

Travlr Getaways Full Stack Web Application

# CS 465 Project Software Design Document

Version 1.0

## Table of Contents

[CS 465 Project Software Design Document 1](#_Toc185569327)

[Table of Contents 2](#_Toc185569328)

[Document Revision History 2](#_Toc185569329)

[Executive Summary 3](#_Toc185569330)

[Design Constraints 3](#_Toc185569331)

[System Architecture View 3](#_Toc185569332)

[Component Diagram 3](#_Toc185569333)

[Sequence Diagram 5](#_Toc185569334)

[Class Diagram 6](#_Toc185569335)

[API Endpoints 7](#_Toc185569336)

[The User Interface 8](#_Toc185569337)

UI Summary 11

## [Document Revision History](#_heading=h.lnxbz9)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 0.1 | 07/16/2023 | Kennedy U | Completed comprehensive summary and design constraints review. |
| 0.2 | 07/17/2023 | Kennedy U | Sketch a diagram for the system architecture design view (wireframe) |
| 0.3 | 07/26/2023 | Kennedy U | Implemented API endpoints, draw a sequence diagram and explanation. |
| 0.4 | 07/29/2023 | Kennedy U | Refactored, created new class and diagram with explanation. |
| 0.5 | 08/16/2023 | Kennedy U | Implemented additional API endpoints, user interfaces, updated functionality, and adjusted design constraints. |

## [Executive Summary](#_heading=h.35nkun2)

The full stack web application will be built using the MEAN stack (MongoDB, Express.js, Angular, and Node.js). The web application will have static customer site frontend, displaying content for the site, and a customer admin frontend for managing the web application. Node.js will serve as the application runtime.

The static web application frontend will be built using Express.js, Handlebars, and MongoDB. Express.js is a server-side web application framework for Node.js. MongoDB together with Handlebars will be used to generate the static content that Express.js can serve. Handlebars is a templating language that will be used to dynamically populate a static page with content. MongoDB is a NoSQL (non-relational) database that will be used to store static content to be displayed in the Handlebars templates.

The admin frontend web application will be a single page application (SPA) and will use Angular (SPA) framework that will allow users to perform different kinds of administrative operations. The landing page will require the user to login and authenticate before continuing. Once logged in and authenticated, the users will be able to manage the content of the web application. For example, perform add, update, or remove content via APIs that route through the static side of the web application. This will be achieved by updating MongoDB’s documents via the Angular admin portal.

## [Design Constraints](#_heading=h.1ksv4uv)

The are several design constraints to consider when developing a web application using the MEAN technology stack. First, it needs to be easy to update the content on the static portion of the site without redeploying the entire website. The design will have to be responsive, providing good user experience regardless of the device the app is running on. The admin portal needs to be a single page application (SPA) that can be used to dynamically display the different components needed to let the admin manage the web application.0

While MongoDB is the suited database in the MEAN stack, it can suffer performance issues when the dataset becomes too large. Express, Angular, and Node.js work well together to complete the MEAN tech stack but can sometimes result in large codebase that could become difficult to maintain. Note: in cases where an existing site is transformed to use the MEAN technology stack, it can become difficult to revert these changes due to application dependencies, or stack trace.

## [System Architecture View](#_heading=h.44sinio)

### Component Diagram

The web application is composed of three major components: the Client component, the Server component, and the Database component. Each of these components are made up of their own parts. The Client component contains components for client sessions, the web browser, site portfolio, and graphic library. The web browser and graphic library provide an interface. The client session and site portfolio require the web browser interface. The travelr portfolio also requires the Graphic library interface, and the interface provided by the database component. The client session interacts with a port on the client component to connect to the required server component interface.

The database component has a single component for MongoDB, which provides an interface to the client and server components.

Relationships Description:

The Client Session established between the web browser and the server enables communication and data exchange between the client-side and server-side components.

The Traveler Portfolio component on the client side interacts with the Graphic Library to visualize the portfolio data retrieved from the MongoDB database.

The Server Session on the server side interacts with the Traveler Database to access and modify the traveler's data.

The Mongoose ODM acts as an intermediary between the Server Session and the MongoDB database, facilitating efficient data querying and manipulation.



Here is a description of the significant components and their relationships:

**Client-Side Components:**

**Client- client session**

Client Session <-> Web Browser -> Traveler Portfolio

Traveler Portfolio <-> Graphic Library -> MongoDB

***Client Session****:* Represents the session established between the client (web browser) and the server. It maintains the state and context of the client's interaction with the application.

***Web Browser***: The web browser serves as the user interface for the application. It displays the Traveler Portfolio, which is the graphical representation of the traveler's portfolio data.

***Traveler Portfolio***: This component utilizes a graphic library to render and display the traveler's portfolio. It retrieves data from the MongoDB database to populate the portfolio with relevant information.

**Server-Side Components:**

**Server**

Authentication Server <-> Client Session

Server Session <-> Traveler Database

Server Session <-> Mongoose ODM -> MongoDB

***Authentication Server***: Handles user authentication and authorization. It validates client sessions and ensures secure access to protected resources.

***Server Session***: Represents the session established between the server and the client. It maintains the state and context of the server's interaction with the client, including the traveler's database information.

***Traveler Database***: Stores the traveler's data, including their portfolio information. The server session interacts with the traveler database to fetch, modify, or update the relevant data.

***Mongoose ODM***: Stands for Object-Document Mapping and provides a higher-level interface for interacting with MongoDB. It allows the server session to communicate with MongoDB efficiently, providing data retrieval and manipulation capabilities.

**Database –** MongoDB

### Sequence Diagram

The sequence of the operation starts with the actor, in this case the user. The user enters a route in the web browser from a computer, then gets directed to one of the views/templates of the site by the front-end router. The view interacts with the corresponding front-end controller, which will populate the template, render, and return the view information to be displayed to the user. The front-end controller makes calls to functions within the HTTP service to retrieve pieces of information. The results, callbacks, and promises of the functions are passed back to the controller.

The HTTP service connects the frontend to the backend by making the API calls to specific routes. The router on the backend receives the route from the frontend, exchange information, then calls the appropriate backend controller. Once initiated/called by backend router, the backend controller makes a call to the database using Mongoose. The controller takes the returned data and passes the result back to the calling front-end http service. Finally, MongoDB as the database receives the query from the backend controller, processes the request, and returns the result.

How Backend and Frontend Controllers Work Together

Following a **single workflow**:

1. **Frontend Controller**: The user fills out a trip form and clicks "Submit." The Angular addTrip method sends a POST request to /api/trips with the trip data.
2. **Backend Middleware**: On the server, getUser middleware validates the user’s JWT token to ensure they are authenticated.
3. **Backend Controller**: The tripsAddTrip function receives the request, validates the data, and inserts it into the database.
4. **Backend Response**: If the trip is successfully added, the backend sends a 201 Created response with the trip details.
5. **Frontend Controller**: The Angular addTrip method processes the response and updates the UI with the newly added trip.

A diagram with different colored squares

Description automatically generated

If the backend controller needs to retrieve or store data in the database, it communicates with MongoDB by making database queries using Mongoose models, MongoDB ORM. The backend controller handles the database response and prepares the data to be returned to the client.

Throughout this process, the data flows from the client to the server, and database and back to the client, enabling a dynamic and interactive web application experience. The controller acts as an intermediary, handling the data processing and communication between the client, server, and database.

## Class Diagram

The CruiseInfo, FlightInfo, and HotelInfo classes all contain a name property and other fields that are unique to each mode of travel. Each one also inherits the TripInfo class which contains properties for the start and return date, as well as origin and destination locations. CruiseBooking, FlightBooking, and HotelBooking each have an association with their corresponding Info class and the TravellerInfo class. There are zero-to-many relationships between the Booking classes and the TravelAgent class in both directions. The TravelAgent class has also has associations with the CruiseInfo, FlightInfo, HotelInfo, and TravellerInfo classes and a one-to-many relationship with the MembershipAdmin class. The TravellerInfo class inherits the MemberAccount class. The MemberAdmin class has an aggregate relationship with the MemberAccount class. The Itinerary class has an aggregate relationship with the CruiseInfo, FlightInfo, and HotelInfo classes.

A screenshot of a computer screen

Description automatically generated

## [API](#_heading=h.2jxsxqh) Endpoints

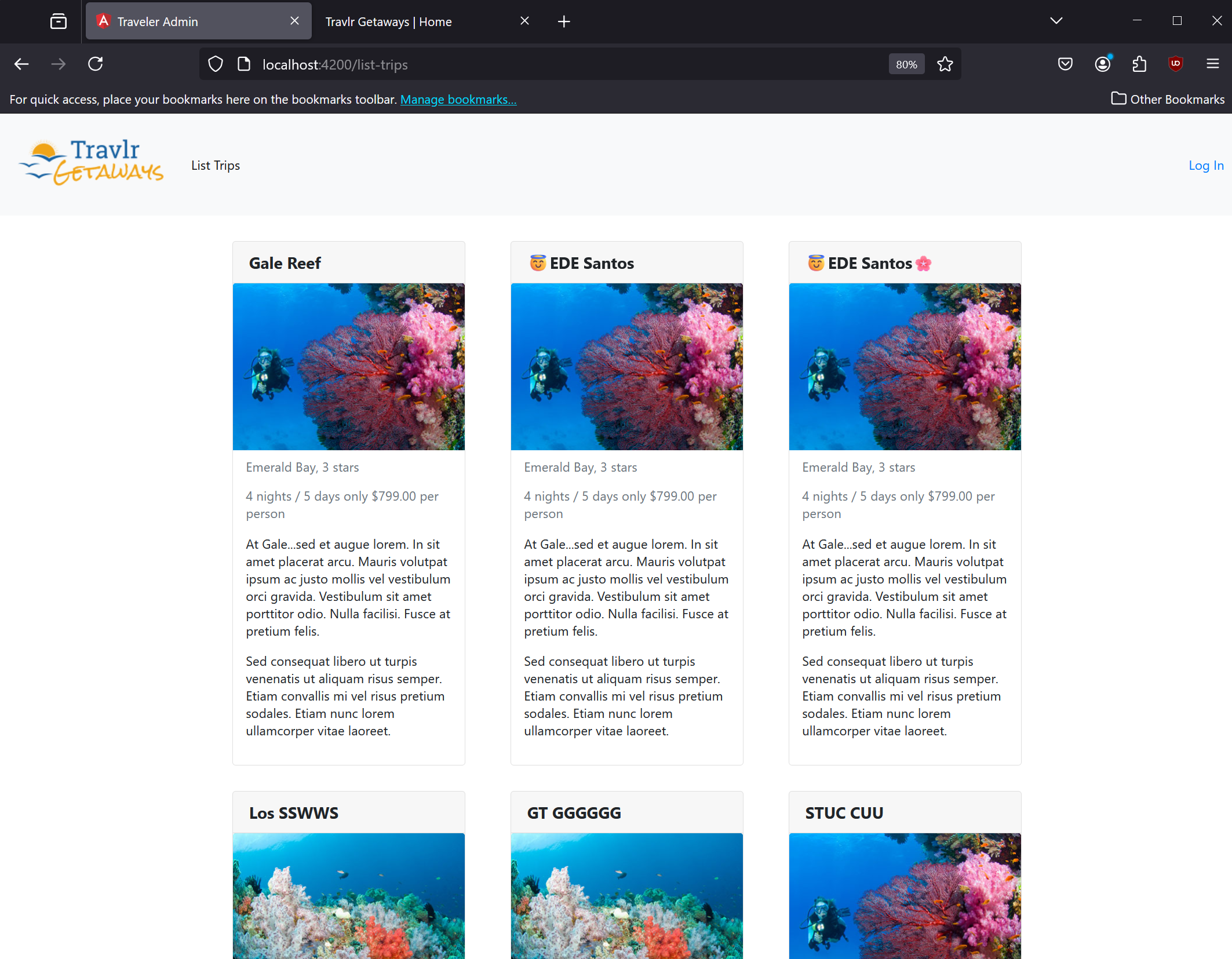
Exposing RESTful endpoints is a design approach to enable an application to participate in a larger ecosystem. Here is a document showing each endpoint in the table below, including the HTTP method, purpose, URL, and notes.

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **POST** | Login a user | /api/login | Authenticates a user and returns a JWT |
| **POST** | Register a user | /api/register | Add a new user to the database and returns a JWT |
| **GET** | Retrieve list of meals | /api/meals | Returns all meals |
| **GET** | Retrieve single piece of meal | /api/meals/:mealsCode | Returns single meal, identified by the meal code at end of URL |
| **GET** | Retrieve list of news | /api/news | Returns all news content |
| **GET** | Retrieve a single piece of news content | /api/rooms/:newsCode | Returns single news piece, identified by the news code at the end of URL |
| **GET** | Retrieve list of rooms | /api/room | Returns all rooms |
| **GET** | Retrieve single room | </api/rooms/:roomCode | Returns single room, identified by the news code at the end of URL |
| **GET** | Retrieve list of trips | /api/trips | Returns all trips |
| **POST** | Add a trip | /api/trips | Add a new trip to the database |
| **GET** | Retrieve a single trip | /api/trips/:tripCode | Returns single trip, identified by the trip code at the end of URL |
| **PUT** | Update single trip | /api/trips/:tripCode | Updates single trip, identified by the trip code at the end of URL |
| **DELETE** | Delete single trip | /api/trips/:tripCode | Deletes single trip, identified by the trip code at the end of URL |



## The User Interface

**Trip Listing screen**

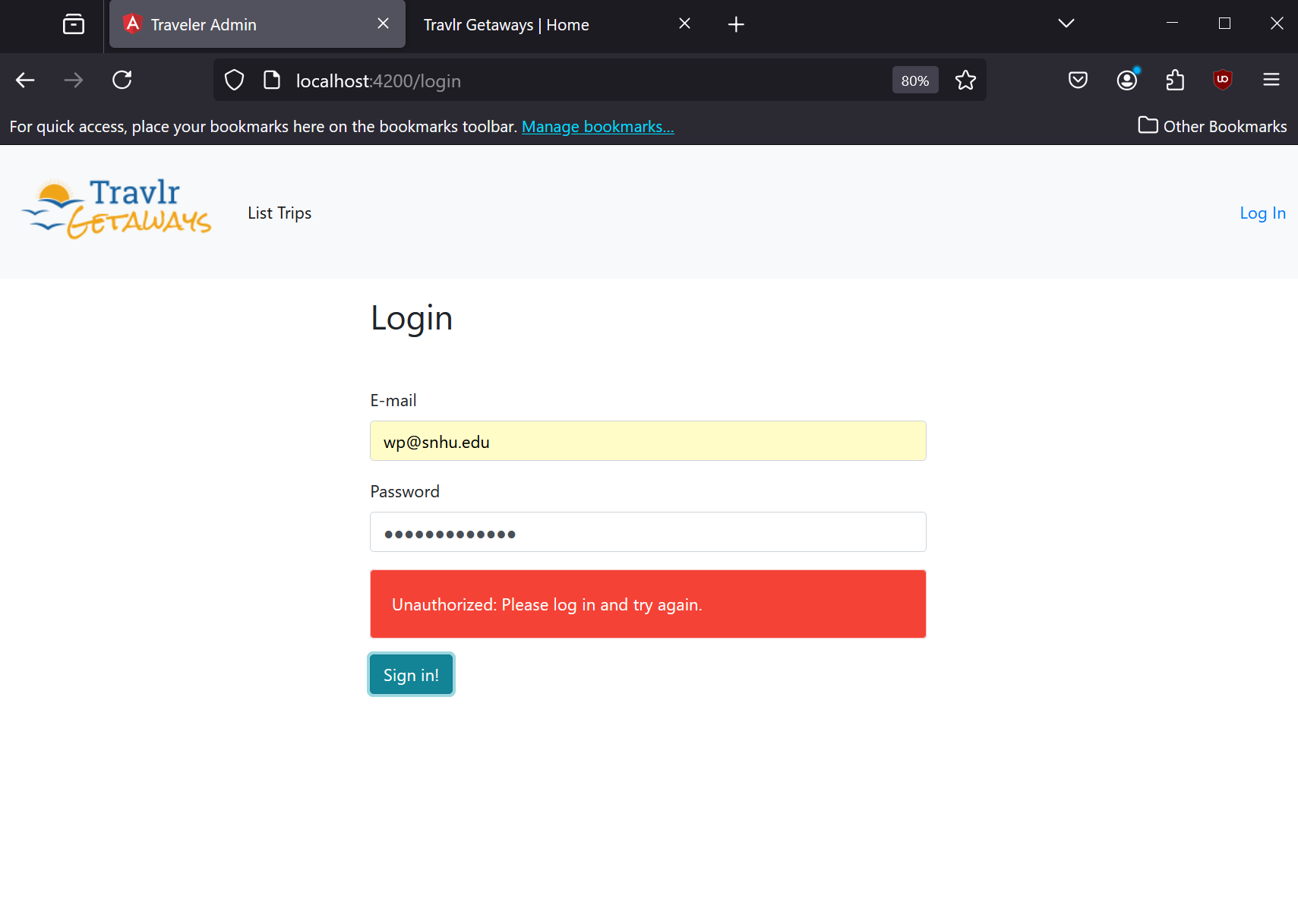


**Login**

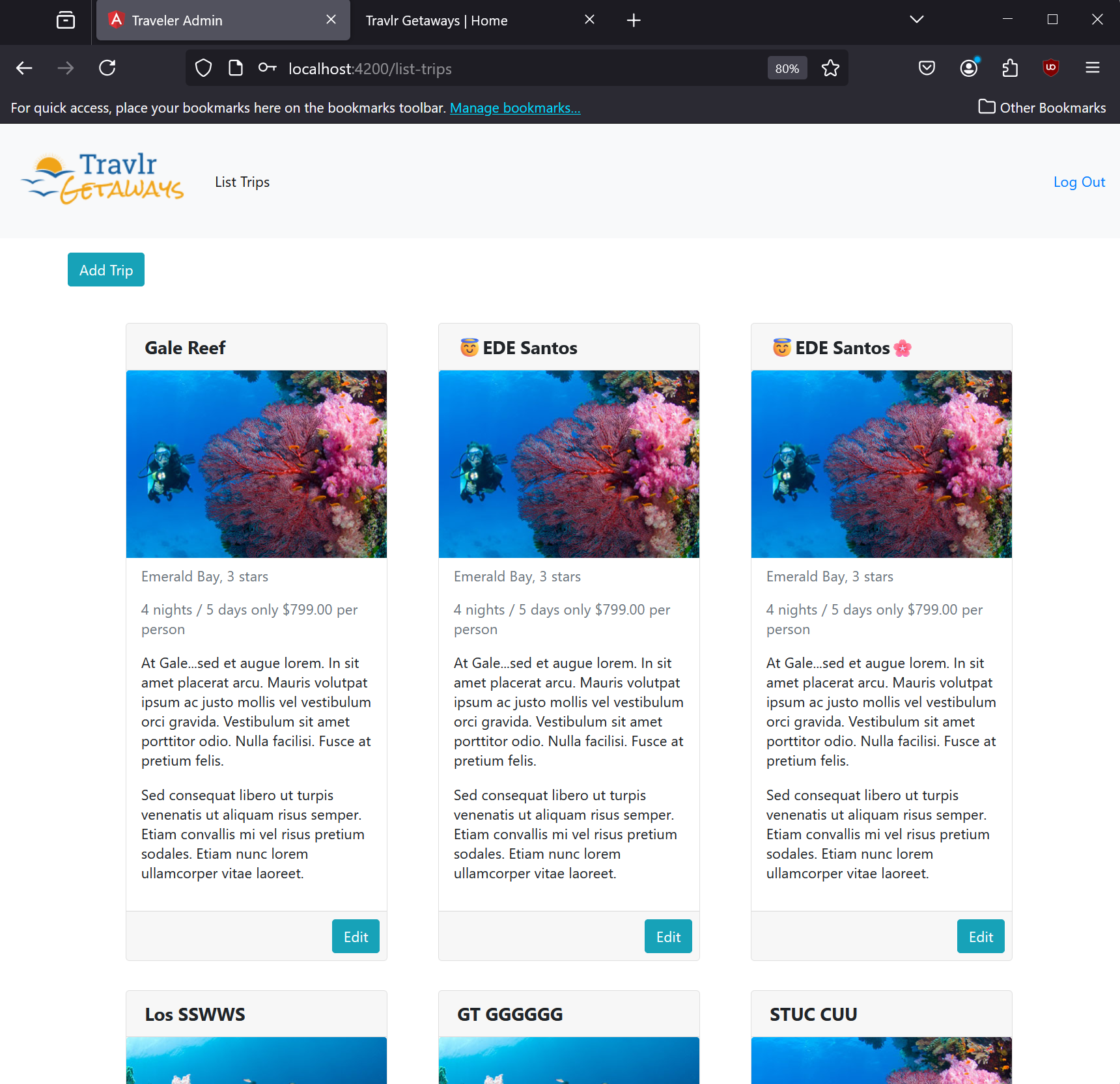
A screenshot of a computer

Description automatically generated

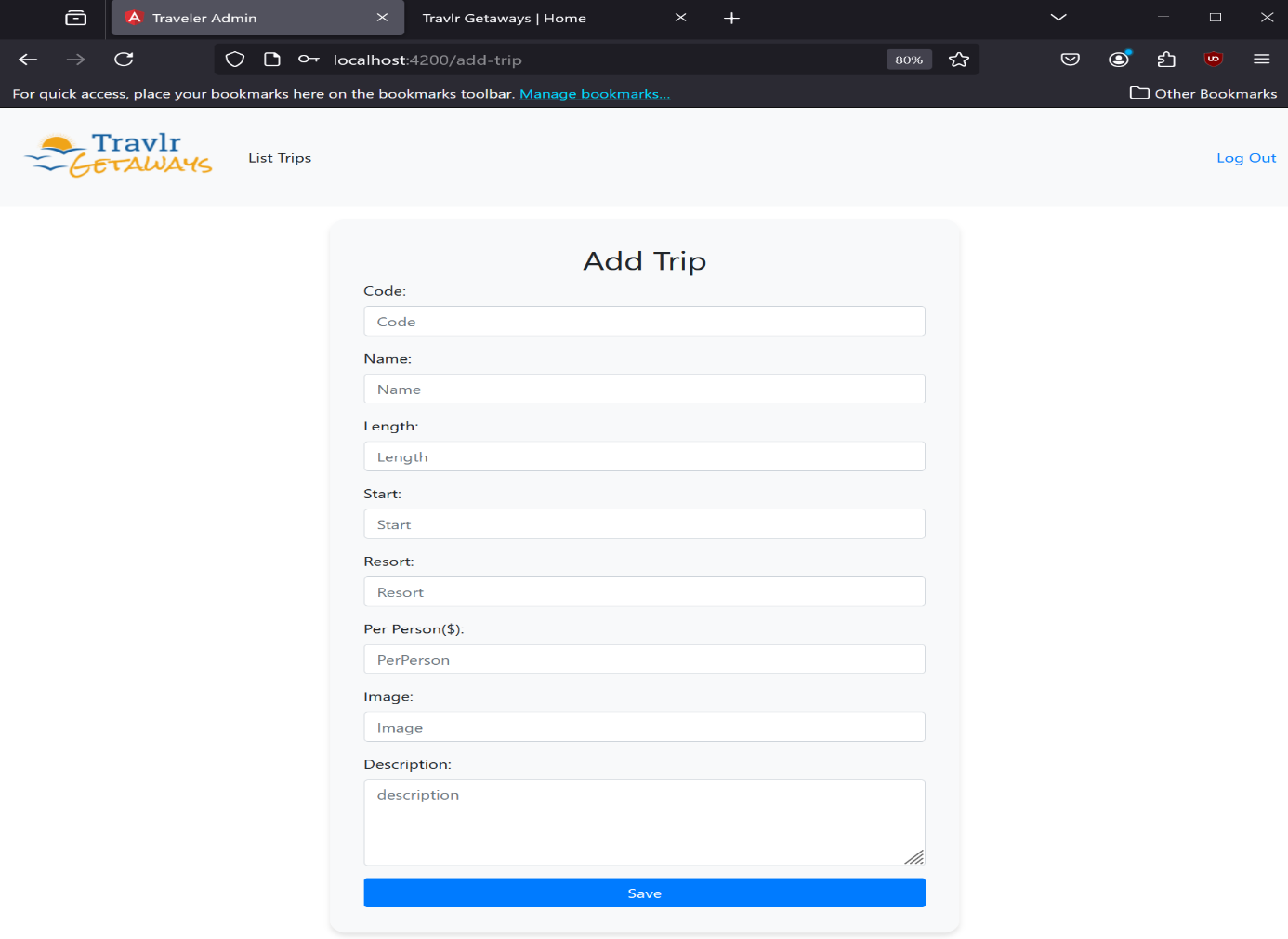
**Auth**

****

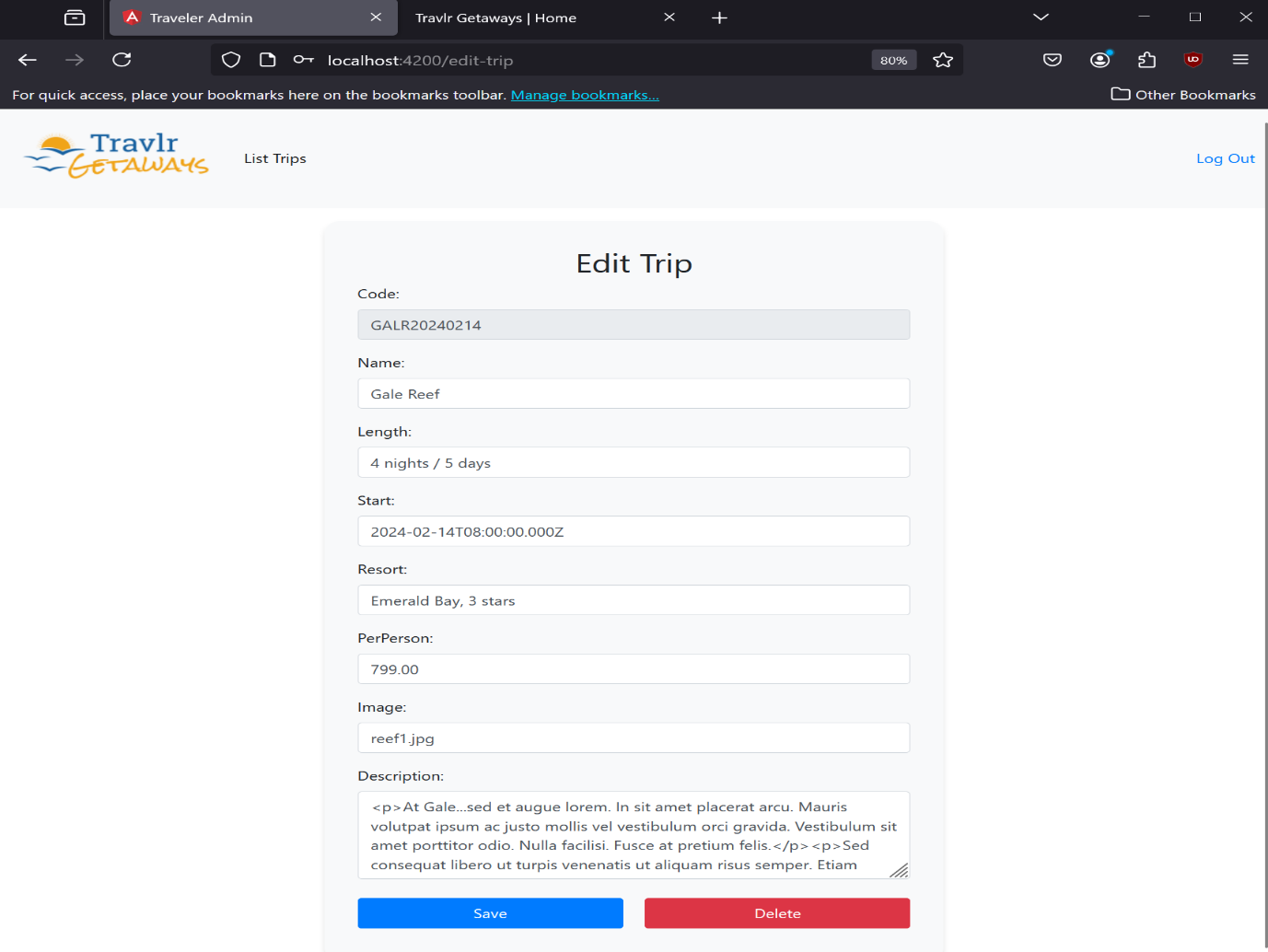
**Logged In**

****

**Add Trip**



**Edit Trip**

****

**UI Summary**

Angular is a front-end framework with the views rendered on the client side, whereas Express is a backend framework with the views rendered on the server side and sent to the client. The Angular project is made up of models, services, routes, and components. The Express site is made up of views, controllers, and page content on the client. Angular uses reusable components to make up parts of the site. Express uses a templating engine, in this case handlebars, to dynamically generate content on the server before sending to the client. Both use APIs to retrieve or send data.

Some advantages of the SPA functionality include:

* Reduced server load due to only needing to send the initial page.
* SPAs can create very interactive sites with lots of functionality.
* Faster user experience due to elimination of full page reloads and only necessary data is retrieved from the server.

Some disadvantages of SPA functionality include:

* A long initial load time due to needing to retrieve the entire Javascript application.
* They can be difficult to optimize for SEO.

Additional SPA functionality of simple web applications include:

* Client-side routing which removes the need for additional server requests and provides a smooth transition between views.
* SPAs have the ability to offer offline support after the initial page load.

**Testing to make sure the SPA API calls to GET and PUT data in the database works as expected.**

One of the ways to test to make sure that the SPA API calls GET and PUT data in the database works as expected is to use the admin site as reference. The GET API calls can be tested by loading the trip-list view. If trips show up, then the API GET call worked to retrieve the trips from the database. To test the PUT API call, a trip would need to be added to the database via the addTrip form and schema. Once the form is open, filled and saved, it should display the new trip info on trip-list view. If it displays, then that’s evidence that the PUT API call worked correctly to insert a trip data in the database. The Express server application backend has to be running for the Angular SPA to successfully make server-database communication, because without the API controllers running on the server, there is no way for the SPA to communicate with database.