Healthcare Database Design

Information Management - CW1

5551408

Table of Contents

Introduction

This report details the design and development of the attached database prototype for a healthcare user management system. The database is a relational model, incorporating tables for Patients, Staff, Doctors, Medical Records, and Appointments. It utilises a Role-Based Access Control (RBAC) to restrict user operations and add defence in depth when combined with secure back-end logic. The prototype was designed and normalised to 3rd Normal Form to keep redundant data to a minimum.

Data Model

An Entity Relationship Diagram has been created using crows-foot notation to show the relations between each table in the database (See Figure 1).

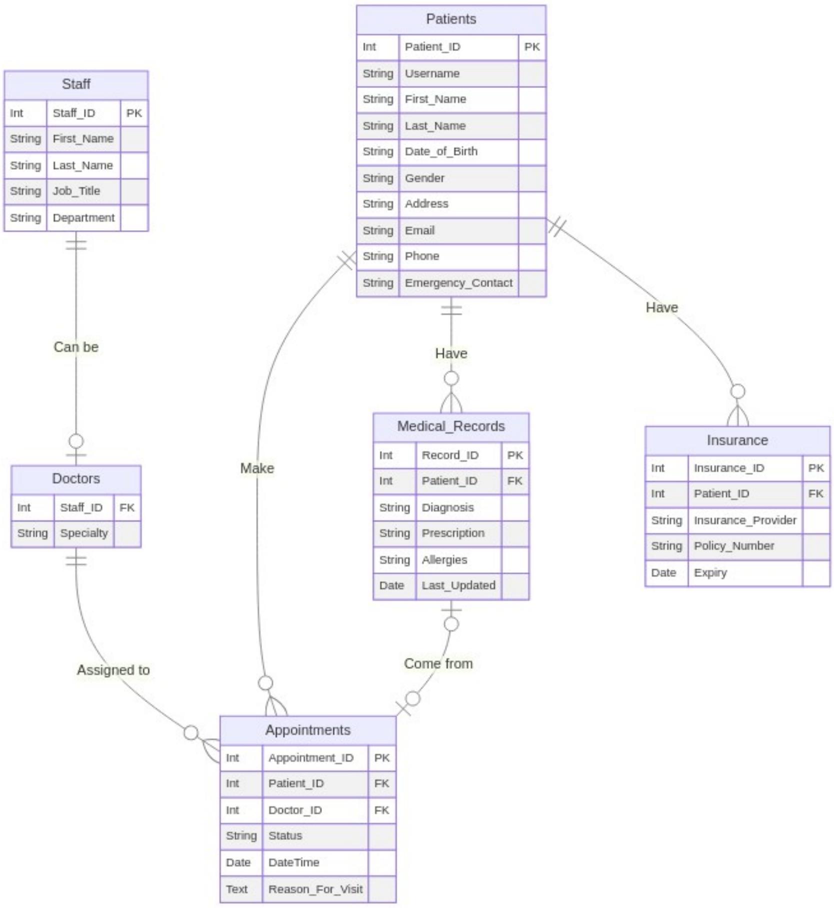


Figure 1: Entity Relationship Diagram

Table Descriptions

Patients Table

The “Patients” table holds all of the information about a patient. This includes their name, date of birth, contact number. A patient should be able to view and edit their own record in this table.

Staff and Doctors Tables

The staff and doctors have been split into two related tables: Staff, and Doctors. The staff table acts as a template for all staff members, storing information that is relevant to all staff, including doctors. This can be extended with another table (e.g. Doctors) for specific role-based information with a one to zero-or-one relation. This allows additional information to be stored for doctors (like speciality), while keeping data redundancy to a minimum and maintaining 3rd normal form. Because it extends the Staff table, it uses a foreign-primary key of Staff_ID from the Staff table.

If a staff member is a doctor, this will be shown in the ‘Job_Title’ column of the staff table, and they will have a corresponding record in the “Doctors” table.

Appointments

Appointments can have five different states: ‘Pending’, ‘Approved’, ‘Completed’, ‘Cancelled’, and ‘Missed’. These are each stored in a column named “Status” in the database, and are used to track the process of an appointment. Because any patient can create an appointment, there is a possibility for a denial of service attack where a patient creates lots of false appointments; This reduces the impact as each appointment must be approved by an authorised staff member before assigning any resources (such as doctors).

Medical Records Table

Medical records are assigned to patients, and can be linked to an appointment. If it is linked to an appointment, the appointment status is automatically updated to ‘Completed’. y, as a patient may need more than one medical record to view change over time, or have separate records for different types of medical information. This also keeps the option of updating the current medical record.

Insurance

Although out of scope, an insurance table has been included in the Entity Relationship Diagram (See Figure 1), as this shows how it would connect to the rest of the database if it were implemented.

Access Control

Roles

As different users will need to interact with the database differently, and require different levels of access. The database implements role-based access control (See Table 1). This is an effective and scalable way to enhance security by enforcing the principle of least privilege, which restricts users to only the necessary access.

These roles have been enforced using a series of different features: row-level security has been used to limit certain actions to a specific row, such as Patients only being able to access their data; Views have been used to restrict the data columns that certain users can see; Functions have been used to create an extra layer of abstraction between the user and the database, and will be used to prevent giving the user direct access to the database.

Role Name	**Access**	**Members (with login)**
Patient	Can view and edit their own data, Can view medical records, Can request appointments	Patient1, Patient2
Staff	Can view all patient data, Can view own staff data, Can edit appointment status and assign a doctor	Staff1
Doctor	Can view upcoming appointments, Can view patient data, Can edit or create patient medical records	Doctor1, Doctor2
Admin	Can view all staff data, Can create or remove staff members, Can add or remove doctors	Admin1
Postgres	Superuser	None

Table 1: List of user roles and their corresponding access

Functions

No roles (except postgres) will have direct access to write operations (e.g. INSERT, UPDATE). Instead, these are performed with the use of designated plpgsql functions, with different roles having access to different functions.

A list of all functions and parameters required can be seen in table 2, along with the parameters each function takes.

Function Name	**Description**	**Parameters**
registerPatient	Will create a new patient in the “Patients” table	First Name :: varchar, Last Name :: varchar, DoB :: date(YYYY-MM-DD), Gender :: varchar, Email :: varchar, Phone :: varchar, Emergency Phone :: varchar, Username :: varchar
registerStaff	Will create a new staff member in the “Staff” table	First Name :: varchar, Last Name :: varchar, Department :: varchar, Username :: varchar
registerDoctor	Will create a new record with the “Job_Title” of ‘Doctor’ in the “Staff” table, and create a corresponding record in the “Doctors” table	First Name :: varchar, Last Name :: varchar, Department :: varchar, Username :: varchar, Speciality :: varchar
requestAppointment	Create a new record in “Appointments” with “Status” set to ‘Pending’ and without a doctor assigned	Username :: varchar, Date and Time :: timestamp without time zone, Reason for visit :: text
approveAppointment	Sets “Status” of an appointment to ‘Approved’, and assigns a doctor by ID	Appointment ID :: int Doctor ID :: int
cancelAppointment	Sets the “Status” of an appointment to ‘Cancelled’	Appointment ID :: int
createMedicalRecord	Creates a new medical record linking to an appointment and sets the appointment’s “Status” to ‘Completed’	Patient ID :: int Appointment ID :: int Diagnosis :: text Prescription :: text Allergies :: text

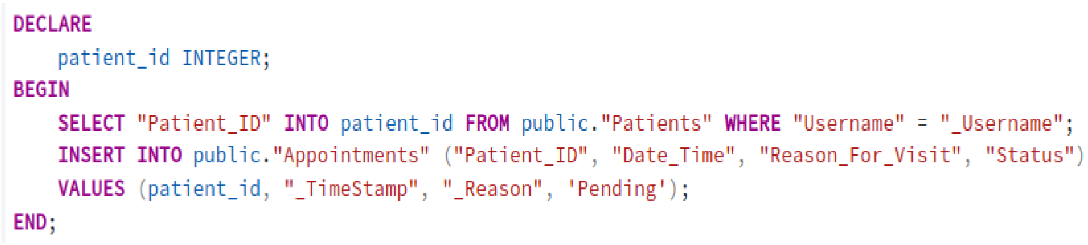
Table 2: List of plpgsql functions

Function Implementation

The following section demonstrates the implementation of the back-end logic by showing and explaining the plpgsql code used to create a handful of the functions in the database.

Appointment Scheduling

When a new appointment is created, it will automatically be given the status ‘Pending’. Admins or receptionists should then be able to view all appointments with this status, and update them by assigning a doctor and changing the status to ‘Approved’. It should be possible for either the patient or the admin to cancel the appointment.

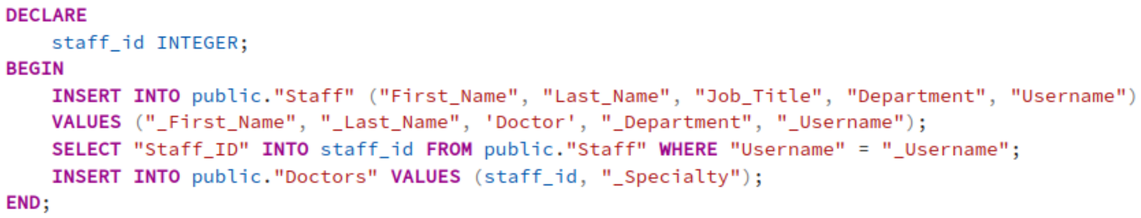


```
DECLARE
    patient_id INTEGER;
BEGIN
    SELECT "Patient_ID" INTO patient_id FROM public."Patients" WHERE "Username" = "_Username";
    INSERT INTO public."Appointments" ("Patient_ID", "Date_Time", "Reason_For_Visit", "Status")
    VALUES (patient_id, "_TimeStamp", "_Reason", 'Pending');
END;
```

Figure 2: Plpgsql for requesting an appointment

Registering a Doctor

Registering a doctor involves also creating a record in the “Staff” table. This is done first, followed by retrieving the automatically generated primary key, and copying it to the “Doctors” table, along with the speciality.

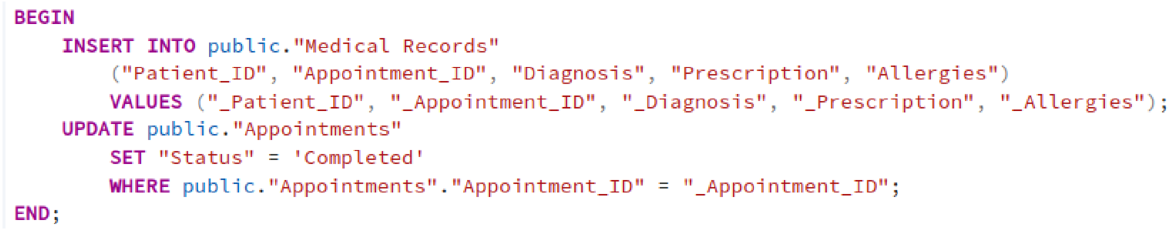


```
DECLARE
    staff_id INTEGER;
BEGIN
    INSERT INTO public."Staff" ("First_Name", "Last_Name", "Job_Title", "Department", "Username")
    VALUES ("_First_Name", "_Last_Name", 'Doctor', "_Department", "_Username");
    SELECT "Staff_ID" INTO staff_id FROM public."Staff" WHERE "Username" = "_Username";
    INSERT INTO public."Doctors" VALUES (staff_id, "_Specialty");
END;
```

Figure 3: Plpgsql code for registering a doctor

Creating a Medical record

Creating a medical record works in a similar way to the others, and inserts a new record, however if an Appointment ID passed in as a parameter, it will automatically set the status of the appointment to ‘Completed’.

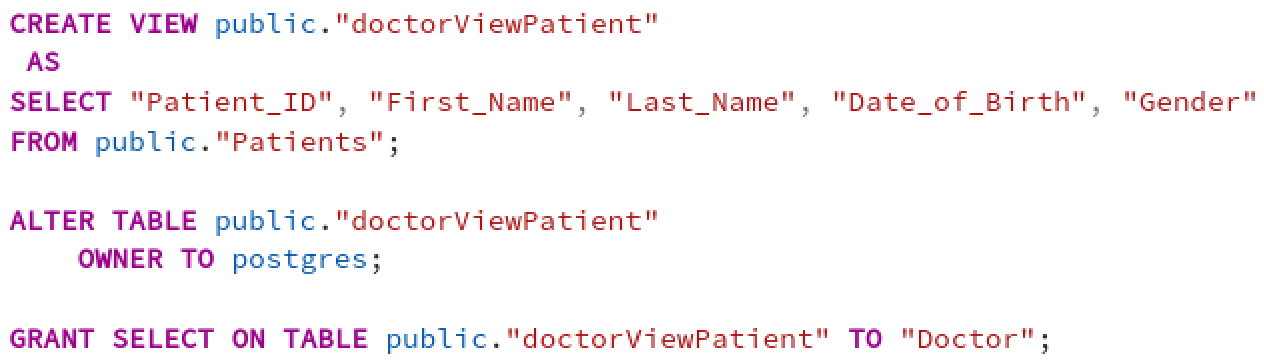


```
BEGIN
    INSERT INTO public."Medical Records"
        ("Patient_ID", "Appointment_ID", "Diagnosis", "Prescription", "Allergies")
        VALUES ("_Patient_ID", "_Appointment_ID", "_Diagnosis", "_Prescription", "_Allergies");
    UPDATE public."Appointments"
        SET "Status" = 'Completed'
        WHERE public."Appointments"."Appointment_ID" = "_Appointment_ID";
END;
```

Views

Users can still have access to read operations (e.g. SELECT), but these will leverage row-level security and views to restrict access, enforce least privilege and maintain GDPR compliance.

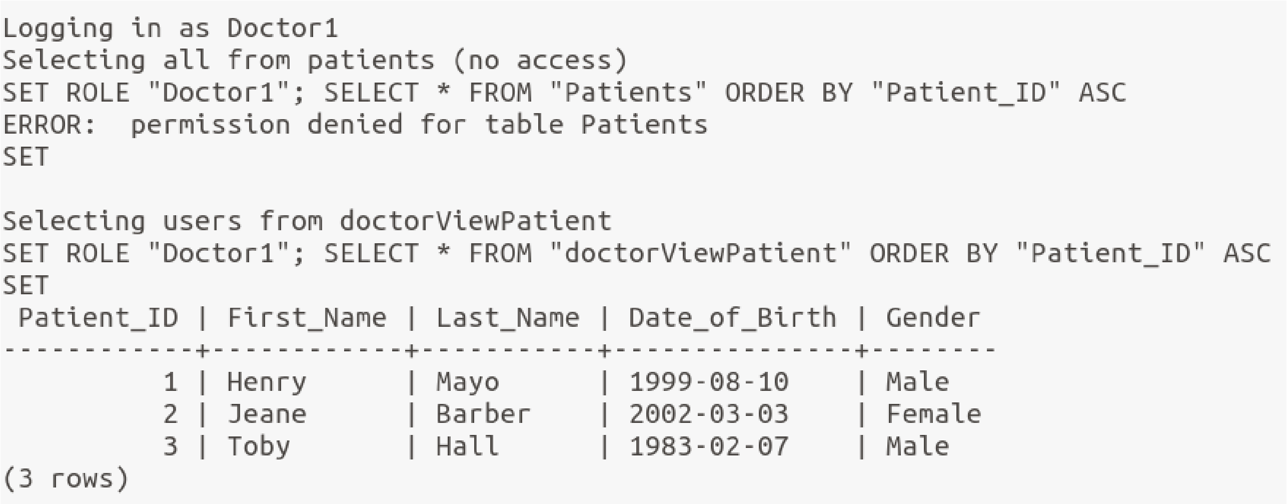
Views have been used to create a temporary table, which contains only the necessary information for the user to see. To show how these should be implemented, the view “doctorPatientView” has been created, which the “Doctor” role has been granted SELECT access on (See Figure 4).



```
CREATE VIEW public."doctorViewPatient"
 AS
SELECT "Patient_ID", "First_Name", "Last_Name", "Date_of_Birth", "Gender"
FROM public."Patients";

ALTER TABLE public."doctorViewPatient"
    OWNER TO postgres;

GRANT SELECT ON TABLE public."doctorViewPatient" TO "Doctor";
```

Figure 4: SQL code to create a view for the "Doctor" role on the "Patients" table

As well as this, the “Doctor” role is denied access to the regular “Patients” table (See testing in Figure 5).



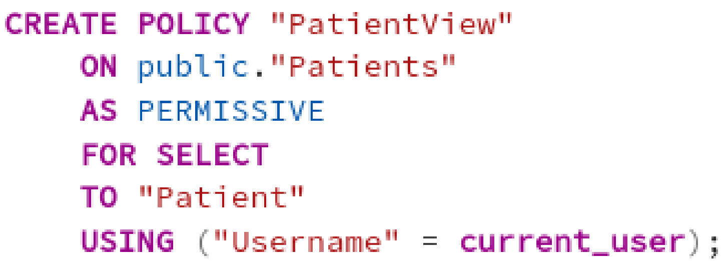
```
Logging in as Doctor1
Selecting all from patients (no access)
SET ROLE "Doctor1"; SELECT * FROM "Patients" ORDER BY "Patient_ID" ASC
ERROR:  permission denied for table Patients
SET

Selecting users from doctorViewPatient
SET ROLE "Doctor1"; SELECT * FROM "doctorViewPatient" ORDER BY "Patient_ID" ASC
SET
 Patient_ID | First_Name | Last_Name | Date_of_Birth | Gender
------------+------------+-----------+---------------+--------
          1 | Henry      | Mayo      | 1999-08-10    | Male
          2 | Jeane      | Barber    | 2002-03-03    | Female
          3 | Toby       | Hall      | 1983-02-07    | Male
(3 rows)
```

Figure 5: Testing "doctorPatientView" as “Doctor1”

Row-Level Security

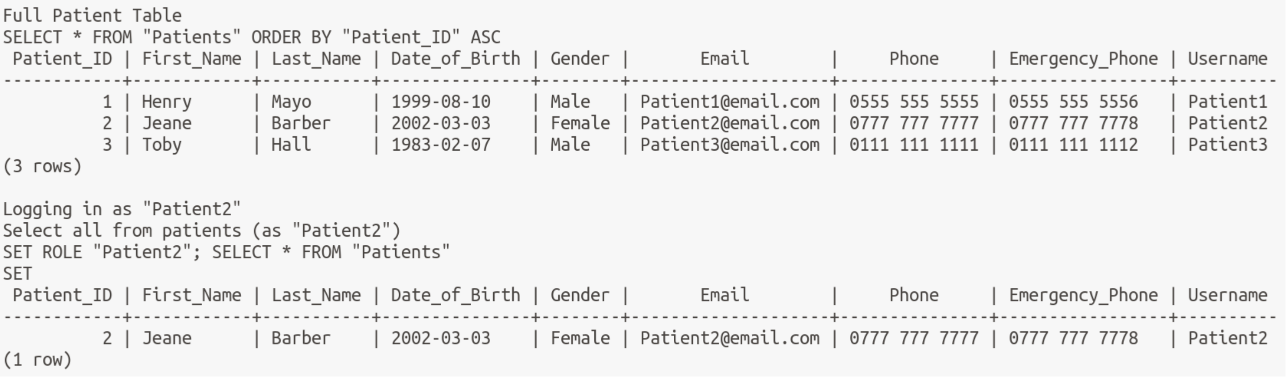
Row-Level Security is also being used to ensure that users can only view their own information, adding defence in depth when combined with secure back-end logic. This will be simulated in the database with the use of individual roles for different users, although this will be handled differently during implementation for better scalability. For simplicity, a username has been added to match the name of the user role.

To show how it should work, RLS has been implemented on the “Patients” table, ensuring that a patient is unable to view the information of another patient. Figure 6 shows the policy used to do this, and figure 7 shows this being tested.



```
CREATE POLICY "PatientView"
    ON public."Patients"
    AS PERMISSIVE
    FOR SELECT
    TO "Patient"
    USING ("Username" = current_user);
```

Figure 6: A policy for RLS on the "Patients" table



```
Full Patient Table
SELECT * FROM "Patients" ORDER BY "Patient_ID" ASC
 Patient_ID | First_Name | Last_Name | Date_of_Birth | Gender |       Email        |     Phone     | Emergency_Phone | Username
------------+------------+-----------+---------------+--------+--------------------+---------------+-----------------+----------
          1 | Henry      | Mayo      | 1999-08-10    | Male   | Patient1@email.com | 0555 555 5555 | 0555 555 5556   | Patient1
          2 | Jeane      | Barber    | 2002-03-03    | Female | Patient2@email.com | 0777 777 7777 | 0777 777 7778   | Patient2
          3 | Toby       | Hall      | 1983-02-07    | Male   | Patient3@email.com | 0111 111 1111 | 0111 111 1112   | Patient3
(3 rows)

Logging in as "Patient2"
Select all from patients (as "Patient2")
SET ROLE "Patient2"; SELECT * FROM "Patients"
SET
 Patient_ID | First_Name | Last_Name | Date_of_Birth | Gender |       Email        |     Phone     | Emergency_Phone | Username
------------+------------+-----------+---------------+--------+--------------------+---------------+-----------------+----------
          2 | Jeane      | Barber    | 2002-03-03    | Female | Patient2@email.com | 0777 777 7777 | 0777 777 7778   | Patient2
(1 row)
```

Figure 7: Test of RLS on "Patients" table

Testing Script

A testing script called ‘testing_script.sh’ has been attached along with the database files. This will show tests done on the created functions, views and row-level security by running, showing the commands used, and printing the output to the terminal.