

# **CS 465 Project Software Design Document**

Version 2.0

## 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 | 8/6/24 | Brian Chmura | First revisions |
| 2.0 | 8/16/24 | Brian Chmura | Major Update |

## 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 Travlr Getaways website will be developed using the MEAN stack, which stands for MongoDB, Express.js, Angular.js, and Node.js. These technologies work together to create an efficient website, enabling smooth handling of JSON data and ensuring timely completion of the project. The MEAN stack's architecture is known for its efficiency.

Angular.js will manage the web side of the architecture, providing elegant designs for web pages. Node.js, utilizing the Express.js framework, will handle the server side. MongoDB will manage the database side, allowing for quick retrieval of information and ensuring fast website performance. Specifically, Angular.js will request data from Node.js, which in turn retrieves data from MongoDB via Express.js, and sends it back to Angular.js for display.

For users, the website will showcase available trips and enable bookings. To enhance user experience, especially for those sensitive to light, a dark mode feature will be implemented. Although the Multi-Page Application (MPA) built with Express.js and Handlebars might take a few seconds to load, it will deliver fast performance once loaded.

The administrator side will feature a Single Page Application (SPA) built with Angular.js. This design will consolidate all website information into a single page, making it easier for administrators to edit content and manage posts. Changes made will be instantly available to all users, streamlining the administrative process.

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

The website will face certain limitations, both creatively and technically:

**MongoDB:**

* The maximum size for a BSON document is 16 megabytes to prevent excessive RAM usage.
* The nesting depth of BSON documents is limited to 100 levels, counting each object or array as a level. Hitting this limit likely indicates overly complex data structures.
* Database names are case-insensitive, and it is standard practice to use lowercase names without special characters, with a maximum length of 64 characters.
* Field names must be unique within a document.

**Express.js:**

* Express.js will be used to handle server-side rendering of the website, crucial for delivering content efficiently to users.

**Angular.js:**

* Angular.js will be employed to develop the user interface for both administrators and end users. Additionally, Handlebars will be needed to integrate with JavaScript and deliver dynamic content.

**Node.js:**

* Node.js will ensure scalability, allowing the website to handle a large number of users seamlessly, especially when used in conjunction with MongoDB.

These limitations need to be considered during the development process to ensure optimal functionality and performance of the Travlr Getaways website.

## [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).

**Client Component:**

* **Sub-components:** Client Session, Graphic Library, Traveler Portfolio, Web Browser.
* The process initiates with the user launching a Web Browser, which opens a Client Session. The Client Session communicates with the Authentication Server to verify that the user is not already logged in through another connection. Upon successful authentication, the Traveler Portfolio becomes available, allowing the user to access all website information. The Traveler Portfolio also activates the Graphic Library to enhance the visual aspects of the main website.

**Database Component:**

* **Sub-component:** MongoDB.
* After the website is accessed via the Traveler Portfolio, MongoDB ensures the website reflects the latest data from the database. This synchronization is crucial for the proper functioning of the website.

**Server Component:**

* **Sub-components:** Authentication Server, Mongoose ODM, Server Session, Traveler Database.
* MongoDB works in tandem with Mongoose ODM, which facilitates efficient schema management and accelerates coding processes. The Server Session relies on Mongoose ODM for proper functionality. It validates user credentials against the database and enables user or admin logins. The Authentication Server verifies the user's legitimacy and communicates this information back to the Client Session, completing the cycle.

This system leverages the MEAN stack architecture, using Handlebars for the website's user interface, ensuring a robust and efficient web application.

### Sequence Diagram

A diagram of a trade end

Description automatically generated

The process begins with an actor (you) who accesses a computer and navigates to `https://(websitelink).com/(website-page)`. Upon reaching this site, the browser loads a client-side view/template, displaying the relevant webpage information to the actor. When the user interacts with elements containing anchor tags, a controller responsible for managing page navigation activates an HTTP Client. This HTTP Client functions as an intermediary between the client-side and server-side, sending a request to the API controller for the necessary page information. The API controller processes the request and retrieves the required data from a MongoDB database. The database then returns the requested information, which is displayed to the user on the webpage. The system remains ready to handle and display any new requests for information as they come in.

## Class Diagram

A diagram of a travel geysers class diagram

Description automatically generated

Here is the detailed description of the 12 classes based on the provided information:

1. **Membership\_Admin**
   * Aggregates: MemberAccount
   * Has: One or more Itinerary
   * Methods:
     + public boolean creditpoints(): Returns an Itinerary.
     + public int getpoints(): Returns an int (membernum) and a string (frequent\_airline).
     + public boolean validate(): Returns an int (membernum) and a string (frequent\_airline).
2. **MemberAccount**
   * Inherited by: TravelAgent
   * Aggregated by: Membership\_Admin
   * Variables:
     + public int membernumber
     + public string frequent\_airline
     + public int memberstatus
     + public string memberclub
3. **Travel\_Agent**
   * Inherits: MemberAccount
   * Realized by: Itinerary, HotelBooking, FlightBooking, CruiseBooking
   * Variables:
     + public int companionnum
4. **Itinerary**
   * Realizes: Travel\_Agent
   * Associated with: Membership\_Admin (must be 1 or more)
   * Associations: CruiseInfo, FlightInfo, HotelInfo, HotelBooking, FlightBooking, CruiseBooking
   * Methods:
     + public Itinerary BookPackage(): Returns an Itinerary.
     + public FlightInfo BookFlight(): Returns an Itinerary.
     + public HotelInfo BookHotel(): Returns an Itinerary.
     + public CruiseInfo BookCruise(): Returns an Itinerary.
5. **CruiseBooking**
   * Realizes: Travel\_Agent and CruiseInfo
   * Associated with: Itinerary (0 to many of either class, CruiseBooking or Itinerary)
   * Methods:
     + public CruiseInfo getCruise(): Returns TravelerInfo and CruiseInfo.
6. **FlightBooking**
   * Realizes: Travel\_Agent and FlightInfo
   * Associated with: Itinerary (0 to many of either class, FlightBooking or Itinerary)
   * Methods:
     + public FlightInfo getFlight(): Returns TravelerInfo and FlightInfo.
7. **HotelBooking**
   * Realizes: Travel\_Agent and HotelInfo
   * Associated with: Itinerary (0 to many of either class, HotelBooking or Itinerary)
   * Methods:
     + public HotelInfo getHotel(): Returns TravelerInfo and HotelInfo.
8. **CruiseInfo**
   * Aggregated by: TravellerInfo
   * Inherits: TripInfo
   * Realized by: Itinerary and CruiseBooking
   * Variables:
     + public string name
     + public string cabintype
     + public float price
9. **FlightInfo**
   * Aggregated by: TravellerInfo
   * Inherits: TripInfo
   * Realized by: Itinerary and FlightBooking
   * Variables:
     + public string name
     + public string seatclass
     + public float price
10. **HotelInfo**
    * Aggregated by: TravellerInfo
    * Inherits: TripInfo
    * Realized by: Itinerary and HotelBooking
    * Variables:
      + public string name
      + public int star
      + public string location
      + public int roomsrequested
      + public float price
11. **TripInfo**
    * Inherited by: CruiseInfo, FlightInfo, HotelInfo
    * Variables:
      + public int starting\_date
      + public int returning\_date
      + public string origin
      + public string destination
12. **TravellerInfo**
    * Aggregates: CruiseInfo, FlightInfo, HotelInfo
    * Variables:
      + public float totalprice
      + public int totalmiles
      + public string stopover

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

| **Method** | **Purpose** | **URL** | **Notes** |
| --- | --- | --- | --- |
| **GET** | Retrieve list of blogs | /api/blogs | Returns all active blogs |
| **GET** | Retrieve list of ‘latest’ posts | /api/latest | Return all active ‘latest’ posts |
| **GET** | Retrieve list of meals | /api/meals | Returns all active meals |
| **GET** | Retrieve list of news | /api/news | Returns all active news |
| **GET** | Retrieve list of rooms | /api/rooms | Returns all active rooms |
| **GET** | Retrieve list of Testimonies | /api/testimonials | Return active testimonies |
| **GET** | Retrieve list of trips | /api/trips | Returns all active trips |
| **GET** | Retrieve Single Blog | /api/blogs/:blogCode | Returns single blog post based on title passed to request |
| **GET** | Retrieve single ‘latest’ post | /api/lastest/:latestCode | Returns the single latest post passes to the request |
| **GET** | Retrieve meal | /api/meals/:mealCode | Returns one meal called on by mealName |
| **GET** | Retrieve one news post | /api/news/:newsCode | Returns single post, called by posterName |
| **GET** | Retrieve single room | /api/rooms/:roomCode | Returns single room by name |
| **GET** | Retrieve single testimony | /api/testimonials/:testimonalCode | Returns single testimonial by person |
| **GET** | Retrieve single trip | /api/trips/:tripcode | Returns a single trip |
| **POST** | Create single blog | /api/blogs/ | Creates a single blog post |
| **POST** | Create single latest post | /api/latest/ | Creates latest post |
| **POST** | Create a meal | /api/meals/ | Creates a single meal |
| **POST** | Create a news post | /api/news/ | Create a single post |
| **POST** | Create a room | /api/rooms/ | Creates single room |
| **POST** | Create a testimonial | /api/testimonials/ | Creates a testimonial instance |
| **POST** | Create a trip | /api/trips/ | Creates a trip instance |
| **PUT** | Update latest post | /api/latest/:latestCode | Updates latest post identified by the title |
| **PUT** | Update blog | /api/blogs/:blogCode | Updates blog post identified by title |
| **PUT** | Updates meal | /api/meals/:mealCode | Updates meal by mealName |
| **PUT** | Update news post | /api/news/:newsCode | Updates a poster by posterName |
| **PUT** | Update room | /api/news/:roomCode | Updates single room |
| **PUT** | Update one testimonial | /api/testimonials/:testimonialCode | Updates a testimonial by person |
| **PUT** | Update one trip | /api/trips/:tripcode | Updates a single trip |
| **DELETE** | Delete single blog | /api/blogs/:blogCode | Deletes a blog by title |
| **DELETE** | Delete latest post | /api/latest/:latestCode | Deletes latest post identified by title |
| **DELETE** | Delete meal | /api/meals/:mealCode | Deletes a meal by mealName |
| **DELETE** | Delete news post | /api/news/:newsCode | Deletes a post by the posterName |
| **DELETE** | Delete one room | /api/rooms/:roomCode | Deletes a single room instance |
| **DELETE** | Delete a testimonial | /api/testimonials/:testimonialCode | Deletes a testimonial identified by the person |
| **DELETE** | Delete Trip | /api/trips/:tripcode | Deletes a trip by the code request |

## The User Interface

A screenshot of a website

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Angular differs significantly in appearance from Express's HTML customer-facing pages due to their distinct purposes. Angular powers a Single-Page Application (SPA), whereas the Express HTML pages create a Multi-Page Application (MPA). In an MPA, the content is delivered using Express, HTML, and Handlebars, which together render the pages that users interact with. Conversely, the SPA relies on Angular and Cross-Origin Resource Sharing (CORS). CORS allows a server to permit resource sharing across different origins, enhancing flexibility in resource loading.

These frameworks combine to ensure that the webpage efficiently displays only the travel packages while allowing administrators to add or delete trips without affecting other parts of the site. Angular follows the Model-View-ViewModel (MVVM) architecture, where data binding facilitates communication between the View and ViewModel, with the model managing logic and data asynchronously. Meanwhile, Express uses the Model-View-Controller (MVC) architecture, wherein the controller handles user inputs and manipulates the model, which in turn updates the view.

SPAs offer advantages like client-side rendering, dynamic content, a modular design, and improved mobile experiences. These features enhance user engagement and simplify the developer's task of modifying parts of the SPA. However, SPAs also have some drawbacks, such as longer initial loading times, potential challenges with Search Engine Optimization (SEO), and occasional compatibility issues with older browsers.

Testing the SPA's database is straightforward with Postman, a popular testing tool. Start by creating a new collection and a new request within Postman. Input the API endpoint, such as http://localhost:3000/api/trips, in the ribbon at the top of the window. Using the dropdown menu to the left, select GET to retrieve all current database information. To add a new trip, select POST, and provide trip details in the request's body. Similarly, to update an existing trip, choose PUT and adjust the API endpoint to the specific trip, for example, http://localhost:3000/api/trips/TEST2. Modify the request body with updated data, and the trip will be changed accordingly.

Once you run these operations, you can verify updates by performing another GET request. Be aware of potential errors, such as authentication issues, data validation errors, or network and syntax problems. These errors often stem from incorrect formatting but can be easily fixed with proper adjustments.