

<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 | <07/23/23> | <Joshua Perez> | <Module 3-2: Executive Summary, Design Constraints, System Architecture View> |
| 1.1 | <07/30/23> | <Joshua Perez> | <Module 5-2: Sequence Diagram, Class Diagram, API Endpoints> |
| 1.2 | <08/20/23> | <Joshua Perez> | <7-2 Project Submission> |

## 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)

I am writing to present our comprehensive solution for building a travel booking website tailored to meet the requirements of Travlr Getaways. Our team of software developers has carefully analyzed your software requirements and wireframes, and we are excited to propose an architecture that aligns perfectly with your vision. Our proposed architecture for the travel booking web application leverages the power of the MEAN stack, incorporating MongoDB, Express.js, Angular, and Node.js to ensure a robust and efficient platform. This full-stack technology enables seamless integration of front-end and back-end components, facilitating a dynamic and responsive user experience. The MEAN stack is an ideal choice for developing your travel booking website. It offers several key advantages, such as real-time data interaction, rapid development, and scalability, ensuring that your application can handle a growing customer base with ease. Additionally, the use of Angular in building single-page applications enhances user engagement and responsiveness. Our team will construct a customer-facing web application that will allow users to create accounts, search for travel packages based on location and price point and make reservations with Travlr Getaways. The front end will be designed according to the provided wireframe, providing a visually appealing and user-friendly experience. Customers will be able to access their trip itineraries before their journeys, enhancing their overall travel experience. The administrator SPA will be accessible only to authorized Travlr Getaways personnel. It will feature a secured interface enabling administrators to maintain a customer base, manage available trip packages, and set pricing for each item and package. The SPA will allow administrators to perform CRUD operations efficiently, ensuring streamlined management of the website's backend data.

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

There are many constraints that will be present for the development of the Travlr Getaways application that will need to be accounted for. One of the main constraints in any project is time, the time constraint poses a challenge in terms of meeting the project deadlines. It may require us to prioritize certain features and functionalities in order to ensure that the core requirements are implemented within the specified timeframe. Another constraint is the budget, which impacts the allocation of resources and technology choices. Optimal utilization of available resources is crucial to ensure that development efforts remain within the budget. Another constraint to consider is security and privacy, as the application deals with sensitive user data, strict security measures are essential. Implementing encryption and secure authentication mechanisms is critical to protect user information and prevent unauthorized access. Compliance with data protection regulations must also be ensured. While there are many constraints to consider, one that has the potential to cause some issues is the potential for increased user traffic over time, therefore the application must be designed to handle scalability gracefully. Scalability considerations must be considered at the database, server, and application levels to accommodate future growth without performance degradation.

## [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 of the web application for Travlr Getaways can be described based on the component diagram provided. The three major components are the Client component, Server component, and the Database component. The Client component represents the customer-facing side of the web application and includes the Web Browser, Client Session, Traveler Portfolio, and Graphic Library. The Server component represents the backend of the web application and includes the Authentication Server, Server Session, Traveler Database, and Mongoose ODM. The Database component represents the MongoDB database, which serves as the storage for various application data. The client component has several relationships defined under its umbrella. The Web Browser interacts with both the Client Session and Traveler Portfolio components. This relationship indicates that users access the application through the web browser, and the Client Session manages user sessions, while the Traveler Portfolio handles traveler-specific data and actions. The Traveler Portfolio also interacts with the Graphic Library component, indicating that the Traveler Portfolio uses the Graphic Library's resources to render graphical assets and content on the customer-facing interface. Additionally, the Traveler Portfolio component connects to the MongoDB component, implying that it interacts with the database to store and retrieve traveler-related data. Under the server components relationships, we have the Authentication Server, which interacts with the Client Session component. The server session component also connects to both the Client Session and Traveler Database components. The Traveler Database connects to the Mongoose ODM, which indicates that Mongoose serves as the Object Data Modeling tool to structure and interact with MongoDB for storing and retrieving data efficiently. Under the Database component relationships, we have the MongoDB component which connects to the Mongoose ODM. This signifies that Mongoose is the bridge between the application's server and the MongoDB database, allowing for structured interactions and data manipulation.

### Sequence Diagram

A diagram with text and a cross

Description automatically generated with medium confidence**Sign In Process:**

1. Browser/View/Template (AngularJS) - User interacts with the sign-in form.
2. HTTP Client (AngularJS) - Sends a request to the appropriate route on the server.
3. Controller/Model (Node.js/Express) - Handles the sign-in request. Then communicates with the MongoDB data tier. Verifies user credentials and generates an authentication token.
4. HTTP Client (AngularJS) - Receives the authentication token.
5. Browser/View/Template (AngularJS) - Stores the authentication token for future requests. Navigates to the appropriate view (e.g., user dashboard).

**Trips Process:**

1. Browser/View/Template (AngularJS) - User interacts with the trips interface.
2. HTTP Client (AngularJS) - Requests the list of trips from the server.
3. Controller/Model (Node.js/Express) - Handles the request for trips data. Retrieves trip data from the MongoDB data tier.
4. HTTP Client (AngularJS) - Receives the list of trips.
5. Browser/View/Template (AngularJS) - Displays the list of trips in the view.

**Admin Interactions:**

1. Browser/View/Template (AngularJS) - Admin user interacts with the admin interface.
2. HTTP Client (AngularJS) - Sends a request for admin-related data to the server.
3. Controller/Model (Node.js/Express) - Handles the admin request. Retrieves admin-related data from the MongoDB data tier.
4. HTTP Client (AngularJS) - Receives admin-related data.
5. Browser/View/Template (AngularJS) - Displays admin-related data in the view.

## Class Diagram

A diagram of a travel censored class

Description automatically generated

**TravellerInfo** Class

Properties:

totalprice (float)

totalmiles (int)

stopover (string)

**CruiseInfo** Class:

Properties:

name (string)

cabintype (string)

price (float)

**MemberAccount** Class:

Properties:

membernumber (int)

frequent\_airline (string)

memberstatus (int)

memberclub (string)

**FlightInfo** Class:

Properties:

name (string)

seatclass (string)

price (float)

**TripInfo** Class:

Properties:

starting\_date (int)

returning\_date (int)

origin (string)

destination (string)

**HotelInfo** Class:

Properties:

name (string)

star (int)

location (string)

roomsrequested (int)

price (float)

**Membership\_Admin** Class:

Methods:

creditpoints(itinerary) (bool)

getpoints(membernum, frequent\_airline) (int)

validate(membernum, frequent\_airline) (bool)

**Itinerary** Class:

Methods:

BookPackage(Itinerary) (Itinerary)

BookFlight(Itinerary) (FlightInfo)

BookHotel(Itinerary) (HotelInfo)

BookCruise(Itinerary) (CruiseInfo)

**HotelBooking** Class:

Methods:

getHotel(TravelerInfo, HotelInfo) (HotelInfo)

**Travel\_Agent** Class:

Properties:

companionnum (int)

**FlightBooking** Class:

Methods:

getFlight(TravelerInfo, FlightInfo) (FlightInfo)

**CruiseBooking** Class:

Methods:

getCruise(TravelerInfo, CruiseInfo) (CruiseInfo)

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

<Exposing RESTful endpoints is a design approach to enable an application to participate in a larger ecosystem. Document each endpoint in the table below, including the HTTP method, purpose, URL, and notes.>

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | <Retrieve list of things> | </api/things> | <Returns all active things> |
| **GET** | <Retrieve single thing> | </api/things/:thingId> | <Returns single thing instance, identified by the thing ID passed on the request URL> |
| **POST** | <Authenticate user / generate authentication token> | </api/auth/signin> | <Requests a username and password, returns authentication token> |
| **POST** | <Allow a user to book a trip> | </api/trips/book> | <Requests authentication token and ID of trip, returns success message and confirmation> |
| **PUT** | <Update details of a specific trip> | </api/trips/:tripId> | <Requests authentication token and updates details of a trip, returns success message> |
| **PUT** | <Update user profile information> | </api/profile> | <Requests authentication token and updates details of a user’s profile, returns success message> |
| **DELETE** | <Delete a specific trip> | </api/trips/:tripId> | <Requests authentication token and deletes trip, returns success message> |

## The User Interface

I noticed a clear distinction in the project structure between the Angular front end and the Express HTML customer-facing page. The Angular project embraced a modular and component-driven approach, with organized directories for components, services, models, and modules. In contrast, the Express HTML page structure felt simpler, catering primarily to rendering dynamic content directly to the client. The Angular project structure is organized around the concept of a Single Page Application (SPA). The typical structure includes folders such as src, e2e, app, assets, environments, and files such as index.html, angular.json, tsconfig.json, package.json, styles.css. While the Express project structure for a server-side application might include folders like routes, controllers, models, config, public, and files such as app.js, index.js, package.json.

A Single Page Application (SPA) provides a more seamless and dynamic user experience compared to a simple web application. In a SPA, most resources are loaded once, and subsequent interactions involve fetching data from the server and updating the content on the page without reloading the entire page. This reduces server requests and enhances user engagement. Additionally, the front-end and back-end could communicate asynchronously, enhancing overall performance. However, this dynamic behavior also poses challenges.

Ensuring that the SPA's interaction with the API for data retrieval (GET) and modification (PUT) functioned as intended involved meticulous testing. Rigorous testing using tools like Postman became vital to validating the API's behavior. I began testing the SPA interaction using Postman to test several facets of the process, such as the user authentication system, the login endpoint which took in an input for email address and password and returned the JWT token, and a couple of other components.