

<Name of Software Application>

# **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 |
| --- | --- | --- | --- |
| 1.0 | 4/19/25 | Christian Wallace | Created documentation |

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

The Travlr Getaways web app is being developed using the MEAN stack MongoDB, Express.js, Angular, and Node.js. This tech stack supports a fast, scalable, and fully featured application, with JavaScript running seamlessly across both the front and back end.

The app will have two main interfaces. On the front end, there’s a customer facing website where users can browse travel packages and trip options, all dynamically rendered using Handlebars templates with Express. On the back end, there’s an admin-facing Single Page Application (SPA) built with Angular, designed for real-time management of trip data—allowing administrators to create, update, or delete listings with ease. By separating the customer and admin experiences, this setup keeps things organized, user-friendly, and easy to maintain as the project grows.

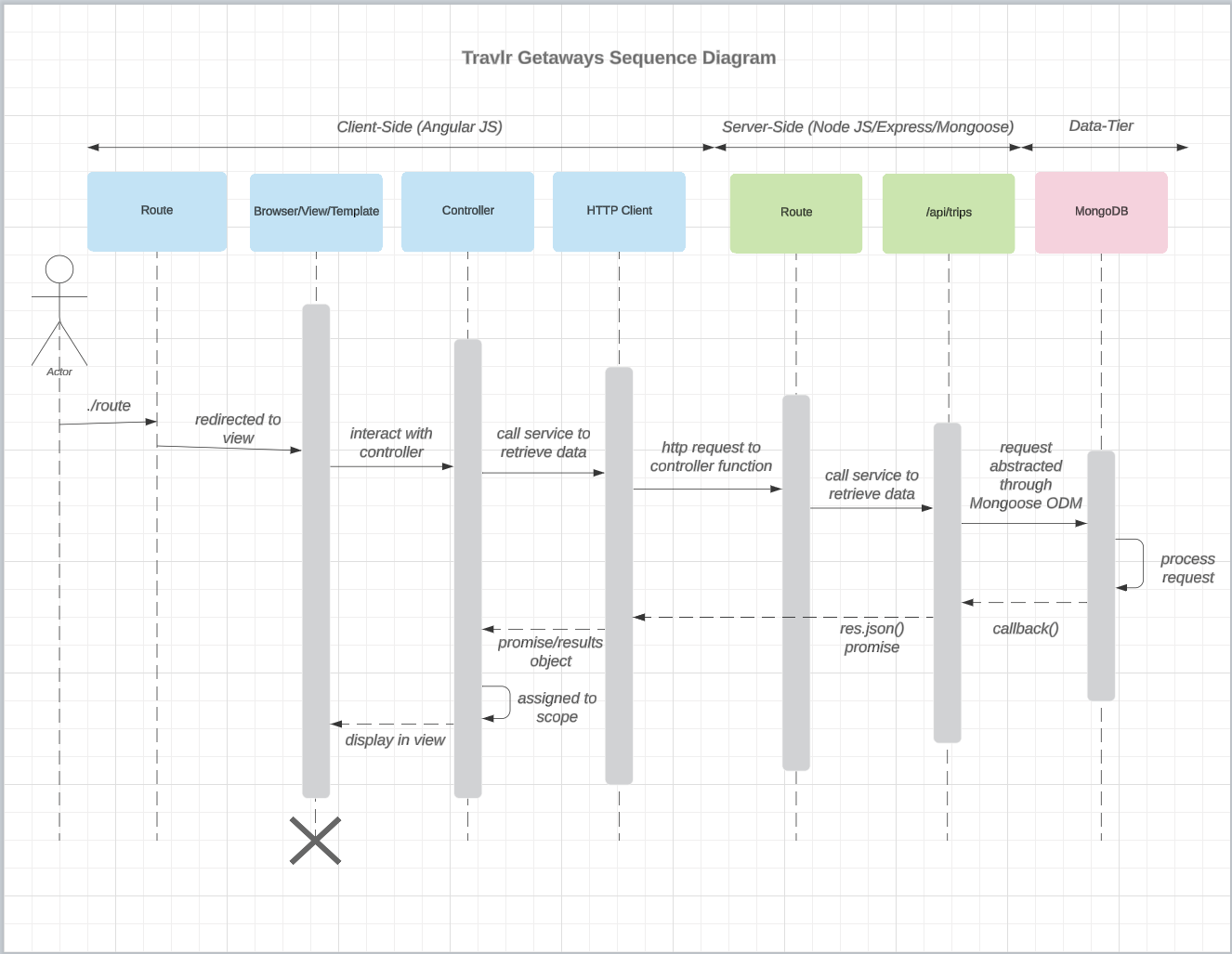
## [Design Constraints](#_heading=h.1ksv4uv)

The tech stack is locked in we’re required to use the MEAN stack. That means everything has to be built with JavaScript or TypeScript on the front and back end, and the database must be NoSQL. There’s no room for other frameworks or languages. The design and functionality need to closely follow the client’s wireframes and requirements. That limits how much we can deviate in terms of UI and user experience we’re working within a pretty set vision.

Security is also a big focus. Admin features need to be protected, so things like authentication and route access have to be implemented carefully to prevent unauthorized access. The app also has to work smoothly across all modern browsers, which rules out the use of cutting edge or experimental web features that might not be fully supported everywhere.

Everything must run within a Node.js environment. That means any libraries or custom code we use have to be compatible with that setup to avoid issues during deployment. These constraints help shape the development process and force us to stay focused, while still leaving room for creative problem solving within the boundaries.

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

The flow of logic in the Travlr Getaways web application begins with a user interacting with the client-side interface. When the user navigates to a route (/trips), the request is handled by the Angular route, which redirects the user to the appropriate view or template. This view is connected to an Angular controller, which manages the application state and triggers the logic to retrieve data.

The controller makes a call through the HTTP client to send a request to the server. The request is received by the Express route handler on the Node.js server, which delegates the logic to a controller function. The controller interacts with the Mongoose model to query the MongoDB database. The model abstracts direct database access using the Mongoose ODM, sending the appropriate request to MongoDB.

Once MongoDB processes the request and returns the data, the model passes the results back to the controller. The controller then formats the data as JSON and sends it back to the client. The Angular HTTP client receives the response, and the controller assigns it to a scoped variable. The view renders the dynamic content for the user to see, completing the full round-trip flow.

### 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 system architecture of the Travlr Getaways web application is composed of three primary components: the client, the server, and the database. On the client side the application is accessed through a web browser, which serves as the main interface for both customers and administrators. Within the client component, a client session manages session data such as authentication tokens or cookies, while the traveler portfolio represents the interactive user interface that allows customers to view available trips. Additionally, a graphic library is used to enhance the user experience by supporting dynamic rendering of visual content.

The server component handles the core business logic and request processing. It includes an authentication server responsible for verifying user credentials and managing access to restricted features. Server sessions are used to maintain state and track logged-in users. The server also interacts with the traveler database component, which handles all operations related to trip data, such as retrieval, updates, and deletions. These operations are facilitated through the use of Mongoose, an Object Data Modeling (ODM) library that provides a structured interface between the server and the MongoDB database.

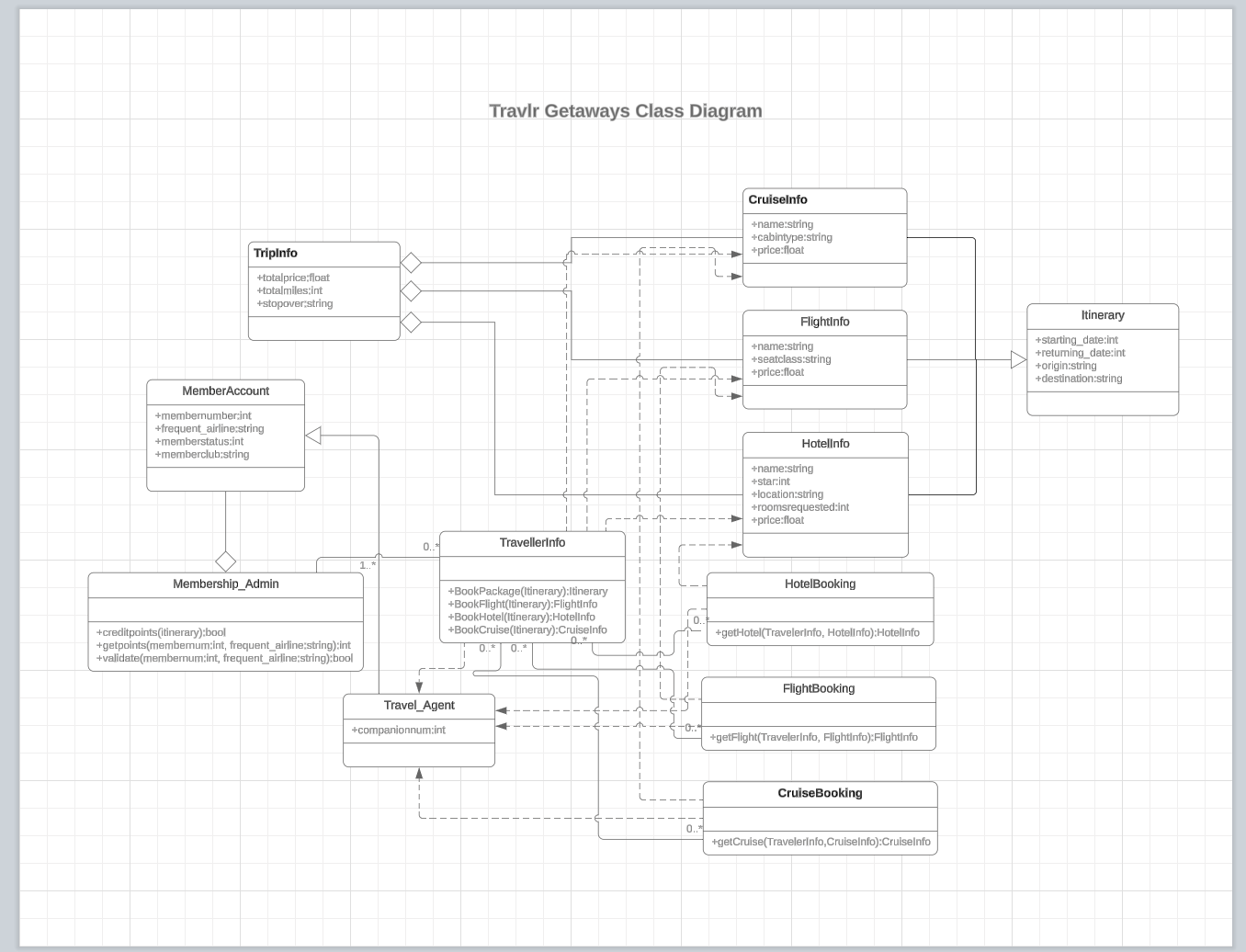
The database component consists of MongoDB, which stores all persistent data related to trips, users, and application state. It connects directly with the server via Mongoose, ensuring data consistency and enabling the application to perform real-time read and write operations. Overall, the architecture supports a responsive and scalable full stack web application by clearly separating concerns between the front-end client, the server-side logic, and the underlying database.

### Sequence Diagram

The logic behind how the Travlr Getaways web app works follows a smooth, layered flow from user interaction to backend processing. When a customer visits the site and heads to the /trips page, their browser sends a GET request to the Express server. Express then talks to the MongoDB database to pull the relevant trip data. Once that data comes back in JSON format, Express plugs it into a Handlebars (HBS) template to build a dynamic HTML page, which is then sent back to the user’s browser. This process makes it easy for customers to view real-time trip listings pulled straight from the database.

On the admin side, when an admin enters their credentials and hits submit, a POST request is sent to the server. Express handles that request by checking the provided credentials against the data in MongoDB. If everything checks out, the server responds with an authentication token or similar session-based method and redirects the admin to their dashboard. From there, admins can manage the trip listings by adding new trips, updating existing ones, or removing outdated entries. This flow keeps everything secure and ensures that both public and admin users see the right data at the right time, based on their role and access level.

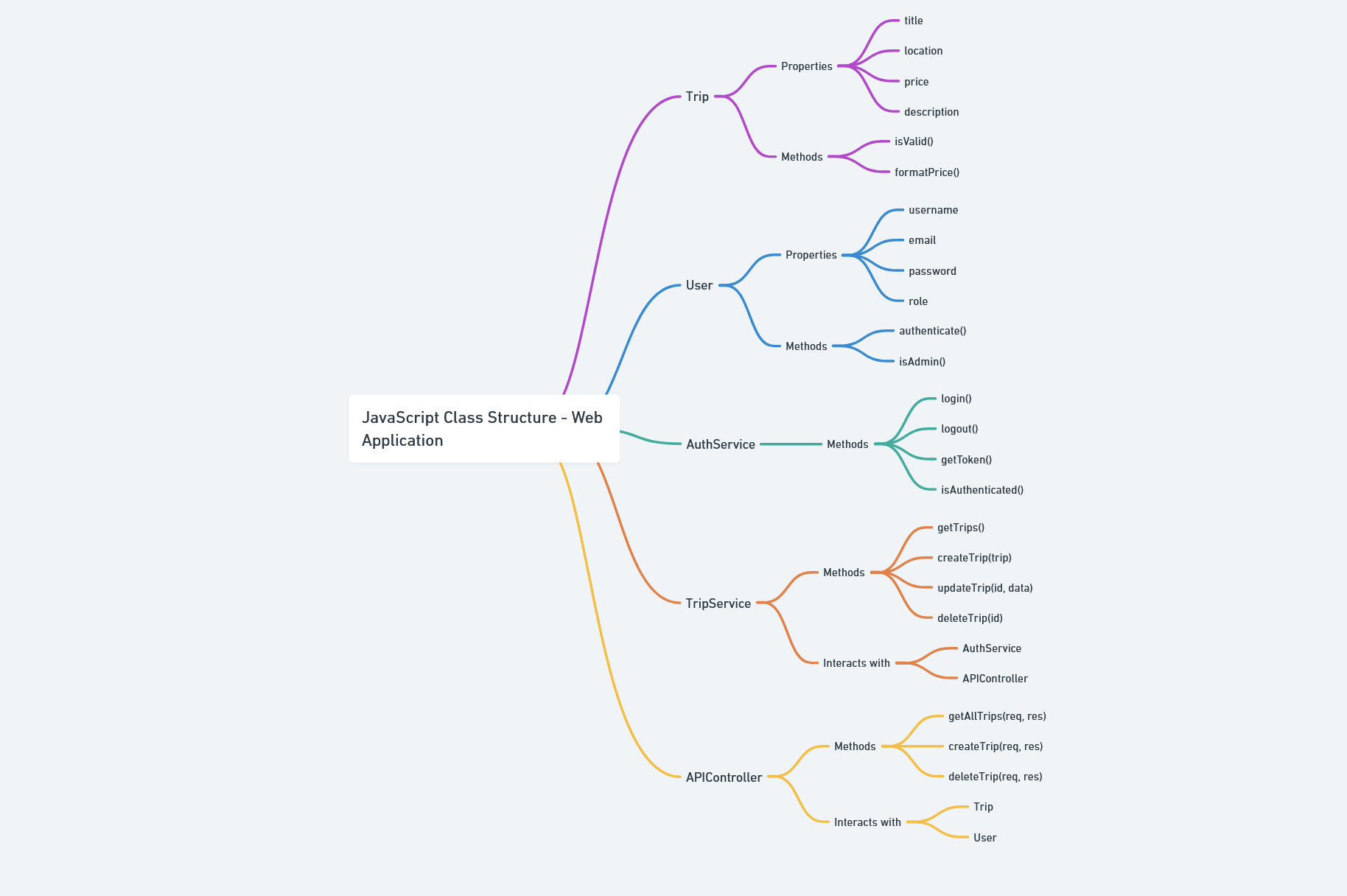
## Class Diagram

The class diagram for the Travlr Getaways web application outlines the major objects and their relationships, following the model view controller pattern. At the center is the Itinerary class, which serves as the container for travel details, such as trip dates, destination, and origin. It connects to TripInfo, which stores general metrics like total price, miles, and stopovers.

Travel specific options are modeled through dedicated classes: FlightInfo, CruiseInfo, and HotelInfo. These store structured data including pricing, seating or cabin type, room availability, and location. The Booking classes FlightBooking, CruiseBooking, and HotelBooking encapsulate logic for retrieving and reserving these options based on the traveler's itinerary.

The TravellerInfo class acts as the orchestrator, calling specific methods to retrieve flight, hotel, or cruise options. It interacts with the MemberAccount class, which contains member specific data such as loyalty status and frequent flyer affiliations. Validation and reward logic are handled in Membership\_Admin, which includes methods to assign credit points and validate memberships.

Travel\_Agent represents administrative users who can assist with bookings and manage companion travel, completing the application’s real world modeling of a travel planning system.



At the core is the **Trip** class, which represents individual travel offerings. It includes properties such as title, location, price, and description, and includes methods like isValid() and formatPrice() to ensure data integrity and formatting.

The **User** class is used to manage user data for both customers and administrators. It includes typical authentication properties such as username, email, and password, as well as a role to differentiate permissions. Its methods, such as authenticate() and isAdmin(), are used to validate credentials and manage access.

The **AuthService** class handles authentication logic, particularly on the front-end. It includes methods like login(), logout(), getToken(), and isAuthenticated(), which control how sessions and user access are managed in the client-side application.

The **TripService** class is responsible for interacting with the back-end API to manage trip data. It includes methods such as getTrips(), createTrip(trip), updateTrip(id, data), and deleteTrip(id). This class communicates directly with both the **AuthService** and the **APIController**, ensuring that only authorized users can modify trip data.

The **APIController** class is part of the Express back-end and handles incoming HTTP requests related to trips. It includes methods like getAllTrips(req, res), createTrip(req, res), and deleteTrip(req, res), and interacts with both the Trip and User classes to process requests and return responses accordingly.

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

<Exposing RESTful endpoints is a design approach to enable an application to participate in a larger ecosystem. Document each endpoint in the table below, including the HTTP method, purpose, URL, and notes.>

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve list of trips | /api/trips | Returns all available trips in the database |
| **GET** | Retrieve single trip | /api/trips/:tripId | Returns trip details by ID from the request parameter |
| POST | Create new trip | /api/trips | Accepts trip data in request body and adds a new trip to the database |
| PUT | Update existing trip | /api/trips/:tripId | Updates an existing trip using its ID and request body data |
| DELETE | Delete a trip | /api/trips/:tripId | Removes a trip from the database based on the trip ID |

## The User Interface

The Angular project is built using a component-based, modular structure that keeps the code organized and scalable. It separates logic into reusable components, services for business logic and API calls, and modules for grouping features. Key folders include src/app/ for the app’s core structure, assets/ for static files, and environments/ for managing different build settings.

On the backend, the Express app follows the MVC pattern. It organizes routes, controllers, and views into app\_server/routes/, app\_server/controllers/, and app\_server/views/, respectively. Handlebars is used for rendering templates, and static files are served from the public/ folder. While Angular handles client-side rendering and interactivity, Express focuses on server-side logic and routing.

The Angular Single Page Application (SPA) provides a dynamic user experience by updating content in the browser without full page reloads. This makes the app feel faster and more responsive, with features like data binding, form validation, and routing all handled on the client side.

To test the SPA’s connection with the API, tools like Postman or browser dev tools are used to simulate requests. Angular services like getTrips() and updateTrip() are called to verify that data is retrieved or updated correctly. These actions are validated by checking both the UI and database, with support from Angular’s built-in testing tools like HttpClient, Jasmine, and Karma.