LAB 10- HEALTH AND FITNESS APPLICATION

Dynamic Web Applications

AISHA BHUDYE
33734353

Contents

Outline

The Clinic Appointment Manager is a web application that allows patients to book, view, and search for appointments at a clinic. It also includes a user authentication system for staff members, tracking login attempts in an audit log. The application is built using Node.js with Express as the web framework, MySQL for the database, and EJS for server-side templates. Users can submit appointment requests via forms, while staff can manage users and view logs of login activity.

Architecture

Technologies & Components:

- Application tier: Node.js, Express, EJS templates, bcrypt for password hashing
- Data tier: MySQL database with tables for `appointments`, `users`, and `login_audit`

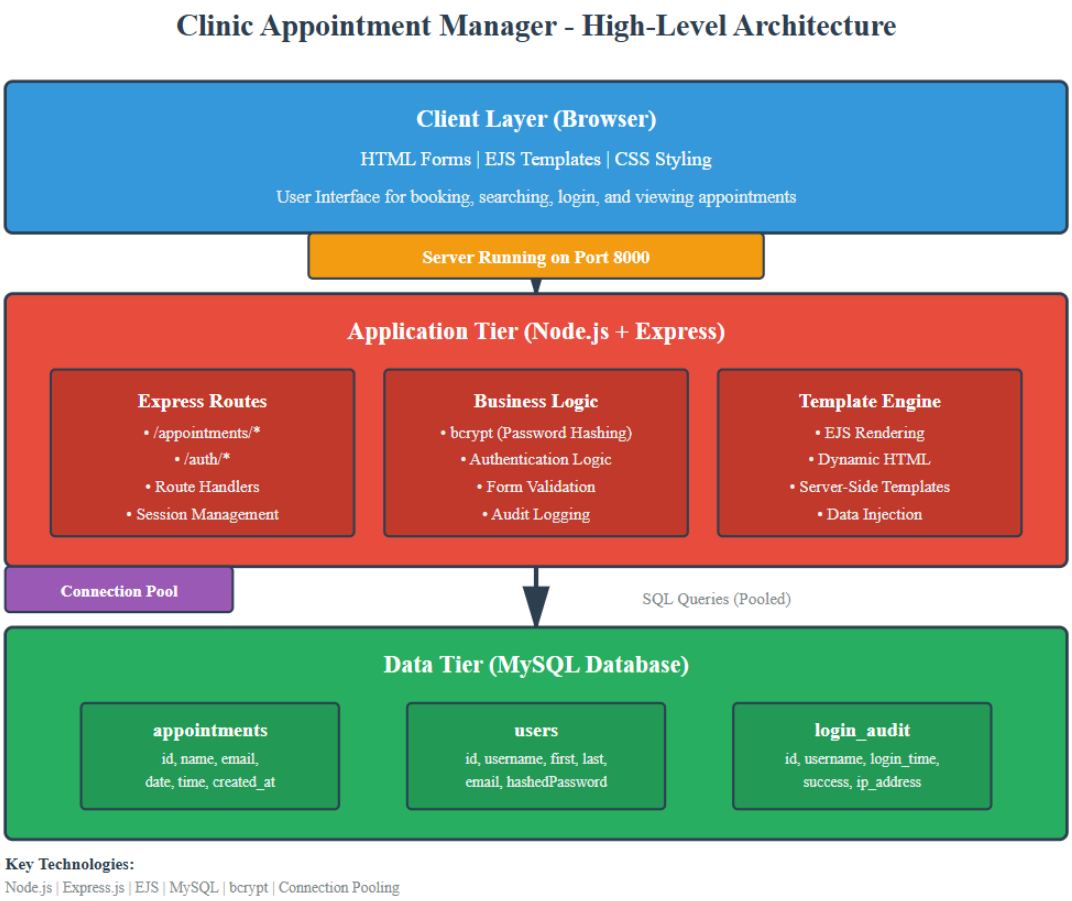
Deployment & Environment Configuration

The application is designed to be fully installable on the marker's machine. The environment variables required for database connection use the recommended configuration:

- HEALTH_HOST='localhost'
- HEALTH_USER='health_app'
- HEALTH_PASSWORD='qwertyuiop'
- HEALTH_DATABASE='health'

Running `npm install` installs all dependencies, and running `node index.js` starts the application on port 8000.

High-level Architecture Diagram:

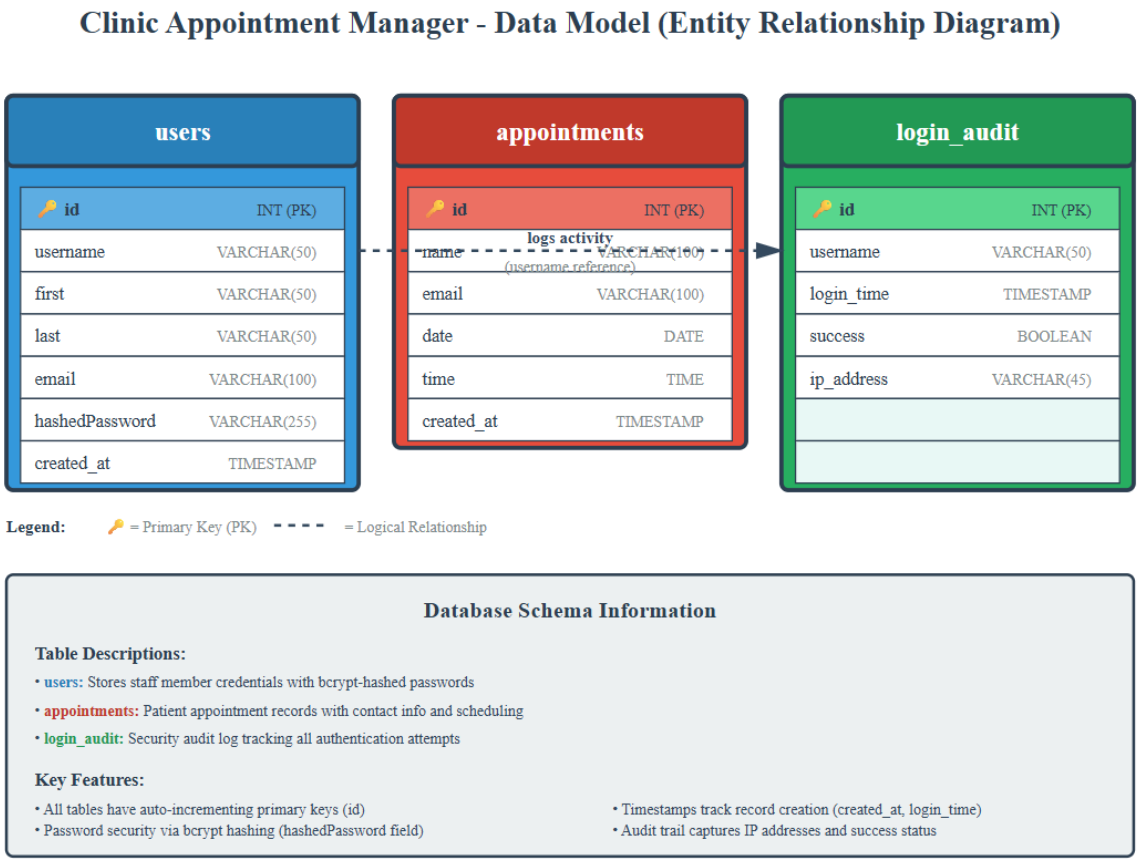


Data Model

The database has three main tables:

1. appointments – Stores patient appointments (id, name, email, date, time, created_at)
2. users – Stores staff user accounts (id, username, first, last, email, hashedPassword, created_at)
3. login_audit – Logs login attempts (id, username, login_time, success, ip_address)

Data Model Diagram:



Database Installation Scripts

Two SQL scripts are included as required:

- **create_db.sql** – Creates the health database and defines all tables (appointments, users, login_audit).
- **insert_test_data.sql** – Inserts initial data including the default staff user (**gold/smiths**) and optional example appointments.

These scripts ensure the database can be recreated from scratch during marking.

User Functionality

1. Book Appointment – Users fill a form (`/appointments/book`) with name, email, date, and time. On submission, the appointment is stored in the database.
2. List Appointments – Users or staff can view all appointments (`/appointments/list`) in a table sorted by date and time.
3. Search Appointments – Users can search for appointments by patient name (`/appointments/search`).
4. User Authentication – Staff can register (`/auth/register`) and log in (`/auth/login`). Passwords are hashed with bcrypt.
5. Audit Log – All login attempts are stored in `login_audit`, accessible via `/auth/audit`. This logs username, IP address, success status, and timestamp.

The application includes a Home page that provides navigation to all major features of the system, and an About page describing the purpose of the clinic appointment manager and the technologies used.

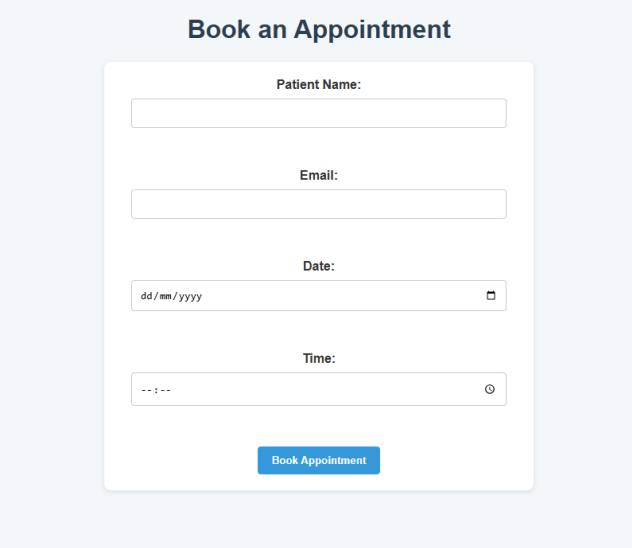
A default user account has been created as required by the brief:

- Username: gold
- Password: smiths

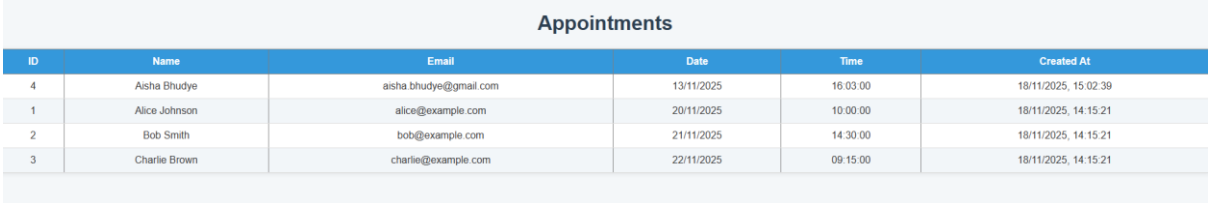
This account is inserted automatically through the insert_test_data.sql script when the application is deployed. It allows the marker to log in and access staff-only features such as viewing users and audit logs.

Example Screenshot Descriptions:

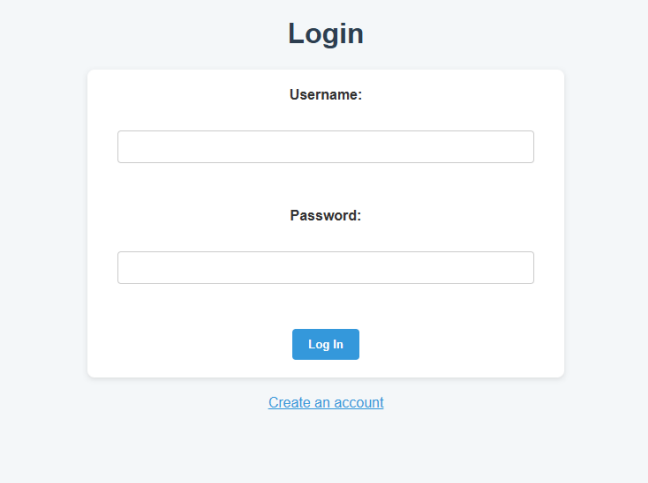
Booking form submission



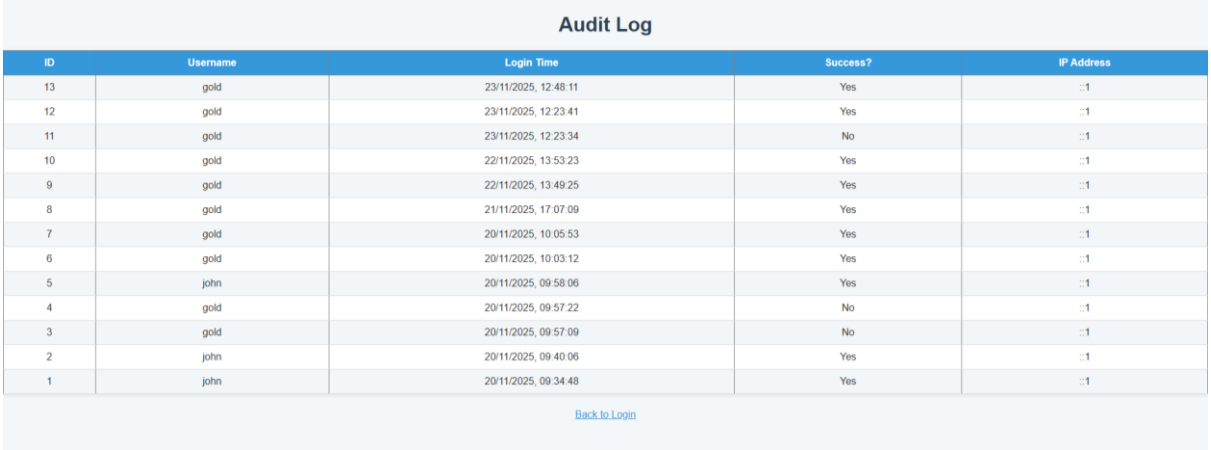
Appointment list table



Login page



Audit log showing successful and failed login attempts.



Advanced Techniques

1. Password Hashing with bcrypt

All passwords are stored securely using bcrypt with a salt round of ten. This ensures that even if the database is compromised, passwords cannot be easily reversed. Example from routes/auth.js:

```
const bcrypt = require('bcrypt').

const saltRounds = 10.


// Hashing password during registration

bcrypt.hash(plainPassword, saltRounds, (err, hashedPassword) => {

  if (err) return next(err).
```

```
  const sql = `
    INSERT INTO users (username, first, last, email, hashedPassword)
    VALUES (?, ?, ?, ?, ?)
  `;
  const params = [username, first, last, email, hashedPassword].

  global.db.query(sql, params, (err) => {
    if (err) return next(err).
    res.send(` Registration Successful` ).
  });
});
```

During login, passwords are compared using bcrypt's compare function:

```
bcrypt.compare(password, hashedPassword, (err, match) => {
  if (match) {
    req.session.userId = username.
    // Log successful login
    global.db.query(
      "INSERT INTO login_audit (username, success, ip_address) VALUES (?, ?, ?)",
      [username, true, req.ip]
    );
    return res.send("Login successful! Welcome " + username).
  }
  // Manage failed login
});
```

2. Input Validation with express validator

The application implements server-side input validation using the express-validator middleware to prevent malformed or malicious data from being processed. Validation rules are applied to the registration route in routes/auth.js:

```
const { check, validationResult } = require("express validator").

router.post(
```

```
"/registered",
[
  check("email")
    .isEmail()
    .withMessage("Invalid email"),

  check("username")
    .isLength({ min: five, max: 20 })
    .withMessage("Username must be 5–20 characters"),

  check("password")
    .isLength({ min: 8 })
    .withMessage("Password must be at least 8 characters"),

  check("first")
    .notEmpty()
    .withMessage("First name is required"),

  check("last")
    .notEmpty()
    .withMessage("Last name is required")
],
async (req, res, next) => {
  const errors = validationResult(req).

  if (!errors.isEmpty()) {
    // Re-render form with validation errors
    return res.render("register", {
      clinicData: req.app.locals.clinicData,
      errors: errors.array()
    });
```

```
    }

    // Proceed with registration if validation passes
    // ...
  }
);
```

This validation ensures:

- Email addresses are properly formatted.
- Usernames are between 5-20 characters.
- Passwords are at least eight characters long.
- First and last names are not empty.

If validation fails, the form is re-rendered with specific error messages displayed to the user, improving user experience and data quality.

3. Input Sanitization

To prevent XSS (Cross-Site Scripting) attacks, all user inputs are sanitized before being stored in the database. The application uses the expressSanitizer middleware to clean input data in routes/auth.js:

```
// Sanitise fields before database insertion
let first = req.sanitize(req.body.first).
let last = req.sanitize(req.body.last).
let username = req.sanitize(req.body.username).
let email = req.sanitize(req.body.email).
```

Sanitization removes or escapes potentially dangerous HTML and JavaScript code from user inputs, preventing malicious scripts from being injected into the application. This is configured in index.js:

```
const expressSanitizer = require('express sanitizer').
app.use(expressSanitizer()).
```

This technique protects against attacks where users might try to inject scripts like <script>alert('XSS')</script> into form fields.

4. Session-Based Authentication with Middleware

The application implements session-based authentication to protect sensitive routes. A custom middleware function redirectLogin ensures that only authenticated users can access protected pages. Implementation in routes/auth.js:

```
const redirectLogin = (req, res, next) => {
```

```
  if (!req.session.userId) {
    res.redirect("./login").
  } else {
    next().
  }
};

// Protected routes using the middleware
router.get("/list", redirectLogin, (req, res, next) => {
  const sql = ` SELECT username, first, last, email FROM users ` .
  global.db.query(sql, (err, results) => {
    if (err) return next(err).
    res.render("listusers", { users: results }).
  });
});

router.get("/audit", redirectLogin, (req, res, next) => {
  const sql = `
    SELECT id, username, login_time, success, ip_address.
    FROM login_audit
    ORDER BY login_time DESC
  `;
  global.db.query(sql, (err, results) => {
    if (err) return next(err).
    res.render("audit", { audits: results }).
  });
});
```

When a user successfully logs in, their username is stored in the session:

req.session.userId = username.

The redirectLogin middleware checks if req.session.userId exists before allowing access to protected routes. If not authenticated, users are automatically redirected to the login page. This prevents unauthorized access to sensitive functionality like user lists and audit logs.

Logout functionality destroys the session completely:

```
router.get("/logout", redirectLogin, (req, res) => {
  req.session.destroy(err => {
    if (err) return res.redirect("/").
    res.send("You are now logged out. <a href='/'>Home</a>").
  });
});
```

5. Database Connection Pooling

Using mysql2.createPool to manage multiple simultaneous connections efficiently in index.js:

```
const db = mysql.createPool({
  host: 'localhost',
  user: 'clinic_app',
  password: 'qwertyuiop',
  database: 'clinic_db',
  connectionLimit: ten
});
global.db = db.
```

Connection pooling creates a pool of reusable database connections rather than opening a new connection for each query. This significantly improves performance under high load by:

- Reducing connection overhead
- Managing concurrent requests efficiently
- Automatically managing connection failures and reconnections
- Limiting total connections to prevent database overload

The connectionLimit: ten parameter ensures a maximum of ten simultaneous database connections, preventing resource exhaustion.

6. Comprehensive Audit Logging

Every login attempt is logged with SQL insertions directly in the /auth/login route, capturing both successful and failed authentication attempts. Implementation in routes/auth.js:

```
const ipAddress = req.ip.


// Log failed login (user not found)
if (result.length === 0) {
```

```
  global.db.query(
    "INSERT INTO login_audit (username, success, ip_address) VALUES (?, ?, ?)",
    [username, false, ipAddress]
  );
  return res.render("login", {
    errors: [{ msg: "User not found" }]
  });
}

// Log successful login
if (match) {
  req.session.userId = username.
  global.db.query(
    "INSERT INTO login_audit (username, success, ip_address) VALUES (?, ?, ?)",
    [username, true, ipAddress]
  );
  return res.send("Login successful! Welcome " + username).
}

// Log failed login (incorrect password)
global.db.query(
  "INSERT INTO login_audit (username, success, ip_address) VALUES (?, ?, ?)",
  [username, false, ipAddress]
);
```

The audit log captures:

- **Username** - Who attempted to log in.
- **Timestamp** - When the attempt occurred (automatically via CURRENT_TIMESTAMP)
- **Success status** - Boolean indicating if login succeeded.
- **IP address** - Source IP of the request (req.ip)

This provides a complete security trail that can be used to:

- Detect brute force attacks (multiple failed attempts)

- Identify unauthorized access attempts.
- Track user activity for compliance.
- Investigate security incidents.

The audit log is accessible through a protected route (/auth/audit) that displays all login attempts in reverse chronological order.

7. Dynamic Templating with EJS

EJS is used to render appointment lists, login pages, and validation errors dynamically with server-side data. Example from views/register.ejs:

```
<% if (errors.length > 0) { %>
  <div class="alert alert-danger">
    <ul>
      <% errors.forEach(function(error) { %>
        <li><%= error.msg %></li>
      <% }); %>
    </ul>
  </div>
<% } %>
```

This allows validation errors to be displayed dynamically to users, showing specific error messages for each field that fails validation. Similarly, appointment lists and audit logs are rendered dynamically by iterating over database results passed from route handlers to EJS templates, providing a seamless integration between backend data and frontend presentation.

Summary of Security Enhancements

The application implements multiple layers of security:

1. **Password Security** - bcrypt hashing with salt rounds.
2. **Input Validation** - express-validator ensures data integrity.
3. **XSS Prevention** - Input sanitization removes malicious code.
4. **Authentication** - Session-based access control with middleware
5. **Audit Trail** - Comprehensive logging of all authentication attempts.
6. **Performance** - Connection pooling for efficient database access

These techniques work together to create a secure, robust web application that protects user data and prevents common web vulnerabilities.

A links.txt file is provided in the root of the GitHub repository, containing direct links to the deployed application pages, including Home, About, Search, Appointment Booking, Login, and Audit Log.

AI Declaration

I acknowledge the use of ChatGPT (OpenAI, 2025) and Grammarly (Grammarly Inc., 2025) to assist with proofreading, grammar checking, and improving sentence structure and clarity in this assessment. Both tools were used solely for language refinement; no generated text, ideas, or analytical content were included in the submitted work. ChatGPT was also used to help create the diagrams in this report. ChatGPT was accessed via https://chat.openai.com/ in November 2025 using the prompt:

“Proofread and improve the grammar and flow of my report.”

Grammarly was used for grammar correction and sentence structure review via https://www.grammarly.com/.

Reference List

OpenAI (2025) ChatGPT [Generative AI model]. Available at: https://chat.openai.com/ (Accessed: 10 November 2025).

Grammarly Inc. (2025) Grammarly [AI writing assistance tool]. Available at: https://www.grammarly.com/ (Accessed: November 2025).