1. Abstract

This project develops a Single Page Application (SPA) to display location information from the Hong Kong Cultural Programmes dataset. The app supports user and admin access, with features including location listing (sorted/filtered), map visualization, detailed location views with comments, and favorite management. Admins can perform CRUD operations on events and users. Extra features including a dark/light theme switch, adaptive layouts, and secure authentication enhance usability. A representative screenshot shows the location list with sorting and filtering options.

2. Methodologies

2.1. Files with short description:

File/folder name	Description (What is it for?)
/controller	Handles business logic, processes requests, and returns responses.
/middleware	Contains functions that run between requests and responses (e.g., authentication checks).
/models	Data schemas and models.
/routes	Defines API endpoints and maps them to corresponding controller functions.
/services/ dataFetcher.js	Preprocess the data and import data into database.
/components	Reusable webpage components such as navbar.
/js	Stores frontend JavaScript logic (e.g., API calls, DOM manipulation).
/pages	Folder of pages that will be rendered.
/Route/ AppRouter.jsx	Configures frontend routing using React Router, managing page navigation.
/style	CSS stylesheets.

2.2. Pre-processing of the dataset:

Data Validation

When processing venue and event data, we only accept data with valid name (English), date, latitude and longitude. For those locations and events that don't meet the criteria, we will skip them. To ensure an event in our database is hosted by a venue in the dataset, we first process the dataset of the venue. After we get the mapping of the location, we validate the existence of event venues. Events having venue that do not exist in our location mapping will be skipped.

Data Cleaning

Beside the required field which must not be empty, other non-necessary field such as name(Chinese), description and

presenter are allowed to be empty. However, when we process the data, we will assign a default value to those fields. For example, if title (Chinese) is '--' or missing, it defaults to '暫無中文名稱'. Description and presenter defaulte to 'No Description' and 'No Presenter' respectively.

Data Filtering

After performing above steps, we will filter out top 10 locations hosting the most event. Locations are sorted by eventCount (descending) and only the top 10 are retained. Sometimes, there are not enough locations that have 3 events hosted in the dataset, we will choose locations with the highest eventCount that is less than 3.

Data Updating

If locations are already imported to the database, we will only update the event list for the 10 locations selected. After modifying events, it updates each location's events list, eventCount, and lastUpdated timestamp to reflect the latest state.

2.3. Core functionality:

Favourite Location System Functions

Users can add or remove locations from their personal favourite list. Favourite data is associated with the authenticated user and stored persistently in the database, ensuring favourites remain available across sessions. Users are restricted to favouriting locations only, preventing invalid associations with other entities.

The backend provides APIs to retrieve a user’s complete favourite list, enabling frontend features such as a dedicated favourites page and visual indicators in the location list.

Single Location View and Comment Handling Functions

Detailed information for a single location can be retrieved using its unique identifier. The backend returns full location details, associated event counts, and computed distance.

A comment system allows authenticated users to post comments on locations, while all users can view existing comments. Each comment is stored with references to the user and location, along with a timestamp. Comment identifiers are automatically generated by MongoDB, ensuring uniqueness and consistency.

Administrative Operations Functions

Administrative users are granted permission to manage user and event data through protected APIs. Role-based middleware ensures that these operations are inaccessible to normal users, maintaining system integrity and security.

User Module Functions and Algorithms

The user module manages user-related data and operations. User-related APIs allow administrators to create,

retrieve, update, and delete user accounts, while normal users are restricted from accessing sensitive user information and user management operations. User records store only essential attributes such as username, encrypted password, and administrative role indicators. Password validation and duplication checks are performed during registration and admin-only updates to ensure data integrity and security supported by the middleware (userAuth.js).

Event Module Functions

The event module manages event data associated with specific locations. Each event record maintains references to its corresponding location, enabling efficient aggregation and event counting at the venue level. Events are primarily accessed in a read-only manner by normal users, while administrators are permitted to perform full CRUD operations.

Since events' names are not required to be unique by the event schema, the program internally gets events by their unique IDs. During creation and updates, the input parameters of events are validated to handle their inconsistencies with the requirements of event schema supported by the middleware (eventAuth.js). Event data is also used to dynamically calculate the number of events at each location, which is required for sorting and filtering in the location listing feature.

2.4. Extra features:

Dark/light theme switch button

The dark/light theme switch is a user-centric feature designed to enhance accessibility and visual comfort by allowing users to toggle between light and dark color schemes. Implemented via a dedicated toggle button (displaying sun/moon icons) in the navigation bar, this feature dynamically adjusts the application's color palette to suit user preferences or environmental lighting.

User Information Encode(tokenization)

The "User Information Encode (Tokenization)" feature in the application is implemented using JSON Web Tokens (JWT) to securely represent and transmit user identity information, replacing direct transmission of sensitive credentials (e.g., passwords) during authenticated interactions. This mechanism ensures user data remains protected while enabling stateless authentication across the application.

2.5. Important algorithms/functions/components:

Authentication and Authorization Functions

User authentication is implemented using JSON Web Tokens (JWT). Users register and log in through dedicated authentication APIs. Upon successful login, the backend generates a JWT containing the user ID, username, admin role indicator, and expiration time.

The authorization process that decides what the user can do is tiered into “protect” and “adminOnly” and supported by a middleware (auth.js). Protected APIs require the token to be included in the HTTP Authorization header. The middleware verifies the token, decodes user information, and attaches the user document (if exist) to the request object. "AdminOnly” authorization is enforced by reading the admin role indicator from the user document attached to the request object, ensuring that administrative operations are restricted to authorized users.

Location Management Algorithms

The system supports retrieving a list of locations with sorting, filtering, and distance computation. Locations can be sorted by name, distance from CUHK, or number of events. Sorting behavior is dynamically controlled using query parameters, allowing flexible backend behavior without redundant endpoints.

Distances are calculated at request time using the Haversine formula, based on the geographic coordinates of CUHK and each venue. Distance values are not stored in the database, avoiding redundancy and ensuring accuracy.

Filtering mechanisms include keyword search, area-based filtering, and distance threshold filtering. These filters are implemented through conditional database queries and post-processing logic, enabling dynamic frontend updates.

Key Design Decisions of Security Functions

Middleware is extensively used to centralize authentication and authorization logic, reducing code duplication and improving security consistency. JWT-based authentication enables a stateless design, enhancing scalability and simplifying frontend integration. Controllers are modularized by domain, improving maintainability and testing efficiency.

2.5. Design of data schemas and models:

There are 4 models in our database: Location, Event, User, Comment.

Location:

Field	Type	Constraints	Description
locationId	Number	Required, unique	Numeric ID for the location imported from API.
nameC	String	Trim	Chinese name of the location.
nameE	String	Required, trim	English name of the location.
latitude	Number	Required	Geographic latitude imported from API.
longitude	Number	Required	Geographic longitude imported from API.
events	[ObjectId]	References Event model	Array of events hosted at this location.
eventCount	Number	Default: 0	Count of events hosed at this location.
comments	[ObjectId]	References Comment model	Array of comments about the location.

lastUpdated	Date	Default: Date.now()	Timestamp of the last data update.

Event:

Field	Type	Constraints	Description
eventId	Number	Required, unique	Numeric ID for the event imported from API.
titleC	String	Trim	Chinese title of the event.
titleE	String	Required, trim	English title of the event
venue	ObjectId	Required, references Location model	The location hosting the event
dateTime	Date	Required	Date and time when the event occurs.
description	String	Trim	Detailed description of the event.
presenter	String	Trim	Host/presenter of the event.

User:

Field	Type	Constraints	Description
username	String	Required, unique, trim	User's unique username
password	String	Required, hashed	User's password (stored as a hash)
isAdmin	Boolean	Default: false	Flag indicating if the user is an admin.
favoriteLocations	[ObjectId]	References Location model	Array of locations marked as favorites by the user.

Comment:

Field	Type	Constraints	Description
content	String	Required, trim	The actual comment is shown on the page.
user	ObjectId	Required, references User model	The user who wrote the comment.
location	ObjectId	Required, references Location model	The location the comment is about.

createdAt	Date	Default: Date.now()	Timestamp when the comment was created.

3. Contact Person

Lo Siu Fai 1155161402

4. Appendix

Workload distribution:

Name	SID	Word load
CAO Boyu	1155236597	-Frontend Page
DENG Zhen	1155233370	-Frontend Page
GAN Sikun	1155208855	-Backend development: responsible for location, favorite, and comment modules controller.js, and routes.js with corresponding functionalities and algorithms. - Front end React js polishing and transformed Map.js into Map.jsx format
LO Siu Fai	1155161402	Backend: - set up the MVC structure (server.js, routes/, controllers/) - Coded **User** and **Event** module (routes/controllers/middleware) Frontend: - set up the SPA structure using React (App.js, index.js, index.html) - Designed and coded the **signup** and **user manager** pages - Transformed **navigation bar** and **login page** into React format
TANG Tsz Hin	1155192635	-Data proprocessing -Designing Data Schemas -Frontend Page

Category	Method & Endpoint (with params)	Auth	Description	Notes / Params
Locations	GET /api/locations?sortBy=name|distance|events&order=asc|desc&keyword=...&area=...&maxDistance=...	JWT	List locations for table +	All query params are optional: sortBy, order, keyword, area, maxDistance (km). Example: /api/locations?sortBy=distance&

			map, with optional sorting and filters.	order=asc&maxDistance=2
Locations	GET /api/locations/:id	JWT	Get full details for a single location (for the single-location view), plus distanceKm.	:id = Location Mongo _id (from the list above). Example: /api/locations/665a1234...
Favourites	GET /api/favorites/locations	JWT	Get current user's full favourite locations list.	Uses user from token (req.user). No body/query needed.
Favourites	POST /api/favorites/locations	JWT	Add a location to curren	JSON body: { "locationId": "<Location _id>" }

			t user’s favourites.	
Favourites	DELETE /api/favorites/locations/:locationId	JWT	Remove a location from current user’s favourites.	:locationId = Location Mongo _id. Example: /api/favorites/locations/665a1234...
Comments	GET /api/comments/location/:locationId	JWT	Get all comments for a specific location (for the single-location page).	:locationId = Location Mongo _id. Example: /api/comments/location/665a1234...
Comments	POST /api/comments/location/:locationId	JWT	Create a new comment on a specific location	:locationId = Location Mongo _id. JSON body: { "content": "text..." }

			(visible to all users).	
Comments	DELETE /api/comments/:commentId	JWT	Delete a comment (only if current user is the owner or an admin).	:commentId = Comment Mongo _id. Example: /api/comments/777c1234...

Category	Method & Endpoint (with params)	Auth	Description	Notes / Params
Auth	POST /api/auth/register body:{username, password}	/	Register new user (not admin), and return { message, userId, username} upon success	Password should be no less than 8 char
Auth	POST /api/auth/login body:{username, password}	/	Login and return a json({message: 'Login successful', username: user.username, isAdmin: user.isAdmin, token}); Token: { id: String, username: String, isAdmin: Boolean}, 'dev_jwt_secret',	Token will expire in 1 hour by default. Decode by: const decoded = jwt.decode(token); The token should be deleted by frontend when logout.

			{ expiresIn: '2h' }	
User	GET /api/user/info	JWT	Get user info (exclude password) from token	Token needs to be included in the header
User	GET /api/user/info/:userId	JWT	Get user info from userId	userId: ObjectId e.g. user._id
User	GET /api/user/favorites	JWT	Get user favorite locations list from token	Token needs to be included in the header
User	GET /api/user/favorites/:userId	JWT	Get a list of favorite locations from userId	userId: ObjectId e.g. user._id
User	POST /api/user/	Admin	Create: new user	body: {username: String, password: String, isAdmin: Boolean}
User	GET /api/user/	Admin	Read: Get a list of all users' document	
User	GET /api/user/:username	Admin	Read: get user document by username	username: Str
User	PUT /api/user/:userId	Admin	Update: find user by Id and update the existing user's fields (optional)	userId: ObjectId e.g. user._id body: {username: String, password: String, isAdmin: Boolean}
User	DELETE /api/user/:userId	Admin	Delete: delete existing user by Id	userId: ObjectId e.g. user._id
Event	POST /api/events/	Admin	Create: new event, check input: - eventId (Number): unique - title (Str): at least one of E and C is provided	Body: { eventId: Number, titleE: Str, titleC: Str, venue: Str, dateTime: Date object, description: Str, presenter: Str }

			- dateTime (Date) - venue (Str): exists	
Event	PUT /api/events/byId/:id	Admin	Update: by ID, check input: - correct types - venue validity	Body:{title: Str, venue: Str, dateTime: Date, description: Str, presenter: Str}
Event	DELETE /api/events/:id	Admin	Delete: delete by ID and return the deleted event object	id: Str
Event	GET /api/events/	Admin	Read: return a list of all events	
Event	GET /api/events/byTitle/:title	Admin	Read: return the events whose name contains the given title (case-insensitive)	title: Str
Event	GET /api/events/random	JWT	Get a random event	