

**Travlr Getaways Web 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 | 11/19/2024 | Juan Rodriguez | Completed executive summary and design constraints. |
| 2.0 | 11/30/2024 | Juan Rodriguez | Completed Sequence Diagram, Class Diagram, and [API](#_heading=h.2jxsxqh) Endpoints |
| 3.0 | 12/14/2024 | Juan Rodriguez | Full app completed |

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

The Travlr Getaways web application will be developed using the MEAN stack, which includes MongoDB, Express.js, Angular, and Node.js. This stack provides a modern and efficient way to build both the customer-facing and admin sides of the web application. Node.js will serve as the runtime environment, allowing for the seamless execution of the entire application.

The customer-facing side will leverage Express.js, Handlebars, and MongoDB. Express.js, a framework for Node.js, will efficiently serve static content. Handlebars, a templating engine, will dynamically generate content by populating templates with data pulled from MongoDB, our chosen NoSQL database. This setup ensures the site is flexible and easily maintainable.

The admin side will be built as a Single Page Application (SPA) using Angular. This approach allows administrators to manage content in an intuitive and efficient manner. Admins will need to authenticate before accessing the portal, where they can add, update, or remove content. Any changes made in the admin interface will directly update documents in MongoDB, reflecting instantly on the customer-facing side of the site without needing to redeploy the entire application.

Using the MEAN stack ensures seamless integration between components, making the application scalable, maintainable, and efficient. Key features include dynamic content management, authentication via JSON Web Tokens (JWT), and a responsive design optimized for all devices.

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

Several design constraints must be considered for Travlr Getaways’ web application using the MEAN stack. The primary requirement is that the content on the customer-facing site must be easy to update without redeploying the application. Additionally, the design must be fully responsive, ensuring a consistent user experience across all devices. The admin portal should be a well-optimized SPA that dynamically loads and manages content components.

When considering the MEAN stack, MongoDB offers flexibility and scalability, but performance may degrade if the data set becomes excessively large. The integration of Express, Angular, and Node.js is powerful yet can lead to a substantial codebase, posing challenges in maintainability.

Sensitive operations like login and content management must be protected using robust authentication mechanisms like JWT.

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

### Component Diagram



**Client-Side Components**

•**Web Browser**: The interface used by the end user to interact with the application. It sends and receives data from the server and displays the user interface rendered by Angular.

•**Client Session**: Manages user sessions on the client side, maintaining temporary data such as login status and session information.

•**Traveler Portfolio**: The main feature of the client interface, responsible for displaying travel content, booking information, and user interactions.

•**Graphic Library**: Handles the graphical aspects of the user interface, such as rendering maps or images related to travel content.

Client-side includes the Web Browser, Client Session, Traveler Portfolio, and Graphic Library. The Web Browser and Graphic Library serve as interfaces, with the Client Session and Traveler Portfolio relying on these interfaces. The Traveler Portfolio also interacts with the Database component to retrieve and display travel content dynamically. The Client Session communicates through a designated port to connect with the Server component.

**Server-Side Components**

•**Authentication Server**: Manages user authentication and authorization processes, ensuring secure access to different parts of the application.

•**Server Session**: Maintains server-side session data, coordinating with the Client Session to track user activities and manage session states.

•**Traveler Database**: Stores travel-related data, such as trips, bookings, and user information.

•**Mongoose ODM**: An Object-Document Mapping tool that provides an abstraction layer between the server and the MongoDB database, simplifying data operations and schema management.

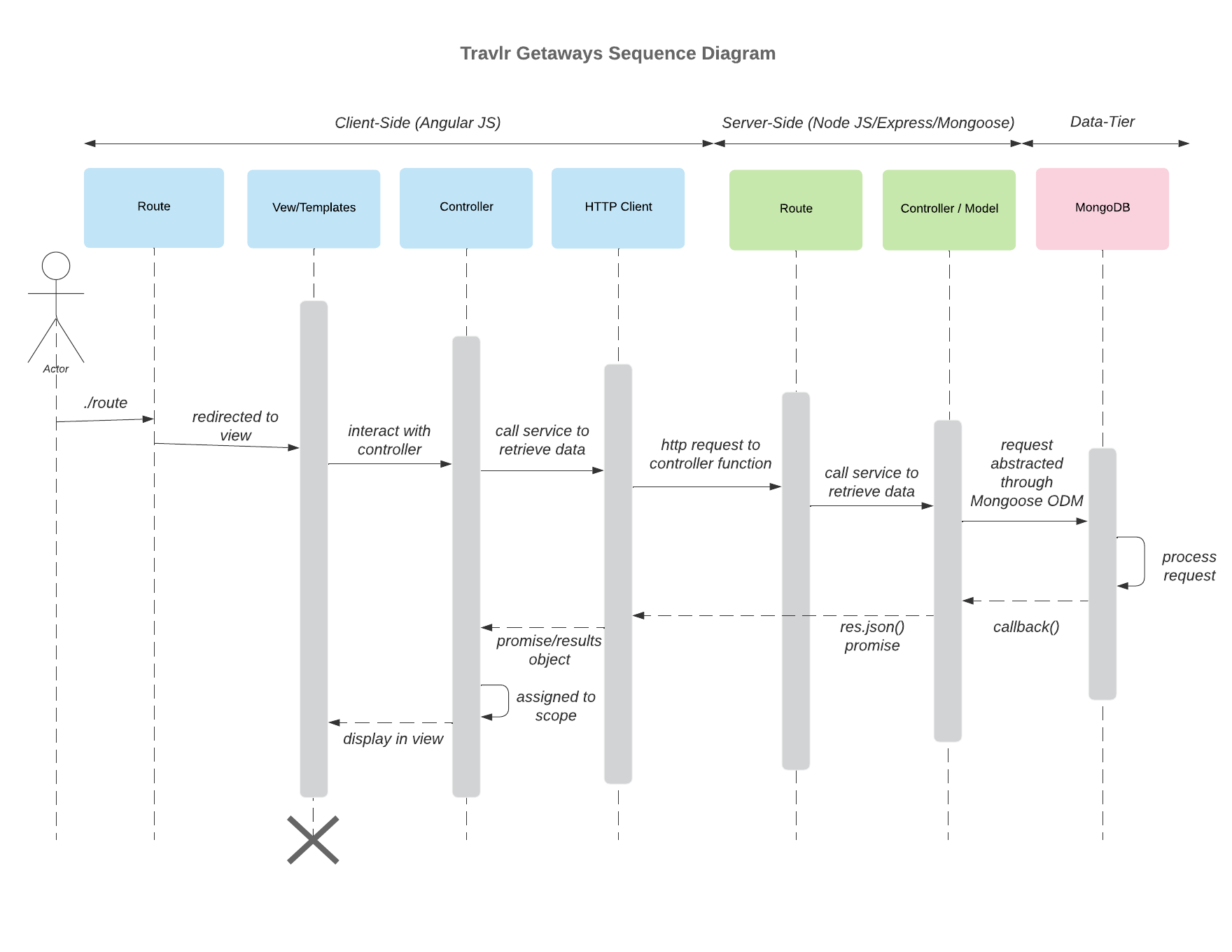
Server-side is composed of the Authentication Server, Server Session, Traveler Database, and Mongoose ODM. The Server Session and Mongoose ODM components provide essential interfaces. The Traveler Database depends on the Server Session interface, while the Server Session relies on Mongoose ODM for database operations. Mongoose ODM interacts directly with the Database component, facilitating efficient data management. The Authentication Server and Server Session connect through a designated port, providing secure interfaces for the Client component.

**Database Component**

•**MongoDB**: The main database where all persistent data, including user details, travel information, and booking records, are stored. It interfaces with Mongoose ODM to streamline data queries and updates.

The Database layer consists of MongoDB, which serves as the core data storage solution. MongoDB provides interfaces to both the Client and Server components, allowing seamless data access and updates.

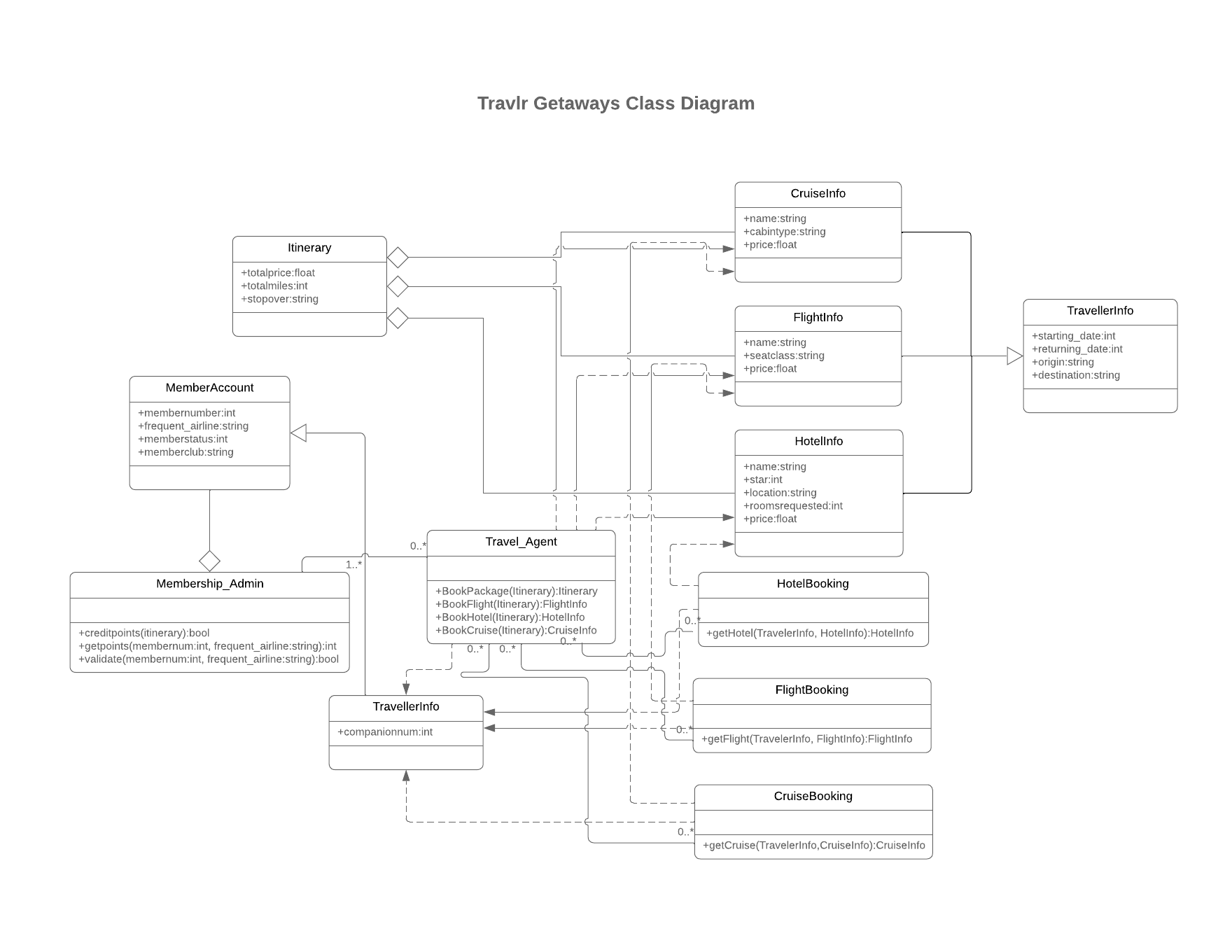
### Sequence Diagram



A user enters a route in the browser’s address bar. The browser interprets the route and forwards the request to the appropriate controller. The application’s app.js file defines the available routes, enabling the browser to determine which routes to use and where to locate them. The route directs the browser to the corresponding controller, which retrieves the requested data from the server-side models. The controller then generates an HTML page that includes the requested data and sends it back to the browser for the user to view.

If the browser requests data stored in the database, a second API controller is employed to fetch and deliver this data. For example, if the page needs to display database content, the process begins similarly, with the initial controller handling the request and rendering the basic HTML page. During this rendering process, a call is made to the second API controller, which uses middleware to retrieve the required data. In this case, the middleware “Mongoose” interacts with MongoDB to fetch the data. If data is found, it is returned to the API controller as one or more JSON objects. The API controller forwards this response to the HTTP client, which then passes it along with the HTML content back to the first controller. Finally, the combined data is rendered in the browser for the user.

## Class Diagram

****

The CruiseInfo, FlightInfo, and HotelInfo classes each contain attributes such as name, price, and other specific properties relevant to their respective types of bookings. These classes inherit from the Itinerary class, as they are key components of a travel itinerary, encompassing essential details about a cruise, flight, or hotel for a trip. The Itinerary class also includes general trip details like totalprice, totalmiles, and stopover. The HotelBooking, FlightBooking, and CruiseBooking classes include methods for booking their respective components, such as getHotel, getFlight, and getCruise. These classes interact with the TravellerInfo class and the corresponding \_\_\_\_Info classes to facilitate booking functionality.

The Travel\_Agent class serves as a central entity, responsible for accessing and coordinating interactions between various classes. It handles operations such as booking a package (BookPackage), booking a flight (BookFlight), booking a hotel (BookHotel), and booking a cruise (BookCruise) using the associated itinerary information. On the administrative side, the Membership\_Admin class aggregates all MemberAccount instances. The MemberAccount class provides member-related details like membernumber, frequent\_airline, and memberstatus. The Membership\_Admin class offers methods to calculate creditpoints, retrieve member points through getpoints, and validate member accounts via the validate method. Additionally, the TravellerInfo class, derived from MemberAccount, includes attributes like companionnum to indicate the number of travel companions, as well as trip-specific details such as starting\_date, returning\_date, origin, and destination.

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

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve one trip by code | /api/trip/:tripcode | Retrieves a single trip object from the database based on the specified “tripcode,” which serves as the reference for identifying the desired trip. |
| **GET** | Retrieve all trips | /api/trips | Returns all trips from the  Database |
| **POST** | Add a trip to the database | /api/trips | Adds a trip to the database |
| **POST** | Register a new user | /api/trip/register | Takes a name, email, and password and places new user in database after hashing the password |
| **POST** | Login user | /api/login | Use email and password to login |
| **PUT** | Update a single trip in  the database | /api/trip/:tripcode | Updates a single trip object  using inputs from an html  form in Angular application |
| **DELETE** | Delete a single trip in the  database | /api/trip/:tripcode | Removes a trip entry from the database by its tripcode. |

## The User Interface

A screenshot of a computer

Description automatically generated

A screenshot of a unique trip was added.

A screenshot of a computer

Description automatically generated

Screenshot of edit trip form.

A screenshot of a computer

Description automatically generated

A screenshot of the trip update PRICE AND NAME WAS UPDATED.

The Angular framework powers the Single Page Application (SPA) for the Travlr Getaways administrative interface, offering a modular and dynamic approach to managing data. Angular’s routing connects reusable components to backend services, enabling seamless navigation and real-time updates without requiring full-page reloads. This ensures an efficient, intuitive user experience for administrators, who can add, edit, or delete content with instant database updates.

The backend, built with Express.js, handles API requests and integrates with MongoDB to store and retrieve data. While Angular focuses on the user interface, Express ensures efficient backend logic and routing. This combination provides a cohesive architecture that bridges front-end functionality with back-end processing.

Compared to traditional web applications, the Angular SPA is more dynamic, allowing users to interact fluidly without disruptive page reloads. Testing plays a vital role in ensuring this seamless integration. Postman, a widely used API testing tool, is instrumental in validating key endpoints, such as GET and PUT, to ensure data retrieval and updates work flawlessly. Additionally, browser developer tools help debug potential issues, providing insights into routing and API interactions.

In summary, the Angular SPA and Express backend architecture deliver a dynamic, reliable, and user-friendly application. With Postman testing ensuring proper integration.