

Travlr Getaways

# **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 | 07/16/2024 | Joseph Les | Initial Draft |
| 1.2 | 08/12/2024 | Joseph Les | Final Draft |

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

Travlr Getaways aims to create an intuitive and comprehensive travel booking website that allows customers to seamlessly book travel packages, manage their reservations, and view detailed itineraries. Our solution will leverage the MEAN stack; MongoDB, Express.js, Angular, and Node.js, to deliver a robust, scalable, and efficient web application. This modern stack enables the development of dynamic, single-page applications (SPAs) for both customers and administrators. Customers will enjoy a user-friendly interface to create accounts, search for travel packages by location and price, book reservations, and regularly check their itineraries. Concurrently, administrators will benefit from a dedicated SPA to efficiently manage customer data, travel packages, and pricing, ensuring optimal service and updated offerings. This architecture ensures a streamlined experience for all users and provides a solid foundation for future scalability and enhancements.

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

1. Technology Stack:

The MEAN stack (MongoDB, Express.js, Angular, Node.js) is chosen for its cohesive environment and JavaScript consistency, but it limits the project to its capabilities. The development team must be proficient in these technologies, and any limitations may impact features and performance.

Implications: Requires team proficiency in the MEAN stack; limitations may affect feature set and optimization.

2. Security and Privacy:

Handling personal and financial data necessitates stringent security measures like data encryption and secure authentication, complying with regulations such as GDPR or CCPA.

Implications: Increases development complexity and requires regular security audits and updates.

3. Performance and Scalability:

The application must perform well under varying loads and be scalable, especially during high-traffic periods.

Implications: Requires performance optimization and scalable infrastructure, impacting cost and resource allocation.

4. Integration with External Services:

Integrations with payment gateways, third-party travel APIs, and email services are necessary.

Implications: Introduces dependencies and potential points of failure, requiring careful management and testing.

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

### 

### Component Diagram



Client Components

1. Web Browser: This is the interface through which the end-users (both customers and administrators) will interact with the application. It is responsible for rendering the web pages and executing client-side scripts.
2. Client Session: Manages the state and interactions of a user’s session on the client-side. This includes storing temporary data and user interactions.
3. Traveler Portfolio: A component that handles the display and management of user-specific data, such as booked travel packages, itineraries, and personal information.
4. Graphic Library: A utility component used for rendering graphical elements in the browser, such as charts, maps, and other visual data representations.

Server Components

1. Authentication Server: Responsible for managing user authentication and authorization. It handles login, registration, and session management.
2. Server Session: Manages the server-side session states, keeping track of user interactions and data exchanges between the client and server.
3. Traveler Database: Interfaces with the database to retrieve and store traveler-related data, such as user profiles, travel packages, and booking information.
4. Mongoose ORM: An Object-Relational Mapping (ORM) tool that facilitates interaction between the server and the MongoDB database, providing an abstraction layer for database operations.

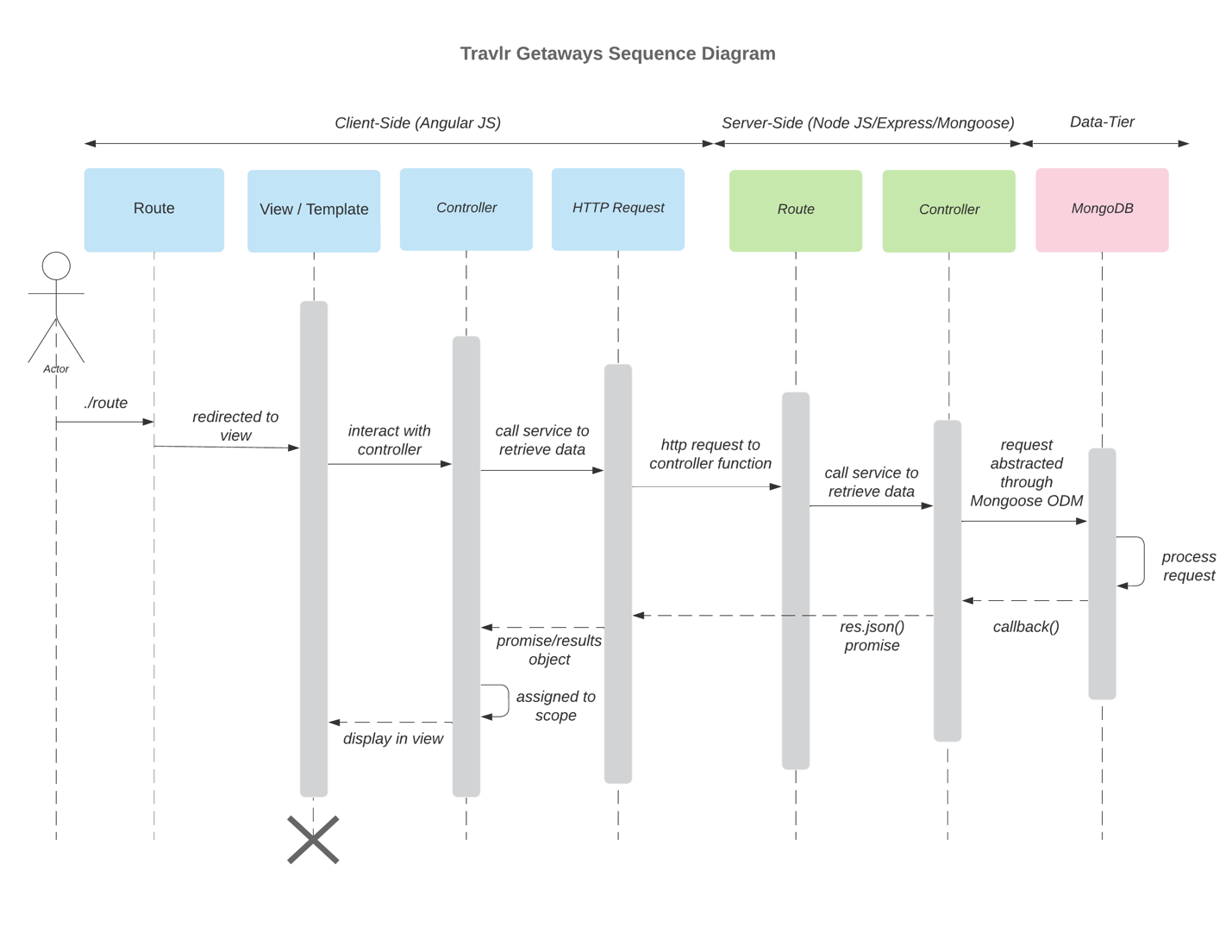
Database Component

1. MongoDB: The primary database used for storing all application data, including user information, travel packages, bookings, and other relevant data.

Component Relationships

1. Web Browser communicates with Client Session, Traveler Portfolio, and Graphic Library to provide a responsive and interactive user experience.
2. Client Session and Traveler Portfolio interact with the Authentication Server and Server Session on the server side to manage user sessions and data retrieval.
3. Server Session and Traveler Database interface with the Mongoose ORM to perform database operations.
4. Mongoose ORM interacts directly with MongoDB to store and retrieve application data.

### Sequence Diagram



The sequence diagram for the Travlr Getaways web application illustrates the flow of logic across the client-side (AngularJS), server-side (Node.js/Express), and data-tier (MongoDB) components. When a user initiates an action by navigating to a route, the client-side route redirects them to the appropriate view and interacts with a controller. The controller manages data and logic, making HTTP requests to the server-side API when necessary. The server-side route receives these requests, and the associated controller processes them, interacting with the MongoDB database via Mongoose to retrieve or modify data. The server then responds with the necessary data, which the client-side controller uses to update the view, displaying the results to the user. Significant processes, such as user sign-in, viewing trips, and admin interactions (like managing trips), are seamlessly handled through this architecture, ensuring dynamic and responsive application behavior.

## Class Diagram

A diagram of a class diagram

Description automatically generated

The class diagram for the Travlr Getaways web application depicts the relationships and attributes of various JavaScript classes. The primary class, `Travel\_System`, serves as the central hub, connecting other classes like `User\_Profile`, `Itinerary`, and `Booking`. The `User\_Profile` class manages user-related information and links to `Booking`, which handles booking details and interactions. The `Itinerary` class stores travel schedules, associating with the `Trip` class, which contains trip-specific data. The `Trip` class is connected to `Flight\_Booking`, `Hotel\_Booking`, and `Car\_Rental\_Booking` subclasses, each managing different types of reservations. The `Admin\_Profile` class, linked to the `Travel\_System`, includes methods for administrative actions, such as managing trips and user profiles. The class diagram illustrates a well-organized structure for handling user information, bookings, and administrative functionalities, ensuring a cohesive and comprehensive management of the travel application's data and processes.

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

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve list of Trips | /api/trips | Returns a list of all active trips available in the system. |
| **GET** | Retrieve Single trip | /api/trips/ | Return details of a single trip identified by the tripId parameter. |
| **POST** | Create a new trip | /api/trips | Adds a new trip to the database; requires trip details in the request body. |
| **PUT** | Update a trip | /api/trips/ | Updates the details of an existing trip identified by the tripId. |
| **DELETE** | Delete a trip | /api/trips/ | Removes a trip from the database, identified by the tripId. |
| **GET** | Retrieve list of bookings | /api/bookings | Returns all bookings made by users. |
| **GET** | Retrieve single booking | /api/bookings/ | Returns details of a specific bookings, identified by the bookingsId. |
| **POST** | Create a new booking | /api/bookings | Creates a new booking; requires booking details in the request body. |
| **PUT** | Update a booking | /api/bookings/ | Updates an existing booking identified by the bookingId. |
| **DELETE** | Cancel a booking | /api/bookings/ | Cancels a booking, removing it from the system. |

## 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.>

A screenshot of a website

Description automatically generated

A screenshot of a phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated A screenshot of a trip

Description automatically generated

A screenshot of a website

Description automatically generated

The Angular project structure is organized around the concept of modules, components, services, and routing. At a high level, the Angular project is a single-page application (SPA) that is broken down into different modules, each containing components responsible for specific pieces of functionality. The main module serves as the root module and bootstraps the application. Each component within the module is self-contained, with its own HTML template, CSS styles, and TypeScript logic. Angular's services provide a way to inject dependencies, such as the HTTP client used to interact with the backend API, which is critical for making GET and PUT requests to retrieve and update data.

In contrast, an Express project is organized around the concept of middleware, routes, and controllers. Express is a minimalist web framework for Node.js, where the primary structure involves defining routes that respond to HTTP requests. Controllers in Express handle the logic for these routes, such as querying a database or processing form data. Middleware functions are used to handle requests at various stages, like authentication or data validation. The overall architecture is less opinionated than Angular's, allowing for greater flexibility but requiring more manual setup for things like dependency injection and state management.

Testing the SPA to ensure it works correctly with the API involves both unit and integration testing. Unit tests are typically written to validate the functionality of individual components and services in isolation. Integration testing involves checking how the components interact with the backend API. This may involve using tools such as Postman, ensuring that the components can properly handle GET requests to fetch data and PUT requests to update data in the database. These tests confirm that the SPA correctly processes and displays data from the API, providing a reliable and functional user experience.