

Full Stack Web Application for Travlr Getaways

# **CS 465 Project Software Design Document**

Version 1.0

## Table of Contents

[**CS 465 Project Software Design Document** 1](#_rfjqe1xx9qrh)

[Table of Contents 2](#_3neftd6y7mi)

[Document Revision History 2](#_x4g7ec83zm4r)

[Instructions 2](#_u01sxxw4tku4)

[Executive Summary 3](#_w962l6c1sboo)

[Design Constraints 3](#_zhf5o2ywmc7k)

[System Architecture View 3](#_lwiibc6smzql)

[Component Diagram 3](#_72pijohhaq3q)

[Sequence Diagram 4](#_r9hvtwtsqsl1)

[Class Diagram 4](#_h8hiaoy3v1pl)

[API Endpoints 4](#_ttme2d1snme4)

[The User Interface 4](#_av1u5wtbodbc)

## [Document Revision History](#_b9bxhdkv845s)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 03/23/25 | Christine Petrush | -Create executive summary  -Identify and explain the design constraints  -Describe the overall system architecture and identify the significant components. |
| 1.1 | 04/05/25 | Christine Petrush | -Add Sequence diagram and description  -Add Class diagram and description  -Add API Endpoints |
| 1.2 | 04/21/25 | Christine Petrush | -Edit the Sequence and Class diagrams  -Add The User Interface |

## 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](#_tmq5m6i0rh69)

Our client, Travlr Getaways, is looking to create a fully functional travel web application that meets their requirements for both a customer-facing side of the web application and an administrator single-page application. The full stack web application for Travlr Getaways will be built using MEAN stack architecture: MongoDB, Express, Angular, and Node.js. MongoDB is the database, Express is the web framework, Angular is the front-end framework and Node.js is the web server.

Customers must be able to create an account, search for travel packages by location and price point, and book reservations with the travel agency. In addition, they must also be able to visit the website regularly before their trip to see their itineraries. To accomplish this, the customer-facing side of the application will use an Express framework with routes, controllers, views, and data models and will align with the wireframe provided to meet Travlr Getaways’ vision. HTML, CSS, and JavaScript will be used in the construction of the static customer-facing website.

Travlr Getaways also requires an admin-only site where administrators can maintain a customer base, available trip packages, and pricing for each item and package. The administrator single-page application (SPA) will use the Angular command line interface (CLI) to build components and services. Once completed, the application will be tested with the RESTful API to make certain the server returns the data properly.

## [Design Constraints](#_8yz1gt88pyht)

Design constraints are limitations or restrictions in the design process. Some common types of design constraints include technical constraints and project-specific constraints.

Technical Constraints:

While the MEAN stack architecture has numerous benefits, there are some constraints related to scalability and performance that should be considered. While MongoDB is a powerful NoSQL database, it can become a bottleneck for highly complex applications. The performance of Node.js and Express can degrade under heavy workloads or with poorly optimized code. And Angular can have a steep learning curve and can lead to performance issues if not optimized correctly. Careful optimization of code, database queries, and front-end rendering is necessary to ensure smooth application performance under high traffic.

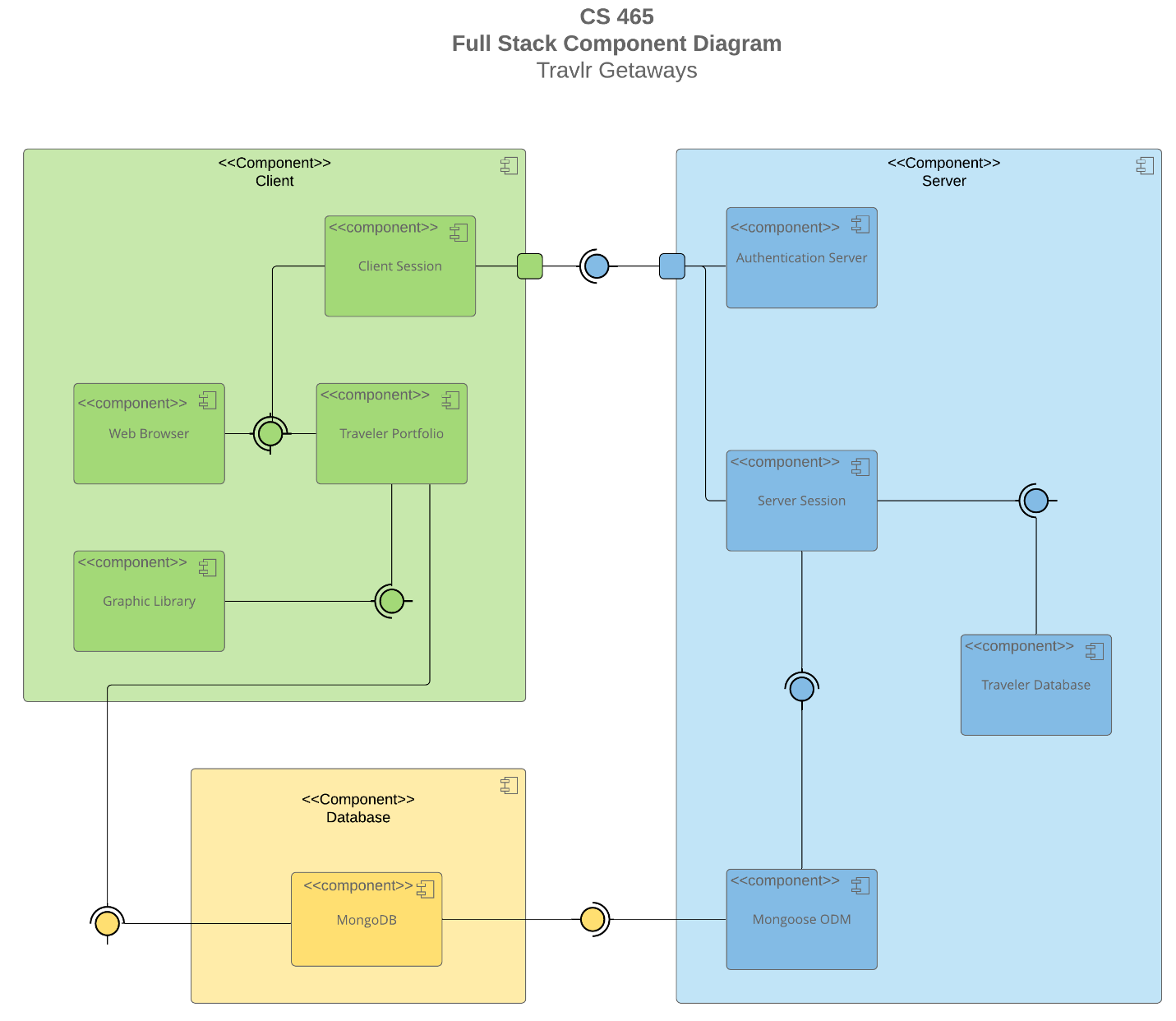
Project-specific constraints:

Travlr Getaways is requesting a fully functional travel web application with a customer-facing website and an admin-only site using specs with a wireframe provided by their marketing department as a guide. Some project-specific constraints regarding the customer-facing side of the web application are that customers must be able to create an account, search for travel packages by location and price point, book reservations with Travlr Getaways’ travel agency, and be able to visit the website regularly before their trip to see their itineraries. Project-specific constraints regarding the admin-only site are to be able to maintain a customer base, available trip packages, and pricing for each item and package. In addition, budget and timelines are project specific constraints. Financial constraints can impact many areas of the design process, including human resources, tools, user research, project scope, and technology. While meeting timeline constraints is crucial for project success ensuring timely delivery and preventing delays.

## 

## [System Architecture View](#_cpvecrxx6q3p)

### 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).

A component diagram visually represents how components of a software system relate to one another. They should communicate the scope of the system, the overall structure of the system, and goals that the system helps achieve. Components are model elements that represent independent, interchangeable parts of a system. The system architecture of the web application consists of three main components: Client, Server, and Database.

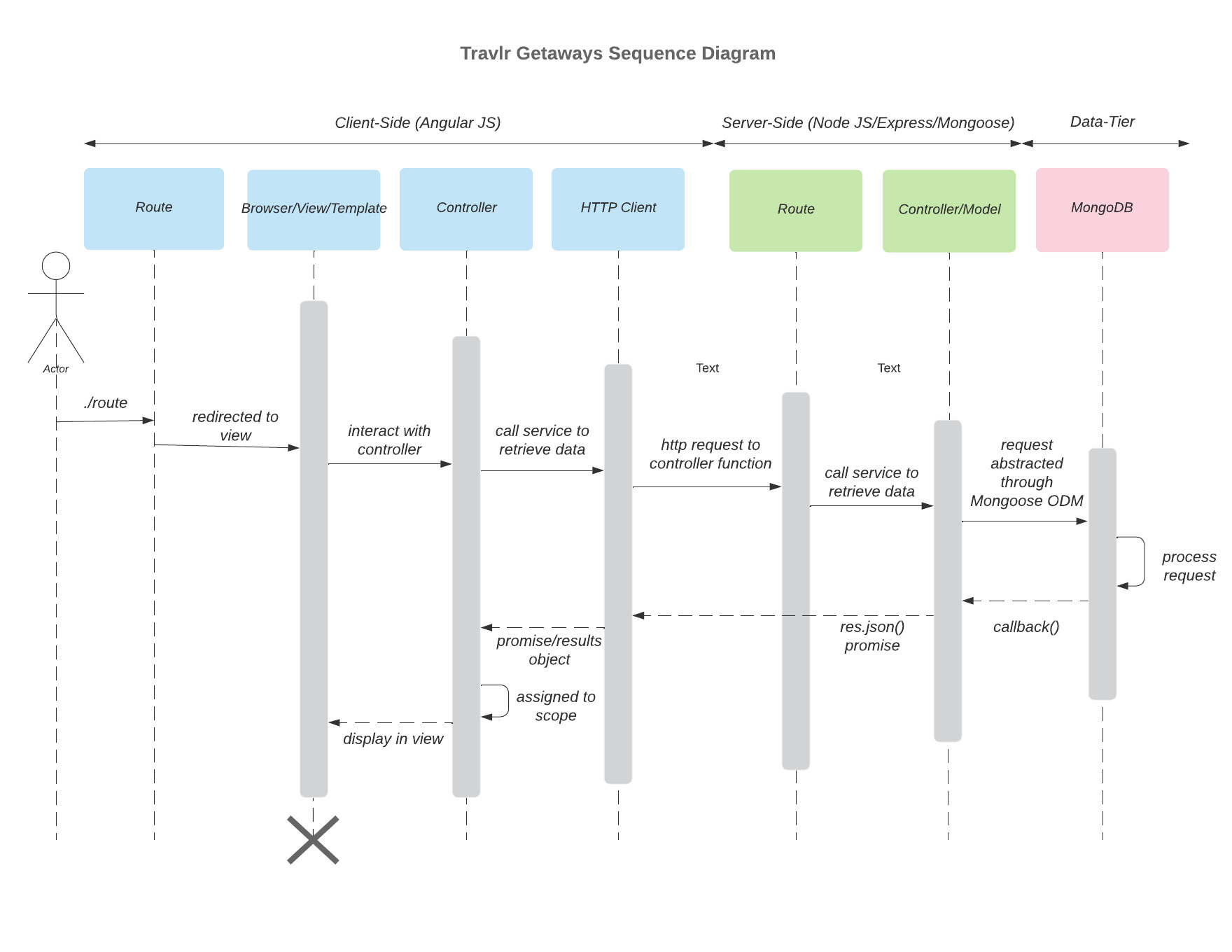
Each main component has sub components nested within. The Client component has four sub components: Client Session, Web Browser, Traveler Portfolio, and Graphic Library. The Server component also has four sub components: Authentication Server, Server Session, Traveler Database, and Mongoose ODM. The Database component has one subcomponent being MongoDB.

The open half circle represents a required interface which is something needed by a component for it to function. For example, in the Client component, the Client Session requires the Traveler Portfolio and the web browser requires the traveler portfolio. In the Server component, the server session requires the Mongoose ODM.

The closed circle represents a provided interface that offers a service to another component. For example, in the Client component, the traveler portfolio provides a service to both the client session and web browser while the graphic library provides a service to the traveler portfolio. In the Server component, the Mongoose ODM provides a service to the server session. The MongoDB located in the Database component provides a service to both the Mongoose ODM and Traveler portfolio. The MongoDB could have many recipients or none. It neither knows or cares; it just provides an interface.

An assembly connection represents the mating of a required and a provided interface. In the Client component, this is shown where the client session and the web browser meet the traveler portfolio. Also where the traveler portfolio meets the graphic library and MongoDB. The square symbol represents a port which is an explicit and often formal interface through which all communication travels.

### Sequence Diagram

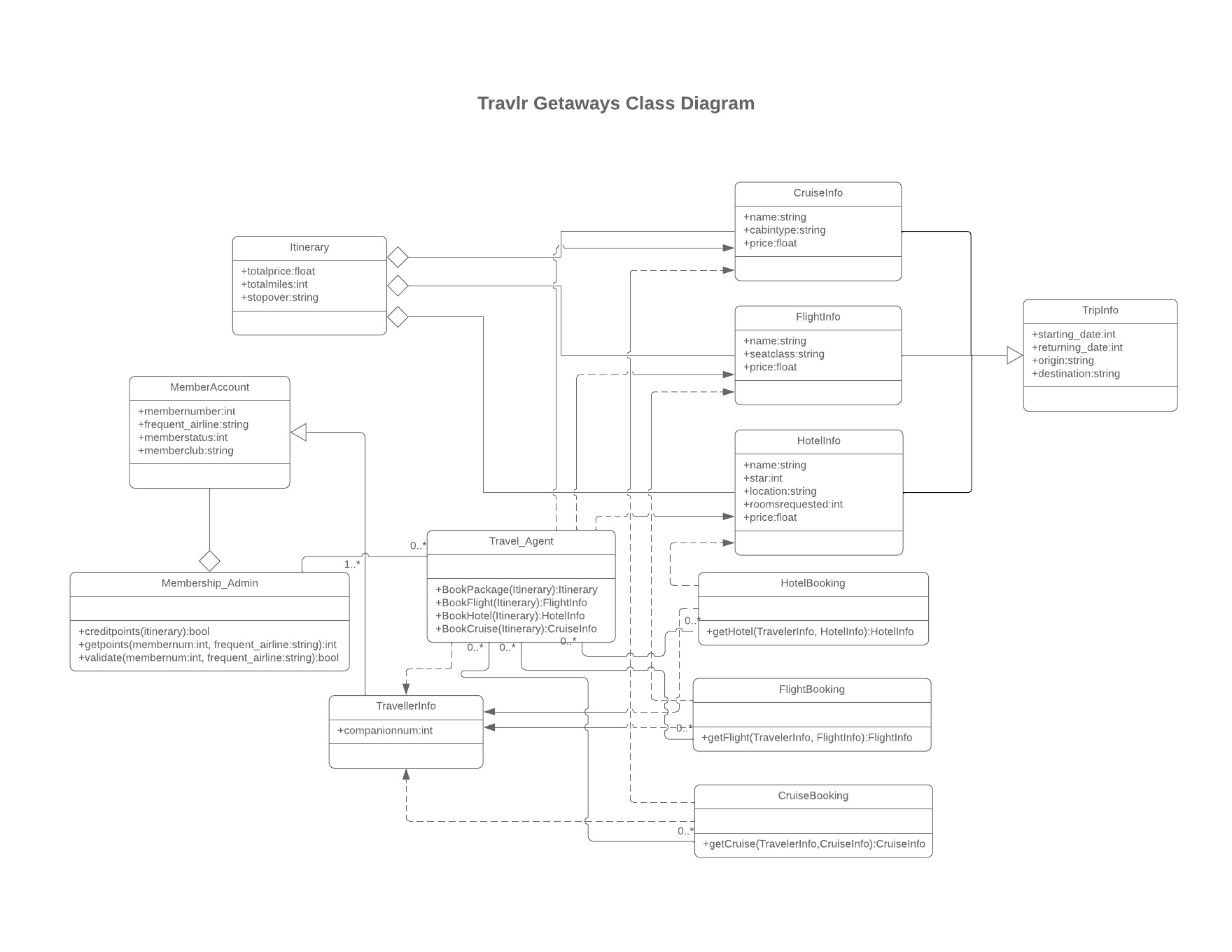


The sequence diagram is an interaction diagram that details how operations are carried out. The actor is a type of external role that is played by an entity that interacts with the subject. In this case the actor is the user. The boxes at the top of the diagram are object symbols which represent a class or object demonstrating how an object will behave in the system. The route takes the user to the intended website. Once the user accesses the website, the browser which displays the web content, data is presented to the user by way of view/template. The controller is responsible for the way that a user interacts with the application. The final step in the client side of the sequence is the HTTP Client which is when the client sends a request to the server. The server-side of the sequence diagram receives the request through the controller. After receiving input from the controller, the model is responsible for managing the data of an application. The request to retrieve data is then routed to the database and the database processes the request. The long rectangular boxes shown under the lifeline symbols are activation boxes. These represent the time needed for an object to complete a task.

## 

## 

## Class Diagram

  
A class diagram is used to map out what the system will look like in static form. It shows what the classes need, their functionality and their relationships with other system elements. The class name appears at the top of the box. The attributes are in the second section. Attributes are the values that define a class. The attribute type is shown after the colon. For example, in the Itinerary class, totalprice is an attribute with float as the attribute type. In the third section are the operations. Operations are the services that the class provides. The return type of the method is shown after the colon at the end of the method signature. For example, in the Travel\_Agent class, BookPackage is an operation with Itinerary as the return type. The symbols that are shown before an attribute and operation name denotes the visibility. For instance, a plus sign means public and a minus sign means private. Other symbols that could appear are # for protected, ~ for package and / for derived.

Relationships in a class diagram are shown by the arrows and lines drawn between the classes. A solid line represents a bidirectional association. This means that the two classes are aware of their relationship with each other as shown between the Membership\_Admin and the Travel\_Agent. An open arrow with a solid line represents inheritance. This indicates a child-parent relationship between two classes. The child class is a specialized, subclass of the parent. This is shown between the Travel\_Agent and the TravellerInfo class with the TravellerInfo being the child of the MemberAccount. A dotted line with a closed arrow represents dependency. This is when one class depends on another as shown with the TravellerInfo and Travel\_Agent classes. The traveller info depends on the travel agent. The open diamond with a solid line represents aggregation. This is a unilateral relationship between classes. One class is part of, or subordinate to another. In an aggregation relationship one class can exist independently from the other. This is shown between the Itinerary class and the CruiseInfo, FlightInfo, and HotelInfo classes. Finally, the small numbers or notations near the ends of the relationship lines represent multiplicity, indicating how many instances of one class can be related to instances of another. “0..” represents zero or more instances (flexible relationship) and “1..” represents one or more instances (mandatory relationship).

## [API](#_1rayohu990sb) Endpoints

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieves a list of all trips | /api/trips | Returns all trips in the Database |
| **GET** | Retrieve a single trip | /api/trips/:tripId | Returns a single trip identified by the trip ID passed on the request URL |
| **POST** | Creates a new trip | /api/trips | Creates a new trip and adds it to the database |
| **PUT** | Updates a single trip | /api/trips/:tripId | Updates a specific trip identified by the trip ID passed on the request URL |
| **DELETE** | Deletes a single trip | /api/trips/:tripId | Deletes a specific trip identified by the trip ID passed on the request URL |
| **POST** | Creates a new user account | /api/register | Creates a new user account based on information entered on form |
| **POST** | Authenticates a user | /api/login | Authenticates a user’s login based on information entered |
| **GET** | Retrieves a single user profile | /api/profile/:profileId | Returns user profile information identified by the profile ID passed on the request URL |
| **GET** | Retrieves a list of all users | /api/admin/users | Returns a list of all users in the Database |
| **PUT** | Updates a single user’s information | /api/admin/users/:userId | Updates a user’s role or other admin related data identified by the user ID passed on the request URL |
| **DELETE** | Deletes a single user | /api/admin/users/:userId | Deletes a specific user identified by the user ID passed on the request URL |

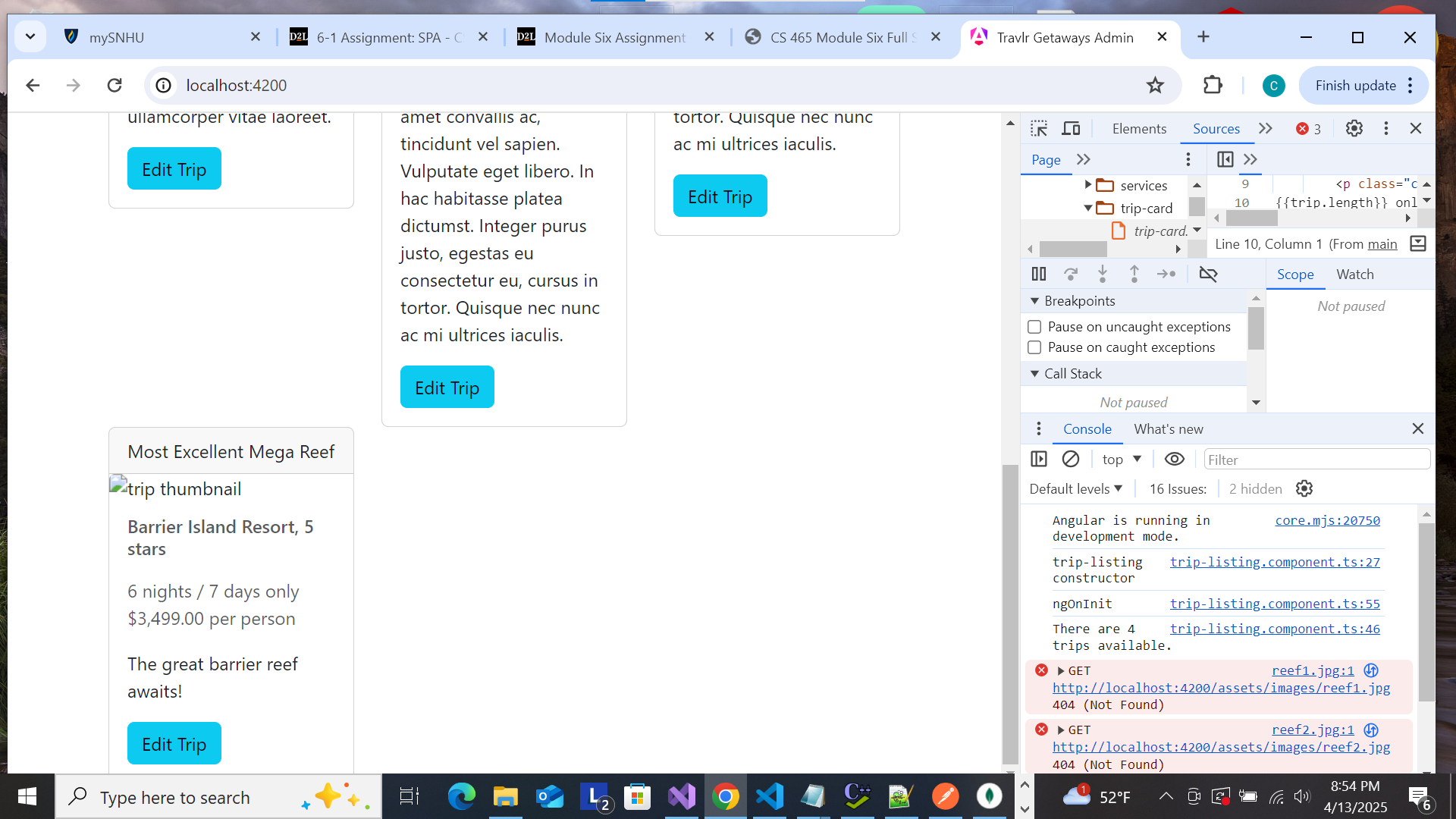
## 

## 

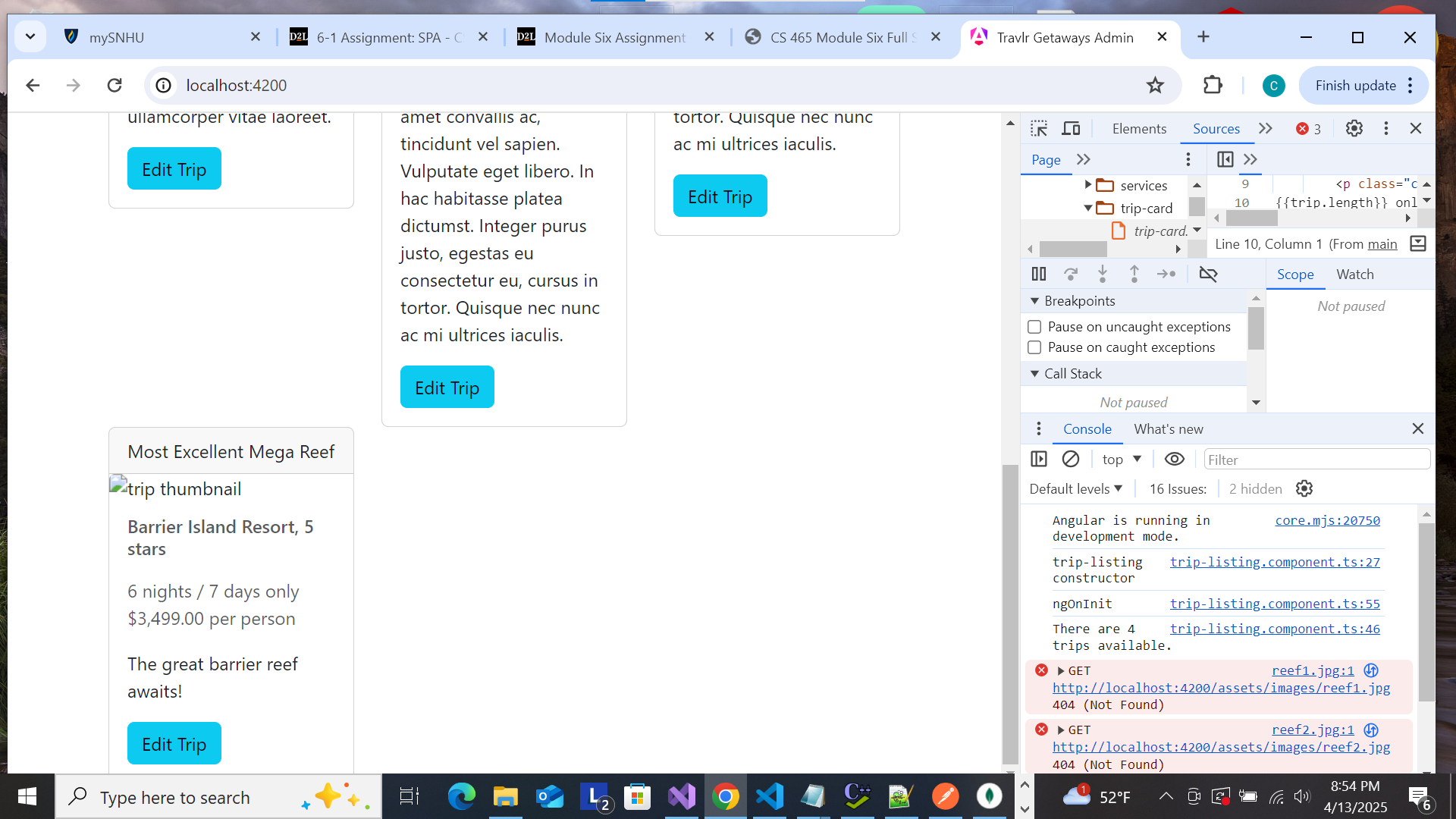
## 

## The User Interface

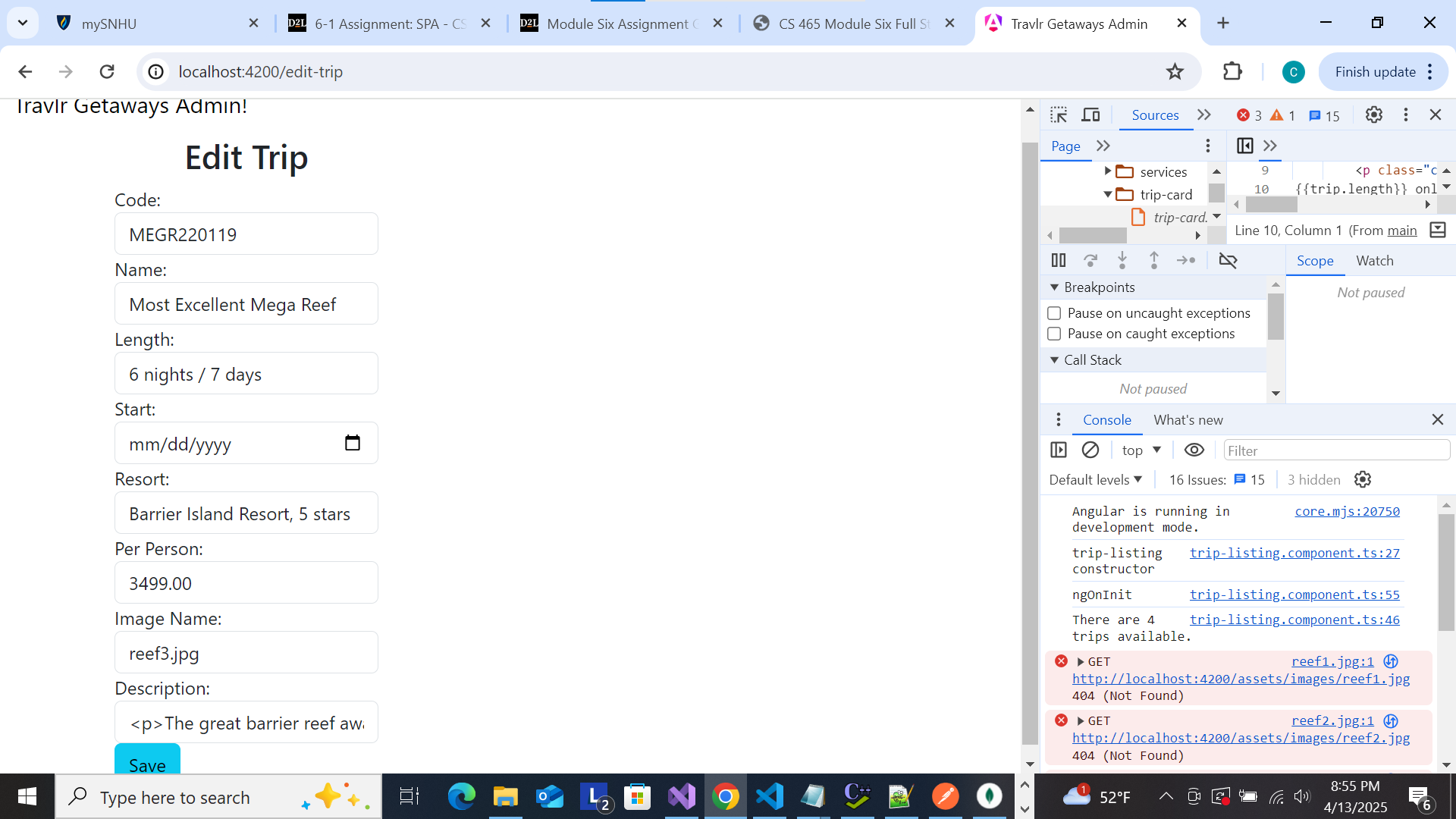
Trip added:



Edit screen:



Update Screen:



<Insert screenshots from the development of the SPA development to show the following: (1) a unique trip, added by you, (2) the Edit screen, and (3) the Update screen.>

The Angular project structure differs from the Express HTML customer-facing page in that Angular handles the front end, building dynamic and interactive user interfaces. Express manages the back end, handling the server-side logic and API requests. Angular follows a modular, component-based approach with specific folders and files to organize code logically focusing on reusable components. Express follows an MVC pattern with separate files for views, controllers, and models emphasizing server-side rendering and flexibility (ElHousieny, 2023 and Manahdhar, 2022).

A single-page application (SPA) functionality has both advantages and disadvantages. Some advantages are speed and responsiveness, enhanced user experience, reduced server load, cross-platform compatibility and offline functionality. Some disadvantages are Search Engine Optimization (SEO) challenges, longer initial loading time, browser history and bookmarks may not work as expected, complex development, and security concerns (Vaishak, 2023). The additional functionality that is provided by a SPA compared to a simple web application interaction is a more interactive and responsive user experience. By leveraging JavaScript and API’s, a SPA is able to prioritize a seamless, interactive, and fast-paced user experience. A SPA dynamically updates content on the page allowing for real-time updates and interactions (Adobe Experience Cloud Team, 2023).

API testing is the process of validating the functionality, reliability, performance, and security of an API by sending requests to the API and verifying that the responses match the expected outcome. Ensuring functional correctness, improving reliability, facilitating integration, and enhancing security are several reasons why API testing is important. APIs often interact with databases to store, retrieve, and manipulate data and testing is crucial for ensuring data integrity and consistency. Database integration in API testing includes: verifying database interactions, setting up test data, and validating database state. Best practices for API testing and database integration should be followed to ensure effectiveness and efficiency. One best practice is to maintain test data integrity by implementing mechanisms so that test data remains consistent and isolated to prevent interference with production data. Another best practice is to use test environments to avoid impacting production systems during testing. Implementing data seeding and tear down is another best practice. Automating the process of seeding test data and tearing down ensures a clean slate for each test run. Leveraging test automation ensures that changes to the codebase does not introduce regressions or data inconsistencies while embracing test parallelization improves test execution speed and efficiency. The best practice of monitoring and logging test execution captures detailed information about test failures, database interactions, and performance metrics. Finally, implementing appropriate measures to safeguard sensitive data used for testing ensures data privacy and security (Chiu, 2024). An API error occurs when a server fails to locate the requested resource from the API provider. Some common causes of API errors are issues in the endpoint, incorrect parameters, or problems with the API key during the request call. Other API errors include HTTP/HTTPS errors, method mixups, missing content, authorization errors and data formatting errors (Hassan, 2025).

Adobe Experience Cloud Team. (2023). *A guide to traditional vs. single-page apps (SPAs)*. [www.business.adobe.com](http://www.business.adobe.com)

Chiu, Tioka. (2024). *API Testing and Database Integration: A Comprehensive Guide*. [www.medium.com](http://www.medium.com)

ElHousieny, Rany. (2023). *Understanding Angular Project Structure*. [www.linkedin.com](http://www.linkedin.com)

Hassan, Daniyal. (2025). 6 *Frequently Occuring API Errors And How to Prevent Them From Happening*. [www.astera.com](http://www.astera.com)

Manandhar, Geshan. (2022). *Organizing your Express.js project structure for better productivity.* [www.blog.logrocket.com](http://www.blog.logrocket.com)

Vaishak. (2023). *The Pros and Cons of Single-Page Applications (SPAs)*. [www.medium.com](http://www.medium.com)