Band Finder - Report, Makowski Michał, csd6101

1. Added columns to database & used APIs

At the beginning, I would like to mention that one, important column has been added to the database we have been provided with. The column is named availability_dates and it is used in "bands" table. It has TEXT type, and it contains all the dates that each bands is available on. The records are stored after commas, so it looks like: "2026-01-09, 2026-01-10, 2026-01-12, 2026-01-14" for example. They appear in the dataset after selecting availability dates by each band via the calendar. This column functions as a dynamic pool of free slots: when an event (public or private) is created (also requested), the specific date is automatically removed from this string to prevent double-booking. If an event is deleted or rejected, the date is automatically restored to the list. Bands can't set a new public event for dates that are outside their availability range. They need to add this date first, then they are free to add a new public event. I have used a few APIs, which are:

a) Gemini API for ChatBot and turning text prompt into SQL,
b) Weather API for displaying weather for 5 following days in the footer of the website,
c) Rapid API for being able to sort public events based by distance between their home address and the event's address, when it comes to logged users.

2. Classes & Methods

The project was built using a functional approach in JavaScript language rather than Object-Oriented programming, so there are no classes. Below, key used methods (functions and endpoints) have been listed. The logic is based on server-side endpoints (app.js file) and client-side functions (mainly, in separate .js files). Additionally, I implemented specific server-side helper functions in app.js to handle complex logic regarding calendar availability:

- *removeDateFromAvailability(band_id, date)*: Automatically removes a specific date from the band's availability string when an event is created.
- *addDateToAvailability(band_id, date)*: Restores a date to the availability string when an event is deleted or rejected.

3. Server-side: app.js (endpoints, requests, REST)

Main job of this file is to handle all requests from users, connect to the MySQL database, and communicate with external APIs, that have been enlisted above. It also uses specific HTTP methods (GET, POST, PUT, DELETE) to manage data. Below is the list of all endpoints implemented in my app.js. GET for fetching data, POST for creating resources (registration, messages), PUT for updating profiles/statuses, and DELETE for removing events or users.

3.1. Database Administration

Enlisted endpoints in this section has been provided in the hy359_A3_project_2025-26_start_code.

a) GET '/initdb': initializes the database by creating all necessary tables.
b) GET '/dropdb': deletes the entire database to reset the system.

3.2. Registration & Login (Authentication)

a) GET '/RegistrationAndLogin/checkUsername': Checks if a username is already taken in the ‘users’ or ‘bands’ table. It return *true* or *false* (used for real-time validation).
b) GET '/RegistrationAndLogin/checkEmail': Check if an email address is already in use.
c) POST '/RegistrationAndLogin/Register': Registers a new simple user. It validates inputs, cleans data to prevent XSS attacks (used from previous assignments) and saves the user to the database.
d) POST '/RegistrationAndLogin/RegisterBand': Registers a new band. It converts numbers (like member count) to integers and saves the band profile.
e) POST '/RegistrationAndLogin/login': Logging in for users, bands, and the admin. It creates a session so the server remembers the logged-in person and keeps its details.
f) GET '/RegistrationAndLogin/checkSession': Checks if the user is currently logged in and returns their role (user, band or admin).
g) POST '/RegistrationAndLogin/logout': Destroys the sessin to log the user/band/admin out.

3.3. Profile Management

a) GET '/RegistrationAndLogin/getUserInfo': Gets full profile details for the logged-in user.
b) GET '/RegistrationAndLogin/getBandInfo': Gets full profile details for the logged-in band.
c) PUT '/RegistrationAndLogin/updateInfo': Updates profile information (for example: description, phone number etc.) for the logged-in user or band.

3.4. Public Events

a) GET '/getPublicEvents': Retrieves a list of public events. It supports filtering by genre, city and date using SQL WHERE clauses.
b) GET '/api/getMyPublicEvents': Returns only the events created by the logged-in band.
c) POST '/api/addPublicEvent': Allows a band to create a new public event. It automatically removes the selected date from the band's ‘availability_dates’ column to ensure the date cannot be booked again.
d) PUT '/api/updatePublicEvent': Allows a band to edit the details of their existing event.
e) DELETE '/api/deletePublicEvent/:id': Allows a band to delete an event. When an event is deleted, the system automatically adds the date back to the ‘availability_dates’ column, making it available for new bookings.

3.5. Band Listing & Details

a) GET '/getAlltheBands': Retrieves a list of bands. It supports filtering by genre, city and year that the band was founded.

b) GET '/api/getBandDetails/:id': Gets detailed data for a specific band to show in a pop-up window. It fetches the band’s info, their public events and their reviews.
c) GET '/api/getOccupiedDatesForBand': This endpoint fetches all dates currently occupied by public or private events for a logged-in band. It is used by the frontend to block specific days in the Flatpickr calendar, preventing double-booking logic visually.

3.6. AI & APIs

a) POST '/api/gemini-chat': Sends a user’s message to Google Gemini AI and returns the answer. Thanks to having it in app.js it hides the API key.
b) POST '/api/ai-execute-filter': Takes a user’s text prompt (for example: “Show me rock bands in Heraklion”), then sends it to AI to generate SQL code and executes that SQL to return results for user.
c) GET '/api/getWeatherForecast': It is used to get weather data through OpenWeatherMap API for Heraklion for 5 following days.
d) POST '/api/getDrivingDistances': Uses the Trueway Matrix API provided in the requirements .pdf file to calculate the distance between a user’s home (lan, lot from registration form) and the event location.

3.7. Private Events

a) GET '/api/getMyPrivateEvents': Loads private event requests sent by users or received by bands, depends whether the session is used by logged user or logged band.
b) PUT '/api/updatePrivateEventStatus': Used when a band accepts or rejects an event. If the band rejects the request (status changes to 'rejected'), the system automatically adds the date back to the ‘availability_dates’ pool so other users can book it.
c) DELETE '/api/deletePrivateEvent/:id': Deletes a private event request if it was rejected.
d) POST '/api/addPrivateEvent': A user requests a band for a private event. The status is set to ‘requested’. The requested date is immediately removed from the band's availability list to reserve the slot while the band makes a decision.

3.8. Chat System between an User and a Band

a) GET '/api/chat/getMessages': Loads the chat history for a specific private event.
b) POST '/api/chat/sendMessage': Saves a new massage in the database.

3.9. Reviews & Admin

a) POST '/review': Adds a new review for a band and sets its status as ‘pending’.
b) GET '/reviews/:band_name': Gets reviews with filters for rating and status for a single band or all the bands (depends on the request sent from the user).
c) PUT '/reviewStatus/:review_id/:status/:date_time': Admin changes a review status and then the date_time also changes. Because if the admin accepts a review, a week after someone put it into the database as ‘pending’, then we should publish it with the date it has been accepted (that’s the date, the review appeared on the website).

d) DELETE '/deleteUser/:user_id': Admin deletes a user account.
e) GET '/users/:user_name': Admin is able to see all or a single user in order to be able to delete them by using a button.
f) GET '/api/admin/statistics': Calculates stats for charts (for example: revenue, number of bands per city).

---

4. Libraries & HTML Elements

I have used 6 main external libraries to make the application better. I also used specific HTML structures to improve user experience, which we also have been told to use during previous assignments.

4.1. JavaScript Libraries

a) jQuery: It is the main library used in my project. I used it for all Ajax requests – communicating with the server without having to reload the page – and to easily change HTML elements on the page.
b) OpenLayers: Used to display the interactive map and put markers on it. It helps users to see where the events are taking place.
c) Flatpickr: I have used this library for an interactive calendar on "logged_band.html" page. It allows bands to select multiple dates for their availability, which are then saved to the database as a text string. The same calendar, now with the band's selected dates, was then shown to user on "logged_user.html" page in order to provide user with choosing a date for their private event.
d) Google Charts: This has been used in "admin_panel.html" only to draw pie charts, column charts. It visualizes website statistics, for example: number of bands per city or public vs private events count.
e) Validator: A library used on the server-side (app.js). It cleans user inputs (sanitization) to prevent XSS attacks. It was also used in previous assignments.

4.2. HTML Elements

a) Bootstrap Tabs: I used tabs to create a Single Page Application (SPA) view. Thanks to this, for example, in the dashboards (user/band/admin/guest) users can switch between sections like "Public Events", "Bands", "Your Availability" and else instantly without the page having to refresh.
b) Forms: Standard HTML5 forms are used for data entry (login, register, add event and else). I used them with *required* attributes and *pattern* regex to validate input data before sending it to the server.
c) I used HTML5 Semantic Elements (header, footer, main, section) to provide a clear document structure.

5. Detailed Ajax Usage

I used Ajax (through jQuery) a lot throughout the application to communicate with the app.js server without reloading the page. Below is the list of some, most important specific features where Ajax was implemented:

5.1. Registration & Authentication

- Real-time Validation: When a user types a username or email in the registration for, Ajax sends a request to check if it is already taken (endpoints: *checkUsername, checkEmail*). The user gets immediate feedback.
- Login & Logout: Logging in sends credentials to the server and logging out destroys the session, both happening quickly without a full page refresh.

5.2. Dashboard & Profile Management

- Loading Profile Data: When a user or band logs in, Ajax fetches their specific details (like description, phone number) to display on the dashboard.
- Updating Information: When a band edits their profile, a *PUT* request is sent via Ajax to save changes immediately.
- Availability Calendar: When a band selects dates in the calendar, Ajax converts them to a string and saves them to the database.

5.3. Events & Bands

- Dynamic Filtering: When a guest searches for "Rock music in Heraklion", Ajax fetches only the matching events or bands and updates the list instantly.
- Sorting by Distance: For logged users, Ajax sends the user's coordinates and event locations to the server to calculate driving distances (using Trueway Matrix) and reorders the event list, while also displaying the distance the sorting is based on.
- Managing Events: Bands use Ajax to add, update or delete their public and private events directly from their dashboard.

5.4. Communication & AI

- AI ChatBot: When a user asks a question, Ajax sends the text to the Gemini API endpoint and appends the answer to chat the window dynamically.
- Private Messaging: The chat between users and bands relies on Ajax to send new messages and fetch message history for a specific event.

5.5 Admin & Reviews

- Review System: Users submit reviews through Ajax. Admins also use Ajax to change a review's status or delete users from the system.
- Statistics: The admin dashboard uses Ajax to fetch calculated data to draw Google Charts.

6. CSS Style & Design

To ensure the application looks professional and consistent, I used a combination of the Bootstrap framework and my own custom styling. The design focuses on a clear color palette (orange, white, grey) and responsiveness.

6.1. Custom CSS (style.css)

I created a dedicated *style.css* file containing over 400 lines of code to handle specific design requirements that Bootstrap could not provide. Key elements include:

a) Global Theme: I set a global font family to Open Sans and defined a consistent color scheme. Commonly used color is orange, which is used for all buttons (*.submit-button, .navbar-btn)*, active tabs and highlights. I also added a hover effect: *.submit-button:hover { background-color: rgb(202, 131, 0);}*.
b) Layout & Grid: I used modern CSS layouts like *grid* and *flexbox*. For example, the class *.form-two* uses *display: flex* to evenly place two input fields next to each other. The *.weather-days-list* uses *display: flex* to evenly distribute weather icons in the footer.
c) Chat Interface: To make the chat look like a real messaging app, I created specific classes for message view. The *.message-left* class aligns received messages to the start (left side of the screen), while *.message-right* aligns sent messages to the end (right side of the screen).
d) Custom Pop-up Window: For viewing details of each band I used a simple but effective pop-up window logic. Using *.popup-overlay* with *position: fixed* and a semi-transparent background. This ensures the pop-up always stays in the middle of the screen, even if the user scrolls website down. Also the semi-transparent background helps user focus on the pop-up window.
e) Event Cards: I designed the *.event-band-card* class to display event summaries. It features a distinctive left orange border (*border-left: 3px solid orange*) to make it stand out from the white background and for aesthetic purposes.
f) Library Overrides: I customized the used Flatpickr calendar library to match my orange-grey theme. I overrode its default classes, like *.flatpickr-day.selected* to change the selection color to orange and increased the size of date cells for better usability.

6.2. Inline HTML Styles

While most styles are kept in the external file, I occasionally used inline styles like *style=”margin-top: 20px;”* directly within the HTML files. I used this approach for small, one-time adjustments where creating a separate class would be unnecessary. This was mainly used for adding specific margins to separate elements, setting custom widths for images or else. Sometimes I have also been using certain class, but I wanted to overrule some details with *!important*.

7. Client-Side Functions & Logic

Since the project relies heavily on JavaScript running in the browser to create a dynamic experience, I implemented many functions across different files. Below are the most significant ones, grouped by their purpose. It would be very inconvenient to describe all of them.

7.1. Authentication & Validation Logic

These functions ensure that data entered by users is correct and safe before it is sent to the server.

- *form_submition(event)*: This is the main handler for registration forms. It prevents the default page reload (*event.preventDefault()*), collects all input values into a JavaScript object and sends them to the server via an Ajax POST request.
- *password_strength()*: This function analyzes the password field in real-time. It checks if the password contains uppercase letters, numbers and special symbols using specific conditions. If the password is weak, it displays a warning message to the user.
- *is_password_the_same()*: It compares the "password" and "confirm password" fields. If they do not match, it instantly shows an error message.
- *forbidden_sequences(password)*: A security function that checks if the password contains easy-to-guess words related to the site (like "music", "band").

7.2. Map & Location

- *initialize_map()*: This function creates an instance of the OpenLayers map. It sets the default view to Heraklion and loads the OpenStreetMap layer.
- *setPosition(lat, lon)*: A helper function that transforms standard GPS coordinates into the coordinate system used by the web map.
- *verify_location()*: When a user types an address during registration, this function sends the text to an external Geocoding API. It receives the Latitude and Longitude and saves them in hidden form fields.
- *handleDistanceSort()*: A complex function for logged users. It takes the user's home coordinates and the coordinates of all displayed events. It sends this data to the server (Trueway Matrix API) to calculate precise driving distances. It also reorders the event HTML list from closest to furthest.

7.3. Dynamic Rendering

These functions take raw data (JSON) from the server and turn it into visible HTML elements.

- *renderEvents(eventsList, container)*: It loops through the list of events received from the database. For each event, it generates HTML code displaying the date, price and description, then it adds it to the page. It also adds a marker to the map for each event.
- *bandInformation(band_id)*: When a user clicks "Get to Know this Band", this function triggers. It performs an Ajax call to fetch full details about the band (including reviews and private event history) and displays them in a custom pop-up window.
- *loadMyPublicEvents()*: It fetches only the events belonging to the logged-in band and creates a table where the band can edit or delete them.

7.4. Chat & AI

- *sendChatQuery()*: This function handles the AI Music Assistant. It takes the text from the input field, sends the prompt to the server's Gemini API endpoint, and displays the AI's answer in the chat window.
- *loadChatMessages(private_event_id)*: It fetches the conversation history between a user and a band for a specific private event. It automatically formats the messages (left side for received, right side for sent).

- *sendChatMessage()*: It sends a new message from the user/band to the database and immediately refreshes the chat window to show the new content.

7.5. Admin

- *initAvailabilityCalendar(dates)*: Initializes the Flatpickr library. It reads the comma-separated string of dates from the database. Since the server updates this string automatically when events are added/removed, this calendar always shows the true, current availability of the band.
- *updateStatus(reviewId, newStatus)*: Used by the Admin. It sends a PUT request to change a review's status and refreshes the list.

8. Overall System Architecture

The application follows a client-server architecture. The frontend (client) is built as a dynamic web interface using HTML, CSS, and JavaScript, operating partially as a single page application (SPA) through the use of bootstrap tabs and ajax. The backend (server) is powered by node.js (app.js) and express, acting as a middleware that handles business logic, security (sanitization), and session management. The data layer consists of a MySQL database that stores persistent information about users, bands and events, which has been provided at the beginning.