

# Milestone One

# **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 | 07/17/25> | Darrell Walker | Initial draft of Milestone One completed. Added Executive Summary, Design Constraints, and System Architecture View sections for Travlr Getaways project. |
| 1.1 | 08/02/25 | Darrell Walker | Added completed Sequence Diagram, Class Diagram with descriptions, and API Endpoints section for Travlr Getaways project. |
| 1.2 | 08/14/2025 | Darrell Walker | Added architecture + sequence content, SPA/Express comparison, GET/PUT testing, and UI screenshots |
| 1.3 | 08/17/2025 | Darrell Walker | Completed the Project UI/layout to meet customer needs via Wireframe. |

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

The Travlr Getaways web application will be built using the MEAN stack (MongoDB, Express.js, Angular, and Node.js), providing a robust, scalable, and maintainable full stack solution. This architecture ensures that both front-end and back-end development share a common language (JavaScript) for efficiency and consistency.

The system will feature two primary components:

* A customer-facing website that allows users to browse, search, and book travel destinations. This interface will offer a user-friendly experience accessible from desktop and mobile devices.
* An administrator single-page application (SPA) that provides Travlr Getaways staff with tools to manage trips, review bookings, and perform administrative tasks efficiently. This SPA will leverage Angular’s framework for dynamic, real-time updates without full page reloads.

By adopting this architecture, Travlr Getaways will deliver a seamless, interactive experience to both travelers and administrators, ensuring scalability and performance for future growth.

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

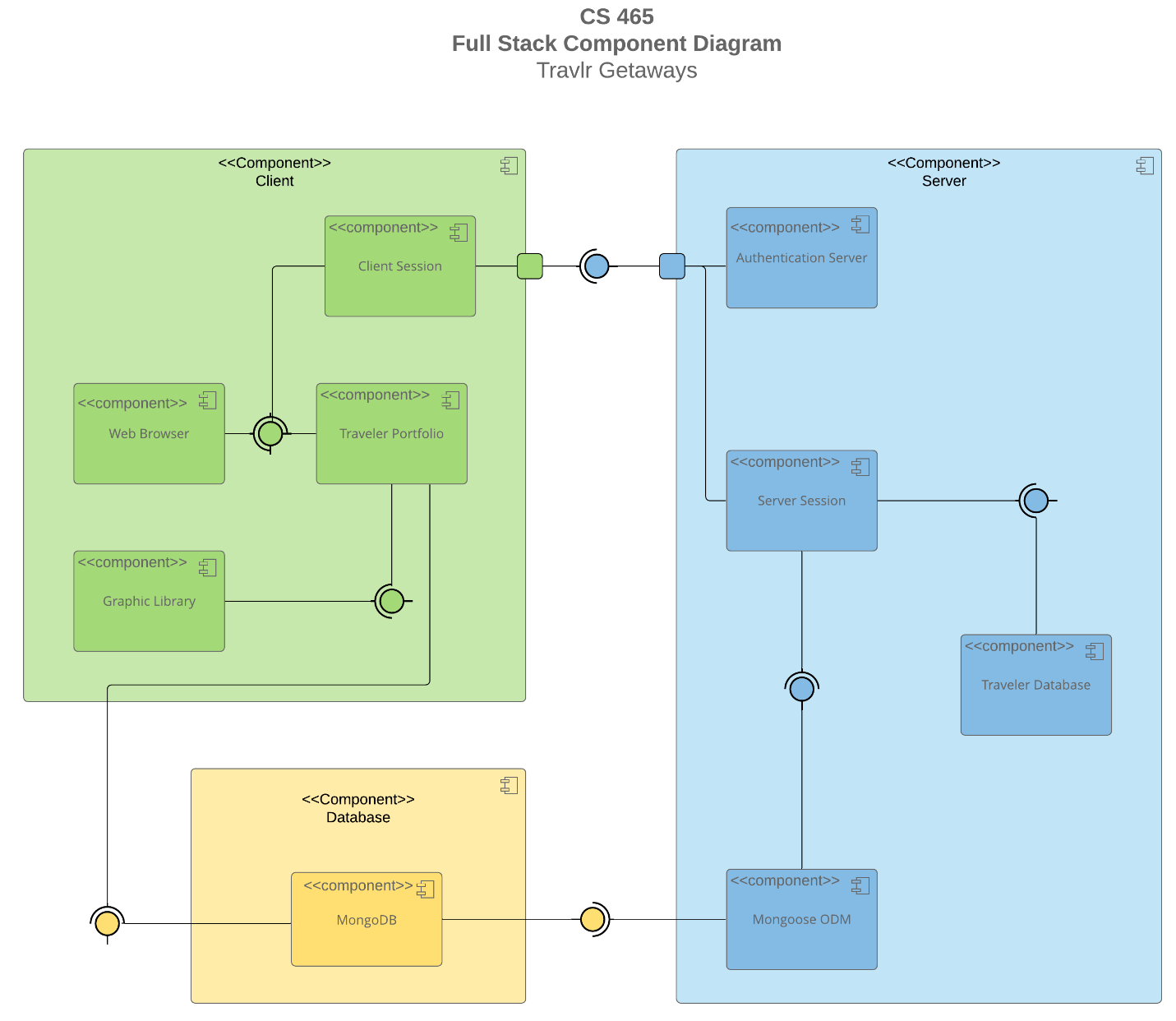
The development of the Travlr Getaways application is subject to the following constraints:

* Technology constraint: The application will use the MEAN stack exclusively, which restricts the back-end to Node.js and Express and the front-end to Angular.
* Deployment constraint: The application must be deployable on a cloud-based hosting platform that supports Node.js, such as Heroku or AWS Elastic Beanstalk.
* Performance constraint: The web application must scale efficiently to support potentially thousands of concurrent users without degradation in performance.
* Security constraint: The application must adhere to web security best practices, including secure authentication, authorization, and protection against common vulnerabilities.
* Time constraint: The project must be completed within the defined academic term, requiring careful prioritization and iteration.

**Implications**:  
These constraints guide architectural decisions and ensure that the project stays within scope, uses appropriate technologies, meets performance expectations, and maintains security and reliability standards. The exclusive use of the MEAN stack, for example, dictates data storage format (JSON-like documents) and influences API design. The deployment constraint requires compatibility with cloud platforms that support Node.js and MongoDB environments.

## [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 overall system architecture follows a classic three-tier design, integrating front-end, back-end, and database components, connected over secure HTTP(S) protocols.

Significant components and their relationships:

* Angular Front-End SPA:
  + Hosts the customer-facing interface and administrator dashboard.
  + Communicates with the Express.js API server via HTTP requests (REST API).
* Express.js Server (API Layer):
  + Acts as middleware, handling routing, authentication, and API requests.
  + Processes client requests and forwards them to the MongoDB database when needed.
* MongoDB Database:
  + Stores all persistent data, including trip details, user accounts, and booking records.
  + Interfaces with the Express.js server using MongoDB drivers and libraries (e.g., Mongoose).

The diagram illustrates a clear separation of concerns:

* The front-end Angular SPA interacts with users and sends requests to the Express server.
* The Express server manages business logic, handles API endpoints, and interacts with the MongoDB database for data storage and retrieval.

This modular architecture ensures scalability, maintainability, and flexibility for enhancements and future integrations.

### Sequence Diagram

A screenshot of a graph

AI-generated content may be incorrect.

The Travlr Getaways application follows a standard MEAN stack request–response cycle between the client, server, and database. The sequence begins when an Actor interacts with the application by navigating to a specific route in the browser.

1. **Routing and View Rendering (Client-Side – AngularJS)**
   * The user request triggers the AngularJS route handler.
   * The router redirects the user to the correct view, which is linked to an Angular controller.
   * The controller initializes and prepares to retrieve the necessary data for display.
2. **Controller to Service Call (Client-Side)**
   * The Angular controller calls a service responsible for retrieving the required data.
   * This service makes an HTTP request to the server-side API endpoint.
3. **HTTP Request to Controller Function (Server-Side – Node.js / Express)**
   * The request reaches the Express.js server, which routes it to the correct controller function.
   * The controller calls a **server-side service** to retrieve the requested data.
4. **Data Retrieval from Database (Data Tier – MongoDB via Mongoose ODM)**
   * The server-side service sends a query to the MongoDB database using the Mongoose Object Data Modeling library.
   * MongoDB processes the request and returns the result via the Mongoose callback.
5. **Returning Data to the Client**
   * The Express controller sends the result back to the Angular service as a JSON response.
   * The Angular service receives a promise/results object containing the requested data.
   * The controller assigns the data to the Angular scope, updating the view.
6. **Displaying Results**
   * The Angular view is updated dynamically without a page reload, showing the retrieved data to the user.

## Class Diagram

A computer screen shot of a diagram

AI-generated content may be incorrect.

The Travlr Getaways web application is made up of JavaScript classes that represent the main parts of the travel booking process. These classes work together to handle trip planning, travel service details, membership management, and booking steps.

Itinerary – Stores the main trip information, including total price, total miles, and any stopovers. It connects all the parts of the trip into one plan.

CruiseInfo, FlightInfo, HotelInfo – Store details for each travel service type.

CruiseInfo holds the cruise name, cabin type, and price.

FlightInfo holds the flight name, seat class, and price.

HotelInfo holds the hotel’s star rating, location, number of rooms requested, and price.

TripInfo – Holds important trip details like the start date, return date, starting location, and destination.

MemberAccount – Stores membership information such as member number, frequent airline, membership status, and club membership.

Membership\_Admin – Handles membership management. It has functions to give credit points, check points, and confirm membership information.

Travel\_Agent – Handles the booking process. It books entire packages, flights, hotels, or cruises using functions like BookPackage(), BookFlight(), BookHotel(), and BookCruise().

TravellerInfo – Stores information about the traveler, like how many companions are traveling, and links to bookings.

HotelBooking, FlightBooking, CruiseBooking – Handle bookings for each type of travel service. They work with traveler information and service details to make and confirm reservations.

These classes are designed to work in an organized way. They keep different parts of the system separate but connected, making it easier to manage travel services, store customer and membership data, and complete bookings through one central system.

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

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve all trips | /api/trips | Returns a list of all available trips |
| **GET** | Retrieve single trip | /api/trips/:tripId | Returns details for a specific trip using the trip ID from the URL |
| **POST** | Create new trip | /api/trips | Creates a new trip in the database (admin only) |
| **PUT** | Update existing trip | /api/trips/:tripId | Updates details for a specific trip (admin only) |
| **DELETE** | |  | | --- | |  |   Delete trip | /api/trips/:tripId | Deletes a specific trip from the database (admin only) |
| **GET** | Retrieve all bookings | /api/bookings | Returns a list of all bookings for the logged-in user or all users if admin |
| **POST** | Create booking | /api/bookings | Creates a new booking for the logged-in user |
| **GET** | Retrieve single booking | /api/bookings/:bookingId | Returns details for a specific booking |
| **PUT** | Update booking | /api/bookings/:bookingId | Updates details for a specific booking |
| **DELETE** | Cancel booking | /api/bookings/:bookingId | Deletes or cancels a specific booking |

## The User Interface

Has been adjusted to meet the customer requirements via wireframe.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a cell phone

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a computer screen

AI-generated content may be incorrect.

**Executive summary**This web application uses the MEAN stack: MongoDB for data storage, Express and Node.js for the REST API, and Angular for a single-page application used by administrators. The customer-facing side is a simple Express site that shows read-only trip information and serves public pages. It makes lightweight GET requests to the API when needed and keeps server logic separate from the UI. The administrator side is an Angular SPA that handles sign-in, route-protected pages, editable forms, and fast, in-place updates. After a successful login, the server issues a JSON Web Token (JWT). Angular stores this token and sends it with each protected request, while the API enforces validation and access rules before writing to MongoDB. This separation keeps the public site fast and the admin tools responsive and secure.

**System architecture view**  
At a high level, the browser runs the Angular SPA. It contains feature components for listing, viewing, and editing trips, services that call the API over HTTPS, an HTTP interceptor that attaches the JWT, and route guards that block admin screens when the user is not authenticated. On the server, Express maps URLs to controllers. The authentication controller uses a local strategy to check credentials on login and a JWT strategy to protect write operations. The trips controller handles reading and updating trip data through Mongoose models. The Trip and User schemas define required fields and include methods for password hashing, password checks, and token creation. MongoDB stores the documents for trips and users. Requests flow from Angular to Express, from controllers to Mongoose and MongoDB, and then back up the stack to update the UI.

**Sequence of key interactions**  
For sign-in, the Angular login form sends the email and password to the API. The API validates the user and returns a JWT, which Angular saves (for example, in local storage). For public browsing, the trips list view asks the API for trips and renders the returned array without a full page reload. For admin editing, Angular first retrieves a specific trip to prefill the form. When the user saves changes, Angular sends a PUT request with the Authorization header set to “Bearer <token>.” The API verifies the token, updates the matching trip (for example, by code) using Mongoose, and returns the updated document. Angular then refreshes only the affected parts of the page and shows a success message or any validation errors.

User interface and comparison of Angular to Express  
Angular focuses on the client experience. It organizes code into components for the views and services for the data calls, uses client-side routing for smooth navigation, and adds features like instant form validation, error messaging, and optimistic UI updates. Express focuses on the server responsibilities. It wires middleware for JSON parsing, URL-encoded bodies, CORS, and Passport, exposes routes under /api, and keeps business rules in controllers and schemas. Compared to a simple server-rendered web app, the SPA is richer because it changes views instantly without reloading, updates only what changed, and centralizes data access and error handling in services. This makes the app feel faster and keeps the code easier to maintain.

**Testing the SPA with the API for GET and PUT**Testing starts at the API and then moves to the UI. In Postman, first confirm that GET /api/trips returns a 200 with an array and that GET /api/trips/:code returns the correct trip or a 404 when it does not exist. Next, verify security by calling PUT /api/trips/:code without a token and expecting a 401 Unauthorized. Then log in with valid credentials, capture the returned JWT, and repeat the PUT with the Authorization header set. The server should accept the request and return the updated trip. Negative tests should include sending bad input or required fields that are missing to make sure the API returns a clear 400 with validation messages. After the API passes, open the Angular admin screens and confirm that the list view loads from GET, that the edit form prepopulates from GET by code, and that saving issues a PUT with the token. Clearing the stored token should cause protected requests to fail with 401, and the SPA should react by redirecting to login or showing an authorization error. This end-to-end process proves that the SPA and API work together correctly for both reading and updating data.