

Travlr Getaways

**Software Design Document**

Version 3.0

## Table of Contents

[Software Design Document](#bookmark) 1

[Table of Contents](#bookmark1) 2

[Document Revision History](#bookmark2) 2

[Instructions](#bookmark3) 2

[Executive Summary](#bookmark4) 3

[Design Constraints](#bookmark5) 3

[System Architecture View](#bookmark6) 3

[Component Diagram](#bookmark7) 3

[Sequence Diagram](#bookmark8) 4

[Class Diagram](#bookmark9) 4

[API Endpoints](#bookmark10) 4

[The User Interface](#bookmark11) 4

## [Document Revision History](#bookmark12)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 09/18/22 | Jacob Winters | Added executive summary, design constraints, and system architecture view component diagram. |
| 2.0 | 09/30/22 | Jacob Winters | Completed sequence diagram, class diagram, and API endpoints. |
| 3.0 | 10/14/22 | Jacob Winters | Revised executive summary, system architecture view, and completed user interface section. |

## [Executive Summary](#bookmark13)

The client, Travlr Getaways, is looking to create a travel booking site that utilizes the architecture of a full stack web application. The application is designed to use the MEAN stack, consisting of MongoDB, Express, Angular, and Node.js. The customer side of the application will allow customers to create an account, search for travel packages by location or price point, book reservations with the travel agency, and be able to view their itineraries online at any time before their trip. Travlr Getaway administrators will be able to access an admin-only site, or administrator single-page application (SPA). The administrator page will provide the ability to maintain a customer base, available trip packages, and pricing information for each item and package.

## [Design Constraints](#bookmark14)

For a web-based application, there are several design constraints that can be identified. The initial HTML/CSS/JavaScript site template will have to be modified to be compatible with a MongoDB database. A NoSQL database with data models and a schema is used for the data files and storage. While this is helpful in storing travel booking trips, it requires that the database be populated manually in order to view the seeded collections and documents within the database. Furthermore, Travlr Getaways requests web tokens for web-login authentication, which will require utilizing Angular security features.

## [System Architecture View](#bookmark15)

### Component Diagram



The overall system architecture of the web application consists of three major components, the client, server, and database. Within the client component, there is four internal components, which is the web browser, client session, traveler portfolio, and graphic library. Moving on to the database component, we have the MongoDB component, which handles all database inquiries that will be passed onto the server. Finally, within the server component, we have the authentication server, server session, traveler database, and Mongoose ODM components.

The relationships between the system components work in varying ways. The client session component passes data along to the web browser and travel portfolio components. From here, the traveler portfolio sends information to the graphic library and then to the MongoDB database. Additionally, data from the client session is passed to the authentication server, which then goes through the server session and traveler database. Lastly, MongoDB provides output to the Mongoose ODM component, which then moves back to the server session component.

### Sequence Diagram

### Image

Based on the sequence diagram, the flow of logic within the web application moves in a linear pattern. Beginning with the client-side components, Angular routes allow us to handle the navigation between each view, or template. These consist of our individual components that define the visual aspects of the application, such as a TypeScript class and an HTML template. The controller maintains specific application data that attaches specific behavior to HTML elements. This is where we would implement a sign in process, utilizing a login controller that calls upon the HTTP client.

From here, the data moves to the server-side of the application, routing through a controller/model, and passing data to the MongoDB database in order for the application to authenticate the login information. After the process is requested, we have a callback function that allows the enclosing method to execute a promise object, i.e. to access information on the result of the wrapped operation. Depending on the result, data is passed back to the client-side of the application, where the user receives an updated view based on the assigned scope. At this point, the process essentially continues, where it will provide a view of the trip information, and change depending on the trip information within our database.

## Class DiagramImage

Based on the class diagram, many significant observations can be made as to how the JavaScript classes work together within the web application. Beginning with the interface, which is the Itinerary, we have our attributes and methods for booking trips or returning trip information. Next, there is an inheritance relationship between the Itinerary interface and the Travel\_Agent class, which contains a companion number attribute, stored as an integer value, and must be zero or many. From here, there is an association relationship with the MemberAccount class, which holds attributes for the member number (integer), frequent airline (string), member status (integer), and member club (string). Additionally, there is a multiplicity between these two classes of one or many. The Membership\_Admin class has an aggregation relationship with the MemberAccount class, containing attributes for credit points, a points getter method, and a validation method. The credit points attribute returns a bool value based on the itinerary. For the points getter and validation method, we utilize integers and strings from the MemberAccount class, and return these as integers and bool values, respectively. Notice that we draw a direct connection between the Itinerary interface and Membership\_Admin class, identifying the multiplicity of zero or many.

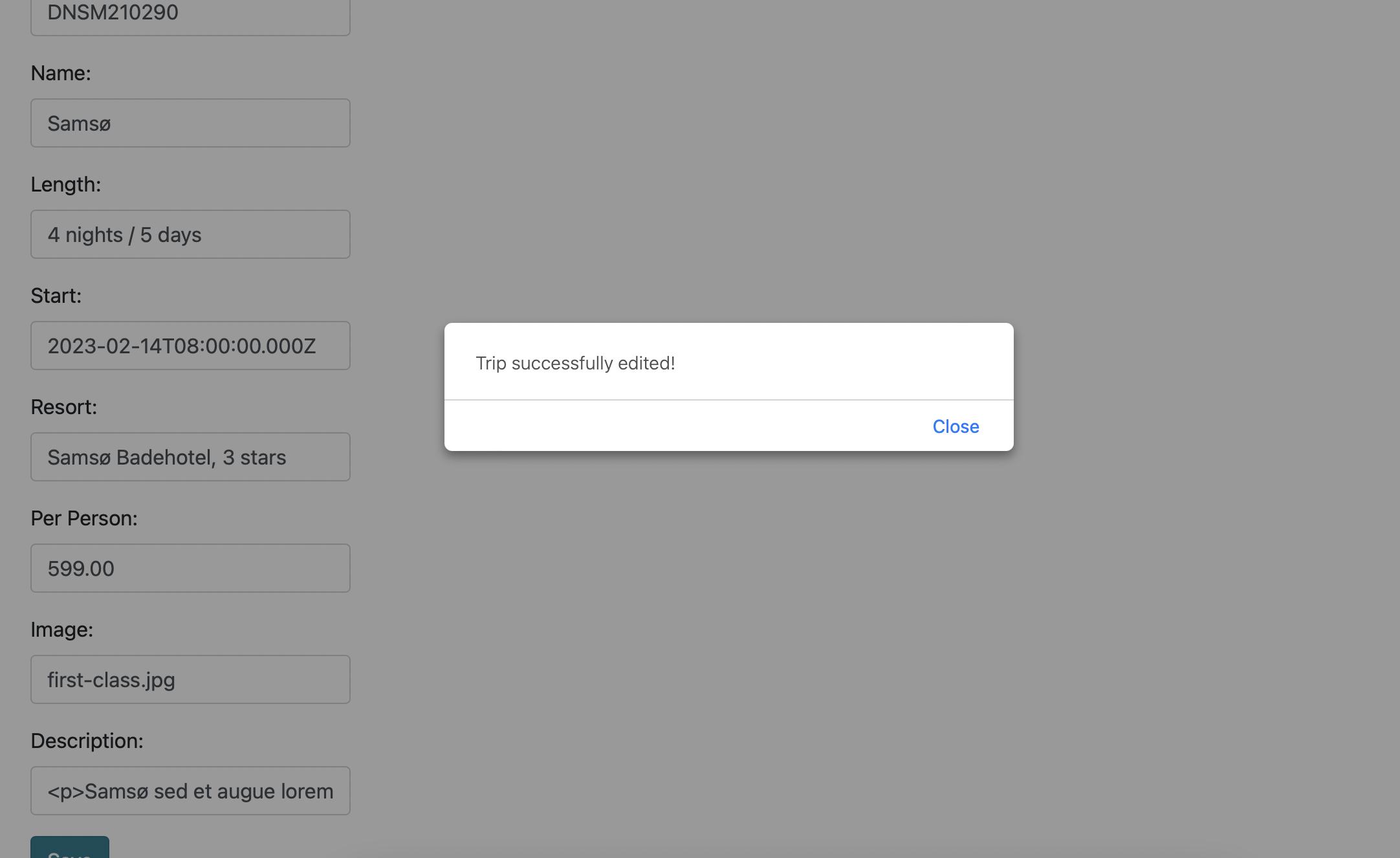
Moving onto the information classes, i.e. CruiseInfo, FlightInfo, etc., there is an aggregation between these classes and the TravelerInfo class. TravelerInfo contains attributes for the total price (float), total miles (integer), and stop over (string). Furthermore, the information classes have a generalization relationship with the TripInfo class, containing attributes for the starting date (integer), returning date (integer), origin (string), and destination (string). From the diagram, we can also see that the Itinerary interface has a dependency relationship with all of the information classes. Starting with the CruiseInfo class, we have attributes for the name (string), cabin type (string), and price (float.) For the FlightInfo class, the attributes consist of the name, price, but also seat class (string). Lastly, the HotelInfo class also contains a name and price attribute, but it also contains the location (string), star (integer), and rooms requested (integer). The final three classes are for booking purposes, i.e. HotelBooking, FlightBooking, and CruiseBooking. Note that all of these booking classes have a dependency between the corresponding information classes. The booking classes all contain getter methods to retrieve the related traveler info, and trip information, such as hotel information. The data retrieved from the booking classes is then shared with the Itinerary interface via association relationships. Finally, we have a dependency relationship between all of the booking classes and the Travel\_Agent class, which provides the companion number value, with a multiplicity of zero or many.

## [API](#bookmark16) Endpoints

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve a list of all trips | /api/trips/ | Returns all stored trips in the database |
| **GET** | Retrieve a single trip | /api/trips/{tripCode} | Returns a single trip instance, identified by the tripCode passed on the request URL, redacting the curly brackets |
| **POST** | Register user account | /api/register/ | Sign up for a user account to be able to add or edit trips within the database |
| **POST** | Login to user account | /api/login/ | Login to a pre-existing user account to modify database contents |

## The User Interface

## ImageImage



One of the major differences that I noticed between Angular project structure versus an Express HTML customer-facing page is the usage of a single-page application to change what the user sees by showing or hiding specific display components, instead of grabbing new pages from the server. In order to achieve this, the project structure heavily relies on handling the navigation between each view by using an Angular router, which interprets a browser URL as an instruction to change the view.

In a single-page application, all of the application’s functions exists within a single HTML page. Since the browser only renders the necessary components to the user, we can expect improved performance and a better user experience. One of the biggest disadvantages of using a SPA is that Angular has large security flaws, due to it’s client-side nature, and so things like input validation must be handled on a server-side basis.

A helpful method for testing whether the SPA is working with the API to GET and PUT data in the database is to utilize the Postman application. First, we can refer to our terminal console log to show the POST sent to the API endpoint, along with the response that is returned to the caller. For testing purposes, our caller would be Postman. If the SPA is working correctly, the console log will indicate a POST message with an HTTP status code of 201, letting us know that the trip was created. If it is not working, we would expect a status code of 400, indicating that there is a bad request or invalid content.

If we refer to Postman and enter in http://localhost:300/api/trips for the POST field, we should again see an HTTP status, below the returned keys and values. Furthermore, the body tab should display the full JSON object data, along with the internal MongoDB unique ID. Assuming that a new trip was not successfully created with our SPA, we would not be able to find our new trip within Postman, nor would it be persisted within MongoDB.