HealthTrack

Outline

HealthTrack is a web-based fitness tracking application that enables users to log and monitor their health and fitness activities. Users can register securely, log various types of physical activities with details such as duration, intensity, and notes, and search through all logged activities using keyword-based queries. The application provides personalised activity histories for each user and implements secure authentication with password hashing. Built with modern web technologies, it demonstrates comprehensive full-stack development skills, including database design, server-side routing, form validation, and session management.

Architecture

The application follows a three-tier architecture consisting of:

Application Tier: - **Node.js**: JavaScript runtime environment for server-side execution - **Express 5.2.0**: Web application framework handling routing and middleware - **EJS**: Templating engine for dynamic HTML generation - **express-ejs-layouts**: Layout management for consistent page structure - **bcrypt**: Password hashing for secure authentication - **express-session**: Session management for user authorization - **express-validator**: Server-side form validation - **express-sanitizer**: Input sanitization to prevent XSS attacks

Data Tier: - **MySQL 2/MySQL2**: Relational database for persistent data storage - Three main tables: users (authentication), health_entries (activity logs), login_audit (security tracking)

Architecture Diagram:

[Client Browser]
↕
[Express Server] ← Routes (main.js, users.js, entries.js)
↕
[EJS Templates] ← Views (layout.ejs, index.ejs, etc.)
↕
[MySQL Database] ← Tables (users, health_entries, login_audit)

Data Model

The application uses three primary database tables:

users table: - Stores user account information including id (primary key), username (unique), first_name, last_name, email, hashedPassword, and created_at timestamp - Implements unique constraint on username to prevent duplicates

health_entries table: - Records fitness activities with id (primary key), user_id (foreign key to users), date, activity_type, duration_minutes, intensity, and notes - Foreign key relationship ensures referential integrity with users table

login_audit table: - Tracks authentication attempts with id (primary key), username, login_time, status, and message - Designed for security monitoring (currently implemented in schema but not actively used in routes)

Entity Relationship: Users have a one-to-many relationship with health_entries, where each user can log multiple activities, but each activity belongs to exactly one user.

Functionality

1. User Registration (/users/register) The registration page allows new users to create accounts with username, first name, last name, email, and password. Server-side validation ensures usernames are 2-20 characters, passwords meet complexity requirements (minimum 8 characters with uppercase, lowercase, number, and special character), and email addresses are valid. Passwords are hashed using bcrypt with 10 salt rounds before storage. Duplicate usernames are rejected with a clear error message.

2. User Login (/users/login) Registered users authenticate using their username and password. The application queries the database for the username, then uses bcrypt to compare the submitted password with the stored hash. Successful login creates a session storing the user ID and username. Invalid credentials display an error message without revealing which field was incorrect (security best practice).

3. Home Page (/) The home page displays different content based on authentication status. Logged-in users see a personalised welcome message with their username and quick-access buttons to log new activities or view their activity history. Non-authenticated visitors see a general welcome message with links to login or register. The page also lists key application features, including secure login, data logging, search functionality, and validation.

4. Add Activity (/entries/add) Protected by login middleware, this page presents a form for logging fitness activities. Users enter the date, activity type (e.g., Running, Cycling), duration in minutes, intensity level (Low/Moderate/High dropdown), and optional notes. All inputs are sanitised before database insertion to prevent SQL injection attacks. The form posts to /entries/added which inserts the record and displays a confirmation message with navigation links.

5. My Activities (/entries/my) This page displays a table of all activities logged by the current user, ordered by date (most recent first). The table shows date, activity type, duration, intensity, and notes for each entry. If no activities exist, the page displays a helpful message with a link to add the first activity. This demonstrates database querying with WHERE clause filtering and ORDER BY sorting.

6. Search Activities (/search) The search page provides a keyword-based search across all users' activities. Users enter a search term, which is sanitised and used to query both activity_type and notes fields using SQL LIKE with wildcards. Results display in a table showing date, username, activity type, duration, intensity, and notes. This demonstrates JOIN operations between the health_entries and users tables, as well as complex SQL queries with multiple LIKE conditions.

7. About Page (/about) The about page describes the application's purpose, lists core technologies used (Node.js, Express, EJS, MySQL, bcrypt, express-session, express-validator, express-sanitizer), and highlights key features. This page serves as technical documentation for users and demonstrates static content rendering with EJS.

8. Logout (/users/logout) The logout route destroys the user's session and displays a confirmation message. It's protected by the redirectLogin middleware to ensure only authenticated users can access it. After logging out, users are provided links to return home or log back in.

Advanced Techniques

1. Connection Pooling for Database Performance The application implements MySQL connection pooling instead of single connections, improving performance and scalability under concurrent user load.

```
// index.js lines 40-47
const db = mysql.createPool({
  host: process.env.HEALTH_HOST || 'localhost',
  user: process.env.HEALTH_USER || 'health_app',
  password: process.env.HEALTH_PASSWORD || 'qwertyuiop',
  database: process.env.HEALTH_DATABASE || 'health',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0,
});
```

Connection pooling maintains multiple database connections that can be reused, significantly reducing connection overhead and improving response times when multiple users access the application simultaneously.

2. Custom Middleware for Route Protection Custom authentication middleware (redirectLogin and redirectHome) implements the Don't Repeat Yourself (DRY) principle by centralizing authorization logic.

```
// routes/users.js lines 10-22
const redirectLogin = (req, res, next) => {
  if (!req.session.userId) {
    res.redirect('/users/login');
  } else {
    next();
  }
};

const redirectHome = (req, res, next) => {
  if (req.session.userId) {
    res.redirect('/');
  } else {
    next();
  }
};
```

This middleware is applied to routes declaratively (e.g., router.get('/add', redirectLogin, (req, res) => {...})), making code more maintainable and demonstrating understanding of Express middleware architecture.

3. SQL Injection Prevention with Parameterised Queries All database queries use parameterised statements rather than string concatenation, preventing SQL injection attacks.

```
// routes/entries.js lines 23-40
const sql = `
  INSERT INTO health_entries
  (user_id, date, activity_type, duration_minutes, intensity, notes)
  VALUES (?,?,?,?,?,?)
`;

const params = [
  userId,
  date,
  activity_type,
  duration_minutes,
  intensity,
  notes
];

db.query(sql, params, (err, result) => {
  if (err) return next(err);
  // ...
});
```

The ? placeholders are safely replaced by the database driver, ensuring malicious SQL code in user inputs cannot be executed.

4. Express-Validator Integration for Complex Validation Server-side validation uses express-validator's chain-able validation middleware, demonstrating advanced form handling beyond basic HTML5 validation.

```
// routes/users.js lines 43-50
router.post('/registered',
  [
    check('username', 'Username must be 2-20 characters long').isLength({ min: 2, max: 20 }),
    check('first_name', 'First name is required').notEmpty(),
    check('last_name', 'Last name is required').notEmpty(),
    check('email', 'Invalid email address').isEmail(),
    check('password', 'Password must be 8+ chars, with 1 uppercase, 1 lowercase, 1 number, and 1
symbol')
      .isLength({ min: 8 })
      .matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[^\da-zA-Z]).{8,}$/)
  ],
  async (req, res, next) => {
    const errors = validationResult(req);
    // ...
  }
);
```

This approach validates multiple fields with custom rules, collects all errors, and returns them to the user in one response, improving user experience.

5. Session-to-Template Data Flow The application makes session data available to all templates through middleware, enabling dynamic UI rendering based on authentication state.

```
// index.js lines 56-59
app.use((req, res, next) => {
  res.locals.session = req.session;
  next();
});
```

This allows templates to conditionally render content (e.g., “Login” vs “Logout” links in layout.ejs lines 23-31) without passing session data explicitly in every route handler.

6. Environment Variable Configuration The application uses dotenv for environment-specific configuration, following twelve-factor app principles for deployability.

```
// index.js line 7
require('dotenv').config();

// index.js lines 40-45
const db = mysql.createPool({
  host: process.env.HEALTH_HOST || 'localhost',
  user: process.env.HEALTH_USER || 'health_app',
  password: process.env.HEALTH_PASSWORD || 'qwertyuiop',
  database: process.env.HEALTH_DATABASE || 'health',
  // ...
});
```

This separates configuration from code, allowing different settings for development, testing, and production environments without code changes.

AI Declaration

AI assistance (Claude) was used in the following ways during this assignment:

1. **Code Structure Guidance**: Asked for best practices in organizing Express routes and middleware structure
2. **Debugging Assistance**: Used AI to troubleshoot bcrypt hashing issues during authentication implementation
3. **SQL Query Optimization**: Consulted AI for advice on JOIN syntax and parameterized query patterns
4. **Documentation Writing**: AI assisted in structuring this report and ensuring all required sections were comprehensively covered
5. **Code Comments**: AI helped generate detailed inline comments explaining complex logic in routes and middleware

Prompts:

What is the recommended folder structure for a Node.js Express application using EJS and MySQL? Please explain how to separate routes, controllers, middleware, and database logic for maintainability.

My Express login route always fails password comparison using bcrypt. Can you help me debug why bcrypt.compare() is returning false even when the correct password is entered?

What is the correct way to write a MySQL JOIN query to retrieve user details along with their related records, and how can I safely parameterize the query in Node.js?

Can you help structure my coursework report for a Node.js/Express web application?

Can you add clear inline comments to this Express route explaining how request validation, database queries, and error handling work?

All prompts were entered into Claude in December 2025

All core functionality was implemented independently, with AI serving as a reference similar to documentation or Stack Overflow. No code was copied verbatim from AI suggestions; all implementations were written and understood by the developer.