29 – March – 2020

ADVANCED TOPICS IN CLOUD COMPUTING – CSCI 5409

FINAL PROJECT REPORT

Secured Lightweight Tourism Application for Canada

TEAM – 8

Name	Banner ID	Email
Gaurav Anand	**B00832139**	**gr874432@dal.ca**
Akshay Reddy Poturi	**B00822804**	**ak427072@dal.ca**
Shrey Rameshbhai Vaghela	**B00834792**	**sh367824@dal.ca**
Shruthi Kalasapura Ramesh	**B00822766**	**sh871216@dal.ca**

Project Requirements:

Overview:

The project requires to develop a tourism application for Canada. The users should be able to access the application from mobile and computers. In order to fulfil this, an android app and a website are required. The app should be able to allow users to search for tourist spots like tourist attractions (e.g. Niagara Falls), national parks, major beaches and cities. The application should highlight key features of the searched location. In order to book a ticket, a user should have an account and should be logged in. After the successful payment of the trip, a sample ticket/invoice is to be generated. Some of the key requirements of the project are-

- Security – A secured app where 2-step authentication, secure data transmission, and data encryption is implemented.
- Efficiency – The app should be developed which should be efficient while fulfilling the project requirements in the best way possible.
- Accuracy – The accuracy of the application is must while fetching data about the searched location, buses available, and key highlights from the database.
- Responsiveness – The app must react quickly and should be able to cater to as many users as possible.

The application needs to be secured and lightweight. All the computation like fetching data from database, rendering content to the users, etc., should be performed on cloud. The cloud server should use different independent services to make the application more efficient.

Functional Requirements:

- The app should communicate with the services deployed on cloud for user searches, booking, payments and analysis of destinations like the total tickets booked on a day for a destination.
- The APIs to fetch the data should be running on cloud using container-services such as Dockers/Kubernetes.
- The app should be made up of loosely coupled modules so that a non-functioning module does not affect working of other modules.
- Testing of each module is required before integration to the cloud.
- For transaction and validation of the booking a dummy card 1111-1111-1111-1111 is to be used.

Job Roles:

Banner Id	Name	Tasks
B00832139	Gaurav Anand	1. Designed initial database schema to store app information and developed backend API for search functionality. 2. Implemented search component with cards and location info for mobile app. 3. Created Order History page for past orders in Android application. 4. Configured ECS, load balancers, auto scaling, services to deploy app on cloud.

B00822804	Akshay Reddy Poturi	1. Developed Login, Registration modules with 2-factor authentication for front end in web and Android application using the AWS Cognito. Implemented an API in the server to store the registration details of users in database. 2. Hosted the client application using Amazon S3 static hosting feature in cloud. Also, all the static data like images in search page of the application are stored in S3 bucket. 3. Configured API gateway in AWS for routing the API's between client and server applications. 4. Configured ECS, load balancers, auto scaling, services to deploy app on cloud.
B00834792	Shrey Rameshbhai Vaghela	1. Developed initial database schema to store the information about locations, users, buses, etc. 2. Developed booking, payment, analytics and invoice modules for android application. 3. Created docker images of Flask APIs and pushed it to the Elastic Container Registry (ECR).
B00822766	Shruthi Kalasapura Ramesh	1. Developed search, order details, analytics, booking and payment components in web application front-end. 2. Implemented Flask APIs for order details, analytics, booking and payment modules. 3. Modified database to make it compliant with the application requirements. 4. I was responsible for the unit testing of all the fragments that I implemented.

Knowledge Sharing:

- We had team meetings twice every week and discussed the progress in all the modules by sharing the knowledge of AWS services to all the team members.
- When the tasks assigned to any of the members were finished in any milestone, we shared the work with other team members to ensure all the milestones are finished as per our initial plan during the feasibility phase.

Completeness of Feasibility:

Development

- We developed client-side web and mobile applications for user interactions. Web application was built using react.js: a JavaScript library, and bootstrap: a CSS framework. We created Android mobile application to provide the same functionalities as web application to mobile users. Android Java was used to implement the mobile application.
- Server-side application was written in python Flask – a python-based web framework. We used the same back end for both web and mobile applications to provide the intended process flow.

- The following functionalities were successfully implemented on client-side as well as on server-side:
    a. End-to-end search functionality: to provide the user with all the available details about a place that the user searched for.
    b. Order details – to present a user with his/her past purchase history details such as order ID, purchase date, source, destination and number of passengers for each trip booked by the user.
    c. Analytics module was provided to help the users understand the trends in bookings like the total number of bookings to various tourist locations in the last few months.
    d. We implemented booking module and a payment gateway to enable the user book bus tickets for their trip and pay for the tickets they booked respectively.
    e. We implemented Registration/Login modules to help the users to register and login to the application.

Cloud services

- We made use of AWS Relational database service (RDS) to configure a relational database that was required for the project. We created MySQL database on RDS, and we could successfully store all the data related to the project. RDS offered data security with the use a virtual private cloud [1].
- We utilized Amazon EC2 instances within ECS for creating resizable and secure virtual servers to host the back-end application [2]. This provided efficient memory and network resources management and computational power for the application.
- Amazon Elastic Container Service (ECS), which is a container orchestration service, was used to manage docker containers [3]. We created microservices for various modules in the project, to make efficient use of resources and for reusability of application components, using ECS containers.
- We used Amazon Elastic Container Registry (ECR); a registry to store and manage docker container images [4]. The main reason to use ECS was it is secure, scalable and reliable. We created a private docker repository with ECR which was accessible only by the specified cluster (EC2 instances). Any modification to the images was made using Docker CLI. This ECR was associated with ECS we used.
- Amazon S3, we used number of buckets to store all the images used in the application, and one bucket for deploying client application.
- We employed AWS Cognito to build an access control system for our project. We integrated sign-in, sign-up modules of our application with AWS Cognito in order to manage a user directory on Cognito user pool and allow only authenticated users to access all the features of the application.
- As a result of our research on load balancing systems for cloud applications, we found out that the Application Load Balancer which is a subtype of Elastic Load Balancing (ELB) is ideal for request routing among microservices and containers since it allows dynamic port mapping for multiple requests of single service [5]. Hence, to implement a system for automatic load balancing, we devised application load balancer.
- Auto scaling is a part of cloud that manages the number of tasks handled by the server. We enabled autoscaling for ECS clusters by configuring auto scaling groups.
- In order to manage a plethora of concurrent API calls effectively, we used AWS API gateway as it supports containerized workloads and offers many useful services that are necessary for our application such as traffic management, cross origin request support etc.,

Final design:

Our application is a combination of different modules that work together to provide end-to-end functionality. The architecture that we followed to build the holistic application contains four major components.

Client-layer

This layer is the origin for all the processes. This is a front-end of the application that is operated by users. Client-layer for our application is divided into web application and mobile application. Client layer is mainly responsible for obtaining inputs from users and making appropriate Rest API request based on the users' action. We have hosted our web client-application on Amazon S3 bucket for convenience [6].

Middle layer

This layer aims to provide an interface between client-layer and server-layer. This layer mainly consists of two parts: AWS Cognito and API Gateway. As mentioned earlier, Cognito provides an authentication system for the application for security purpose and also to maintain a user base. The second part is API Gateway, and this manages the API calls and the relative routing to services running containers in the form of tasks.

Server-layer

This is considered as the paramount component of the entire application. We have the backend of our application hosted on AWS cloud platform. This layer manages all the cloud services used for deploying the application as well as the application specific functionalities: API calls in particular. Following are the services that we have used in this layer:

- Auto-Scaling Listeners – Listens to the number of requests/ API calls from the API Gateway. Manages the scaling of resources (EC2 instances) required to serve large number of requests coming to the server (containers in this case).
- Application Load Balancer – It is a type of Elastic Load Balancer to distribute the load of service requests amongst the available EC2 container instances.
- Elastic Container Service (ECS) – A service which manages a large number of containers running in the form of services. It orchestrates these containers on the available infrastructure for performance optimization.
- ECS Clusters – A cluster is a group of EC2 instances which act as infrastructure for running various independent services using ECS.
- Amazon S3 – A cloud storage space made available by Amazon AWS to store the static contents like images, videos, data, etc. Those static contents are called objects which can be retrieved through Access Points in the S3 bucket.

Data-layer

This layer contains a storage system to store and transfer data as requested by the components in Web Application Server Layer. We required a relational database to store the application related data such as user, trips, booking and payment information.

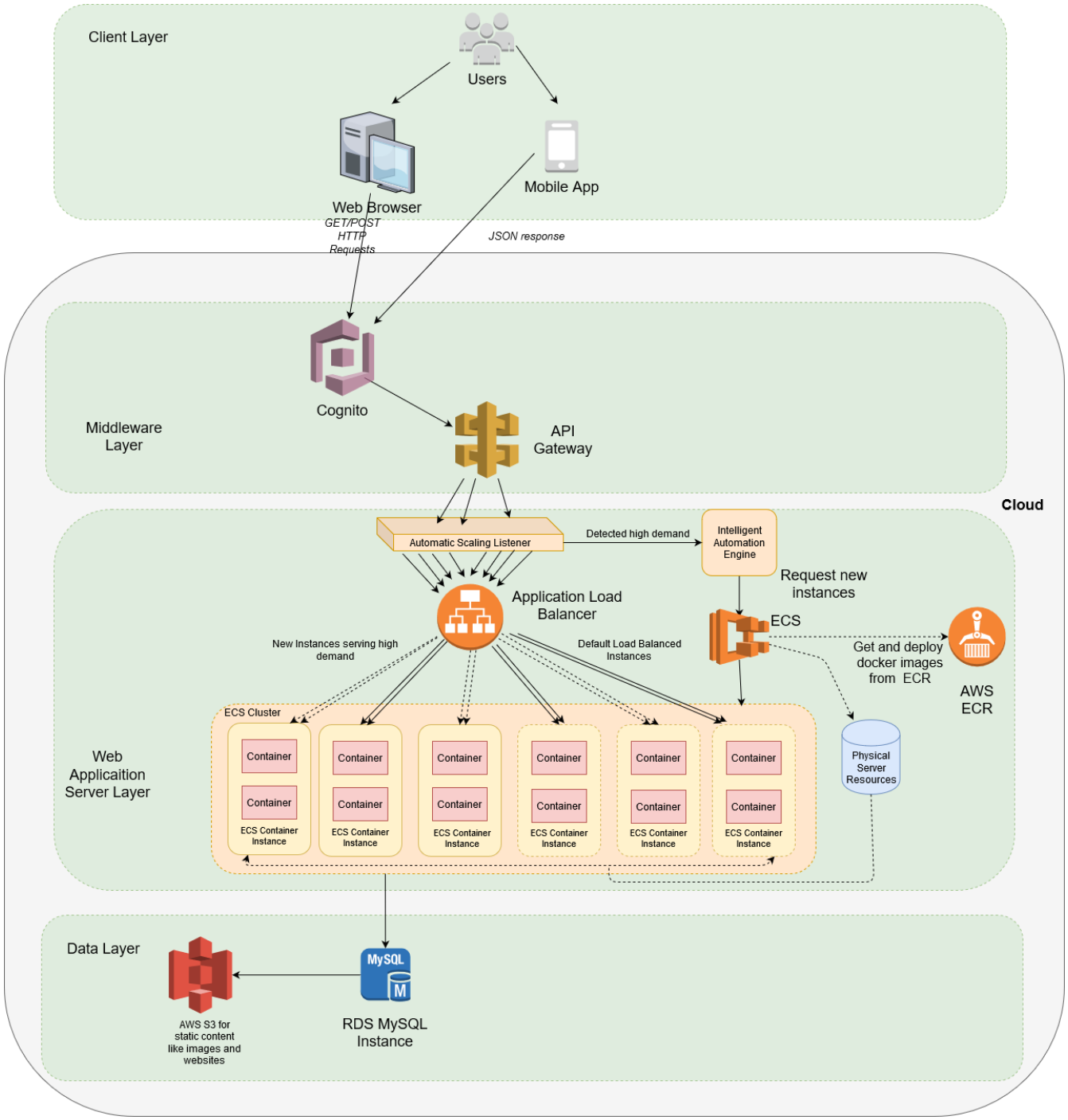

Figure 1 System Architecture

Implementation Details:

Milestones Achieved

- We were not able to finish the milestones as per the deadlines mentioned in the feasibility report. That is why, we had extended the deadlines for all the milestones and were able to finish them (milestone 1, milestone 2, milestone 3, milestone 4) successfully with all the tasks that were mentioned in the project feasibility report.
- We have successfully implemented all the functionalities as per the initial requirements in both client and server-side applications. Additionally, we also developed a service to help the users to see their past order details.

Challenges Faced and Solutions

- As there is no clear documentation about how to connect to Cognito for user management from android application, it was one of the challenges during the application development. We have spent a good amount of time for finding the various options from various sources to find a solution to use the AWS Cognito services from client application.
- We faced problems while connecting both mobile and web applications to a single backend server. But it became easy when we divided all our functionalities into loosely coupled APIs. After that, we could fire queries to each of the services quite easily.

Cloud Functionality implementation details

- Login and Registration modules were implemented using the AWS Cognito service. User session management is handled in the application using the “auth” method in “AWS-Amplify” library in client application.
- Multi-factor authentication is implemented in our application using the AWS Cognito. Once the user is verified with the given credentials (username and password), Cognito will send a one-time password to the user registered mobile number and user can provide the one-time password through client application which will be authenticated in AWS Cognito.
- The client application is deployed in AWS S3 bucket using the static website hosting option. This made the client application to be accessible to the users from any system with internet connection.
- Docker images of all the flask APIs were created and saved in repositories on Elastic Container Registry (ECR) which were linked with ECS to run those APIs on docker containers.

To maintain database for our cloud application, we decided to employ RDS (Relational Database Service) of AWS because the data for the application had a fixed schema which could be deployed on a relational database in the form of tables. RDS provides cost-effective, configurable capacity for an industry-standard database and manages usual database administration tasks [7]. Moreover, Amazon RDS provides backup and restore option along with an automatic failure detection which can notify the administrator in case of any trouble.

In order to make our RDS instance secure, we used Amazon VPC to host it into a Virtual Private Cloud. It created a virtual networking environment for our database where we could control access to our database using security groups. We defined the RDS instance with t3-medium instance type supporting 2 vCPU, and 4 GBs of RAM. We chose T3 instance because these are burstable instance types that provide a baseline level of CPU performance with an ability to burst CPU usage at any time for as long as required. We have

attached 100 GB general purpose SSD memory which supports three input/output operations per second per GB i.e. our database can withstand up to 300 I/O operations quite easily. To support more number of I/O operations, we could use provisioned SSD storage which supports up to 1000 requests per second. Our RDS instance can be auto scaled up to 1000 GBs. In addition, the database can retain backups for last 7 days with an option of automatic backups. Furthermore, all the error, general, and query logs can be displayed using CloudWatch which helps to monitor and track load and functioning of various AWS services. We gathered data for our application from external website sources for Canada tourism [8,9,10].

After developing the backend APIs for all the functionalities supported in our application, we decided to make use of Docker containers in order to run each of the functionality as an independent service. In addition, we wanted a container management service which could efficiently assign compute and memory resources to these services. For this, we chose ECS (Elastic Container Service) of AWS because it provides a highly scalable and fast container management service that makes it easy to run and manage Docker containers on a cluster of Amazon EC2 instances and thereby eliminating the need to worry about scaling management infrastructure.

To start with, we created Docker images for each of the six services and pushed them to ECR (Elastic Container Registry) which is a private Docker repository for ECS. It allows users to access saved repositories and images through Docker CLI to push, pull, and manage images. Next, we created an ECS cluster which is a group of EC2 instances managed by ECS. Initially, we defined six EC2 instances with t2.micro instance type, one for each of the service. Cluster creation automatically created an Auto Scaling group for our set of EC2 instances. Auto Scaling group ensures that there are enough instances to meet the application's desired capacity. Here, we defined desired number of instances as 6 and maximum number of instances as 12 to meet changing conditions in the future. It continues to maintain a minimum number of instances even if an instance becomes unhealthy. In case any instance goes down, it terminates the unhealthy instance and launches another one to replace it.

After pushing all the Docker images, we had to define a Task for each of the service. Each of the Task definition consists of at least one Docker image along with the CPU power, memory, and ports associated with the host. We used dynamic port mapping so that we could run multiple tasks from a single service. Moreover, this was the reason that we chose Application Load Balancer over other available options. Next, we went on to create an ELB (Elastic Load Balancer) which could distribute the incoming load among the available EC2 instances. We decided to create a load balancer for each of the service since our services' tasks were running on many EC2 instances, and we had to distribute load of each service among these tasks. Then, we created each of the six services defined by a corresponding task (Docker container). For each service, we specified minimum number of tasks running all the time as two. And, it could increase to maximum five in situation of high load for that specific service. With this, we were able to run all of our backend services in the form of tasks running on multiple EC2 instances which were scalable and supported traffic load balancing. Figure 2 shows how we fetched Docker images from ECR and executed containers in the form of tasks on multiple EC2 instances (a cluster). Here, ECS is mentioned as a task scheduler which orchestrates Docker containers (tasks) on the available infrastructure (cluster). Figure 3 shows an overview of composition of tasks, services, EC2 instances and finally a cluster.

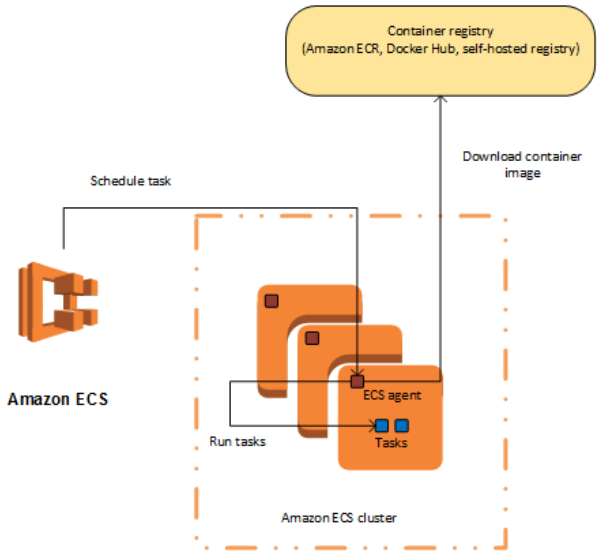

Figure 2 Design of our server module using ECS

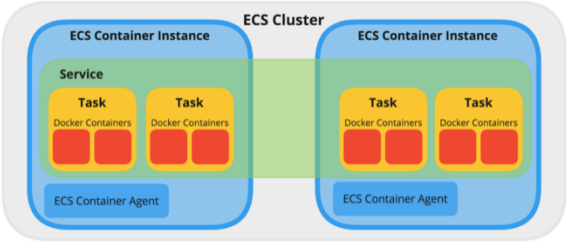

Figure 3 ECS Cluster

Testing:

- **Test case 1: Allow the users to login to application, only upon authenticating the user successfully.**
  As shown in the below screenshot, users will not able to sign into the application with incorrect credentials.

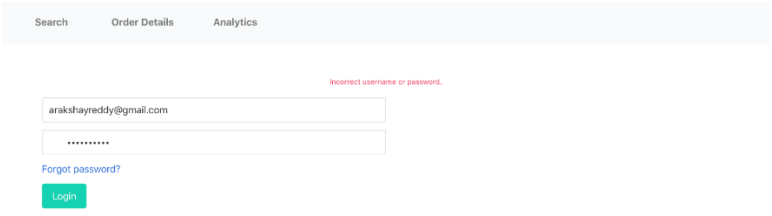

Figure 4 Testing Login Module

Upon entering the correct credentials, perform the multi factor authentication using one-time password to registered mobile number as shown below.

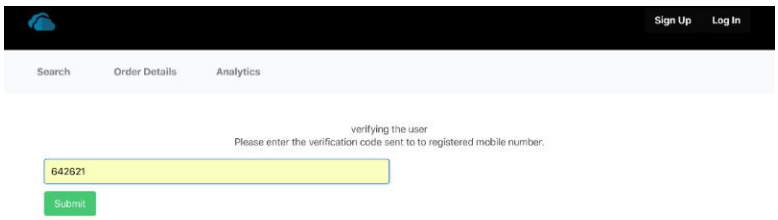

Figure 5 Testing Multi-factor Authentication

- **Test Case 2: Allow the users to see their booking history only upon signing into their account.** As shown in the below screenshot, if the user is not signed-in and trying to access the booking history, the application will display message asking the user to sign-in first.

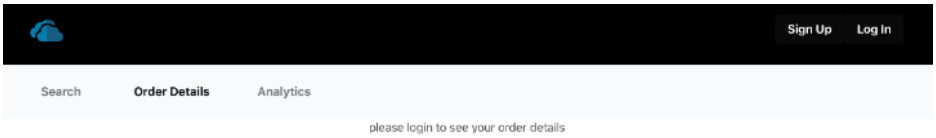

Figure 6 Testing integration of Booking and Login Module

Once the user is logged into the application, the recent order details of the user will be displayed as shown below.

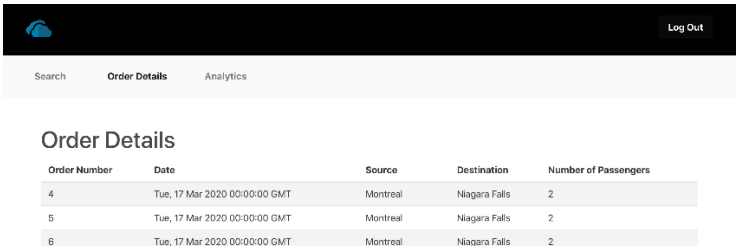

Figure 7 Testing integration of Order details and Login Modules

- **Test case 3: Allow users to search for a place only there is a search keyword**
The application throws an error if a user clicks on 'Search' button when no keyword is entered.

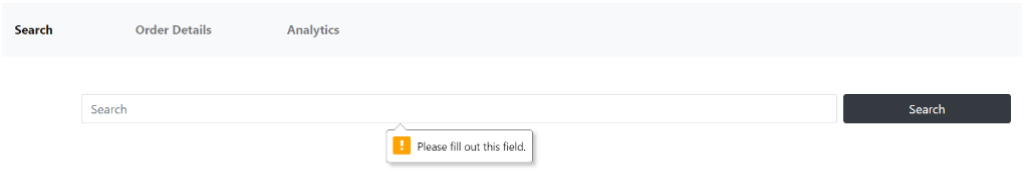

Figure 8 Tesing of Search Module

- **Test case 4: Acknowledge the user when there is no search result for a keyword.**
A pop comes up that states "No data found" to notify the user in case of absence of results.

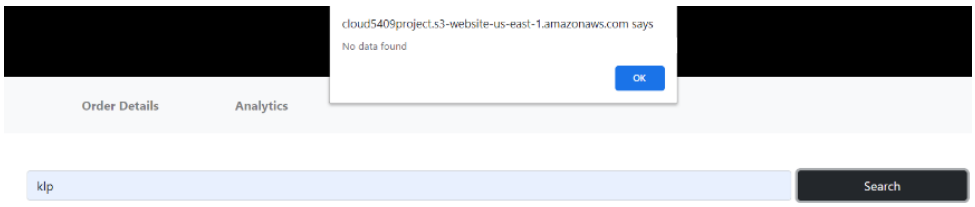

Figure 9 Testing data validation in Search Module

- **Test case 5: Do not allow a user to see buses unless a travel date is selected**
  The application throws an error if a user clicks on 'See buses' button when no date is entered.

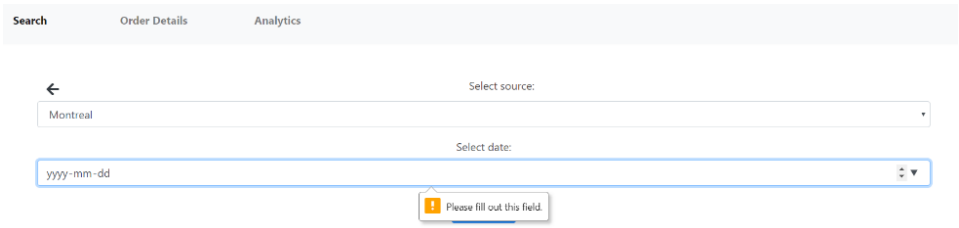

Figure 10 Testing data validation in Booking Module

- **Test case 6: Do not let the user select same source and destination to book a trip.**
  The list of source location gets filtered based the destination selected in the previous page.

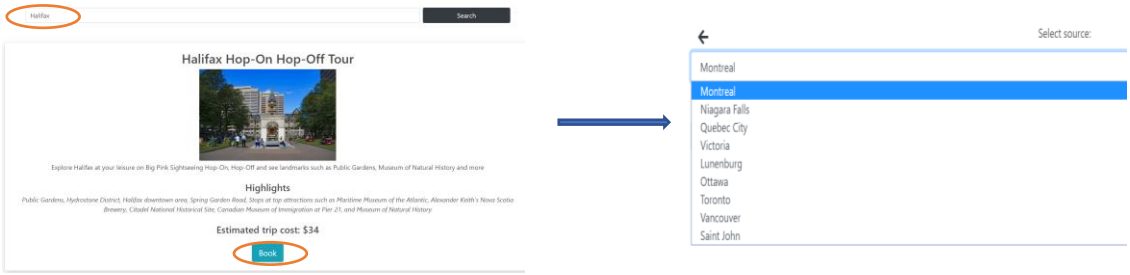

Figure 11 Testing data validation across Search and Booking Module

Limitations:

- Client-side android application does not provide an option for the user to save the invoice upon successful payment.
- Users have to begin the process of booking a trip starting from entering a keyword in case the payment fails.
- The card details must be same as following in order to make a successful payment. Any changes in the details or format will end up with payment failure
  - Card Name: Anything
  - Card Number: 1111111111111111 (16 digits)
  - Expiry date: 00/00
  - CVV: 999

Future Work:

Following are the functionalities or implementation changes that we intend to do as a part of our future work

- As the user base of the application will increase and spread across the globe, AWS CloudFront will help to accelerate the response of website and application by caching the response at edge locations server. To add more security, we will add field-level encryption on top of HTTPS connection to CloudFront.
- Developing user profile page in the application where the user can see their personal details and flexibility for the users to update their personal information.
- Currently, users can setup a new password for their account if the password is forgotten or lost using “forgot password” link in login page. We intend to extend this feature so that users should be able to change their password anytime to make it more convenient to the users using change password option in user profile page.
- RDS multi AZ [11] is a feature that we could not use for the current application, because of the cost associated with it, but can be implemented in future. RDS basically creates a primary instance for the current availability zone of database and concurrently updates instance replicas on other availability zones. This approach enhances the availability of data and thus efficiently manages the database workload.

References:

[1]"Amazon Relational Database Service (RDS) – AWS", *Amazon Web Services, Inc.*, 2020. [Online]. Available: https://aws.amazon.com/rds/. [Accessed: 29- Mar- 2020].

[2]"Amazon EC2", *Amazon Web Services, Inc.*, 2020. [Online]. Available: https://aws.amazon.com/ec2/. [Accessed: 29- Mar- 2020].

[3]"Amazon ECS - Run containerized applications in production", *Amazon Web Services, Inc.*, 2020. [Online]. Available: https://aws.amazon.com/ecs/. [Accessed: 29- Mar- 2020].

[4]"Amazon ECR | Amazon Web Services", *Amazon Web Services, Inc.*, 2020. [Online]. Available: https://aws.amazon.com/ecr/. [Accessed: 29- Mar- 2020].

[5]"Elastic Load Balancing - Amazon Web Services", *Amazon Web Services, Inc.*, 2020. [Online]. Available: https://aws.amazon.com/elasticloadbalancing/. [Accessed: 29- Mar- 2020].

[6]"Deploy ReactJS App with S3 Static Hosting", *Medium*, 2020. [Online]. Available: https://medium.com/serverlessguru/deploy-reactjs-app-with-s3-static-hosting-f640cb49d7e6. [Accessed: 29- Mar- 2020].

[7]"What Is Amazon Relational Database Service (Amazon RDS)? - Amazon Relational Database Service", *Docs.aws.amazon.com*, 2020. [Online]. Available: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html. [Accessed: 29- Mar- 2020].

[8] 2020. [Online]. Available: https://www.canada.travel/. [Accessed: 29- Mar- 2020].

[9]"Tourism in Canada", En.wikipedia.org, 2020. [Online]. Available: https://en.wikipedia.org/wiki/Tourism_in_Canada. [Accessed: 29- Mar- 2020].

[10]"Book Canada Tours & Things to do in Canada - View All Tours - Gray Line", Grayline.com, 2020. [Online]. Available: https://www.grayline.com/things-to-do/canada/. [Accessed: 29- Mar- 2020].

[11]"Amazon RDS Multi-AZ Deployments", *Amazon Web Services, Inc.*, 2020. [Online]. Available: https://aws.amazon.com/rds/features/multi-az/. [Accessed: 29- Mar- 2020].