

Full Stack MEAN Web Application

# **CS 465 Project Software Design Document**

Version 1.2

## 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 | 07/18/2025 | Fayaz Shaikh | Added the Executive Summary, Design Constraints, and System Architecture View. |
| 1.1 | 08/02/2025 | Fayaz Shaikh | Completed System Architecture View (Sequence Diagram and Class Diagram) and API Endpoints sections |
| 1.2 | 08/14/2025 | Fayaz Shaikh | Update API Endpoints, and completed User interface section, as well as refine Design Constraints |

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

Travlr Getaways is seeking to incorporate a new full stack web-based solution where customers can create accounts, search through travel packages by location and price point, and book reservations. This must also include an admin-only SPA (single page application) site where administrators may maintain the customer base, trip packages, and pricing models.

The stack will incorporate MEAN, consisting of a MongoDB database for storing all the relevant customer and package data, Express.js to create APIs that tie together the server and client, Angular as a front end framework for creating the client side site, and Node.js enabling the actual JavaScript code to be run server side.

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

The website will use the MEAN architecture as mentioned before. The main limitations here include the type of content that can be updated in the MongoDB database. The database can include titles and descriptions as well as pricings for booking options, but images are a big area that are limited. MongoDB can by default store files up to 16MB, which can be an issue for larger images, and means graphics and videos cannot be uploaded by an admins on the SPA site when managing bookings. Images are also tricky to sent via REST APIs, and often require to be stringified such as using Base64. The images that are on the web page will be part of the graphic library and images subfolder on the static html side, with dynamic elements specifying which subdomain to pull the image from.

Another design constraint to discuss is the usage of multiple different technologies, and having to ensure correct coordination to get all four of them working together. In the MEAN stack, we need to ensure that we are writing everything effectively so that data for example is not being passed around and converted multiple times. This includes the JSON data types we sent with our REST APIs to our Express/Node backend to populate our MongoDB database, along with similar datatypes being unpacked and displayed on the front end with Angular.

Another design constraint to consider is how the software, specifically the single page application hosted on the web, will be interacted with and displayed on various devices of varying sizes and interaction types. For example, some elements may work best with a keyboard and mouse, and struggle with a touch screen interface like that of a smart phone. Careful consideration must be made to make the UI friendly to a wide variety of mediums to be used.

Lastly, a large design constraint occurs with the concept of using a single page application. With dynamic rendering, web data cannot be easily scraped, with costs a page to have worse performing SEO, as search engines cannot easily index and rank the web page. This is a limitation that needs to be considered, and can lead towards the consideration of static routing.

## [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 overall system architecture consists of the client component, server component, and database component. The server consists of an authentication server that communicated directly with the client where the front end web page is hosted. Once authenticated, the server session is initiated where users can access the Travlr Database. This Travlr Database contains all the relevant information to trip information including images, descriptions, titles, and other elements related to and and controlled by Travlr. The server also uses Mongoose ODM to talk to the MongoDB database, which takes us to the next component, the database.

The MongoDB database contains all the relevant storage for customer information, including personal identifiable information as well as bookings tied to each user. It is imperative that this database is modifiable by authenticated users from both the server side, so that admins logged into the SPA can interface with the booking information, as well as the client-side users, where customers of Travlr can create booking and update their personal information, which ties to the database. This takes us to our final component, the client.

The client, which also interfaces with the database once authenticated to a specific customer, consists of the client session which communicated with the authentication server to log in, as well as the traveler portfolio, which is displayed on the user’s web browser, as well as the graphic library which hosts all of the information related to how the website should look. This graphic library will also host all the static images that can be specified dynamically through string routes passed from the server, but stored permanently on the client side of the web page. Finally, as mentioned, the user will communicate with the database when making reservations or updating their information.

**Sequence Diagram**

A diagram of a diagram

AI-generated content may be incorrect.

Starting from the client side in Angular JS, the flow of logic in the web application starts with the actor on the far left. The actor calls a route on the web page. For the purpose of this example, let’s consider the lookup of a trip interaction. This route, along with the tripcode, is then passed as the route, which is redirected to a view on the Browser / View / Template. This then interacts with the controller, who calls a service on the HTTP Client to retrieve the data, formatting a REST API. This includes the tripCode that was passed by the actor, along with the formatted GET request.

This HTTP request from the client goes to the controller function on the server side (Node JS/Express/Mongoose), where the controller calls the controller/model service to retrieve the proper trip data that was requested. This goes to the Data Tier, with the MongoDB database, as the request is abstracted through Mongoose ODM. MongoDB then processes the request, searching for the trip code in this case, and returns the result. The callback is sent back to the controller/model, which returns a json formatted response promise all the way back to the HTTP Client on the client side. This json body should include the trip that matches the tripCode.

Back on the client side, the promise results object is passed to the controller, which assigns it to scope. It then passes it back to the browser/view/template to display this response in the view, where we end at the X at the bottom. Here, we are passed back the trip data for the matching trip, where the browser formats it in the way we want, for example, as a trip block with the picture, title, and description.

## Class Diagram

A diagram of a travel company

AI-generated content may be incorrect.

The completed class diagram can be seen above, filled with all the appropriate class titles based on the titles provided. Starting from the first top left entry, we have the base itinerary, which has a price, miles, and stop over which can be stored as a string. The price can be stored as a float to accommodate cents. This breaks down to an aggregate relationship to the three main classes that can form an itinerary: the cruise information, flight information, and hotel information, logically containing them but allowing them to exist independently. These classes work together to get the appropriate information, with the booking classes using accessors with the traveler’s information to get the correct trip information. Each type of trip shared a general trip info, with a starting date, return data, origin, and destination.

Moving back from the booking process, we have the Travel\_Agent who has the ability to book packages, flights, hotels, and cruises. This links to the Traveler info as well, which contains the number of companions that each trip should accommodate.

Taking another step back, we get to the Membership\_Admin, where we can see the functions that a lot us to credit points, get points, and validate. This has an aggregate relationship to the MemberAccount, which contains the member number, frequent airline, member status (as an integer), and member club.

Ultimately, all these classes work together to book the trips and create an itinerary to be displayed. Each box represents a JavaScript class in the web application. Each class encapsulates the data and methods as described above to be used in the front-end to display details and back-end for management functions, such as the Membership\_Admin.

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

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve a list of all trips | /api/trips | This GET method returns a list of every trip that is available. If there is nothing available, a fallback message is displayed stating this. |
| **GET** | Retrieve a single trip information given the trip code. | /api/trips/:tripCode | This GET method returns the JSON data for a single trip, based on the entered tripcode that matches. |
| **POST** | Adds a new trip to the list | /api/trips | This post method adds a *new* single trip to the database. A JSON body type that contains all the object’s parameters are required. **Authentication is required.** |
| **PUT** | Update a single trip information given the trip code. | /api/trips/:tripCode | This PUT method requires a JSON body type that contains an updated trip with all of the parameters of the object. The new body replaces the trip in the database with a matching code. **Authentication is required.** |
| **POST** | Logs in the user with credentials | /api/login | This POST method attempts to log in the user, given an email address and password. |
| **POST** | Creates a new user account with credentials. | /api/register | This POST method registers a new user account in the database with the name, email, and password supplied. |

## 

## The User Interface

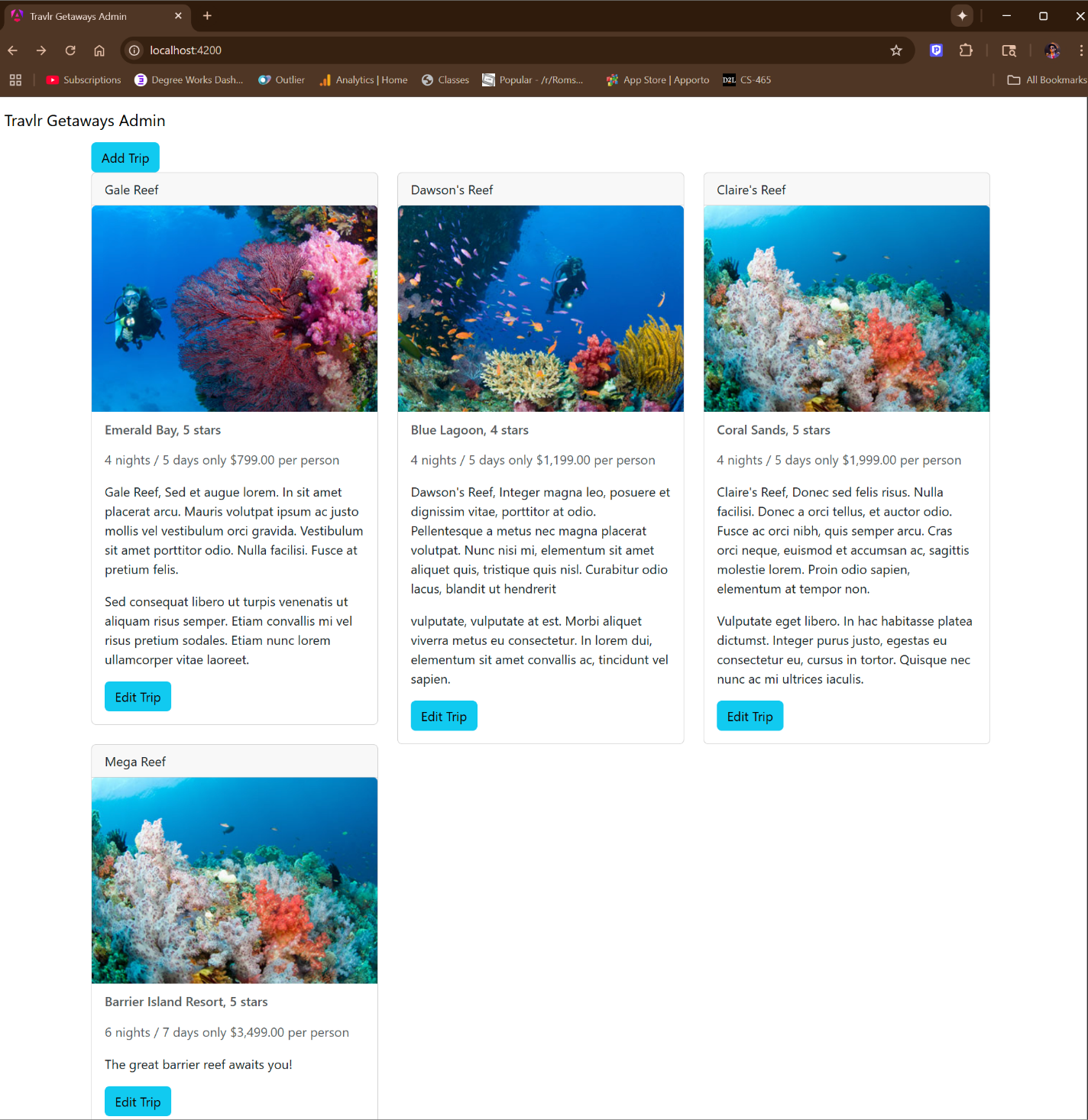


Figure 1: The main screen, including a new entry added.

As you can see in the screenshot above, the main screen shows a listing of all of the trip data formatted. A fourth custom trip has been added at the bottom. Each trip has an edit button which we explore in the next screenshot.

A screenshot of a trip

AI-generated content may be incorrect.

Figure 2: The update screen for editing the data of one of the trips.

Here is the edit screen for a trip. All data of the trip is pulled and prepopulated into the fields to be edited. Here you can see I’ve changed the name and description of the first trip. Additionally, I’ve written logic to ensure the start date is formatted correctly to be used in the picker.

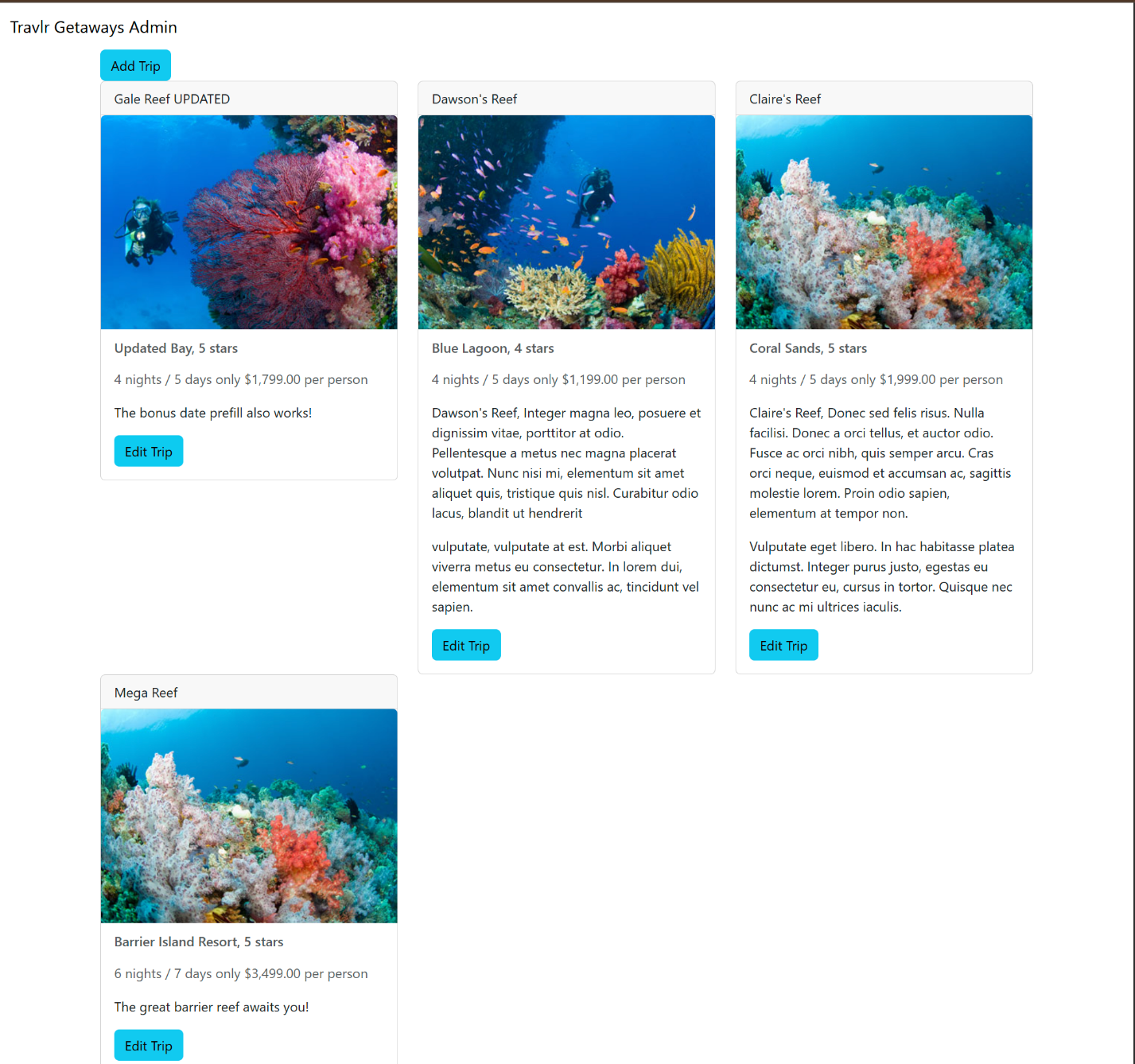


Figure 3: The updated screen, showing the new values for the first trip.

Finally, we can see the results of pressing the save button, with the trip data being updated and reflected on the main screen. Aan authenticated user will see the edit button and can make these changes.

**Angular vs Express**

The structure with Angular differed from the structure we built with Express for the HTML customer-facing page. Firstly, with Express, we used handlebars that allowed us to pass values into the html page. With Angular, we were able to create modular components, each one with its own folder package containing css styling elements, a typescript file detailing what imports and functions we'll use, and of course the html file itself of how the element should look. Our top level module in Angular used a culmination of the multiple modules to pull and display all the information we wanted on the customer facing page. With Express, we followed a little bit less of a modular method, and instead had routing to each page that was rendering its own HTML that had a lot of code copied in multiple places. While we did modify this to use a consistent header and footer, it was originally a less modular approach.

The single page application we configured with Angular had many advantages and disadvantages. Firstly, it can be a lot less complicated for the end user. All interaction on the page to progress is managed through interaction with certain fields, which can then pass information to the next module-- in our case, using local storage. It's a fast user experience that doesn't require page reloads. This does come with a big trade off though, as all the content loads when you first connect, leading to longer upfront loading times. It also becomes more challenging to ensure everything works-- if something is broken, the user cannot simply refresh, or they may go back to square one. Instead, it's important to ensure that every interaction works and allows you to go forward and back at any step. This can be a pain to deal with as a project grows, as compared to the routes we used with Express having different pages you can directly go to for each interaction.

**Testing**

There was a threefold process of testing the SPA to ensure the API for the GET and PUT requests were being processed correctly in the database. Firstly, a simple request could be sent with Postman. Postman allows us to configure any sort of REST API request to be sent to a given URL, in our case, localhost on the right port. Here, I entered GET to ensure the API was returning the correct values, as well as POST to test adding trips and filling in all the information required. It was essential to format the JSON data correctly with the expected values so the request could be processed. One issue I was running into is I kept getting connection refused, and I was getting a little discouraged, when I decided to do a sanity check and restart my servers when I realized the Express server wasn't even live, so it couldn't process any requests. After starting it up again, it immediately started returning 201s like I had configured it to. Then, I needed to check MongoDB Compass to ensure that the values were actually being passed correctly to the database. Here, I was able to connect to the localhost again and see the travlr database being used, and confirmed new entries showed up. I manually deleted them, and then redid the POST and confirmed it still worked. Next, I created the PUT function and needed to test it. Specifying a specific trip to update, I followed the same process in Postman just updating the URL and method. This worked and showed on MongoDB Compass, so it was time for the third and final step of testing, ensuring it worked on the Angular side. Here, I entered example values and hit the add and update buttons (for creating a trip and editing a trip) and both worked!

Looking at the console, I noticed as pointed out by the full stack guide that the date being received from the database in ISO format could not be parsed and refilled in the date slot of the edit form. I did some extra logic here to convert in place the date to convert it from ISO format to the expected yyyy-mm-dd format, and confirmed date prefill was now working as expected when editing trip data. Editing it in place as compared to populating the formatted data as a separate entity afterwards meant that it never tried to populate an invalidly formatted date, so never gave errors.