

Travel Booking Website

# **Project Software Design Document**

Version 1.2

## Table of Contents

[Project Software Design Document 1](#_Toc1434941580)

[Table of Contents 1](#_Toc1931053771)

[Document Revision History 2](#_Toc2066962960)

[Instructions 2](#_Toc398870858)

[Executive Summary 2](#_Toc240147649)

[Design Constraints 3](#_Toc2146515204)

[System Architecture View 4](#_Toc917307881)

[Component Diagram 4](#_Toc943450007)

[Sequence Diagram 5](#_Toc1832738266)

[Class Diagram 6](#_Toc158671796)

[API Endpoints 8](#_Toc703719369)

[The User Interface 8](#_Toc430663969)

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

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 11/12/20 | Joseph Tindle II | Added Executive Summary, Design Constraints, and System Architecture View |
| 1.1 | 11/21/20 | Joseph Tindle II | Added Sequence Diagram, Class Diagram and API Endpoints |
| 1.2 | 12/9/20 | Joseph Tindle II | Added User Interface, screenshots to User Interface, and updated API Endpoints |

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

Travlr Getaways wants to develop a travel booking website for their customers to book travel packages. Their customers must be able to create an account, search for travel packages by location and price point, and book reservations with the travel agency. Customers must also be able to visit the website regularly before their trip to see their itineraries. Lastly, Travlr Getaways would like an admin-only site where their administrators are able to maintain a customer base, available trip packages, and pricing for each item and package. The marketing department at Travlr Getaways has provided the specs and a wireframe to guide the development of the website. Our goal is to produce a fully functional travel web application that meets Travlr Getaways’ requirements.

To fulfill these requirements, we have begun using the single-page web app architecture of a full stack web application that uses the MEAN stack. MEAN stands for the essential full stack tools: MongoDB, Express, Angular, and Node.js. We will first need to map out the architectural components, set up the development environment, and customize the customer-facing web page to align to the wireframe provided and Travlr Getaways’ vision. This includes completing the initial setup of the Node.js server and the Express framework. The application will be a static customer-facing (frontend) website using HTML, CSS, and JavaScript elements that each provide the visual aesthetics of the application. This includes the layout, style, design, and functionality required to meet the clients’ needs.

To facilitate and optimize the application's building, we will complete the client single-page application (SPA) using the Angular framework to extend HTML with HTML attributes called directives. We chose the Angular web application framework because it offers many fully integrated tools to facilitate the build. An example would be the Angular Command Line Interface (CLI) that can be used to build the components and services for the client-facing frontend. One of the most important steps throughout the process will be the addition of security features to protect the data of Travlr Getaways and their customers. In addition, Travlr Getaways wants to add a layer of security that applies to server-side applications to produce web tokens for web-login authentication. Once complete, a final testing on the full stack web application with the API to ensure the server is returning the data properly will be required.

To code the backend of the application, we will need to create and configure a NoSQL database with data models and a schema for data files and storage. Using the MongoDB program, we will be able to store travel booking trips and create JSON files containing initial data for seeding the database to enable testing of the RESTful API routes. Next, we will populate the database and be able to view the seeded collections and documents within. Lastly, we will need to wire the database to the server, test the RESTful API, and refactor the code to work successfully with the frontend.

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

The development and integration of the MEAN stack of software applications into a customer-facing and administrator single-page application could be a major constraint. Specific skills and a technical understanding related to web development and management, the interaction between its components, the development environment, and the applications’ optimization will be required. As the software stack uses an open-source framework, the source code is readily available to view and edit for developmental purposes. As we are using the MEAN stack, knowledge and proficiency in developing applications using JavaScript and exchanging data with JSON pose technical constraints. The addition of a layer of security that applies to server-side applications will also pose a technical constraint, mostly when producing web tokens for web-login authentication.

A web server is essentially a system on which a website runs; It works by sharing a website’s information with any browser that requests it and is permitted access. Web servers hold a lot of private information, such as usernames and passwords, that will be stored in the database. This would be a technical constraint when storing and protecting the database and its interactions through the RESTful API as a specific set of skills and knowledge are required to develop secure communications between services. Several solutions exist to mitigate the vulnerability of the application, each providing a different design constraint. Such solutions include performing consistent website backups, hardening the wp-config.php file, and disabling the XML-RPC protocol. Backups enable developers to save their website’s latest data in the event of a break-in or file corruption. Hardening certain PHP (Hypertext Preprocessor) files can protect the database from SQL Injection attacks, while disabling the XML-RPC protocol can prevent DoS (Denial of Service) Attacks.

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

### Component Diagram



There are three major components in the overall system that make up its functionality: The Client, the Server and the Database. The Client component has four main components that make up its functionality and interactions within the overall system. These four components are the Client Session, the Web Browser, the Traveler Portfolio, and the Graphic Library. Beginning with the Client Session that requires information from the Web Browser and the Traveler Portfolio components, this component will be the user generating a request by typing in Travlr Getaways’ web link (URL) in their browser’s address bar. The Web Browser will receive the user’s request, determine the site’s location, and then request access from the Server.

Within the diagram, there are ports that connect the Client and Server components that specify a separate interaction point between the component (Client) and the environment (Server). The Client Session requires information from the Authentication Server and the Server Session to perform its required functions. Through this interaction, the Server sends the requested data to the Web Browser. The Web Browser then translates the information it receives and displays it to the user as a website. While viewing the website, the user will be able to login to their account through the Client Sessions’ connection with the Authentication Server, as part of their Server Session.

Both the Web Browser and the Graphic Library require information from the Traveler Portfolio to display images of travel destinations and itineraries to the user while logged in during their Client Session. The second major component, the Database, comprises the MongoDB cross-platform document-oriented database program. This program will store travel booking trips as it provides the information needed by the Traveler Portfolio to perform its required functions. The final major component, the Server, contains a component called the Mongoose ODM (Object Data Modeling), which requires information from MongoDB to perform its required functions. The Mongoose ODM component is a library for MongoDB and Node.js.

It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB. This library will be used to store the data in the Server side of the application. The Server Session requires the information from Mongoose ODM and the Traveler Database to display all the working features of the application. These features include allowing customers to create accounts, search for travel packages by location and price point, book travel packages by making reservations with the travel agency and see their itineraries before each trip.

### Sequence Diagram

A picture containing diagram

Description automatically generated

The Sequence Diagram above displays a high-level overview of the request/response data flow between the Client, Server and Database components depicted in the Component Diagram section. In this diagram, the user generates a request by typing in Travlr Getaways’ web link (URL) in their browser’s address bar. The application then begins with the Client-Side AngularJS front-end web framework, where the request is seen as the Route redirecting the user to the Browser/View/Template object. Upon receipt, the Browser/View/Template interacts with the Controller which calls the HTTP Client service factory to retrieve the requested data. From here, the Client-Side and Server-Side meet when the HTTP Client sends the request to the controller function in the Route object. This is where AngularJS interacts with Node.js and Express on the Server-Side through the HTTP-based RESTful API, where the Browser/View/Template requests the URL from the Server.

On the Server-Side, the Route redirects the request to the Controller/Model function, which in turn abstracts the request through the Mongoose ODM to call MongoDB. In other words, Node.js with Express uses the component library Mongoose ODM to call MongoDB to process the request. Should the request prove successful, MongoDB sends a response with the JSON data payload back to the Controller/Model component. It is through the interaction between the Controller/Model, HTTP Client and Client-Side Controller that the Server sends the requested data to the Web Browser. The Controller is the component that translates the information it receives to display the website to the user. While viewing the website, the user can perform processes such as signing in, creating an account, searching for travel packages, or booking travel destinations.

## Class Diagram

Diagram, schematic

Description automatically generated

The Class Diagram above displays the different classes, their attributes and their methods that each play a role in the make-up of the travel booking software application. The first class, the Membership Admin, is made up of an aggregation of Member Accounts. This class is based on the admin of a specific membership and has the methods to credit points, get points, and validate the membership status of a customer using their itinerary, member number and frequent airline choice. There can be zero-to-many Membership Admins associated with a Travel Agent. The Member Account class has attributes like member number, frequent airline, and member status to represent a traveler's account.

The Traveler Info class inherits the attributes assigned to their Member Account to represent a customer logged into their account. The Travel Agent class represents a travel agent booking a trip for a customer, shown in its Book Package and Book Flight methods that implement the information contained within classes such as Hotel Info and Cruise Booking. The Trip Info class is an aggregation of the Cruise Info, Flight Info, and Hotel Info classes, where it contains attributes such as total price and total miles for the entirety of a trip. The three classes: Cruise Info, Flight Info, and Hotel Info, are very important classes as they make up the information a customer will look at when deciding on which trip to book. The Cruise Info class, for instance, has attributes that give the customer the name of the cruise, the cabin type in which they will reside, and the price of the cabin.

Likewise, the Flight Info and Hotel Info classes contain information about the flight or hotel a customer may wish to book. These three classes each inherit the attributes of the Itinerary class, where the customer can view their start and end date, and the origin and destination information regarding their trip. The final three classes deal with the booking of a trip. These classes are Hotel Booking, Flight Booking, and Cruise Booking. The purpose of these three classes is to implement the methods that book a hotel, flight or cruise, respectively. They use the information given in the Traveler Info, Hotel Info, Flight Info and Cruise Info classes.

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

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieves a list of all trips | http://localhost:4200/list-trips | Returns a list of all trips based on the defined trip schema. |
| **GET** | Retrieves a single trip | http://localhost:3000/api/trips/:tripCode | Returns a single trip instance based on the defined trip schema. Instance is identified by the trip code ID passed on the request URL. |
| **POST** | Allows a user to login | http://localhost:3000/api/login | Allows a user to login to their account. |
| **POST** | Creates a single trip | http://localhost:4200/add-trip | Adds a new trip and saves it to the trip listing. |
| **PUT** | Updates a single trip | http://localhost:4200/edit-trip | Edits an existing trip and saves it to the trip listing. |

## The User Interface

Graphical user interface, website

Description automatically generated

Graphical user interface, application, Teams

Description automatically generated

Graphical user interface, application

Description automatically generated

The Express HTML customer-facing web page utilized static HTML files to render site content to the web browser. Its project structure consisted of the MVC architecture with folders designated for different aspects of the application. Using MVC routers to refactor the code into Handlebar template views, content such as the homepage and traveler page were converted and then rendered as JSON data. This simplified the project's code and allowed the page to be rendered and driven dynamically by the data passed into the template. From there, the view was linked to new API endpoints to pull data from the database application MongoDB, instead of the static JSON file. It was by using routers that requests from the client to the server were made. These steps culminated in the rendering of a list of trips on the traveler page, where the controllers rendered the page, the views specified the content seen, and the model defined the schema of the data to be displayed.

In contrast, the Angular project structure utilized the creation of "components," or building blocks, that have properties such as selector, template and style which specify the meta data required to process the component. These components were a folder with three main files (HTML, CSS and TypeScript) that were the MVC architecture. Each of these properties specified the module that was bootstrapped to render in the web browser (selector), the template that contained the data to be displayed (template), and the style or appearance of that data respectively (style).

To test the SPA to ensure it is working with the API to GET and PUT data in the database, a developer can use the browser's Developer Tools functionality to see the requests being made and by which module in the application. You can also visually inspect your web application's functionality in a web browser by seeing the default content you added to the component property in the router module. Some of the errors I ran into included the update page not working and the trip listing page leaving out data that was expected to be displayed. I debugged the issues down to typing in the wrong variable in a method and missing a defined variable in the form initialization logic.