Outline

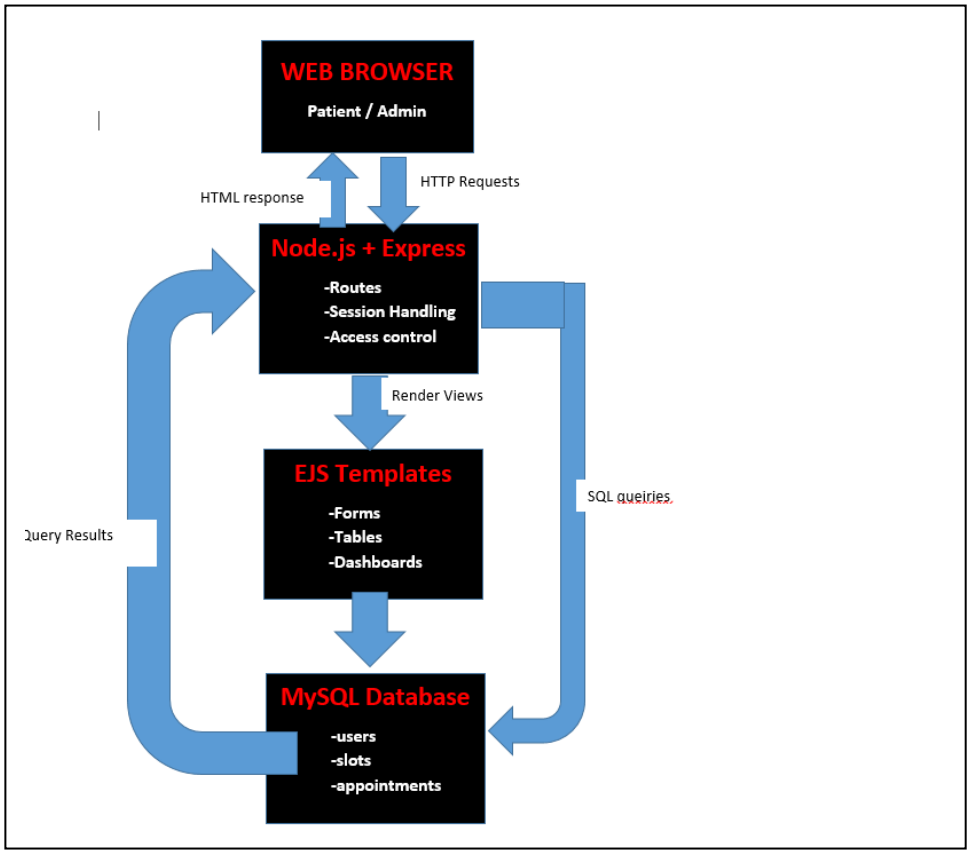
This project is a small clinic appointment booking system. It is aimed at a simple setting where patients book appointments online and a clinic administrator manages the available time slots and bookings. The app lets new patients register, log in, choose an available slot, and see the status of their appointments. The administrator can create slots across a date range, review all bookings, approve or reject them, and search the records.

The focus of the work is on a clear data model, sensible validation, and a straightforward user experience rather than lots of visual effects. Both admin and patient accounts share the same login screen, but their access is separated on the server side so that admin-only tools are protected. The application runs on Node.js and Express, uses EJS for server-side rendering, and stores all persistent data in a MySQL database called ‘health’.

Architecture

The system follows a three-tier architecture.

- **Presentation layer:** EJS templates (home.ejs, login.ejs, patient-dashboard.ejs, admin-dashboard.ejs, etc.) plus a shared CSS file provide the pages, forms, tables, and navigation the user sees.
- **Application layer:** Node.js and Express handle routing, sessions, authentication, input sanitisation, slot creation logic, pagination, and access control. The main entry point is index.js, which wires together route files such as auth.routes.js, patient.routes.js and admin.routes.js.
- **Data layer:** A MySQL database called health holds three tables: users, slots, and appointments. All data access goes through parameterised queries using a shared connection pool.



Data Model

The database has three main tables.

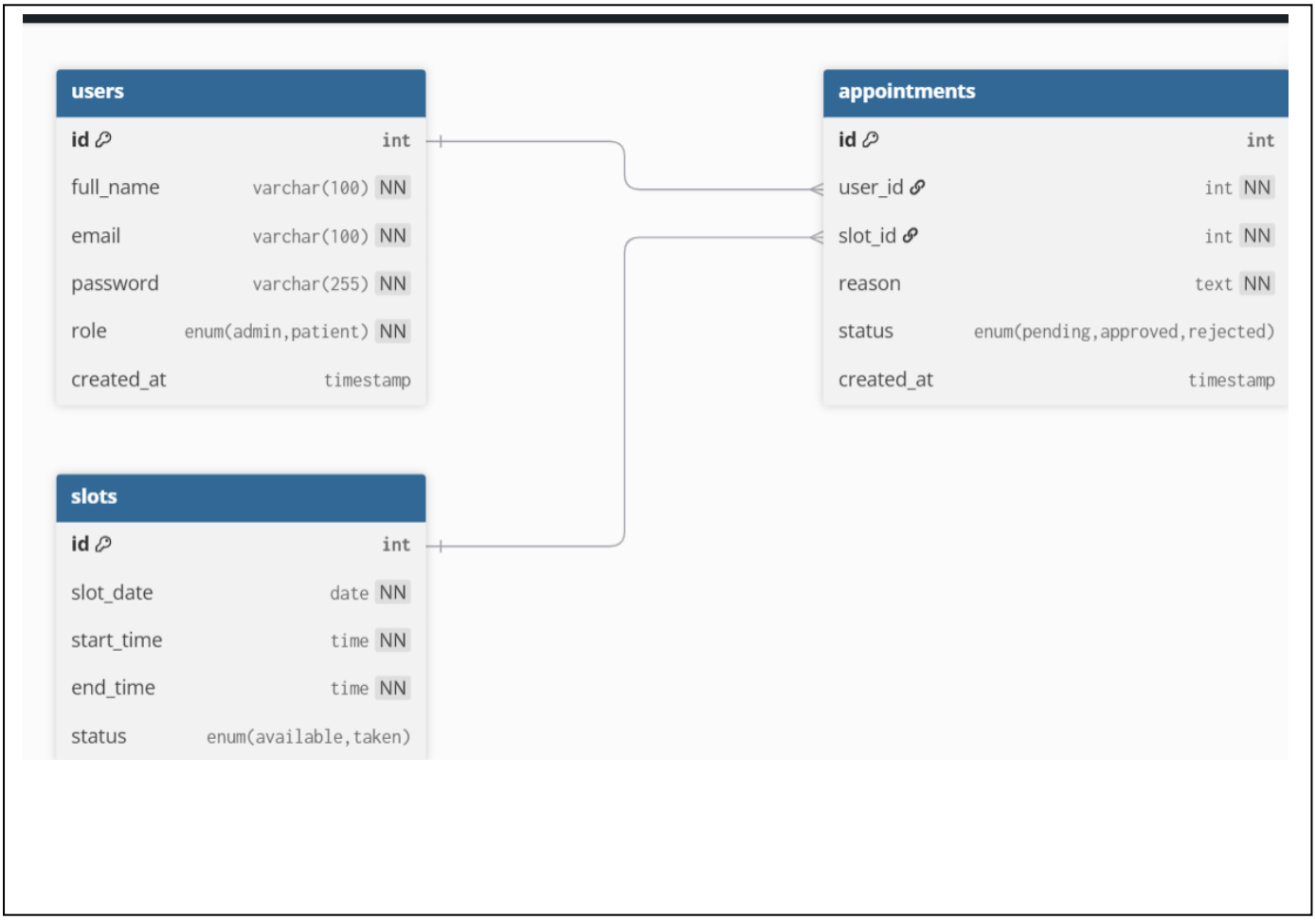
- **users** – stores both patients and admin accounts. Fields: id, full_name, email, password (hashed), role (‘admin’ or ‘patient’), and created_at.

  For the administrator account, the username is stored in the email field so that the same authentication logic can be reused for both roles.

  slots – stores appointment windows created by the admin. Fields: id, slot_date, start_time, end_time, and status (‘available’ or ‘taken’).

- **appointments** – links a user to a slot. Fields: id, user_id, slot_id, reason, status (‘pending’, ‘approved’, ‘rejected’), and created_at.

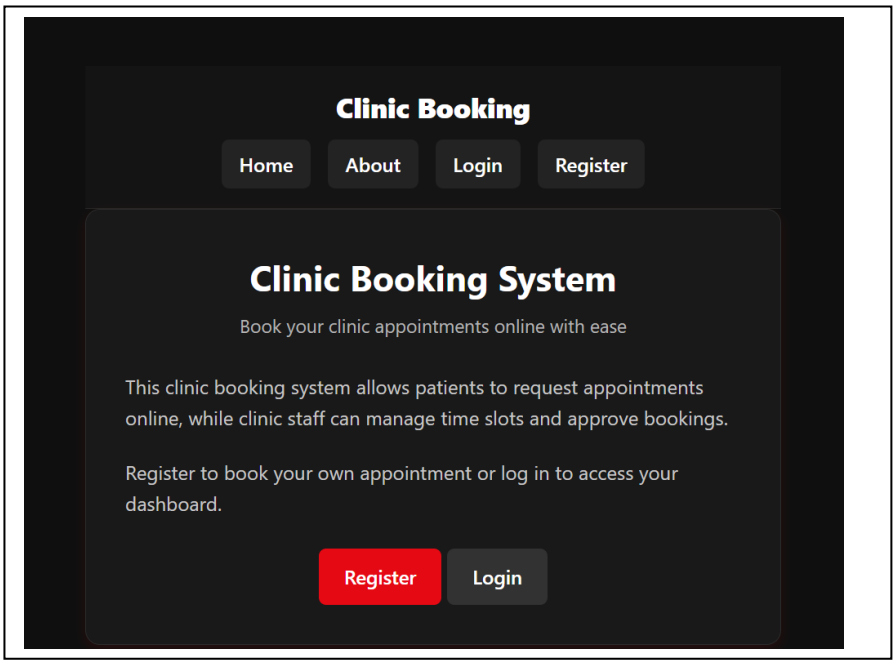
Foreign keys enforce referential integrity and cascade deletes, and a UNIQUE(slot_id) constraint prevents the same time slot from being booked more than once.



User Functionality

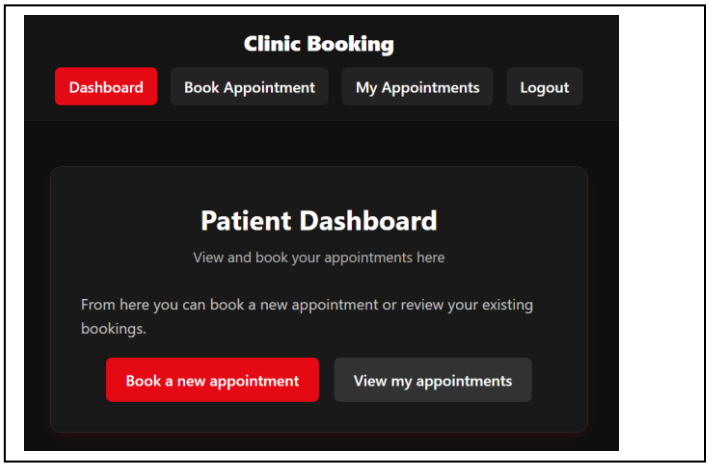
The application has two roles: **patient** and **admin**.

The **Home** page introduces the clinic booking system and gives clear links to register or log in. The **About** page briefly explains the purpose of the system and how patients and staff are expected to use it.



The **Register** page lets a new patient create an account by entering full name, email and password. Basic validation is performed server-side and duplicate emails are rejected using the database constraint. The **Login** page accepts either an email address (for patients) or the special username gold (for the marker's admin account), plus the password. After login, the server stores the user in the session and redirects to the correct dashboard based on their role.

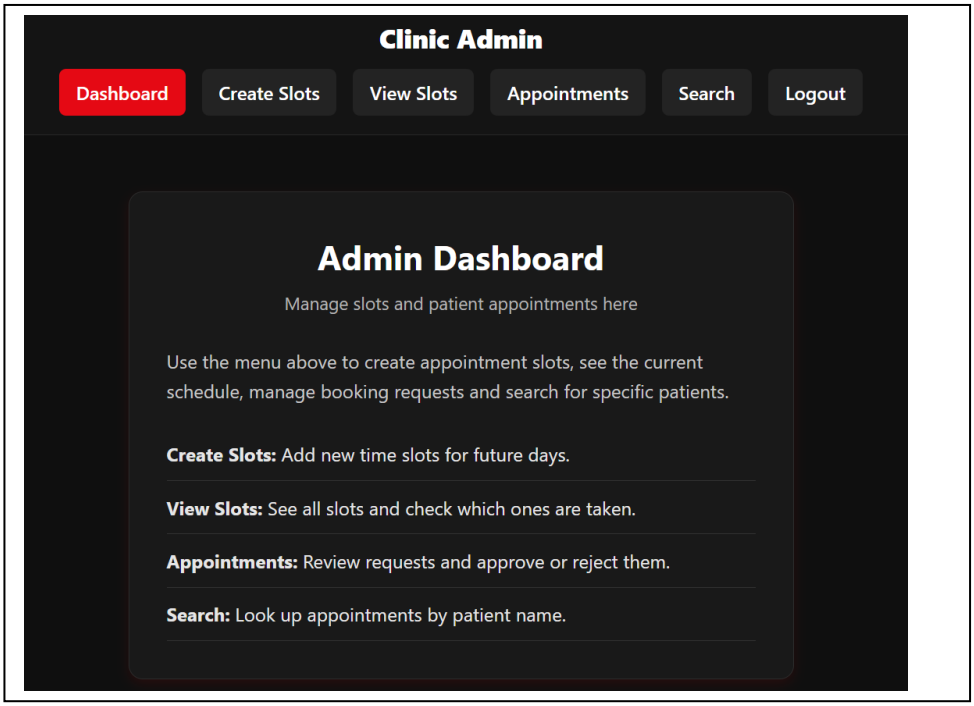
The **Patient Dashboard** gives a short welcome message and links to "Book Appointment" and "My Appointments". From here the patient can either request a new slot or review their bookings.



On **Book Appointment**, the system queries all slots where status = 'available' and shows them in a dropdown as date start_time - end_time. The patient enters a short reason and submits the form. The server double-checks that the selected slot is still available, creates an appointments row with status = 'pending', and updates the slot to taken.

The **My Appointments** page lists the patient's own bookings in a table with columns for date/time, reason and status. The list is paginated so that only a few appointments are shown per page, with simple "Previous / 1 2 3 / Next" links at the bottom.

The **Admin Dashboard** is only accessible to users with the admin role. It gives short explanations and links to create slots, view slots, manage appointments and run searches.



On **Create Slots**, the admin chooses a start date, end date, start time, end time and a repeat pattern ("every day" or "weekdays only"). When the form is submitted, the server loops through each day in the range, skips weekends if required, checks for overlapping slots on that date, and then inserts new slots rows where it is safe to do so. A success message tells the admin how many slots were created, or explains that no new slots were added.

Manage Slots shows all slots in a table with date, time and status, ordered chronologically. This table is paginated to keep the page readable when many slots exist. The admin can use this view to quickly see how busy the schedule is.

Manage Appointments lists bookings from all patients. Each row shows the appointment ID, patient name, session time, reason, and status. For pending rows, there are "Approve" and "Reject" buttons which send a POST request to the corresponding route. Approved and rejected rows simply display "No action". This page is also paginated.

Finally, the **Search** page allows the admin to search across appointments using a patient name fragment. Results are displayed in a small table with the same core fields so the admin can quickly jump to the relevant booking.

[Screenshots for each main page here]

Advanced Techniques

A few features go slightly beyond the basic lab patterns and are worth highlighting.

1. Date-range slot generation with overlap checks

In admin.routes.js, the "Create Slots" feature lets the admin generate many slots in one go by choosing a date range and a time window. The server converts the start and end dates to Date objects and walks through the range using a helper function. For each day it decides whether to include it based on the repeat pattern (every day vs weekdays only). Before inserting a slot, it queries existing slots on that date and checks for overlaps using the condition:

```
if (!(endTime <= slot.start_time || startTime >= slot.end_time)) {
   overlap = true;
}
```

Only when no overlap is found is a new row inserted. This avoids duplicate or clashing slots without relying purely on database errors.

2. Pagination on slot and appointment lists

Both the admin slot list (/admin/slots) and the appointment lists (/manage-appointments and /my-appointments) use server-side pagination. The pattern is:

- Read page from the query string (default 1).
- Use a fixed limit and compute offset = (page - 1) * limit.
- Run a COUNT(*) query to work out totalPages.
- Run a second query using LIMIT ? OFFSET ? to fetch only the current slice of rows.

The EJS templates then render simple page links based on currentPage and totalPages. This keeps the pages usable even when there are many records and shows that I can manage data in a more scalable way.

3. Shared login with role-based access

Instead of having separate auth flows, the app uses a single login form. Patients log in with their email, while the special admin account can log in with the username `gold`. After authentication, the user's `role` determines whether they are sent to `/patient` or `/admin`. Middleware functions like `ensureAdmin` and `ensurePatient` then guard the restricted routes. This keeps the UI simple while still enforcing clear access control rules.

AI Declaration

I used AI as a support tool mainly during the planning and design stages of this assignment. It helped me think through possible features, break the work into steps, and sketch out how the overall system could be structured. I also asked for general advice on layout ideas and how to word parts of my documentation. Where AI guidance was used, I treated it as a rough prototype and then implemented the final version myself so that I fully understood how everything worked.

Also, during deployment, I faced an issue where the application behaved differently on my local machine compared to the Goldsmiths virtual server. Locally, the app runs from the root path, but on the server it is hosted under a sub-directory (/usr/350). This caused redirects after login to point to incorrect URLs on the hosted version.

To understand the problem, I used AI tools to help diagnose the issue. AI suggested handling this using an environment-specific base path. Following this guidance, I introduced a HEALTH_BASE_PATH environment variable, which is left empty when running locally and set to /usr/350 on the virtual server. This resolved the routing issue without changing the core application logic.

AI was used only to assist with identifying the cause of the problem and suggesting a general approach. The final implementation and configuration were carried out manually.