

Travlr Getaways – J. Pierce Waren

# **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 4](#_Toc36198467)

[System Architecture View 5](#_Toc36198468)

[Component Diagram 6](#_Toc36198469)

[Sequence Diagram 7](#_Toc36198470)

[Class Diagram 8-9](#_Toc36198471)

[API Endpoints 10](#_Toc36198472)

[The User Interface 11](#_Toc36198473)-13

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

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 09/12/2023 | J. Pierce Waren | Completed the executive summary, design constraints, and system architecture view portions of the design document. |
| 2.0 | 09/27/2023 | J. Pierce Waren | Completed the sequence diagram, class diagram, and API endpoints portion of the design document. |
| 3.0 | 10/15/2023 | J. Pierce Waren | Completed the user interface portion of the design document. |

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

1. Architecture Overview:

* The MEAN stack (MongoDB, Express.js, Angular, Node.js) will be used for the creation of this web application that is used for both the customer and administrator interfaces. HTML, CSS, and JavaScript will be used for the creation of the user interface, while Angular will provide a structure for a seamless and dynamic user experience. Data will be stored and accessed from the MongoDB database with Express.js being used on the back end to ensure the data is being accessed reliably and consistently.

1. Customer Facing Portion of The Web Application:

* Back-End (Express and Node.js): In the architecture for the Travlr Getaways web applications, Node.js will be used as the runtime environment while Express.js will be used in developing RESTful APIs. These APIs will allow for interactions between the MongoDB database and the front end of the application. Express.js will handle all the important back-end tasks like user authentication, data management, handles the trip booking process, retrieval of trip/itinerary information, and many more critical tasks within the application.
* Front-End (Angular): For the front end of this application, angular will be used as the primary front-end framework. Since Angular is very modular, I will be able to create reusable components, which will make the application more efficient and maintainable. Angular’s built-in routing capabilities will also be used to allow for smooth navigation that allows the user to easily move between multiple web pages and views.
* Database (MongoDB): MongoDB will be used to store all the data for the application. This includes various trips, itinerary, and vacation information. To maintain simplicity and structure within the DB, data will be logically organized into collections. The data within these collections will be accessed and queried using MongoDB's query language, ensuring efficient and structured retrieval and manipulation of data for the application's functionalities.

1. Administrator Single-Page Application (SPA):

* The administrator single-page application (SPA) will be used as a separate interface that will allow Travlr Getaways staff to oversee and manage various aspects of the web application. This will include things like what content is displayed, user accounts, bookings, etc. The SPA will be developed using the Angular stack and proper authentication and authorization protocol will be used to improve security of the system. Additionally, the SPA will seamlessly interface with the previously discussed Node.js and Express.js backend, streamlining communication and data flow for administrative functions.

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

1. Probably the most common and prevalent design constraint would be staying within the budget. Although not specified within the scope of this project, I am sure that Travlr Getaways would have a proposed budget to stay under. Based on what the actual budget is, this constraint will heavily influence many important decisions like technology choices, scalability, number of developers needed, and what features can be included. It is always important to consider the cost of everything being developed and make sure the client has full transparency throughout the whole process.
2. Time is another significant design constraint. Although not specified in the scope of this assignment, the client, Travlr Getaways, will undoubtedly have a time deadline for the expected launch of the application. Adhering to this schedule might necessitate potential limitations on the number of features or complexity within the application before its release. To ensure that the development team stays on track, it is essential to conduct proper planning with a realistic timeframe as a guiding benchmark.
3. The third constraint I'll address is security. Given that the Travlr Getaways application will handle sensitive personal data, the risk of data theft or leaks is a significant concern. Ensuring the utmost security not only adds to development time but also requires strict adherence to regulatory standards like HIPAA or GDPR. The development team must prioritize security measures to safeguard user information effectively.
4. There is a plethora of other design constraints that could be discussed, but they will not have as heavy of an impact as the constraints previously discussed.

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

### Component Diagram



There are 3 major components that are included to make up the system architecture for the web application. These major components include the Database, Server, and Client. The client is accountable for presenting the UI and overseeing the user sessions. The server component will handle the back-end operations/functions like authentication, authorization, interactions with the database, handling server sessions, etc. Lastly, the database will serve as the storage place for all information and data related to the Travlr Getaways application. I will go into more detail about each component below.  
  
**Client**: The client component contains several additional components that include the Client Session, Web Browser, Traveler Portfolio, and Graphic Library. I will go into more detail for each of these subcomponents below.

* Graphics Library: The Graphics Library will provide any necessary graphics elements (photos, videos, etc.) that the client will need for rendering.
* Traveler Portfolio: This will contain all the unique data for the user pertaining to their travel file. This component will need to have communication with the graphic library to display all needed UI and graphics elements to the user.
* Client Session: Keeps track of the current user session on the client.
* Web Browser: This will be where the user interacts with the application and the UI will update dynamically depending on the task the user is performing. The browser has relationships with both the Traveler Portfolio and Client Session components. With proper communication between these elements the web browser should be able to update and display accurate and dynamic information to the user through a clean and simple UI.

**Database**: The database component only contains one other additional component, and it is the MongoDB. I will go into more detail for each of the MongoDB subcomponents below.

* MongoDB: This will be where all the Travlr Getaways data is stored for quick and easy access within the application. The MongoDB will also interact with the Mongoose ODM and Traveler Portfolio indicating these two components will draw from the MongoDB directly for any necessary data needed while the application is running.

**Server**: The client component contains several additional components that include the Authentication Server, Server Session, Traveler Database, and Mongoose ODM. I will go into more detail for each of these subcomponents below.

* Mongoose ODM: As previously discussed above, the Mongoose ODM will draw data directly from the MongoDB and hopefully streamline all interactions with the server.
* Traveler Database: Stores all the unique data related to the Travlr user. This would contain things like personal information and travel information.
* Server Session: This will handle all the user interactions on the server. To make this happen there will need to be a relationship with the Traveler Database and Client Session.
* Authentication Server: Handles user authentication and authorization. Allows additional user interaction once proper authority and access has been established with the client and server components.

### A diagram of a diagram Description automatically generatedSequence Diagram

The sequence diagram for Travlr Getaways, a travel website where users can view and book various trips, commences with the user, the primary actor. When a user interacts with the site, they initiate a specific action such as signing in, viewing/booking trips or performing admin interactions. This action leads to the initiation of the front-end router, which directs the user to the relevant view or template corresponding to their chosen route.

For instance, when a user attempts to sign in, the frontend router navigates them to the sign-in page. Within this view, the controller is activated, responsible for populating the template and generating the view that will be displayed to the user. To provide the necessary data for the view, the front-end controller communicates with the HTTP service, which, in turn, triggers API calls to the backend.

In the backend, the router receives the route from the frontend and identifies the appropriate backend controller to handle the user's request. For instance, during viewing/booking trips, the router calls the trips Controller. Subsequently, the backend controller engages with the MongoDB database using Mongoose to retrieve or manipulate data relevant to the request, such as trip information.

Once the MongoDB database processes the query, it returns the results to the backend controller, which, in turn, passes this data back to the calling frontend HTTP service. The HTTP service delivers the results to the frontend controller, facilitating the rendering of the view with the updated information, thus creating a responsive and dynamic user experience tailored to the specific action, whether it's signing in, viewing/booking trips or performing admin interactions

This sequence illustrates how the various processes, guided by the user's actions and the frontend and backend components, seamlessly come together to provide users with a smooth and interactive experience on Travlr Getaways.

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

**Itinerary**: The Itinerary class represents the structure of a user's travel itinerary. It contains information (attributes) such as total price, total miles, and stopover for a given trip. This class also has an aggregation association with the CruiseInfo, FlightInfo, and HotelInfo classes to provide more accurate information about the trip itinerary.

**CruiseInfo**: This class stores details (attributes) about cruise trips, including the ship's name, cabin type, and price. There is also a realization relationship with the CruiseBooking and Travel\_Agent classes. This class is also a child of the TripInfo class.

**TravellerInfo**: TravellerInfo is used to capture information about individual travelers. The only attribute in this class is the companionnum. This class has an aggregation association with the Travel\_Agent, HotelBooking, FlightBooking, and CruiseBooking classes. This is also a parent to the MemberAccount class.

**FlightInfo**: This class stores details (attributes) about flights, including the flight name, seat class, and price. There is also a realization relationship with the FlightBooking and Travel\_Agent classes. This class is also a child of the TripInfo class.

**HotelInfo**: This class stores details (attributes) about hotels, including the hotel's name, stars, location, requested room and price. There is also a realization relationship with the HotelBooking and Travel\_Agent classes. This class is also a child of the TripInfo class.

**TripInfo**: TripInfo is a parent class that encapsulates information about basic trip information that includes starting date, returning date, origin, and destination.

**Membership\_Admin**: This class represents administrative functionality for managing user memberships and administrative privileges. The given methods for this class will include credit points, get points, and validate. There is an aggregation relationship with the MemberAccount class as well as a 1 to many relationships with the Travel\_Agent class.

**HotelBooking**, **CruiseBooking**, **FlightBooking**: These classes are used to manage bookings for specific components of a trip, such as hotels, cruises, and flights. They all include a method to get to a specific hotel, flight, or cruise. All these classes have a 0 to many relationships with the TravellerInfo class as well as a realization relationship with their corresponding info class.

**Travel\_Agent**: The Travel\_Agent class represents a user or system entity responsible for facilitating trip bookings and interactions between travelers and the travel services. The methods included within this class are BookPackage, BookFlight, BookHotel, and BookCruise. All the given relationships with this class have already been listed previously.

**MemberAccount**: This class stores user account information (attributes), including member number, frequent airline, member status, and member club. This is also a child class of TravellerInfo, and it inherits their attributes.

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

| **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** | <Create a new list of things> | </api/things> | <Creates a new list of things> |
| **POST** | <Create a single thing> | </api/things/:thingId> | <Creates a single, identified by the thing ID passed on the request URL > |
| **PUT** | <Update an entire list of things> | </api/things> | <Updates and replaces an already existing list of things. All data from the existing list will be removed.> |
| **PUT** | <Update a single thing> | </api/things/:thingId> | <Updates and replaces an already existing thing. All data from the existing thing will be removed.> |
| **PATCH** | <Modify a list of things> | </api/things> | <Updates an existing list of things. This will not remove the previous list. It will only make the requested changes.> |
| **PATCH** | <Modify a single thing> | </api/things/:thingId> | <Updates an existing thing. This will not remove the previous list. It will only make the requested changes and modify the thing.> |
| **DELETE** | <Delete everything in a list of things> | </api/things> | <Deletes an entire list of things.> |
| **DELETE** | <Delete a single thing> | </api/things/:thingId> | <Deletes a single thing, identified by the thing ID passed on the request URL> |

## The User Interface

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

**How is the Angular project structure different from that of the Express HTML customer-facing page?**

Angular is primarily a client-side framework, and its project structure is designed around organizing client-side code. In the travlr project, you'll encounter several essential folders. The 'src' directory houses all the source code, 'assets' is where static resources are stored, 'app' contains services and components, and 'environments' holds configuration files.

On the other hand, the Express framework is responsible for server-side logic. In the Express project's structure, you'll typically find an 'app\_server' folder dedicated to server configuration, a 'routes' folder for managing route handlers, a 'views' directory where server-rendered HTML templates reside, and a 'public' folder designated for serving static assets. The primary distinction lies in the roles these frameworks play: Angular handles client-side code, while Express focuses on server-side logic.

**What are some advantages and disadvantages of the SPA functionality? What additional functionality is provided by a SPA compared to a simple web application interaction?**

Single Page Applications (SPAs) offer several advantages, including faster user experiences with seamless transitions and responsive designs. They reduce server load as they fetch data via APIs instead of entire HTML pages. SPAs also allow smooth navigation without full page reloads and can even support offline use through service worker caching. However, they face SEO challenges due to initial client-side rendering and may have longer initial load times. SPAs can be more complex to develop and maintain, especially for large applications, and their heavy client-side rendering can strain older or less capable devices. In return, SPAs provide dynamic loading, real-time updates, and a consistent, single-page user experience, making them suitable for interactive and responsive applications but requiring careful consideration of their trade-offs.

**What is the process of testing to make sure the SPA is working with the API to GET and PUT data in the database? What are some errors you ran into or what are some errors you could expect to run into?**

I utilized the MongoDB application in conjunction with Google Chrome's built-in developer tools to verify the correctness of the GET and PUT functions within the Travlr application. After making edits or adding trips, I consistently cross-referenced MongoDB to confirm that the corresponding changes were reflected in the database documents. Furthermore, I closely monitored the client and admin web pages to ensure they were dynamically updated to reflect any modifications. However, one issue I encountered was related to data formatting. Specifically, I unintentionally included a '$' sign in the cost field, which led to errors since the system expected an integer value.