

Travlr Getaways Web Application

# **CS 465 Project Software Design Document**

Version 2.0

## 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  1.1  2.0  3.0 | 01/26/2024  02/09/2024  02/09/2024  02/28/2024 | Devin Perry  Devin Perry  Devin Perry  Devin Perry | First drafts of: Executive Summary; Design Constraints; and System and Architecture View: Component Diagram.  Removed instructions.  Added Sequence diagram + explanation. Added Class diagram + explanation. Filled in API Endpoints table.  Summarized project in “The User Interface” section |

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

Travlr Getaways envisions a fresh web application to enhance customer experiences and streamline administrative tasks. Travlr had some specific software requirements in mind, so our proposed architecture for the web application is built upon the robust and versatile MEAN stack. The MEAN stack acronym comprises of MongoDB, Express.js, Angular, and Node.js. Each piece of the MEAN stack solution synergizes with one another. A NoSQL database (MongoDB), a server-side application framework (Express.js), a client-side framework (Angular), and a runtime environment (Node.js), all these combined provide a great solution for our client's diverse needs. The proposed solution has two integral components, one being a customer-facing website and the other an administrator single-page application (SPA). The customer-facing website’s purpose is to deliver a user-friendly interface, incorporating key features such as real-time booking, interactive maps, and personalized recommendations. Along with the customer side, the administrator SPA gives staff management tools, enabling seamless oversight of bookings, customer interactions, and content updates.

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

The development of the Travlr Getaways web application is governed by a set of essential constraints that can restrict and shape the trajectory of the project. These constraints serve as guiding principles, ensuring realistic expectations and an efficient approach to application development. Firstly, the most common restraint of any project is time. The project will operate within a defined timeline, which will require strategic planning and prioritization of features. This constraint pushes us to adopt an agile development methodology. The agile workflow allows for iterative improvements and timely delivery of essential functionalities. The next key constraint will have to be about finances. Financial considerations are integral to project success. Budget constraints guide our decision-making process, influencing aspects of the project such as technology choices, number of workers, and resource allocation. To optimize within the specified budget, we need to commit to cost-effective technologies and minimizing unnecessary features. The last of the main constraints will be a technological constraint. Compatibility and functionality, among other things related to technology, impose constraints on the selection of tools and frameworks. Knowing about these limitations ensures that we can have seamless integration with legacy systems and enhance overall system stability.

Considering the constraints mentioned, we can make some implications based on them. Time constraints necessitate a pragmatic approach to feature development. Core features should be prioritized, allowing for incremental enhancements post-launch. This allows for a quicker launch and the opportunity for good updates down the road. Budget limitations mandate efficient resource allocation. This includes optimizing development and testing processes to maximize output within the specified financial framework. Lastly, we can assume that technological limitations demand a careful selection of tools to ensure compatibility and smooth integration. This constraint reinforces our commitment to a stable and reliable application that aligns with existing technological infrastructure.

## [System Architecture View](#_heading=h.44sinio)

### Component Diagram



The diagram above offers an in-depth analysis of the various components within the design for the Travlr Getaways web application. First, I want to discuss each of the components and their purposes. In the Client component box, there are four components within it. The Web Browser is the entry point for user interaction. Next is the Client Session that manages user-specific session data. The Traveler Portfolio facilitates the storage and retrieval of traveler-related information. Finally, the Graphic Library provides graphical resources for an enriched user interface. The next overarching box is the Server component box, which also has four components. The Authentication Server handles user authentication processes. Also, the Server Session which manages server-specific session data. Third is the Traveler Database that stores comprehensive traveler data. Lastly in this box is Mongoose ODM (Object Data Modeling) which acts as an interface between MongoDB and the application, ensuring seamless data integration. The last “big” component box only has one component within it. The box is the database, and only contains MongoDB which is the NoSQL database that serves as the foundations for data storage.

Now that the components are identified and discussed, it’s time to look at the relationships between them. This comprehensive component diagram shows the intricate relationships between the various components, ensuring a well-orchestrated system that caters to both user experience and data integrity. The Web Browser component connects to the Traveler Portfolio component and the Client Session component. Interfaces denote the required connections for data exchange. The Traveler Portfolio component connects to the Graphic Library component and the MongoDB component. These connections illustrate the essential connections for graphical and data-related operations. The MongoDB component connects to the Mongoose ODM component, representing the integration of MongoDB into the application's data modeling. The Mongoose ODM component connects to the Server Session component, establishing a link between the server and the database for session management. The Server Session component further connects to the Traveler Database component, ensuring efficient data exchange for server-side operations. The Server Session component also connects to the Client Session component, involving a complex connection sequence that includes a square box denoting internal processes, a required interface for communication, and another square box for additional processes. The Client Session component connects to the Authentication Server component, following a similar complex connection sequence, involving internal processes and a required interface for secure authentication.

A diagram of a diagram

Description automatically generated**Sequence Diagram**

The sequence diagram starts with the initiation by an actor, which in our case is a user. The actor selects a route, guided by the front-end router, leading to one of the site's browser/views/tem-plates. The chosen view triggers the corresponding controller, and then right after the controller renders the view and sends it back for display to the actor. Within the front-end controller, calls are made to functions in the HTTP service to retrieve specific information fragments. The outcomes of these functions are then sent back to the controller. In this case the HTTP service acts as a sort of bridge that connects the frontend to the backend through API calls directed to specific routes. On the backend, the router receives the route from the frontend, prompting the invocation of the appropriate backend controller. Once invoked, the backend controller interacts with the database using Mongoose, executing queries, and fetching data. The controller then forwards the obtained data back up to the calling frontend HTTP service. Finally, MongoDB processes the query received from the numerous steps and generates a response that is then returned to complete the sequence.

## A diagram of a travel network Description automatically generatedClass Diagram

There are numerous different classes in the diagram above, and it is important to understand the relationships between the various classes and what each contain. There are three classes that share similarity which are CruiseInfo, FlightInfo, HotelInfo, that contain a name, string, and floats. The HotelInfo class also has two int variables. Each of these three info classes inherit the TripInfo class that has two ints and two strings. Many of these classes deal with pulling variables from one another and the rest can be seen in the diagram. The Itinerary class has aggregate relationships with the three Info classes I just mentioned. Another group of classes that share similarity with each other are the HotelBooking, FlightBooking, and CruiseBooking classes. These each have relationships with their corresponding Info classes which I discussed previously, as well as the TravelerInfo class. Another important relationship is that these three booking classes have a zero-to-many relationships with the Travel\_Agent class. This Travel\_Agent class also must have association with the four three main Info classes and the Traveler one, as well as a one-to-many relation connecting with the MembershipAdmin class. This MembershipAdmin class has an aggregate relationship with the MemberAccount class, and then the TravelerInfo class inherits the MemberAccount class.

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

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Gets all news | </api/news> | Returns all news |
| **GET** | Gets specific news | </api/news/:newsCode> | Returns specific news – identified by newsCode in URL. |
| **GET** | Gets all trips | </api/trips> | Returns all trips. |
| **GET** | Gets specific trip | </api/trips/:tripCode> | Returns specific trip – identified by tripCode in URL. |
| **GET** | Gets all rooms | </api/rooms> | Returns all rooms. |
| **GET** | Gets specific room | </api/rooms/:roomCode> | Returns specific room – identified by roomCode in URL. |
| **GET** | Gets all meals | </api/meals> | Returns all meals. |
| **GET** | Gets specific meal | </api/meals/:mealCode> | Returns specific meal – identified by mealCode in URL. |
| **PUT** | Update specific trip | </api/trips/:tripCode> | Updates specific trip – identified by tripCode in URL. |
| **DELETE** | Delete specific trip | </api/trips/:tripCode> | Deletes specific trip – identified by tripCode in URL. |
| **POST** | Add a specific trip | </api/trips> | Adds a new specific trip to the database. |
| **POST** | Login user | </api/login> | Accepts input for a specific user to allow access. |
| **POST** | Register user | </api/register> | Creates a new login for a specific user. |

## The User Interface

Unfortunately, my application is not running correctly on my machine due to unknown reasons. Therefore, I cannot provide screenshots that show functionality of the application.

The Angular project structure is designed to be dynamic and modular. Components encapsulate specific functionalities, services handle shared logic, and modules group related elements, promoting a modular and scalable architecture. In contrast, the Express project structure is often associated with server-rendered applications, where HTML is generated on the server. This creates a clear separation of concerns, with Angular focusing on the client-side and Express handling server-side functionality.

Single Page Applications offer more functionality versus simple web application interactions. SPAs enable seamless navigation, reduced server load, and easier maintenance. They allow dynamic content loading without full page reloads, resulting in a smoother user experience. The transition between views is seamless, contributing to a more polished and modern user interface. Despite these advantages, SPAs may have longer initial load times and can pose challenges for search engine optimization.

Testing the SPA's interaction with the API involves using tools for unit tests and for end-to-end testing in Angular. It's important to mock API responses during unit tests to isolate client-side logic and to use some other tools for manual test API endpoints. This helps in verifying that the API can handle GET requests to retrieve data and PUT requests to update data in the database.