

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 |
| --- | --- | --- | --- |
| 3.0 | 02/23/25 | Jose Lara Hernandez | User Interface: Summarized the Angular project structure and how it compares to the Express project structure. |

## 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)

Travlr Getaways is building a full-stack web application to bring together a customer-facing website, a secure database, and an admin dashboard for managing operations. The goal is to create a smooth, scalable, and high-performing platform that makes booking getaways easy for travelers while simplifying management for administrators. The application will use the MEAN (MongoDB, Express.js, Angular, Node.js) stack, a reliable JavaScript framework that allows for seamless data flow and efficient performance. The customer-facing website, built with Angular, will allow users to browse travel packages, search destinations, book trips, and manage their profiles. On the backend, Node.js and Express.js will handle API requests, with MongoDB storing user and booking data securely. The admin SPA will give staff an easy way to manage listings, bookings, and customer inquiries, all with role-based authentication for security. This approach makes sure that Travlr Getaways delivers a fast, secure, and user-friendly platform for both customers and administrators.

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

Developing the Travlr Getaways web application comes with several important considerations. These constraints define the boundaries of the project, guaranteeing that it aligns with both business and technical goals while staying within practical limits.

1. Technology Stack Considerations:

* Database Flexibility – MongoDB’s schema-less structure is useful for adaptability, but additional validation is needed to maintain data integrity.
* Backend Performance – Node.js is excellent for handling multiple simultaneous requests but may not be as efficient as other languages like Java or C# for highly complex computations.

2. Security Requirements:

* Authentication & Access Control – A secure login system using JWT, along with role-based access control (RBAC) to manage user permissions appropriately.
* Data Protection – Encrypting sensitive information both in transit and at rest to prevent unauthorized access.

3. Performance & Scalability Challenges:

* Managing Server Load – Since Node.js operates on a single-threaded model, strategies such as clustering or load balancing may be required to handle high traffic.
* Database Efficiency – As data grows, performance optimizations like indexing and query optimization will be necessary to maintain fast response times.

4. Frontend & User Experience Considerations:

* Responsiveness Across Devices – The customer-facing site must function well on desktops, tablets, and smartphones, requiring thorough testing.
* SPA Performance – The admin dashboard is designed as a Single-Page Application (SPA), which improves user experience but may result in longer initial load times if not optimized effectively.

5. Deployment & Hosting Constraints:

* Hosting Infrastructure – The platform must be hosted on a system that fully supports the MEAN stack while balancing cost and performance.
* Resource Management – Limited server resources may restrict advanced features such as auto-scaling and high-availability configurations.

## [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 follows a three-tier architecture, consisting of the Client, Server, and Database layers. Below is a breakdown of the primary components and their interactions.

1. Client Layer (Frontend)

The Client Layer handles all user interactions, providing a responsive interface for travelers. Key components include:

* Web Browser – Acts as the main interface through which users interact with the application.
* Client Session – Manages user sessions, for secure access and smooth transitions between different parts of the site.
* Traveler Portfolio – Stores user-specific information such as travel history, preferences, and saved trips.
* Graphic Library – Supports rendering of dynamic visuals.

2. Server Layer (Backend)

The Server Layer is responsible for processing client requests, managing sessions, and interacting with the database. It includes:

* Server Session – Maintains session data for active users, ensuring continuity and state management across user interactions.
* Traveler Database – Enables data retrieval and storage specific to traveler details and bookings.

3. Database Layer

The Database Layer manages the storage, retrieval, and organization of application data.

* MongoDB – A NoSQL database used to store user data, bookings, and travel information in a flexible, schema-less format.

Component Interactions:

* The Client Layer communicates with the Server Layer through secure API requests, sending user inputs and receiving data in response.
* The Server Layer interfaces with the Database Layer to query, update, or delete data as needed for user operations.
* While the Client Layer does not directly communicate with the Database Layer, any data the client needs—such as travel packages, booking history, or user profiles—is requested through the server. The server then queries the database, retrieves the necessary data, and sends it back to the client. This separation guarantees data security and integrity by controlling all database interactions through the server.

### Sequence Diagram



User Request Initiation (Client-Side):

* A user initiates an action, such as signing in, retrieving trip details, or accessing admin functions, through the Browser/View/Template in the Angular frontend. The client-side controller captures the request and invokes a service function to handle data retrieval or submission. The HTTP Client (Angular HttpClient) then sends an HTTP request to the designated backend route in the Express server.

Backend Processing (Server-Side):

* The backend route in Express receives the request and directs it to the appropriate controller function, which determines whether further interaction with the database is necessary. If data retrieval or manipulation is required, the controller invokes a service function that utilizes Mongoose ODM to interact with MongoDB.

Database Interaction (Data-Tier):

* The Model layer (Mongoose ODM) converts the request into a MongoDB query. MongoDB processes the query, retrieving or updating the necessary data. Once the operation is complete, MongoDB sends the data back to the Model layer for further processing before returning it to the controller.

Data Return & UI Update:

* The controller finalizes the response and sends it back to the frontend using res.json(). The HTTP client processes the response and assigns it to the application’s state. The frontend dynamically updates the view, displaying the retrieved or modified data to the user.
* This process applies across various functionalities such as user authentication (Sign In), trip retrieval, and administrative tasks. For instance, when a user logs in, the system verifies credentials, generates an authentication token, and grants access. Similarly, when retrieving trip details, the request follows the same structured flow from frontend to database and back. Admin functionalities, such as managing bookings, follow the same logic, ensuring secure and efficient data transactions.

## Class Diagram



**Classes and their roles:**

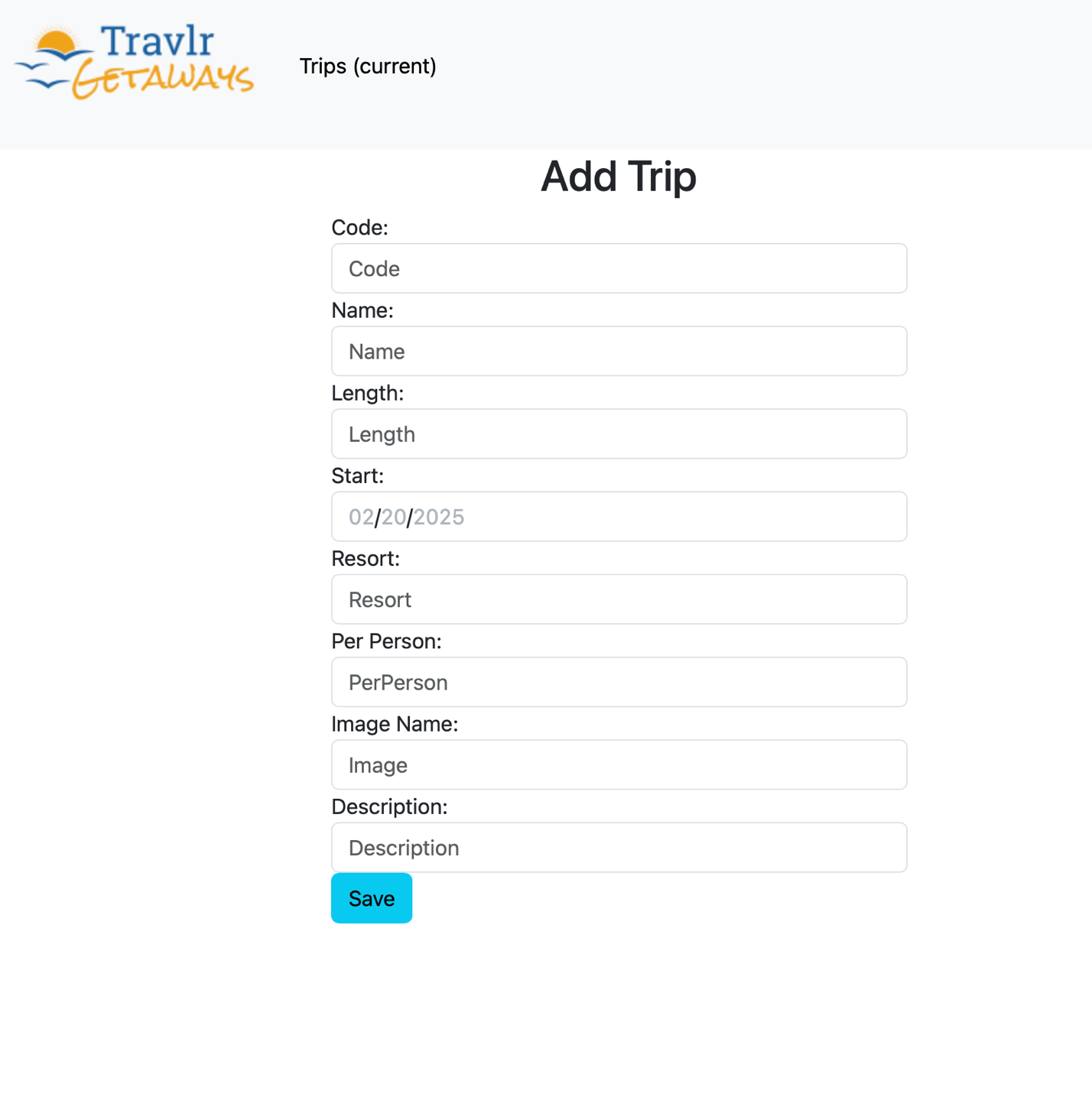
1. Itinerary**- Represents a traveler’s trip details, including the total price, total miles covered, and any stopovers.**
2. TravellerInfo**- Stores traveler-specific details such as membership number, frequent airline, member status, and membership club information.**
3. TripInfo**- Manages essential trip information, including start and return dates, origin, and destination.**
4. FlightInfo**- Contains details about flights, such as airline name, seat class, and price.**
5. HotelInfo**- Stores hotel booking details, including the hotel name, star rating, location, number of rooms requested, and price.**
6. CruiseInfo**- Manages cruise bookings by tracking the cruise name, cabin type, and price.**
7. Membership\_Admin**- Handles membership validation, point allocation, and frequent flyer program interactions.**
8. Travel\_Agent**- Facilitates trip bookings, allowing users to book flights, hotels, and cruises as part of a travel package.**
9. FlightBooking**- Manages flight reservations, retrieving flight details based on traveler information.**
10. HotelBooking**- Handles hotel reservations, retrieving booking details based on traveler information.**
11. CruiseBooking**- Manages cruise reservations, linking traveler details to cruise itineraries.**
12. MemberAccount**- Stores additional account-related information, including companion details for group travel.**

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

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve all trips | /api/trips | Returns list of all trips |
| **GET** | Retrieves a specific trip | /api/trips/:tripId | Returns the details of specific trips by ID |
| **POST** | Create new trip | /api/trips | Adds new trip to database |
| **PUT** | Updates a trip | /api/trips/:tripId | Updates trip details based on trip ID |
| **DELETE** | Deletes trip | /api/trips/:tripId | Deletes trip from database |
| **GET** | Retrieves user profile | /api/users/:userId | Returns user account information |
| **POST** | Creates a new user account | /api/users/register | Registers new users in the system |
| **POST** | User login | /api/users/login | Authenticates users and returns a JWT Token |
| **GET** | Retrieves flight options | /api/flights | Returns available flight details |
| **GET** | Retrieves hotel options | /api/hotels | Returns available hotel options |
| **POST** | Book a trip | /api/bookings | Creates new booking for user |
| **GET** | Retrieves booking information | /api/bookings/:bookingId | Returns information on specific booking |

## The User InterfaceA screenshot of a website AI-generated content may be incorrect.

A screenshot of a trip

AI-generated content may be incorrect.

The structure of an Angular project is different from an Express project because they serve separate purposes. Angular is a front-end framework, meaning it handles everything the user sees and interacts with in the browser. It’s made up of components, services, and routing modules that work together to create a seamless experience. Components manage the UI, services handle data and business logic, and routing allows users to navigate between different pages without actually reloading the entire application. This makes Angular perfect for building Single Page Applications (SPAs), where content updates dynamically rather than forcing full page refreshes like a traditional web application. Express, on the other hand, is a back-end framework that runs on Node.js and is responsible for handling server-side logic. It follows a structure that includes routes, controllers, middleware, and models. Routes define how different API endpoints respond to requests, controllers process those requests and interact with the database, and models define how data is structured. Unlike Angular, which deals with rendering content for the user, Express focuses on data management, authentication, and serving API responses to the front-end. These two frameworks work together in a full-stack application: Express provides the data, and Angular displays it in a user-friendly way.

When I was testing the Angular SPA to make sure it communicated properly with the Express API, I used Postman to send GET and PUT requests. First, I tested the API separately by sending requests to fetch trip listings and verify that the expected data was coming back from the database. Then, I used PUT requests to update trip details and checked that the changes were being reflected correctly in the database. After confirming that the API worked on its own, I moved on to testing the Angular front end. I checked the browser console and network tab to see if Angular was properly making HTTP requests and receiving the correct responses. When something didn’t work as expected, I debugged by checking the API response structure and making sure the Angular service functions were correctly handling the data. Since SPAs load content dynamically, a single issue in the API or front-end logic could break functionality. By testing the API first, I was able to confirm that the back end was returning the correct data before troubleshooting the front end. This process helped me catch errors early, like incorrect endpoint URLs or missing authentication tokens. In the end, everything came together, and I was able to verify that the Angular app was successfully fetching, displaying, and updating data from the Express API, making for a fully functional and interactive application.