

Web Application

# **CS 465 Project Software Design Document**

Version 1.0

## Table of Contents

[**CS 465 Project Software Design Document** 1](#_Toc206104634)

[Table of Contents 2](#_Toc206104635)

[Document Revision History 2](#_Toc206104636)

[Executive Summary 3](#_Toc206104637)

[Design Constraints 3](#_Toc206104638)

[System Architecture View 4](#_Toc206104639)

[Component Diagram 4](#_Toc206104640)

[Sequence Diagram 8](#_Toc206104641)

[Class Diagram 9](#_Toc206104642)

[API Endpoints 11](#_Toc206104643)

[The User Interface 11](#_Toc206104644)

[Contrasting Angular Project Structure Against Express Project Structure 11](#_Toc206104645)

[Testing 12](#_Toc206104646)

[Adding a Unique Trip: 13](#_Toc206104647)

[Deleting a Trip: 16](#_Toc206104648)

[Edit a Trip: 17](#_Toc206104649)

[Express Renders from Mongo: 20](#_Toc206104650)

[User Registration: 22](#_Toc206104651)

[User Login: 22](#_Toc206104652)

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

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 7/16/2025 | Nicholas Kreuziger | Updated sections Executive Summary, Design Constraints, and System Architecture |
| 1.1 | 7/31/2025 | Nicholas Kreuziger | Updated Sequence Diagram, Class Diagram and API Endpoints section |
| 1.2 | 8/14/2025 | Nicholas Kreuziger | Added finishing touches, finished up User Interface Section. Updated API Endpoints section. |

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

Travlr Getaways requires a robust web application to showcase its offerings for Travel, Lodging, and Food in an intuitive interface, alongside administrative tools for efficient management and analytics. We propose building this solution using the MEAN (MongoDB, Express.js, Angular, Node.js) stack. The MEAN stack allows us to build a responsive and scalable application that can handle increasing user traffic as Travlr Getaways grows, ensuring a smooth experience for your customers. The customer-facing side will provide browsing and booking capabilities, while a separate administrator Single Page Application (SPA) provides a secure and efficient interface for managing travel packages and analyzing key business data, empowering you to make informed decisions.

* **Customer-facing website:** This website informs customers about the Travlr Getaways offering for Travel, Lodging and Food in a clean and intuitive interface.
* **Administrative Pages:** The access-controlled administrative page will provide administrative tools and analytics access. This will make getting information from the application centralized, allowing administrators to answer business questions with ease.
* **Scalable:** Leveraging the MEAN stack, we will program the Web Application with Scalability in mind. Data will be cleanly organized, and the Website will be modular, enabling the application to grow as the business grows.

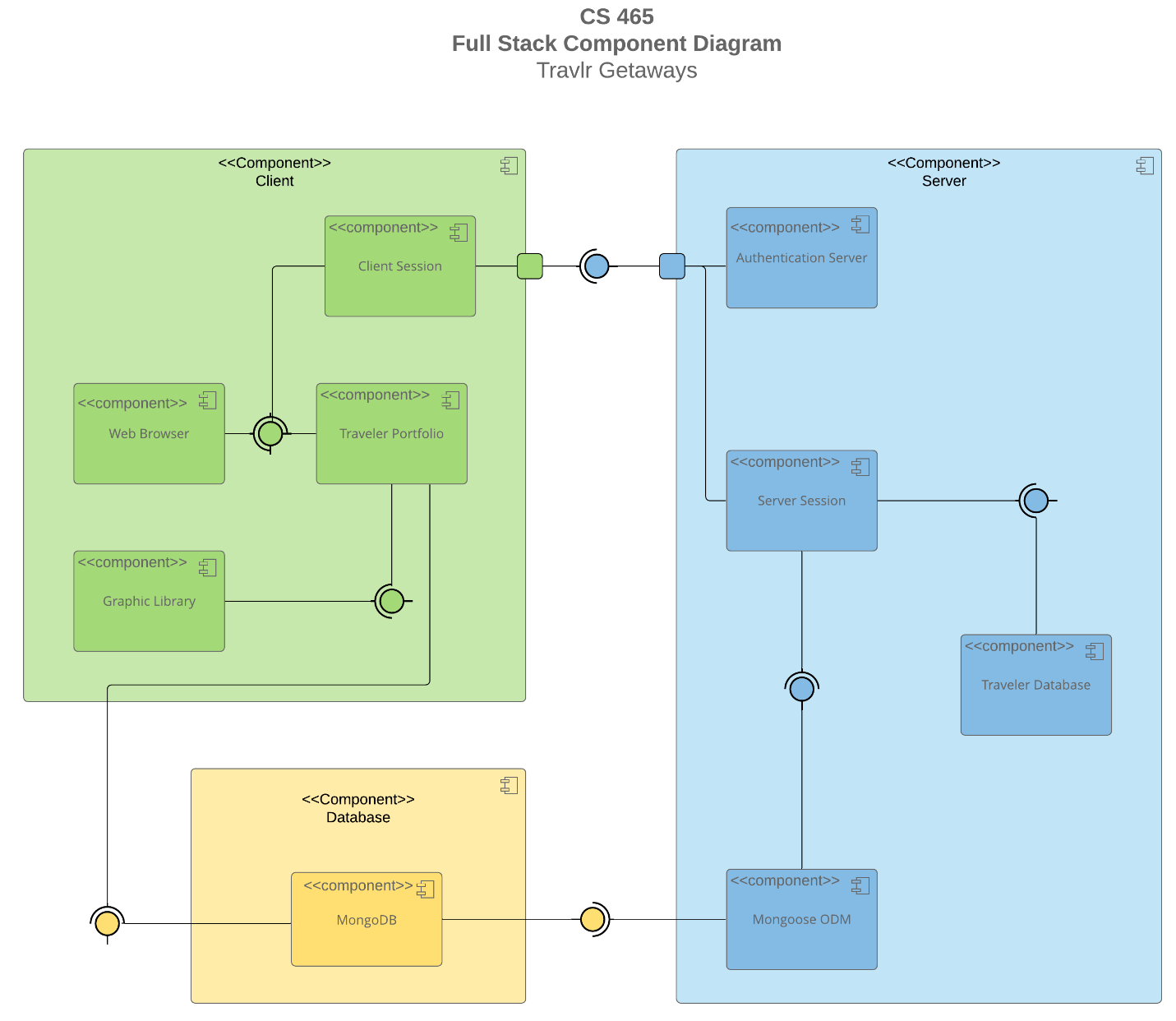
## [Design Constraints](#_heading=h.1ksv4uv)

Developing this application as full-stack does come with certain limitations. The risks we’re aware of are provided in the table below:

|  |  |  |
| --- | --- | --- |
| Constraint | Impact | Mitigation |
| MEAN Stack Limitations | MongoDBs schema flexibility requires robust validation layers. Angular’s complexity requires experienced developers for performance optimization. | We will implement comprehensive server-side validation and data sanitization techniques to ensure data integrity. Our team has extensive experience with Angular, and we'll employ best practices for code optimization and lazy loading to maximize application performance. |
| Real-Time Communication Needs | Implementing WebSocket-based solutions may require additional sever resources and careful error handling. | We will carefully evaluate the need for real-time features (e.g., live chat, instant updates). If required, we'll utilize a scalable WebSocket library and implement robust error handling and connection management to minimize server load and ensure reliability. We’ll also monitor performance closely after deployment. |
| Cross-Platform Compatibility | Ensuring responsive design in Angular apps to support mobile, desktop and tablet users can involve extensive testing and development. | We will adopt a mobile-first approach during development using Angular Material and CSS frameworks designed for responsiveness. Thorough cross-browser and device testing will be conducted throughout the project lifecycle, including automated testing tools where feasible. |
| Security and Compliance | Implementing Payment Gateways (such as Stripe) and Authentication (such as Auth0) require third-party tooling that are not a part of the vanilla MEAN stack. | We will integrate industry-standard security libraries and services like Stripe for secure payment processing and Auth0 for robust authentication and authorization. We’ll adhere to PCI DSS compliance standards and implement regular security audits. |
| API Scalability | REST APIs must be designed with rate limiting, caching and horizontal scaling in mind to handle peak travel booking traffic. | We will design our REST APIs with scalability as a core principle. This includes implementing rate limiting to prevent abuse, utilizing caching mechanisms (e.g., Redis) to reduce database load, and architecting the application for horizontal scaling across multiple servers if needed. We’ll also employ API monitoring tools to identify and address performance bottlenecks. |

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

### 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).

**Summary**

In the component diagram above, the system functionality is outlined. The system is grouped into three overarching parts, the Client Tier (Green), the Server Tier (Blue) and the Database tier (Yellow).

* The **Client** Tier is the section of the application that is client-facing. This is the part of the application responsible for serving web pages, enabling purchase and gathering information about users.
* The **Server** Tier is responsible for Authenticating users before they access the Client, managing individual server sessions and scaling resources to handle those sessions, and managing the database of users.
* The **Database** tier is responsible for storing the details of the Travel package offerings. Here all travel package-related information will be stored and scaled with the business.

The system operates as follows: The user interacts with the Client through a Web Browser. The Client communicates with the Server via API requests. The Server processes these requests and interacts with the Database to retrieve or store data. Data flows bi-directionally between these tiers, enabling a dynamic and interactive user experience.

**Significant Components & Relationships:**

Here's a breakdown of each component and its relationship to others:

**1. Client Tier (Green)**

* **Web Browser:** This is the primary interface for users to access the Travlr Getaways application. It renders the Angular-based user interface and handles user interactions. It initiates requests to the Server tier.
* **Client Session:** Manages temporary data related to a specific user's interaction with the application on the client side (e.g., shopping cart contents, current search filters). This component interacts directly with both the Web Browser and the Authentication Server for session management.
* **Traveler Portfolio:** Represents the core functionality of the customer-facing application – displaying travel packages, allowing users to browse, filter, and select getaways. It communicates with the Server tier to retrieve travel data and update user preferences.
* **Graphic Library:** Provides visual elements and components used within the Angular application (e.g., charts, maps). It supports the Traveler Portfolio in rendering a rich user experience.

**2. Server Tier (Blue)**

* **Authentication Server:** Responsible for handling user authentication and authorization. It verifies user credentials, manages sessions, and issues tokens to authorized users. This component interacts with both the Client Session (for session data) and the Traveler Database (to validate user information).
* **Server Session:** Manages temporary data related to a specific user's interaction on the server side. This is crucial for maintaining state across requests and ensuring security. It communicates with the Authentication Server.
* **Traveler Database:** This component represents the core business logic of the application, handling API requests from the Client tier. It interacts with the MongoDB database to retrieve, create, update, and delete travel data.

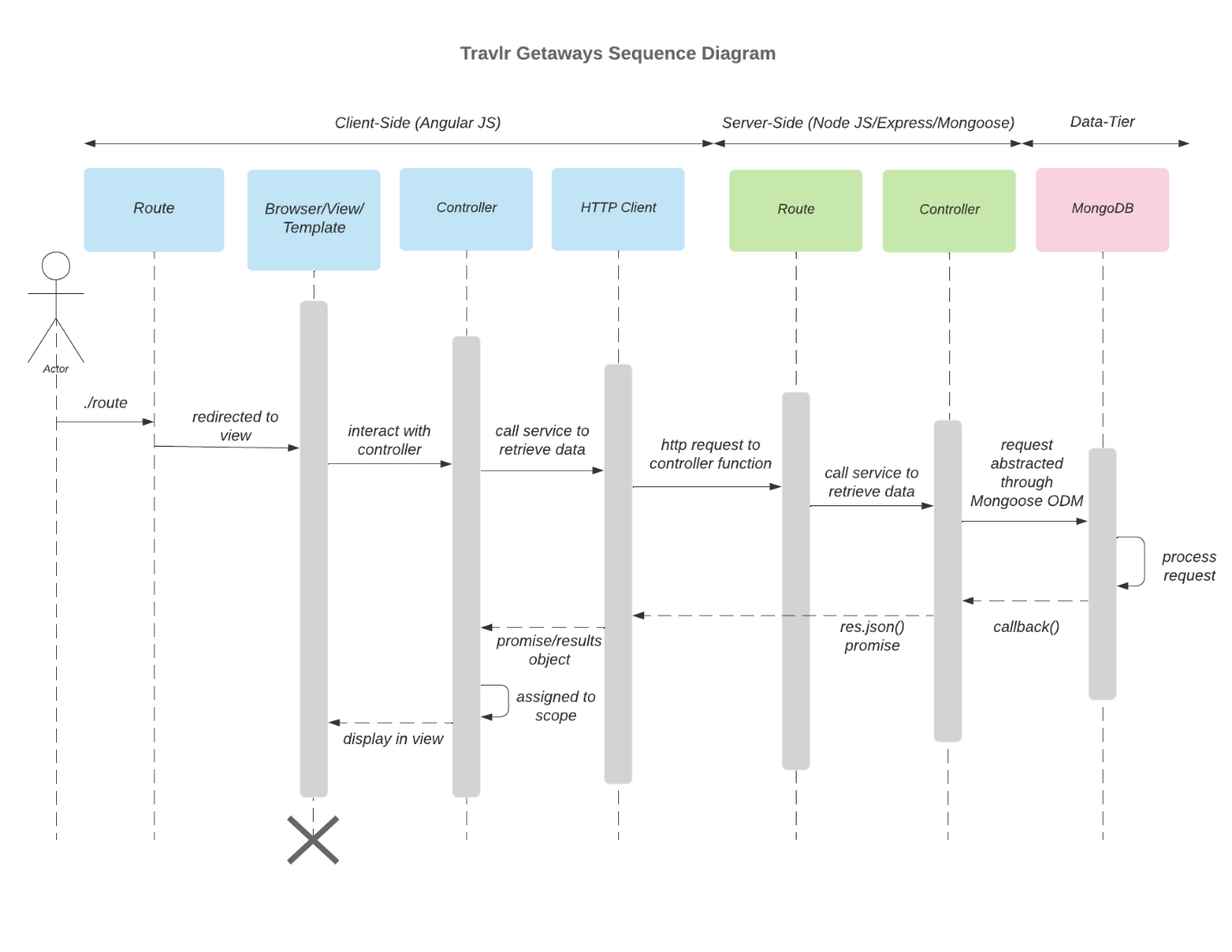
**3. Database Tier (Yellow)**

* **MongoDB:** The NoSQL database used to store all persistent application data, including user accounts, travel packages, booking information, and other relevant details.
* **Mongoose ODM:** An Object-Document Mapper (ODM) for Node.js that provides a schema-based solution for modeling application data within MongoDB. It simplifies interactions with the database by providing an object-oriented interface. This component acts as an intermediary between the Traveler Database and the raw MongoDB instance, enabling efficient data access and manipulation.

Relationships:

* The Web Browser communicates unidirectionally to the Server Tier via API requests (typically RESTful APIs).
* The Authentication Server manages user sessions in coordination with both the Client Session and Server Session.
* The Traveler Database interacts directly with the MongoDB database through the Mongoose ODM, providing a structured interface for data access.
* The Client Tier relies on the Server Tier to provide data and functionality, while the Server Tier depends on the Database Tier for persistent storage.
* This component diagram provides a foundational understanding of how Travlr Getaways will be architected. Further details regarding specific API endpoints, data models, and security protocols will be elaborated in subsequent documentation phases.

### Sequence Diagram



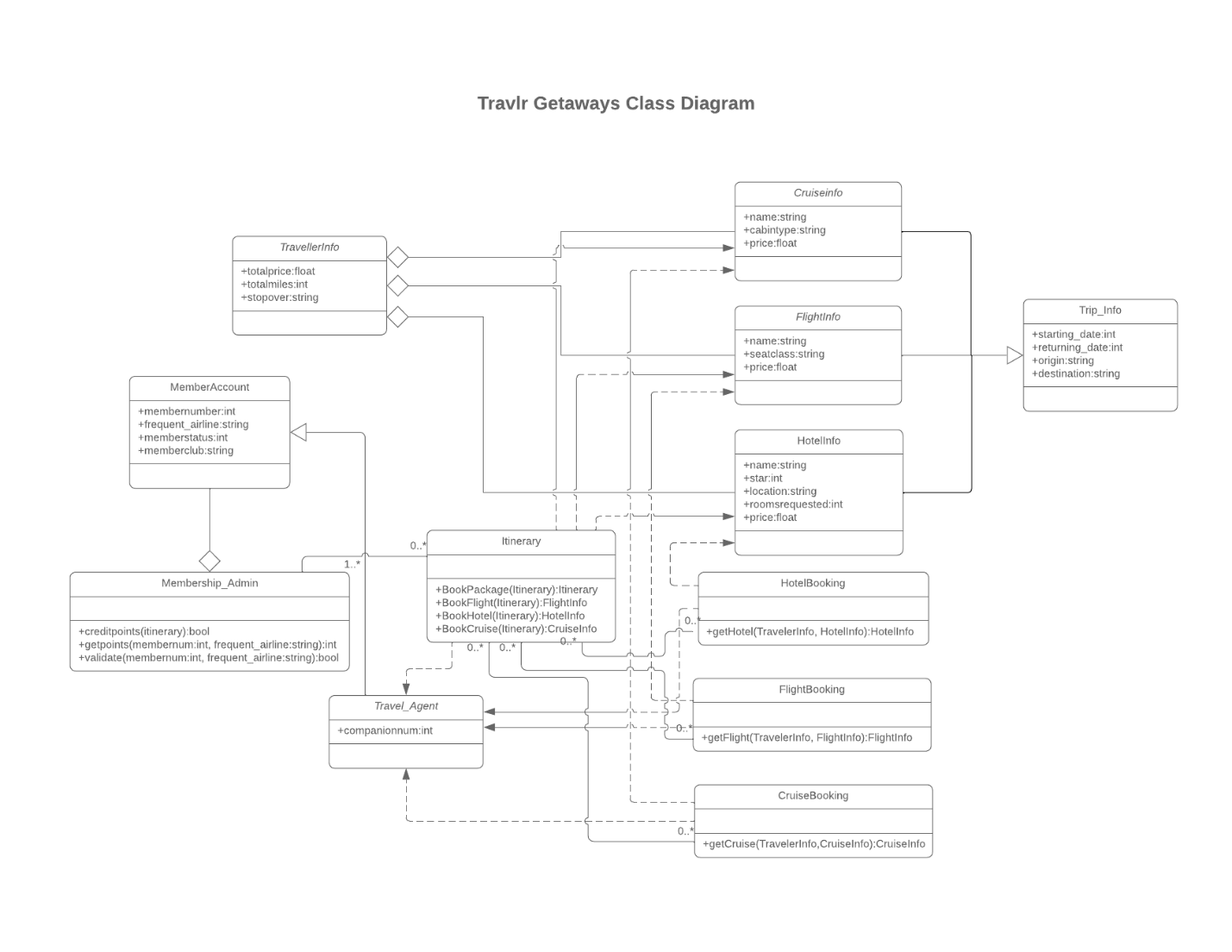
The Travlr Getaways web application follows a classic full-stack architecture, with interactions flowing from the client-side (AngularJS) to the server-side (Node.js/Express), and finally to the data-tier (MongoDB via Mongoose). The sequence diagram illustrates how these layers interact to fulfill requests such as Sign In, Trips management, and Admin functions.

1. **Client-Side Interactions (AngularJS)**:
   * The process begins when a user (actor) accesses a route such as /signin or /trips.
   * The **Route** component on the client detects the path and redirects the user to the appropriate **Browser/View/Template**.
   * The template invokes the relevant **Controller**, which handles logic and state, such as fetching available trips or validating login credentials.
   * The controller makes an asynchronous call to the **HTTP Client** (e.g., Angular's $http service), initiating a request to the backend.
2. **Server-Side Interactions (Node.js/Express)**:
   * The **Route** on the server receives the HTTP request and invokes the appropriate **Controller**.
   * The controller interacts with the service layer or directly with the database abstraction logic to retrieve or update data.
   * This often involves using the **Mongoose ODM**, which translates the application's data model into MongoDB queries.
3. **Data Tier (MongoDB)**:
   * The Mongoose model accesses the **MongoDB** database and performs operations like reading itinerary details, validating member credentials, or storing trip bookings.
   * Once the request is processed, data is returned back through a **callback()** or **res.json()** promise chain.
4. **Response Flow**:
   * The server responds to the HTTP client with the data, which is passed to the AngularJS controller.
   * The controller assigns the response data to $scope, updating the browser view dynamically to reflect trips, admin changes, or member information.

This architecture ensures clear separation of concerns. For example:

* The **Sign In** process validates a user by sending their credentials through the full stack to be validated against the membership system.
* The **Trips** process allows users to query and book vacations by interacting with the itinerary, flight, hotel, and cruise data stored in MongoDB.
* The **Admin** layer (e.g., Membership\_Admin) is used to validate or update user privileges, which flow through similar controller and database layers.

## Class Diagram



The class diagram models the key entities and interactions within the Travlr Getaways application. It outlines how data and methods are structured in the back-end JavaScript (Node.js) logic, enabling robust handling of travel-related services.

1. **MemberAccount & Membership\_Admin**:
   * MemberAccount represents the core user. It includes methods to credit points, validate membership, and calculate frequent flyer rewards.
   * It has a composition relationship with Membership\_Admin, which tracks administrative attributes such as frequent airline and membership club.
2. **Itinerary (Central Hub)**:
   * The Itinerary class serves as the primary organizer of all travel plans. It includes methods like BookPackage, BookFlight, BookHotel, and BookCruise.
   * It maintains associations with booking details for flights, hotels, and cruises, and links back to the TripInfo which summarizes overall cost and mileage.
3. **TravellerInfo**:
   * This class stores aggregated data including totalprice, totalmiles, and stopover, essential for giving users a comprehensive view of their travel package.
4. **Info Classes (CruiseInfo, FlightInfo, HotelInfo)**:
   * These represent metadata about available options and pricing for each travel mode.
   * For instance, FlightInfo includes seat class and price; HotelInfo includes star rating and location; CruiseInfo includes cabin type and pricing.
5. **Booking Classes (CruiseBooking, FlightBooking, HotelBooking)**:
   * These classes handle reservation logic. Each contains a get<Class>() method that takes in TravellerInfo and the associated Info class to retrieve a specific booking.
   * They are tied to TravellerInfo, which holds travel dates and origin/destination data.
6. **Travel\_Agent**:
   * A simplified class with a companionnum attribute that ties into the itinerary, potentially used to manage group bookings.
7. **Trip\_Info**:
   * This standalone class stores per-trip data such as start and return dates, as well as the trip’s origin and destination.

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

| **MongoDB Collection** | **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- | --- |
| Trips | GET | Retrieve list of trips | /api/trips | Returns all active trips |
| Trips | GET | Retrieve single trip | /api/trips/:tripId | Returns a trip by ID |
| Trips | POST | Create a new trip | /api/trips | Creates and returns a new trip object |
| Trips | PUT | Update a trip | /api/trips/:tripId | Updates an existing trip by ID |
| Trips | DELETE | Delete a trip | /api/trips/:tripId | Deletes the specified trip |
| Users | POST | Register a new user | /api/register | Requires Name, Email and Password value |
| Users | POST | Authenticates an Existing User | /api/login | Requires Email and Password |

## The User Interface

## Contrasting Angular Project Structure Against Express Project Structure

The Angular project structure differs significantly from that of the Express HTML customer-facing page. Angular promotes a highly modular design, with a clear separation of concerns through components, services, and routing modules. Each feature is often broken into reusable units with dedicated HTML, CSS, and TypeScript files. In contrast, the Express setup was more linear and flexible—relying on templating engines (like Pug or Handlebars), with looser folder structures and more control over how HTML views and routes are organized. Angular’s CLI encourages a standardized structure, whereas Express requires more manual setup and convention-driven organization.

**Functionality**

A Single-Page Application (SPA) like the one built with Angular offers rich, client-side interactivity compared to a traditional multi-page web app. With SPA, user navigation and data manipulation happen dynamically without requiring full page reloads, resulting in faster and smoother experiences. This allows the use of reusable UI components and reactive forms, making the admin interface more responsive and maintainable.

Some advantages of SPAs include:

* Faster performance after the initial load
* Better user experience due to dynamic content rendering
* Clean API separation between frontend and backend

However, disadvantages can include:

* Longer initial load time due to bundling
* Heavier reliance on JavaScript (can cause issues with SEO or accessibility)
* More complex state management in large applications

Compared to a simple HTML-based web app, SPAs allow for deeper interactivity, better user feedback, and real-time updates, all of which enhance administrative capabilities.

## Testing

Testing the Angular SPA’s connection with the backend API involves verifying that HTTP requests (typically via Angular’s HttpClient) successfully interact with the server endpoints. Tools like Postman can be used to simulate GET, POST, PUT, and DELETE requests, confirming that the server responds as expected and that the frontend updates accordingly.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Feature** | **Tool** | **Request** | **Expected Result** | **Actual Result (Image/JSON)** | **Pass/Fail** |
| List trips | Postman | GET /api/trips | 200 + array of trips |  | ✅ |
| Get trip by code | Postman | GET /api/trips/GALR210214 | 200 + single trip |  | ✅ |
| Add trip (auth) | Angular (Screenshots) | POST /api/trips + JWT | 201 + new trip | Test Case Documentation Below | ✅ |
| Update trip (auth) | Angular (Screenshots) | PUT /api/trips/GALR210214 + JWT | 200 + updated | Test Case Documentation Below | ✅ |
| Delete trip (auth) | Angular (Screenshots) | DELETE /api/trips/GALR210214 + JWT | 200 + deleted doc | Test Case Documentation Below | ✅ |
| Register user | Angular (Screenshots) | POST /api/register | 200 + {token} | Test Case Documentation Below | ✅ |
| Login user | Angular (Screenshots) | POST /api/login | 200 + {token} | Test Case Documentation Below | ✅ |
| Negative: no token | Postman | POST /api/trips (no JWT) | **401** |  | ✅ |
| Negative: bad code | Postman | GET /api/trips/NOPE | **404** |  | ✅ |

Below are visual examples of application functionality (Some of them supplement the above table):

### Adding a Unique Trip:

**Before:**

A screenshot of a website

AI-generated content may be incorrect.

**Add a Trip Form:**

A screenshot of a phone

AI-generated content may be incorrect.

**Updated Screen:**

A screenshot of a computer

AI-generated content may be incorrect.

### Deleting a Trip:

A black screen with white text

AI-generated content may be incorrect.

A screenshot of a website

AI-generated content may be incorrect.

### Edit a Trip:

Before:

A screenshot of a cellphone

AI-generated content may be incorrect.

Edit Form:

A screenshot of a computer

AI-generated content may be incorrect.

After:

A screenshot of a website

AI-generated content may be incorrect.

### Express Renders from Mongo:

A screenshot of a cell phone

AI-generated content may be incorrect.

### User Registration:

A screenshot of a computer

AI-generated content may be incorrect.

### User Login:

A screenshot of a computer

AI-generated content may be incorrect.