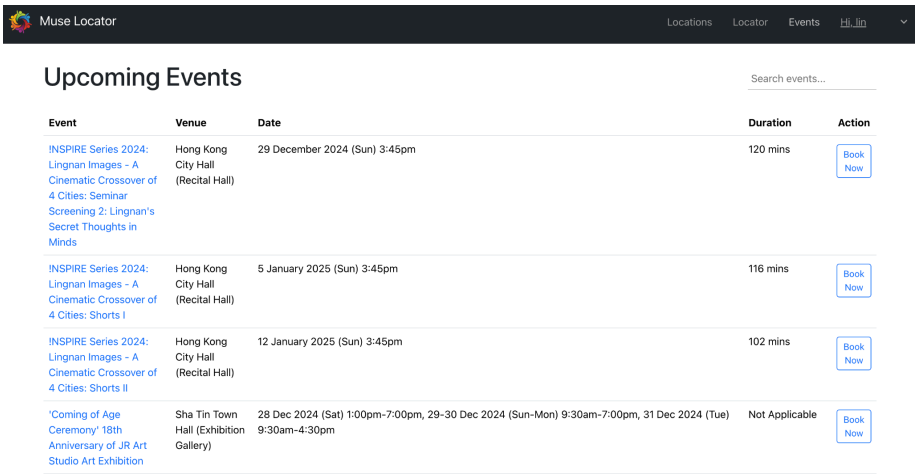
CSCI2720 Project

Group 26: Muse Locator

Member	Student ID
LIN Yi	1155232784
MANAV Suryansh	1155156662
MUI Ka Shun	1155194765
PARK Sunghyun	1155167933
RAO Penghao	1155191490

Table of Contents

1. Abstract



Muse Locator is a directory and booking system that simplifies the process of searching for cultural events in Hong Kong. It fetches comprehensive event and venue data from official government sources and presents it to users clearly and interactively that suits their needs, whether they are browsing nearby activities or have a specific one in mind. The app also features event booking and location bookmarking functions that allow users to access their favorite activities and venues at a glance.

2. Methodologies

A. File introduction

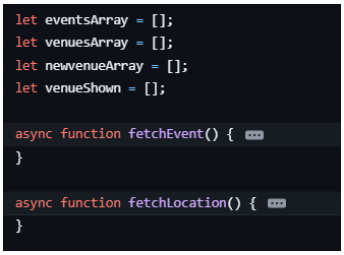
- **backend**
  i. index.js: To be run on Node, specifies server and routes

- **backend/models:** Define Schemas (Booking, Event, Location, User).
- **backend/routes**
    i. auth.js: Define the hashing algorithm to improve the password security and do the login logic in the backend side
    ii. booking.js, event.js, location.js, user.js: Define Schemas (Booking, Event, Location, User).
- **backend/scripts**
    i. addCommentsField.js
- **backend/services**
    i. dataFetcher.js: For fetching data from API and data pre-processing
- **client/public**
    i. index.html: Links to Bootstrap and CSS stylesheet
- **client/src/assets**: contains logo.png for use on the login page and app navbar
- **client/src**
    i. App.css: Defines basic columns and padding
    ii. App.js: Handles login and logout, and defines routes
    iii. index.css: Defines font styles used throughout the app
    iv. index.js: Renders app
- **client/src/components**
    i. Login.js: Structure of login page, as well as logic for checking password/user validity
    ii. Locator.js: Structure of Locator page using Google Maps API, and defines the function to fetch venue information from server
    iii. Locations.js: Structure of Locations page and sorting, searching and favorite functions
    iv. LocationDetail.js: Structure of individual venue view, with location information, map and comment form
    v. Header.js: Structure of navbar and links to different tabs
    vi. Favorites.js: Structure of Favorites page and function to remove favorites
    vii. Footer.js
    viii. Events.js: Structure of Events page and search and booking functions
    ix. EventDetail.js: Structure of individual event view, displaying date, venue, duration, presenter and description. Implementation of booking and map.
    x. Bookings.js: Structure of Bookings page, displaying information on booked venues and booking times. Implements function to cancel the booking.
    xi. AdminDashboard.js: Structure of Admin dashboard page, rendering user, event, and location management tabs.
    xii. AdminDashboard.css, Login.css, LocationDetail.css, Favorites.css, Bookings.css

- **client/src/components/admin**
    i. ManageUsers.js: Structure of user management page. Implementation of CRUD actions to the user data.
    ii. ManageLocations.js: Structure of location management page. Implementation of find locations and list locations.
    iii. ManageEvents.js: Structure of event management page. Implementation of CRUD actions to the event data.
    iv. AdminForms.css: Styling of admin forms.
- **client/src/components/common**
    i. Toast.js
    ii. Toast.css

B. Pre-processing of dataset

In backend/services/dataFetcher.js, there are two functions for fetching data from API, which are fetchEvent() and fetchLocation() respectively and four arrays were created for storing data downloaded from API and saving to MongoDB.



In fetchEvent(), new event data will be downloaded from the API [1] each time the user login. Therefore, the Event.deleteMany() function is placed at the first line of the function to delete existing event data in the MongoDB. Then we use Axios to get the event data in XML format from API and turn it into JSON format using xml2js. After retrieving the event data, we will save each event as an object into eventsArray first. The event object will have several properties e.g. event id, title, date, duration, etc. Some properties will be saved as 'Not Applicable' because they are missing in the API. Finally, Event.insertMany(eventsArray) to save events into MongoDB.

```
async function fetchEvent() {
    try {
        await Event.deleteMany();
        const response = await axios.get('https://api.data.gov.hk/v1/historical-archive/get-file?url=https%3A%2F%2Fwww.lcsd.gov.hk%2Fdatagovhk%2Fevent%2f
        const xmlText = response.data;

        await new Promise((resolve, reject) => {
            xml2js.parseString(xmlText, { explicitArray: false }, (err, result) => {
                if (err) {
                    reject(new Error('Error parsing XML: ' + err.message));
                }

                const items = result.events.event;
                eventsArray = Array.from(items).map(item => ({
                    eventid: item.$.id,
                    title: item.titlee,
                    date: item.predateE,
                    duration: item.progtimee || 'Not Applicable',
                    venue: item.venueid,
                    descre: item.desce || 'Not Applicable',
                    pre: item.presenterorge || 'Not Applicable',
                    eventID: item.$.id + '_'
                }))
                resolve();
            });
        });

        await Event.insertMany(eventsArray);
```

In fetchLocation(), we utilize the same process to delete existing location data and retrieve the data using axios and xml2js from API [2]. The retrieved location data will be stored as an object into venuesArray first with properties venue id, location, latitude, longitude and count. Because some location data from the API are missing latitude and longitude, the value of latitude and longitude will be stored as 'Not Applicable'.

```
try {
    await Location.deleteMany();
    const response = await axios.get('https://api.data.gov.hk/v1/historical-archive/get-file?url=https%3A%2F%2Fwww.lcsd.gov.hk%2Fdatagovhk%2Fevent%2Fvenues.xml&time=20241204-1325');
    const xmlText = response.data;

    await new Promise((resolve, reject) => {
        xml2js.parseString(xmlText, { explicitArray: false }, (err, result) => {
            if (err) {
                reject(new Error('Error parsing XML: ' + err.message));
            }

            const items = result.venues.venue;
            venuesArray = Array.from(items).map(item => ({
                venueid: item.$.id,
                location: item.venuee,
                latitude: item.latitude || 'Not Applicable',
                longitude: item.longitude || 'Not Applicable',
                count: 0
            }));
            resolve();
        });
    });
```

Next, we will filter the venue lacking latitude and longitude and store the location with latitude and longitude into a new array called newvenueArray.

```
// Process venues and events
newvenueArray = venuesArray.filter(venue => venue.latitude !== 'Not Applicable');
```

Then, we will count the events in each venue.

```
// Count events per venue
newvenueArray.forEach(venue => {
    venue.count = eventsArray.filter(event => event.venue === venue.venueid).length;
});
```

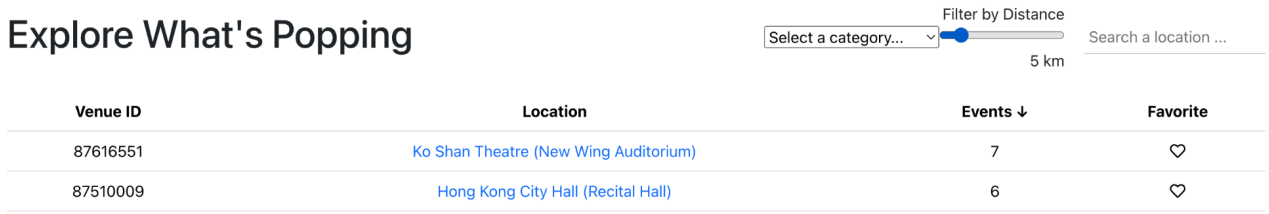
After that, we keep the venues with 3 or more events, randomize the venue and slice the array into 10 venues. Finally, we save this array into MongoDB and drop events with venue IDs not in the saved locations.

```
// Filter venues with 3 or more events
venueShown = newvenueArray.filter(venue => venue.count >= 3)
    .sort(() => 0.5 - Math.random())
    .slice(0, 10);

// Save venues
await Location.insertMany(venueShown);

// Drop events with venue IDs not in the saved locations
const venueIds = venueShown.map(venue => venue.venueid);
await Event.deleteMany({ venue: { $nin: venueIds } });
```

C. Required actions

I. User Actions

LocationTable.js fulfills User Actions 1, 3 and 7. useEffect() fetches the current user's favorites from the backend, as well as a list of venues. It outputs a filtered table of venues that i) fall within a specified radius from the user's location, set by the distance slider; ii) contain any sequence of characters that match the string input in the search bar; and iii) belong to a selected category from a drop-down menu.



We use navigator to obtain the user's current location, which is run through calculateDistance() along with the coordinates of each fetched venue to obtain a value distanceToLocation. Each instance of distanceToLocation is saved in an array "distances", from which the maximum and minimum values are used as endpoints for the slider. This ensures the slider is sensitive enough to filter locations effectively regardless of the user's current location. When the user types in the search bar, the searchTerm is set to the input. useEffect() updates the filteredLocations array based on these two parameters.

The category of venues is extracted from the venue names, which are fetched from the server in useEffect(). The text within parantheses (e.g. Recital Hall) is assigned as a new object in the categories array. The final categories array after this process can be viewed in the drop-down menu.

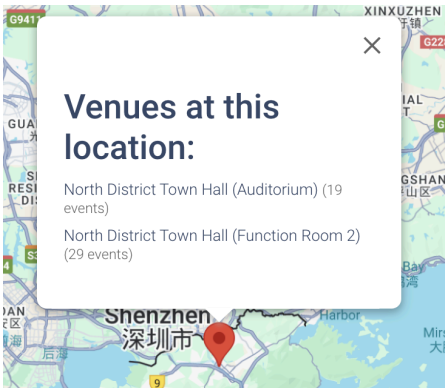
The filtered table of venues can be sorted by their number of events, in ascending or descending order, by toggling the arrows at the "Events" column. handleSort() defines the sorting order and sets the array of venues accordingly. Users also add venues to favorites from the LocationTable by clicking the heart icon.

toggleFavourite() fetches the current user's data and uses PUT to update the favourites list with the selected venue. This is reflected in the server and passed on to Favorites.

Locator.js fulfills User Action 2. In the useEffect(), we fetch venues from the backend to obtain the coordinates required to generate pins for each of the 10 locations fetched. Next in initMap(), we generate a blank map with specified center in Hong Kong, using the API Key obtained from Google Cloud Console.

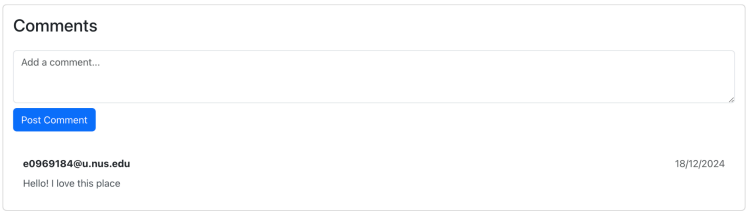
```
useEffect(() => {
    // Load Google Maps script
    const googleMapsScript = document.createElement("script");
    googleMapsScript.src = `https://maps.googleapis.com/maps/api/js?key=AIzaSyC6SJqInCt0heffya0W622QSkiaoXDHKFA`;
    googleMapsScript.async = true;
    googleMapsScript.onload = initMap;
    document.body.appendChild(googleMapsScript);

    function initMap() {
        const map = new window.google.maps.Map(document.getElementById("map"), {
            center: { lat: 22.444477, lng: 114.165562 }, // Hong Kong center
            zoom: 11,
        });
```

Markers are then created for each fetched location with the corresponding coordinates and title. If multiple venues are fetched at the same location, e.g. two function rooms in the same town hall, we group them together and display only one marker. We define the content of the info windows to display the names of all venues at each marker, which contain hyperlinks to the individual LocationDetail pages. We then add event listeners allowing users to view the info windows by mousing over any map marker, and close them by clicking the "X" button. This enables them to click venue names for more information and reduce unnecessary clutter from the map.

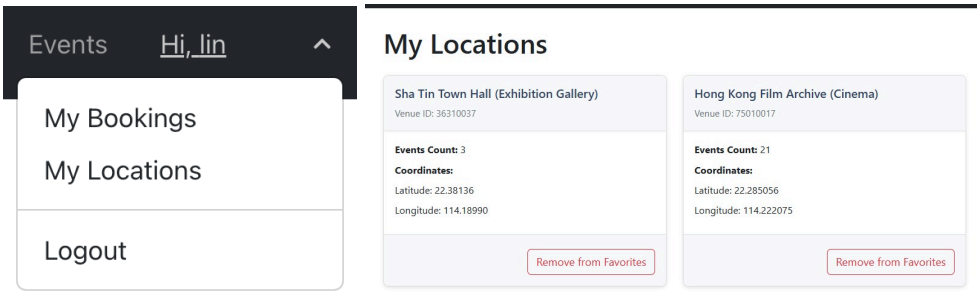



```
const content = `
    <div style="padding: 10px;">
        <h3 style="margin-bottom: 10px; color: #344767;">Venues at this location:</h3>
        <ul style="list-style-type: none; padding-left: 0;">
            ${venuesAtLocation.map(venue => `
                <li style="margin-bottom: 8px;">
                    <a href="/location/${venue.location}" style="text-decoration: none; color: #344767;">
                        ${venue.location}
                        <span style="color: #666; font-size: 0.9em;">
                            (${venue.count} events)
                        </span>
                    </a>
                </li>
            `).join('')}
        </ul>
    </div>
`;
```

LocationDetail.js fulfills User Action 4. useEffect() fetches location details and comment data from the server. initMap() generates a map with a marker on the specific venue's coordinates, which is also the center. The page includes a comment form that takes in text as input. Comments are handled using handleCommentSubmit(), which utilizes axios to post comments to the server along with the commenter's email. Users can see earlier comments stored in local storage.



FavoritesTable.js fulfills User Action 5. The table is accessible from the drop-down menu on the user dashboard, through the link “My Locations”. useEffect() fetches the location IDs that have been favourited by the current user, as well as the list of all locations. This list is then filtered to contain only the favourites, which are displayed as location cards on the app. Clicking “Remove From Favourites” on a location card executes handleRemoveFavourite(), which removes it from the page and the stored list.



Header.js and App.js fulfil User Action 6. The username is visible on the top right corner, with logout button being expandable from the drop-down menu and executing handleLogout() in Header.js. In App.js, handleLogout() sets the User to null and removes the current user’s token and email from the local storage.

II. Admin Actions

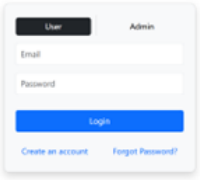
ManageUsers.js performs CRUD actions to the user data. handleCreateUser() sends a request with user data in the request body. If the email in the request body already exists, a response with ‘User already exists’ will be returned. After successfully creating a user, corresponding messages and created user data will be sent in response. hanldeListUsers() sends requests to user.js and retrieves every User data. Then it sets a corresponding message, listing retrieved data in admin-section-content. handleLoadUser() and handleUpdateUser() are implemented to update the user data. handleLoadUser() sends a request with the user’s email and retrieves corresponding user data. Data will be displayed, and user data can be updated by clicking the “Update User” button. handleDeleteUser() also sends a request with the user's email to be deleted by clicking the “Delete User” button. Then user.js will send the response with corresponding message and status.

ManageEvents.js performs CRUD actions to the event data similar to ManageUsers.js. It sends a request to event.js for executing functions. handleCreateEvent() uses POST method with new event data in the request body. Function can be executed by clicking the “Create Event” button after every input box is filled in. It will also retrieve created event data with corresponding messages through the response. handleListEvents() sends a request to find every event data in the database, and will display the whole list. To update event data, it uses the same mechanism as updating user data. handleLoadEvent() will send the request with event id to be loaded, and data can be modified after the current information of the event is displayed. After the data is changed, handleUpdateEvent() will be executed by the interaction with the “Update Event” button. Lastly, handleDeleteEvent() performs exactly the same as deleting user data. URL with event id to be deleted will be sent, and the correlated message and status will be returned.

Additionally, ManageLocations.js performs CRUD actions to the location data. Listing function is only embedded in the UI. handleListLocations() will list every location data stored in the database.

III. Non-user Actions

This works based on the App.js file in the client folder and the auth.js file in the backend side folder. We create a login page on the client side.



In this login page, users and admins can only log in with the correct username and password. We have already prepared the default user name, admin name, user password and admin password for you in README. Only authenticated users have access to the app's contents. You can use the default username and password to log in. And also you can create an account before accessing the user dashboard. Here, when you click the "Create account" button, the app will collect your information and help you to build the account. Just follow the instructions, then you can create an account successfully. Then visitors can log in with the account you create to log in, which will direct you to the user part.

D. Extra features

1, Security.

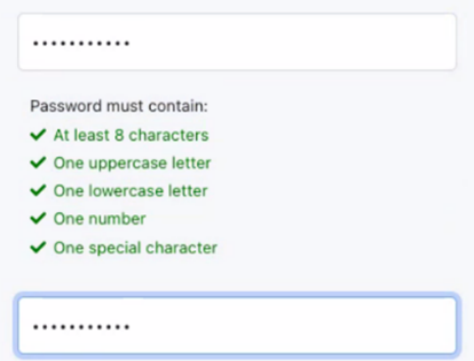
This mainly focuses on the auth.js file. Firstly, we perform the hashing algorithm here to improve the security level of our password. This converts the input password into a fixed-size string, typically a sequence of numbers and letters, instead of storing the password directly. This string output is the hash code. Here we use a package "bcrypt" to help implement this job.

```
// Hash password
const hashedPassword = await bcrypt.hash(password, 10);
```

Hence, when we log in, the password we input should be compared with the hashed one.

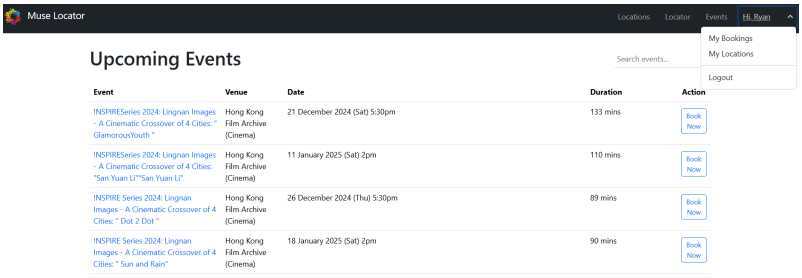
```
// Check if user exists and password matches
if (user && await bcrypt.compare(password, user.password)) {
    req.session.user = user; // Store user info in session
```

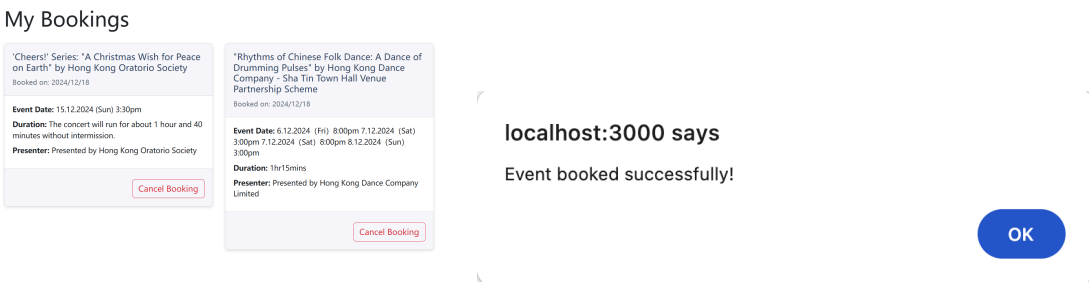
Another part about security here is that when the visitor creates the account, our app requires a strong security password, which should obey some rules about creating a valid password. Only if you fulfill all the requirements can you create a user account with a valid password. The requirements are shown below. Also, when you try to create a password in our app, you will see them.



2. Event page

We also have a page for displaying the upcoming event. The event table has a brief description of the event e.g Event name, Venue, Date, Duration. The event at the same location will be displayed together. When clicking on the event, a detailed description and a map of venue of the event will be shown. Moreover, there is a book button that when you click ‘book’, the booked event will be shown in ‘My Bookings ’ and the system will show a success prompt. When you click the “My Bookings” button, the events that you booked will be shown. You can also cancel the book here. If a user clicks on the “Book Now” button of an event that has already been booked, the system will prompt that the booking is unsuccessful.





Schema Design

User Schema:

```
const userSchema = new mongoose.Schema({
    email: { type: String, required: true, unique: true },
    password: { type: String, required: true },
    firstName: { type: String, required: true },
    lastName: { type: String, required: true },
    role: { type: String, enum: ['user', 'admin'], default: 'user' },
    locations: { type: Array, default: [] }
});
```

Event Schema:

```
const eventSchema = new mongoose.Schema({
    eventid: { type: String, required: true },
    title: { type: String, required: true },
    venue: { type: String, required: true },
    date: { type: String, required: true },
    duration: { type: String, required: true },
    descre: { type: String, required: true },
    pre: { type: String, required: true },
    eventID: { type: String, required: false, default: null }
});
```

Location Schema:

```
const commentSchema = new mongoose.Schema({
    userId: {type: String,required: true},
    text: {type: String,required: true},
    timestamp: {type: Date, default: Date.now}
});

const locationSchema = new mongoose.Schema({
    venueid: {type: String, required: true,unique: true},
    location: {type: String,required: true},
    latitude: {type: String,required: true},
    longitude: {type: String,required: true},
    count: {type: Number, required: true},
    comments: [commentSchema]
});
```

Booking Schema: (for booking event):

```
const bookingSchema = new mongoose.Schema({
    eventId: {type: String,required: true,ref: 'Event'},
    userEmail: {type: String,required: true,ref: 'User'},
    bookingDate: {type: Date,default: Date.now},
    status: {type: String,enum: ['active', 'cancelled'],default: 'active'}
});
```

3. References

[1] “Cultural Programmes - Programme information | DATA.GOV.HK,” *Data.gov.hk*, 2024. https://data.gov.hk/en-data/dataset/hk-lcsd-event-event-cultural/resource/7a364aee-b1d0-4f89-959c-8819eae94257 (accessed Dec. 18, 2024).

[2] “Cultural Programmes - Venues of programmes | DATA.GOV.HK,” *Data.gov.hk*, 2024. https://data.gov.hk/en-data/dataset/hk-lcsd-event-event-cultural/resource/8f5802f9-d9d2-4d1e-9fb9-4fbc6eaa2bdf (accessed Dec. 18, 2024).

[3] Tsang, S. C. (2024). *Lab 9: MongoDB (by Mongoose)* [PowerPoint slides].

[4] Tsang, S. C. (2024). *Lab 8: Node.js and Express* [PowerPoint slides].

[5] Tsang, S. C. (2024). *Lab 7: React Router* [PowerPoint slides].

[6] Tsang, S. C. (2024). *Lab 6: Basic React* [PowerPoint slides].

[7] Tsang, S. C. (2024). *Lecture 10: ReactJS* [PowerPoint slides].