

Travlr Getaways Website

# **CS 465 Project Software Design Document**

Version 2.0

## 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 | 09/27/2024 | Madison Fleitas | Create and implement a customer-oriented front-end architecture that fulfills client software needs. Utilize the HBS templating engine to dynamically render web pages. Organize routes and views to align with the client’s software specifications using the MVC framework. Conduct tests to ensure that static HTML is effectively transformed into dynamic JSON data and displays correctly. |
| 2.0 | |  | | --- | | 10/26/2024 |  |  | | --- | |  | | Madison Fleitas | Upgrade to version 2.0 to improve system scalability and support additional functionalities. Refactor API endpoints to versioned URLs (/api/v2) for better maintenance. Enhance the front-end architecture by optimizing data handling and updating UI components. Extend existing tests to validate new functionalities and ensure compatibility with version 1.0 features. Implement changes to streamline account management processes and improve overall performance. |
|  |  |  |  |

## 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)

The web application for Travlr Getaways is designed using the MEAN stack (MongoDB, Express, Angular, Node.js), which provides a full JavaScript-based development environment for both the server-side and client-side components. This architecture allows for scalability, maintainability, and seamless user experience, meeting the specific requirements of both the customer-facing website and the administrator single-page application (SPA).

The customer-facing portion of the website will focus on providing a visually engaging and intuitive interface for customers to explore Travlr Getaways' offerings, make reservations, and interact with dynamic content such as travel packages. The use of Angular allows for efficient rendering of dynamic content, while the Express framework on the backend will handle server requests and facilitate communication between the front-end and the MongoDB database.

The admin SPA is designed to give Travlr Getaways' management team a powerful interface to manage content, user data, and bookings. Angular will also power the admin side, ensuring fast, real-time updates as administrators manage inventory and make changes to travel packages or customer information. This will allow for efficient data handling and streamlined management of the website's operations.

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

Developing the Travlr Getaways web application involves several design constraints that impact both the customer-facing website and the administrator single-page application (SPA). These constraints must be considered to ensure the final product is scalable, secure, and user-friendly.

Travlr Getaways has a limited development timeline and budget. These constraints restrict the use of complex and expensive technologies or third-party services, requiring the application to be built with open-source tools such as the MEAN stack.

The web application needs to support thousands of users simultaneously, particularly during peak travel seasons. To meet these demands, the application must be optimized for performance with efficient database queries, caching mechanisms, and minimal latency.

As the web application will handle sensitive customer data such as payment information and personal details, stringent security measures must be implemented. This includes encrypting data, ensuring secure authentication, and adhering to industry security standards.

The customer-facing website needs to be responsive and functional across various devices and browsers. Developing a responsive design that works seamlessly across desktops, tablets, and mobile devices while maintaining a consistent user experience adds to the development effort.

## [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 system architecture for the Travlr Getaways web application consists of three main layers: client, server, and database. On the client side, users interact through a web browser, which renders the traveler portfolio, a key feature for managing accounts and bookings. The client session component stores session data, ensuring that user-specific actions, such as browsing and booking, are handled smoothly. A graphic library enhances the visual aspects of the website, supporting dynamic elements like maps or interactive content.

On the server side, an authentication server handles user login and security, ensuring only authorized access. This server communicates with the server session, which tracks user interactions and ensures they are processed properly during actions like making bookings or account updates.

The MongoDB database stores all application data, including user information and travel packages. Mongoose ODM acts as an intermediary between the server and MongoDB, ensuring data is properly validated and managed. Together, these components create a robust, modular system that supports both the customer-facing website and the admin single-page application (SPA), enabling seamless user experiences and efficient data management.

### Sequence Diagram

A diagram of a diagram

Description automatically generated with medium confidence

A diagram of a software

Description automatically generated

The sequence diagram starts with the user initiating a "Sign In Request," which is sent to the Browser/View/Template. The browser then sends an "HTTP Request" to the HTTP Client, which forwards it to the Route as "To Route." From there, the request moves to the Controller, labeled "To Controller." The Controller interacts with the Model by sending a "Query Credentials" request, and the Model communicates with MongoDB through "Access DB." MongoDB returns the "User Data" to the Model, which then sends the "Auth Result" to the Controller. The Controller sends the "Sign In Result" back to the Browser/View/Template, where it's displayed to the user.

Next, the user requests "Trip Info" from the Browser/View/Template. This request is sent as an "HTTP Request" to the HTTP Client, which forwards it to the Route as "To Trip Route," and then to the Trip Controller as "To Trip Controller." The Controller sends a "Query Trip Info" request to the Model, which accesses MongoDB through "Access DB." MongoDB returns the "Trip Data" to the Model, and the Controller processes this data before sending the "Display Trip Info" response back to the Browser/View/Template for the user to see.

For admin actions, the admin user initiates an "Admin Action" request from the browser, which then sends an "HTTP Request" to the HTTP Client. The request is routed to the Admin Route and forwarded to the Admin Controller. The Admin Controller communicates with the Model to "Perform Operation," which involves accessing MongoDB as "Perform Action." Once MongoDB completes the action, it sends back a confirmation to the Model, which then relays a "Success" message to the Admin Controller. The Controller sends an "Admin Confirmation" response back to the Browser/View/Template, completing the action.

## Class Diagram

1. A diagram of a travel gateway

   Description automatically generated<**TripInfo**: This class holds the core details of a trip, such as the trip ID, destination, start and end dates, and overall itinerary. It serves as the main reference for accessing and managing all trip-related information.
2. **Itinerary**: Itinerary handles the specific schedule for a trip, including details like activity names, locations, times, and descriptions. It lets users view and update the planned activities for their trip.
3. **TravellerInfo**: This class manages traveler-specific information, including their name, contact details, and any preferences. It’s used to keep track of each traveler's profile and any special requirements.
4. **FlightInfo**: FlightInfo manages details about flights, such as flight numbers, departure and arrival times, airlines, and ticket info. It helps users keep up with their flight schedules and reservations.
5. **HotelInfo**: This class deals with hotel booking details like the hotel name, check-in and check-out dates, room type, and booking confirmation. It allows users to manage their accommodation plans.
6. **CruiseInfo**: If a trip involves a cruise, CruiseInfo covers cruise-related details such as the cruise line, ship name, itinerary, cabin details, and departure port. This class makes sure all cruise information is easily accessible.
7. **FlightBooking**, **HotelBooking**, and **CruiseBooking**: These classes manage the booking process for flights, hotels, and cruises. They include attributes like booking status, payment info, and any cancellation or modification details. Each one tracks the reservation lifecycle, making sure everything stays up to date.
8. **Travel\_Agent**: The Travel\_Agent class supports travel agents who manage trips for clients. It includes attributes for agent ID, contact info, and a list of assigned trips, enabling agents to access and update trip details on behalf of clients.
9. **MemberAccount**: This class handles user account details, including username, password, membership level, and preferences. It also tracks the user’s trip history and associated bookings.
10. **Membership\_Admin**: Membership\_Admin provides admin capabilities for managing user accounts and membership levels. It includes methods to create, update, or delete accounts, as well as handle membership changes.

## [API](#_heading=h.2jxsxqh) 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 things> | </api/things> | <Returns all active things> |
| **GET** | <Retrieve single thing> | </api/things/:thingId> | <Returns single thing instance, identified by the thing ID passed on the request URL> |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **GET** | |  |  |  | | --- | --- | --- | | |  | | --- | | **Get all trips** |  |  | | --- | |  | |  |  | | --- | |  | | |  | | --- | | **/api/trips** |  |  | | --- | |  | | |  | | --- | | **Returns a list of all trips.** |  |  | | --- | |  | |
| **GET** | |  | | --- | | **Get a specific trip by ID** |  |  | | --- | |  | | |  | | --- | | **/api/trips/** |  |  | | --- | |  | | |  | | --- | | **Needs a valid trip ID to look it up.** |  |  | | --- | |  | |
| **POST** | |  |  | | --- | --- | | **Create a new trip** |  |  |  | | --- | |  | | |  |  |  | | --- | --- | --- | | |  | | --- | | **/api/trips** |  |  | | --- | |  | |  |  | | --- | |  | | |  | | --- | | **Requires trip details in the request body.** |  |  | | --- | |  | |
| |  | | --- | | **PUT** |  |  | | --- | |  | | |  | | --- | | **Update a trip by ID** |  |  | | --- | |  | | |  | | --- | | **/api/trips/** |  |  | | --- | |  | | |  | | --- | | **Needs the trip ID and the updated data.** |  |  | | --- | |  | |
| **DELETE** | |  | | --- | | **Delete a specific trip by ID** |  |  | | --- | |  | | |  | | --- | | **/api/trips/** |  |  | | --- | |  | | |  | | --- | | **Requires a valid trip ID to delete it.** |  |  | | --- | |  | |
| **GET** | |  | | --- | | **Get all flight information** |  |  | | --- | |  | | |  | | --- | | **/api/flights** |  |  | | --- | |  | | |  | | --- | | **Returns a list of flights.** |  |  | | --- | |  | |
| **POST** | |  | | --- | | **Add a new flight** |  |  | | --- | |  | | |  | | --- | | **/api/flights** |  |  | | --- | |  | | |  | | --- | | **Needs the flight details in the request.** |  |  | | --- | |  | |
| **GET** | |  | | --- | | **Get all hotel bookings** |  |  | | --- | |  | | |  | | --- | | **/api/hotels** |  |  | | --- | |  | | |  | | --- | | **Returns a list of hotel bookings.** |  |  | | --- | |  | |
| **POST** | |  | | --- | | **Create a new hotel booking** |  |  | | --- | |  | | |  | | --- | | **/api/hotels** |  |  | | --- | |  | | |  | | --- | | **Requires the booking details in the request.** |  |  | | --- | |  | |
| **GET** | |  | | --- | | **Get all user accounts (Admin only)** |  |  | | --- | |  | | |  | | --- | | **/api/admin/accounts** |  |  | | --- | |  | | |  | | --- | | **Admin access is needed.** |  |  | | --- | |  | |
| **PUT** | |  | | --- | | **Update user account information** |  |  | | --- | |  | | |  | | --- | | **/api/admin/accounts/** |  |  | | --- | |  | | Requires the account ID and updated info. |
|  |  |  |  |

## The User Interface

<Insert screenshots from the development of the SPA development to show the following: (1) a unique trip, added by you, (2) the Edit screen, and (3) the Update screen.>

<Summarize the Angular project structure and how it compares to the Express project structure. Be sure to describe the rich functionality provided by the SPA compared to a simple web application interaction. Describe the process of testing to make sure the SPA is working with the API to GET and PUT data in the database.>