

<Name of Software Application>

# **CS 465 Project Software Design Document**

Version 1.0

## Table of Contents

[**CS 465 Project Software Design Document** 1](#_Toc36198462)

[Table of Contents 2](#_Toc36198463)

[Document Revision History 2](#_Toc36198464)

[Instructions 2](#_Toc36198465)

[Executive Summary 3](#_Toc36198466)

[Design Constraints 3](#_Toc36198467)

[System Architecture View 3](#_Toc36198468)

[Component Diagram 3](#_Toc36198469)

[Sequence Diagram 4](#_Toc36198470)

[Class Diagram 4](#_Toc36198471)

[API Endpoints 4](#_Toc36198472)

[The User Interface 4](#_Toc36198473)

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

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | <10/20/24> | <Nadia Baamrani> |  |

## [Executive Summary](https://docs.google.com/document/d/1paf8ED8obf0qFbz3_2REn4NMWXXaHNWrAmvpaIf3oKs/edit#heading=h.2jxsxqh)

<Describe the appropriate architecture of the web application based on your client’s software requirements. Be sure to reference your use of the MEAN stack for development. Explain both the customer-facing side of the application and the administrator single-page application (SPA).>

The architecture for the proposed web application is based on the MEAN stack, comprising **MongoDB**, **Express.js**, **Angular**, and **Node.js**. This full-stack JavaScript solution offers seamless communication between the customer-facing side and the administrator's single-page application (SPA), allowing for efficient development, scalability, and performance.

**Customer-Facing Side:** The customer-facing side will be served directly from the Node.js server, utilizing Express.js as the backend framework. All customer interactions, such as browsing, searching, and interacting with data, will be handled through RESTful APIs connecting the front and back end.

**Administrator Single-Page Application (SPA):** The admin panel will be a Single-Page Application (SPA) developed using Angular. This approach ensures that administrators can manage the platform seamlessly, with real-time updates and fast navigation between different sections without reloading the entire page.

**Frontend (Angular):** The SPA will leverage **Angular's** modularity and two-way data binding for real-time interactions, allowing the admin to update, manage, and track data efficiently.

**Backend (Node.js):** Similar to the customer-facing side, **Node.js** will handle backend logic, exposing APIs specifically designed for administrative functionalities such as user management, content updates, and analytics.

**Database (MongoDB):** **MongoDB** will serve as the centralized database for both the customer-facing side and the admin side, ensuring that all data is consistent and up-to-date across both interfaces.

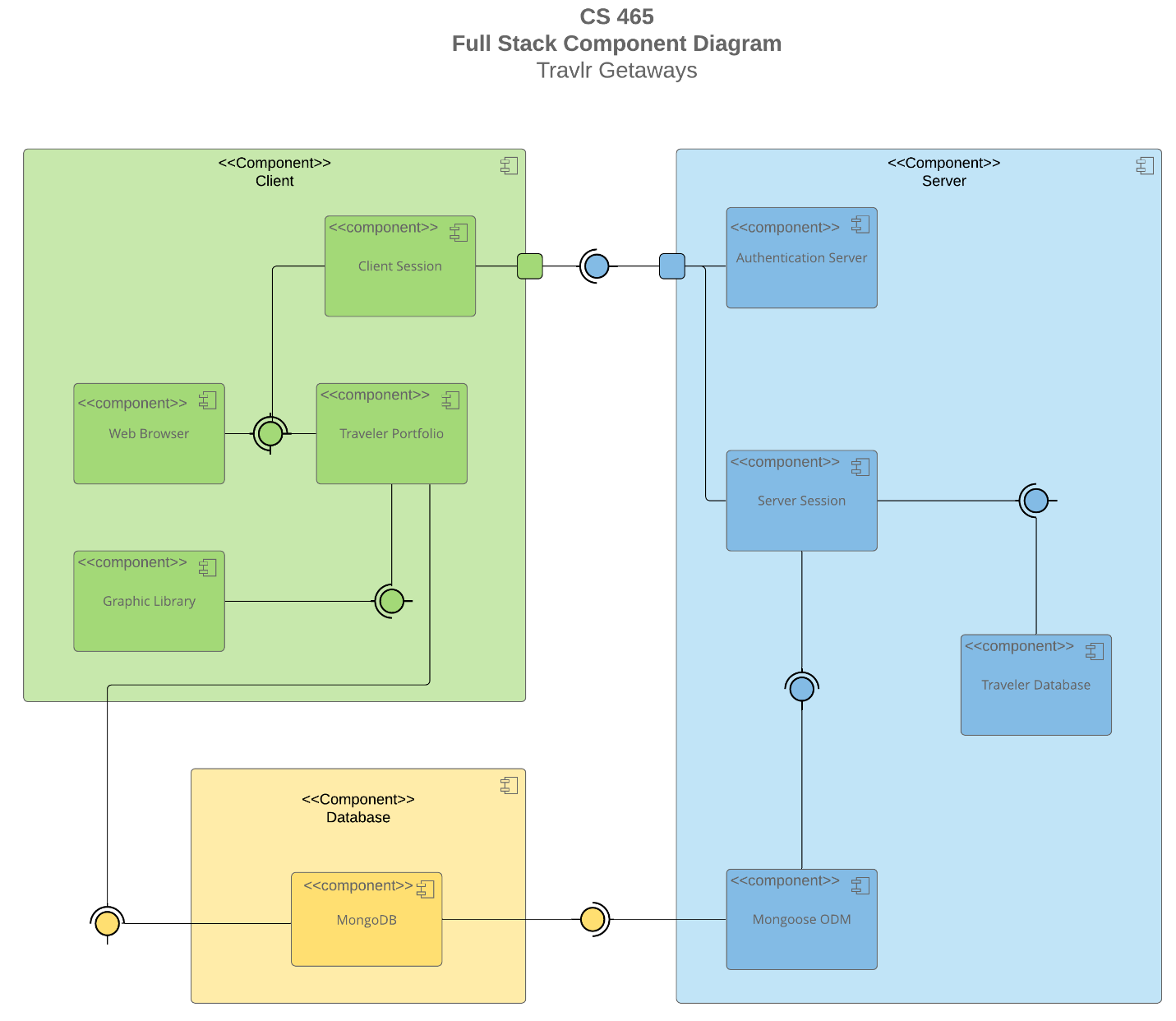
## [Design Constraints](https://docs.google.com/document/d/1paf8ED8obf0qFbz3_2REn4NMWXXaHNWrAmvpaIf3oKs/edit#heading=h.z337ya)

<Identify the design constraints for developing the web-based Travlr Getaways application. Explain the implications of the design constraints on the application development.>

The **Travlr Getaways** application must be scalable to handle growing users, with a responsive design for mobile devices. Data security is critical, requiring encrypted communication and secure storage. Performance optimization through caching and lazy loading ensures fast interactions, while a user-friendly interface provides ease of navigation for both travelers and admins. Offline access with limited functionality via Progressive Web App (PWA) features enhances usability. Integration with third-party APIs like payment gateways and SEO optimization for organic traffic is essential, impacting both the technical implementation and user experience.

## [System Architecture View](https://docs.google.com/document/d/1paf8ED8obf0qFbz3_2REn4NMWXXaHNWrAmvpaIf3oKs/edit#heading=h.3j2qqm3)

### Component Diagram



A text version of the component diagram is available: [CS 465 Full Stack Component Diagram Text Version](https://learn.snhu.edu/d2l/lor/viewer/view.d2l?ou=6606&loIdentId=24342).

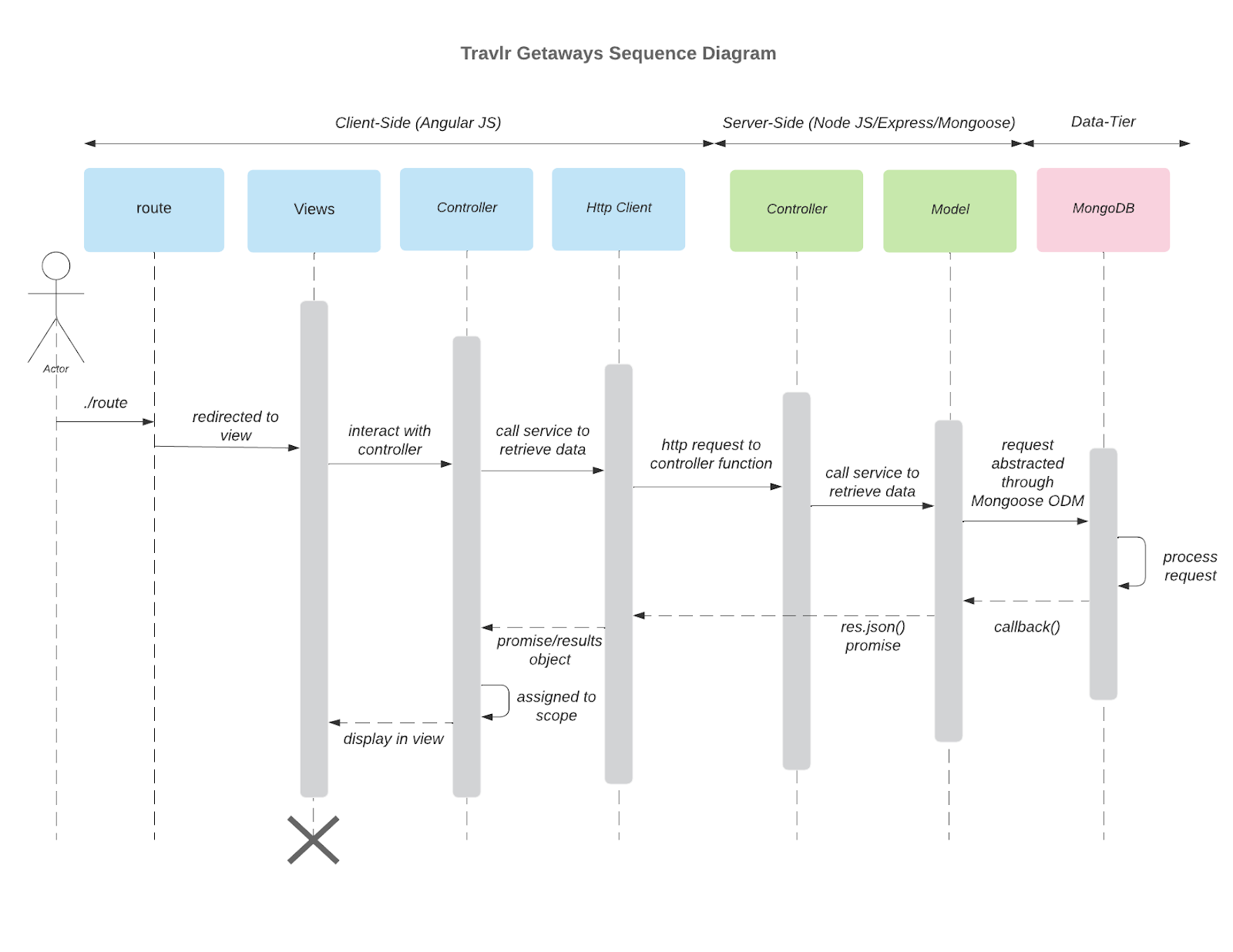
The component diagram outlines a three-layer architecture: **Client**, **Server**, and **Database**.

* **Client Layer**: Includes the web browser, client session, traveler portfolio, and graphic library for user interactions and visual rendering. The client communicates with the server via network requests for authentication and data management.
* **Server Layer**: This layer contains the authentication server, server session, and traveler database, and it manages user sessions, authentication, and data retrieval.
* **Database Layer**: Uses MongoDB and Mongoose ODM for data storage and mapping. The server interacts with the database for data access and persistence.

The system ensures secure communication between layers.

### Sequence Diagram

<Illustrate the flow of logic in a web application by completing a sequence diagram. Insert an image of the sequence diagram here.>



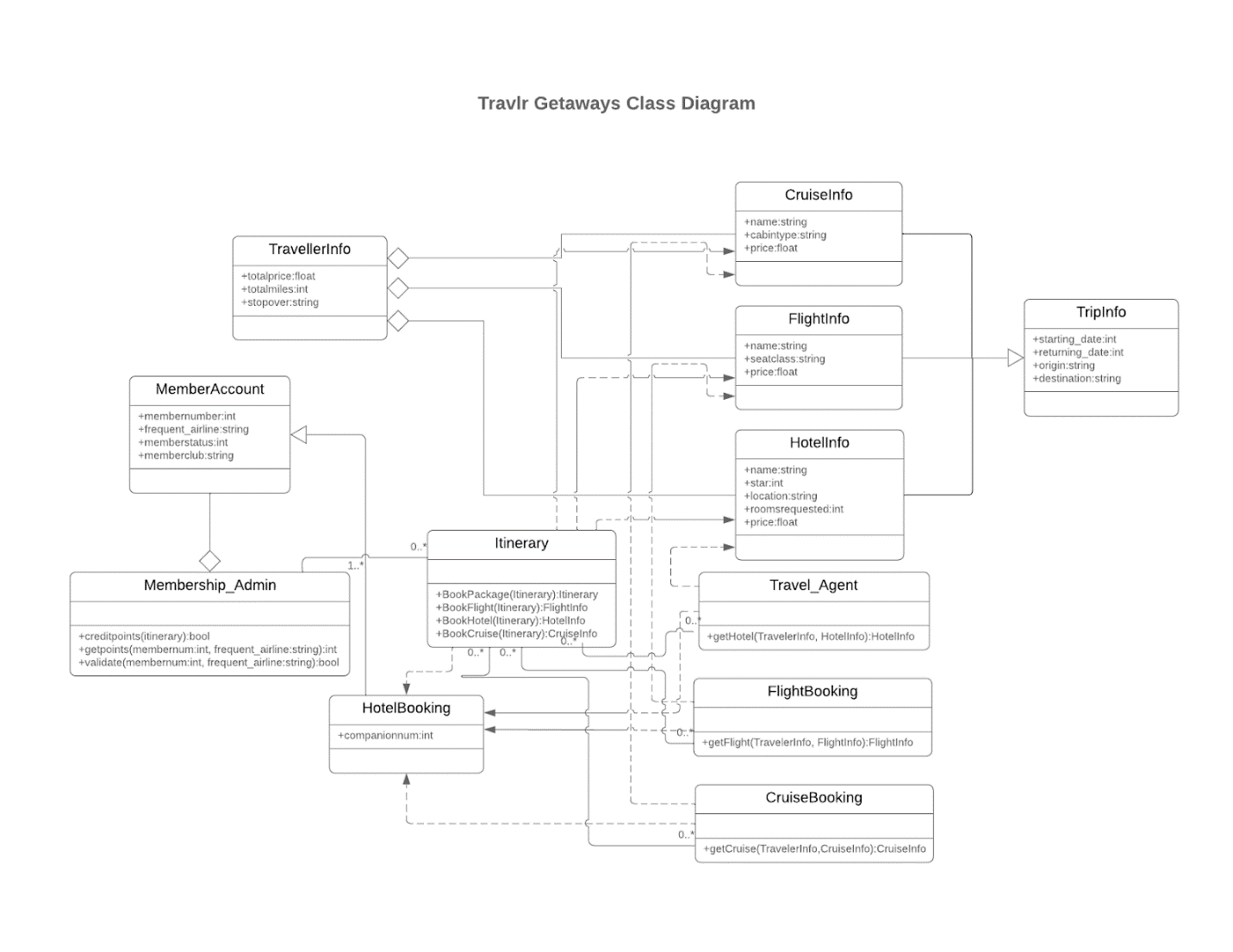
<Describe the flow of logic in the web application based on the sequence diagram. Be sure to describe the interactions between the layers, or tiers, of the full-stack application. Including significant processes such as Sign In, Trips, and Admin interactions will be helpful when referring to the sequence diagram.>

1. **Client-Side (AngularJS)** section:
   * Route: Can remain labeled as ./route.
   * Browser/View/Template: Label the action "redirected to view" as View.
   * Controller: Label "interact with controller" as Controller.
2. **Server-Side (NodeJS/Express/Mongoose)** section:
   * HTTP Client: Label the "http request to controller function" as HTTP Client.
   * Controller: Label "call service to retrieve data" as Controller.
   * Model: Label the part where the service interacts with MongoDB via Mongoose (request abstraction and callback) as Model.
3. **Data-Tier (MongoDB)** section:
   * MongoDB: Label the action "request abstracted through Mongoose ODM" as MongoDB.

Each label can be used as described above and may be reused when the same concept is referenced more than once in the diagram.

## Class Diagram

<Illustrate the JavaScript classes of the web application by completing a class diagram for the web application. Insert an image of the class diagram here.>



<Describe the JavaScript classes of the web application based on the class diagram.>

#### MemberAccount

#### JavaScript

#### class MemberAccount {

#### constructor(memberNumber, frequentAirline, memberStatus, memberClub) {

#### this.memberNumber = memberNumber;

#### this.frequentAirline = frequentAirline;

#### this.memberStatus = memberStatus;

#### this.memberClub = memberClub;

#### }

#### // Example method to update member information

#### updateMemberInfo(newAirline, newStatus, newClub) {

#### this.frequentAirline = newAirline;

#### this.memberStatus = newStatus;

#### this.memberClub = newClub;

#### }

#### }

#### TravelerInfo

#### JavaScript

#### class TravelerInfo {

#### constructor(name, contactInfo, specialRequests) {

#### this.name = name;

#### this.contactInfo = contactInfo;

#### this.specialRequests = specialRequests;

#### }

#### // Example method to update traveler's contact information

#### updateContactInfo(newContactInfo) {

#### this.contactInfo = newContactInfo;

#### }

#### }

#### TripInfo

#### JavaScript

#### class TripInfo {

#### constructor(startingDate, returningDate,  origin, destination, tripDuration) {

#### this.startingDate = startingDate;

#### this.returningDate = returningDate;

#### this.origin = origin;

#### this.destination = destination;

#### this.tripDuration = tripDuration;

#### }

#### // Example method to calculate trip duration

#### calculateTripDuration() {

#### // Logic to calculate duration based on start and end dates

#### }

#### }

#### FlightInfo

#### JavaScript

#### class FlightInfo {

#### constructor(name, seatClass, price) {

#### this.name = name; // This could be the flight number or airline code

#### this.seatClass = seatClass;

#### this.price = price;

#### }

#### }

#### HotelInfo

#### JavaScript

#### class HotelInfo {

#### constructor(name, star, location, price) {

#### this.name = name;

#### this.star = star;

#### this.location = location;

#### this.price = price;

#### }

#### }

#### . CruiseInfo

#### JavaScript

#### class CruiseInfo {

#### constructor(name, cabinType, price) {

#### this.name = name;

#### this.cabinType = cabinType;

#### this.price = price;

#### }

#### }

#### Itinerary

#### JavaScript

#### class Itinerary {

#### constructor(tripInfo, travelerInfo) { // Assuming a 1-to-1 relationship with TripInfo and TravelerInfo

#### this.tripInfo = tripInfo;

#### this.travelerInfo = travelerInfo;

#### this.flights = []; // Array to store FlightInfo objects

#### this.hotels = []; // Array to store HotelInfo objects

#### this.cruises = []; // Array to store CruiseInfo objects

#### }

#### addFlight(flight) {

#### this.flights.push(flight);

#### }

#### addHotel(hotel) {

#### this.hotels.push(hotel);

#### }

#### addCruise(cruise) {

#### this.cruises.push(cruise);

#### }

#### // Example method to calculate total trip cost

#### calculateTotalCost() {

#### // Logic to sum up costs from flights, hotels, cruises

#### }

#### }

#### Travel\_Agent

#### JavaScript

#### class Travel\_Agent {

#### constructor() {

#### // Could potentially have properties to store agent information

#### }

#### // Example methods to interact with other classes and perform booking logic

#### bookFlight(travelerInfo, flightInfo) {

#### // Logic to book a flight

#### }

#### bookHotel(travelerInfo, hotelInfo) {

#### // Logic to book a hotel

#### }

#### bookCruise(travelerInfo, cruiseInfo) {

#### // Logic to book a cruise

#### }

#### }

#### Membership\_Admin

#### JavaScript

#### class Membership\_Admin {

#### constructor() {

#### // Could have properties related to membership management

#### }

#### // Example methods to manage member accounts

#### validateMember(memberNumber, frequentAirline) {

#### // Logic to validate member credentials

#### }

#### updateMemberPoints(memberNumber, points) {

#### // Logic to update member's reward points

#### }

#### }

#### Relationships: The code uses arrays in the Itinerary class to represent the "many" side of the relationships with FlightInfo, HotelInfo, and CruiseInfo.

#### Methods: I've included some example methods to give you an idea, but you'll need to add more based on the specific functionality of your application.

#### Data Validation: Implement appropriate data validation within the constructors and methods to ensure data integrity.

#### Error Handling: Include error handling to gracefully manage unexpected situations.

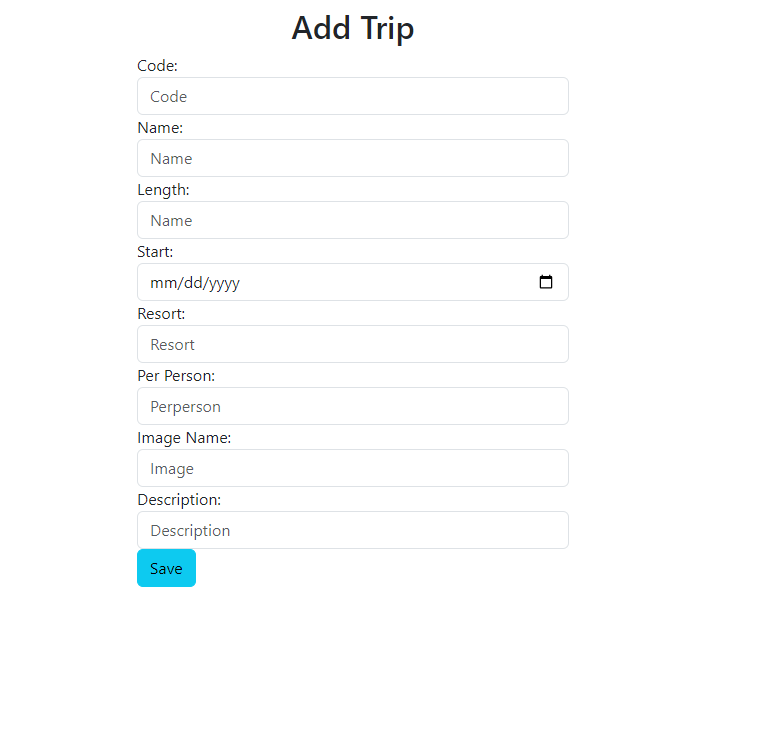
#### UI Interaction: You'll need to write additional JavaScript code to handle user interactions, data input, and display of information on the web page.

## [API](https://docs.google.com/document/d/1paf8ED8obf0qFbz3_2REn4NMWXXaHNWrAmvpaIf3oKs/edit#heading=h.1y810tw) Endpoints

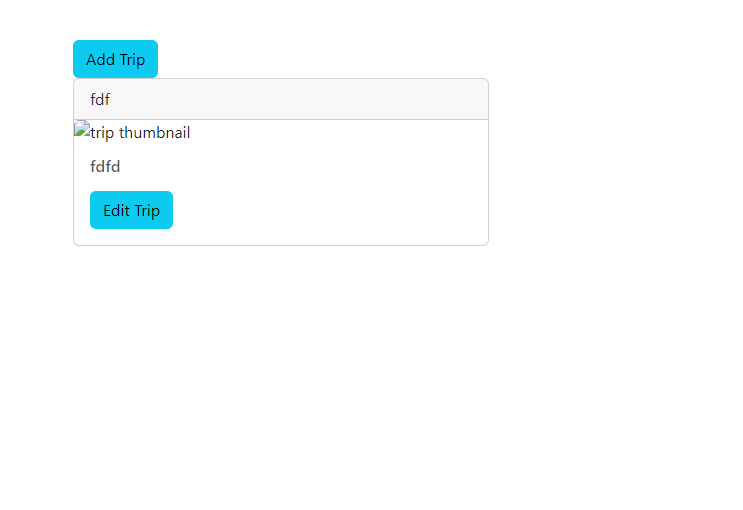
<Exposing RESTful endpoints is a design approach to enable an application to participate in a larger ecosystem. Document each endpoint in the table below, including the HTTP method, purpose, URL, and notes.>

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve List of the Data then user can select see all the data for each trips | /api/trips | Retrieve all the trip data for the page |
| **GET** | Retrieve Single Trip for Edit the user | /api/trips/:tripCode | Retrieve single trip data then the user can edit or view that trip |
| **POST** | Save the new Trip | /api/trips/ | user able to insert new trip for the system |
| **PUT** | Update the trip | /api/trips/: code | Update the trip data |

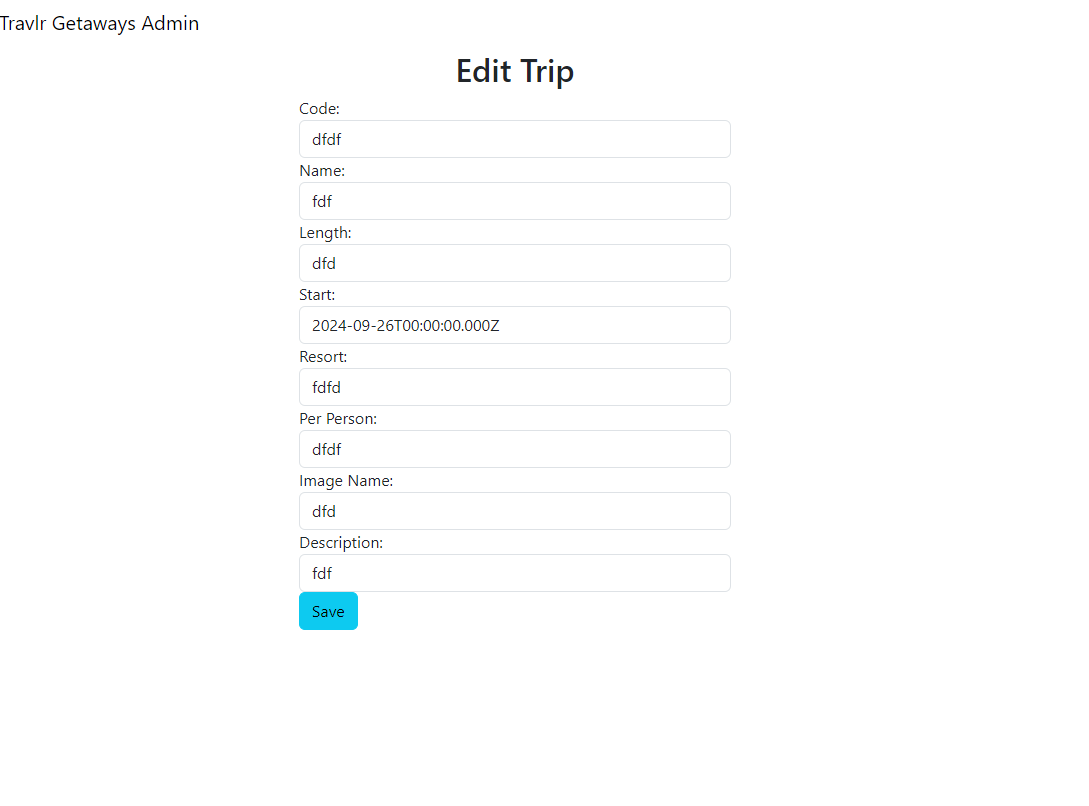
## The User Interface

Here is the interface for the Add Trip page   


Here is the interface for the list of the Trip data



Update Trip data



### Angular Project Structure:

An **Angular** project is structured around modules, components, and services:

* **Modules**: Group-related components and services (e.g., AppModule).
* **Components**: Define the UI with associated HTML, CSS, and TypeScript logic.
* **Services**: Handle business logic, including API calls using HTTP requests.
* **Routing**: Manages navigation between components in a Single Page Application (SPA).

### Express Project Structure:

An **Express** project has a simpler, backend-oriented structure:

* **Routes**: Define endpoints for handling HTTP requests.
* **Controllers**: Implement logic for specific routes.
* **Middleware**: Process requests before they reach the routes.
* **Models**: Interact with the database, usually with the help of an Object-Relational Mapper (ORM) like Mongoose.

### Comparison:

Angular builds **single-page applications (SPA)**, providing a rich, interactive user experience by loading content dynamically without refreshing the page. It manages front-end logic and user interaction. In contrast, **Express** is backend-focused, serving data to the front end and managing server-side logic.

### Testing SPA and API Interaction:

To ensure the SPA works with the **Express API**:

1. **Setup Mock API**: Use tools like Postman to test API endpoints (GET and PUT requests) independently.
2. **Component Testing**: In Angular, use tools like **Jasmine/Karma** to test that components make HTTP requests correctly.
3. **Integration Testing**: Ensure Angular’s service is properly fetching (GET) and updating (PUT) data from the Express API using HTTPClient. Verify response handling and error management.
4. **E2E Testing**: Tools like **Protractor** or **Cypress** test the entire interaction between the Angular front-end and Express backend, ensuring smooth communication with the database.