

Travlr Getaways

# **CS 465 Project Software Design Document**

Version 3.0

## 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 | <03/23/25> | John Munguia | Added Executive Summary, Design Constraints, System Architecture View Description |
| 2.0 | <04/13/25> | John Munguia | Added Sequence Diagram, Class Diagram, API Endpoints |
| 3.0 | <04/26/25> | John Munguia | Added User Interface screenshots and summary |

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

The Travlr Getaways web application is being developed using the MEAN stack, specifically leveraging Node.js and Express.js for backend development and Handlebars as the view engine for rendering dynamic content. The application follows a Model-View-Controller (MVC) architecture, allowing for clear separation of concerns and maintainable code organization. On the customer-facing side, the website initially consisted of static HTML pages, but these have been converted into dynamic Handlebars templates that use partials for reusable components such as headers and footers. Trip information, which was originally hardcoded, is now stored in a JSON file and rendered dynamically using Handlebars looping directives. This approach enables greater flexibility and scalability while maintaining a clean and consistent user experience. The project structure includes separate folders for controllers, routes, views, and data, supporting organized development and efficient rendering of user-facing pages.

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

The development of the Travlr Getaways application is constrained by several key design decisions established early in the project. The use of the MEAN stack, specifically Node.js with Express.js and Handlebars, sets a clear structure for both the server-side logic and the rendering of dynamic content. One constraint is the reliance on Handlebars as the templating engine, which requires all dynamic views to be constructed using its syntax and limits layout flexibility compared to other frontend frameworks. Additionally, the application must serve dynamic content without yet using a database, meaning trip data must be manually stored and accessed through a static JSON file. This impacts performance and scalability, as all data must be loaded from the filesystem during each request. Furthermore, the separation of static assets and server-rendered views necessitates careful file organization and routing to ensure correct content delivery. These constraints shape the application's early architecture, demanding a balance between modular code structure and simplified data handling to support maintainability and future enhancements.

## [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).

The Travlr Getaways web application is structured around three core components: **Client**, **Server**, and **Database**. The client side includes a web browser interface, where users interact with the Traveler Portfolio. Handlebars templates render the content dynamically, with reusable elements managed through partials. A client session tracks user interaction, and visual elements are handled through a graphic library.

The server, built with Node.js and Express.js, processes requests, serves views, and routes data through the MVC framework. Although the component diagram includes authentication and database integration, only the server session and JSON-based data handling are active at this stage. Trip data is read from a static JSON file and passed to the client via Handlebars templates.

While the architecture includes MongoDB and Mongoose ODM for future integration, they are not yet in use. This structure lays the groundwork for a scalable, maintainable web application while supporting dynamic content rendering based on early project requirements.

### Sequence Diagram

A diagram of a trade end

AI-generated content may be incorrect.

The sequence diagram illustrates the flow of logic in the Travlr Getaways web application, which follows a layered full-stack architecture using the MEAN stack. The process begins when a user accesses a specific route, such as /travel, which triggers the client-side to load the appropriate view through Express and Handlebars. Initially, this view was populated using a local trips.json file, but starting in Module 4, the application transitions to retrieving data from a MongoDB database.

In Module 4, a Mongoose schema is created to define the structure of trip data. This data is seeded into the MongoDB database, replacing the earlier JSON file. The server uses Mongoose ODM to interact with the database, abstracting direct queries and enabling a more scalable data management approach.

In Module 5, a RESTful API is implemented on the server side. When the client requests trip data, the browser makes a call through an HTTP client to an API endpoint like /api/trips. This request is routed through the Express server to a controller function that queries the MongoDB database using the Mongoose model. The retrieved trip data is returned as a JSON response and passed back through the HTTP client.

Once the response is received, the client-side controller assigns the data to the view’s context, where it is rendered dynamically using Handlebars templates. This allows users to see live trip information pulled from the database. This updated structure improves on the static rendering by introducing live, database-driven content and RESTful interaction, while still relying on the MVC pattern established earlier in the project.

## Class Diagram

A diagram of a company

AI-generated content may be incorrect.

The class diagram for Travlr Getaways outlines a structured approach to organizing trip-related data and functionality within the application. At the core is the Itinerary class, which represents a user’s full travel plan, including the total price, total miles, and any stopovers.

Several booking-related classes interact with Itinerary. These include FlightInfo, HotelInfo, and CruiseInfo, each containing key attributes such as name, location, class type, and price. These are used by their respective booking classes (FlightBooking, HotelBooking, CruiseBooking) to retrieve travel options based on traveler preferences. The TripInfo class holds the core travel dates and location details, connecting the other components of a trip.

The TravelAgent class serves as the main interface for booking travel. It includes methods like BookPackage, BookFlight, BookHotel, and BookCruise, each taking relevant data objects such as Itinerary, HotelInfo, or CruiseInfo as parameters and returning the appropriate travel booking.

User accounts are managed by the MemberAccount class, which stores frequent flyer numbers, membership status, and club affiliation. This class is used by the Membership\_Admin class to manage and validate user accounts, award credit points, and handle frequent flyer information through methods like creditpoints and validate.

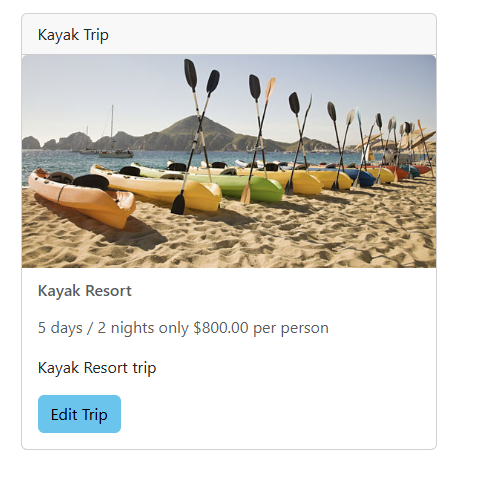
Lastly, the TravelerInfo class supports details like the number of companions, which is used across the various booking classes to customize the user's experience.

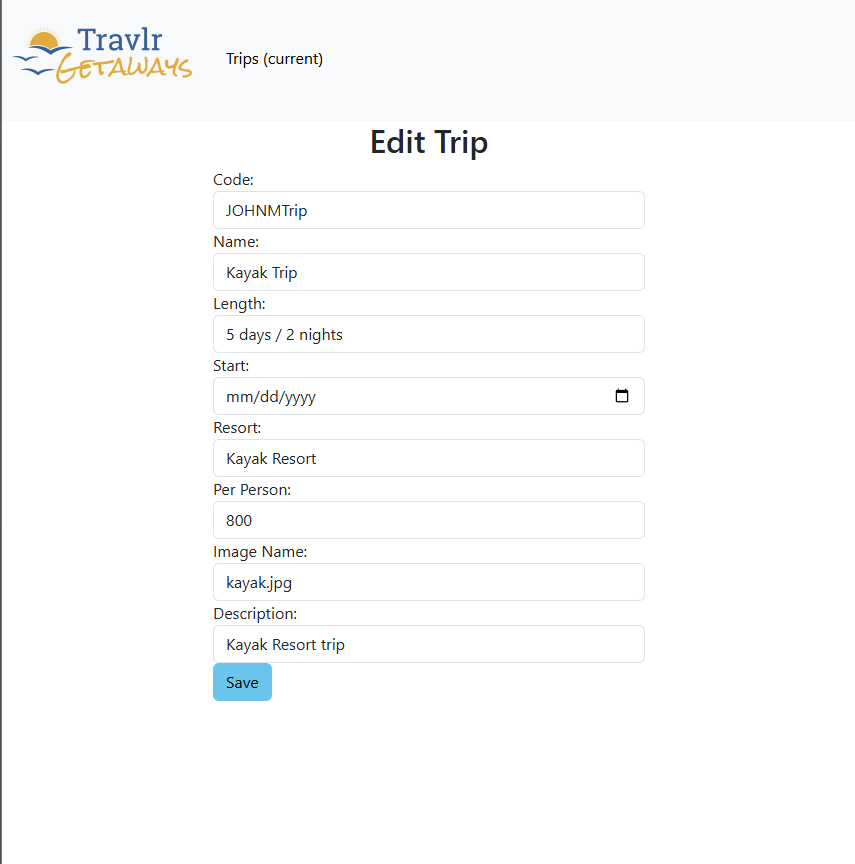
Together, these JavaScript class structures define the application's domain logic, making it easier to manage trip bookings and user account features in an organized, object-oriented way.

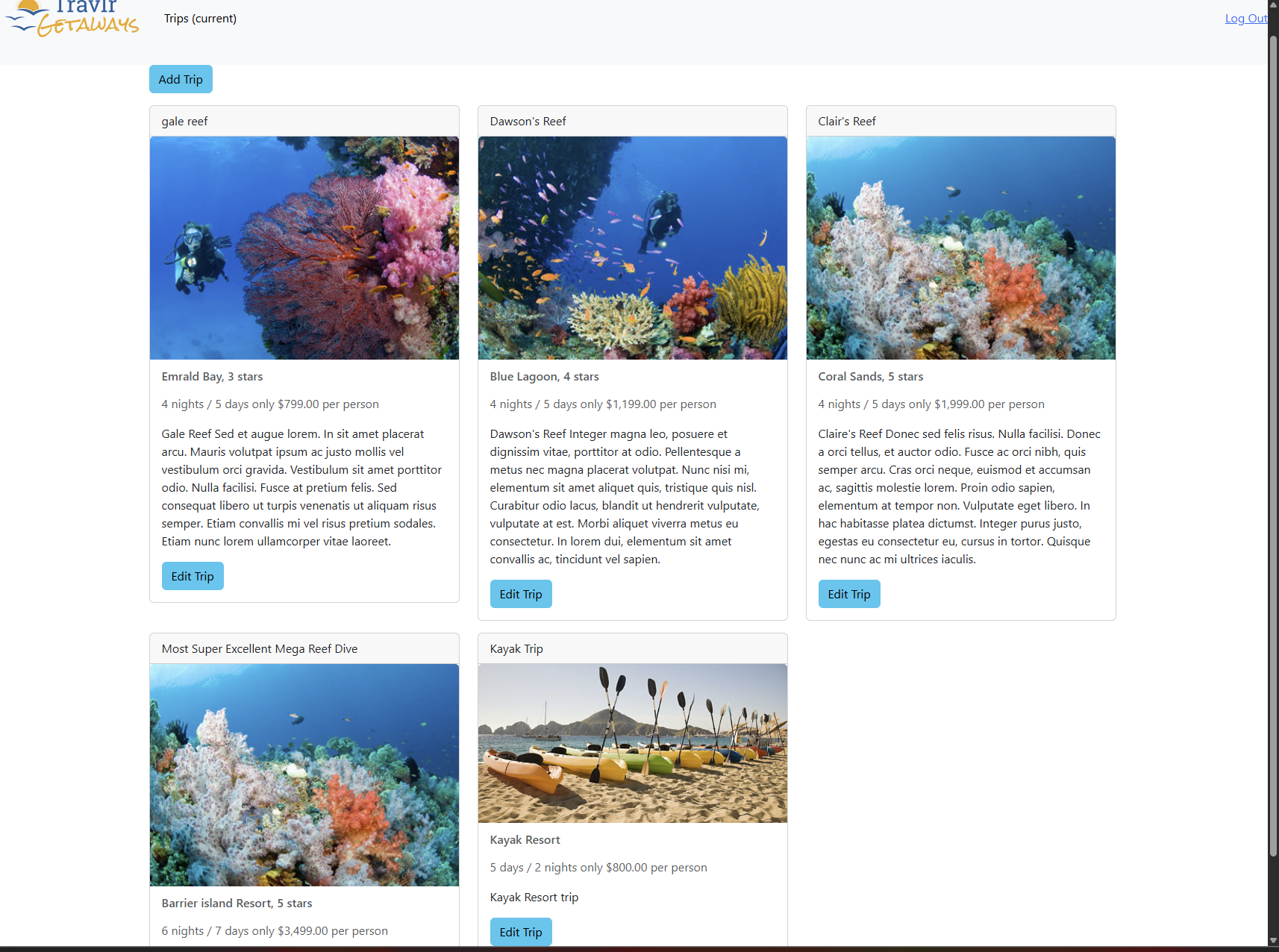
## [API](#_heading=h.2jxsxqh) Endpoints

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve All trips | /api/trips | Retrieves all entries within MongoDB |
| **GET** | Retrieves single trip | /api/trips/tripcode | Returns single trip using its tripcode |

## The User Interface







In this project, the **Angular** and **Express** project structures are organized differently to match their purposes.  
The **Express project** (in travlr/) is focused on building a REST API. It uses folders like app\_api (for API logic) and app\_server (for server-side rendering if needed). The main job of Express here is to handle backend routes, interact with the MongoDB database, authenticate users, and send JSON data. It doesn't handle the visual frontend directly.

The **Angular project** (in travlr/app\_admin/) is a fully separate frontend app. It uses components, services, and models organized into folders like src/app/components, src/app/services, and src/app/models. Instead of loading new pages each time a user clicks something, Angular acts as a **Single Page Application (SPA)**. It updates views dynamically without reloading the page, creating a faster and smoother user experience.

The SPA provides rich functionality compared to a traditional web application. In a simple web app, every action (like submitting a form) usually reloads the entire page. But in the SPA, the app only sends or receives data from the server (through API calls), and Angular updates the visible parts of the page automatically without reloading everything. This makes things feel much faster and more modern for the user.

To test the SPA and make sure it works properly with the API, we first test the backend routes in Postman. We send GET, POST, and PUT requests to make sure data can be retrieved, added, and updated correctly. After confirming the backend works, we then test the Angular frontend by using the web app itself: logging in, adding trips, editing trips, and watching the app update dynamically. We also check that JWT tokens are attached correctly to secured API calls.