

John Miller

# **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 | 03/23/25 | John Miller | Executive Summary, Design Constraints, System Architecture View |
| 2.0 | 04/06/25 | John Miller | Sequence Diagram, Class Diagram, API Endpoints |
| 3.0 | 04/20/25 | John Miller | User Interface, API Endpoints updated |

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

The appropriate architecture of the web application is a full-stack JavaScript solution built using the MEAN stack: MongoDB, Express, Angular, and Node.js. This architecture supports modular development, making it simple to create and manage.

The front end will be built using Angular, which allows navigation using routes between Single-Page Applications (SPAs). The administrator page will also be built using route guards in Angular, which provides for back-end CRUD operations. Handlebars is a templating engine that enables static pages to load on the server side. It is currently being used in the project.

The back-end will be built using Node.js and Express. These are used to handle routing and RESTful API creation. Node and Express are also enabled using the Model View Controller (MVC) architecture. This further separates concerns within the code base. The model is the database; the view is the Angular HTML pages, and the controllers are written in JavaScript to handle requests from the clients and request the model.

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

Several design constraints influence the development of the Travlr Getaways application, affecting its technical and functional aspects. These constraints must be considered to ensure a successful, scalable, and user-friendly application.

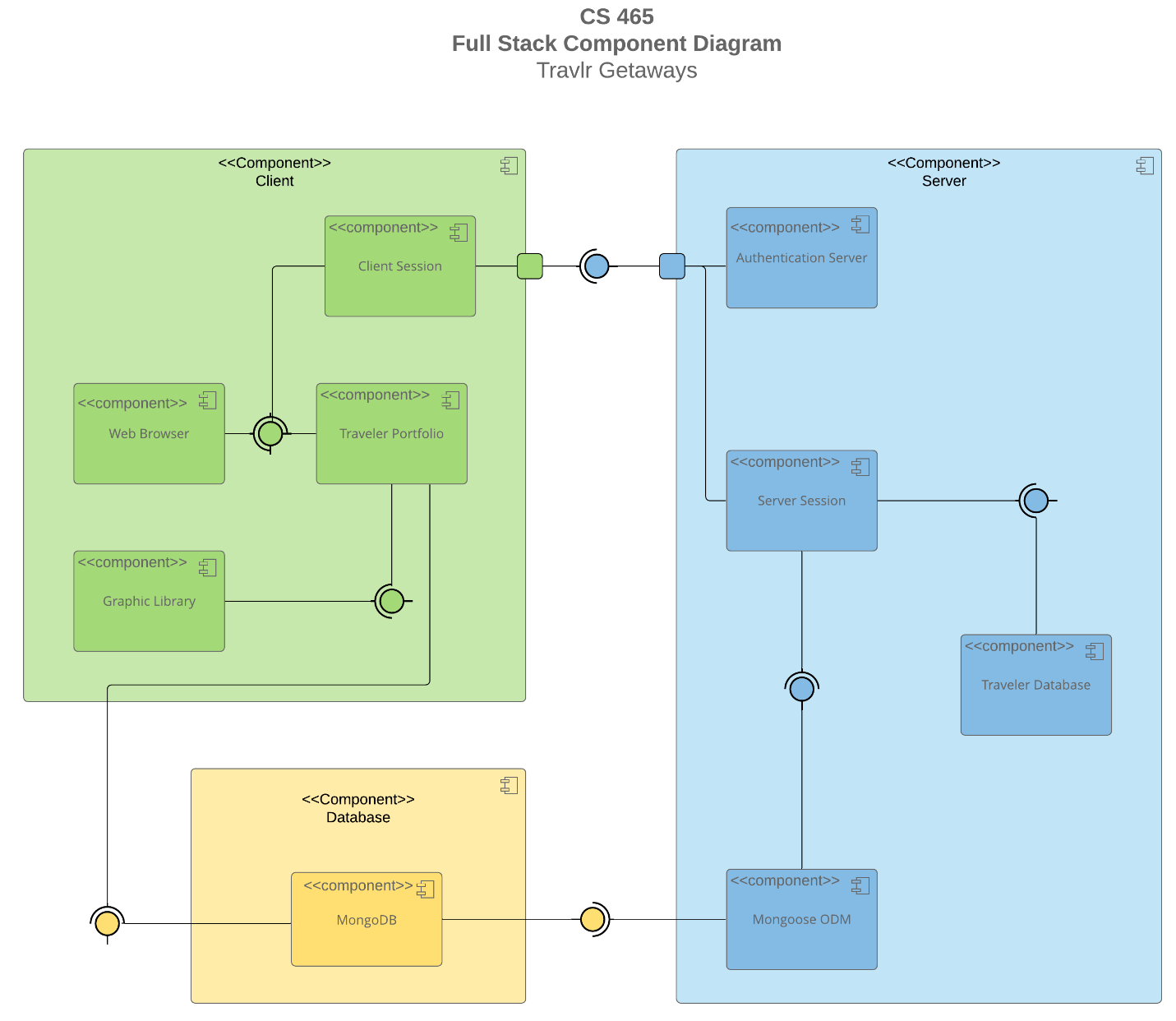
The app must be built using the MEAN stack. This constraint means the back-end logic will be written in JavaScript using Node.js and Express. The front end will be SPA pages built using Angular. The data will be stored in a NoSQL database, MongoDB.’

The app will use the MVC architecture pattern, which implies a separation of concerns and makes the code more modular. MongoDB will handle the model, the views will be Angular, and the controllers will be Node.js and Express.

The app must be scalable and maintainable. The MEAN stack and MVC architecture support modular app development, making the web app more effortless to scale and maintain. The modular design allows one service to be worked on without affecting the others. Node.js is asynchronous, which means it scales exceptionally well.

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

### Component Diagram



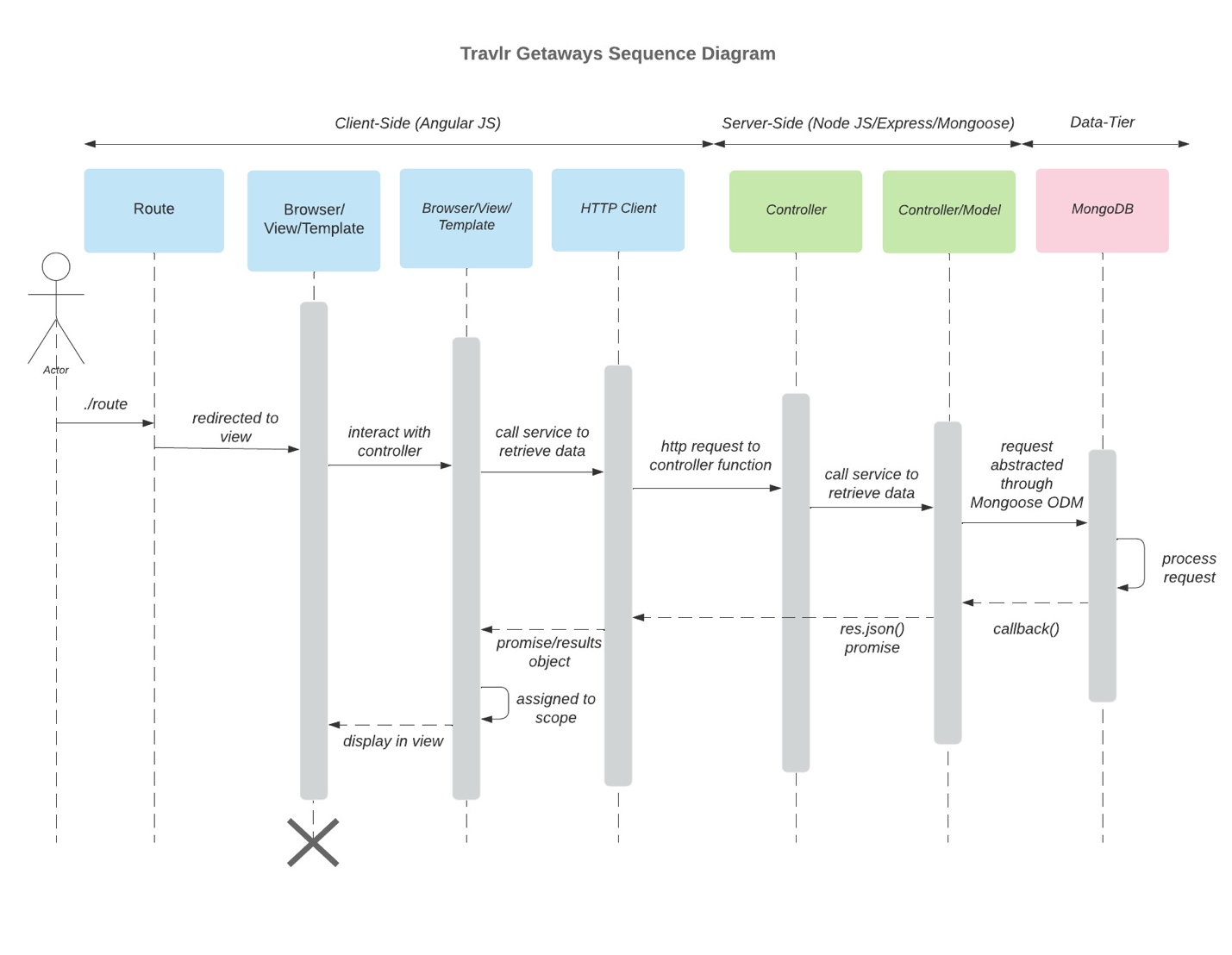
The Travlr Getaways web application follows a modular component-based architecture aligned with the MEAN stack (MongoDB, Express, Angular, and Node.js). It is organized into three primary layers: Client, Server, and Database. These layers interact to support a responsive and secure travel portfolio experience for users and administrators.

The Server layer handles business logic and client-database communication. It comprises the Authentication Server, Server Session, Traveler Database, and Mongoose ODM components. The authentication server validates user credentials. The Server Session maintains the session context. The Traveler Database contains back-end logic for queries. Mongoose ODM allows the conversion of documents to JSON and the setting of schemas for the MongoDB database.

The Client layer represents the user-facing part of the code. Its components include Client Session, Web Browser, Graphic Library, and Traveler Portfolio. Client Session maintains state and session data on the client side. The Web Browser hosts the web application. The Graphic Library serves as the visual renderer that displays interactive elements. The Traveler Portfolio is the central component where users engage with their travel information.

The Database layer uses MongoDB to store all persistent data. MongoDB is a NoSQL database, making it flexible for storing various data with varying structures. This will connect to the Mongoose ODM to assist in validating and formatting data.

### Sequence Diagram



The actor starts the interaction by accessing a specific route in the app, such as “/trips.” Angular JS handles the route. Based on the route, the actor is directed to the appropriate view; the browser loads the Angular Template. The template uses the HTTP client to submit a request to the controller. The controller verifies and validates the request before passing it to the abstraction layer. The controller/model layer acts as an abstraction layer to make and format requests/responses from MongoDB, which will process the request and commit the appropriate CRUD operations.

The server returns a JSON response, fulfilling the promise. AngularJS receives the data, and the results are assigned to the scope, making them available in the template. The view then displays the updated content within the browser.

## 

## Class Diagram

A diagram of a travel getaway

AI-generated content may be incorrect.  
The Travlr Getaways class diagram represents a modular and object-orientated architecture that models the core functions of the booking system. At the heart of the system is the Travel\_Agent class, which acts as a coordinator for booking different travel experiences. This class directly interacts with HotelInfo, CruiseInfo, and FlightInfo to build an Itinerary. The Travel\_Agent uses the Itinerary to employ the respective booking class. TripInfo supports this flow by aggregating relevant data, such as price.

User information is stored in MembershipAccount and TravelerInfo. The former contains the membership number, membership status, and other information, while the latter includes information on companions. The Membersip\_Admin class is responsible for backend validation and reward point handling.

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

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve a list of all available trips. | </api/trips> | Returns a list of all active trips. |
| **GET** | Retrieve single trip. | </api/trips/:tripId> | Returns a single trip based on the tripID passed in the HTML request. |
| **POST** | Adds a new trip. | </api/trips> | Returns a JSON of the added trip. Requires JWT authorization. |
| **PUT** | Edits an existing trip. | <api/trips/:tripID> | Returns a JSON of the edited object. Requires JWT authorization. |
| **POST** | Registers a new user. | <api/register> | Returns a JWT token. |
| **POST** | Logs in an existing user. | <api/login> | Returns a JWT token. |

## The User Interface

**Architecture:**

            The Angular project structure is modular and component-based, while a typical Express HTML is more static and server-rendered. In Angular, the front end is broken into multiple reusable components, each with HTML, CSS, and TypeScript files. A routing module handles navigation between components based on the current URL, which is what makes it a SPA. The router maps URLs to components, updating the view and address bar without triggering a full page reload (Holmes & Harber, 2019). In contrast, an Express HTML customer-facing page serves pre-built HTML files using a templating engine such as Handlebars or Pug (Holmes & Harber, 2019). As opposed to an Angular SPA, each new page load makes a request to the server.

**Functionality:**

Angular SPAs offer many advantages and additional functionality. They deliver a fast and seamless experience, as there are no full-page reloads. This means a speedier transition between views in the app, leading to smoother navigation. The modular component-based structure promotes reusability and scalability (Holmes & Harber, 2019). SPAs offer real-time form validation; for example, in the add-trip component, each field was required and would not submit a POST request to the DB without a valid entry. Angular also has a persistent state across views, allowing different components to share data (Holmes & Harber, 2019).

            There are several disadvantages to an Angular SPA architecture. SPAs initially load a minimal HTML shell and then dynamically render the actual content in the browser. Search engine crawlers can struggle to index content not present in the initial HTML response (Holmes & Harber, 2019). The large bundle being loaded also slows initial load times.

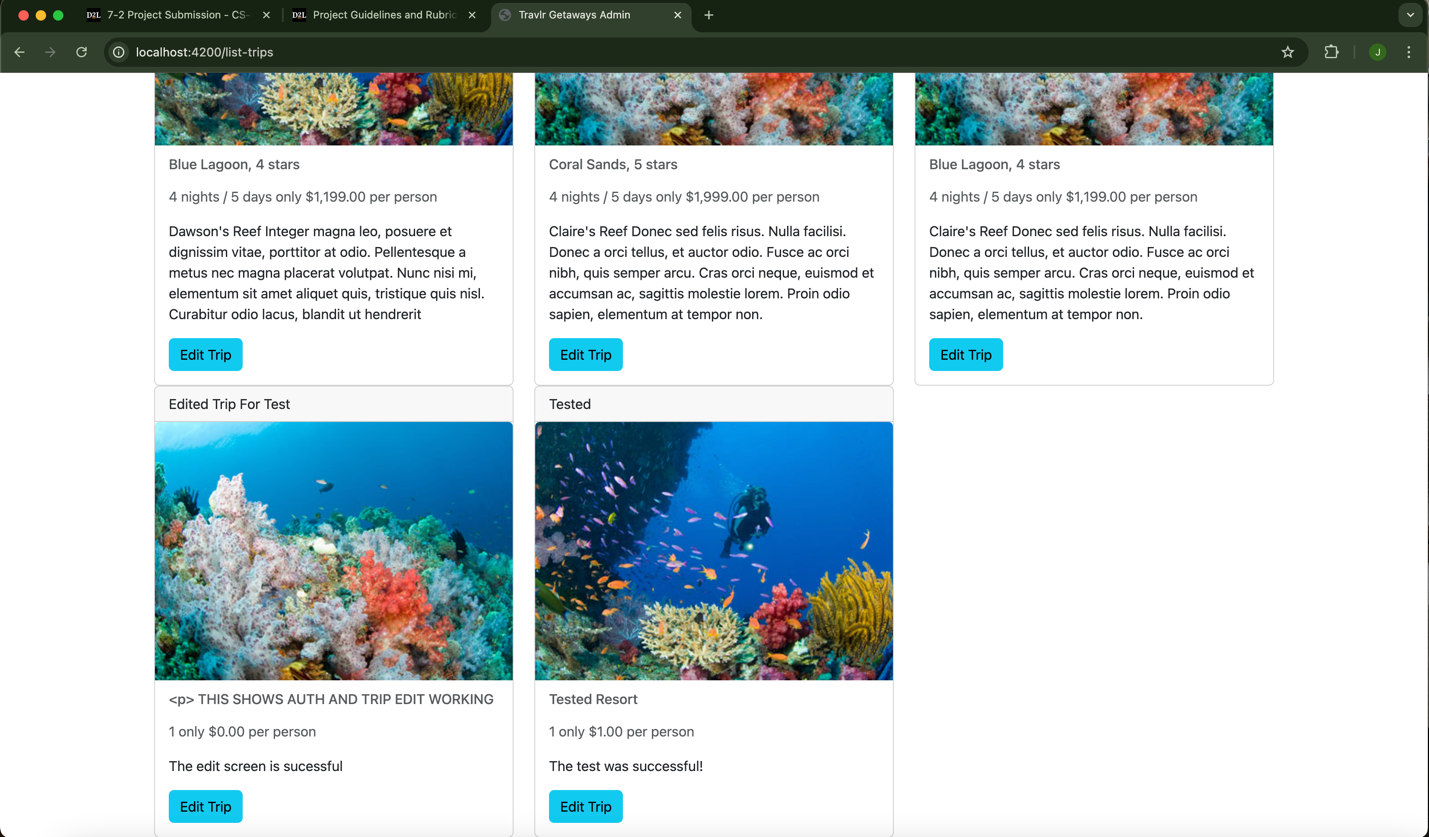
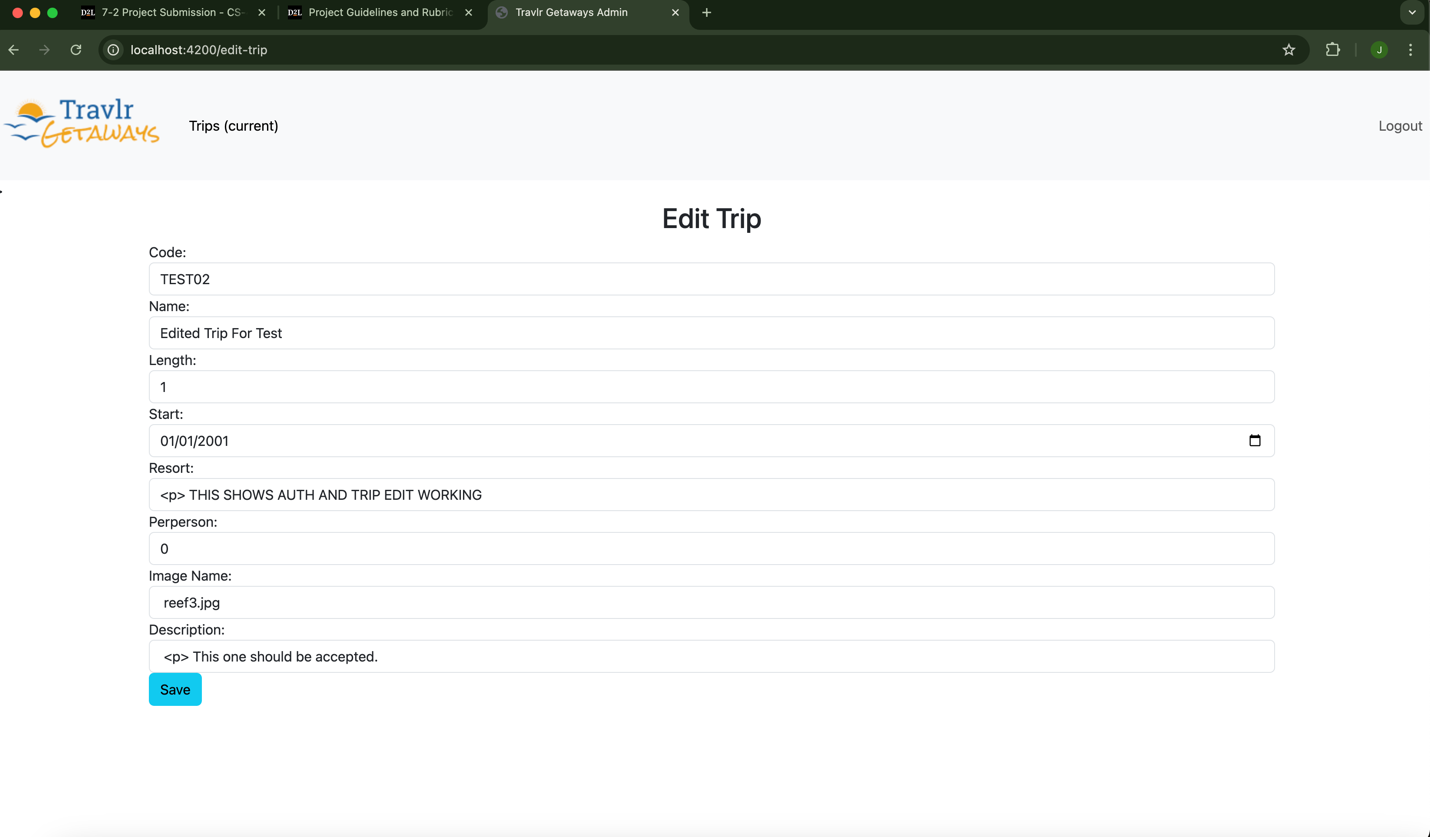
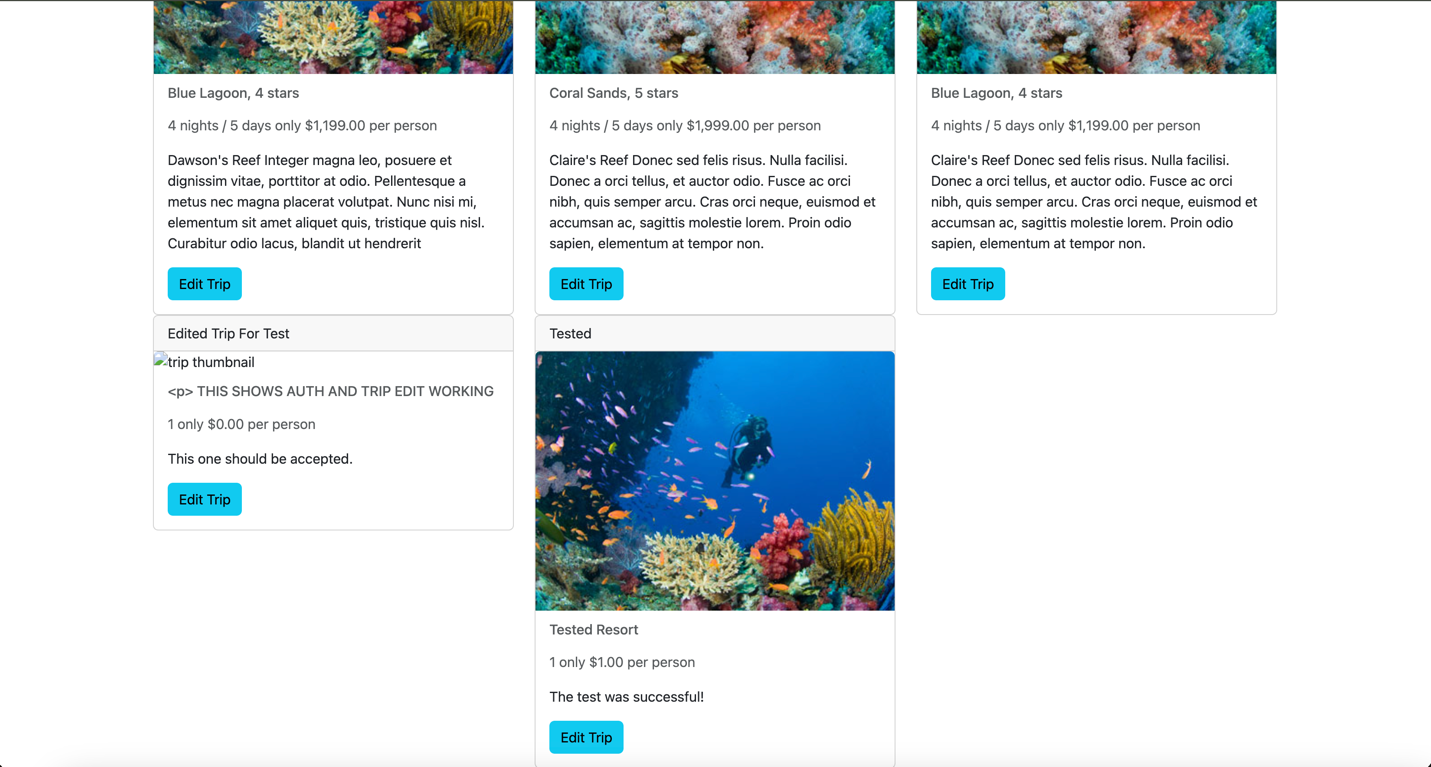
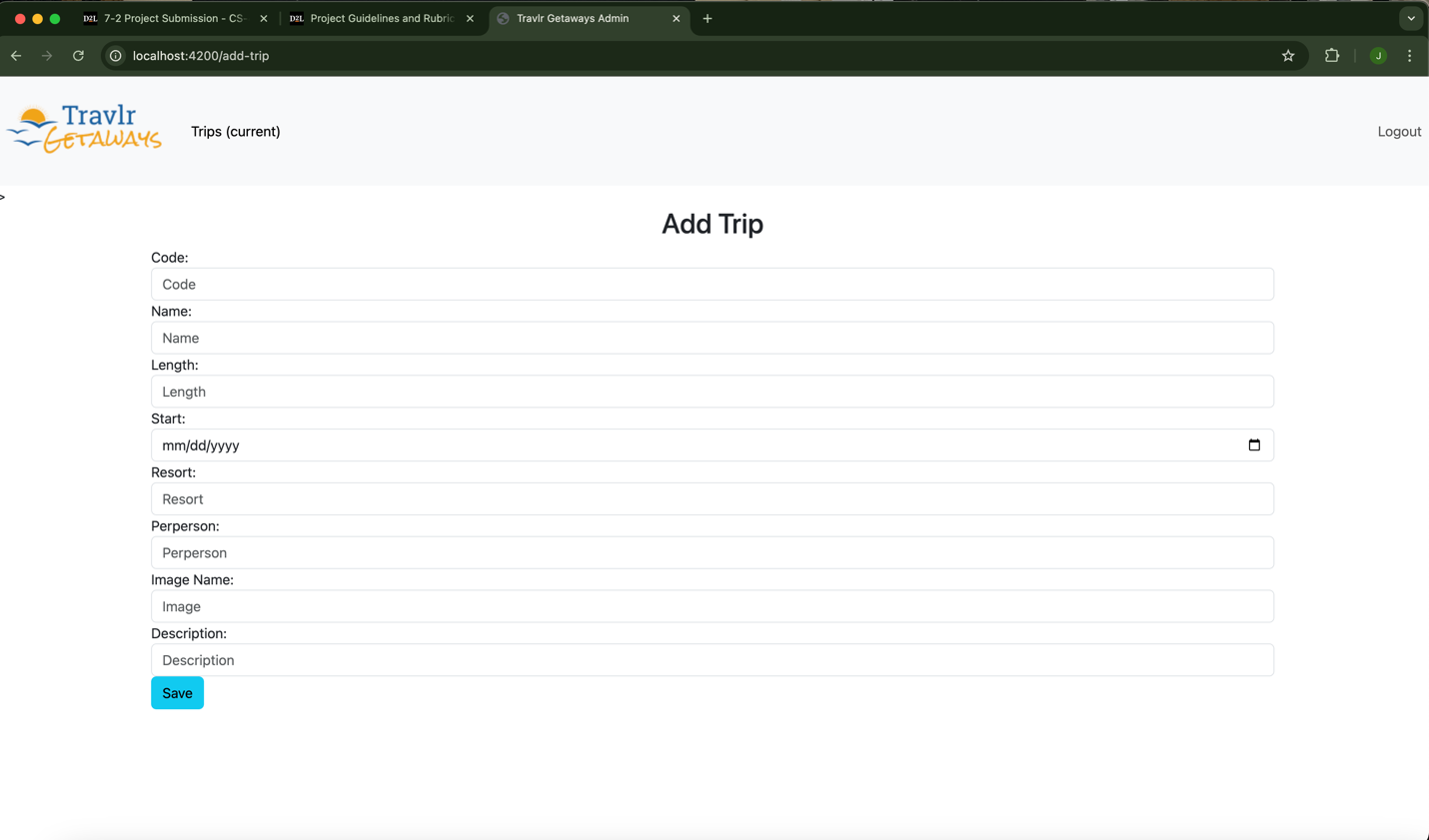
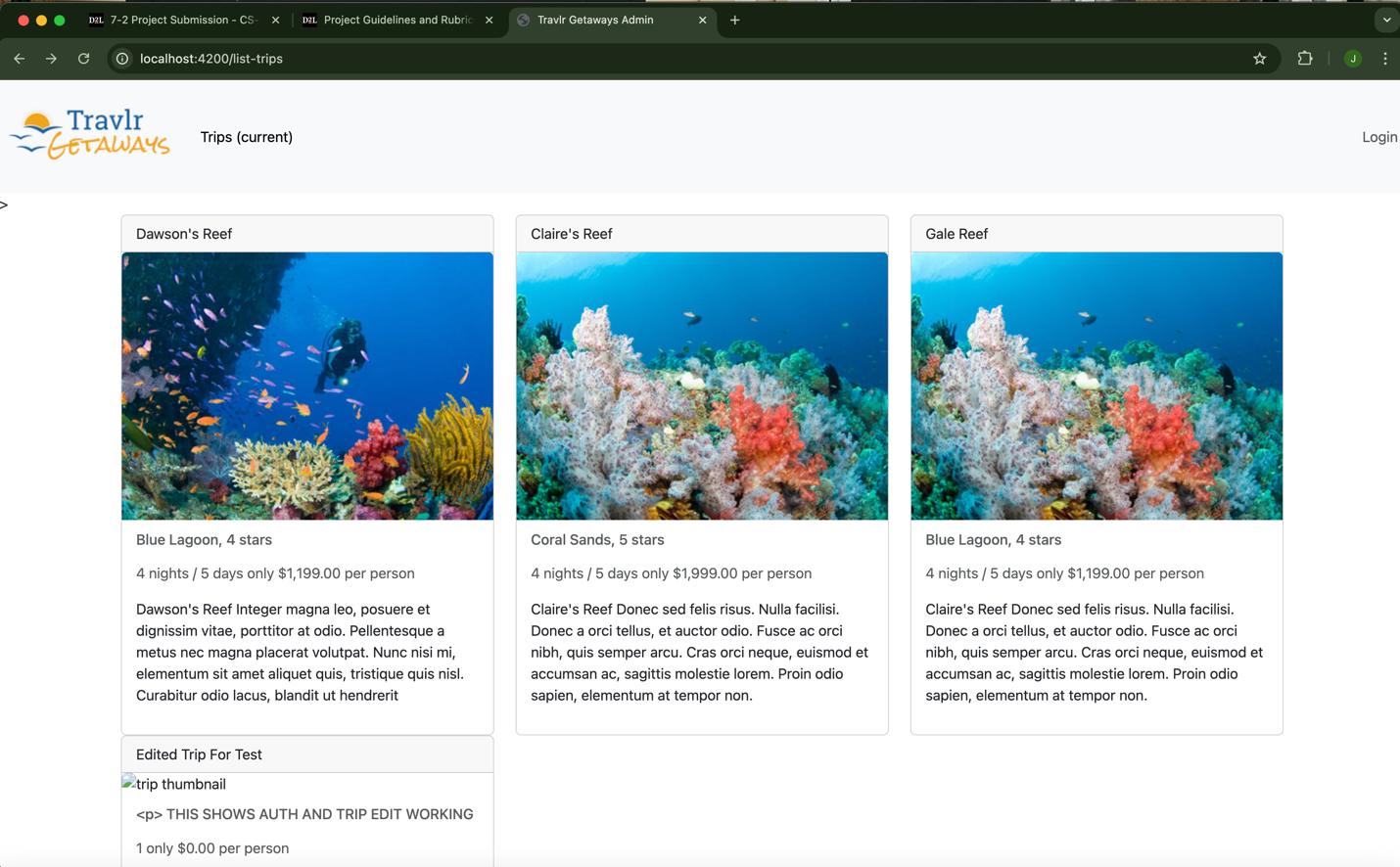
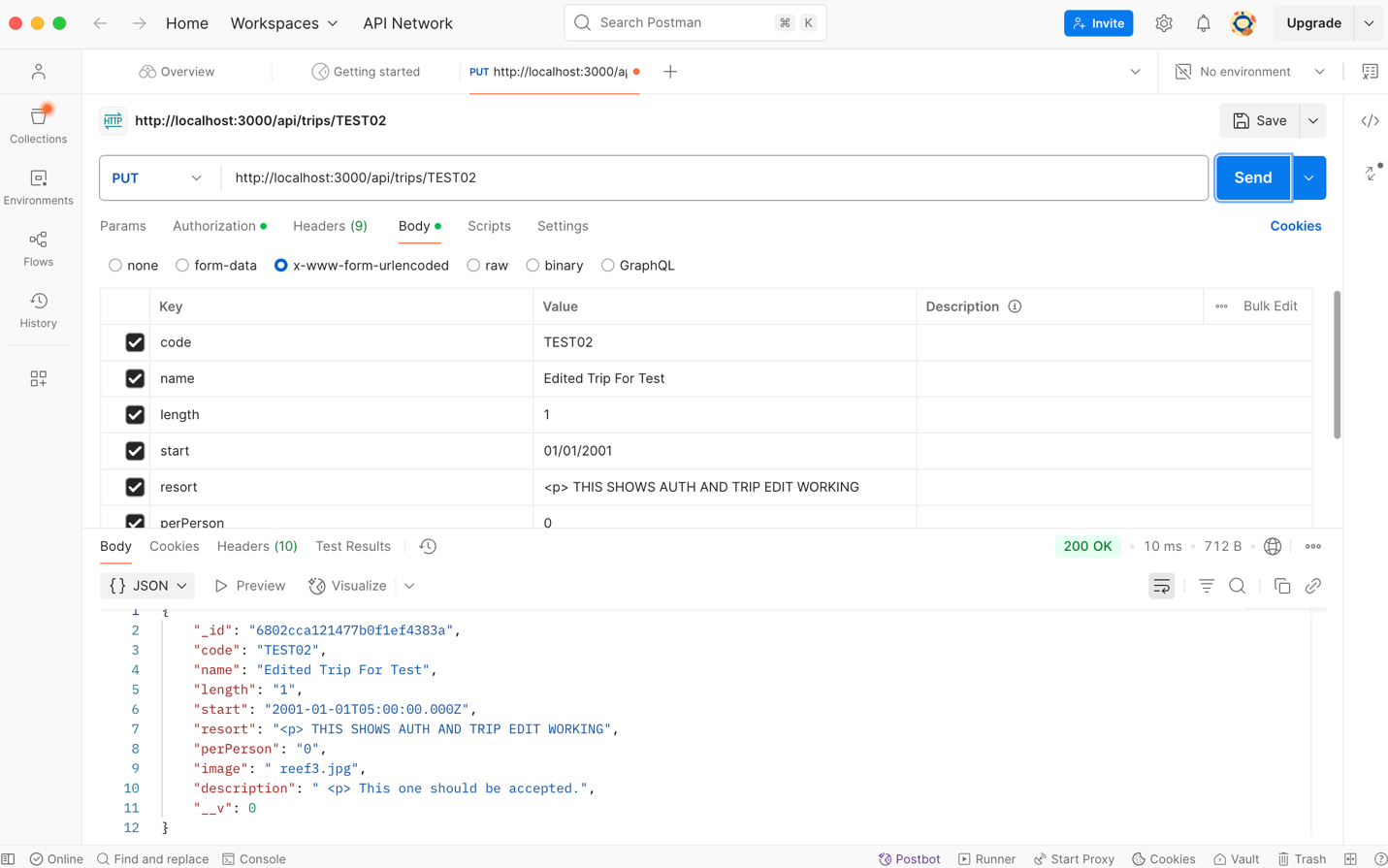
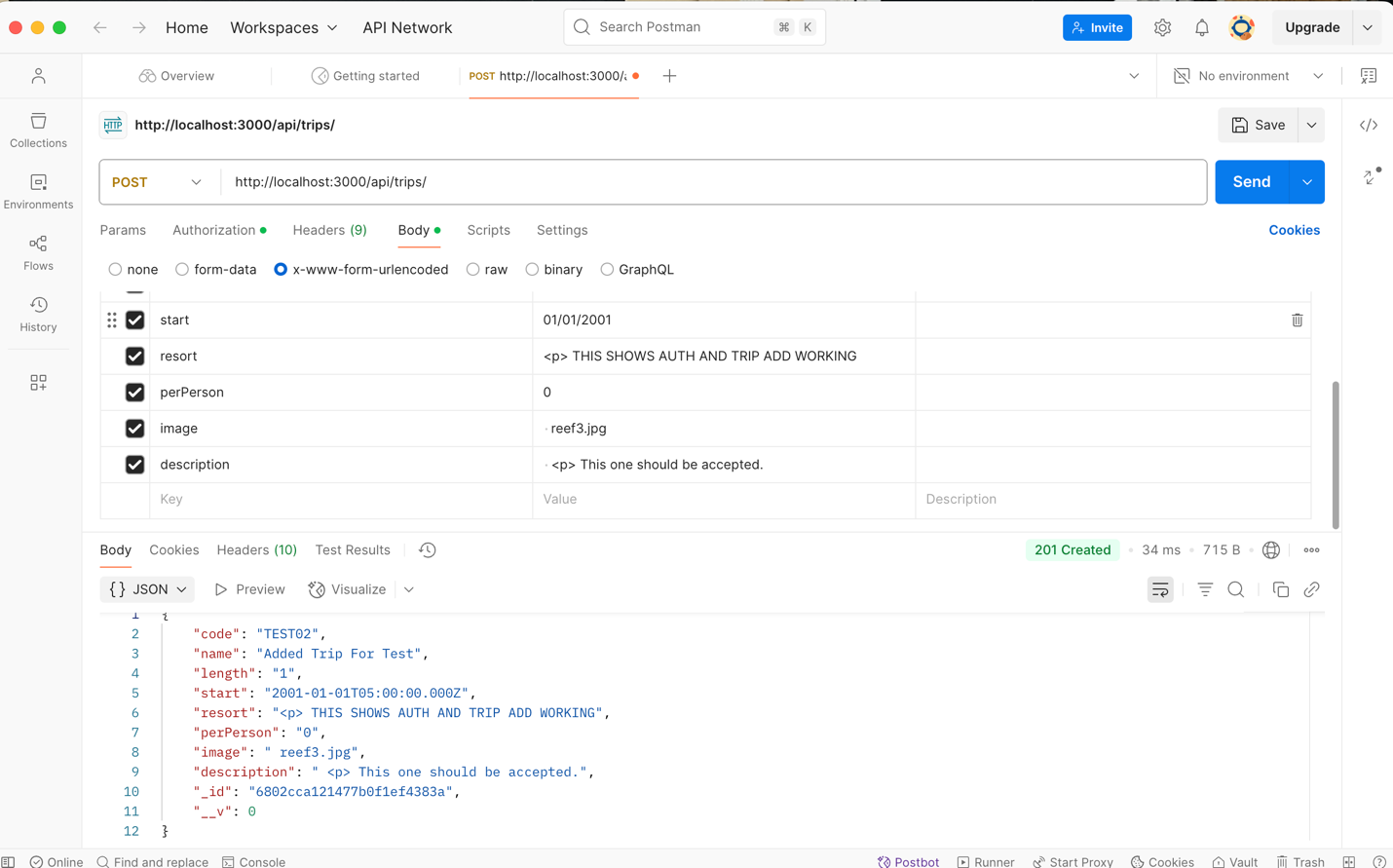
**Testing:**

The testing process took several approaches to ensure the SPA is working with the API to GET and PUT data in the database. The first step was ensuring the Express API properly communicates with MongoDB. This involved manually sending GET and PUT requests into the API using Postman. After verifying the API’s connection, the next step was to confirm the actual change in DB; this was accomplished with DBeaverEE, which allowed me to verify the records adjustment in the DB. I then started testing the connection to Angular. In the initial load, I could confirm that GET was working in the presence of the trips and console logs. I edited a trip to confirm that the PUT requests were working. The edits from the editor persisted, and I could confirm the API calls through the console logs. Finally, I confirmed the Angular adjustments in the DB through DBeaverEE again.

**References:**

Holmes, S., & Harber, C. (2019). Getting MEAN with Mongo, Express, Angular, and Node (2nd ed.). Manning Publications

**Screenshots:**

****