

Travlr Getaways Web 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 | Mar. 27, 2023 | Eric Wallace |  |
| 1.1 | Apr. 6, 2023 | Eric Wallace | Preparation for final project |

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

To develop the Travlr Getaways web application, the MEAN stack (MongoDB, Express.js, AngularJS, and Node.js) provides a SPA (single page application) to the client’s customers which will allow them to find and book vacation getaways. The application will also provide the client with the ability to manage data for the getaways, meals, rooms, and news seen on the application.

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

Below is a list of some of the constraints/requirements for the Travlr Getaways web application. This is not a complete list as I am sure there are a few that have been missed.

* Web application
* Authentication
* Authorization
* Time
* Budget
* Browser Compatibility
* Responsive
* Compliance
* SSL/TLS
* User Accounts
* Data management

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

### Component Diagram

The components shown in the diagram below are three groups of components, each group contains other components that are responsible for sharing their services. There are two types of interfaces, provided interfaces and required interfaces. Provided interfaces are providing the service required interfaces need.

* Client component: Requires authentication from the server component and data from the Database component which is passed to the Traveler Portfolio which is required by the web browser and client session components. The traveler portfolio also requires the graphics library.
* Server Component: Provides authentication and a session to the client, it also requires the services of the Database component through the mongoose ODM.
* Database component: Provides its services to the server component on the Mongoose ODM component and for the client component on the Traveler portfolio



**Sequence Diagram**

**Diagram

Description automatically generated**

Here is a description of the flow as shown above:

* The actor has chosen a route, let’s say travel page.
* The router determines that the travel page has been requested and redirects to the page and continues.
* It then calls on the travel controller which sends a request to the HTTP Client to be sent to the API.
* The HTTP Client makes the request to the API, which is intercepted by the router for the API, it determines that trips has been requested and then calls the trips router.
* The trips router calls the trips controller and the getTrips function.
* The controller with the help of the trips model which uses mongoose will query MongoDB for the trips data.
* MongoDB will process the request and through a callback return the applicable data.
* It is returned to the trip’s controller in the form of a JSON response.
* The API then returns the request back to the client which is received by the HTTP Client.
* The HTTP Client returns it to the controller in the form of an object.
* Finally, the controller returns the data to the view at which time the information is rendered to the user in the browser.

## Class Diagram

## The conclusions I came to on the JavaScript classes were two-fold, first I attempted to use the symbols denoting the relationship between the different classes and their methods. If I found it difficult to determine I then used a little critical thinking and common sense to arrive at my decisions.

* MemberAccount: Used by travelers to keep track of membership perks and information.
* Membership\_Admin: Contains 3 methods which through aggregation are used by MemberAccount to change the points of a member account as well as validate points.

Itinerary: Has four methods to which can be used to book a flight, hotel or cruise. It has an association with the cruise, hotel, and flight booking classes. It has a realization with traveler, cruise, flight, and hotel info classes.

* Travel\_Agent: Has one attribute and no methods, and it gets inheritance from the MemberAccount class.
* CruiseBooking: Has one method, getCruise which is passed traveler and cruise info. It has an association with Travel\_Agent and a realization with CruiseInfo.
* FlightBooking: Has one method, getFlight which is passed traveler and flight info. It has an association with Travel\_Agent and a realization with FlightInfo.
* HotelBooking: Has one method, getHotel which is passed traveler and hotel info. It has an association with Travel\_Agent and a realization with HotelInfo.
* HotelInfo: Has no methods and 5 attributes and inherits with TripInfo.
* FlightInfo: Has no methods and 5 attributes and inherits with TripInfo.
* CruiseInfo: Has no methods and 5 attributes and inherits with TripInfo.
* TripInfo: Has no methods and 4 attributes, this is the super class of the three info classes.
* TravelerInfo: Has no methods and 3 attributes, and has an aggregation with hotel, cruise, and flight info.

## [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 all trips | /api/trips/ | Returns all trips |
| **GET** | Retrieve one trip | /api/trips/:tripCode | Returns details of a single trip and is retrieved by supplying the tripCode |
| **POST** | Create one trip | /api/trips/ | Creates a new trip by taking form input and sending it to API server |
| **PUT** | Update one trip | /api/trips/:tripCode | Updates a trip by supplying the tripCode and the record data. |
| **DELETE** | Deletes one trip | /api/trips/:tripcode | Deletes the trip by supplying the tripCode |

## The User Interface

|  |
| --- |
| Graphical user interface, application, Teams  Description automatically generatedLogin Screen |
| Graphical user interface, application  Description automatically generated  Listing Screen |
| Graphical user interface, application  Description automatically generated Add Trip Screen |
| Graphical user interface, application, website  Description automatically generated Add Screen After Trip Addition |
| Graphical user interface, application  Description automatically generated Edit Screen |

An Angular project structure is based more on components where each component has its own directory. Each components serves a specific purpose which contributes to the overall functionality of the application. Separating the project into components makes it easier to manage and change code.

Express on the other hand is based more on functionality, where a service type is thrown into its own directory. This can make managing and changing code a little more difficult because the structure can become cluttered. It is also good to know the two different frameworks serve two different roles. Angular is for front-end development where Express is for back-end development, which can make is difficult to compare the two frameworks. Front-end or client-side application usually have a totally different structure than a backend or server-side application.

Testing the functionality of an API is by using software like Postman where RESTful API calls are made to the server and receiving the correct response back. If the correct response is not received changes are made to the API server and then tested again. The API server will contain methods which receives http request and then acts upon those requests whether it be requesting data from a database, query local data sources or another API that may contain data.

A developer must pay special attention to errors returned during testing to determine which files must be changed. Making changes to the wrong files can lead to additional problems in other areas of the API. Unit testing is another process a developer can use to test the code of an API, while it doesn’t test the connectivity of the server and the sending and receiving of data, use cases will allow a developer to discover vulnerabilities and issues with their code.