

Travlr Website

# **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 |
| --- | --- | --- | --- |
| 3.0 | <2025/08/16> | Tanner Hunt | Added security features and admin |
| 2.0 | <2025/07/30> | Tanner Hunt | Added class and sequence diagrams. Added API endpoints section |
| 1.0 | <2025/07/16> | Tanner Hunt | Filled out the executive summary, design constraints, and system architecture view: component diagram |

## 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)

This is a web-based application for Travlr Getaways. Through this site, customers are able to create an account, search for travel packages by location and price point, book reservations with the travel agency, and visit the website regularly before their trip to see their itineraries. The website will also support an admin-only single page application where administrative staff can maintain their customer base, available trip packages, and the prices for items and packages.

These operations will be supported through a MEAN stack and RESTful API. MongoDB, a noSQL database, will store information about the customers and travel packages. Express supports connecting the database to web page elements. Angular supports building dynamic, single page web applications. Node.js manages the server.

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

1. Dynamically generated web pages may not receive search engine optimization, resulting in fewer views of the web page. This can be mitigated by creating static content for the website.
2. Single page applications have a slower initial loading speed, which can lead to increased bounce rates – the rate at which users quickly leave a website
3. A team of IT professionals is required to maintain the user base, database, and manage security
4. Dynamically generated content may not be accessible to certain embedded devices, leading to potentially fewer users
5. Users cannot access content without an internet connection
6. The service is susceptible to downtime from power outages, internet outages, natural disasters, etc – leading to lost revenue.
7. The current design does not include a load balancer, which may lead to instability in times of high usage
8. The current design does not include any caching services, which would increase the websites loading speeds and possibly reduce user bounce rate

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

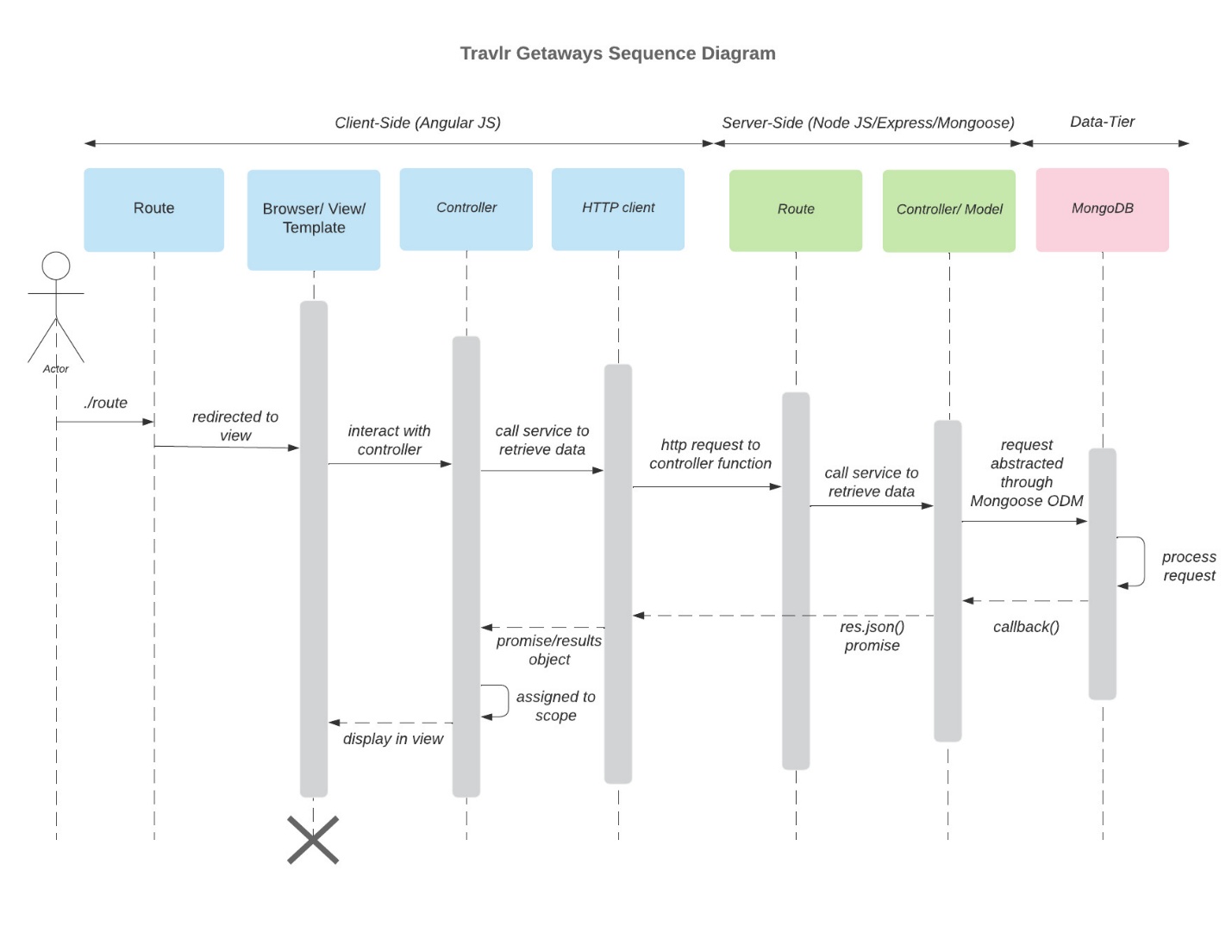
### Component Diagram



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 architecture of the system can be divided into three components: the client side, the server side, and the database. This is also known as the Model-View-Controller (MVC) architecture. The client side contains any element Travlr customers can interact with in their web browser, including their login session, HTML elements (like images, buttons, text, etc.), and their personal portfolio. The server side contains the rules that govern the website. This includes security rules, like validating users, and managing what data can be accessed from or pushed to the database. The database contains the website’s data itself, including descriptions of the travel packages, user data, and images. The client side can request updates from the server, where the server can request the necessary information from the database, then return the data to the client to update the user interface. The benefit of a system architecture like this is that its modularity increases the speed and ease of development and makes adding new features easier in the future. Another benefit of this architecture is how familiar most developers are with it, reducing the time it takes to onboard new engineers to work on the project.

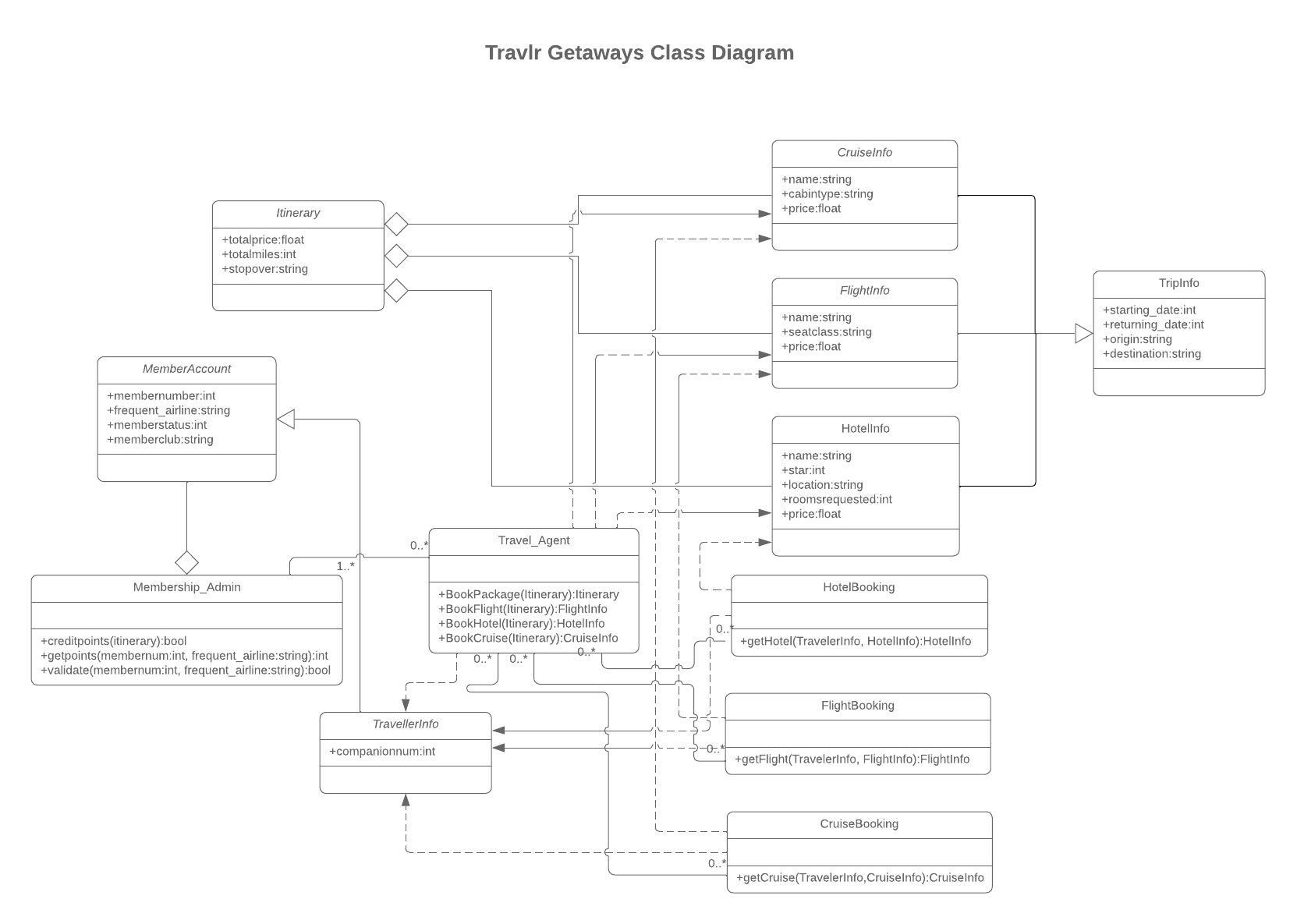
### Sequence Diagram



This diagram describes the flow of data when a user interacts with the website. Broadly, there are three major components data passes through: the client side, the server side, and the database. the website interface (client-side) sends a request for data to the server, the server parses the request and asks the database for the data, the database gives back the data to the server for processing, then the server sends the data back to the client to be displayed. The process is described in greater detail below.

A client looking to find a trip would first find the website on their internet browser, which directs them to the client-side part of the application. The browser then requests the information it needs to render the webpage to the controller. The controller decides what data is necessary to display this, which is a list of all the available trip packages. It then asks the HTTP client – which manages data requests over the internet -- to send a request to your servers for this information. The request is sent to the server’s route, which decides which service should fulfill the request (the controller/ model). In this case, there is a controller that manages requests for trip information. The controller then sends a request for that data to the database. After the database returns the information, the controller/ Model checks the data and formats it correctly. The model then returns the information directly back to the client side through the HTTP client. The HTTP client gives the data back to the client-side controller, which originally requested this information, so that the controller can update the information displayed to the user. In this case, a list of trips is then shown to the user.

## Class Diagram

****

This diagram describes the individual program files that model how scheduling trips works.

Broadly, there are three systems working together in this diagram. There are the trips and trip packages clients can take, there are clients who can schedule trips, and there are travel agents who can manage both clients and their trips.

There are three kinds of trip items: cruises, flights, and hotel stays. These include information like starting and ending dates, origin, destination, price, trip name, and rating. Multiple cruises, flights, and hotel stays can be combined into an itinerary.

Users have information about their number of companions, membership ID, any frequent flyer information, and the status of their membership stored in the service.

Travel agents have access to the membership admin object, which allows them to credit points to members, as well as check their points and validate their frequent flyer information. They can also book individual flights, hotels, or cruises for members – or entire travel packages.

Individual bookings for members are stored in “booking” objects, which reference the information for that booking. This keeps data about individual members trips separate from the trip information everyone else can see.

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

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve list of trips | /api/trips | Return all trips in the database |
| **GET** | Retrieve single trip | /api/trips/:tripsID | Returns single trip instance, identified by the trip ID passed on the request URL |
| **POST** | Create a new trip | /api/trips | Required the trip code, name, length, start date, resort, per person cost, image url, and description |
| **PUT** | Edit a trip | /api/trips/:tripID | Required the trip code, name, length, start date, resort, per person cost, image url, and description |
| **POST** | Create a new account | /api/register | Require account name, email, and password. Generates an account salt and password hash |
| **GET** | Find and return user data | /api/login | Returns a users name and email |

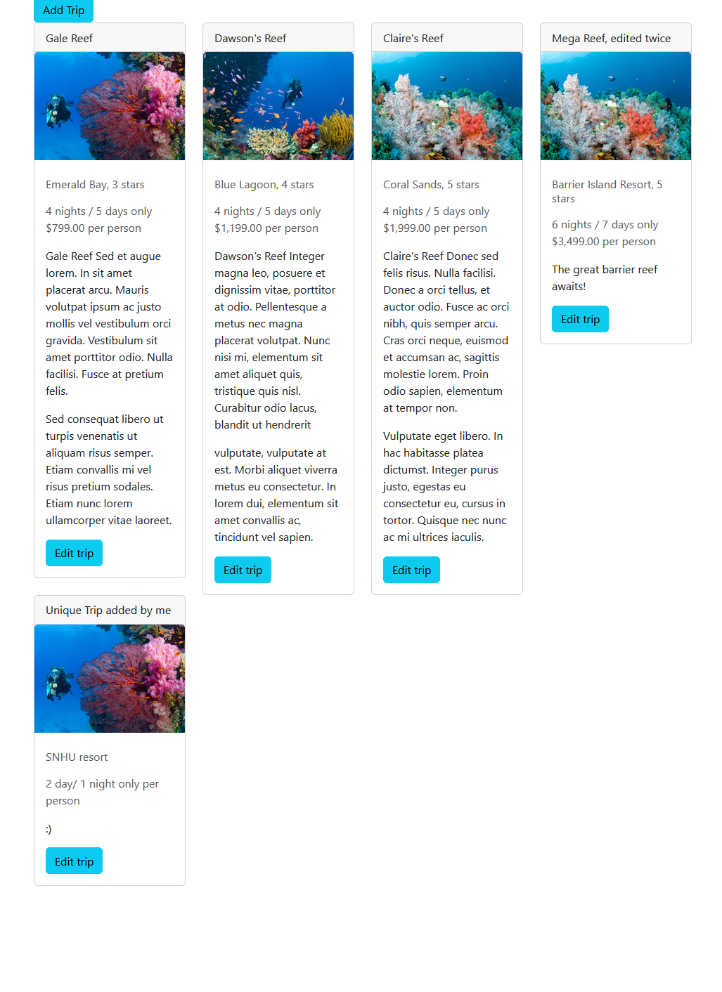
## The User Interface

**Trip Creation:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Trip added to main screen:**

****

**Trip editing:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Trip edit pushed to main screen: A screenshot of a cell phone

AI-generated content may be incorrect.**

The angular project structure is based around reusable components and services. These components manage their own behaviors, dependencies, and repopulate the webpage with content when the screen is loaded. The webpage could then route requests to different components rather than a server. This makes the single page application faster than the Express application, which needs HTML to be served to it for every web page. Testing the GET and PUT API with the SPA was two-step: server-side functionality was tested by sending HTTP responses using Postman, and the interface was tested once it was properly built for each function.