

Travlr Getaways Booking

# **CS 465 Project Software Design Document**

Version 1.1

## Table of Contents

[**CS 465 Project Software Design Document** 1](#_Toc172380687)

[Table of Contents 2](#_Toc172380688)

[Document Revision History 2](#_Toc172380689)

[Instructions 2](#_Toc172380690)

[Executive Summary 3](#_Toc172380691)

[Design Constraints 4](#_Toc172380692)

[System Architecture View 5](#_Toc172380693)

[Component Diagram 5](#_Toc172380694)

[Sequence Diagram 7](#_Toc172380695)

[Class Diagram 8](#_Toc172380696)

[API Endpoints 11](#_Toc172380697)

[The User Interface 11](#_Toc172380698)

[References 12](#_Toc172380699)

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

| Version | Date | Author | Comments |
| --- | --- | --- | --- |
| 1.0 | 07/21/04 | Justin Starr | Updated Executive Summary, Design Constraints, and System Architecture View |
| 1.1 | 8/4/2024 | Justin Starr | Updated Document Version, Sequence Diagram, Class Diagram, and API Endpoints |

## 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)

The architecture of the web application that will be developed for the client Travelr Getaways is a full-stack web application consisting of a customer-facing website, a database, and an administrative single-page application (SPA) that will all work together. To achieve this, a combination of technologies, called a MEAN stack will be implemented to create a database and web server that will operate on the back end, contain application logic and control in the middle of the stack, and will go all the way to the front end with a user interface (Harber & Holmes, 2019).

The MEAN stack is a complete JavaScript stack that is comprised of four main technologies, allowing for the creation and deployment of dynamic web applications. The M in MEAN stands for MongoDB, which is the database, the E stands for Express, which is the web framework, the A stands for Angular, which is used for the front-end framework, and the N stands for Node.js, which is used for the web server.

MongoDB is a NoSQL database that allows the storing of data in JSON-like documents, making it more flexible, and it provides easy scalability for web applications as well as high performance (Eddie, 2022).

Express is the server framework that can serve static HTML content and support using JavaScript to code the necessary logic and behaviors required to meet the client requirements. It is a lightweight framework that provides an easy-to-use interface, which connects NodeJS with MongoDB, allowing you to use RESTful routes that directly connect HTML content to data in the application's database (Eddie, 2022).

AngularJS is used to develop single-page applications, which in the case of this application will be used for the administrative page. Typically, a MEAN stack application’s architecture includes a representational state transfer (REST) API that feeds the single-page application (Harber & Holmes, 2019). It essentially creates a stateless interface to the database that enables applications, such as Angular SPA to work with the data, and it uses JSON throughout the stack, including in the database (Harber & Holmes, 2019).

NodeJS is a platform that is event-driven and it is a non-blocking input/output model, which keeps it lightweight and efficient (Eddie, 2022). This means that it can handle multiple requests at the same time without blocking other code execution (Ramos, 2023).

Express and Node.js work in conjunction with each other to deliver HTML content directly from the server to the client-facing side of the application, which can also be done using templates, such as handlebars. The administrative single-page application will be implemented with Angular on the front end. Using a REST API that is built using Express and Node.js, content can be routed to both the client-facing side of the application and the Angular single-page application. Changes made from the administrative side can be seen on the customer-facing side almost immediately after making any changes.

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

It is essential to consider that while developing the Travelr Getaways Booking application there are design constraints. For example, JavaScript applications are difficult to crawl and index, and search engines, while they look at HTML content on a page, they do not execute or download much JavaScript (Harber & Holmes, 2019). Therefore, it can not be determined precisely how much of the content will be indexed if all of the content is served via a JavaScript application (Harber & Holmes, 2019). A similar issue exists is that automatic previews from social-sharing sites do not work well, similarly because they look at the HTML of the page you are trying to link, where the social-sharing sites attempt to extract some relevant text and images. Just like the search engine problem, they do not run JavaScript on the page, and therefore any JavaScript content that is served up will not be seen (Harber & Holmes, 2019). There are workarounds for these problems, however, they require a large amount of effort, and maintaining them can become problematic (Harber & Holmes, 2019). These concerns are significant because the type of application that is being built would benefit from the exposure that can be gained from social media sharing and search engine traffic. The more people that can either find the web application or hear about it from other people, the better the business will likely grow.

Additionally, another consideration is that SPAs have a slower first-page load than server-based applications. (Harber & Holmes, 2019). This is because SPAs have to bring down the framework and application code before the required view can be rendered in the browser as HTML, whereas server-based applications only have to push the required HTML (Harber & Holmes, 2019). However, based on the size of the client’s application that is being built, this for the time being, should not be much of a problem.

Another consideration of the design is the database itself. Earlier versions of MongoDB do not support ACID transactions, which could result in the loss of data if while processing multiple requests, the application becomes unstable (Eddie, 2022). However, MongoDB version 4.0 began supporting ACID transactions which help to ensure database transactions are processed in a reliable way.

Lastly, implementing a MEAN stack architecture can have drawbacks such as performance issues. Node.js executes JavaScript code one task at a time, unlike how it handles other tasks. While this single-threaded approach does have its advantages, it could potentially lead to bottlenecking during CPU-intensive tasks (Ramos, 2023).

## [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).

As seen in the above component diagram, we have three main components, which are the Client, Server, and Database components. Within these three components contain sub-components that are responsible for one clear aim, and on a need-to-know basis, only interact with other essential elements (Visual Paradigm, 2024). Each component can be recognized by the component icon located in the top right corner of every component rectangle.

The colored circles with half circles around them represent interfaces that indicate that they are either required by one component or provided by a component. The direction for which that half circle around the interface indicates that it is pointing to the required interface, and the other component that the interface connects to is the provided component. Therefore, we can see that the Client component requires the Server component, and therefore, the Server component is provided to the Client component. Also, we see that both the Client and Server components require the Database component, and the Database component is provided to both the Client and Server components.

Both the Client and Server components contain a port, represented using a square placed on the side or edge of both components. They are used to “help expose required and provided interfaces of a component” (Visual Paradigm, 2024).

Lines are used to connect components to interfaces that then connect to another component. Because these types of lines do not have arrows on either end of them, this means that they demonstrate an association between connected interfaces/components. By understanding what components are associated with other components, and knowing whether or not a component is required of or provided to another component, we gain a better understanding of the system and the relationship between components.

Beginning with the Client component, there is a Traveler Portfolio component that is required by the Web Browser component, which is the user interface. Both of these components are required by the Client Session component. The Graphics Library component requires the Traveler Portfolio component, and the Traveler Portfolio component requires the Database component.

The Client Component requires the Server component, however it is not the case that the Server component requires the Client component. A user is able to be directed to the Authentication Server component, where if authenticated, a Server Session is established.

The Server Session Component allows the authenticated user to manage the Traveler Database and update the Database component. The Server Session component requires the Traveler Database and Mongoose ODM Component, and the Mongoose ODM component requires the MongoDB component. The Mongoose ODM (Object Data Modeling) 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” (Hossen, 2022).

### 

### Sequence Diagram

A diagram of a company

Description automatically generated with medium confidence

First, we see that the front-end (Client-Side) is built using Angular JS, and the back-end (Server-Side) is built using Node JS, Express, and Mongoose. The database also exists on the server-side, which is implemented and maintained through MongoDB.

The process or sequence begins with the actor who is the user. The user navigates to a specific path in their web-browser where the application checks that the path is valid. The route redirects the user to the appropriate view for the specified route, it gathers the view data, where it then interacts with the appropriate controller. The controller makes a call to retrieve the data for the route. If the route has an api endpoint, then the HHTP client sends a request to the server controller which it is routed to by the server’s routes. The server's routes find the appropriate controller for the specified api endpoint, where the request is then abstracted through the mongoose ODM (Object Data Modeling). This allows us to work with the data in the application using JavaScript, and it provides easy communication between the application and the database. The request is processed and returns the requested data specific to each endpoint. One endpoint might be requesting all documents, or another may request a specific document. The callback function is then called and response is returned to the HTTP Client in JSON format, which allows the data to be easily read. The controller on the client-side is awaiting the request to be returned. Once the controller has received the results object, it assigns it to the scope and renders the appropriate view in the browser.

One example of a process that can be referenced with the above sequence diagram is the process for trips interaction. The following is a description of this interaction based on the sequence diagram.

For the trips interaction, a user could navigate to the trips endpoint by typing in the address of the api endpoint “localhost:3000/api/trips”. The request begins by being routed to the apiRouter where it will then interact with the tripsController. Based on the specific “/trips” route, the tripsController calls the appropriate function, in this case tripsList, to make a request for all records from the trips collection in the travlr database located on the server. This request is abstracted through the Mongoose ODM, the database processes the request, and the callback() function from the controller waits for the request to be processed. The response is received and the data is sent back to the client. Then, the results object is processed by the controller and returned to be displayed in the browser.

## Class Diagram

A diagram of a travel company

Description automatically generated  
First, it is essential to have an understanding of what the different symbols and arrows mean. An arrow with a closed head that is unshaded means that one or more classes inherit from the class that the arrow is pointing to. A line with an unshaded diamond at the end of it means that the class with the diamond end is an aggregate class. This means that it specifies a whole and its parts, and the class with diamond can exist without the class or classes that it has a relationship with. Lines with no arrows indicate that the two classes have an association, and lines with a solid arrow indicate that one class is dependent upon another.

## In the above class diagram, we see we have 12 JavaScript classes, and they are as follows:

TripInfo class:

In the TripInfo class, we have four public attributes, and this class is the parent class for CruiseInfo, FlightInfo, and the HotelInfo classes. When any of these three class are instantiated, they will inherit the four attributes from the TripInfo class. This class has no methods.

CruiseInfo class:

CruiseInfo has three public attributes and inherits from Tripinfo. It has an aggregate relationship with the TravelerInfo class, meaning TravellerInfo can exist outside or without the CruiseInfo class. We also see that the Itinerary and CruiseBooking classes both point to the CruiseInfo class. This means that both of those classes are aware of the CruiseInfo class and interact with it. This class has no methods.

FlightInfo class:

This class is similar to the Cruiseinfo class. It has three public attributes and inherits from the TripInfo class. TravellerInfo is also an aggregate class of the FlightInfo class, meaning it can exist without or outside the FlightInfo class. The itinerary and FlightBooking classes also are aware of the FlightInfo class and interact with it. This class has no methods.

HotelInfo class:

The HotelInfo class is similar to the previous two classes, however this class has five public attributes, and it also inherits from TripInfo. TravellerInfo is an aggregate class of HotelInfo and can exist outside or without the Hotelinfo class. Both the Itinerary and HotelBooking class are aware of the HotelInfo class and interact with it. This class has no methods.

HotelBooking class:

The HotelBooking class has no class attributes and does not inherit from any other classes. The Itinerary class has a zero to many relationship with the Hotelbooking class. Also, the HotelBooking class is aware of the Travel\_Agent class and interacts with it, and also has a zero to many relationship with it. HotelBooking has one public method. The method getHotel is public and interacts with the HotelInfo class.

FlightBooking class:

The FlightBooking class is similar to the HotelBooking class. It has no class attributes and does not inherit from any other class. The Itinerary class has a zero-to-many relationship with the FlightBooking class. The FlightBooking class is aware of the Travel\_Agent class and interacts with it, and it also has a zero to many relationship with it. FlightBooking has one public method. This method, getFlight is aware of the FlightInfo class and interacts with it.

CruiseBooking class:

This class is also similar to the HotelBooking and FlightBooking class. It has no attributes and does not inherit from any other class. The Itinerary class has a zero-to-many relationship with the CruiseBooking class. The CruiseBooking class is aware of the Travel\_Agent class and interacts with it, and it also has a zero to many relationship with it. CruiseBooking has one public method. This method, getCruise is aware of the Cruiseinfo class and interacts with it.

Itinerary class:

The Itinerary class has no attributes, and it does not inherit from any other classes. It has zero to many association with the Membership\_Admin class. It is aware of and interacts the CruiseInfo, FlightInfo, and HotelInfo classes. It also has a zero to many association with the HotelBooking, FlightBooking, and CruiseBooking classes. The Itinerary class has four public methods: BookPackage, BookFlight, BookHotel, and BookCruise. BookFlight interacts with FlightInfo, BookHotel interacts with Hotelinfo, and BookCruise interacts with Cruiseinfo.

TravelerInfo class:

The TravellerInfo class has three public attributes and does not inherit from any other classes. It is aggregate to the CruiseInfo, FlightInfo, and Hotelinfo classes, meaning that it can exists outside or without any of these three classes. This class contains no methods.

MemberAccount class:

The MemberAccount class has four public attributes, and does not inherit from any other class, however the Travel\_Agent class inherits from this class. The Membership\_Admin class is aggregate to this class, meaning it can exist without this MemberAccount class. The MemeberAccount class has no methods.

Membership\_Admin class:

The Membership\_Admin class has no attributes and does not inherit from any other classes. This class is aggregate to the MemberAccount class, meaning it can exist without or outside of the MemberAccount class. It has a one to many relationship with the Itinerary class, and it contains three methods, creditpoints, getpoints, and validate.

Travel\_Agent class:

The Travel\_Agent class has one public attribute, and it inherits from the MemberAccount class. The Itinerary, HotelBooking, FlightBooking, and CruiseBooking classes have a zero to many relationship with this class, and all four classes are aware of and interact with the travel\_Agent class.

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

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve list of trips | /api/trips | Returns all active trips |
| **GET** | Retrieve single trip | /api/trips/:tripCode | Returns single trip instance, identified by the tripCode passed on the request URL |

## The User Interface

<Insert screenshots from the development of the SPA development to show the following: (1) a unique trip, added by you, (2) the Edit screen, and (3) the Update screen.>

<Summarize the Angular project structure and how it compares to the Express project structure. Be sure to describe the rich functionality provided by the SPA compared to a simple web application interaction. Describe the process of testing to make sure the SPA is working with the API to GET and PUT data in the database.>

## References

Eddie, J. (2022, January 7). *Why MEAN Stack is Best for Web*

*Development?* Medium. <https://enlear.academy/why-mean-stack-is-best-for-web-development-55b7e4a00d72>

Harber, C., & Holmes, S. (2019). *Getting MEAN with Mongo, Express, Angular, and Node, Second*

*Edition* (2nd ed.). Manning Publications.

<https://learning.oreilly.com/library/view/getting-mean-with/9781617294754/>

Hossen, I. (2022, March 4). *Mongoose*. DEV Community.

<https://dev.to/iftakher_hossen/introduction-to-mongoose-409g>

Ramos, M. (2023, September 11). *MEAN Stack Explained*. Kinsta.

<https://kinsta.com/blog/mean-stack/>

Visual Paradigm. (2024). *What is Component Diagram?* [https://www.visual-](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-component-diagram/)

[paradigm.com/guide/uml-unified-modeling-language/what-is-component-diagram/](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-component-diagram/)