Health App: Node.js, Express and MySQL Report

Health App Outline

This project is a small health-tracking web application built with Node.js, Express, EJS, and MySQL. The system allows users to create an account, log in, and record key aspects of their health and activity, focusing on two main areas: workouts and health metrics.

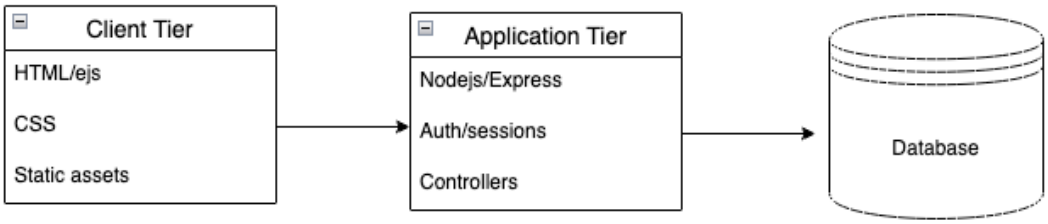
Each logged-in user has a personal dashboard that surfaces recent activity and simple summaries, so they can quickly see how active they have been over the last few days. The workouts section lets users record structured exercise sessions, including the type of workout, date, duration, and intensity, with optional notes. The metrics section allows users to log measurements such as weight, blood pressure, heart rate, steps, and sleep duration.

The application also includes a search page for querying historical data and an admin panel for managing users and reviewing login activity. Additionally, a dedicated weather page integrates with the OpenWeatherMap API so users can check the current conditions for any city, demonstrating secure use of environment variables and external APIs. The overall goal is to demonstrate a realistic multi-user web application that uses a relational database, server-side validation, authentication, role-based access control, and a clean Material Design-inspired user interface.

Architecture

The application follows a classic three-tier web architecture:

- Client tier (browser)
  Users interact via HTML pages rendered with EJS templates and styled with a shared Material Design-inspired main.css. All forms submit data using standard HTTP POST and GET requests, and the same layout/partials are reused across pages (home, dashboard, workouts, metrics, search, admin, and weather).
- Application tier (Express)
  index.js configures middleware (sessions, flash messages, validation, static files) and mounts feature-specific routers under `/`, `/auth`, `/dashboard`, `/workouts`, `/metrics`, `/search`, `/admin`, and `/weather`. Authentication state is stored in the session and exposed to views through `_middleware.js` (attachUserToLocals, requireLogin, requireAdmin). The `/weather` route calls the OpenWeatherMap API using an API key stored in OPENWEATHER_API_KEY.
- Data tier (MySQL)
  A MySQL database `health_app` stores users, workouts, metrics, lookup tables, and login audit logs. Connection pooling and the session store are handled through db.js using mysql2 and express-mysql-session. Weather data is retrieved live from the external API and is not persisted in the database.



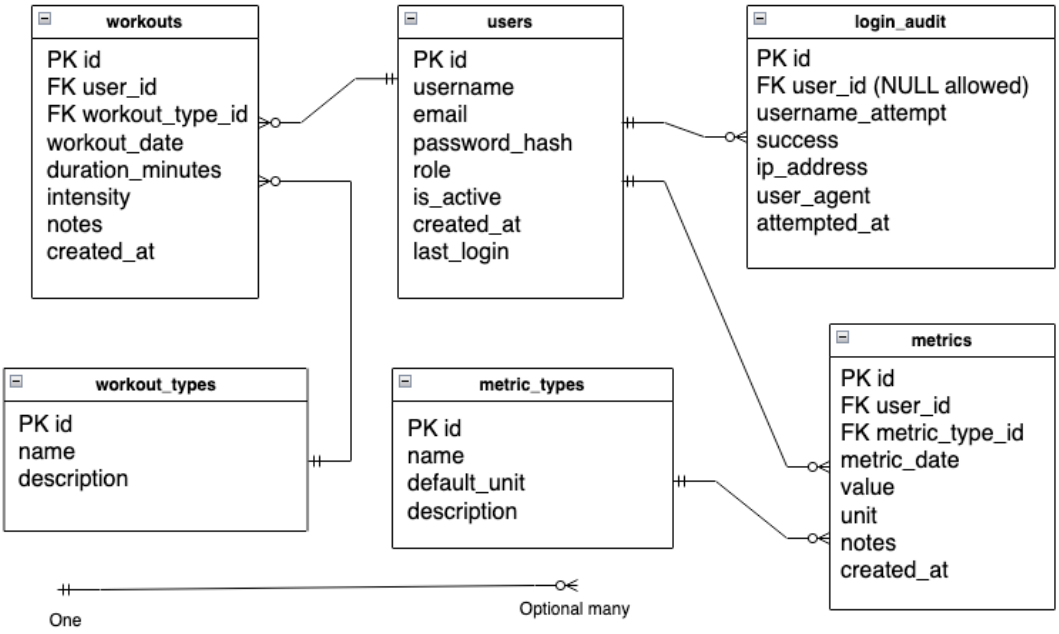
Data Model

The data model is centered on a users table, with related tables for workouts, health metrics, and login logging:

- users stores account details, including username, email, bcrypt password_hash, role (user or admin), is_active, and timestamps (created_at, last_login).
- workout_types and metric_types are lookup tables that define the allowed types of workouts (for example, Running) and metrics (for example, Weight, Blood Pressure), including default units.

- workouts represents individual workout sessions. Each row belongs to one user and one workout type, and stores date, duration, intensity, notes, and created_at.
- metrics represents health measurements for a user on a given date, linked to metric_types. It stores numeric values, optional units, notes, and created_at.
- login_audit logs login attempts with fields such as user_id, username_attempt, success, ip_address, user_agent, and attempted_at, which is used on the admin dashboard.

Foreign keys enforce relationships such as workouts.user_id → users.id and metrics.metric_type_id → metric_types.id. Indexes on (user_id, workout_date) and (user_id, metric_date) support efficient per-user queries. The weather feature uses the OpenWeatherMap API directly and does not introduce additional tables, keeping the core data model focused on users and health-related data.



User Functionality

The application offers different views and capabilities depending on whether the visitor is a guest, a normal user, or an admin.

Guest user

When a guest navigates to the root URL, they see a Home page (`/`) with a short introduction to the health tracker and links to Log in and Register. A separate About page (`/about`) briefly explains the purpose of the application and the types of data it manages (workouts and health metrics).

From the header navigation, a guest can reach:

- `/auth/register` – registration form
- `/auth/login` – login form
- `/about` – information about the app
- `/weather` – weather page (can be open to guests depending on configuration)

Any attempt to access protected pages (such as `/dashboard` or `/workouts`) redirects to the login page.

Registration and login

On the Register page (`/auth/register`), the user fills in a form with username, email, password and confirmation. The server uses express-validator to ensure that:

- The username and email are not empty and follow basic format rules.
- The password meets minimum length requirements.
- The confirmation matches the password.
- The username and email are not already taken.

If validation fails, the user is shown specific error messages next to each field and in a flash message. On success, the password is hashed with bcrypt and a new row is inserted into users with role = 'user' and is_active = 1. The user is then redirected to the login page with a success flash message.

On the Login page (`/auth/login`), the user enters their username and password. The server:

1. Looks up the user by username.
2. Verifies the password using bcrypt.compare.
3. Checks that the account is still active.
4. On success, stores a minimal user object in `req.session.user` and updates last_login.
5. Logs the attempt to login_audit with success/failure, IP address, and user agent.

To reduce the risk of brute-force attacks and accidental overload, the authentication routes are wrapped in an Express rate limiter. Each client IP is limited to 100 requests within a 15-minute window. Requests beyond this threshold are rejected with a clear error response, which protects the login and registration endpoints from high-volume abuse while still allowing normal users to retry a few times if they mistype their credentials.

If the login fails (wrong password or unknown user), a new row is also written to login_audit with success = 0. Feedback is shown using a flash error message. The Logout route (`/auth/logout`) simply clears the session's user and redirects to/with a flash success message.

Dashboard

After logging in, the user is redirected to the Dashboard (`/dashboard`). This page displays:

- A small summary panel with counts of:
    - Total workouts
    - Total metrics
    - Workouts and metrics in the last 7 days
- Two columns with recent items:
    - The latest few workouts (date, type, duration, intensity)
    - The latest few metrics (date, type, value and unit)

This provides the user with a quick overview of their recent activity, eliminating the need to delve into detailed lists. The queries are filtered by user_id so each user only sees their own data.

Workouts

The Workouts section is available at `/workouts` for logged-in users. The page supports:

- A filter form at the top, where the user can:
    - Choose a date range (from/to).
    - Filter by workout type (for example, Running, Walking, Cycling).
- A table of the user's workouts, showing:
    - Date
    - Type
    - Duration in minutes
    - Intensity (low, medium, high)
    - Notes
    - Links or buttons to Edit and Delete

Clicking Add Workout (`/workouts/add`) displays a form with date, type (select from workout_types), duration, intensity, and optional notes.

Server-side validation ensures that duration is a positive integer and that required fields are present. On submission, a new row is inserted into workouts linked to the current user. For Edit (`/workouts/:id/edit`), the form is pre-populated with existing values, and the user can update the entry. A POST form handles deletion to `/workouts/:id/delete`, which removes the row. Authorisation checks ensure users can only edit or delete their own workouts.

Metrics

The Metrics section at `/metrics` is similar in structure. The list view shows:

- Date
- Metric type (Weight, Blood Pressure, Heart Rate, Steps, Sleep)
- Value and unit
- Notes
- Edit and Delete options

Users can filter by date range and metric type. Adding a metric (`/metrics/add`) requires the user to select a metric type, date, and numeric value, with optional notes and unit override. Again, express-validator is used to ensure the value is numeric and mandatory fields are present. Edit and delete routes work in the same way as workouts, enforcing ownership via user_id.

This design allows the user to maintain a continuous log of measurements like weight change over time or daily step counts.

Search

The Search page (`/search`) brings these two data domains together. The user can choose to search:

- Workouts only
- Metrics only
- Both (combined results)

The search form allows the user to enter a free-text query and select a scope. The backend performs parameterised SQL queries that match the text against fields such as notes and type names, filtered by the logged-in user. Results are shown on `/search/results` in grouped tables (workouts section and metrics section), with each row linking back to the relevant module (for example, by date and type).

Weather

The Weather page (`/weather`) is an extra feature that integrates the application with the OpenWeatherMap API. The user can enter a city name such as `London,uk` or `Paris,fr` in a simple form. When the form is submitted:

1. The server reads the city string and the OPENWEATHER_API_KEY from the environment.
2. It calls the OpenWeatherMap current weather endpoint using fetch.
3. On success, the page shows a Material Design-style card with:

    City and country,

    Current temperature and “feels like” temperature,

    Humidity and wind speed,

    A short description such as “clear sky” or “light rain”.

If something goes wrong (for example, the city is not found, the API key is missing, or there is a network error), the page displays a clear error message telling the user what happened and how to fix it (for example, “No weather data found for this city” or “Weather is not configured”). This demonstrates robust error handling and graceful degradation when an external service is unavailable.

Admin

The Admin dashboard (`/admin`) is restricted to users with role = 'admin'. If a normal user tries to access this page, they receive a 403 page.

The admin view includes:

- A stats panel with:

    Number of registered users

    Number of active users

    Number of admins

    Total workouts and metrics

    Recent activity (for example, workouts and metrics created in the last 7 days)

- A Users table showing:

    Username, email, role, active status

    Counts of workouts and metrics per user

Created date and last login

Buttons to:

- Change role (user ↔ admin)
- Toggle active or inactive (with a check to prevent deactivating your own account)

- A Recent logins table from login_audit, showing:

  Username attempted (or “unknown”)

  Success or failure

  IP address

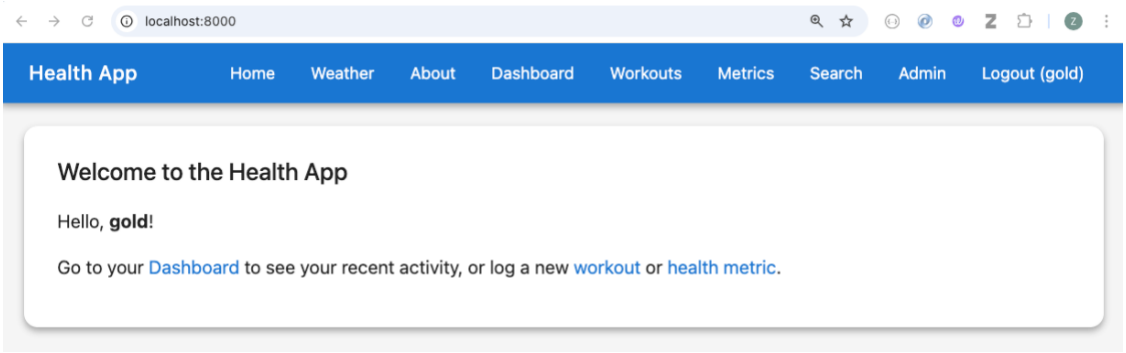
  User agent

  Timestamp

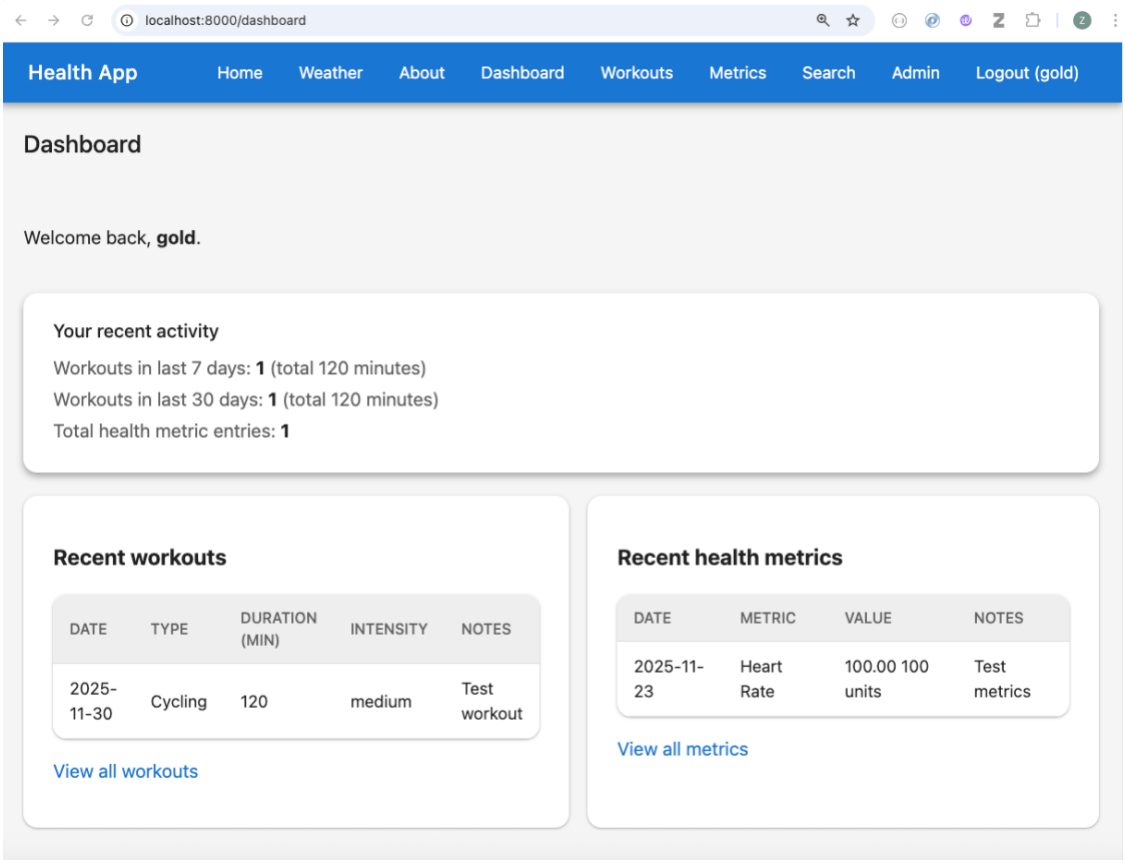
Error handling and UX

The app includes custom 403, 404, and 500 views, so users get friendly error messages instead of raw stack traces. Flash messages are consistently used to provide feedback on actions such as login, logout, validation errors, and successful creation or update of records. The layout and main.css provide a clean, modern, Material Design-inspired UI, with cards, elevated surfaces, rounded corners, and responsive layouts that keep the app usable on desktop and smaller screens.

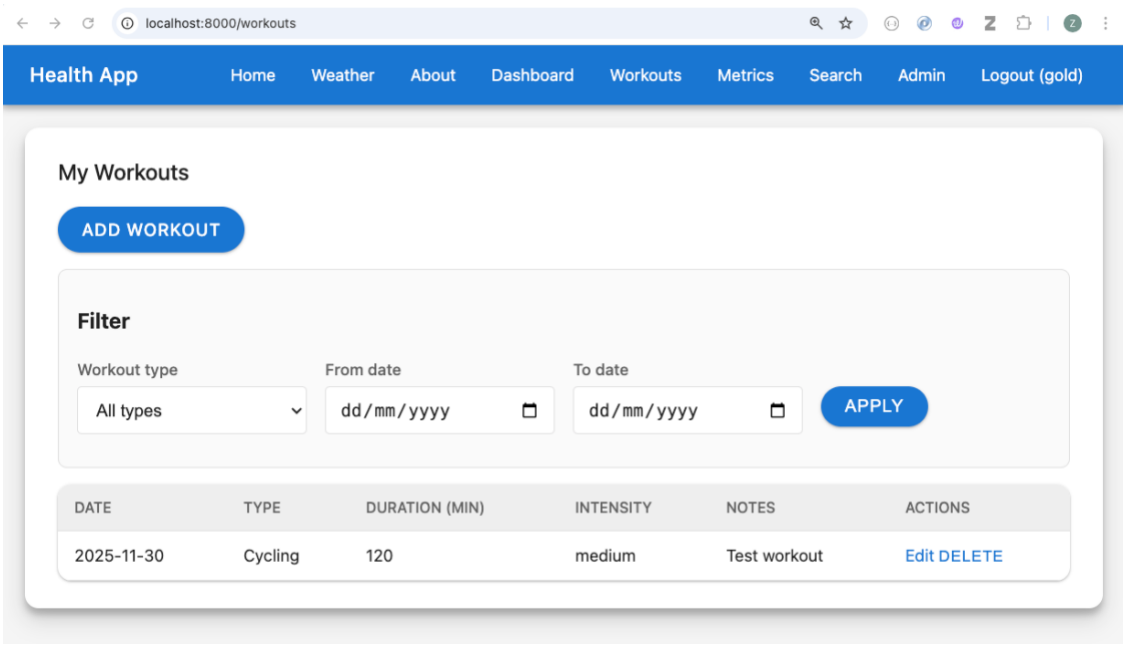
Home Page



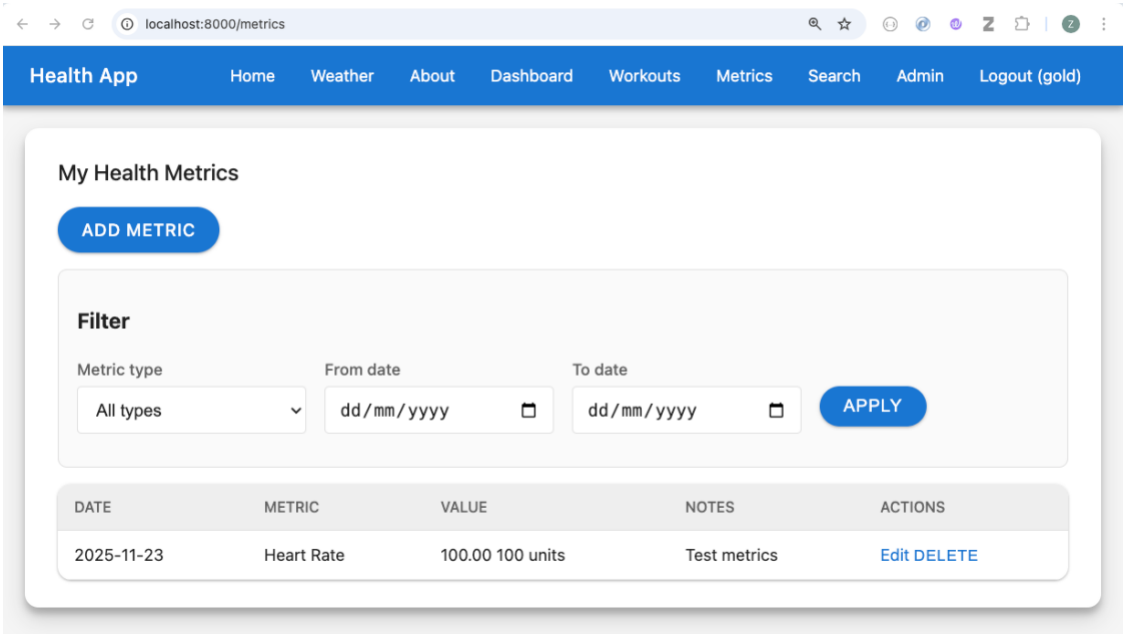
Dashboard Page



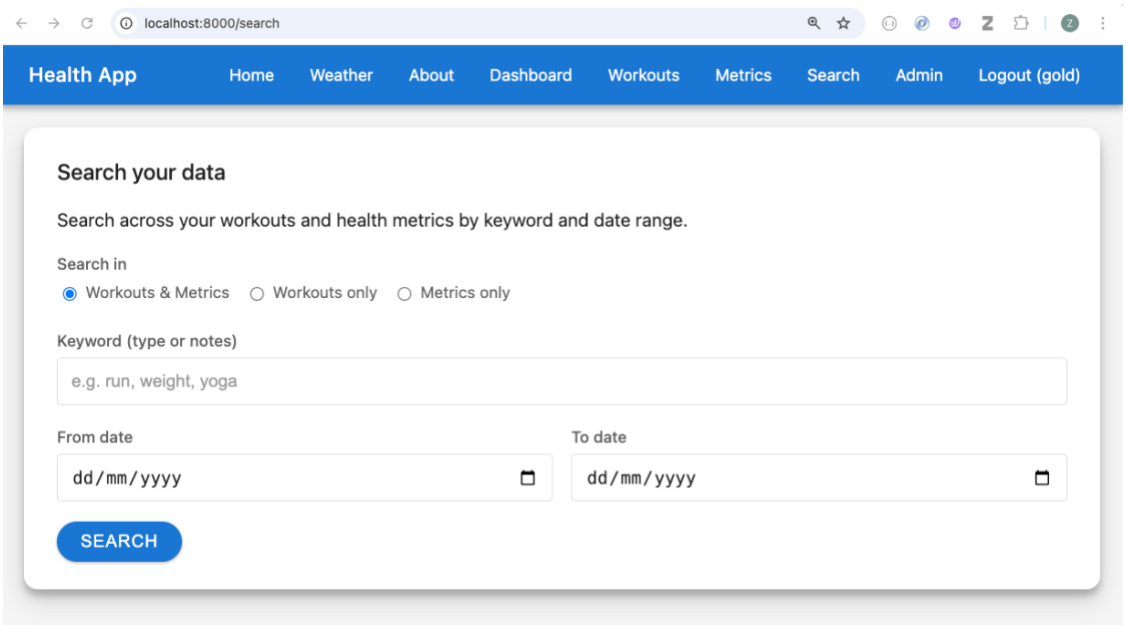
Workouts Page



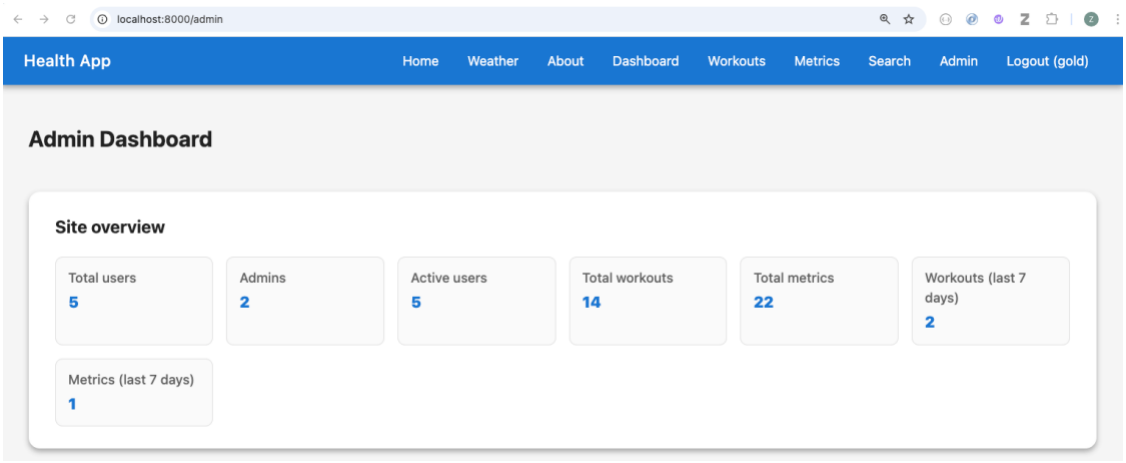
Metrics Page

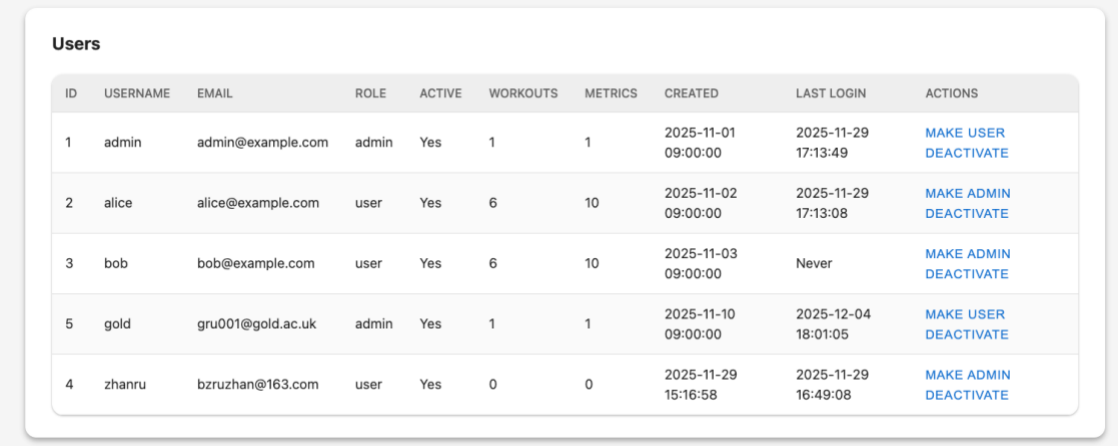


Search page


localhost:8000/search
Health App
Home
Weather
About
Dashboard
Workouts
Metrics
Search
Admin
Logout (gold)
Search your data
Search across your workouts and health metrics by keyword and date range.
Search in
Workouts & Metrics
Workouts only
Metrics only
Keyword (type or notes)
e.g. run, weight, yoga
From date
dd/mm/yyyy
To date
dd/mm/yyyy
SEARCH

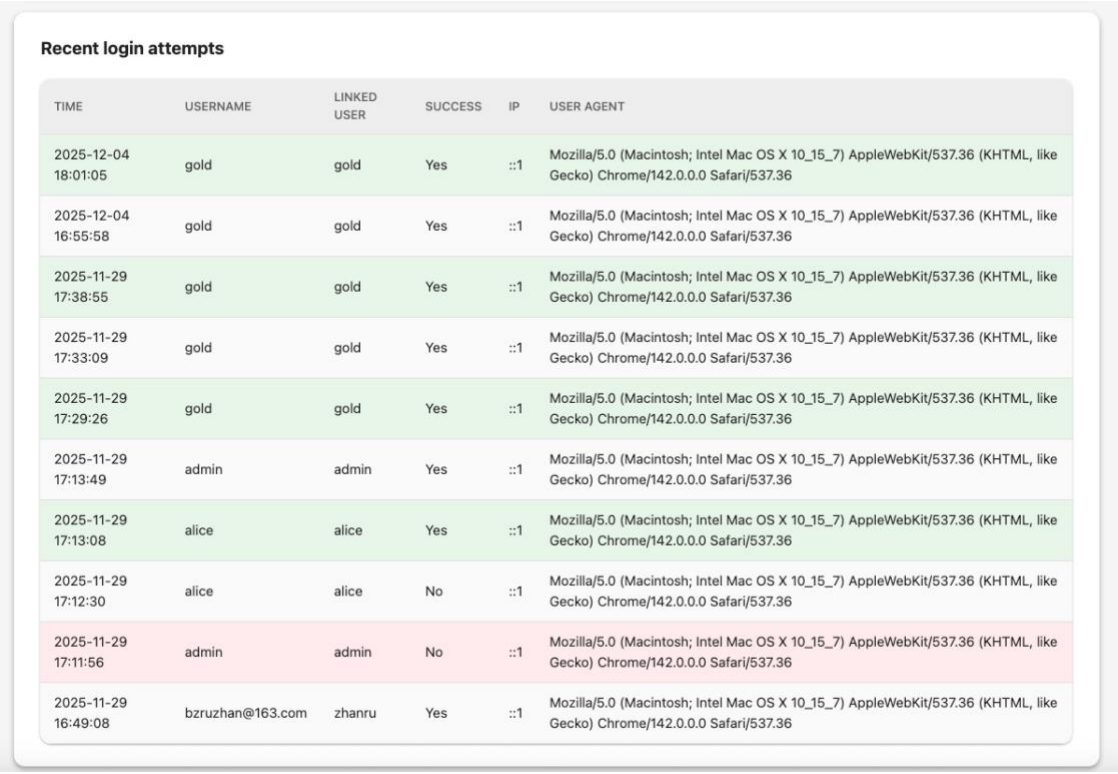
Admin Page


localhost:8000/admin
Health App
Home
Weather
About
Dashboard
Workouts
Metrics
Search
Admin
Logout (gold)
Admin Dashboard
Site overview
Total users
5
Admins
2
Active users
5
Total workouts
14
Total metrics
22
Workouts (last 7 days)
2
Metrics (last 7 days)
1



Users

ID	USERNAME	EMAIL	ROLE	ACTIVE	WORKOUTS	METRICS	CREATED	LAST LOGIN	ACTIONS
1	admin	admin@example.com	admin	Yes	1	1	2025-11-01 09:00:00	2025-11-29 17:13:49	MAKE USER DEACTIVATE
2	alice	alice@example.com	user	Yes	6	10	2025-11-02 09:00:00	2025-11-29 17:13:08	MAKE ADMIN DEACTIVATE
3	bob	bob@example.com	user	Yes	6	10	2025-11-03 09:00:00	Never	MAKE ADMIN DEACTIVATE
5	gold	gru001@gold.ac.uk	admin	Yes	1	1	2025-11-10 09:00:00	2025-12-04 18:01:05	MAKE USER DEACTIVATE
4	zhanru	bzruzhan@163.com	user	Yes	0	0	2025-11-29 15:16:58	2025-11-29 16:49:08	MAKE ADMIN DEACTIVATE



Recent login attempts

TIME	USERNAME	LINKED USER	SUCCESS	IP	USER AGENT
2025-12-04 18:01:05	gold	gold	Yes	::1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/142.0.0.0 Safari/537.36
2025-12-04 16:55:58	gold	gold	Yes	::1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/142.0.0.0 Safari/537.36
2025-11-29 17:38:55	gold	gold	Yes	::1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/142.0.0.0 Safari/537.36
2025-11-29 17:33:09	gold	gold	Yes	::1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/142.0.0.0 Safari/537.36
2025-11-29 17:29:26	gold	gold	Yes	::1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/142.0.0.0 Safari/537.36
2025-11-29 17:13:49	admin	admin	Yes	::1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/142.0.0.0 Safari/537.36
2025-11-29 17:13:08	alice	alice	Yes	::1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/142.0.0.0 Safari/537.36
2025-11-29 17:12:30	alice	alice	No	::1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/142.0.0.0 Safari/537.36
2025-11-29 17:11:56	admin	admin	No	::1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/142.0.0.0 Safari/537.36
2025-11-29 16:49:08	bzruzhan@163.com	zhanru	Yes	::1	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/142.0.0.0 Safari/537.36

Advanced Techniques

Several features go beyond the bare minimum requirements and demonstrate more advanced web development practices:

1. Role-based access control (RBAC)
   The application distinguishes between normal users and admins via the role

column in users. Middleware functions requireLogin and requireAdmin in `_middleware.js` centralise access checks. requireAdmin ensures that only admins can access `/admin` and perform actions such as changing user roles or toggling active status. This is more realistic than a single hard-coded admin account and shows good separation of concerns.

2. Login audit logging
   Every login attempt is recorded in the login_audit table, including unknown usernames and failed attempts. The auth routes capture the username attempted, success flag, IP address, and user agent. The admin dashboard then surfaces this information to admins, providing an example of security monitoring and auditing. This goes beyond a simple login form and demonstrates how to track and inspect authentication behaviour.

3. Input validation and sanitisation (no express-sanitizer)
   All database access uses parameterised queries with mysql2, so user values are never concatenated directly into SQL strings. This is the recommended defence against SQL injection, so the app does not add an extra "express-sanitizer" style layer on top. Instead, user input is validated and normalised using express-validator (for example, trim and numeric checks) before being sent to the database, and EJS's default escaping is used when rendering user data in views to protect against cross-site scripting.

4. Deactivation instead of deletion
   Instead of physically deleting users, the admin panel can toggle a user's is_active flag. The login logic checks this flag and denies access to deactivated accounts. This pattern (often called "soft delete") shows awareness of preserving data integrity and auditability, and is a common real-world approach to user management.

5. Form validation and user feedback with express-validator
   The registration, login, workout, and metric forms all use express-validator to enforce constraints on incoming data (for example, required fields, numeric ranges, email format). Validation errors are displayed next to specific fields and also in flash messages. This improves usability and robustness by preventing invalid rows from being inserted into the database and clearly guiding users to fix their input.

6. MySQL-backed sessions
   Sessions are stored using express-mysql-session, which means login state is persisted in MySQL rather than just in memory. This is closer to what would be used in a production setup and pairs well with the rest of the MySQL-based data model. It also demonstrates knowledge of middleware composition and configuration in Express.

7. Search across multiple domains
   The search functionality is implemented on the server side using parameterised SQL queries that can search workouts, metrics, or both. It combines joins to lookup tables (for type names) with filtering by the current user and (optionally) date

ranges or scopes. This is more complex than a simple single-table lookup and shows how to design flexible search endpoints.

8. Dedicated database user and environment-based configuration
   The create_db.sql script optionally creates a dedicated MySQL user (health_app) with limited privileges, and the application reads database credentials from .env using dotenv. This separation of configuration from code and principle of least privilege are both good development practices and go beyond minimal requirements.

9. External weather API integration
   The `/weather` page integrates the app with the OpenWeatherMap external API. It demonstrates:

   Server-side HTTP calls to a third-party REST service using fetch.

   Secure management of API credentials via OPENWEATHER_API_KEY in .env.

   Robust error handling for city-not-found, missing API key, and network or API failures, with clear user-facing messages.

   Normalising the API response into a simple weather object passed into an EJS template, which then renders a clean card with temperature, humidity, wind speed, and description.

   This feature shows how the health app can be extended with additional context (such as weather conditions before a run) and highlights good practices for working with external APIs.

10. Material Design-inspired UI
    The shared main.css stylesheet has been customised to follow Material Design principles: primary and secondary colour palette, elevated surfaces, rounded cards, pill-shaped buttons, clear focus states, and responsive layouts. Tables, forms, flash messages, and navigation elements all share a consistent look and feel. This not only improves the user experience but also shows attention to front-end design and accessibility within a server-rendered Express and EJS project.

11. Rate limiting on authentication routes
    The application uses `express-rate-limit` to throttle access to the authentication endpoints. Each IP address is limited to 100 requests in a 15-minute window, with appropriate rate-limit headers sent back to the client. This mitigates brute-force password guessing and protects the login and registration routes from automated scripts or accidental refresh storms, while still keeping the system responsive for genuine users.
12. Automated router tests
    A small but focused suite of automated tests (`app.test.js`) uses Supertest and Chai to verify route behaviour. The tests check that public pages (such as `/` and `/about`) return HTTP 200, while protected routes (for example,`/dashboard`, `/workouts`, `/metrics`, `/search`, and `/admin`) correctly redirect unauthenticated users to `/auth/login`. They

also validate server-side form handling for the auth routes by asserting that invalid login or registration submissions return status 422 with meaningful error messages, and confirm that the `/weather` route responds with 200 and displays an appropriate message when the `OPENWEATHER_API_KEY` is missing. Together, these tests provide a regression safety net and demonstrate how to use automated HTTP tests to enforce security and access-control rules at the router level.

AI declaration

In this assignment, I used AI tools in the following ways:

1. To get general advice on possible security practices (for example, parameterised queries and role-based access control), ideas for database structure, and ways to organise routes and pages. All final design decisions and implementation were done by me.
2. To help diagnose and fix technical errors. When I encountered problems such as SQL not working or runtime errors in my Node.js code, I asked AI for guidance on how to understand the error messages and how to correct them.
3. To improve the writing of this report. I used AI to help polish the language, correct grammar and spelling, and make the text clearer and more academic.

All code, design decisions, and final implementations were written, tested, and verified by me, and I take full responsibility for the work submitted.