

TRAVLR Getaways – Merrik Wright

# **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 | 01/25/2025 | Merrik Wright | Initial version with Executive Summary, Design Constraints, and System Architecture 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)

The Travlr Getaways website will be developed using the MEAN stack, an acronym for MongoDB, Express.js, Angular, and Node.js. This technology stack enables seamless development by ensuring efficient handling of JSON data, which enhances communication between the front-end, back-end, and database. The MEAN stack is known for its efficiency, scalability, and ability to streamline development workflows.

* The web interface will be managed using Angular, providing a responsive and visually appealing user experience.
* The server-side logic will be handled by Node.js, utilizing the Express.js framework for efficient request handling.
* The database will be powered by MongoDB, allowing for fast retrieval and storage of trip-related data, ensuring optimal website performance.

In terms of data flow, Angular requests information from Node.js, which processes the request through Express.js. Node.js then interacts with MongoDB to retrieve the necessary data, which is then sent back to Angular for display. This structured data flow ensures a smooth and responsive user experience.

Customer-Facing Web Application:

The customer-facing side of the application will allow users to browse available trips, make bookings, and manage their reservations. The interface will be designed to offer a seamless and enjoyable experience, incorporating features such as dark mode to accommodate users who are sensitive to bright screens.

The application follows an MPA (Multi-Page Application) structure, built using Express.js and Handlebars. While MPAs can sometimes have slightly longer load times between pages, the overall website performance will remain fast and efficient.

Administrator Single-Page Application (SPA):

The admin panel will be developed as a Single-Page Application (SPA) using Angular. This approach ensures that all necessary data is preloaded and dynamically updated, minimizing page reloads and enhancing responsiveness.

Administrators will have the ability to add, edit, and delete trip listings and manage other website content in real-time. Any changes made through the admin panel will be instantly reflected for all users, ensuring efficient content management and seamless updates.

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

When developing the Travlr Getaways web application, several technical and architectural constraints must be considered to ensure scalability, security, and performance. Since the application is built using the MEAN stack (MongoDB, Express.js, Angular, and Node.js), these constraints influence how each component interacts and operates within the system.

Scalability & Performance

* The application must support thousands of simultaneous users, requiring efficient load balancing and optimized database queries.
* MongoDB's BSON document size is limited to 16MB, preventing excessive memory usage. Additionally, nested document depth is capped at 100 levels, ensuring structured and manageable data storage.
* The front-end (Angular) and back-end (Node.js, Express.js) must be optimized for fast load times, potentially incorporating caching mechanisms like Redis to enhance responsiveness.

Security & Data Storage

* Authentication must be implemented using JWT (JSON Web Tokens) or OAuth to ensure secure user access and prevent unauthorized actions.
* MongoDB enforces unique field names and follows case-insensitive database naming conventions, which should be adhered to for consistency and error prevention.
* Data indexing and efficient schema design are critical for quick retrieval of trip details, user profiles, and bookings without performance degradation.

Cross-Platform Compatibility & Deployment

* The application must function seamlessly across desktop and mobile devices, ensuring a responsive design for an optimal user experience.
* Hosted on cloud services such as AWS, Firebase, or Heroku, with proper scaling strategies to handle fluctuating traffic.
* Express.js is used to render dynamic content efficiently, and Angular with Handlebars allows for flexible and user-friendly UI interactions.

By adhering to these design constraints, Travlr Getaways will have a better chance to maintain a secure, scalable, and high-performance system, providing a smooth experience for both customers and administrators.

## [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 built on the MEAN stack architecture, consisting of three main components: Client, Database, and Server. Each component interacts seamlessly to ensure a secure, efficient, and scalable user experience. This can be seen in the graph provided above.

Client Component:

The Client is where user interactions begin and consists of four key sub-components:

* Client Session: Manages active user sessions and sends authentication requests to the server.
* Graphic Library: Ensures smooth rendering of the interface and enhances visual elements.
* Traveler Portfolio: Displays trip details and allows users to interact with the website.
* Web Browser: Serves as the user's gateway to access the application.

When a user launches the web application in their browser, the Client Session is initiated. This session sends a request to the Authentication Server to verify whether the user is already logged in. Once authenticated, the Traveler Portfolio loads, displaying available trips and personalized user data. During this process, the Graphic Library initializes to enhance the interface and ensure smooth rendering.

Database Component:

The Database layer consists of a single but crucial sub-component:

* MongoDB: The central database where all trip data, user profiles, and bookings are stored and managed.

When a user interacts with the website through the Traveler Portfolio, MongoDB dynamically updates to reflect any changes. This ensures that trip availability, bookings, and user preferences are stored and retrieved in real time.

Server Component:

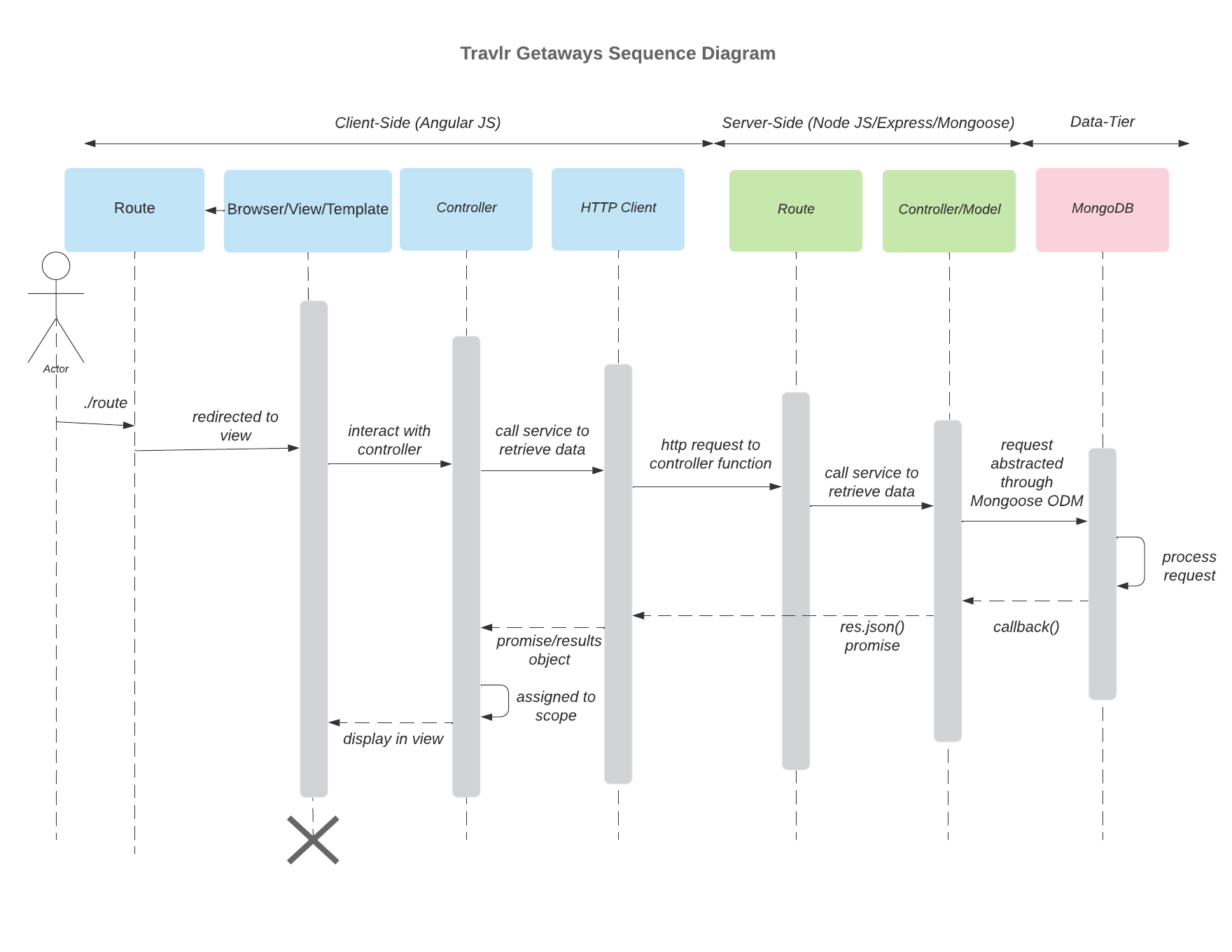
The Server is responsible for handling user authentication, managing sessions, and ensuring smooth data flow between the Client and Database. It consists of four key sub-components:

* Authentication Server: Verifies user credentials and ensures secure access.
* Mongoose ODM: Acts as an Object-Document Mapper, enabling efficient schema management and database interactions.
* Server Session: Manages user session states and validates login credentials.
* Traveler Database: Handles trip-related data and user-specific information.

When a request is made from the Client, the Server Session checks the user’s credentials against the MongoDB database. This is facilitated through Mongoose ODM, which streamlines schema-based operations and speeds up data retrieval. The Authentication Server then confirms the user’s legitimacy and relays the authentication status back to the Client Session, allowing continued interaction with the application.

Overall, this interconnected system ensures that the **Travlr Getaways** web application remains **responsive, scalable, and secure**. By leveraging **MongoDB, Express.js, Angular, and Node.js**, the application provides a seamless experience for both travelers and administrators. Additionally, **Handlebars** is utilized to dynamically render content, reinforcing the efficiency of the MEAN stack architecture.

### Sequence Diagram



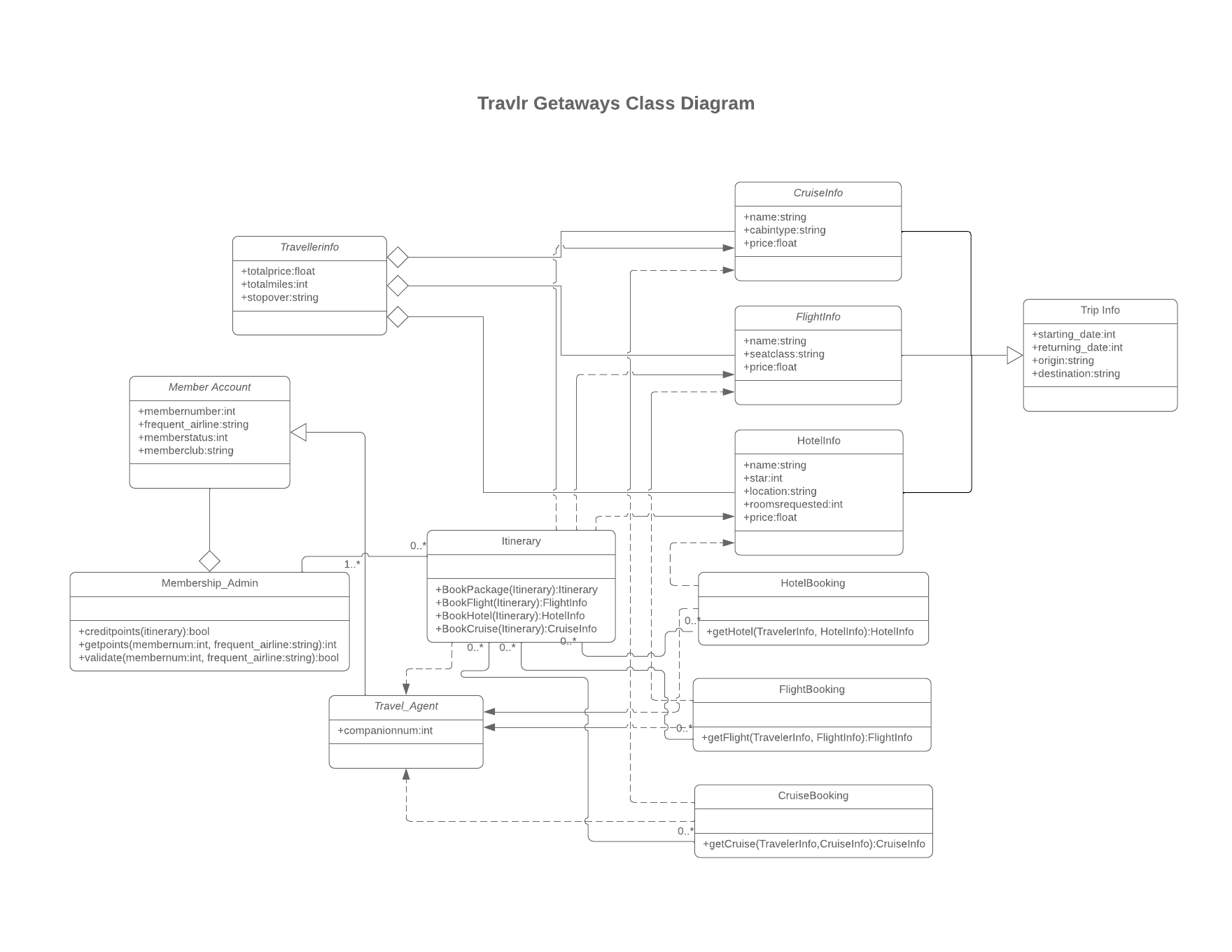
The process begins when a user (the **actor**) accesses the website by navigating to a specific URL. Upon reaching the site, the **browser/view/template** is loaded, rendering the requested webpage for the user. If the user interacts with the page by clicking on a navigational link or performing an action that triggers a data request, the **controller** takes over.

The **controller** then communicates with the **HTTP client**, which serves as an intermediary between the client-side and the backend server. This HTTP client initiates a request to retrieve relevant data from the **API controller** on the server. The request is processed through the **server route**, which directs it to the appropriate **controller/model** for handling.

Next, the **controller/model** queries the **MongoDB database** using Mongoose ODM, abstracting the request and fetching the necessary information. Once the data retrieval process is completed, the database returns the requested information to the **controller/model**, which formats the response and sends it back to the **client-side controller** as a JSON object.

The **client-side controller** then assigns the retrieved data to the application’s **scope**, ensuring it is dynamically displayed in the view for the user. The process continues, responding to further interactions and dynamically updating the webpage as the user navigates and requests additional information.

## Class Diagram



The **Travlr Getaways web application** consists of **12 JavaScript classes**, each representing key components of the system's architecture and interactions.

1. **Membership\_Admin** acts as an aggregator for **MemberAccount** and is associated with one or more **Itinerary** instances.

* creditpoints(itinerary): boolean – Returns whether credit points are applicable.
* getpoints(membernum: int, frequent\_airline: string): int – Retrieves points for a given member and frequent airline.
* validate(membernum: int, frequent\_airline: string): boolean – Validates membership status based on provided credentials.

1. **MemberAccount** is inherited by **Travel\_Agent** and aggregated by **Membership\_Admin**.

* membernumber: int – Unique identifier for the member.
* frequent\_airline: string – Preferred frequent flyer program.
* memberstatus: int – Membership status level.
* memberclub: string – Assigned membership club.

1. **Travel\_Agent** inherits from **MemberAccount** and has relationships with **Itinerary, HotelBooking, FlightBooking,** and **CruiseBooking**.

* companionnum: int – Number of companion travelers allowed.

1. **Itinerary** is linked to **Membership\_Admin** (requiring at least one) and interacts with **CruiseInfo, FlightInfo, HotelInfo, HotelBooking, FlightBooking,** and **CruiseBooking**.

* BookPackage(itinerary): Itinerary – Books a travel package.
* BookFlight(itinerary): FlightInfo – Reserves a flight.
* BookHotel(itinerary): HotelInfo – Books a hotel stay.
* BookCruise(itinerary): CruiseInfo – Reserves a cruise package.

1. **CruiseBooking** is associated with **Itinerary** (where multiple instances can exist) and interacts with **CruiseInfo** and **Travel\_Agent**.

* getCruise(TravellerInfo, CruiseInfo): CruiseInfo – Retrieves cruise details for a traveler.

1. **FlightBooking** follows a similar structure, relating **Itinerary, FlightInfo,** and **Travel\_Agent**.

* getFlight(TravellerInfo, FlightInfo): FlightInfo – Fetches flight details for a traveler.

1. **HotelBooking** also interacts with **Itinerary, HotelInfo,** and **Travel\_Agent**.

* getHotel(TravellerInfo, HotelInfo): HotelInfo – Obtains hotel booking details for a traveler.

1. **CruiseInfo** is associated with **TravellerInfo, Itinerary,** and **CruiseBooking**, while inheriting attributes from **TripInfo**.

* name: string – Name of the cruise package.
* cabintype: string – Type of cabin assigned.
* price: float – Cost of the cruise package.

1. **FlightInfo** is structured similarly to **CruiseInfo**, inheriting from **TripInfo** and interacting with **TravellerInfo, Itinerary,** and **FlightBooking**.

* name: string – Name of the airline or flight provider.
* seatclass: string – Seating class (e.g., economy, business).
* price: float – Ticket price.

1. **HotelInfo** inherits **TripInfo** and is related to **TravellerInfo, Itinerary,** and **HotelBooking**.

* name: string – Hotel name.
* star: int – Hotel star rating.
* location: string – Address or city of the hotel.
* roomsrequested: int – Number of rooms reserved.
* price: float – Cost per room.

1. **TripInfo** is the parent class for **CruiseInfo, FlightInfo,** and **HotelInfo**, defining shared attributes:

* starting\_date: int – Start date of the trip.
* returning\_date: int – Return date.
* origin: string – Departure location.
* destination: string – Arrival location.

1. **TravellerInfo** aggregates **CruiseInfo, FlightInfo,** and **HotelInfo**, storing relevant travel details.

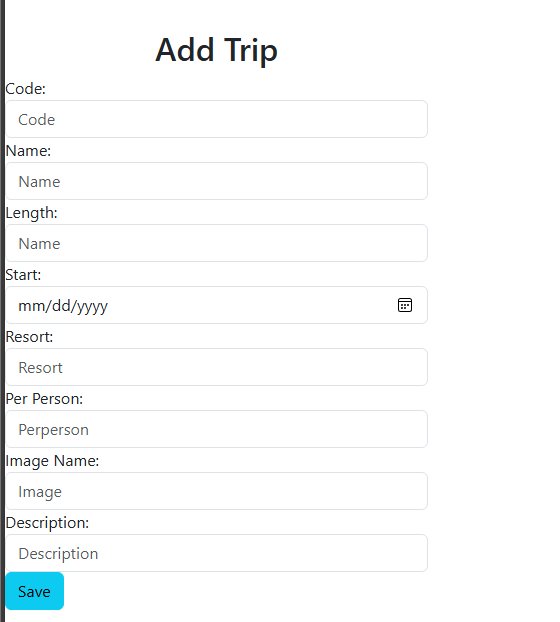
* totalprice: float – Total cost of the trip.
* totalmiles: int – Miles accumulated during the trip.
* stopover: string – Stopover details, if applicable.

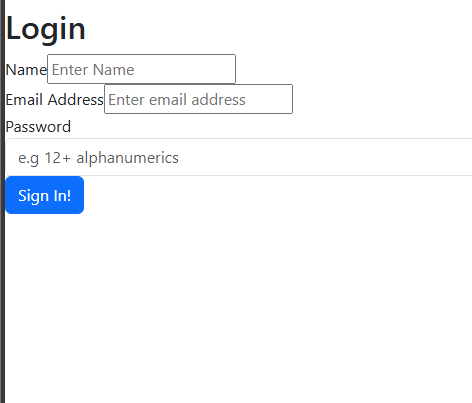
## [API](#_heading=h.2jxsxqh) Endpoints

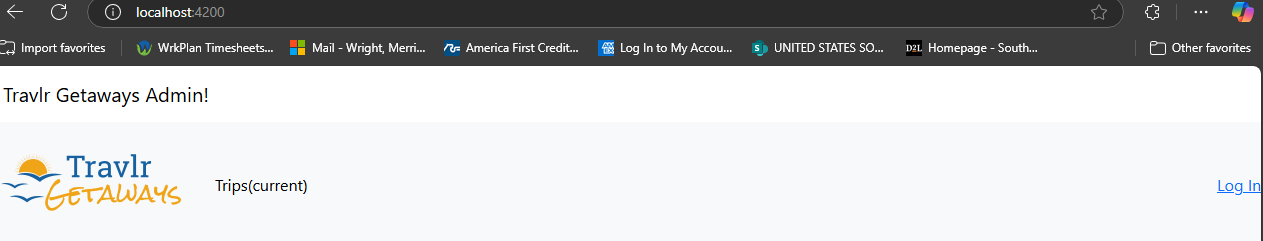
| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | **Retrieve list of blogs** | **/api/blogs** | **Returns all active blog posts** |
| **GET** | **Retrieve list of ‘latest’ posts** | **/api/latest** | **Returns all active ‘latest’ posts** |
| **GET** | **Retrieve list of meals** | **/api/meals** | **Returns all active meals** |
| **GET** | **Retrieve list of news posts** | **/api/news** | **Returns all active news posts** |
| **GET** | **Retrieve list of rooms** | **/api/rooms** | **Returns all active rooms** |
| **GET** | **Retrieve list of testimonials** | **/api/testimonials** | **Returns all active testimonials** |
| **GET** | **Retrieve list of trips** | **/api/trips** | **Returns all active trips** |
| **GET** | **Retrieve single blog** | **/api/blogs/:blogCode** | **Returns single blog instance, identified by the title passed to the request URL** |
| **GET** | **Retrieve single ‘latest’ post** | **/api/latest/:latestCode** | **Returns single ‘latest’ post instance, identified by the title passed to the request URL** |
| **GET** | **Retrieve single meal** | **/api/meals/:mealCode** | **Returns single meal instance, identified by the mealName passed to the request URL** |
| **GET** | **Retrieve single news post** | **/api/news/:newsCode** | **Returns single poster, identified by the posterName passed to the request URL** |
| **GET** | **Retrieve single room** | **/api/rooms/:roomCode** | **Returns single room instance, identified by the name passed to the request URL** |
| **GET** | **Retrieve single testimonial** | **/api/testimonials/:testimonialCode** | **Returns single testimonial instance, identified by the person passed to the request URL** |
| **GET** | **Retrieve single trip** | **/api/trips/:tripCode** | **Returns single trip instance, identified by the code passed to the request URL** |
| **POST** | **Create single blog** | **/api/blogs/** | **Creates single blog instance** |
| **POST** | **Create single ‘latest’ post** | **/api/latest/** | **Creates single ‘latest’ post instance** |
| **POST** | **Create single meal** | **/api/meals/** | **Creates single meal instance** |
| **POST** | **Create single news post** | **/api/news/** | **Creates single poster** |
| **POST** | **Create single room** | **/api/rooms/** | **Creates single room instance** |
| **POST** | **Create single testimonial** | **/api/testimonials/** | **Creates single testimonial instance** |
| **POST** | **Create single trip** | **/api/trips/** | **Creates single trip instance** |
| **PUT** | **Update single blog** | **/api/blogs/:blogCode** | **Updates single blog instance, identified by the title passed to the request URL** |
| **PUT** | **Update single ‘latest’ post** | **/api/latest/:latestCode** | **Updates single ‘latest’ post instance, identified by the title passed to the request URL** |
| **PUT** | **Update single meal** | **/api/meals/:mealCode** | **Updates single meal instance, identified by the mealName passed to the request URL** |
| **PUT** | **Update single news post** | **/api/news/:newsCode** | **Updates single poster, identified by the posterName passed to the request URL** |
| **PUT** | **Update single room** | **/api/rooms/:roomCode** | **Updates single room instance, identified by the name passed to the request URL** |
| **PUT** | **Update single testimonial** | **/api/testimonials/:testimonialCode** | **Updates single testimonial instance, identified by the person passed to the request URL** |
| **PUT** | **Update single trip** | **/api/trips/:tripCode** | **Updates single trip instance, identified by the code passed to the request URL** |
| **DELETE** | **Delete single blog** | **/api/blogs/:blogCode** | **Deletes single blog instance, identified by the title passed to the request URL** |
| **DELETE** | **Delete single ‘latest’ post** | **/api/latest/:latestCode** | **Deletes single ‘latest’ post instance, identified by the title passed to the request URL** |
| **DELETE** | **Delete single meal** | **/api/meals/:mealCode** | **Deletes single meal instance, identified by the mealName passed to the request URL** |
| **DELETE** | **Delete single news post** | **/api/news/:newsCode** | **Deletes single poster, identified by the posterName passed to the request URL** |
| **DELETE** | **Delete single room** | **/api/rooms/:roomCode** | **Deletes single room instance, identified by the name passed to the request URL** |
| **DELETE** | **Delete single testimonial** | **/api/testimonials/:testimonialCode** | **Deletes single testimonial instance, identified by the person passed to the request URL** |
| **DELETE** | **Delete single trip** | **/api/trips/:tripCode** | **Deletes single trip instance, identified by the code passed to the request URL** |
|  |  |  |  |

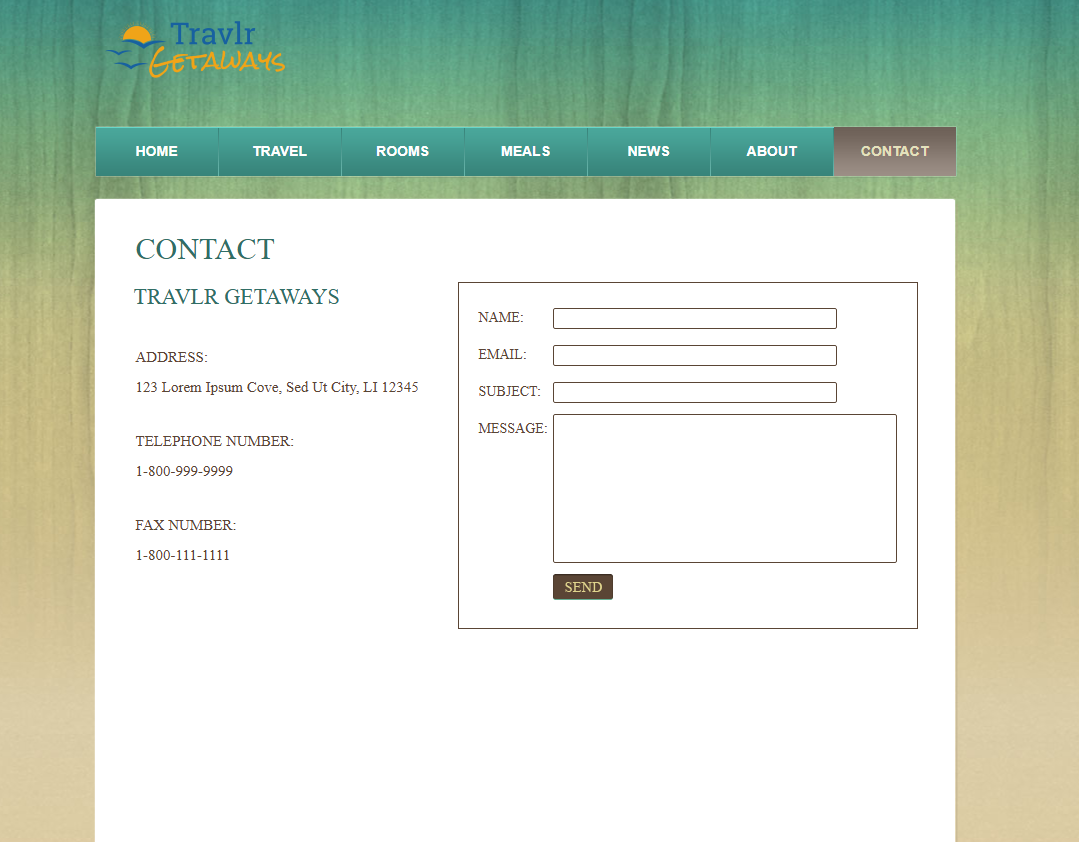
## The User Interface

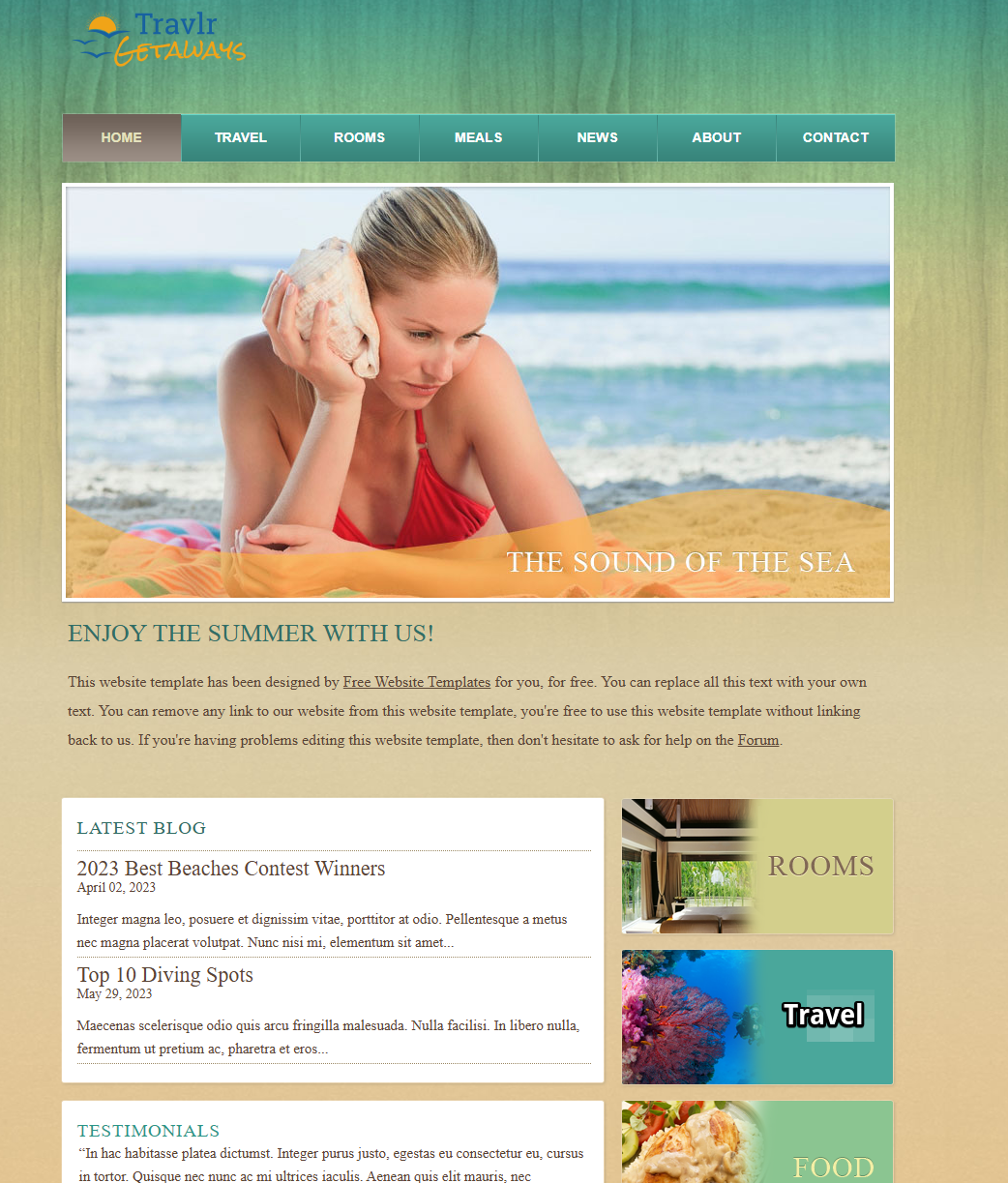
The images display the user interface where administrators can add a new trip and edit existing trips in the database. The form includes fields such as Trip Name, Destination, Start Date, End Date, Price, and Description. The system validates inputs before allowing submission to ensure data integrity. The administrator can change trip details such as pricing, availability, and included services. Once modifications are made, clicking "Save Changes" updates the trip details in the MongoDB database.

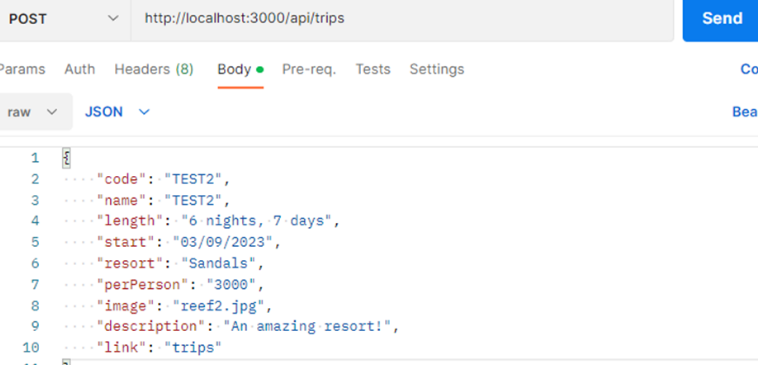












**Angular vs. Express Project Structure**

The **Angular project structure** follows a component-based approach, where each UI element is encapsulated within reusable components. Angular applications are structured with:

* **Modules**: Organize the app into functional sections (e.g., app.module.ts).
* **Components**: Define UI elements (e.g., trip-list.component.ts for trip listings).
* **Services**: Handle API calls and business logic (trip.service.ts manages API interactions).
* **Routing**: Facilitates seamless navigation (app-routing.module.ts manages URL paths).

In contrast, the **Express.js project structure** follows a server-side, MVC (Model-View-Controller) architecture:

* **Routes**: Define API endpoints (e.g., trips.routes.js).
* **Controllers**: Handle business logic for requests (trips.controller.js).
* **Models**: Define database schemas (trips.model.js for trip data structure).
* **Middleware**: Process HTTP requests (e.g., authentication, error handling).

**SPA Advantages Over Traditional Web Applications**

A Single Page Application (SPA) dynamically loads content without full-page reloads, providing a smooth, responsive experience. Compared to traditional multi-page applications (MPAs), SPAs:

* Offer **faster interactions** since they request only necessary data, not full pages.
* Improve **user experience** by minimizing delays and transitions.
* Reduce **server load** since most UI rendering occurs client-side.

**Testing API Integration**

To ensure the Angular SPA communicates effectively with the Express.js backend, the following testing methods were used:

1. **GET Requests Testing**
   1. Used tools like *Postman* and *browser console logs* to verify data retrieval from /api/trips.
   2. Ensured correct trip data was displayed on the UI.
2. **PUT Requests Testing**
   1. Modified trip details in the UI and checked the API request payload.
   2. Confirmed database updates by fetching the modified trip data.
3. **Error Handling & Debugging**
   1. Implemented error handling for failed API calls (e.g., network issues).
   2. Verified that invalid inputs return appropriate error messages.

By integrating automated tests and manual API verification, the system ensures a seamless user experience while maintaining data integrity.

In conclusion, the **Travlr Getaways** project demonstrates the implementation of a modern and efficient web application using the **MEAN stack** (MongoDB, Express.js, Angular, and Node.js). By leveraging this technology stack, the system provides a seamless experience for both travelers and administrators, ensuring responsive interactions and secure data handling.

The **customer-facing multi-page application (MPA)** was designed to offer a user-friendly interface for browsing, booking, and managing trips. Meanwhile, the **administrator single-page application (SPA)** facilitates real-time content management, allowing quick updates to trip listings, user information, and bookings. The adoption of Angular for the admin panel enhances performance by minimizing page reloads, improving overall responsiveness.

From a **technical perspective**, the project adheres to best practices in web development, including:

* **Optimized database structure** using MongoDB and Mongoose ODM to efficiently manage travel-related data.
* **Robust API design** with clearly defined RESTful endpoints, ensuring smooth communication between the front-end and back-end.
* **Security implementations**, including authentication via **JWT (JSON Web Tokens)** and secure database queries to protect user data.
* **Scalability and performance considerations**, such as caching mechanisms and load balancing strategies, to support future growth.

The **testing phase** confirmed that the application successfully integrates all components, ensuring smooth data flow between the client, server, and database. API testing using **Postman** validated data retrieval and updates, while front-end testing ensured correct rendering of dynamic content.

Overall, the **Travlr Getaways** project showcases an effective full-stack application that meets functional and performance requirements. The system is designed to be **scalable, secure, and user-friendly**, providing a strong foundation for future enhancements, such as AI-driven travel recommendations, multi-language support, and additional third-party API integrations.

This project serves as a testament to the power of modern web technologies in delivering efficient and engaging digital experiences.