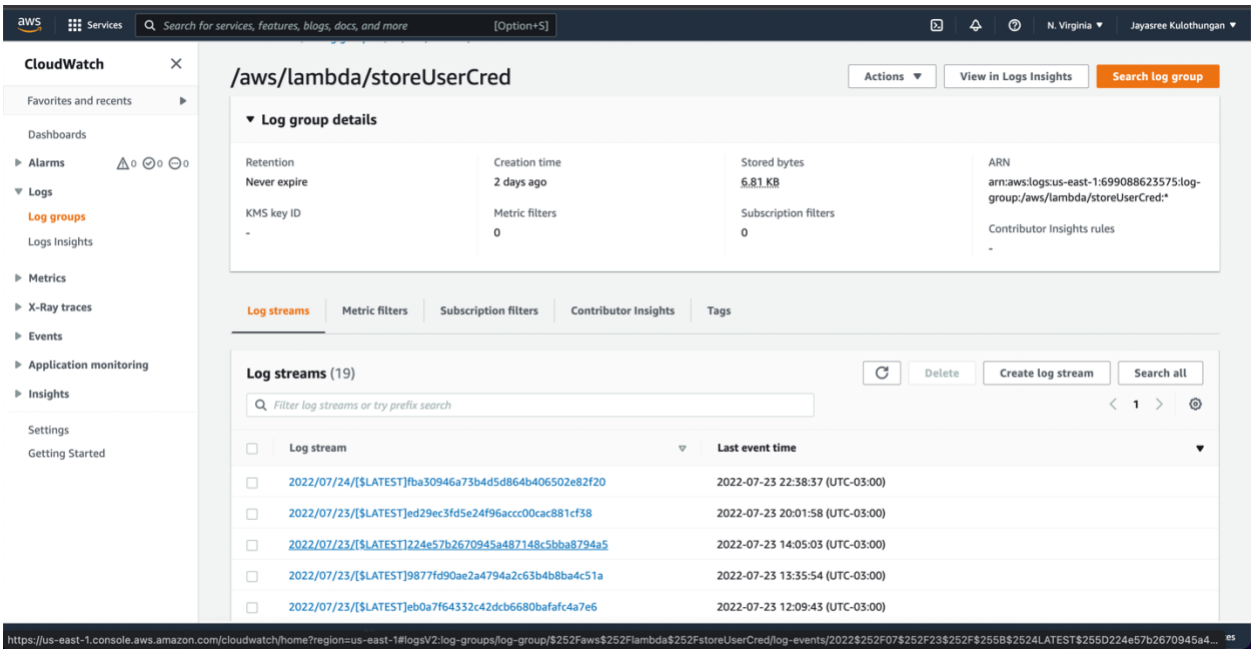


Figure 137: Cloudwatch logs

Visualizations

The visualizations module generates graphs and charts that help in visualizing the ServerlessBnB application data. As part of this module, we have generated three visualizations:

- A bar graph that shows the number of available rooms by its type
- A bar graph that shows the number of active bookings by room type
- A pie chart that shows the proportion of orders by item

Implementation Details

The visualizations are generated by utilizing the following cloud services:

- **Cloud Functions**: Cloud functions [10] are used in this module to gather application data exposed by other cloud services and generate CSV files from the gathered data. In addition, they also upload the created CSV files to a Google Cloud Storage bucket.
- **Google Cloud Storage [22]**: This service is used to store the CSV files generated by the cloud functions and acts as a data source to the Google Data Studio service.
- **Google Data Studio [23]**: This service is responsible for pulling the data from the CSV files stored on the Cloud Storage [22] bucket and creating a report that shows the visualizations.

Following is the sequence of actions that take place from the initial view visualization request to the generation of a report with visualizations:

1. Admins of the ServerlessBnB application would be able to request the generation of visualizations and view them on the user interface of the website.

2. When the admin of the application visits the webpage that shows the visualizations, **generateVisualizations** cloud function is triggered by sending an HTTP request to the cloud function.
3. The **generateVisualizatons** cloud function gathers the application data exposed by the other services of the application and generates CSV files for the data.
4. The generated CSV files are uploaded to the **serverlessbnb-visualizations** cloud storage bucket.
5. The Google Data Studio [23] cloud service, with Cloud Storage as its data connector, pulls the data from the CSV files stored on bucket and creates a visualizations report.
6. The report is embedded into the frontend of the application by using the embed link associated with the report produced by Data Studio [23].
7. The admin can see the visualizations on the /visualizations route of the web application.

Flow Chart

Figure 138 shows the flow chart of the actions that place in the tour operator service

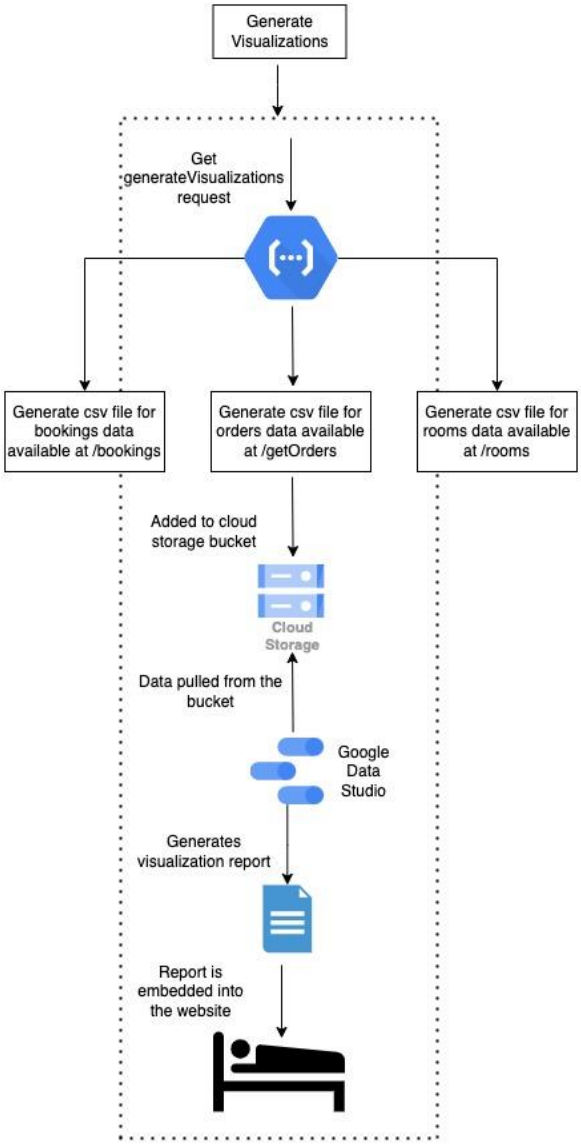


Figure 138: Visualizations flow chart

Cloud services screenshots

Following are the screenshots of the cloud services that enable the generation of visualizations:

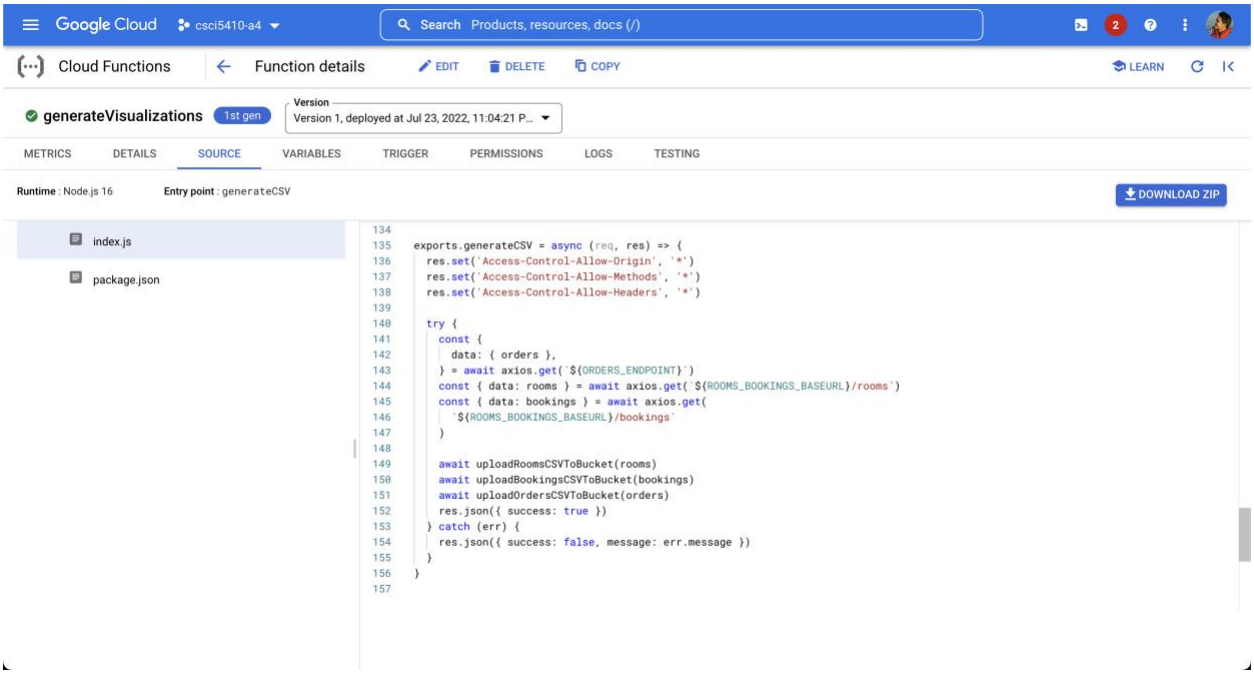


Figure 139: generateVisualizations cloud function that generates and stores CSV data on Cloud Storage bucket

Figure 140 shows the Cloud Storage bucket that stores the CSV files generated by the cloud function

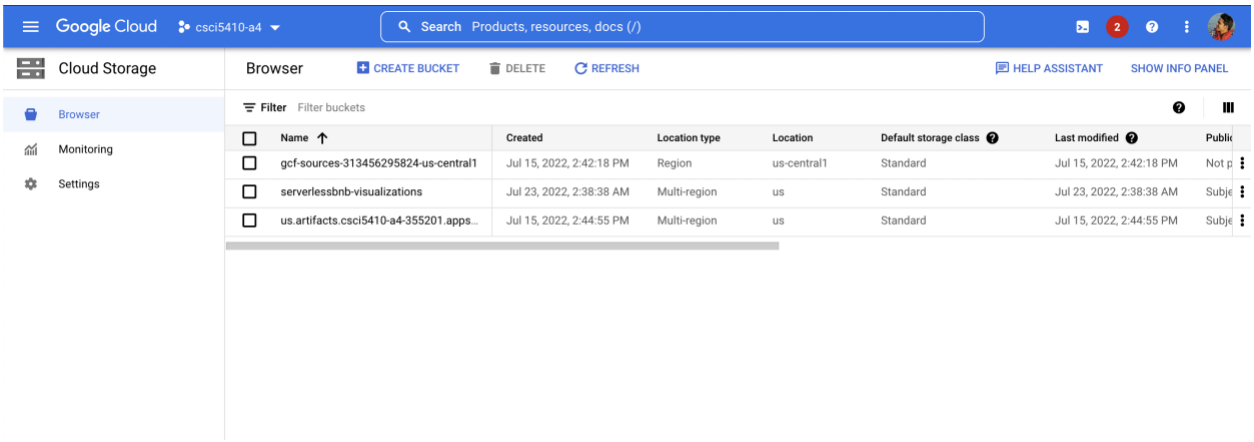


Figure 140: Cloud Storage bucket that stores the CSV files generated from application data

Figure 141 shows the CSV files that are generated and stored on the Cloud Storage bucket by the **generateVisualizations** cloud function.

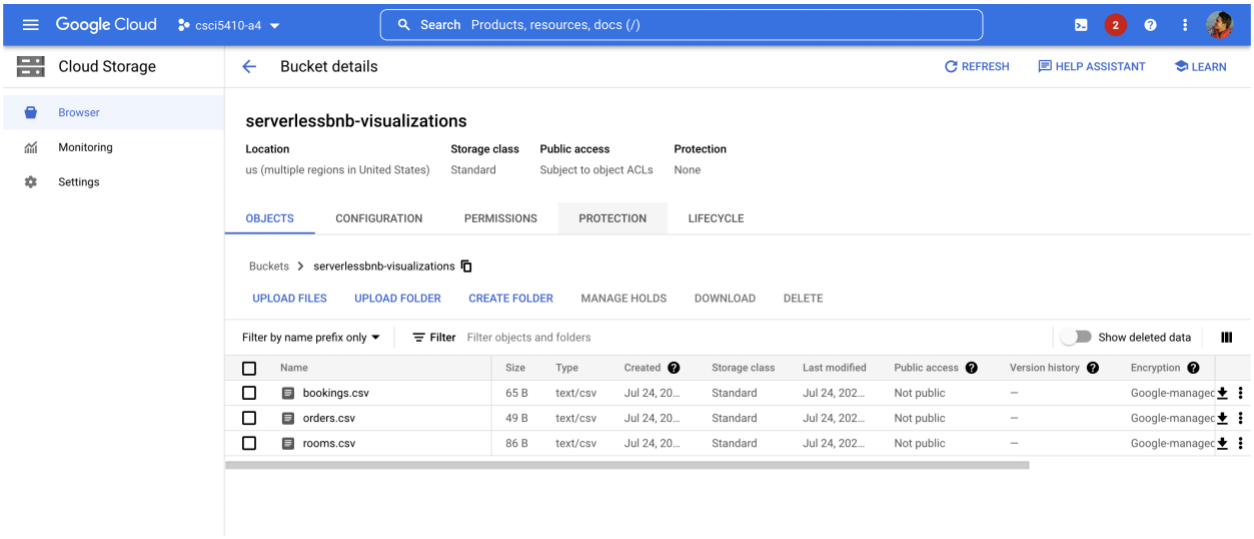


Figure 141: CSV files generated and stored on Cloud Storage bucket

Figure 142 shows the report on Data Studio dashboard that contains the visualizations from the application data

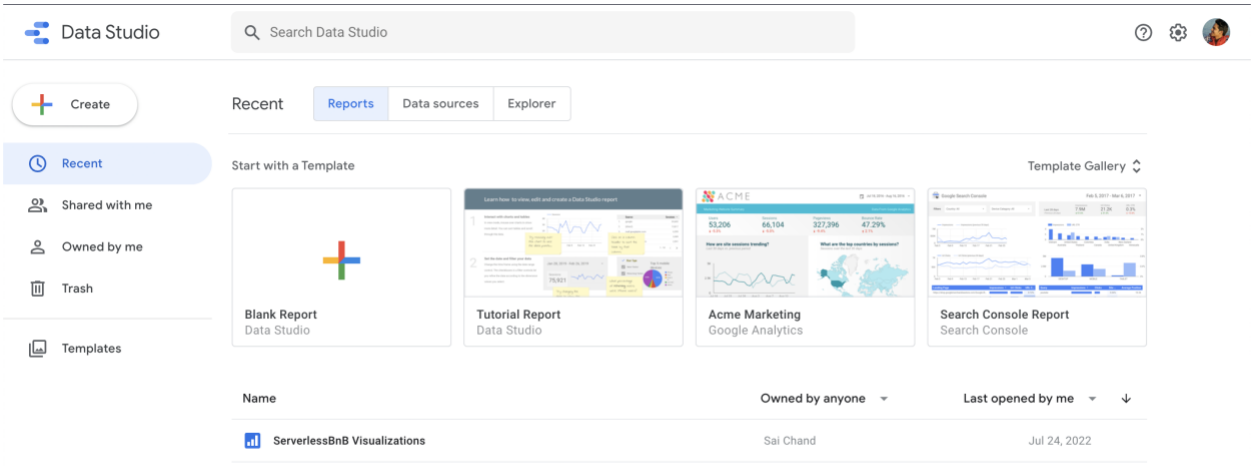


Figure 142: Data Studio dashboard showing the ServerlessBnB Visualizations report

Figure 143 shows the visualizations generated by Data Studio.

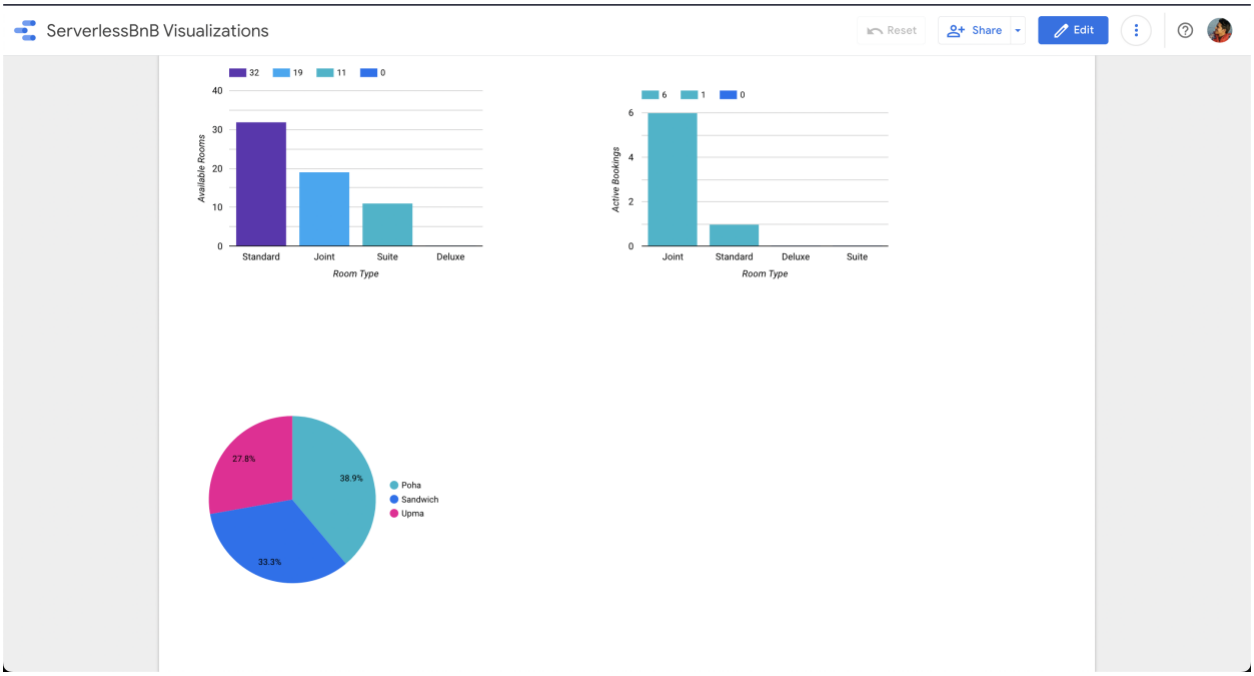


Figure 143: Visualizations inside ServerlessBnB Visualizations report created by Data Studio

Code Snippets

generateVisualizations.js: This file contains the code for the **generateVisualizations** cloud function that gathers application data from the endpoints exposed by other services of the application, transforms the data into CSV files and uploads it to the Cloud Storage bucket. Following are the contents of this file:

```
const axios = require('axios')
const { Storage } = require('@google-cloud/storage')
const { EOL } = require('os')

const storage = new Storage()

const VISUALIZATIONS_BUCKET = 'serverlessbnb-visualizations'
const ROOMS_BOOKINGS_BASEURL =
 'https://xqet4nlofotprm2yiiz6rvrcm40giliq.lambda-url.us-east-1.on.aws'

const ORDERS_ENDPOINT =
 'https://kitchen-service-kc2rqvhqga-uc.a.run.app/getOrders'

async function createBucket(bucketName) {
 try {
  // Citation: https://googleapis.dev/nodejs/storage/latest/Bucket.html#exists
  // The following line of code is written by referring to the source cited above
```

```
    const [bucketExists] = await storage.bucket(bucketName).exists()

    if (!bucketExists) {
      // Citation: https://googleapis.dev/nodejs/storage/latest/Storage.html#createBucket
      // The following line of code is written by referring to the source cited above
      await storage.createBucket(bucketName)
      console.log(`Bucket "${bucketName}" created successfully`)
    } else {
      console.log('Bucket exists already')
    }
  } catch (err) {
    console.log(err.message)
  }
}

async function uploadCSVToBucket(
  bucketName,
  targetFileName,
  csvStrings,
  csvHeader
) {
  await createBucket(bucketName)
  const targetFile = await storage.bucket(bucketName).file(targetFileName)
  // Citation: https://googleapis.dev/nodejs/storage/latest/File.html#createWriteStream
  // The following line of code is written by referring to the source cited above
  const fileStream = await targetFile.createWriteStream()
  fileStream.write(csvHeader)
  fileStream.write(csvStrings.join(EOL))
  fileStream.end()
}

async function uploadRoomsCSVToBucket(rooms) {
  const csvStrings = rooms.map(
    ({ type, available, price }) => `${type},${available},${price}${EOL}`
  )
  await uploadCSVToBucket(
    VISUALIZATIONS_BUCKET,
    'rooms.csv',
    csvStrings,
    `Room Type,Available Rooms,Price${EOL}`
  )
}

async function uploadBookingsCSVToBucket(bookings) {
  const csvStrings = []

  const activeBookingsFrequencyMap = new Map()
```

```
 for (const booking of bookings) {
  const { type, active } = booking

  if (activeBookingsFrequencyMap.has(type)) {
   if (active) {
    activeBookingsFrequencyMap.set(
     type,
     activeBookingsFrequencyMap.get(type) + 1
    )
   }
  } else {
   if (active) {
    activeBookingsFrequencyMap.set(type, 1)
   } else {
    activeBookingsFrequencyMap.set(type, 0)
    console.log(`Setting ${type} to 0`)
   }
  }
 }

 const activeBookingsFrequencyObject = Object.fromEntries(
  activeBookingsFrequencyMap
 )

 for (const [type, activeBookingsCount] of Object.entries(
  activeBookingsFrequencyObject
 )) {
  const csvString = `${type},${activeBookingsCount}${EOL}`
  csvStrings.push(csvString)
 }

 await uploadCSVToBucket(
  VISUALIZATIONS_BUCKET,
  'bookings.csv',
  csvStrings,
  `Room Type,Active Bookings${EOL}`
 )
}

async function uploadOrdersCSVToBucket(orders) {
 const csvStrings = []

 const ordersFrequencyMap = new Map()

 for (const _order of orders) {
  const { order } = _order

  if (ordersFrequencyMap.has(order)) {
```

```
    ordersFrequencyMap.set(order, ordersFrequencyMap.get(order) + 1)
  } else {
    ordersFrequencyMap.set(order, 1)
  }
 }

 const ordersFrequencyObject = Object.fromEntries(ordersFrequencyMap)

 for (const [item, count] of Object.entries(ordersFrequencyObject)) {
  const csvString = `${item},${count}${EOL}`
  csvStrings.push(csvString)
 }

 await uploadCSVToBucket(
  VISUALIZATIONS_BUCKET,
  'orders.csv',
  csvStrings,
  `Item,Number of orders${EOL}`
 )
}

exports.generateVisualizations = async (req, res) => {
 res.set('Access-Control-Allow-Origin', '*')
 res.set('Access-Control-Allow-Methods', '*')
 res.set('Access-Control-Allow-Headers', '*')

 try {
  const {
   data: { orders },
  } = await axios.get(`${ORDERS_ENDPOINT}`)
  const { data: rooms } = await axios.get(`${ROOMS_BOOKINGS_BASEURL}/rooms`)
  const { data: bookings } = await axios.get(
   `${ROOMS_BOOKINGS_BASEURL}/bookings`
  )

  await uploadRoomsCSVToBucket(rooms)
  await uploadBookingsCSVToBucket(bookings)
  await uploadOrdersCSVToBucket(orders)
  res.json({ success: true })
 } catch (err) {
  res.json({ success: false, message: err.message })
 }
}
```

Testing Screenshots

Figure 144 shows the initial load on the visualizations page which triggers the **generateVisualizations** cloud function.

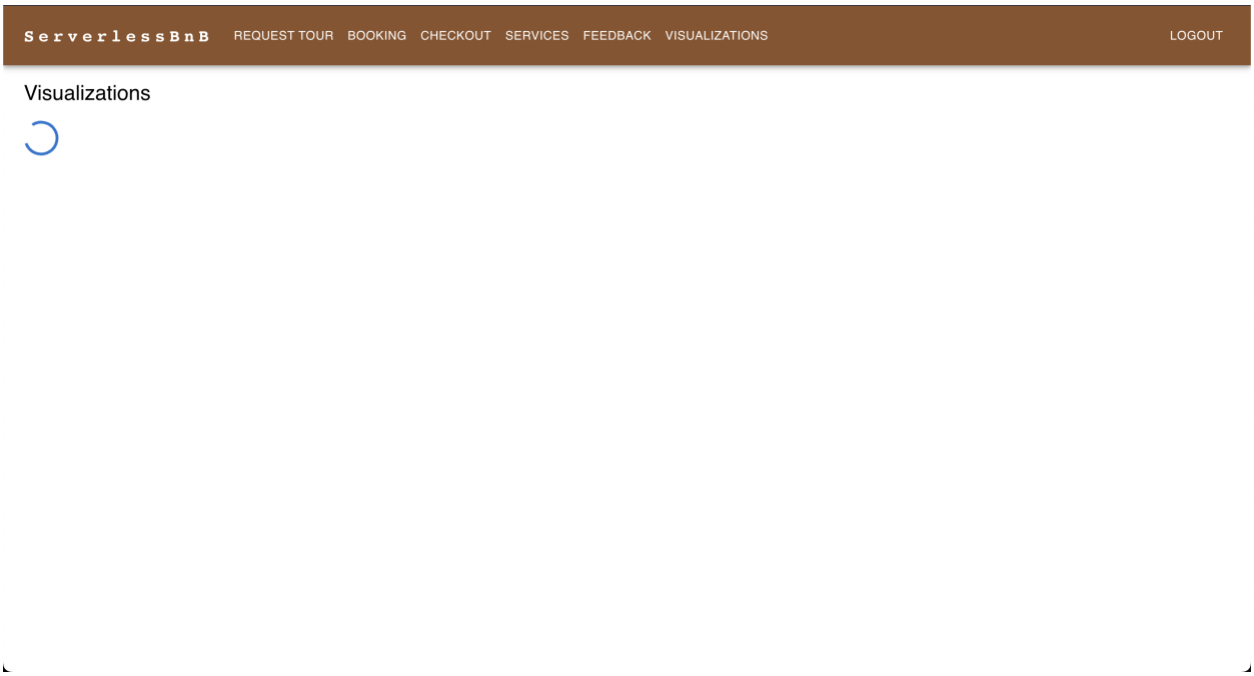


Figure 144: generateVisualizations cloud function trigger that updates the visualizations with latest application data

Figure 145 shows the visualizations generated on the website

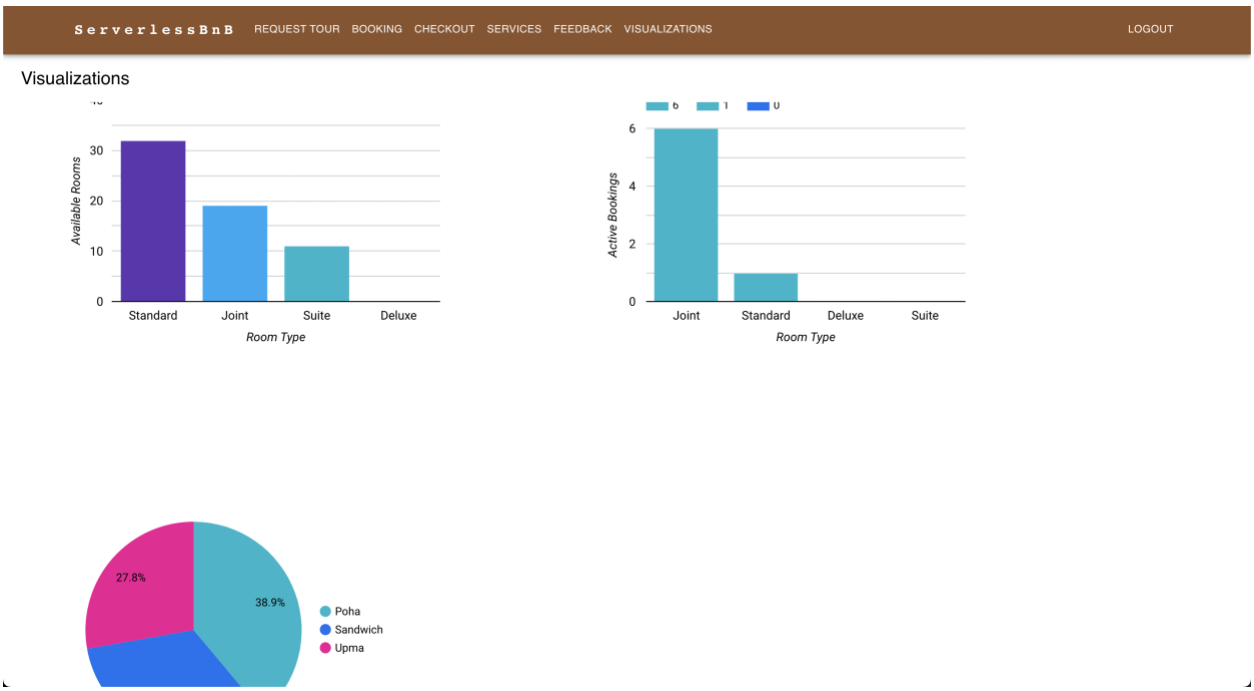


Figure 145: Visualizations generated on the webpage

Website building and hosting

The frontend of our project is built using the React frontend library [2]. The frontend is deployed to a S3 bucket on AWS that hosts the static assets generated by building the React frontend code [2]. Following is the sequence of actions that take place in the deployment process:
1. A git commit is made, and the frontend code is pushed to the **main** branch of the project repository on GitLab [24].
2. The GitLab CI/CD pipeline builds the static assets from the React code using the **npm run build** command and uploads the static assets to an S3 bucket [4].
3. The S3 bucket is configured to support website hosting. The website can be accessed using the URL generated for the bucket [4].

Code snippets

.gitlab-ci.yml: This file is responsible constructs a CI/CD pipeline that builds and deploys the frontend of the application to the S3 bucket. Following are the contents of the file:

```
stages:
 - build
 - deploy

build_stage:
 image: node:alpine
 stage: build
 only:
  - main
 script:
  - npm install --prefix frontend
  - CI=false npm run build --prefix frontend
 artifacts:
  paths:
   - frontend/build/
  expire_in: 1 hour

deploy_stage:
 image: python:latest
 stage: deploy
 only:
  - main
 script:
  - pip install awscli
  - aws s3api create-bucket --bucket $S3_BUCKET
  - aws s3api put-public-access-block --bucket $S3_BUCKET --public-access-block-configuration
"BlockPublicAcls=false,IgnorePublicAcls=false,BlockPublicPolicy=false,RestrictPublicBuckets=false"
  - |
```

```
    aws s3api put-bucket-policy --bucket $S3_BUCKET --policy "{ \"Version\": \"2012-10-17\", \"Statement\": [ { \"Sid\": \"PublicReadGetObject\", \"Effect\": \"Allow\", \"Principal\": \"*\", \"Action\": \"s3:GetObject\", \"Resource\": \"arn:aws:s3:::$S3_BUCKET/*\" }] }"
  - aws s3 website s3://$S3_BUCKET --index-document index.html
  - aws s3 sync ./frontend/build s3://$S3_BUCKET
```

Meeting Logs

Meeting Number : 1
Date Of Meeting : May 16, 2022
Time: 9:00pm - 9:20pm (20 minutes)
Place: WhatsApp
Agenda : Introductory Meeting
Discussion and Outcomes: We all introduced ourselves and discussed about the technologies we were comfortable with. At the end of the meeting, we knew the technology stack each team member was comfortable with.

Meeting Number: 2
Date of the meeting: May 21, 2022
Time: 9:30pm - 9:50pm (20 minutes)
Place: WhatsApp
Agenda: Modules distribution
Discussion and Outcomes: We distributed the core component modules among ourselves to explore the cloud services they used. At the end of the meeting, each team member had a cloud service to explore.

Meeting Number: 3
Date of the meeting: May 25, 2022
Time: 9:00pm - 9:30pm (30 minutes)
Place: Microsoft Teams
Agenda: Project conceptual report discussion
Discussion and Outcomes: We went through the project conception report requirements and distributed the components to work on. At the end of the meeting, each team member had a component to report for the initial draft of the project conception report.

Meeting Number: 4
Date of the meeting: May 26, 2022
Time: 9:00pm - 9:15pm (15 minutes)
Place: Microsoft Teams
Agenda: Project conception report discussion

Discussion and Outcomes: We combined our work done for the initial project conception report draft into one document and decided to explore the other aspects of the report. At the end of the meeting, we had the core component part of the document ready.

Meeting Number: 5
Date of the meeting: May 27, 2022
Time: 9:00pm - 9:20pm (20 minutes)
Agenda: Finalizing the tasks for the modules remaining for the project conception report
Discussion and Outcomes: Finalized the tasks needed to complete the project conception report and distributed the tasks.

Meeting Number: 6
Date of the meeting: June 23, 2022
Time: 9:00pm - 9:23pm
Place: Microsoft Teams (See Figure 146)
Agenda: Project conception report feedback discussion and initial database design
Discussion: Analysed the feedback given for the project conception report submission to incorporate the suggestions made in the future project submissions. We created an initial database design subject to additional changes in the future.

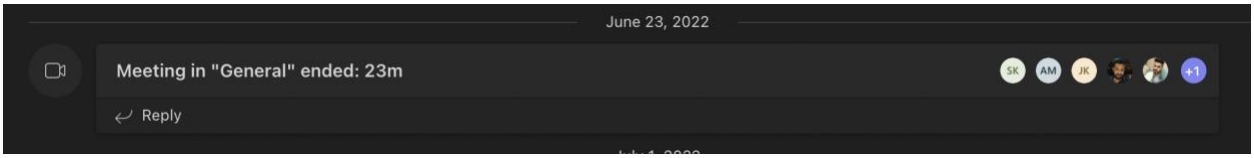


Figure 146: Microsoft Teams meeting record for meeting 6

Meeting Number: 7
Date of the meeting: July 1, 2022
Time: 10:00pm - 10:10pm
Place: Microsoft Teams (See Figure 147)
Agenda: Studying the design document requirements
Discussion: Speculated over design document requirements and formed a rough draft of the document highlighting the sections to be included in the document. We decided to meet in person to work on the document.

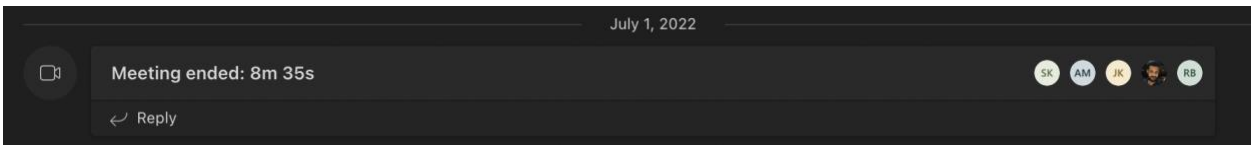


Figure 147: Microsoft Teams meeting record for meeting 7

Meeting Number: 8
Date of the meeting: July 3, 2022

Time: 1:00pm - 5:30pm
Place: Collaborative Health Education Building
Agenda: Project architecture and roadmap discussion
Discussion: Discussed how the overall architecture of the project should be designed and studied the interactions between the services used in building the project. We revisited the requirements document to better understand the scope of the project. We also made changes to our initial database design to include other related fields. We went through the design document requirements and decided to distribute the work.

Meeting Number: 9
Date of the meeting: July 4, 2022
Time: 2:00pm - 2:15pm
Place: Microsoft Teams (See Figure 148)
Agenda: Task distribution for design document
Discussion: Distributed the tasks among the team members and assigned responsibilities of completing their portion of the design document.

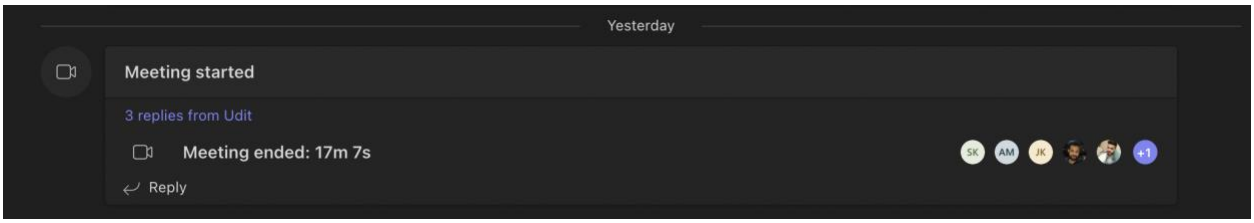


Figure 148: Microsoft Teams meeting record for meeting 9

Meeting Number: 10
Date of the meeting: July 12, 2022
Time: 12:00pm - 12:10pm
Place: Microsoft Teams (See Figure 149)
Agenda: Status updates on development and suggested changes
Discussion: All the members of the team gave their status update on development of the modules and conducted a peer review of the implementation and suggested changes.

Figure 149: Microsoft Teams meeting record for meeting 10

Meeting Number: 11
Date of the meeting: July 18, 2022

Time: 12:00pm - 12:25pm
Place: Microsoft Teams (See Figure 150)
Agenda: Testing of the modules completed till date on production environment
Discussion: The modules developed till date were Integrated and tested on the production environment.

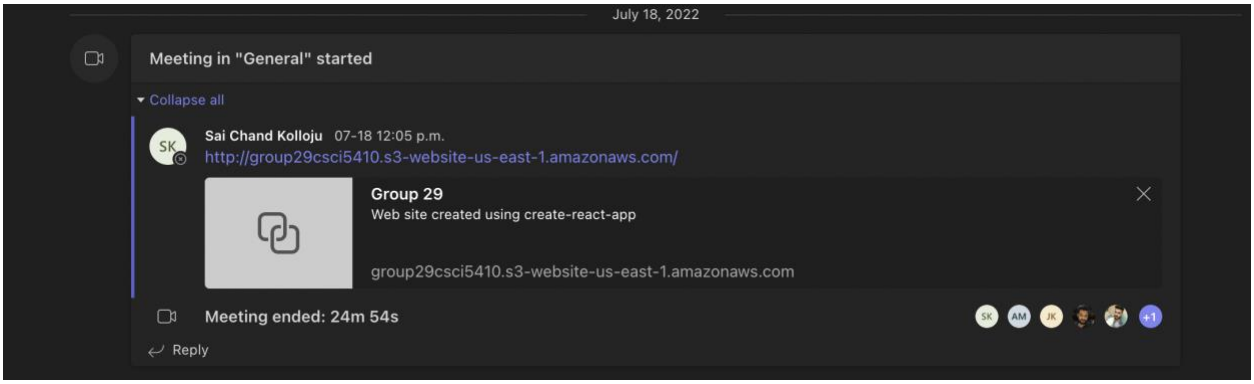


Figure 150: Microsoft Teams meeting record for meeting 11

Meeting Number: 12
Date of the meeting: July 24, 2022
Time: 9:00pm - 9:23pm
Place: Microsoft Teams (See Figure 151)
Agenda: Demo preparation
Discussion: Final testing and code changes were made as part of the demo preparation

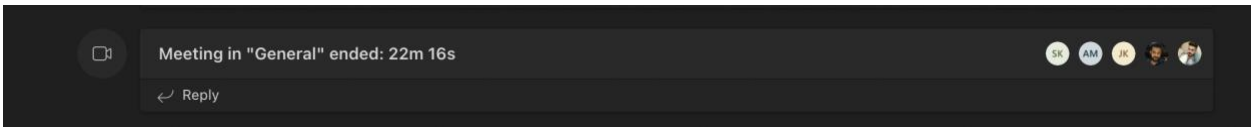


Figure 151: Microsoft Teams meeting record for meeting 12

References

[1] Aditya Mahale, Jayasree Kulothungan, Rishika Bajaj, Sai Chand Kolloju, Sourav Malik, Udit Gandhi, “Project Conception Report,” *Dalhousie University*, 2022. [Online]. [Accessed: 21-Jul-2022]

[2] Aditya Mahale, Jayasree Kulothungan, Rishika Bajaj, Sai Chand Kolloju, Sourav Malik, Udit Gandhi, “Project Design Document,” *Dalhousie University*, 2022. [Online]. [Accessed: 21-Jul-2022]

[3] “React – a JavaScript library for building user interfaces,” *– A JavaScript library for building user interfaces*. [Online]. Available: https://reactjs.org/. [Accessed: 25-Jul-2022].

[4] “S3,” *Amazon*, 2002. [Online]. Available: https://aws.amazon.com/s3/. [Accessed: 21-Jul-2022].

[5] “Firestore: Nosql document database | google cloud,” *Google*. [Online]. Available: https://cloud.google.com/firestore/. [Accessed: 21-Jul-2022].

[6] D. Rangel, “DynamoDB: Everything you need to know about Amazon Web Service's NoSQL database,” *Amazon*, 2015. [Online]. Available: https://aws.amazon.com/dynamodb/. [Accessed: 22-Jul-2022].

[7] H. Roose, “Cognito,” *Amazon*, 1987. [Online]. Available: https://aws.amazon.com/cognito/. [Accessed: 22-Jul-2022].

[8] "Conversational AI and Chatbots - Amazon Lex - Amazon Web Services", *Amazon Web Services, Inc.*, 2022. [Online]. Available: https://aws.amazon.com/lex/. [Accessed: 22- Jul-2022]

[9] D. A. V. I. D. MUSGRAVE, “Lambda,” *Amazon*, 2022. [Online]. Available: https://aws.amazon.com/lambda/. [Accessed: 23-Jul-2022].

[10] “Cloud functions | google cloud,” *Google*. [Online]. Available: https://cloud.google.com/functions/. [Accessed: 22-Jul-2022].

[11] “Pub/Sub for Application & Data Integration | google cloud,” *Google*. [Online]. Available: https://cloud.google.com/pubsub/. [Accessed: 23-Jul-2022].

[12] “Vertex ai | google cloud,” *Google*. [Online]. Available: https://cloud.google.com/vertex-ai. [Accessed: 23-Jul-2022].

[13] “Cloud natural language | google cloud,” *Google*. [Online]. Available: https://cloud.google.com/natural-language/. [Accessed: 22-Jul-2022].

[14] “Secure standardized logging - AWS CloudTrail - Amazon Web Services.” [Online]. Available: https://aws.amazon.com/cloudtrail/. [Accessed: 21-Jul-2022].

[15] “Amazon Cloudwatch - application and Infrastructure Monitoring.” [Online]. Available: https://aws.amazon.com/cloudwatch/. [Accessed: 21-Jul-2022].

[16] “X.400 gateway API specification ; X.400 API Associations,” *Amazon*, 1989. [Online]. Available: https://aws.amazon.com/api-gateway/. [Accessed: 25-Jul-2022].

[17] Node.js, *Node.js*. [Online]. Available: https://nodejs.org/. [Accessed: 25-Jul-2022].

[18] “Cloud run: Container to production in seconds | google cloud,” *Google*. [Online]. Available: https://cloud.google.com/run/. [Accessed: 25-Jul-2022].

[19] “Email delivery, API, Marketing Service,” *SendGrid*. [Online]. Available: https://sendgrid.com/. [Accessed: 23-Jul-2022].

[20] "API gateway | google cloud," *Google*. [Online]. Available: https://cloud.google.com/api-gateway. [Accessed: 25-Jul-2022].
[21] "Cloud Automl Custom Machine Learning models | google cloud," *Google*. [Online]. Available: https://cloud.google.com/automl/. [Accessed: 25-Jul-2022].
[22] "Cloud storage | google cloud," *Google*. [Online]. Available: https://cloud.google.com/storage/. [Accessed: 25-Jul-2022].
[23] *Google Data studio overview*. [Online]. Available: https://datastudio.google.com/. [Accessed: 21-Jul-2022].
[24] "The One DevOps platform," *GitLab*. [Online]. Available: https://about.gitlab.com/. [Accessed: 21-Jul-2022].