Spider: A new way to Bus

[GitHub Link](https://github.com/DevonARP/CC_Project) | [Videos](https://drive.google.com/drive/u/0/folders/12NxYKBSCeZ-dKPdQfswyUpWD44rI8Tdu)

Anil Poonai Sakshat Katyarmal Jayvardhan Patyal

[Ap52254@nyu.edu](mailto:Ap52254@nyu.edu) [sk9428@nyu.edu](mailto:sk9428@nyu.edu) [jp6207@nyu.edu](mailto:jp6207@nyu.edu)

**Abstract**

The main idea behind out application is to have a bus system that goes to every state, most buses will just stay one part of the country or get the coasts and not the middle part of the country. We want spider to access every part of the country. We also included a streaming service for users to access movies from, this allows us to have entertainment for our passengers that other buses won’t have.

**RDS**

We used a relational database from MySQL as most of the data being acquired and created is transactional. We have a predefined schema for everything and most of the queries involve complex joins with medium velocity needed.

**EC2 and Frontend**

We used an EC2 instance to host our application as it is very scalable and has high speed and performance as well as let’s us have cheaper costs while having more server-side processing abilities than an S3 website hosting bucket. We also used the Flask framework to design the frontend as it keeps the application as simple as possible for users to use as well as keeping it scalable and lightweight.

**Cognito**

We used Cognito for sign-up and authentication for our video streaming service. This gave us an easy and secure system for our streaming service. Regarding our actual bus system, we designed our own authentication system for flexibility. The credentials for both the bus and streaming service do not have to be the same for each user.

**S3**

We decided to use an S3 bucket to hold out streaming service content, this made it easy to manage and store as well as left it highly interactable for CloudFront and kept the content safe.

**CloudFront**

We used CloudFront as a CDN for our streaming content, this allowed it to be cheap and be streamed in real time as well as keeping it secure and flexible.

**API Gateway**

We used API Gateway to keep delivery simple and give us a lot of flexibility. This made it easy to combine API calls for data retrieval and update.

**Lambdas**

We used serverless functions from AWS called Lambdas to just focus on the code that API calls will run. This allowed us to minimize costs, have scalability, and have less operational management involved.

**SES**

We used SES to verify email addresses and send confirmation emails of bookings and refunds. You don’t need to verify your email to use the application, but it is a more enjoyable experience if it is verified.

**QuickSight**

We used QuickSight for an easy and quick visualization of our data. It is connected to our RDS and is very good as a self-service analysis tool, highly scalable and flexible, and can be accessed anywhere.

**EventBridge**

We used EventBridge to update the lambda functions easily. This lets us update lambdas as a reaction to events and on fixed timeframe.

**MapCrow**

We gathered our location data including the locations and distances from each other with the MapCrow API, this allowed us easy and cheap access to the data.

**Features:**

Sign-in/ Registration: To use the app/website users need to login via email. It will be authorized by the data in RDS. If the user is not registered, they need to register and confirm their verification for SES to send them a confirmation email.

Booking: Users can book their tickets by specifying their start and end location and the number of tickets required.

Profile Update: In case the user needs to update their profile details, they can update it and RDS will update it as well.

Payments: For any transaction, the user needs to add a credit card beforehand in the payments section of the website. They can remove/add multiple cards or update the existing one.

Travel History: Their past trips and transactional values will be stored in a separate RDS database.

Refunds: Any future trips can be cancelled, and refund will be issued to customers.

Rewards: For any trip the users will get some rewards, depending on the range of their transaction. These rewards can be redeemed in future trips to get some discount.

**Architecture and API’s:**

Chart, diagram, box and whisker chart

Description automatically generated

The following API’s have been used In our implementation:

Addpayment: Creates payment in database

Bookingid: Information regarding that booking

Createbooking: Creates booking in database

get-price: Price of trip

get-values: Finds if credentials are authorized

pasttrips: Grabs past trips of user

payment: Grabs payment options of user

payments: Grabs payment options for removal

removecard: Removes card from database

post-data: Adds a new profile to the database

profile: Grabs user information

profileupdate: Updates user information

refund: Refunds booking

trip: Finds upcoming trips for planned locations

upcoming: Grabs user’s upcoming trips

Implementation:

**Implementation:**

The hosting of our static website is done on EC2 instance and Flask was used to implement the frontend of our project. The RDS database stores the user data including email and password. The user needs to login to access the app, SES will send a verification email to users. Once the user logs in, they can plan a trip, edit a future trip, add payment methods, or stream entertainment. We used Cognito for sign-up and authentication for our video streaming service. For the streaming part CloudFront is used as a CDN. The trip locations and distances are fetched from MapCrow to give an accurate cost per trip per person. Whenever a trip is planned EventBridge updates the lambda functions to store new/updated values in RDS. We used QuickSight to analyze the data in RDS to get a better understanding of users and how to improve the website.

**Future Scope**

* Add more online rewards to obtain
* Add more locations to choose from
* More content than just streaming videos for entertainment
* Implement Elasticsearch for faster searches
* Better UI and UX
* Possibly add an API key for security