

<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 | 05/25/2025 | Youssef Tazi | I finished the Executive Summary, Design Constraints, and System Architecture View: Component Diagram. |
| 2.0 | 06/08/2025 | Youssef Tazi | I completed the Sequence Diagram, Class Diagram, and API Endpoints sections |

## Instructions

Fill in all bracketed information on page one (the cover page), in the Document Revision History table, and below each header. Under each header, remove the bracketed prompt and write your own paragraph response covering the indicated information.

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

For the Travlr Getaways project, I am building a full stack web application using the MEAN stack. This includes MongoDB, Express.js, Angular, and Node.js. This stack allows me to create both the customer-facing website and the admin single page application in a clean and scalable way.

On the customer side, I am using Express.js along with Handlebars as the view engine to serve dynamic pages. This part of the application lets users browse available trips, view descriptions, and see images from the database. I chose Handlebars because it helps keep the HTML templates organized while still letting me render dynamic content.

For the admin side, I am creating a single page application using Angular. This admin interface lets administrators manage trip data. They can add, update, and delete trips through a modern and responsive interface. The Angular application communicates with the same backend API as the customer site, which helps keep everything in sync.

All trip and user data are stored in MongoDB. I am using Mongoose to define the schema and manage the database interactions. Both parts of the application use RESTful APIs to send and receive data. By keeping the frontend and backend separate, I am following a structure that makes the application easier to manage and update as I continue working on it.

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

While working on the Travlr Getaways application, I must consider few important design constraints that affect how I build the system.

First, I am required to use the MEAN stack, so I have to design everything around MongoDB, Express.js, Angular, and Node.js. This limits me to specific tools and frameworks, but it also helps keep the project consistent and scalable.

Second, the application needs to support both a customer-facing website and an admin single page application. This means I have to make sure that both frontends can connect to the same backend API and use the same data from the database. It also means that I have to carefully plan how the routes, components, and data are structured.

Another constraint is how I manage security. I need to use JSON Web Tokens (JWT) to secure parts of the application, especially for admin features. This adds complexity because I have to set up authentication and make sure the tokens are used properly in the frontend and backend.

Also, since this is a web application, I have to make sure it performs well in a browser and can scale to support multiple users at the same time. I must write efficient code, use asynchronous functions when needed, and follow the best practices for API design.

All these constraints guide me in how I build each part of the application. They help define what tools I use, how I organize the code, and how I test the features as I go.

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

### Component Diagram



The overall architecture of the Travlr Getaways application follows a three-layer structure: the presentation layer, the application layer, and the data layer. I used the component diagram provided to understand how these layers connect and work together.

The presentation layer includes the customer-facing website built with Express and Handlebars, and the admin interface built as a Single Page Application using Angular. Both of these frontends communicate with the backend through API calls.

The application layer is powered by Node.js and Express. This is where I created all the routes, controllers, and logic to handle requests from both the customer site and the admin SPA. I also built an API under the /api path that the Angular application uses to send and receive trip data.

The data layer is handled by MongoDB. I used Mongoose to create schemas and models for trips and users. The backend connects to the MongoDB database to perform all data operations.

Each part of the system plays a specific role. The frontends focus on the user interface and experience, the Express server handles business logic and routes, and the MongoDB database stores all the actual data. By separating these responsibilities and connecting them through RESTful APIs, I am building an application that is easier to manage and scale.

### Sequence Diagram

A diagram of a diagram

AI-generated content may be incorrect.

When a user enters a route like “/travel” in the browser, the AngularJS client manages the request by redirecting to the right view. This view is connected to a controller, which immediately runs and starts the process of retrieving data. The controller uses a service that acts as an HTTP client to send a GET request to the Express backend.

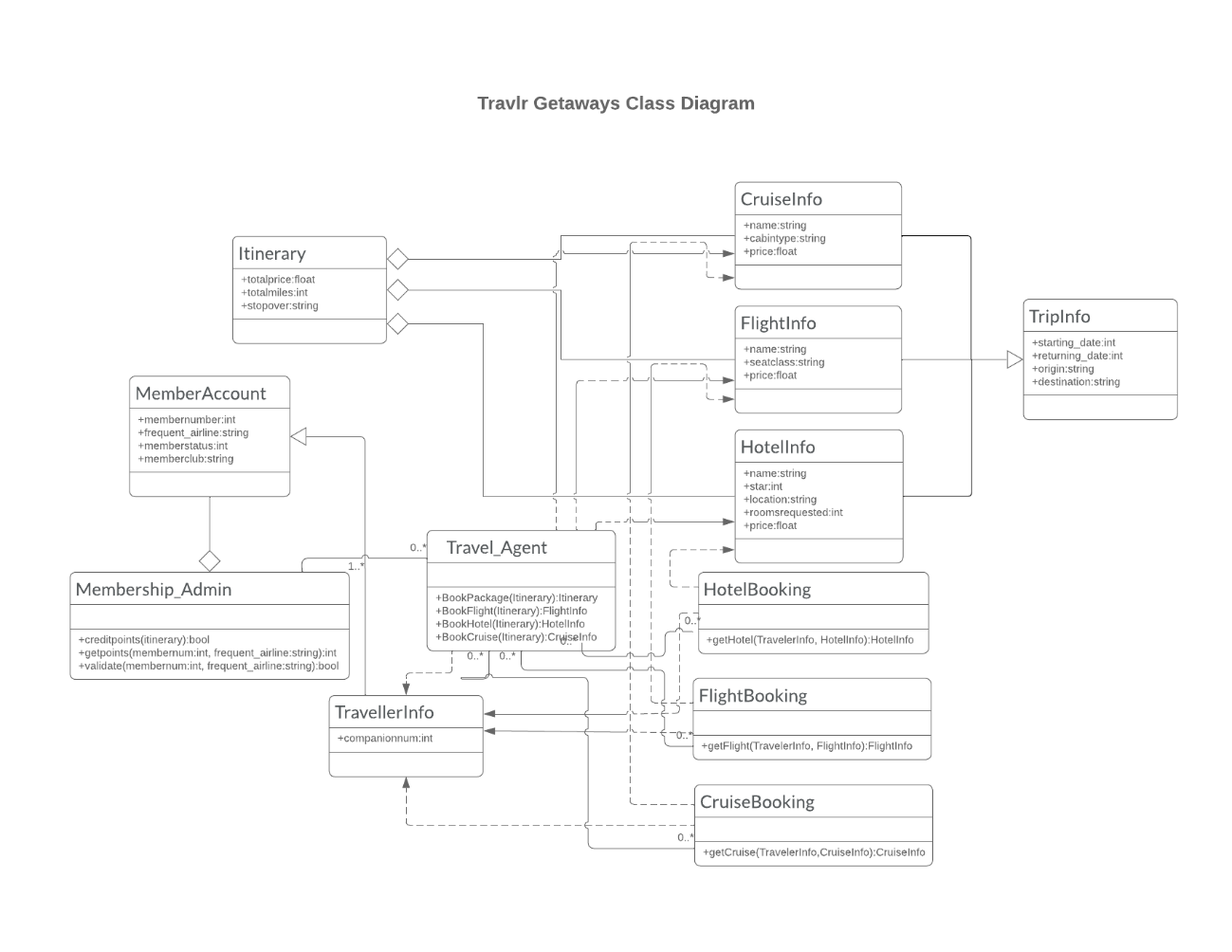
On the server side, the request first hits a route defined in the “app.js” file. This route sends the request to the correct controller function. The controller then calls the model logic using Mongoose to connect with MongoDB.

Mongoose sends the query to the database. Once the data is found, MongoDB returns it through a callback. The controller wraps the data using “res.json()” and sends it back to the frontend.

The Angular client receives this data as a promise. The result is assigned to the controller’s scope, and the page updates to show the data dynamically inside the view.

This full process, shown in the sequence diagram, highlights how the MEAN stack components work together: from client route to server logic, through database access, and finally back to the user in the browser.

## Class Diagram



This class diagram defines the core data structure of the Travlr Getaways application. At the center is the “Itinerary” class, which holds overall travel package details such as” totalprice”, “totalmiles”, and “stopover”. This class is associated with “CruiseInfo”, “FlightInfo”, and “HotelInfo”, each of which includes specific attributes like “name”, “cabinType”, or “seatClass”, along with pricing.

Bookings are handled by classes such as “HotelBooking”, “FlightBooking”, and “CruiseBooking”. Each booking class includes methods that connect user details (stored in “TravellerInfo”) with the appropriate type of travel information. The “TripInfo” class includes scheduling details and helps define when and where a trip occurs.

The application supports both “MemberAccount” and “Membership\_Admin”. Admins manage point tracking and validation using methods like “getpoints()” and “validate()”. The “Travel\_Agent” class enables booking functionality with methods like “BookFlight()” and “BookPackage()”, supporting scalability and separation of responsibilities.

The structure supports object reuse, encapsulation, and inheritance principles. It is designed to support both customer-facing views and admin/agent functionality while allowing the system to track users, book multiple travel types, and manage data efficiently.

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

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve all trips | |  | | --- | | /api/trips |  |  | | --- | |  | | Returns an array of all trips in JSON format |
| **GET** | Retrieve one trip | /api/trips/:tripCode | Returns details for a single trip |
| **POST** | Add a new trip | /api/trips | Requires JWT authentication |
| **PUT** | Update a trip | /api/trips/:tripCode | Requires JWT; updates trip by tripCode |
| **DELETE** | Delete a trip | /api/trips/:tripCode | Requires JWT; removes a trip from DB |
| **POST** | Register new user | /api/register | Creates user account and returns JWT |
| **POST** | Log in existing user | /api/login | Authenticates user and returns JWT |
| **GET** | Verify token/auth | /api/profile | Validates JWT and returns user info |

These RESTful API endpoints allow the Angular frontend and Express backend to communicate through clearly defined HTTP routes. Admin actions such as adding, editing, or deleting trips require JWT authentication to ensure secure access. Users can also register and log in through the API, receiving a token for further interaction. The use of RESTful patterns allows easy integration, testing, and expansion of the system as the application grows.

## The User Interface

The Angular project is built using a component-based structure. This means each feature, like showing trips, adding a trip, or editing one, is placed in its own component. I also used services in Angular to send and receive data from the backend API. This structure helps keep the project organized and makes it easy to work on each part separately.

This is different from the Express side of the project. The customer-facing website is built using Express and Handlebars. There, I used the MVC structure with routes, controllers, and views. Handlebars were used to create dynamic HTML templates that show trip data from the database.

The admin side, built with Angular, is a single-page application (SPA). It lets me update the screen without refreshing the page, and it feels faster and more modern. It also has a login system with JWT authentication to protect admin features.

To make sure everything works, I tested the GET and PUT requests in Angular. When I added or edited a trip in the admin panel, I checked that the new data appeared right away on the customer-facing site. I also tested the login and logout features to make sure only authorized users can make changes.

The screenshots below show how the admin can add a trip, edit it, and how the updated data appears for users.

These first two screens before adding my trips

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

The Edit Trip screen

A screenshot of a computer

AI-generated content may be incorrect.

The Add Trip screen  
A screenshot of a computer

AI-generated content may be incorrect.

Two unique trips, added by me:

A screenshot of a computer

AI-generated content may be incorrect.

Customer-facing site with updated trips:

A screenshot of a computer

AI-generated content may be incorrect.