

Travlr Getaways

# **CS 465 Project Software Design Document**

Version 2.0

## Table of Contents

[**CS 465 Project Software Design Document** 1](#_Toc164629705)

[Table of Contents 2](#_Toc164629706)

[Document Revision History 3](#_Toc164629707)

[Instructions 3](#_Toc164629708)

[Executive Summary 4](#_Toc164629709)

[Design Constraints 4](#_Toc164629710)

[System Architecture View 5](#_Toc164629711)

[Component Diagram 5](#_Toc164629712)

[Sequence Diagram 6](#_Toc164629713)

[Class Diagram 7](#_Toc164629715)

[API Endpoints 8](#_Toc164629716)

[The User Interface 9](#_Toc164629717)

**Figures Table**

[Figure 1. Component Diagram 4](#_Toc164629661)

[Figure 2. Sequence Diagram 5](#_Toc164629662)

[Figure 3. Class Diagram 6](#_Toc164629663)

[Figure 4. Add Trip Screen 8](#_Toc164629664)

[Figure 5. Trip Added Screen 8](#_Toc164629665)

[Figure 6. Edit Trip Screen 9](#_Toc164629666)

[Figure 7. Successfully Updated Trip Screen 9](#_Toc164629667)

## [Document Revision History](#_heading=h.lnxbz9)

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 03/28/2024 | Elora Newcomb | Initial Release |
| 1.1 | 04/05/2024 | Elora Newcomb | Sequence and Class Diagrams, API Model |
| 2.0 | 04/20/2024 | Elora Newcomb | Angular Structure, Testing, and added Figures Table |

## 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)

Travlr Getaways uses cutting-edge technology to deliver an unparalleled user experience, utilizing the MEAN stack - MongoDB, Express.js, Angular, and Node.js - to seamlessly integrate data management with dynamic content delivery.

By combining Express.js and Handlebars, users can benefit from efficient and appealing features at once: in the context of static content, Express.js is responsible for rendering, while Handlebars dynamically inject data from MongoDB's database onto each page. Meanwhile on the admin side, Travlr Getaways' features a single-page application (SPA) that empowers administrators with unmatched control their virtual domain, like web application CRUD (Create, Read, Update, Delete) changes, remove permissions in case they are super admins, manage customer’s access to website, and manage generated LEADS for marketing purposes.

We also ensure customer and admin access is secured by combining Angular's power with MongoDB's architecture, requiring token authentication and verification.

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

1. Compatibility: Ensure the app works on all devices, special focus on mobile.

2. Performance: App must be still operational under heavy traffic in case we ever run promotional campaigns for bookings.

3. Security: Protect user data with encryption along with internal auditing to ensure information isn't being leaked from within the organization.

4. Scalability: Plan database ahead for user and data growth.

5. Compliance: Follow laws like GDPR for EU users.

6. User Experience: Make sure navigation is easy to use and engaging for a wide range of age groups.

7. Budget and Time Constraints - Plan features based on budget and set our timeline restrictions ahead of time, before minor issues during development evolve into massive delays in production.

The major constraints cited above should be taken into account whenever developing a web application as they shape the entire workflow not just for engineers, but for the operations team as well. By guiding these major decisions, we can create an application that meets users' needs effectively as we navigate the inevitable hiccups that will inadvertently occur on the way. Constraints are also beneficial in that they provide a roadmap for engineers to follow, allowing for the development team to focus on key tasks as well as identify which features are most critical for each release.

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

### Component Diagram

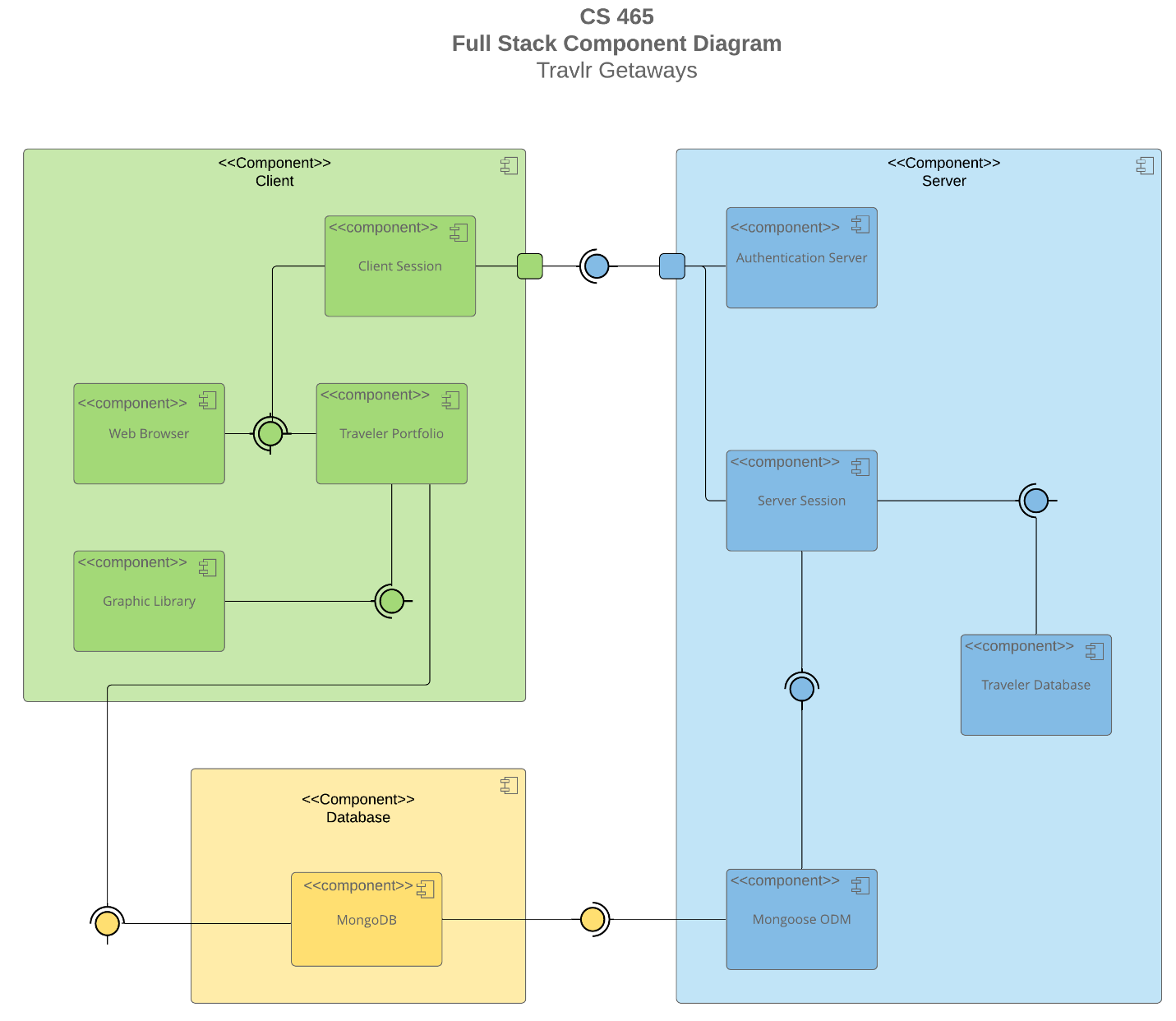


Figure 1. Component Diagram

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 frontend, or client-side, operates on Angular.js and includes various components facilitated by the MEAN stack, such as an image library and session handling for logged-in users. For instance, authentication leads to a session starting once the user types in their unique log-in credentials through the client-side. The client sends the request to the backend for evaluation, which in turn is responsible for validating said credentials by comparing them with those residing in the database. The backend, or server component, is responsible just for that; to manage authentication and communication between the MongoDB and frontend for data schema validation (we are able to manipulate stored information more easily by employing the use of Mongoose).

Important to note that the database, while physically on the server, is a separate component that can connect independently of the client or server, by storing site information such as images and package descriptions in MongoDB which can be served to either the frontend or the backend depending on the type of call - also known as server requests - that is being made.

According to the diagram above, the server element is composed of four parts that intrinsically rely on each other: *The Authentication Server, Server Session, Traveler Database, and Mongoose ODM*. This is because the Traveler Database needs the interface provided by the Server Session, and the Server Session needs the Mongoose ODM interface. Finally, the Travlr Getaways’ Database element includes a single MongoDB component that provides an interface to both its Client and Server elements.

### Sequence Diagram

### 

Figure 2. Sequence Diagram

In the sequence diagram above we illustrate a web application that utilizes a MEAN stack architecture, with a client-side (AngularJS), a server-side (Node.js/Express/Mongoose), and database (MongoDB) components.

Whenever a user is directed to any of our content, AngularJS will route to the appropriate view. Routing essentially means determining which handlers within our application will be managing request and path, respectively, and through a browser or template the user can interact with the specified content.

HTTP client service requests are what bind the user view to controllers in order to retrieve relevant data from the server-side. The controllers interact with models in the source code, which in turn communicate with the MongoDB database through Mongoose (an Object Data Modeling (ODM) library for MongoDB) in order to determine which data matches that format.

For example, during a Sign In process the server-side controller would handle authentication logic, possibly querying the database for user credentials; for Trips, the controller would retrieve or update our trip information in the database. Though we haven't set up the Admin page yet, those interactions might include managing trips, updating existing ones, or even setting user roles and permissions in case the user has been previously designated as an administrator.

MongoDB's task is to process the Mongoose ODM request and return the results to the server. The server-side controller will then send the results back to the client as a JSON response (often wrapped in a promise for asynchronous handling). Once those previous promises/result objects reach the client, proper data can be extracted and assigned to the scope, displaying the updated information on the user's screen (e.g., the list of past trips, or a successful login action).

## Class Diagram

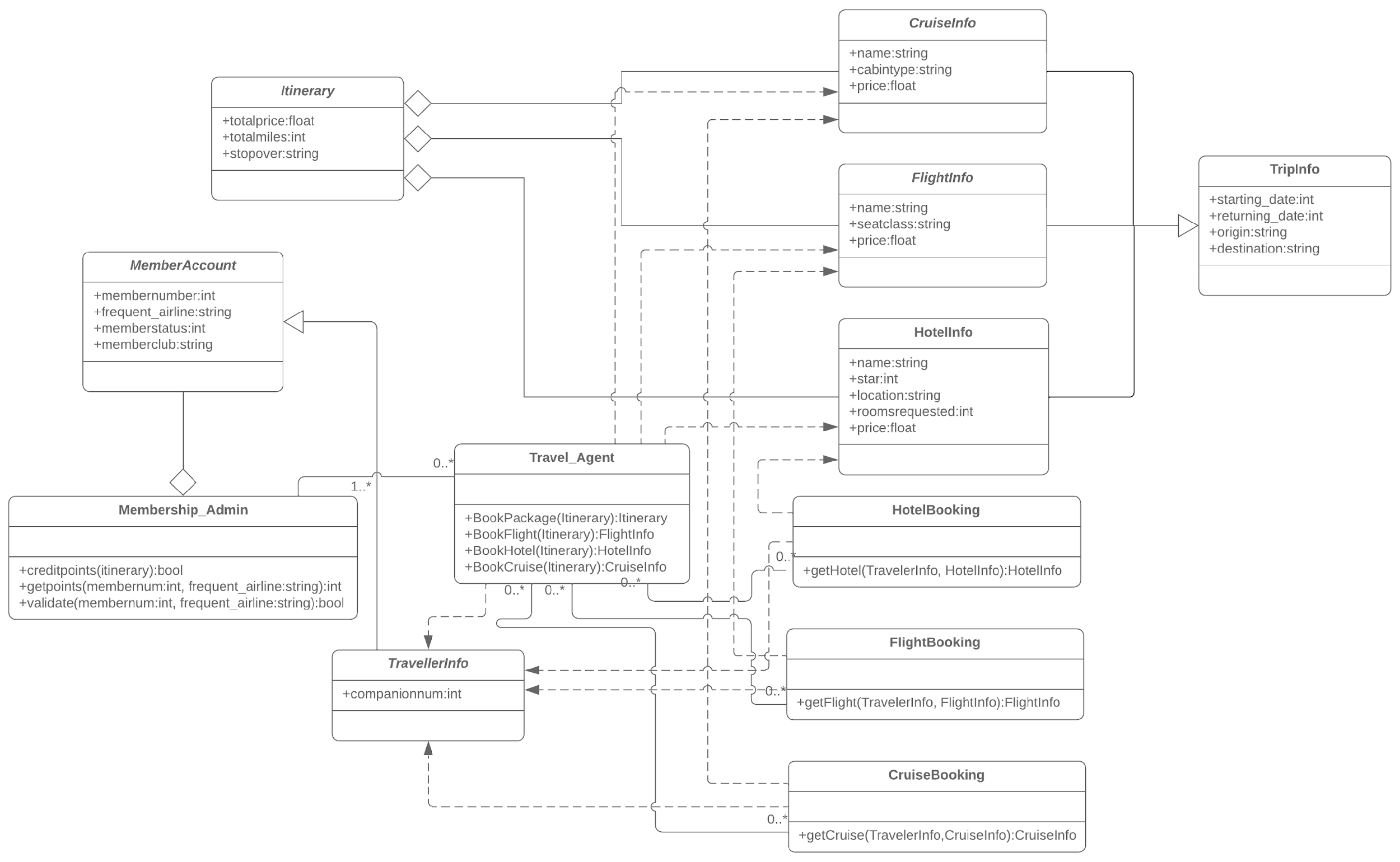


Figure 3. Class Diagram

*MemberAccount* is our starting point after creating an account in the application. *Membership\_Admin* is an aggregate of this class, which provides information and validates points if any have been earned. When we first create the account, however, the number of points will automatically default to zero since no trips have been recorded yet.

After the member has acquired enough points, the *Travel\_Agent* class will be employed in a one-to-many to a zero-to-many relationship. In other words, you can't have *Travel\_Agent* without having *Membership\_Admin* prior, since this class manages the points system for bookings. In a similar manner, we have the class *TravellerInfo*, which according to my interpretation of the class diagram, can also serve as a way to book packages when the user has no points recorded yet. Important to note that *Travel\_Agent* methods directly depend on this class, and that it inherits user information from the *MemberAccount* class, only adding the method *companionnum* for listing the number of companions that will be joining the traveling member when they’re about ready to checkout.

The *TravelAgent* class has a complex relationship with other classes, since it is responsible for converting points into travel packages for members by searching for hotels, cruises, or flights within their redeemable range (*FlightInfo, HotelInfo, CruiseInfo*) first and then making calls to the *HotelBooking, FlightBooking*, or *CruiseBooking* classes.

*TravellerInfo CruiseInfo, FlightInfo, and HotelInfo* classes describe the nature of each trip, inherit information from the *TripInfo* parent class, and share zero-to-many relationships with the *TravelAgent* class both ways. *Itinerary* class aggregates *CruiseInfo, FlightInfo, and HotelInfo*. This means that after making a selection and customizing their own packages in the *“\_\_\_\_Booking”* classes, the user’s choices will be placed in a container called *Itinerary* within the application.

(P.S. I must admit that this class diagram proved very challenging for me, even though it took me extra days after submission researching online to gather more information. I'm still not quite sure if I correctly understood what the *TravellerInfo* class is, since the method *companionnum* also isn’t self-explanatory. Professor, if you could clarify a little more in the grading I would greatly appreciate it.)

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

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve all trips in the database. | /api/trips | GET method returns all JSON data entries pertaining trips that can be found in the database. |
| **GET** | Retrieves a single trip by *tripCode,* their unique ID in MongoDB. | /api/trips/:tripCode> | GET method returns one JSON data entry pertaining trips that can be identified by its unique ID passed on the requested URL |

## The User Interface

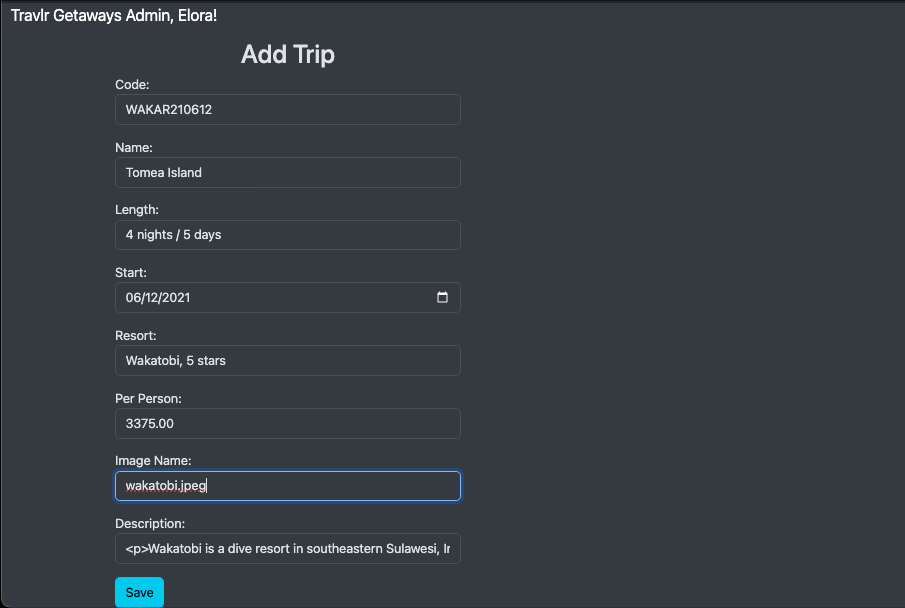


Figure 4. Add Trip Screen

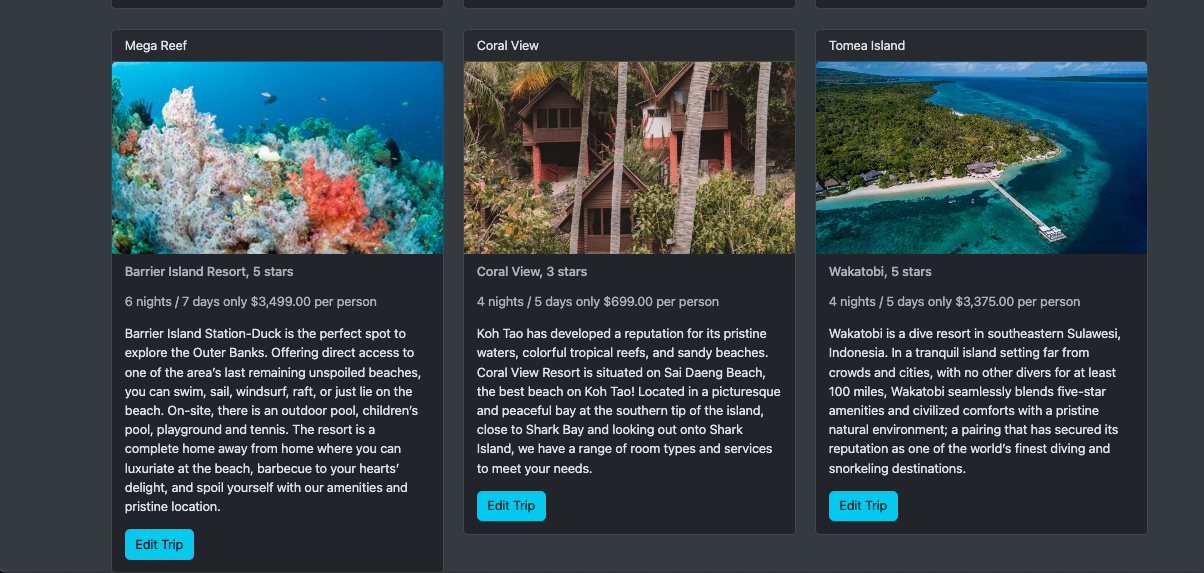


Figure 5. Trip Added Screen

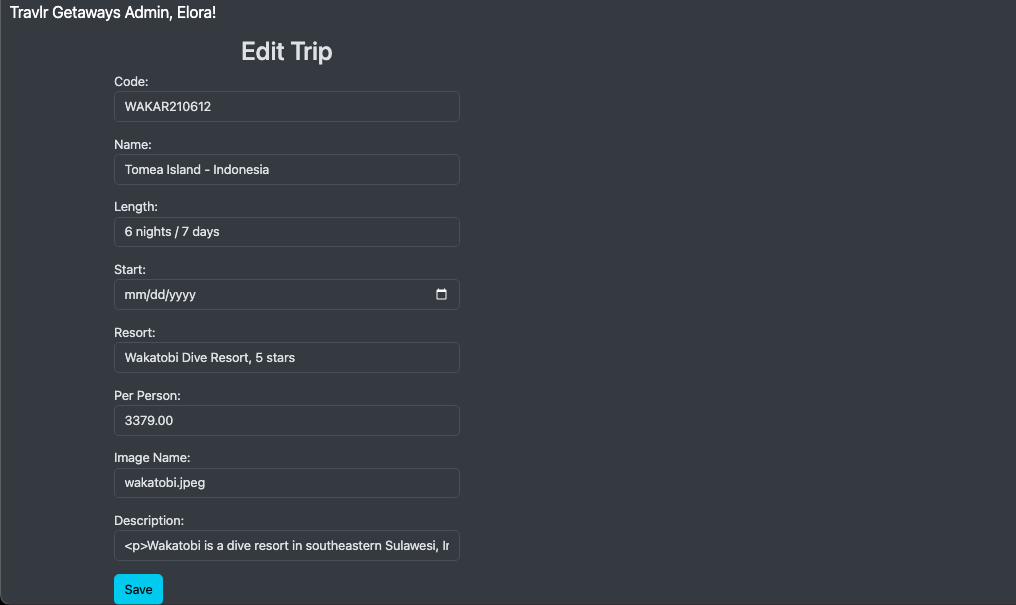


Figure 6. Edit Trip Screen

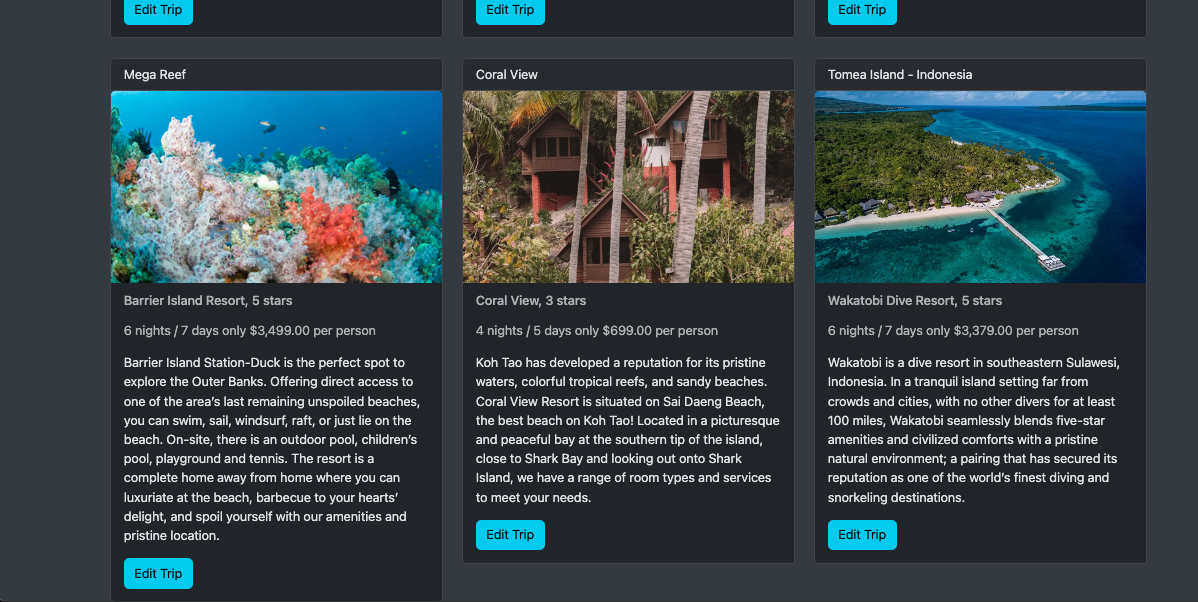


Figure 7. Successfully Updated Trip Screen

**Angular Project Structure**

With its powerful client-side framework, Angular boasts a sophisticated project structure that encapsulates components, services, modules, and more within its web application architecture. An Angular project follows a modular structure, where each component plays a vital role in harmonizing the user experience. On the other hand, Express HTML customer-facing pages exhibit a more traditional approach to web development. With Express as a server-side framework for Node.js applications, the focus lies on the simplicity of serving static files or dynamically generated HTML pages that cater to immediate user interactions without the need for complex client-side logic. In essence, while Angular dazzles with its robust front-end capabilities and structured project layout tailored to single-page applications (SPAs), Express is better suited for handling server-side logic and running back-end application, since it is a lightweight framework that enables routing, HTTP requests, and error handling.

**SPA vs Simple Web Application**

Some advantages can include: faster loading, since it only needs to fetch data once from the server, reducing unnecessary page reloads; users can navigate through content without interruptions or delays due to loading new pages; some features can still work offline due to local caching, enabling users to access limited content, even if they don’t have an internet connection. Regarding some additional functionality that cannot be achieved in non-SPAs, I would cite the real-time updates with WebSocket integration for live chat features or notifications, something undoable in regular web applications; robust form validation with reactive forms in Angular as we are creating in the current project; lazy loading images or videos for optimized performance and, finally, client-side routing for seamless navigation between different views within the same page.

Some disadvantages can be cited as well: Since most content is loaded dynamically via JavaScript in SPA frameworks like Angular or React, search engine crawlers may struggle to index relevant information for ranking purposes; initial load time of a SPA can be longer compared to traditional web applications as it needs to download larger bundles upfront; older browsers, like Safari, may not fully support modern JavaScript functionalities actively used in SPAs.

**Testing HTTP Requests**

Establishing clear communication protocols between the SPA frontend and the backend API is the first step in the process. This involves meticulously defining endpoints for GET requests to fetch data from the database, as well as PUT requests that can update or modify existing information. The API should respond appropriately with relevant data payloads or status codes based on these interactions. Next comes the testing phase where we can use Postman and DBeaverEE for debugging and ensuring the connection isn't being blocked and data can be created and recorded in the database. By examining response times, headers, payloads returned from each request against the expected results outlined in the documentation, anomalies can be identified and fixed. In terms of potential errors that may arise during this testing endeavor the most prominent one, which I also had trouble with, is misconfigured CORS policies, which can cause cross-origin resource sharing errors when attempting AJAX calls from the local server. Other instances I experienced include parsing errors due mismatched JSON structures sent or received, and unexpected shutdown of backend services disrupting connectivity. Additionally, errors related to authentication tokens not being passed correctly or expired sessions causing unauthorized access attempts are common issues reported in forums like Reddit or Stackoverflow.

**The Overall Testing Experience**

For more accurate evaluation, a robust testing environment must be set up to simulate a real-world scenario. This involves creating mock data sets that mirror database entries to ensure comprehensive coverage during testing phase. By scrutinizing each function's behavior under varying conditions, potential bugs or discrepancies can be identified early on before they escalate into larger issues. Furthermore, data insertion tests using applications like DBeaverEE can assess how well the server is interacting with the database when retrieving or modifying data via HTTP requests. Verifying proper routing functionality and response handling ensures smooth communication between client-side interfaces and server-side endpoints. This holistic approach guarantees that all features are functioning together as intended without glitches disrupting user experiences. In conclusion, diligent attention to detail throughout every phase of testing, from unit assessments to integration checks and end-to-end validations, is paramount in guaranteeing flawless synchronization between a SPA built on a MEAN stack framework and its underlying API infrastructure. Only through rigorous examination can we deliver uncompromising quality under the most demanding conditions prior to launch.