

<Name of Software Application>

# **CS 465 Project Software Design Document**

Version 2.0

## Table of Contents

[**CS 465 Project Software Design Document** 1](#_Toc211635484)

[Table of Contents 2](#_Toc211635485)

[Document Revision History 2](#_Toc211635486)

[Design Constraints 3](#_Toc211635487)

[Sequence Diagram 7](#_Toc211635488)

[2. Class Diagram: 8](#_Toc211635489)

[3. API Endpoints 11](#_Toc211635490)

[Auth 11](#_Toc211635491)

[Trips (public) 12](#_Toc211635492)

[Bookings (customer) 12](#_Toc211635493)

[Admin (CRUD on trips) 13](#_Toc211635494)

[The User Interface 14](#_Toc211635495)

[Summarization of Angular Project Structure and the SPA (Extended Version) 18](#_Toc211635496)

[Angular Application Structure and Modularity 19](#_Toc211635497)

[Integration of Third-Party Libraries and UI Enhancements 19](#_Toc211635498)

[Testing, Debugging, and Developer Workflow 20](#_Toc211635499)

[Project Reflection 20](#_Toc211635500)

[References 21](#_Toc211635501)

## [Document Revision History](#_heading=h.lnxbz9)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | |  | | --- | | 10/02/2025 |  |  | | --- | |  | | Shekhar Chaudhary | Initial draft for Milestone One submission |

|  |  |  |  |
| --- | --- | --- | --- |
| 2.0 | 10/15/2025 | Shekhar Chaudhary | Final Document, submitted |

**Executive Summar**

The Travlr Getaways web application will be built using the **MEAN stack** (MongoDB, Express.js, Angular, and Node.js) to meet the client’s requirements for a scalable and maintainable full stack solution. The system will be composed of two primary user interfaces:

* **Customer-facing Website**: A responsive, public-facing interface that allows users to browse travel packages, search for destinations, view detailed trip information, and manage bookings. This site prioritizes usability, speed, and accessibility across devices.
* **Administrator Single Page Application (SPA)**: An Angular-based admin dashboard that enables staff to manage trip data, update destination details, and monitor customer activity in real-time. The SPA provides rich, interactive functionality with seamless transitions and reduced load times compared to traditional multi-page applications.

By leveraging the MEAN stack, the application will be highly modular, allowing efficient updates and future scalability. The architecture also supports RESTful APIs that ensure smooth communication between the client and server layers.

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

The Travlr Getaways application must be developed under several design constraints:

1. **Technology Stack**: Development is constrained to the MEAN stack, which standardizes the tools and ensures consistency across the front end and back end. This reduces flexibility to use alternative frameworks or databases but ensures better integration.
2. **Performance and Scalability**: The application must support potentially thousands of users simultaneously. This requires careful database indexing, efficient API design, and optimized front-end rendering.
3. **Security**: Since customer information and booking data are sensitive, the application must implement secure authentication, authorization, and data encryption practices. These requirements may add complexity and additional development time.
4. **Time and Scope**: The application must be delivered within the academic project timeframe. This constraint limits the number of features that can be included in the first release, requiring prioritization of core functionalities.

**Implications:** These constraints guide design decisions, ensuring that the application remains technically feasible and aligns with client expectations. However, they also require trade-offs such as focusing on essential features first and leaving advanced functionality for future iterations.

**System Architecture View: Component Diagram**

The Travlr Getaways web application will follow a layered architecture aligned with the MEAN stack. The main components include:

* **MongoDB (Database Layer)**: Stores trip information, user data, and booking records in a flexible NoSQL document format.
* **Express.js (Server Layer)**: Provides a lightweight back-end framework for defining RESTful API endpoints and handling server logic.
* **Angular (Client Layer)**: Powers the customer-facing website and administrator SPA with dynamic, component-based front-end views.
* **Node.js (Runtime Environment)**: Executes server-side JavaScript and manages communication between the database and client.

**Relationships between components:**

* Angular clients (both customer-facing and admin SPA) send HTTP requests to the Express.js server.
* The Express.js server processes these requests and communicates with MongoDB to retrieve or update data.
* Node.js provides the runtime environment that supports Express.js and manages concurrent client requests.
* RESTful APIs connect the client and server layers, ensuring modular and reusable interactions.

This architecture enables scalability, modularity, and maintainability, ensuring the Travlr Getaways application can evolve alongside client needs.



### Sequence Diagram

Purpose: To illustrate the flow of logic in the Travlr Getaways web application.

Flow Summary:  
The application follows a three-tier architecture using the MEAN stack (MongoDB, Express.js, Angular, Node.js). The diagram should demonstrate how data flows between the client, server, and database through the following key interactions:

1. User Login – Angular client sends a POST /api/auth/login request to Express. Express validates credentials in MongoDB and returns a JWT.  
2. View Trips – Client sends GET /api/trips, server queries MongoDB, returns JSON.  
3. Book Trip – Authenticated user submits POST /api/bookings, server validates and creates a booking.  
4. Admin Update Trip – Admin sends PUT /api/admin/trips/:id, updates MongoDB, and returns confirmation.

Description: The diagram should illustrate request flow between Angular frontend, Express middleware, Node.js runtime, and MongoDB database. Label each layer clearly. Insert the completed sequence diagram image below.

A diagram of a diagram

AI-generated content may be incorrect.

HTTP clint

MongoDB

M/C

Route

Controller

Route

View

## 2. Class Diagram:

Purpose: To illustrate the JavaScript classes and services that form the foundation of the Travlr Getaways web application.

Classes & Services Overview:  
- User: Represents users with profile and credentials. One user can have many bookings.  
- Trip: Contains trip details such as title, summary, price, dates, and seats.  
- Booking: Links users to trips and stores booking status.  
- AuthService: Handles JWT authentication and session logic.  
- TripService: Manages trip listings and updates.  
- AdminService: Handles admin CRUD operations.

Angular Interfaces (Frontend DTOs):  
interface UserDTO { id: string; email: string; role: 'customer' | 'admin'; }  
interface TripDTO { id: string; title: string; price: number; startDate: string; endDate: string; seats: number; }  
interface BookingDTO { id: string; userId: string; tripId: string; partySize: number; status: 'pending' | 'confirmed' | 'canceled'; }

Description: The class diagram should depict relationships such as one-to-many (User → Booking). Service classes and controllers should be distinct. Insert the class diagram image below.

A diagram of a class diagram

AI-generated content may be incorrect.

Classes & Services: Domain entities include User, Trip, and Booking with obvious relationships (a User can have many Bookings; a Booking links one User to one Trip). Service classes encapsulate business logic and API orchestration: AuthService (auth/JWT), TripService (listing, retrieval, update/validation), and AdminService (admin-only create/update/delete). Angular components consume these via HTTP; Express controllers call the services; services access MongoDB models.

Angular side (DTO/Interfaces you render)

export interface UserDTO {

id: string;

email: string;

role: 'customer' | 'admin';

}

export interface TripDTO {

id: string;

title: string;

summary: string;

price: number;

startDate: string; // ISO

endDate: string; // ISO

seats: number;

published: boolean;

images: string[];

availableSeats: number; // computed on server

}

export interface BookingDTO {

id: string;

userId: string;

tripId: string;

partySize: number;

status: 'pending' | 'confirmed' | 'canceled';

}

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

Exposing RESTful endpoints enables the app to participate in a larger ecosystem (web SPA, mobile, future partners). All responses are JSON; secure endpoints require a Bearer JWT in Authorization header.

## Auth

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **URL** | **Purpose** | **Notes** |
| POST | /api/auth/login | Sign in with email + password; returns JWT + profile | Body: {email, password}. 200 on success; 401 on invalid creds. |
| GET | /api/auth/me | Get current user profile | Requires JWT; useful for restoring sessions. |
| POST | /api/auth/refresh | Issue a new short-lived access token | Requires refresh token (if implemented). |
| POST | /api/auth/logout | Invalidate refresh token (optional) | Server-side revocation list if implemented. |

## Trips (public)

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **URL** | **Purpose** | **Notes** |
| GET | /api/trips | List trips with optional filters | Query params: q, priceMin, priceMax, startAfter, duration, limit, page. |
| GET | /api/trips/:id | Get a single trip by ID | Returns 404 if not found. |

## Bookings (customer)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | **Method** | **URL** | **Purpose** | **Notes** | | POST | /api/bookings | Create a booking for a trip | Body: { tripId, partySize }. Requires JWT (customer). Validates seat availability. | | GET | /api/bookings | List current user’s bookings | Requires JWT (customer). | | GET | /api/bookings/:id | Get a single booking | AuthZ: owner or admin. | | DELETE | /api/bookings/:id | Cancel a booking | Transitions status to canceled; idempotent. | |

## Admin (CRUD on trips)

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **URL** | **Purpose** | **Notes** |
| POST | /api/admin/trips | Create a new trip | Body: {title, summary, price, startDate, endDate, seats, images, ...}. Admin JWT required. |
| PUT | /api/admin/trips/:id | Update an existing trip | Partial updates allowed; server validates fields; 409 on stale version if ETag used. |
| PATCH | /api/admin/trips/:id/status | Toggles publish/unpublish | Useful for workflow without editing full document. |
| DELETE | /api/admin/trips/:id | Delete a trip | Soft delete recommended (flag) to preserve booking history. |

| **Method** | **URL** | **Purpose** | **Notes** |
| --- | --- | --- | --- |
| GET | /api/health | Liveness/readiness | Returns {status: "ok", uptime}. No auth. |

**Common Status Codes:** 200 OK, 201 Created, 204 No Content, 400 Bad Request (validation), 401 Unauthorized, 403 Forbidden, 404 Not Found, 409 Conflict (optimistic concurrency), 422 Unprocessable Entity (semantic validation), 500 Internal Server Error.

## The User Interface

1. **Unique Trip (Customer UI) — “Everest Base Camp Explorer”**
   * Card/list view with hero image, title, summary, price, dates, and “View Details” button.
   * Detail page shows full description, seat availability, gallery, and “Book Now.”
   * Evidence it’s unique (e.g., a badge “Added by Shekhar” or a distinct slug).

A screenshot of a computer

AI-generated content may be incorrect.

1. **Admin – Edit Trip Screen**
   * Reactive form (Angular) with fields: title, summary, price, startDate, endDate, seats, images, published.
   * Inline validation messages (e.g., “Price must be greater than 0”, “Start date must be before end date”).
   * “Save” (PUT), “Cancel” (router back), and form pristine/dirty indicators.

A screenshot of a computer

AI-generated content may be incorrect.

1. **Admin – Add Confirmation**
   * Success toast/snackbar (“Add Trip updated”), and the updated trip visible in the admin list.
   * Optimistic UI refresh or a reload of the list/detail showing the new values.

A screenshot of a computer

AI-generated content may be incorrect.

1. Log-in page

A screenshot of a computer

AI-generated content may be incorrect.

Angular Frontend Architecture (Admin SPA)

A screenshot of a computer

AI-generated content may be incorrect.

The Angular application in this project functions as a **Single Page Application (SPA)** dedicated to the administrative (backend) interface of Travlr Getaways. Its responsibilities include:

* Providing a dynamic, responsive user interface for creating, editing, and deleting trip offerings
* Handling client-side routing so that navigation within the admin interface does not trigger full page reloads
* Communicating with the backend REST API (Node/Express) through HTTP (GET, POST, PUT, DELETE) to fetch and update trip, booking, and user data
* Managing authentication state (e.g. storing/verifying JWT) to protect admin-only routes and features
* Using TypeScript-built services, components, and modules to structure the frontend code in a modular, maintainable fashion
* Mapping backend DTOs to frontend models (interfaces / classes) so that data received from the API can be used cleanly and consistently in Angular components

**Typical Angular project structure** (in this project) includes:

* components/ — UI components (e.g. trip list, edit form, login)
* services/ — Angular services for API calls (e.g. TripService, AuthService)
* models/ — TypeScript interfaces or classes for domain objects (User, Trip, Booking)
* app-routing.module.ts — Client-side route definitions
* assets/, environments/, etc. — Supporting files

Because this is an SPA, once the site loads, the UI updates dynamically based on user interactions and data responses, without requiring full page refreshes. This results in a smoother and more responsive admin experience.

## ****Summarization of Angular Project Structure and the SPA (Extended Version)****

The core of this project is built using the **MEAN stack** — MongoDB, Express, Angular, and Node.js — which collectively power a responsive and dynamic single-page application (SPA). The backend integrates **Node.js** and **MongoDB** to manage and serve trip-related information for authenticated administrative users, while **Angular** handles the entire client-side experience. This unified stack allows seamless communication between the frontend and backend, providing a smooth, app-like user experience directly in the browser.

### ****Angular Application Structure and Modularity****

Angular is designed for scalability and modularity, and this application leverages that fully. The project’s structure is component-based, meaning every feature — from trip listings and navigation to form handling — is encapsulated as a reusable module. Each component handles its own logic, view, and style, which can be dynamically inserted into the main layout without rewriting code.This modularity doesn’t just streamline development; it also makes maintenance, debugging, and future expansion significantly easier.

Angular’s **routing system** is another key feature of the SPA. Instead of loading new HTML pages from the server, Angular uses **client-side routing** to dynamically update content within the same page. This architecture allows users to navigate through multiple sections (for example, viewing trips, editing itineraries, or managing users) without full page reloads. Each route communicates with the backend API through a centralized **service layer**, which handles HTTP calls such as GET, POST, PUT, and DELETE. This separation of concerns keeps the app organized and the codebase clean.

Unlike traditional server-rendered applications like those built purely with Express, Angular’s SPA model allows the user interface to remain constant while only the necessary data changes. This approach not only enhances performance but also creates a more fluid, native-app-like user experience.

### ****Integration of Third-Party Libraries and UI Enhancements****

To accelerate development and improve the visual design, the application incorporates **Bootstrap** and other third-party UI libraries. These libraries provide pre-built components such as dropdowns, modals, navigation bars, and dynamic buttons, which can be customized and reused across multiple views. The result is a consistent, responsive interface that adapts to different screen sizes — a key requirement for modern web apps.

By combining Angular’s modular architecture with Bootstrap’s UI toolkit, the project achieves both **functionality and maintainability**. Refactoring or extending features becomes straightforward, as developers can update individual components without impacting others.

### ****Testing, Debugging, and Developer Workflow****

Testing and debugging play a vital role in ensuring the reliability of the SPA. Before full deployment, **API routing** is verified using **Postman**, which allows simulation of all backend HTTP requests and responses. This includes testing login endpoints, data retrieval from MongoDB, and CRUD operations for trip management.

Once the routes are confirmed, the frontend can be tested in a browser with developer tools enabled. Console logs (console.log()) and JSON outputs (.json({ message: "..." })) help trace potential issues, particularly when verifying backend connections or troubleshooting asynchronous data flow. Because the project is built entirely in **JavaScript**, debugging across both the frontend and backend becomes intuitive and efficient.

For deeper issue tracking, **Visual Studio Code** (or Visual Studio) serves as the main development environment. The inclusion of Angular and Node.js extensions enhances productivity, enabling live debugging, syntax highlighting, and real-time linting. These tools make identifying and resolving integration bugs faster, especially when multiple components or services depend on each other.

### ****Project Reflection****

What stands out in this project is how Angular’s SPA architecture transforms the user experience. Instead of reloading entire pages or waiting for server responses to render new content, the app dynamically updates relevant sections — creating a seamless, application-like feel. From a developer’s perspective, this structure promotes cleaner design, modular growth, and rapid feature iteration.

Moreover, by combining **Angular**, **Node.js**, and **MongoDB**, this project demonstrates how full-stack development frameworks can work together to build highly interactive, data-driven web applications that are both performant and scalable.

### ****References****

MongoDB. (2022). What is the MEAN stack? Introduction & examples. Retrieved September 24, 2022, from <https://www.mongodb.com/mean-stack>

Google Developers. (2023). Angular Overview. Retrieved from https://angular.dev/overview

Mozilla Developer Network (MDN). (2023). Single Page Applications (SPA). Retrieved from https://developer.mozilla.org/en-US/docs/Glossary/SPA

freeCodeCamp. (2023). The MEAN Stack Explained – MongoDB, Express, Angular, and Node. Retrieved from https://www.freecodecamp.org/news/introduction-to-mean-stack/

W3Schools. (2024). Angular Components and Routing. Retrieved from https://www.w3schools.com/angular/angular\_routing.asp

Thank you!!!