

Travlr Getaways

# **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 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 | 6/1/2025 | Brad Peterson | Add Executive Summary, Design Constraints, and Component Diagram descriptions. |
| 2.0 | 6/14/2025 | Brad Peterson | Added Sequence Diagram, Class Diagram, and Endpoint documentation |
| 3.0 | 6/29/2025 | Brad Peterson | Added UI demo and project architecture descriptions. |

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

*Travlr Getaways* will consist of an easily navigable front end for users, a single-page application for admin users to manage content without needing to directly interface with the database and will be constructed with modern technologies on the MEAN stack. The database will use MongoDB (M in MEAN), the controller will use the Express framework and Node.js, and the front end will utilize the Angular framework. These technologies work well to create a full-stack solution that is quick to build and manage the data flow from one component to the next and allow for the product to be used on a variety of platforms.

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

Database size: MongoDB is excellent for rapid prototyping and limited amounts of data. But when a product scales into a broad application it will be slower than other database choices. However, *Travlr Getaways* is fairly limited in scope so I don’t foresee this being a major concern, but it is something to keep in mind as development continues.

The SPA for database management means more rigid control is given to administrators than if they were to directly modify the database as they will still communicate in a predetermined way through an API.

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

### Component Diagram



These three components (Client, Server, and Database) will all communicate with one another through our RESTful API. This API is a structure used to allow each component to know how to request data from another component as well and know what form that data will be communicated in so that it can be accessed and used appropriately. The Client component will request information from the Database to populate data regarding the Traveler Portfolio subcomponent. Similarly, it will request authentication when users log in to the Server which will control access to the actual database. While the Server and the Database components are similar, they are separate components that will integrate via the Mongoose tool. The Database will use MongoDB and primarily store JSON data and images that can be used by either the Client or the Server depending on which component requests that information. The Server will hold information in the form of XML file and JSON data that is less intended to be directly used by the frontend for display purposes and is more architectural in nature.

### Sequence Diagram

A diagram of a diagram

AI-generated content may be incorrect.First, the page address is put in by the user as a Route. This triggers the browser to send a request to our Controller as described in our program. From here, the selected Controller puts out a call to retrieve data from our server. From here, that request is funneled by the server’s Routes to the right Model on the server. The Model will then get the information from the actual MongoDB backend. At this point, the data flows back through each step until it is displayed by the browser on the HTML page that originated the request.

## Class Diagram

A diagram of a travel application

AI-generated content may be incorrect.

**Explanation of class diagram:**

**CruiseInfo/ HotelInfo /FlightInfo**

These classes all contain relevant information about each respective method of transit taken on a trip. They are independent of one another as not all trips will use all of them. Each of them inherits basic information from the parent class TripInfo.

**TripInfo**

This class is the basis of information for a trip, containing general information. It is the parent class of the Info classes that will then get more specific about a certain kind of trip that may not be relevant to other trip types.

**Itinerary**

This class is the aggregate of CruiseInfo/TravellerInfo/FlightInfo. It is reliant on at least one of these classes existing as it is the cumulative total of all three methods of transit.

**TravellerInfo**

Subclass of MemberAccount so it contains all that information as well as the number of people along with that person on the trip.

**Membership\_Admin**

This class contains information pertinent to MemberAccounts and is an aggregate of MemberAccounts. If a travel agent is used, they can also use this class’s functions however Membership\_Admin is independent of an agent being used or not.

**HotelBooking/CruiseBooking/FlightBooking**

These classes can implement their associated info classes. (i.e. CruiseBooking can implement CruiseInfo) an are how an agent or user would be able to set up the bookings.

**Travel\_Agent**

This class is how a 3rd party would book a trip for a member. It is capable of implementing any of the Info classes. It can use the booking classes to retrieve information it may need. Its association with those is 0 to many as the reality is that some trips may not have an agent used and agents may have many different trips they are working on. It also shares a similar relationship to Membership\_Admin.

**MemberAccount**

This class contains a traveler’s account information that is specific to them. It is the parent of TravellerInfo.

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

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve all possible trips | /api/trips | Returns all trips in database |
| **GET** | Retrieve a specific trip | /api/trips/:tripCode | Returns only the trip information that was queried by the tripCode attribute |
| **POST** | Adds a new trip to the database | /api/trips | Fields required are:  code, name, length, start, resort, perPerson, image, description |
| **PUT** | Edits an existing trips data | /api/trips:tripCode | Modifies the data queried by tripCode attribute.  Fields required are:  code, name, length, start, resort, perPerson, image, description |

## The User Interface

The following screenshot is the baseline of the app. The cards included are from our initial seed data.

A screenshot of a screenshot of a coral reef

AI-generated content may be incorrect.

The user is able to add new vacations with the “Add Trip” button which takes them to the following form screen:

A screenshot of a computer

AI-generated content may be incorrect.

The results add a new card, A screenshot of a computer

AI-generated content may be incorrect.

If the user needs to make a change they can do so by clicking the “Edit Trip” button on the card itself which brings up a form that is quite similar to the Add form. A screenshot of a computer

AI-generated content may be incorrect.

Before: A screenshot of a website

AI-generated content may be incorrect. After: A screenshot of a website

AI-generated content may be incorrect.

Or as the change appears in Postman.

Before: A screenshot of a computer

AI-generated content may be incorrect.

After: A screenshot of a computer

AI-generated content may be incorrect.

**Project Architecture Comparison**

This project consisted of two different front ends. The first, was a static page built with Express and followed a standard MVC architecture. This was the consumer facing portion of the project and didn’t allow much interaction with the data itself. The M, model, took care of all the business logic for the webpage. The V, view, handled the rendering of the webpage itself and then sends that to the client. And the C, controller, is the portion that would handle requests from the server and hand it off to the corresponding view. This means that the server is handling the rendering and then it gets handed off to the client-side.

The second, was our admin site that was build with Angular. This site used a component based SPA architecture to load in data. These components allow for a large amount of flexibility as they can often be reused and localize the amount of data needing to be loaded to that one component instead of the whole webpage. It also had more functionality built into it with the addition of features to add and edit vacation data on the database. The backend was shared between the two though.

One of the main differences between these is that Express is a server-side application which means it takes more effort for our servers to load all the data every time a page is loaded. The Angular SPA based webpage only needed to reload data that had changed to server new or updated components on a web page.