STePS Networking Module

Developer's Guide

Designed by:

Darryl Lai Zhin Hou	A0122534R
Eric Ewe Yow Choong	A0112204E
Koh Ling Ling Jean	A0126125R
Muhammad Adam	A0121813U
Tay Siang Meng Sean	A0121409R

Table of Contents

CHAPTER
PAGE

1. Introduction

1.1 What is STePs Networking Module?

SoC Term Project Showcase (STePS) is a bi-annual event organized by NUS to showcase students' talents through their projects. These projects are usually module capstone project and final year projects. Although compulsory, most of these projects are designed, innovated and coded by students themselves and thus is able to demonstrate their individuality as well as creativity and effort.

STePS itself has a system which has event workflow management, registration, check-in and voting modules. As the event is getting bigger, we will like to enhance the system with a networking module to enable attendees to network meaningfully and effectively before, during and after events. Our project, thus, is a module which classify/group users based on their interest and purpose of attending STePS, allow users to post/comment in forums of each STePS run, have personal chat with other attendees.

In short, it is a application designed primarily to network students with future employers, co-founders and teammates through project showcases and ultimately, beyond STePS.

1.2 Techstack

This project is a web application designed mobile-first and to be used mainly on Google Chrome.

Environment:
Node.js (>= v6.3.0)
ES6 Javascript

Front-end:
React.js
HTML5
Sass
Socket.io

Back-end:
Express.js
MongoDB (Server)
Mongoose (ORM)

Dev-Ops:
Gulp (build)
Webpack (bundle)
ESLint (Airbnb JS)
Babel (transpile)

Testing:
Mocha
Chai

Deployment:
Travis (integration)
Heroku (host)
Vagrant

Styling:
Bootstrap 4

1.3 Installation and Development

We are developing on https://github.com/nus-mtp/steps-networking-module and hosting it on https://steps-networking-module.herokuapp.com.

For first time installation and development, to begin, ensure that you have a working Node.js environment. Check the version by typing `node -v` in the command line. After that, follow these steps:

1. Clone the repository from Github
2. Run `npm install` at the root of the repository
3. Run `gulp` in the command line
4. Access the web application with the port displayed on the command line e.g. localhost:3000 when the port displayed is Port 3000.

To have access to the data, ensure that you have MongoDB installed and set the path environment to use MongoDB (or just run it directly).

1. Run `mongod`
2. Run `mongo`
3. Run `gulp full-convert` to populate the database with all STePS data or `gulp active-convert` to populate database with active STePS data

2. Architecture

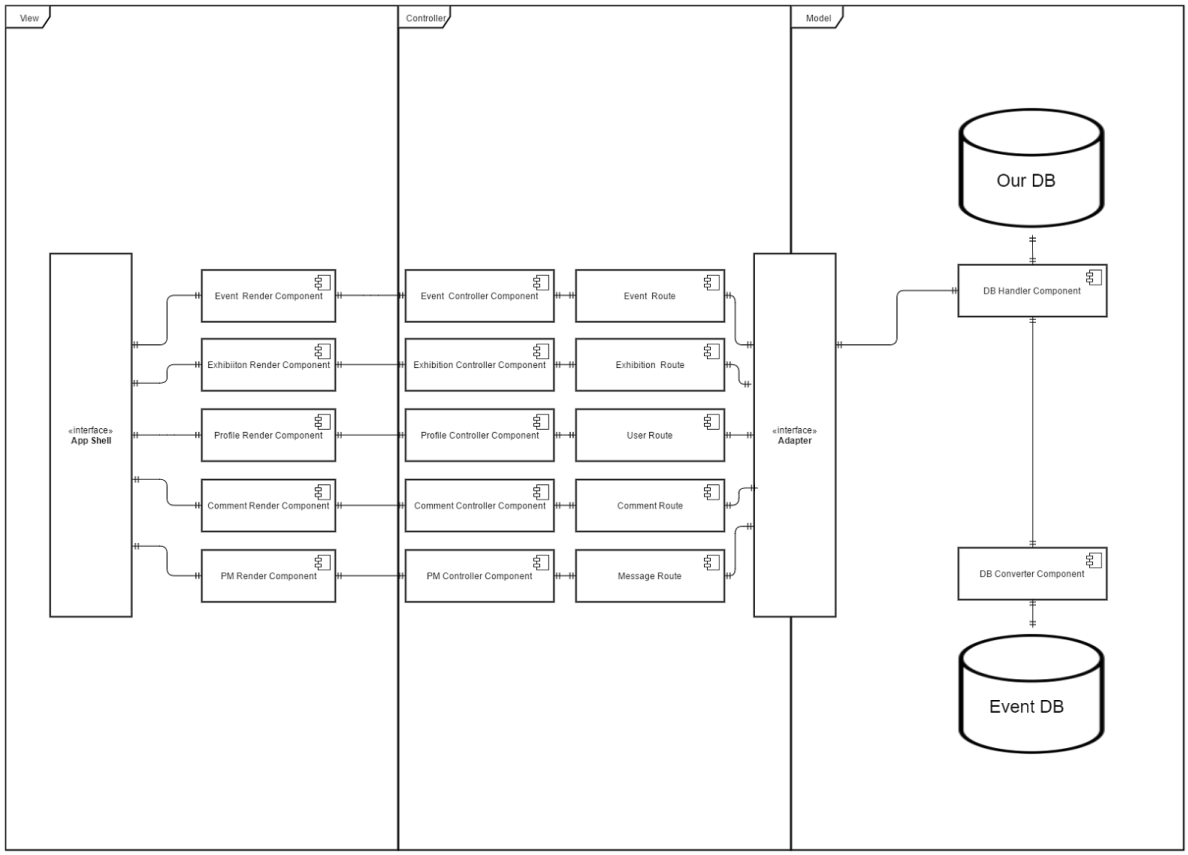

Figure 1: Architecture Diagram

Our Application Architecture (high-level) follows the Model-View-Controller pattern.

The presentational aspect of our site is rendered through the View Components.
The interactive aspect of our site is handled by the Controller Components.
The stateful aspect of our site is managed by the Model Components.

3. Components

3.1 View

The view is developed modularly - each view component is linked to its own controller. All of the views are then hosted on the Application Shell.

3.1.1 Application Shell

The application shell functions as the component that hosts all other views. As the application itself is a Single Page Application (SPA), all other views will share the subcomponents of the application shell.

The main subcomponent of the application shell is the Navigation Bar. The navigation bar contains links that redirects to other pages. In pages other that the authentication pages and the home page, the navigation will also host the Search component.

The application shell is responsive such that when viewed on mobile, the navigation bar becomes a dropdown toggled by the 'hamburger' menu icon.

Clickable Links

Links	**Route**
Logo (logged in)	`/`
Profile (logged in)	`/profile/:email`
Chat (logged in)	`/chat` `/chat/:email`
Logout (logged in)	`/logout`
Login	`/login`
Signup	`/signup`
Search Input (only when not in homepage)	Same routes as section 3.1.3

3.1.2 Home

The homepage or the landing page consist of the Search, Events and Tabs. For Search, refer to *section 3.1.3*. The homepage is where user will be able to see all the different Events categorized in different Tabs and their respective descriptions. User will be able to toggle between different Events Tabs based on the current date.

Clicking on the Event will open up a card underneath the event, giving the user the basic description of the said event.

If the user is participating as an exhibitor for a particular event, he/she will be able to see two tags - “attending” and “participating”, that are tied to the event. These tags for the particular event will be locked. The user will be able to toggle their attendance for events that they are not participating as an exhibitor. An “attending” tag will appear once user confirms their attendance.

After the user’s attendance has been confirmed, they will be able to see another row of details in the event card. This is one of the major features of our project, matchmaking which is explained further in *section 3.1.10*.

Clickable Links

Links	**Routes**
Red icon on event	`/event/:eventName`
Users in match	`/match/:email`

3.1.3 Search Input

The Search component is rendered in all pages with the navigation bar. However, on the homepage, the Search is extracted from the navigation bar and rendered onto the body.

The Search pulls the content from database through the use of routes depending on the filter selected. The four filter choices are:

1) Event
2) Exhibition
3) Skills (User skills)
4) Tags (Exhibition tags)

Upon selecting the filter, the information is retrieved (refer to *section 3.2.2* for more depth) and a list is generated in the search suggestion list. The search suggestion list is rendered only after two or more characters are inputted into the search bar.

Clickable Links

Links	**Routes**

Search icon (depending on filter)	`/event/:eventName`
	`/exhibition/:eventName/:exhibitionName`
	`/search/exhibition/:tags`
	`/search/user/:skills`

3.1.4 Search Results

Only searching with user skills or exhibition tags filter will redirect users to the search results page. If the search is valid, it will contains results with all the users or exhibitions with the skills or tags that is searched by the user. If the search is invalid, it will display, a feedback indicating that the search is invalid. Whereas for Event and Exhibition, invalid searches will show an invalid event page and 404 page not found respectively.

Clickable Links

Links	**Route**
Exhibition results	`/exhibition/:eventName/:exhibitionName`
User results	`/profile/:email`

3.1.5 Events

Each event has its own individual page. Each page contains information on the event which is read directly from the database when the event page is mounted based on the event name input as url. The information received are:

1) Event
2) Attendance (all users attending)
3) Exhibitions (all exhibitions in the event)

Attendance and exhibitions are toggleable with buttons that will expand respective lists. The event image is rendered with HTML `<embed>` and is therefore an iframe (inline frame) on it's own. There are also organizer details and a sitemap image included which is currently hard-coded and will be removed in future versions.

Clickable Links

Links	**Routes**
User in attendees	`/profile/:email`
Exhibition in exhibitions	`/exhibition/:eventName/:exhibitionName`
Organizer website (hard-coded)	`https://www.comp.nus.edu.sg/`

3.1.6 Exhibitions

Each exhibition has its own individual page. Each page contains information on the exhibition which is read directly from the database when the exhibition page is mounted based on the event name and exhibition name input as url. The information received are:

1) Exhibition
2) Attendance (all users that are involved with the exhibition)

Only the exhibition tags is editable and it has it's own button which enables edit input to be displayed or closed.

There is a comment section view which renders the comments and are categorized by user emails. Each comment have a timestamp and the comment text. We have decided to sort it by user emails as we want to discourage users from having long conversations in the comments. This is to prevent any overloading of a single exhibition page and encourage users to utilise the Chat (refer to *Section 3.1.8*) instead. This arrangement can be changed in future development.

Clickable Links

Links	**Routes**
User in attendees	`/profile/:email`

3.1.7 Profiles

Each user has their own profile. The profile page receives information based on the email input as url. The information received are:

1) User
2) Event (event that the user is attending)
3) Exhibitions (exhibitions that the user is participating)

There are two accordions at the bottom of the page which went clicked, will render the events and exhibitions that the user is involved in. Upon mounting, a edit icon button or a chat icon button will be rendered depending on whether the user enter his/her own url or other another user's profile.

The edit button renders the edit view which enables user to edit and change their own profile information, including their reasons for exhibiting in an event to enable matching, further explained in *Section 3.1.10*. The chat button redirects to the chat page (*section 3.1.8*).

Clickable Links

Links	**Routes**
Chat icon	`/chat/:email`
Exhibition in exhibitions involved	`/exhibition/:eventName/:exhibitionName`
Events is events involved	`/event/:eventName`

3.1.8 Chat

Chat consists of only one page, and the view updates internally. The chat draws messages sent and received by the user from the database and displays it to the user. If the user has not sent or received any messages, the chat page will display an empty page with a message that there

are no chats available. From the empty page, the user is only able to add new people to talk to by clicking the chat icon via the profiles or inputting the email of the user s/he wishes to talk to in the chat url.

Once the user has another user to talk to and has exchanged messages, the chat's full functionality is displayed. The main view in chat is covered in the subcomponent ChatBody, which displays the messages sent by the user on the right in green and those sent to the user on the left in grey. There is also an input box pinned at the bottom of the page that allows the user to type in their own messages. In desktop mode, a list of users that the user has conversations with is displayed on the left, displayed through the subcomponent ChatTabs. ChatTab buttons can be used to change the messages being displayed in ChatBody. In mobile mode, these tabs are condensed into a dropdown list pinned at the top of the page.

Clickable Links

Links	**Routes**
Add user to talk to by email	`/chat/:email`
Buttons to change user	`Uses internal state changes (this.state.current)`

3.1.9 Authentication

The authentication consists of the login view and the signup view. The login view for familiar users whereas the sign up view is for first-time users. No user can use the application without first being registered. All routes leads to the login page when not logged in.

Clickable Links

Links	**Routes**
Login (when on the signup page)	`/login`
Signup (when on the login page)	`/signup`

3.1.10 Match

Once a user confirms their attendance for an event, users will be able to see other users who have the same interest as you by ticking the appropriate checkboxes. By clicking on the other user, users will be able to see their profile (*Section 3.1.7*) and chat with them (*Section 3.1.8*). If you click on 'see more', a different page will be rendered and you will be able to see all your matches by pressing next or previous.

3.1.11 404 Not Found

This is rendered when no such client side route or when no such event, exhibition or profile found.

3.2 Controller

Only certain views are linked to a controller. This because not all views are dynamic and some of them only serve as either a link to other pages or to improve the user experience in terms of page flow. All views that have a controller has its own controller logic.

3.2.1 Home

Every event will be presented in different tabs as mentioned earlier. We do so by comparing the current date and the event date. On tab change, it will re-render the HomeView component with the correct filter. `setDefaultView()` will choose which tab to be active upon page load according to the priority → Ongoing, Upcoming, Past. When the active tab is empty on load, the active tab will change to next highest priority.

Every event is linked with its respective collapsable. The collapsable is a responsive component which changes its width based on the number of event displayed in a row. Event listener is added to the component for any changes in the window width so that the collapsable of an event span the entire row.

Notable APIs

Method	Description
`retrieveData()`	Calls server side routes via HTTP GET to retrieve information on all events
`getRelevantUsers(array)`	Calls server side routes via HTTP GET to retrieve information on matching users based on the checkboxes that the user selected

3.2.2 Search Input

The search input is read from the text input bar. This value is bound to the `search` state. Upon user input, the `getSearchAsync()` filters the cached data, which is the `searchDefault` state retrieved from the database. It will filter by returning the event or exhibition that contains the user query as a substring and thus renders the suggestion (refer to *Section 3.1.3* for the suggestion rendering). Which cache to filter and render depends to the `searchFilter` state. The `searchFilter` state is updated via a listener attached to the filter. The listener in turn calls the `filterSearch()` function to alter the `searchFilter` state on change of the filter dropdown.

Notable APIs

Method	Description
`filterSearch()`	Calls server side routes via HTTP GET to retrieve information on event, exhibitions or users depending on the `searchFilter` state

3.2.3 Events

Each event page will have their own attendance list and project list upon initialization. `updateDisplayedExhibitions()` allow the component to filtered out those exhibitions that the user does not want to see. Each exhibition will have two tags tied to them, namely *eventName* and *projectModule,* in that order. Hence, the items in the dropdown list may differ from one event to another as it will extract out the tags.

3.2.4 Exhibition

The controller for exhibition mainly concerns on the Tags and comments. Each exhibition has a state to cache the tags and comments.

The tags' logic is handled mainly by an external dependency called `ReactTags` installed via `npm`. When a tag is modified, the data is first cached. The HTTP POST call is made by the `saveTags()` function and this is only done after the ***Save*** button is clicked.

For comments, the user input is read and mapped to the state `currentComment`. If the ***Submit*** button is clicked, the HTTP POST request is called by `saveComments()`. After the comment is saved, the components reloads the comments from the database and re-renders the comments.

Notable APIs

Method	Description
`saveTags()`	Calls server side routes via HTTP POST to save exhibition tags onto the server
`saveComments()`	Calls server side routes via HTTP POST to save user comments onto the server
`retrieveData()`	Calls server side routes via HTTP GET to retrieve information on exhibition, attendance and comments

3.2.5 Profiles

The user profile can only be edited by the user him/herself. The profile component keeps a Upon clicking the edit icon, editable is enabled. There are three editable components which are:

1) Description
2) Skills
3) Links
4) Attendance reasons (for exhibition)

Each of the editable component is bound to its own handler and cache. Skills and links uses the `ReactTags` dependency. Description is bound to the state and updates with a `onChange` listener. All values will not be saved unless the ***Save*** button is clicked. This will send a HTTP POST to the database via the `setUserInfo()` and `saveReasons()` function. After the data is updated, the cache states are updated directly.

Notable APIs

Method	Description
`saveReasons()`	Calls server side routes via HTTP POST to save reason for attending each exhibition
`setUserInfo()`	Calls server side routes via HTTP POST to save user edited information which are skills, links and description
`retrieveData()`	Calls server side routes via HTTP GET to retrieve information on user, event, exhibition and attendance for both event and exhibition

3.2.6 Chat

This component is divided into 3 subcomponents: ChatView, ChatBody and ChatTabs. ChatView is the wrapper for ChatBody and ChatTabs and is the main component displayed by the App shell. ChatView renders ChatBody and ChatTabs appropriately depending on the mobile or desktop view, which is checked using a MediaQuery. ChatView also contains the sockets required for the chat to function, sending a connection request to the server in advance, allowing ChatBody to send messages to the server after.

The flow in which Chat is mounted is as follows:

- User A enters the Chat component
- As the ChatView component is initialised, the email of User A is obtained from local storage and sent to the database to query a list of Users that User A has conversations with.
- At the same time, ChatView initialises the socket and sends a 'new user' request to the socket manager of the server.
- When the database returns with the list, ChatTabs is passed the information if in Desktop mode to render the list of Users.
- By default, the first person in the list is set to render for ChatBody. However, if a specific user is requested via URL or linked from the Profiles, that User is selected to be rendered instead.
- ChatBody uses the sockets initialised by ChatView to obtain the sent and received messages of the current conversation. It is then rendered.
- If the list of Users returned by the server is empty and chat is not linked from a URL or Profiles, a page indicating that there is no content is rendered in place of ChatTabs and ChatBody.

The flow in which messages are sent and received are as follows:

- User A types a message to User B and hits Enter to send
- The message content is obtained from the input box and sent to the server via sockets
- The server receives the content and stores it in the database before sending a notification request to both User A and User B. (Sent to User A as well in the case that there are more than one instance of User A's account open)

- User A and User B receive the notification and are prompted to call the refresh function, which fetches the data from the server via the sockets
- View is updated with the new content

Notable APIs

Method	Description
`ChatView: getAllUsers()`	Calls server side routes via HTTP POST to obtain the list of users that the current user has messages with. This is saved into `this.state.users` which is then provided to the ChatBody component
`ChatBody: retrieveAllMessages(senderEmail, recipientEmail)`	Sends a socket request to server side to obtain messages. To obtain all messages between User A and User B, this function is called twice, once with User A as `senderEmail` and User B as `recipientEmail` and then once more with both attributes swapped.
`ChatBody: sendMessage(senderEmail, recipientEmail, content)`	Sends a socket packet with the information embedded whenever the user submits a new message. The current user is `senderEmail`, while `recipentEmail` is the person the message is being sent to, while content is the message itself.
`ChatBody: getMessages()`	Merge sorts the messages obtained by `retrieveAllMessages()` and wraps them into their respective customised , providing the main view of the chat.
`ChatBody: getName(fixToTop)`	`fixToTop` is set as true when in mobile mode. Handles the dropdown list that is displayed in mobile mode, letting the user switch between conversations since ChatTabs is not displayed in mobile omde.

3.2.7 Authentication

Each input on the login and signup form has each own handler. The HTTP POST will only be executed when the user clicks the ***Submit*** button. The function `processForm()` will make the call and upon valid submission of the login/signup form, the user will be redirected to the homepage.

Notable APIs

Method	Description
`processForm()`	Calls server side routes via HTTP POST to authenticate user credentials and retrieve a token from the server

3.2.8 Match

As mentioned earlier, when user is attending an event, they will be presented with a predefined checkboxes in the collapsable of the event. HTTP POST will be executed when user check/uncheck any of the tickboxes. The function `getRelevantUsers()` will be invoked. This in turn will collect all the relevant users from the database.

Notable APIs

Method	Description
`getRelevantUsers(array)`	Calls server side routes via HTTP POST to gather users who matched the elements in the array.

3.3 Model

The application specific model comprises of 4 main parts. The lowest level involves the database schemas (table definitions), which are defined using the Mongoose ORM. The next level involves the ModelHandlers, which defines specific helper functionality related to Mongoose. The next level are general 'Object Classes' which act as an abstraction / facade for the database-specific operations, each manipulating data related to only one database schema each. Finally, the topmost layer are the routes / controllers, which implement functionalities that involve one or more Object Classes.

An ad-hoc part of the model involve the stepsConverters, which are functions that, once executed, attempt to read off the steps database and transfer and convert information into this system's database.

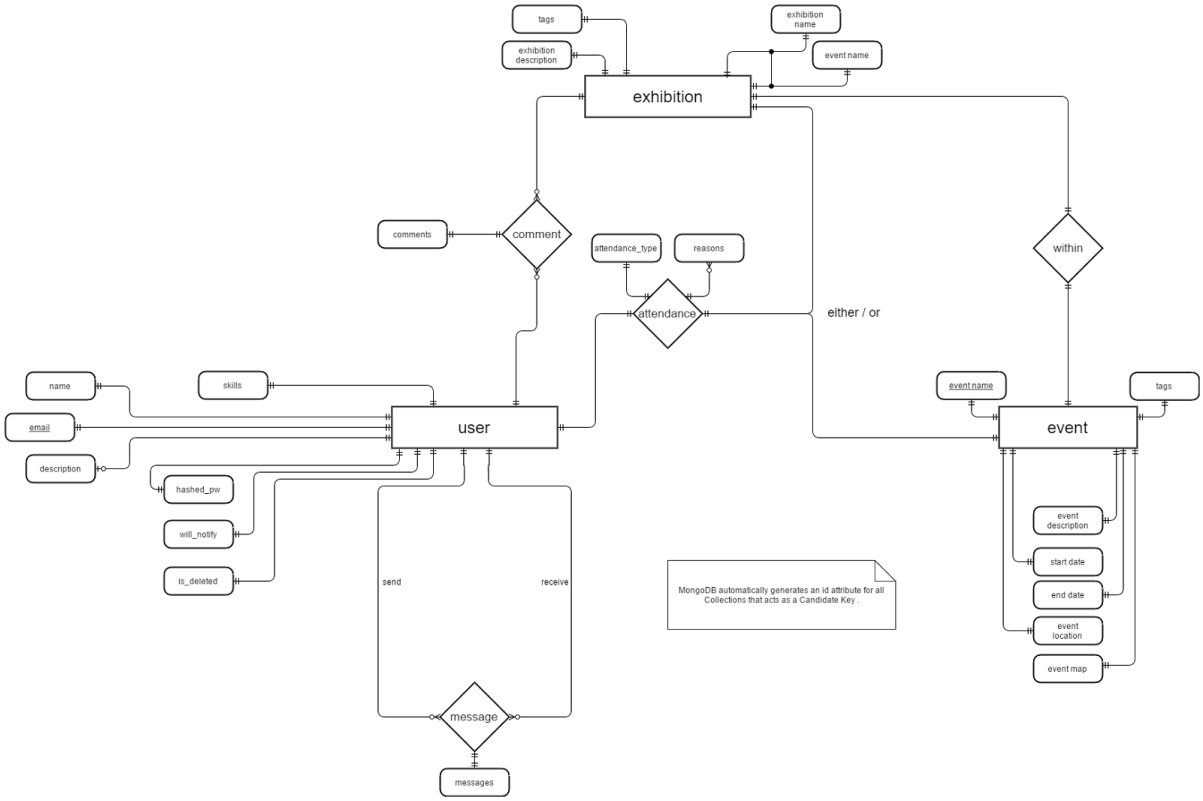

Figure 2: Entity Relation Diagram of Model

3.3.1 Schema Definitions

Each Schema is defined with the Mongoose ORM for MongoDB. Hence, each instance of the Schema has a **_id** automatically generated, which is unique for that instance throughout the whole database.

Currently in the project there are 2 different schema definition groups. One group is for storing the state of our system, and the other is for drawing information from STePs to populate our database.

3.3.1.1 Our Schemas

Described below are the various Schemas and their attributes. Underlined attributes denote required attributes.

3.3.1.1.1 User

The User Schema captures information related to a User of our system.

1) **email**: A unique, automatically trimmed, String identifier for each User. Acts as a candidate key for this Schema.
2) **password**: Tracks a User's specified secret in a String, to be used as part of logging in.
3) **name**: A non-unique String identifier for each User.
4) **description**: A String that stores a User's self-description.
5) **will_notify**: A Boolean that is to be used in deciding if a User should be alerted to certain state changes.
6) **is_deleted**: A Boolean that tracks whether a User has opted to delete his account from the system. The system should not actually delete the information - but rather lock it from any non-automated modification for simplicity and data integrity concerns.
7) **profile_picture**: A String that stores an URL to an externally hosted visual depiction of the User.
8) **links**: An array of automatically trimmed Strings representing URLs to external sites that the User wants to show to others. Does not check for duplicates.
9) **skills**: An array of automatically trimmed, lowercase Strings that tracks the subject matters a User is proficient at. Does not check for duplicates.
10) **bookmarked_users**: An array of ids of other Users that the User wants to track. Does not check for duplicates.

3.3.1.1.2 Event

The Event Schema captures information related to an Event stored in our system.

1) **event_name**: A unique, automatically trimmed, String identifier for each Event. Acts as a candidate key for this Schema.
2) **event_description**: A String that stores the Event's description.
3) **start_date**: A Date object that denotes the start timing of the Event. Defaults to the current time if not specified.
4) **end_date**: A Date object that denotes the end timing of the Event. Defaults to the current time + 1 day if not specified.
5) **event_location**: A String that stores the address of the Event.
6) **event_map**: A String that is supposed to store a URL to an externally hosted visualization of the Event's location.
7) **event_picture**: A String that stores a URL to an externally hosted visual representation of the Event.
8) **tags**: An array of automatically trimmed, lowercase Strings that each represent a label for the Event. Does not check for duplicates.

3.3.1.1.3 Exhibition

The Exhibition Schema captures information related to an Exhibition (Exhibit) within an Event. The event_name and exhibition_name form a compound candidate key.

1) **exhibition_name**: An automatically trimmed String identifier for each Exhibition. Acts as a candidate key for this Schema.
2) **event_name**: Stores an automatically trimmed identifier of the Event this Exhibition is hosted under. Acts as a Foreign Key reference. Does not check for the validity of the supplied identifier.
3) **exhibition_description**: A String that stores the Exhibition's description.
4) poster: A String that stores a URL to an externally hosted visual representation of the Exhibit.
5) **images**: An array of Strings that each represent a URL to externally hosted supporting visual representations of the Exhibit. Does not check for duplicates.
6) **videos**: An array of Strings that each represent a URL to externally hosted videos that supply more information about each Exhibit. Does not check for duplicates.
7) **website**: A String that stores a URL to an external site for the Exhibition.
8) **tags**: An array of automatically trimmed, lowercase Strings that each represent a label for the Exhibition. Does not check for duplicates.

3.3.1.1.4 Attendance

The Attendance Schema captures attendance information between one User and one Event, or Exhibition. If a User is participating in an Exhibition, an Attendance document should be generated for both the Event and Exhibition. The user_email and attendance_key form a compound candidate key.

1) **user_email**: An automatically trimmed reference to the email of a User. Does not check for the validity of the supplied User email.
2) **attendance_key**: An id of either an Event or Exhibition.
3) **attendance_type**: An Enum String that denotes which type of activity this attendance tracks. Can only either be 'event' or 'exhibition'. Defaults to 'event' if unspecified.
4) **reason**: An array of automatically trimmed, lowercase Strings that tracks the reasons why the User is participating in that activity.

3.3.1.1.5 Post

The Post Schema is not intended to be a standalone Schema - rather it is meant to model a typical forum post - with content and timestamp.

1) **content**: A String that denotes the text in the post.
2) **timestamp**: A Date object that tracks when the Post was created. Defaults to the current time if not specified.

3.3.1.1.6 Message

The Message Schema is designed to capture all messages sent from one User to another User(monodirectional conversation). The recipient_email and sender_email form a compound candidate key.

1) **recipient_email**: An automatically trimmed String reference to the email of the recipient User in this monodirectional conversation.
2) **sender_email**: An automatically trimmed String reference to the email of the sender User in this monodirectional conversation.
3) **messages**: An array of Posts that represent the messages sent by a User to another User.

3.3.1.1.7 Comment

The Comment Schema is designed to capture all comments made by a User to an Exhibition. The user_email and exhibition_key form a compound unique key.

1) **user_email**: An automatically trimmed String reference to the email of a User. Does not check for the validity of the supplied User email.
2) **exhibition_key**: The id of the Exhibition that the User comments are directed at. This is used as it is more convenient than having to supply both the Event and the Exbition name to uniquely identify a Comment object.
3) **comments**: An array of Posts that represent the comments made by a User to a Exhibition.

3.3.1.2 STePs Schemas

This section gives a brief overview of the STePs Schemas and their related attributes that our system reads off the STePs database. Information derived from these attributes is to be added into the application's database through the STePs converter functions, which will be explained in Section 3.3.5. Underlined attributes denote required fields.

3.3.1.2.1 stepsUserSchema

The stepsUserSchema captures information related to one User (either Admin / Creator, Staff, or Participant) at STePs.

1) **email**: A unique, automatically trimmed, lowercase String identifier for the STePs User.
2) **name**: A non unique, automatically trimmed String identifier for the STePs User.

3.3.1.2.2 stepsEventSchema

The stepsEventSchema captures information related to one Event at STePs.

1) **code**: A unique, automatically trimmed, lowercase String identifier for the STePs event.
2) **name**: A non-unique, automatically trimmed, String identifier for the STePs Event.
3) **description**: An automatically trimmed String containing a brief explanation of the STePs Event.
4) **startTime**: A Date object tracking the start time of the STePs Event.
5) **endTime**: A Date object tracking the end time of the STePs Event.
6) **location**: A String containing a description of the address of the Event.

7) **promote**: A nested object which contains links to externally hosted media that provides more information about the STePs Event.
8) **isDefault**: Is the currently active Event at STePs.

3.3.1.2.3 stepsGuestSchema

The stepsGuestSchema essentially captures ticketing information for all STePs Events. This includes the Users, as well as external guests not captured in the system.

1) **event**: A String that is a Foreign Key reference to the STePs Event that this person is attending.
2) **name**: An automatically trimmed String that is an identifier for the person involved.
3) **email**: A String that acts as another identifier for the person involved.

3.3.1.2.4 stepsModuleSchema

The stepsModuleSchema captures the Module information (or tracks) for a Module being hosted in a given Event.

1) **event**: A String that is a Foreign Key reference to the STePs Event this STePs Module is being hosted under.
2) **code**: An automatically trimmed String that acts as an identifier for the STePs Module.
3) **projects**: An array of stepsProject objects that this STePs Module hosts.

3.3.1.2.5 stepsProjectSchema

The stepsProjectSchema captures the STePs Project information for one STePs Project being hosted in a given Module. This Schema is never directly used in the code, and is only brought in for reference purposes when reading the projects attribute of the stepsModuleSchema.

1) **name**: A String that is an identifier for the STePs Project.
2) **description**: A String that stores information about the STePs Project.
3) **posterLink**: A String that stores a URL to an externally hosted visual depiction of the Project.
4) **imageLinks**: An array of Strings that each store URLs to externally hosted supporting visual depictions for the Project.
5) **videoLink**: A String that stores a URL to an externally hosted video that delivers additional information about the Project.
6) **urlLink**: A String that stores a URL to an externally hosted Project website.
7) **members**: An array of ids of STePs Users that is participating in this Project.

3.3.2 Model Handlers

With Mongoose, it is required that the Schemas are 'applied' to the connections established to the database. The resultant object that is returned upon successful application of a Schema to the database connection is a Model object. Through this Model object, one is able to perform CRUD operations on the database collections.

As mentioned earlier, there are two different groups of Schemas. Hence, there are also two different classes of Models. The codebase of this project designates each group one Javascript Class whose methods support the creation, initialization and maintenance of the related Models.

The ModelHandlers have some common functionality, described in the table below.

Notable APIs

Return Type	Method and Description
`ModelHandler`	`initWithConnection(mongoose.Connection db)` Initializes a supplied database connection with our Schemas, creating and storing the Models produced. Returns a reference to itself to allow for chaining function calls.
`ModelHandler`	`initWithUri(String mongoURI)` Initializes a new database connection, given the uri of the running Mongo server process, and applies our Schemas, creating and storing the Models produced. Returns a reference to itself to allow for chaining function calls.
`mongoose.Connection`	`getConnection()` Returns the database connection the ModelHandler has initialized the Schemas on. Typically used together with `initWithUri()`.
`void`	`disconnect(callback)` Closes the connection the Modelhandler has initialized the Schemas on. Executes the callback function when the operation completes. Typically used when the ModelHandler was initialized with the URI (`initWithUri()`).

3.3.2.1 ModelHandler

This section denotes the getter functions for the Models produced by applying our Schemas to a database connection.

Notable APIs

Return Type	Method and Description
`mongoose.Model`	`getUserModel()` Returns the Model produced when the User Schema is applied to the database connection.
`mongoose.Model`	`getEventModel()` Returns the Model produced when the Event Schema is applied to the database connection.
`mongoose.Model`	`getExhibitionModel()` Returns the Model produced when the Exhibition Schema is applied to the database connection.
`mongoose.Model`	`getAttendanceModel()` Returns the Model produced when the Attendance Schema is applied to the database connection.
`mongoose.Model`	`getCommentModel()` Returns the Model produced when the Comment Schema is applied to the database connection.
`mongoose.Model`	`getMessageModel()` Returns the Model produced when the Message Schema is applied to the database connection.

3.3.2.2 StepsModelHandler

This section denotes the getter functions for the Models produced by applying the STePs Schemas to a database connection.

Notable APIs

Return Type	Method and Description
`mongoose.Model`	`getUserModel()` Returns the Model produced when the stepsUserSchema is applied to the database connection.
`mongoose.Model`	`getGuestModel()` Returns the Model produced when the stepsGuestSchema is applied to the database connection.
`mongoose.Model`	`getModuleModel()` Returns the Model produced when the stepsUserSchema is applied to the database connection.

`mongoose.Model`	`getEventModel()` Returns the Model produced when the stepsUserSchema is applied to the database connection.

3.3.3 Object Classes

There are a total of 6 object classes, namely - User, Event, Exhibition, Attendance, Message and Comment. Each object class contains the ability to create an instance, or Mongoose Documents, of the namesake Models, as well as functions to fully execute certain operations on the database. In order to use the object classes methods, they must be initialized with an open connection to the database. Each object class comes with it's own connection API which will ensure that there is only a single instance of connection running at a time.

For full API on object classes, please refer to *Appendix*.

Notable APIs

Return Type	**Static Method and Description**
`void`	`setDBConnection(Mongoose.Connection database)` Takes in a mongoose connection object, and if there is no valid connection made, a new connection will be made to the database, and obtains the respective model from ModelHandler.
`void`	`checkConnection()` Checks if there is an existing connection, mongoose model and if the database is in a ready state.

Note that most of the methods in the object classes are static, and do not return anything. Instead, they utilize callback functions. This is because these methods mainly deal with asynchronous database operations, and a regular return statement will be in danger of being executed before the prior asynchronous operations are completed.

A callback function here usually returns one or more items. The first item tends to be an error state, and the second is a result state.

The only methods that are not static in each object class are the constructor and save methods.

3.3.3.1 User

The User object class is used to represent a single person's account. It uses the same attributes naming as the User Schema.

Notable APIs

Callback	Methods and Descriptions

No callback required.	`Constructor(`
	`String userEmail,`
	`String userName,`
	`String userDescription,`
	`String userPassword,`
	`boolean willNotify,`
	`boolean isDeleted,`
	`String profilePic,`
	`Array<String> links,`
	`Array<String> skillsets,`
	`Array<mongoose.Schema.ObjectId>`
	`bookmarkedUsers)`
	Takes in parameters as shown above and creates a new User JSON.

`callback(err, userObject)`	`saveUser(callback)` Creates a new Mongoose Document using the User JSON and saves it into the database. It will return the newly saved User object in the callback.

`callback(err, userObject)`	`getUser(`
	`String userEmail,`
	`callback)`
	Goes through database to find a document with the matching email and returns it. It will return a single User Object.

`callback(err, Array <`
`userObject>)`
	`searchUsersByMultipleSkills(`
	`Array <String> skillToBeSearched,`
	`callback)`
	Finds users that has all the specific skills specified.

`callback(err, userObject)`	`updateUserDescription(`
	`String userEmail,`
	`String description,`
	`callback)`
	Finds the matching User document, updates the description and saves it back into database. The updated User object is passed into the callback.

`callback(err, userObject)`	`setSkillsForUser(`
	`String userEmail,`
	`Array <String> skills,`
	`callback)`
	Finds the matching User document, sets the array of skills for the user and saves it back into database. The updated User object is passed into the callback.

`callback(err, userObject)`	`setUserAsDeleted(`

	`String userEmail,`
	`boolean isDeleted,`
	`callback)`
	This method sets whether the User is marked as deleted. They will not be deleted from the database, but will be considered as an invalid User. An invalid User is one whose information cannot be changed, other than through automated processes. This is to minimize data integrity issues.

3.3.3.2 Event

An Event object is used to identify a single event. In a single Event, there can be multiple Exhibitions going on, but for each Exhibition, it can only be in a single Event. The Event object contains event details as its attributes as listed in the Event schema.

Notable APIs

Callback	**Methods and Descriptions**

No callback required.	`constructor(`
	`String eventName,`
	`String eventDescription,`
	`Date startDate,`
	`Date endDate,`
	`String location,`
	`String map,`
	`String eventPicture,`
	`Array <String> tags)`
	Creates a new Event JSON object and stores internally.

`callback(err, eventObject)`	`saveEvent(callback)` Save the Event JSON object into a Mongoose Model Document and saves it into the database. The saved Event object is then passed into the callback.

`callback(err, eventObject)`	`getEvent(`
	`String eventName,`
	`callback)`
	Search through the database to find a matching Event object based on the event name specified. The search result is then passed back into the callback.

`callback(err, Array <eventObject>)`	`getAllEvents(callback)` Retrieves all the Event objects found in the database and passed as an array of Event Object into the callback.

`callback(err, Array <eventObject>)`	`searchEventsByTags(`
	`Array <String> tags,`
	`callback)`
	Search the database to find all Event objects that contains all the tags.

`callback(err, eventObject)`	`updateEventTag(`
	`String eventName,`
	`Array <String> tags,`
	`callback)`
	Search the database to find the matching event and updates the tags attribute. The updated Event object is then passed into the callback.

`callback(err, eventObject)`	`deleteEvent(`
	`String eventName,`
	`callback)`
	Search the database to find the matching Event and removes it from the database. Does not enforce data integrity by removing any Exhibition or Attendance objects referencing it.

3.3.3.3 Exhibition

An Exhibition object tracks the information of a sub-activity happening within an Event. Each Exhibition should be tied to one existing Event.

Notable APIs

Callback	**Method and Descriptions**

No callback required.	`constructor(`
	`String exhibitionName,`
	`String exhibitionDescription,`
	`String eventName,`
	`String posterURL,`
	`Array<String> images,`
	`Array<String> videos,`
	`String website,`
	`Array<String> tags)`
	Creates a new Exhibition JSON and stores it internally.

`callback(err, result)`	`saveExhibition(callback)` Save the Exhibition JSON object into a Mongoose Model Document and saves it into the database. The saved Exhibition object is then passed into the callback. Note that this does not validate the supplied Event name.

`callback(err, exhibitionObject)`	`getExhibition(`
	`String eventName,`
	`String exhibitionName,`
	`callback)`
	Retrieve an Exhibition object from the database.

`callback(err, Array<exhibitionObjects>)`	`searchExhibitionsByEvent(`
	`String eventName,`
	`callback)`
	Retrieve all Exhibitions hosted under a single Event.

`callback(err, exhibitionObject)`	`setTagsForExhibition(`
	`String eventName,`
	`String exhibitionName,`

	`Array<String> tags,`
	`callback)`
	Tag an Exhibition with the numerous Strings specified in the 'tags' argument.

`callback(err)`	`deleteExhibition(`
	`String exhibitionName,`
	`callback)`
	Delete an Exhibition object from the database. Does not enforce data integrity by removing any Attendance or Comment objects referencing it.

3.3.3.4 Attendance

An Attendance object tracks the User's visitation to an Event, or participation in an Exhibition. The Attendance object uses **attendanceType** to differentiate whether it tracks the attendance for a Event or Exhibition for that particular User. As there might be Events with similar Exhibition names, we have decided to use the object ID, **_Id**, of the specified Event/Exhibition as a form of unique identification. If a User is an exhibitor, they will have another Attendance object as a guest within the same Event. The purpose of this is to ensure that the exhibitors will have all the same rights and privileges as the guests, and maybe more.

Notable APIs

Callback	**Methods and Descriptions**

No callback required.	`constructor(`
	`String userEmail,`
	`mongoose.Schema.ObjectId attendanceKey,`
	`String attendanceType,`
	`Array <String> reason)`
	Creates a new Attendance JSON object and stores internally.

`callback(err, attendanceObject)`	`saveAttendance(callback)` Saves the Attendance JSON object into a Mongoose Model Document and saves it into the database.

`callback(err, attendanceObject)`	`getAttendance(`
	`String userEmail,`
	`mongoose.Schema.ObjectId attendanceKey,`
	`callback)`
	Retrieve a specific Attendance object from the database using the userEmail and attendanceKey.

`callback(err, Array <attendanceObject>)`	`searchAttendancesByUser(`
	`String userEmail,`
	`callback)`
	Retrieve all the Attendance object by a specific User. An Array Attendance objects

	are passed into the callback.
`callback(err, Array <attendanceObject>)`	`searchAttendancesByKey( mongoose.Schema.ObjectId attendanceKey, callback)` Retrieve all the Attendance object involving a certain Event or Exhibition.
`callback(err, attendanceObject)`	`updateReason( String userEmail, mongoose.Schema.ObjectId attendanceKey, String reason, callback)` Search through the database to locate a specific Attendance object and updates the reason for attending. The updated Attendance Object is then returned in the callback.
`callback(err, attendanceObject)`	`deleteAttendance( String userEmail, mongoose.Schema.ObjectId attendanceKey, callback)` Removes a specific Attendance Object from the database

3.3.3.5 Message

A Message object tracks the entirety of a monologue, from one User to another. Two Message objects are required to simulate a full dialogue.

Notable APIs

Callback	**Methods and Descriptions**
No callbacks required	`constructor( String senderEmail, String recipientEmail, String content, Date timeStamp)` Creates a new Message JSON object and stores internally.
`callback(err, messageObject)`	`saveMessage(callback)` Saves the Message JSON stored internally into the database.
`callback (err, Array <String> listOfEmails)`	`getEmailsInvolvingUser( String userEmail, callback)` Search the database for all Message objects that involves a specific user.
`callback(err, messageObject)`	`getConversation(`

	`String senderEmail,`
	`String recipientEmail,`
	`callback)`
	Search the database for the specific Message object that is sent from a sender User to a receiving User.

`callback(err, messageObject)`	`addMessage(`
	`String senderEmail,`
	`String recipientEmail,`
	`String content,`
	`Date timeStamp,`
	`callback)`
	Search the database for the specific Message object that is from a sender User to a receiving User and appends the specified message to the array of messages tracked by the Message object and updates the database. The updated Message object is then passed back into the callback.

3.3.3.6 Comment

A Comment object tracks all the comments made by one User for one Exhibition.

Notable APIs

Callback	**Methods and Description**

No callback required.	`constructor(`
	`String userEmail,`
	`mongoose.Schema.ObjectId exhibitionKey,`
	`String comment, Date date)`
	Creates a new Comment JSON object and saves it internally.

`callback(err, commentObject)`	`saveComment(callback)` Saves the Comment JSON object stored internally into the database.

`callback(err, commentObject)`	`addCommentForExhibition(`
	`String userEmail,`
	`mongoose.Schema.ObjectId exhibitionKey,`
	`String comment,`
	`Date date,`
	`callback)`
	Adds a comment made by a specific User for the Exhibition with the specified Id into a Comment object. This Comment object is assumed to be existing before this function call.

`callback(err, Array <commentObject>)`	`getCommentsForExhibition(`
	`mongoose.Schema.ObjectId exhibitionKey,`
	`callback)`

	Retrieve all the Comment object made by all Users for a specific Exhibition. An Array of Comment Objects are passed into the callback.

3.3.4 Routes

The routes here refer to the server endpoints that are provided for the Views to use when attempting to get and set data from / into the database via an XMLHttpRequest. Each route calls one or two middleware functions - one if the route is mainly for retrieval of information, two if the route handles modification of user-related data. All routes typically return a HTTP response status, and the results in the form of a JSON object.

When returning object data from the database, an extractor function is used to change the key names of the object to be returned, as well as to hide sensitive information away (for example, the hashed password from the User object). The former means that the attributes of the final JSON object sent over will not correspond directly to the names of the Schema attributes.

For the routes with two middleware functions, the first middleware function enables a authentication and authorization check to be conducted to verify that the User is really who they state to be. The check is mainly done to ensure that an authenticated User does not modify the information of another User. This middleware function requires that a JsonWebToken be passed in in the request, which is generated by the '/auth' route upon successful verification of credentials at the beginning of each session.

It is important to note that the middleware function only checks for the authentication status of a User - it is up to the routes that use the middleware to do the authorization checks themselves.

Status Code - Response	Request Parameters	Middleware - Description
401 - if the supplied token is invalid in any way. nil - if the supplied token is successful.	`headers.authorization`	auth-check - a function that decodes a given JsonWebToken passed in through the HTTP request headers. If authentication was successful, it attaches the User email as the request parameter auth_user_email to the request object. This is to allow for the subsequent middleware function to authorize the request.

Therefore, the routes below that are similarly highlighted in red uses the auth-check Middleware function, and thus, require that the JsonWebToken to be additionally passed into the Request Parameters.

Additionally, there are 5 common HTTP status codes returned by the server endpoints described below. These are:

Status Code	**Short Description**
200	If the HTTP Request is successful, and there is some data to send back. The 'Response Deliverables' for each route assumes this status.
204	If the HTTP Request is successful, but there is no data to send back.
400	If the HTTP Request does not have all the parameters required by the route.
403	If the Authorization check fails (meaning that the supplied JsonWebToken contains information that matches one User, but the HTTP Request is attempting to modify another User's related information).
423	If the resource the HTTP request is trying to access / modify is locked.
500	If the server cannot complete the operation for some reason.

3.3.4.1 Auth

The auth routes handle the authentication of users when they login or sign up.

Note that all auth routes have a '/auth/' portion prefixed onto the final route url.

Routes

Response Deliverables	Request Parameters	Route - Request Type - Description
Boolean success *String message* *Token token* *Object userData*	String params.name, String params.email, String params.password String params.confirmPassword	'/signup' - POST - Validate the user inputs and create a new User object and a generate a token.
Boolean success *String message* Token token Object data	String params.eventName, String params.exhibitionName	'/login' - POST - Validate user inputs and authenticate user. Upon authentication, token is generated and returned.

3.3.4.2 user

The user routes handle database operations related to one or more Users.

Note that all user routes have a '/user/' portion prefixed onto the final route url.

Routes

Response Deliverables	**Request Parameters**	**Route - Request Type - Description**
A User with the requested userEmail, if found in the database.	`String params.userEmail`	'/get/profile' - GET - Gets information related to one User based on the supplied email.
Multiple Users with the requested userEmails. Only the userEmails which map to a User in the database will have their profiles in the end JSON.	`CSV String params.userEmails`	'/get/profiles' - GET - Gets information related to one or more Users based on the supplied emails. Removes duplicates.
The partial profiles of the bookmarked Users of one User.	`String params.email`	'/get/chat' - GET - Gets the partial profiles of the bookmarked Users of one User.
The modified User object.	`String body.userEmail, String body.userDescription`	'/post/profile/set/description' - POST - Sets the description of one User's profile.
The modified User object.	`String body.userEmail, String body.userProfilePicture`	'/post/profile/set/picture' - POST - Sets the profile picture of one User's profile.
The modified User object.	`String body.userEmail, String body.userNotification`	'/post/profile/set/notification' - POST - Sets the notification settings of one User's profile.
The modified User object.	`String body.userEmail, String body.userLink`	'/post/profile/add/link' - POST - Appends a given link to the User's list of links in their profile.
The modified User object.	`String body.userEmail, String body.userSkill`	'/post/profile/add/skill' - POST - Appends a given skill to the User's list of skills in their profile.
The modified User object.	`String body.userEmail, String`	'/post/profile/add/bUser' - POST - Appends another User's Id to the list of

	`body.bookmarkedUserId`	bookmarked Users on the User's profile. If the Id does not reference an existing User, nothing is done to the state.
The modified User object.	`String body.userEmail, String body.userLink`	'/post/profile/remove/link' - POST - Removes a specified link from the User's profile. If the link does not exist, nothing is done to the state.
The modified User object.	`String body.userEmail, String body.userSkill`	'/post/profile/remove/skill' - POST - Removes a specified skill from the User's profile. If the skill does not exist, nothing is done to the state.
The modified User object.	`String body.userEmail, String body.bookmarkedUserId`	'/post/profile/remove/bUser' - POST - Removes another User's Id from the list of bookmarked Users on the User's profile. If the Id does not reference an existing User, nothing is done to the state.
The matched User objects.	`CSV String body.userSkills`	'/post/search/skills' - POST - Returns all Users that have all the specified skills on their profile.
The modified User object.	`String body.userEmail, CSV String body.userLinks`	'/post/profile/set/links' - POST - Sets the list of links on the specified User profile.
The modified User object.	`String body.userEmail, CSV String body.userSkills`	'/post/profile/set/skills' - POST - Sets the list of skills on the specified User's profile.
The modified User object.	`String body.userEmail, CSV String body.bookmarkedUserIds`	'/post/profile/set/bUsers' - POST - Sets the list of other Users' Ids that represent Users to be bookmarked by this specified User. If the Id does not reference an existing User, it is not added to the list.

3.3.4.3 event

The event routes handle database operations related to one or more Events.

Note that all event routes have a '/event/' portion prefixed onto the final route url.

Routes

Response Deliverables	**Request Parameters**	**Route - Request Type - Description**
All Event objects.	`Nil`	'/get/allEvents' - GET - Gets all Event objects stored in the database.
One Event object.	`String params.eventName`	'/get/oneEvent' - GET - Gets one Event object stored in the database.
All matched Event objects.	`String params.tag`	'/get/searchTag' - GET - Gets all Event objects with the specified tag.
Modified Event object.	`String body.eventName, String body.map`	'/post/updateMap' - POST - Sets the map of the Event object.
Modified Event object.	`String body.eventName, String body.eventPicture`	'/post/updateEventPicture' - POST - Sets the display picture of the Event object.
All matched Event objects.	`String body.tags`	'/post/search/tags' - POST - Gets all Event objects with the specified tags.
Modified Event object.	`String body.eventName, String body.tags`	'/post/updateTags' - POST - Sets the tags for one Event object.

3.3.4.4 exhibition

The exhibition routes handle database operations related to one or more Exhibitions. In addition to the Exhibition object class, they also use the Event object class for checking data integrity.

Note that all exhibition routes have a '/exhibition/' portion prefixed onto the final route url.

Routes

Response Deliverables	**Request Parameters**	**Route - Request Type - Description**
All Exhibition objects.	`Nil`	'/get/allExhibitions' - GET - Gets all Exhibition objects stored in the database.
Matched Exhibition objects.	`String params.eventName`	'/get/oneEventExhibitions' - GET - Gets all Exhibition objects hosted under one Event.
One Exhibition object.	`String params.eventName, String params.exhibitionName`	'/get/oneExhibition' - GET - Gets one Exhibition object.
One Exhibition object.	`String params.exhibitionId`	'/get/oneExhibitionById' - GET - Gets one Exhibition object by its Id.
Matched Exhibition objects.	`CSV String body.tags`	'/post/search/tags' - POST - Gets all Exhibition objects that are each tagged with all specified tags.
Modified Exhibition object.	`String body.eventName, String body.exhibitionName, CSV String body.tags`	'/post/oneExhibition/set/tags' - POST - Sets the list of tags an Exhibition object has.

3.3.4.5 attendance

The attendance routes handle database operations related to one or more Attendances. In addition to the Attendance object class, they also use the User, Event and Exhibition object classes for checking and enforcing data integrity.

Note that all attendance routes have a '/attendance/' portion prefixed onto the final route url.

Routes

Response Deliverables	Request Parameters	Route - Request Type - Description
One Attendance object.	`String params.email, String params.id`	'/get/oneUserAttendance' - GET - Gets one Attendance object. The id parameter can take in either an Event id or an Exhibition id.
Matched Attendance objects.	`String params.email`	'/get/oneUserAttendances' - GET - Gets all Attendance objects of a specified User.
Matched Event and Exhibition objects.	`String params.email`	'/get/oneUserEventsAndExhibitions' - GET - Gets all the Events and Exhibitions a User is participating / has participated in.
Matched Exhibition objects.	`String params.email`	'/get/oneUserExhibitions' - GET - Gets all the Exhibitions a User is participating / has participated in.
Matched Event objects.	`String params.email`	'/get/oneUserEvents' - GET - Gets all Events a User is attending / has attended.
Matched Exhibition objects.	`String params.email, String params.eventName`	'/get/oneUserExhibitionsInEvent' - GET - Gets all Exhibitions a User has participated in under one Event.
Matched User objects.	`String params.id`	'/get/oneActivityAttendees' - GET - Gets all Users attending an Event / Exhibition. The id parameter will determine if the above query is w.r.t an Event or Exhibition.
Matched User objects.	`String params.eventName`	'/get/oneEventAttendees' - GET - Gets all Users attending a specified Event.
Matched User objects.	`String params.eventName, String`	'/get/oneExhibitionExhibitors' - GET - Gets all Users participating in a specified Exhibition.

	`params.exhibitionName`	
Matched User objects.	`String params.id`	'/get/oneEventExhibitors' - GET - Gets all Users participating in at least one Exhibition for a given Event. The id parameter here should only be an Event Id.
Matched User objects.	`String body.id, CSV String body.reasons`	'/post/search/activity/reasons' - POST - Searches for Users who are attending a specified Event or Exhibition for similar reasons. The id parameter here will automatically determine if the above query is w.r.t an Event or Exhibition.
Matched User objects.	`String body.id, CSV String body.reasons`	'/post/search/event/exhibitors/reasons' - POST - Searches for Users who are participating in an Event as an Exhibitor, for given reasons. The id parameter here should only be an Event id.
'Attendance Added!' or 'Attendance Removed!'	`String body.userEmail, String body.eventName`	'/post/oneEventAttendance' - POST - Toggles the User's Attendance for an Event.
Modified Attendance object.	`String body.userEmail, String body.id, CSV String body.reasons`	'/post/oneAttendanceReasons' - POST - Sets the reasons for the attendance of a User to either an Event or Exhibition. The id parameter here can automatically determine if the Id passed in is an Event or Exhibition Id.

3.3.4.6 comment

The comments routes handles the addition of new comments onto an exhibition page, and retrieval of comments from the database through the use of the Comment object class. In addition, it uses the Exhibition object class to check and enforce data integrity.

Note that all comment routes have a '/comment/' portion prefixed onto the final route url.

Routes

Response Deliverables	Request Parameters	Route - Request Type - Description
Matched Comment objects.	`String params.userEmail, String params.eventName, String params.exhibitionName`	'/get/userComments' - GET - Gets the Comment object that tracks the comments posted by one User to one Exhibition.
Matched Comment objects.	`String params.eventName, String params.exhibitionName`	'/get' - GET - Gets the Comment objects from all Users under one Exhibition.
'Added'	`String body.eventName, String body.exhibitionName, String body.userEmail, String body.comment`	'/post/newComment' - POST - Creates a new Comment object - to be used if the User has not commented on the Exhibition before.
'Added'	`String body.eventName, String body.exhibitionName, String body.userEmail, String body.comment`	'/post/addComment' - POST - Appends a comment to an existing Comment object - to be used if the User has commented on the Exhibition before.

3.3.4.7 messageSocket

The routing for the Messages is different from the other routers. It does not use http requests, instead, socket.io is utilised. We wanted the chat to be have live updates, hence we have decided to adopt socket.io into our application. Within the websocket functions, it uses methods from the Message object class.

It keeps track of the users who are currently online and their socketID(s). When they leave the chat page or close the tab, disconnect is then called which removes them from the list of users and socket ID..

Notable APIs

Callback	**Parameters**	**Socket function and description**
No callback required	`Socket` Sent to server when a new instance of Socket is made.	`io.on('connection', function (socket))` Obtain the new Socket connection from the client side.
`callback(socketID)`	`String userObject.userEmail`	`socket.on('new user', function (userObject, callback))` Waits on the client to emit 'new user' to store the socket ID linked to the User email.
`callback(err, listOfEmails)`	`String messageObject.userEmail`	`socket.on('get all emails involving user', (messageObject, callback))` Waits on client to emit 'get all emails involving user', calls the Message class to retrieve all related object and concatenate it with the User's bookmarked users and is passed into callback.

`callback(err, messageObject)`	`String messageObject.senderEmail`
	`String messageObject.recipientEmail`
	`Token token`
		`socket.on('get message', function (messageObject, token, callback))`
		Waits on client to emit 'get message'. It first verify if user is existing before retrieving all the message

`callback(isSuccessful)`	`String messageObject.senderEmail`
	`String messageObject.senderEmail`
	`String messageObject.content`
	`Token token`
		`socket.on('add message',function (messageObject, token, callback))`
		Waits on client to emit 'add message'. Calls Message object class and appends existing message object. If does not exist, a new Message object is

		created.
No callback required.	`Object messageObject` Client to take in parameter.	`socket.to(socketId).emit('refresh message', messageObject)` Emits a notification to a specific User's Socket by sending over the messageObject.
No callback required.	`Nil`	`socket.on('remove user', function())` Waits on client to call 'remove user'. The user's current socket ID will be removed from the tracked lists.
No callback required.	`Nil`	`socket.on('disconnect', function())` This is called when user closes the browser. It closes the entire socket connection. The user's current socket ID will be removed from the tracked lists.

3.3.5 STePs Converters

As mentioned at the start of section 3.3, our system has the capability to read of the STePs database and populate its own database. This process is not yet automated - the relevant gulp task needs to be run manually at this point of time in the project to invoke the transfer.

There are two main functions that bring in the information at varying degrees, but both use the same subroutines and function definitions defined in one script.

3.3.5.1 stepsConverterUtil

This script defines common functionality between the two converters. Upon importing this script, the script will define some functions that converts the actual data, as well as initializing two database connections. The functions are documented in the table below.

Notable APIs

Result	Function / Description
`callback(err)`	`upsertEvent(Object stepsEventObject):` Converts a stepsEventObject into an Event object. Specially searches for a non-empty entry under the stepsEventObject.promote.links array to set as the Event's picture.
`callback(err)`	`upsertUser(Object stepsUserObject)` Converts a stepsUserObject into a User object. Currently, all User passwords brought in with this function will be set to ''. Additional work must be done in order to generate and deliver proper passwords for new accounts via email.
`callback(err)`	`upsertModule(Object stepsModuleObject)` Extracts out Exhibition and Attendance objects, given a stepsModuleObject containing Projects. If a Project name is deemed to be invalid, it will not bring in the relevant information. For each exhibitor, both Event Attendance and Exhibition Attendance objects will be created upon attendance data insertion.
`callback(err)`	`upsertGuest(Object stepsGuestObject)` Extracts out User and Attendance objects, given a stepsGuestObject. As with `upsertGuest`, all User passwords are initialized to be ''. Additional work must be done in order to generate and deliver proper passwords for new accounts via email.

3.3.5.2 activeStepsConverter

This is one of the two main functions. This brings in only information related to the currently active STePs Event, upon execution. This function is meant to be the one used in the live updating of this application's database.

3.3.5.3 fullStepsConverter

This is the other of the two main converter functions. This brings in all information stored in the STePs database to our database, upon execution. This function is meant to be executed only at the system's initial startup.

4. Execution flow

In order to further understand how does the STePS Networking Module works, we will use the sequence diagram as shown in figure 3 below. Whenever a user gives an input, it will be taken in by the view component and calls the respective controller API. The controller will authenticate the user and then call the Model to check for further user verification and data processing. The Model will return the status, and necessary data if any. Using the returned input, the controller will control the view to change accordingly and call the Model for any additional information and the final result is what the user will see.

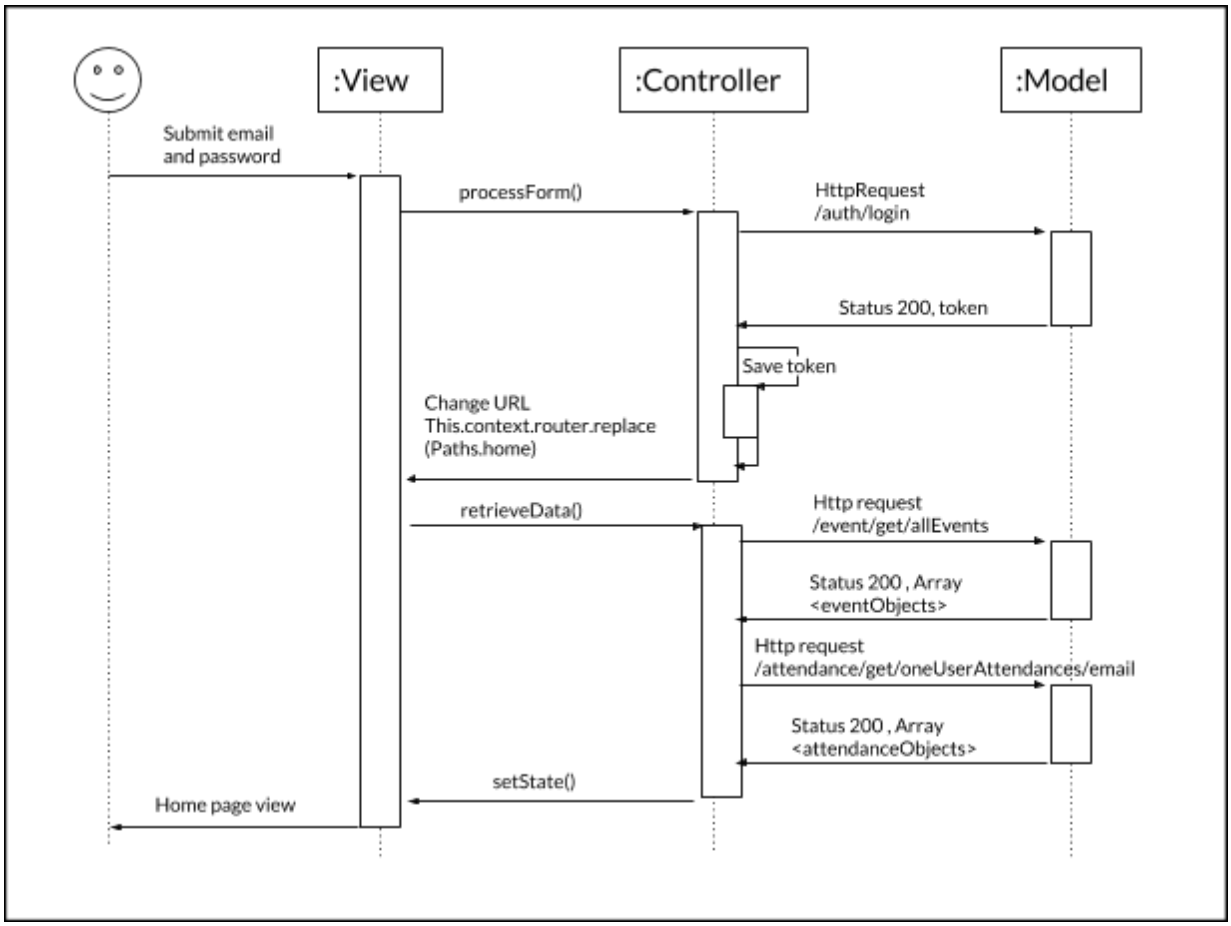

Figure 3: Sequence Diagram for User Login to Home page

5 Patterns and Principles

For our development, we separate our job-scope to client and server side development. Thus, we will address the methodology, patterns and principles differently. The frontend consist of the view and controller whereas the backend is the model.

5.1 Coding Standards

All code, regardless of client or server-side, should adhere and will be linted according to the Airbnb Javascript Style Guide found at https://github.com/airbnb/javascript, barring a few exceptions such as

- react/jsx-no-bind
- jsx-a11y/no-static-element-interactions
- eslint-disable array-callback-return
- jsx-a11y/no-static-element-interactions
- class-methods-use-this
- no-nested-ternary
- consistent-return
- No-unrestricted-syntax (for)

For linting, run `gulp lint` to generate the list of code violations on the console.

5.2 Client Side

Separation of Concerns
As shown in the architecture diagram (Section 2), each client-side component has it's own view and controller. The logic in each controller only relates to its own view. Should a component require the use of another component, such as Homepage requiring the search input component, it imports the component and renders it as whole. This is to reduce the dependency on one another and thus, each component is open-ended and can be used freely.

Interface Segregation Principle
Each UI component is segregated from one another. Components should not have any redundant methods or extractable components. Our example here is the Search component which is rendered twice and does not have a high dependency with the navbar or the homepage. Integrating the search together would bring cause redundancy and very 'fat' code.

Mediator Pattern
Only the Homepage adheres to this pattern. The Event and Collapsable View are takes props which includes a function to interact with the homepage. The function will then return the value to its sibling. The Event cannot interact with the Collapsable or vice versa.

5.3 Server Side

Facade
The object classes act as a facade for the database specific operations. This means that it is possible to switch database providers without affecting the route level, should the need arise.

6. Testing

6.1 Unit Testing

Unit testing was done for the Object classes. Testing is done here as to ensure that data is extracted and stored correctly into the database. Unit testing was done using Mocha as it allows for simple testing with asynchronous functions.

Each object class has it's own respective test files which are further divided into sections. Sections are created based on Create, Read, Update and delete (CRUD) components if any, (Message and Comment does not support delete functions). Each section are packaged into a `describe` structure, and the respective functions are tested in individual test case wrapped in an `it` structure.

In each section, hooks (before) are used to set up the database to populate it with predefined data and at the end of a section, hooks (after) are used again to wipe the database. Callbacks are also used, called done in our cases, to ensure that the function has properly executed before moving on. This is especially important for the hooks to ensure that there are valid data to work with. For some of the test cases, such as Attendance tests, it requires additional dependencies such as having valid Exhibition or Event Objects before being able to execute any testing.

In each test, object classes methods are used to extract, modify and store data. After which, `assert` is used to check if the output is as expected. If the assert fails, the test case will fail.

In order to run the unit test, simple follow instruction below:

1. Run the command window at the root of the repository
2. Run `gulp test` in the command line

6.2 Integration Testing

For integration testing, a bottom-up approach is first used. After ensuring that our unit tests are working and the client side is able to render data, we progressively test higher level of combination, such as client side retrieving data from server, and rendering them. Afterwards, testing was done for posting data from client side to server side. Each components will conduct their own testing with the server before any further testing.

At the current progress, with the application's completion, a top-down approach is now used to test the entire application. We have a set of scenarios to mimic users when they are using the application. You must ensure that all tested features are working when running through the test. These are also the scenario task that we provide to user for evaluation testing. These scenarios can includes:

- As an Employer, you have attended an event, and are interested in some of the projects and the members. You wish to contact them to offer any students interested in finding employment through chat.
    - Functions and features tested:
        - Event page
            - Extraction of data from database
        - Exhibition page
            - Extraction of data from database
        - Home page
            - marking reasons for attending
        - Matching
            - Correct matching is made
        - Profile
            - Extraction of data from database
        - Search
            - Skills
        - Chat
            - Live update

- As a student, you are participating as an exhibitor in STePS and are interested in finding employment/ job/ partners.
    - Functions and features tested:
        - Profile
            - Updating of skills
            - Updating of reasons for attendance
        - Exhibition page
            - Extraction of data from database

- You are interested in finding out about a specific event and the exhibitions inside the event, however you cannot remember the name, and decided to use the search functions. Leave comments for improvements for some of the exhibitions.
    - Functions and features tested:
        - Search
            - Event
            - Exhibition
            - Tags
        - Exhibition Page
            - Commenting

Appendix

Full list of API Table for object classes.

Users APIs

Callback	Methods and Descriptions

No callback required.	`Constructor(`
	`String userEmail,`
	`String userName,`
	`String userDescription,`
	`String userPassword,`
	`boolean willNotify,`
	`boolean isDeleted,`
	`String profilePic,`
	`Array<String> links,`
	`Array<String> skillsets,`
	`Array<mongoose.Schema.ObjectId>`
	`bookmarkedUsers)`
	Takes in parameters as shown above and creates a new User JSON.

`callback(err, userObject)`	`saveUser(callback)` Creates a new Mongoose Document using the User JSON and saves it into the database. It will return the newly saved User object in the callback.
`Callback(err, isExisting)`	`isValidUser(userEmail, callback)` Checks whether a specified User exists, and if it does, has it been marked as deleted. It will return true if User exist and is not marked as deleted.
`callback(err,` `Array<userObject>)`	`getAllUsers(callback)` Goes through database and retrieves a copy of all the User documents that are stored in the Database. An array of user Objects are passed into the callback.

`callback(err, userObject)`	`getUser(`
	`String userEmail,`
	`callback)`
	Goes through database to find a document with the matching email and returns it. It will return a single User Object.

`callback(err, userObject)`	`getUserByID(`
	`mongoose.Schema.ObjectId userId,`
	`callback)`
	Goes through database to find a document with the matching object ID and returns the single User Object.

`callback(err, Array <`
`userObject>)`
	`searchUsersBySkills(`
	`String skillToBeSearched,`
	`callback)`

	Finds users that has the specific skill.
`callback(err, Array < userObject>)`	`searchUsersByMultipleSkills( Array <String> skillToBeSearched, callback)` Finds users that has all the specific skills specified.
`callback(err, userObject)`	`updateUserDescription( String userEmail, String description, callback)` Finds the matching User document, updates the description and saves it back into database. The updated User object is passed into the callback.
`callback(err, userObject)`	`updateUserNotification( String userEmail, boolean willNotify, callback)` Finds the matching User document, updates if the users want to enable notification and saves it back into database. The updated User object is passed into the callback.
`callback(err, userObject)`	`updateUserProfilePicture( String userEmail, String profilePicURL, callback)` Finds the matching User document, updates the string used to identify the profile picture of users and saves it back into database. The updated User object is passed into the callback.
`callback(err, userObject)`	`addLinkToUserLinks( String userEmail, String link, callback)` Finds the matching User document, adds a new link and saves it back into database. The updated User object is passed into the callback.
`callback(err, userObject)`	`removeLinkToUserLinks( String userEmail, String link, callback)` Finds the matching User document, removes specific link from the user and saves it back into database. The updated User object is passed into the callback.
`callback(err, userObject)`	`setLinksForUser( String userEmail, Array <String> links, callback)` Finds the matching User document, sets the array of links for the user and saves it back into database. The updated User object is passed into the callback.
`callback(err, userObject)`	`addSkillToUserSkills(`

	`String userEmail,`
	`String skill,`
	`callback)`
	Finds the matching User document, adds a new skill for the user and saves it back into database. The updated User object is passed into the callback.

`callback(err, userObject)`	`removeSkillFromUser(`
	`String userEmail,`
	`String skill,`
	`callback)`
	Finds the matching User document, removes specific skill from the user and saves it back into database. The updated User object is passed into the callback.

`callback(err, userObject)`	`setSkillsForUser(`
	`String userEmail,`
	`Array <String> skills,`
	`callback)`
	Finds the matching User document, sets the array of skills for the user and saves it back into database. The updated User object is passed into the callback.

`callback(err, userObject)`	`addBookmarkedUserForUser(`
	`String userEmail,`
	`mongoose.Schema.ObjectId bookmarkedUserId,`
	`callback)`
	Finds the matching User document, adds a new object ID representing another existing user into the current user's bookmarks. The user is then updated in the database and passed into the callback.

`callback(err, userObject)`	`removeBookmarkedUserFromUser(`
	`String userEmail,`
	`mongoose.Schema.ObjectId bookmarkedUserId,`
	`callback)`
	Finds the matching User document, removes the object ID representing another existing user from the current user's bookmarks. The user is then updated in the database and passed into the callback.

`callback(err, userObject)`	`setBookmarksForUser(`
	`String userEmail,`
	`Array<mongoose.Schema.ObjectId>,`
	`bookmarkedUserIds,`
	`callback)`
	Finds the matching User document, sets the array of bookmark users and saves it back into database. The updated User object is passed into the callback.

`callback(err, userObject)`	`updateUser(`
	`String email,`
	`String name,`
	`String description,`
	`String password,`
	`boolean willNotify,`
	`boolean isDeleted,`
	`String profilePic,`
	`Array <String> links,`

	`Array <String> skillSets,`
	`Array <mongoose.Schema.ObjectId>`
	`bookmarkedUserIds,`
	`callback)`
	This method overwrites the entire User document and saves it back into the database. The updated user object is passed into the callback.

`callback(err, userObject)`	`setUserAsDeleted(`
	`String userEmail,`
	`boolean isDeleted,`
	`callback)`
	This method sets whether the User is marked as deleted. They will not be deleted from the database, but will be considered as an invalid User. An invalid User is one whose information cannot be changed, other than through automated processes. This is to minimize data integrity issues.

Event APIs

Callback	Methods and Descriptions

No callback required.	`constructor(`
	`String eventName,`
	`String eventDescription,`
	`Date startDate,`
	`Date endDate,`
	`String location,`
	`String map,`
	`String eventPicture,`
	`Array <String> tags)`
	Creates a new Event JSON object and stores internally.

`callback(err, eventObject)`	`saveEvent(callback)` Save the Event JSON object into a Mongoose Model Document and saves it into the database. The saved Event object is then passed into the callback.

`callback(err, isExisting)`	`isExistingEvent(`
	`String eventName,`
	`callback)`
	Checks if the Event exists in the database.

`callback(err, eventObject)`	`getEvent(`
	`String eventName,`
	`callback)`
	Search through the database to find a matching Event object based on the event name specified. The search result is then passed back into the callback.

`callback(err, eventObject)`	`getEventById(`
	`mongoose.Schema.ObjectId eventId,`
	`callback)`
	Search through the database to find a matching Event object based on the object ID specified. The search result is then passed back into the callback.

`callback(err, Array <eventObject>)`	`getAllEvents(callback)` Retrieves all the Event objects found in the database and passed as an array of Event Object into the callback.

`callback(err, Array <eventObject>)`	`searchEventsByTag(`
	`String tag,`
	`callback)`
	Search the database to find all Event objects that contains the specific tag.

`callback(err, Array <eventObject>)`	`searchEventsByTags(`
	`Array <String> tags,`
	`callback)`
	Search the database to find all Event objects that contains all the tags.

`callback(err, eventObject)`	`updateEventMap(`
	`String eventName,`
	`String map,`
	`callback)`
	Search the database to find the matching event and updates the event_map attribute. The updated Event object is then passed into the callback.

`callback(err, eventObject)`	`updateEventPicture(`
	`String eventName,`
	`String eventPicture,`
	`callback)`
	Search the database to find the matching Event and updates the event_picture attribute. The updated Event object is then passed into the callback.

`callback(err, eventObject)`	`updateEventTag(`
	`String eventName,`
	`Array <String> tags,`
	`callback)`
	Search the database to find the matching event and updates the tags attribute. The updated Event object is then passed into the callback.

`callback(err, eventObject)`	`deleteEvent(`
	`String eventName,`
	`callback)`
	Search the database to find the matching Event and removes it from the database. Does not enforce data integrity by removing any Exhibition or Attendance objects referencing it.

Exhibition APIs

Callback	Method and Descriptions

No callback required.	`constructor(`
	`String exhibitionName,`
	`String exhibitionDescription,`
	`String eventName,`
	`String posterURL,`
	`Array<String> images,`
	`Array<String> videos,`
	`String website,`
	`Array<String> tags)`
	Creates a new Exhibition JSON and stores it internally.

`callback(err, result)`	`saveExhibition(callback)` Save the Exhibition JSON object into a Mongoose Model Document and saves it into the database. The saved Exhibition object is then passed into the callback. Note that this does not validate the supplied Event name.

`callback(err, exhibitionObject)`	`isExistingExhibition(`
	`String eventName,`
	`String exhibitionName,`
	`callback)`
	Checks whether the Exhibition exists within the database.

`callback(err, exhibitionObject)`	`getExhibition(`
	`String eventName,`
	`String exhibitionName,`
	`callback)`
	Retrieve an Exhibition object from the database.

`callback(err, exhibitionObject)`	`getExhibitionById(`
	`mongoose.Schema.ObjectId exhibitionId,`
	`callback)`
	Retrieve a specific Exhibition object from the database, using its Id.

`callback(err, exhibitionObject)`	`getAllExhibitions(callback)` Retrieve all Exhibition objects from the database.

`callback(err, Array<exhibitionObjects>)`	`searchExhibitionsByTag(`
	`String tag,`
	`callback)`
	Retrieve all the Exhibitions tagged with the specified String.

`callback(err, Array <exhibitionObjects>)`	`searchExhibitionsByTags(`
	`Array<String> tags,`
	`callback)`
	Retrieve all Exhibitions tagged with **all** the specified Strings.

`callback(err, Array<exhibitionObjects>)`	`searchExhibitionsByEvent(`
	`String eventName,`
	`callback)`
	Retrieve all Exhibitions hosted under a single Event.

`callback(err, exhibitionObject)`	`setTagsForExhibition(`
	`String eventName,`
	`String exhibitionName,`
	`Array<String> tags,`
	`callback)`
	Tag an Exhibition with the numerous Strings specified in the 'tags' argument.

`callback(err, exhibitionObject)`	`updateExhibition(`
	`String exhibitionName,`
	`String exhibitionDescription,`
	`String eventName,`
	`String posterURL,`
	`Array<String> images,`
	`Array<String> videos,`
	`String website,`
	`Array<String> tags)`
	Updates all information of an Exhibition object, and saves the changes to the database.

`callback(err)`	`deleteExhibition(`
	`String exhibitionName,`
	`callback)`
	Delete an Exhibition object from the database. Does not enforce data integrity by removing any Attendance or Comment objects referencing it.

Attendance APIs

Callback	Methods and Descriptions
No callback required.	`constructor( String userEmail, mongoose.Schema.ObjectId attendanceKey, String attendanceType, Array <String> reason)` Creates a new Attendance JSON object and stores internally.
`callback(err, attendanceObject)`	`saveAttendance(callback)` Saves the Attendance JSON object into a Mongoose Model Document and saves it into the database.
`callback(err, attendanceObject)`	`getAttendance( String userEmail, mongoose.Schema.ObjectId attendanceKey, callback)` Retrieve a specific Attendance object from the database using the userEmail and attendanceKey.
`callback(err, Array <attendanceObject>)`	`getAllAttendances(callback)` Retrieve all the Attendance object from the database. An array of Attendance objects are passed into the callback.
`callback(err, Array <attendanceObject>)`	`searchAttendancesByUser( String userEmail, callback)` Retrieve all the Attendance object by a specific User. An Array Attendance objects are passed into the callback.
`callback(err, Array <attendanceObject>)`	`searchAttendancesByKey( mongoose.Schema.ObjectId attendanceKey, callback)` Retrieve all the Attendance object involving a certain Event or Exhibition.
`callback(err, attendanceObject)`	`searchAttendanceByUserAndKey( String userEmail, mongoose.Schema.ObjectId attendanceKey, callback)` Search for a specific Attendance object by a User and the Event or Exhibition.
`callback(err, Array <attendanceObject>)`	`searchAttendancesByReason( String reason, callback)` Searches for a specific reason within the database regardless of Event or Exhibition and returns an array to callback.
`callback(err, Array <attendanceObject>)`	`searchAttendancesByKeyAndReason(mongoos`

	e.Schema.ObjectId attendanceKey, String reason, callback) Searches for a specific reason within the database within a certain Event or Exhibition and returns an array to callback.
callback(err, Array <attendanceObject>)	searchAttendancesByKeyAndReasons(mongoose.Schema.ObjectId attendanceKey, Array<String> reasons, callback) Searches the database to Attendance object within an Event or Exhibition who have all the specified reasons for attending.
callback(err, attendanceObject)	updateReason(String userEmail, mongoose.Schema.ObjectId attendanceKey, String reason, callback) Search through the database to locate a specific Attendance object and updates the reason for attending. The updated Attendance Object is then returned in the callback.
callback(err, attendanceObject)	deleteAttendance(String userEmail, mongoose.Schema.ObjectId attendanceKey, callback) Removes a specific Attendance Object from the database

Message APIs

Callback	Methods and Descriptions

No callbacks required	`constructor(`
	`String senderEmail,`
	`String recipientEmail,`
	`String content,`
	`Date timeStamp)`
	Creates a new Message JSON object and stores internally.

`callback(err, messageObject)`	`saveMessage(callback)` Saves the Message JSON stored internally into the database.

`callback(err, Array <messageObject>)`	`getMessagesToUser(`
	`String recipientEmail,`
	`callback)`
	Search the database for all the messages send to a specific user.

`callback(err, Array <messageObject>)`	`getMessagesFromUser(`
	`String senderEmail,`
	`callback)`
	Search the database for all the messages that were sent by a specific user.

`callback (err, Array <String> listOfEmails)`	`getEmailsInvolvingUser(`
	`String userEmail,`
	`callback)`
	Search the database for all Message objects that involves a specific user.

`callback(err, messageObject)`	`getConversation(`
	`String senderEmail,`
	`String recipientEmail,`
	`callback)`
	Search the database for the specific Message object that is sent from a sender User to a receiving User.

`callback(err, messageObject)`	`addMessage(`
	`String senderEmail,`
	`String recipientEmail,`
	`String content,`
	`Date timeStamp,`
	`callback)`
	Search the database for the specific Message object that is from a sender User to a receiving User and appends the specified message to the array of messages tracked by the Message object and updates the database. The updated Message object is then passed back into the callback.

Comment APIs

Callback	Methods and Description
No callback required.	`constructor( String userEmail, mongoose.Schema.ObjectId exhibitionKey, String comment, Date date)` Creates a new Comment JSON object and saves it internally.
`callback(err, commentObject)`	`saveComment(callback)` Saves the Comment JSON object stored internally into the database.
`callback(err, commentObject)`	`addCommentForExhibition( String userEmail, mongoose.Schema.ObjectId exhibitionKey, String comment, Date date, callback)` Adds a comment made by a specific User for the Exhibition with the specified Id into a Comment object. This Comment object is assumed to be existing before this function call.
`callback(err, commentObject)`	`getUserCommentsForExhibition( String userEmail, mongoose.Schema.ObjectId exhibitionKey, callback)` Retrieve the Comment object made by a User for a specific Exhibition. An Array of Comment Objects are passed into the callback.
`callback(err, Array <commentObject>)`	`getCommentsForExhibition( mongoose.Schema.ObjectId exhibitionKey, callback)` Retrieve all the Comment object made by all Users for a specific Exhibition. An Array of Comment Objects are passed into the callback.
`callback(err, commentObject)`	`clearCommentsForExhibition( mongoose.Schema.ObjectId exhibitionKey, callback)` Removes all the Comment object made to a specific Exhibition.