Database Design for a Healthcare Provider

5539006

Introduction

This report details the design of a prototype back-end database for a healthcare provider. The database is designed to execute stored functions to fulfil data requirements for the client, simplifying systems that are used in conjunction with the database.

Requirements

The database is designed to be integrated with a separately-developed web application which will keep the database hidden, providing basic services e.g. user registration and authentication. It is therefore assumed that user authentication is handled by the web application, and that queries are trusted.
The database needs to be able to handle data about employees and patients, including medical, insurance, and appointment data. It should also be scalable so that the business may grow to include services such as remote patient monitoring or telehealth consultations. This is not currently implemented, but the architecture of the database must be capable of accommodating such growth.

Functional requirements explicitly provided by the client:

- Patients can register to the system.
- Patients can manage their medical records.
- Patients can book appointments.
- Patients can view their billing history.
- Patients can manage insurance information.
- Medical staff can access and update patient records.
- Administrative staff manage scheduling of appointments.
- Administrative staff manage billing of patients for appointments.

Non-functional requirements explicitly provided by the client:

- System is scalable to support future features.
- System should be GDPR-compliant, and therefore only storing minimal necessary data.

Data Model and Definition

The data model that this database is based on is described below using an Entity-Relationship Diagram (ERD) with "Crow's Foot" Notation, designed to be scalable and easy to use. It is also normalised according to Codd's 3NF, involving atomic values and key dependencies.

The postgresql extension "pgcrypto" has been used to generate random uuids for the objects.

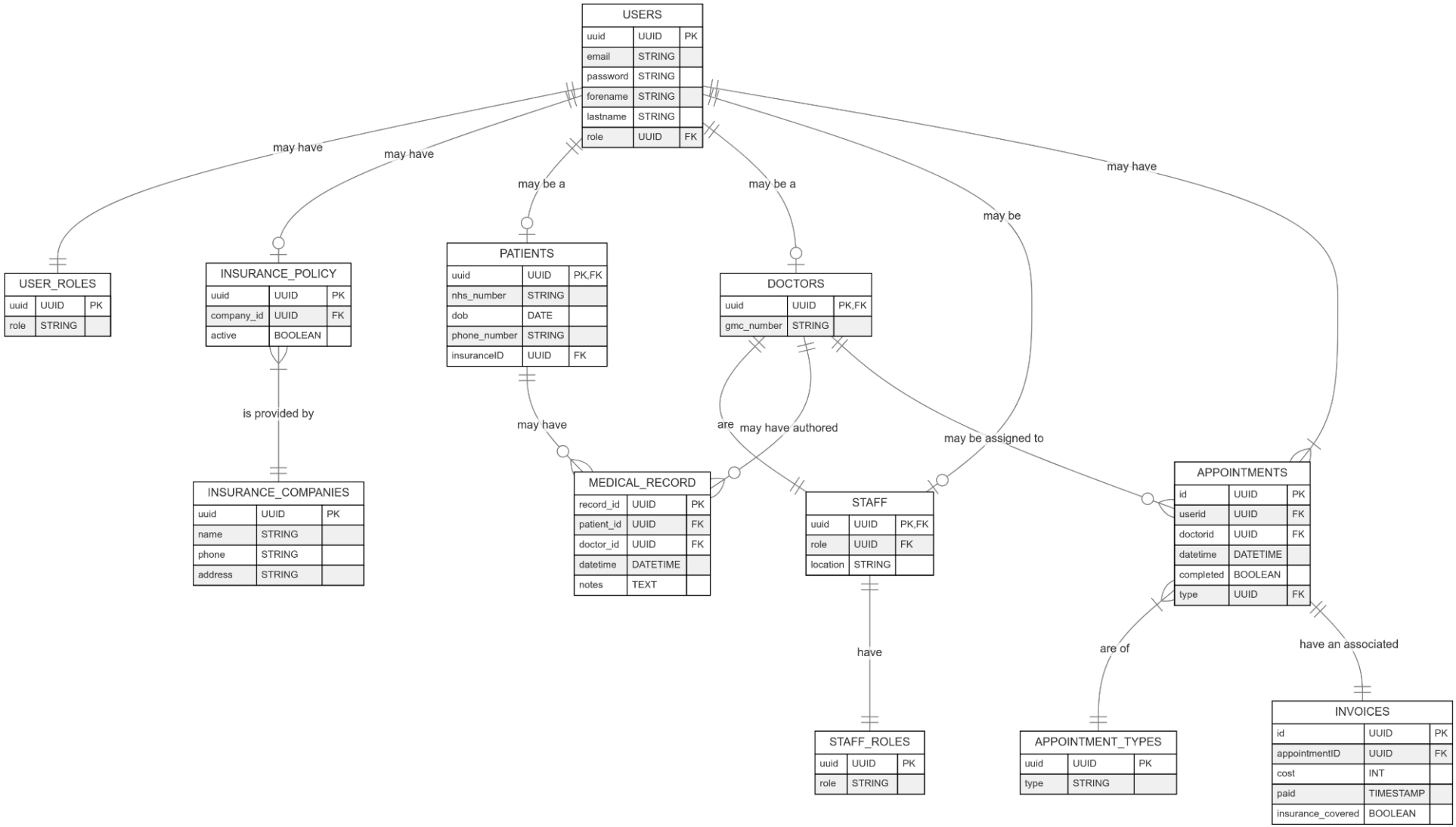

Figure: An ERD defining the database structure

Users

This table contains any user of the system, including both patients and staff, since they will all need to have names, passwords, emails etc. This avoids duplicating the same data structure across multiple tables, and further information can be obtained from specific-role tables.
The password for the users is hashed using SHA-256, and it is assumed that other information has been checked against formatting requirements in the web application (e.g. email).

```
CREATE TABLE
    IF NOT EXISTS USERS (
        uuid UUID NOT NULL PRIMARY KEY DEFAULT gen_random_uuid(),
        email TEXT NOT NULL,
        password TEXT NOT NULL,
        forename TEXT NOT NULL,
        lastname TEXT NOT NULL,
        role UUID REFERENCES USER_ROLES (uuid)
    );
```

Figure: the DDL definition of the users table

When an INSERT or UPDATE operation is performed on any row, the trigger hashes the password before it is stored, to ensure only hashes are stored, rather than plaintext passwords.

```
CREATE OR REPLACE FUNCTION hash_password() RETURNS TRIGGER AS $$
    BEGIN
        NEW.password = digest(NEW.password, 'sha256');
        RETURN NEW;
    END
$$ LANGUAGE plpgsql;

CREATE TRIGGER user_update_insert_trigger
BEFORE INSERT OR UPDATE ON USERS
FOR EACH ROW EXECUTE FUNCTION hash_password();
```

Figure: the definition of the password hashing function

User_Roles

This table is an ENUM-like structure which contains the set of roles a user may have, but they may only have one. By creating the roles in a separate table, it is very easy to expand to incorporate more roles for users.

```
CREATE TABLE IF NOT EXISTS USER_ROLES (
    uuid UUID NOT NULL PRIMARY KEY DEFAULT gen_random_uuid(),
    role TEXT NOT NULL UNIQUE
);

INSERT INTO USER_ROLES (role) VALUES ('Patient');
INSERT INTO USER_ROLES (role) VALUES ('Staff');
```

Figure: the definition and insertion of the User_Roles table

Staff

This table contains a list of users that are staff within the business, identified by their user uuid. It also contains a role from the STAFF_ROLES table that identifies the type of staff the user is. It also includes a location column in case the staff for a certain site need to be viewed.

```
CREATE TABLE IF NOT EXISTS STAFF (
    uuid UUID NOT NULL PRIMARY KEY REFERENCES USERS(uuid),
    role UUID NOT NULL REFERENCES STAFF_ROLES(uuid),
    location TEXT NOT NULL
);
```

Figure: the definition for the Staff table

Staff_Roles

This table features an ENUM-like design, containing the more specific roles a member of staff might have. Again, they may only have one role, and it's very easy to scale to add new staff roles because of this structure.

```sql
CREATE TABLE STAFF_ROLES (
    uuid UUID NOT NULL PRIMARY KEY DEFAULT gen_random_uuid(),
    role TEXT NOT NULL UNIQUE
);

INSERT INTO STAFF_ROLES (role) VALUES ('Doctor');
INSERT INTO STAFF_ROLES (role) VALUES ('Administrator');
```

Figure: the definition and insertion of the Staff_Roles table

Doctors

This table stores the specific details about doctors, identified by their user uuid, including their General Medical Council (GMC) number that uniquely identifies medical professionals in the UK. It is assumed that when a doctor is registered, that their GMC number is verified to be in the correct format by the web application.

```sql
CREATE TABLE IF NOT EXISTS DOCTORS (
    uuid UUID NOT NULL PRIMARY KEY REFERENCES USERS(uuid),
    gmc_number TEXT NOT NULL
);
```

Figure: the definition of the Doctors table

Insurance_Companies

This table stores information about the companies that patients have taken insurance out from, including contact details to claim invoices for patient appointments.

```
CREATE TABLE IF NOT EXISTS INSURANCE_COMPANIES (
    uuid UUID NOT NULL PRIMARY KEY DEFAULT gen_random_uuid(),
    name TEXT NOT NULL,
    phone TEXT NOT NULL,
    address TEXT NOT NULL
);
```

Figure: the definition of the Insurance_Companies table

Insurance_Policy

This table stores specific information about a patient's insurance policy (if they have one), including the provider and the current status of the policy.

```
CREATE TABLE IF NOT EXISTS INSURANCE_POLICY (
    uuid UUID NOT NULL PRIMARY KEY DEFAULT gen_random_uuid(),
    company_id UUID NOT NULL REFERENCES INSURANCE_COMPANIES(uuid),
    active BOOLEAN NOT NULL
);
```

Figure: the definition of the Insurance_Policy table

Patients

This table stores information specific to patients such as their NHS number, date of birth, phone number and insurance policy ID (if they have private health insurance).

```
CREATE TABLE IF NOT EXISTS PATIENTS (
    uuid UUID NOT NULL PRIMARY KEY REFERENCES USERS(uuid),
    nhs_number TEXT NOT NULL,
    dob DATE NOT NULL,
    phone_number TEXT NOT NULL,
    insuranceID UUID REFERENCES INSURANCE_POLICY(uuid)
);
```

Figure: the definition of the Patients table

Appointments

This table records all scheduled and past appointments, including the attending doctor, the state of the appointment and the type of appointment (e.g. face-to-face etc.)

```
CREATE TABLE IF NOT EXISTS APPOINTMENTS (
    id uuid NOT NULL PRIMARY KEY DEFAULT gen_random_uuid(),
    userid uuid NOT NULL REFERENCES PATIENTS(uuid),
    doctorid uuid REFERENCES DOCTORS(uuid),
    datetime DATE NOT NULL,
    completed BOOLEAN NOT NULL DEFAULT FALSE,
    type uuid NOT NULL REFERENCES APPOINTMENT_TYPES(uuid)
);
```

Figure: the definition for the Appointments table

Appointment_Types

This table, similar to User_Roles and Staff_Roles, stores the possible types that an appointment can be in an ENUM-like data structure. Currently, this is only face-to-face appointments but this may change in future, and having this table makes it easy for the database to accommodate that.

```
CREATE TABLE IF NOT EXISTS APPOINTMENT_TYPES (
    uuid UUID NOT NULL PRIMARY KEY DEFAULT gen_random_uuid(),
    type TEXT NOT NULL UNIQUE
);

INSERT INTO APPOINTMENT_TYPES (type) VALUES ('Face-to-face');
```

Figure: the definition and insertion of the Appointment_Types table

Invoices

This stores the details about an invoice for an appointment (identified by appointmentID), including the cost, whether or not it has been paid, and whether a patient's insurance covered it.

```
CREATE TABLE IF NOT EXISTS INVOICES (
    id uuid NOT NULL PRIMARY KEY DEFAULT gen_random_uuid(),
    appointmentID uuid NOT NULL REFERENCES APPOINTMENTS(id),
    cost INT NOT NULL,
    paid BOOLEAN NOT NULL DEFAULT FALSE,
    insurance_covered BOOLEAN
);
```

Figure: the definition of the Invoices table

Medical_Record

This table stores records of medical encounters, including the patient_id, doctor_id, and any notes they made about the appointment.

```
CREATE TABLE IF NOT EXISTS MEDICAL_RECORD (
    record_id UUID NOT NULL PRIMARY KEY DEFAULT gen_random_uuid(),
    patient_id UUID NOT NULL REFERENCES PATIENTS(uuid),
    doctor_id UUID NOT NULL REFERENCES DOCTORS(uuid),
    timestamp DATE NOT NULL,
    notes TEXT NOT NULL
);
```

Figure: the definition of the Medical_Record table

User Journeys

User journeys are standard, typical user interactions with the system, and the stages involved in these processes. Functional logic is implemented using postgreSQL functions. It is assumed that authentication is handled at the web application layer, and as such only authorised requests are queried. However, this may not be the case so additional authorisation checks can be performed as functions.

Registration

A user's data will need to be added to the database the first time they use the service. This user could have any role, whether patient, doctor, or other staff. It is within the context of the web application to ensure that the correct function for the role is called to register a user e.g. not allowing public users to register as staff members.
Depending on the type of user, different information is required, so I have created a function for each role.

```
CREATE OR REPLACE FUNCTION register_patient(
    email TEXT, password TEXT, forename TEXT, lastname TEXT,
    nhs_number TEXT, dob DATE, phone_number TEXT
) RETURNS UUID LANGUAGE plpgsql AS
    $$
        DECLARE
            userid UUID;
            roleid UUID;
        BEGIN
            SELECT uuid FROM user_roles WHERE role = 'Patient' INTO roleid;
            INSERT INTO users (email, password, forename, lastname, role)
            VALUES (email, password, forename, lastname, roleid)
            RETURNING uuid INTO userid;
            INSERT INTO patients (uuid, nhs_number, dob, phone_number)
            VALUES (userid, nhs_number, dob, phone_number);
            RETURN userid;
        END
    $$;
```

Figure: the function to register a new patient [register_patient]

```
CREATE OR REPLACE FUNCTION register_doctor(
    email TEXT, password TEXT, forename TEXT, lastname TEXT,
    location TEXT, gmc_number TEXT
) RETURNS UUID LANGUAGE plpgsql AS
    $$
        DECLARE
            userid UUID;
            roleid UUID;
            staffroleid UUID;
        BEGIN
            SELECT uuid FROM user_roles WHERE role = 'Staff' INTO roleid;
            SELECT uuid FROM staff_roles WHERE role = 'Doctor' INTO staffroleid;
            INSERT INTO users (email, password, forename, lastname, role)
            VALUES (email, password, forename, lastname, roleid)
            RETURNING uuid INTO userid;
            INSERT INTO staff (uuid, role, location)
            VALUES (userid, staffroleid, location);
            INSERT INTO doctors (uuid, gmc_number)
            VALUES (userid, gmc_number);
            RETURN userid;
        END
    $$;
```

Figure: the function to register a new doctor [register_doctor]

```
CREATE OR REPLACE FUNCTION register_administrator(
    email TEXT, password TEXT, forename TEXT, lastname TEXT, location TEXT
) RETURNS UUID LANGUAGE plpgsql AS
    $$
        DECLARE
            userid UUID;
            roleid UUID;
            staffroleid UUID;
        BEGIN
            SELECT uuid FROM user_roles WHERE role = 'Staff' INTO roleid;
            SELECT uuid FROM staff_roles WHERE role = 'Administrator' INTO staffroleid;
            INSERT INTO users (email, password, forename, lastname, role)
            VALUES (email, password, forename, lastname, roleid)
            RETURNING uuid INTO userid;
            INSERT INTO staff (uuid, role, location)
            VALUES (userid, staffroleid, location);
            RETURN userid;
        END
    $$;
```

Figure: the function to register a new administrator [register_administrator]

Authentication

On the other hand, one authentication function works for every role. The user has their entered email and password checked against the database, after hashing. The *authenticate* function here calls the compare_passwords method to check the user details.

```
CREATE OR REPLACE FUNCTION hash_password(password TEXT)
RETURNS TEXT LANGUAGE plpgsql AS
    $$
    BEGIN
        RETURN digest(password, 'sha256');
    END
    $$;
```

Figure: the function to hash and return a password using the SHA-256 algorithm [hash_password]

```
CREATE OR REPLACE FUNCTION compare_passwords(password TEXT, hash TEXT)
RETURNS BOOLEAN LANGUAGE PLPGSQL AS
    $$
        BEGIN
            RETURN digest(password, 'sha-256') = hash::bytea;
        END
    $$;
```

Figure: the function to compare a given password with a given hash using SHA-256 [compare_passwords]

```
CREATE OR REPLACE FUNCTION authenticate (input_email TEXT, input_password TEXT)
RETURNS UUID LANGUAGE plpgsql AS
    $$
        DECLARE
            userid UUID;
        BEGIN
            SELECT uuid FROM users WHERE email = input_email AND compare_passwords(input_password, password) INTO userid;
            RETURN userid;
        END
    $$;
```

Figure: the authenticate function to check if a user's inputted credentials are valid [authenticate]

Authorisation

It is likely that authorisation will be handled by the web application, but in case of bugs, or malicious attacks, some authorisation can happen at the backend as well, by checking the role(s) of the given user uuid. I have created four procedures (different from functions as they don't return values) that will throw exceptions if a user doesn't have permission. The four privilege levels are: patient, staff, administrator, and doctor.

```
CREATE OR REPLACE PROCEDURE check_is_patient(userid UUID)
LANGUAGE plpgsql AS
    $$
        BEGIN
            IF NOT EXISTS (SELECT 1 FROM patients WHERE uuid = userid) THEN
                RAISE EXCEPTION 'Unauthorised';
            END IF;
        END;
    $$;
```

Figure: procedure to raise exception if user is not a patient [check_is_patient]

```
CREATE OR REPLACE PROCEDURE check_is_staff(userid UUID)
LANGUAGE plpgsql AS
    $$
        BEGIN
            IF NOT EXISTS (SELECT 1 FROM staff WHERE uuid = userid) THEN
                RAISE EXCEPTION 'Unauthorised';
            END IF;
        END;
    $$;
```

Figure: procedure to raise exception if user is not staff [check_is_staff]

```
CREATE OR REPLACE PROCEDURE check_is_administrator(userid UUID)
LANGUAGE plpgsql AS
    $$
        DECLARE
            roleid UUID;
        BEGIN
            SELECT uuid FROM staff_roles WHERE role = 'Administrator' INTO roleid;
            IF NOT EXISTS (SELECT 1 FROM staff WHERE uuid = userid AND role = roleid) THEN
                RAISE EXCEPTION 'Unauthorised';
            END IF;
        END;
    $$;
```

Figure: procedure to raise exception if user is not an administrator [check_is_administrator]

```
CREATE OR REPLACE PROCEDURE check_is_doctor(userid UUID)
LANGUAGE plpgsql AS
    $$
        BEGIN
            IF NOT EXISTS (SELECT 1 FROM doctors WHERE uuid = userid) THEN
                RAISE EXCEPTION 'Unauthorised';
            END IF;
        END;
    $$;
```

Figure: procedure to raise exception if user is not a doctor [check_is_doctor]

Managing Appointments

To book an appointment, a user takes the necessary steps and selects a time slot for the appointment. However, it is also necessary for an administrator to allocate a doctor to the appointment, depending on availability. It has been assumed, for simplicity, that all appointments start on the hour, and are an hour long. In future, this could be changed to reflect the needs of the client.

```
CREATE OR REPLACE FUNCTION make_booking(userid UUID, datetime TIMESTAMP)
RETURNS UUID LANGUAGE PLPGSQL AS
    $$
        DECLARE
            typeid UUID;
            apptid UUID;
        BEGIN
            CALL check_is_patient(userid);
            IF EXTRACT(MINUTE FROM datetime) <> 0 OR
            EXTRACT(SECOND FROM datetime) <> 0 THEN
                RAISE EXCEPTION 'Invalid Appointment Time';
            END IF;
            SELECT uuid from appointment_types WHERE type = 'Face-to-face' INTO typeid;
            INSERT INTO appointments (userid, datetime, type) VALUES (userid, datetime, typeid)
            RETURNING id INTO apptid;
            RETURN apptid;
        END
    $$;
```

Figure: the function allowing a patient to make a booking [make_booking]

The administrative staff need to find out which doctors are free at the appointment time, so they can assign one to that appointment.

```
CREATE OR REPLACE FUNCTION get_free_doctors(at_time TIMESTAMP)
RETURNS SETOF UUID LANGUAGE plpgsql AS
    $$
        BEGIN
            RETURN QUERY
                SELECT uuid FROM doctors
                LEFT JOIN appointments ON appointments.doctorid = doctors.uuid
                AND appointments.datetime = at_time
                GROUP BY doctors.uuid HAVING COUNT(appointments.id) = 0;
        END
    $$;
```

Figure: function that searches for doctors with no appointments at the specified time [get_free_doctors]

The administrative staff then need to assign a doctor to the appointment.

```
CREATE OR REPLACE PROCEDURE assign_doc_appt(patient UUID, appt UUID, doc UUID)
LANGUAGE plpgsql AS
    $$
        BEGIN
            CALL check_is_administrator(patient);
            UPDATE appointments SET doctorid = doc
            WHERE id = appt;
        END
    $$;
```

Figure: procedure to assign a doctor to an appointment [assign_doc_appt]

And a patient needs to be able to cancel their appointment.

```
CREATE OR REPLACE PROCEDURE cancel_appt(patient UUID, appt_time TIMESTAMP)
LANGUAGE plpgsql AS
    $$
        DECLARE
            apptid UUID;
        BEGIN
            SELECT uuid FROM appointments WHERE userid = patient AND datetime = appt_time INTO apptid;
            CALL check_is_patient(patient);
            DELETE FROM appointments WHERE userid = patient AND uuid = apptid AND datetime > NOW();
        END
    $$;
```

Figure: procedure for a patient to cancel their appointment [cancel_appt]

Managing Medical Records

Patients need to be able to see and manage their medical records. In this context, I have interpreted manage to mean withdrawing consent for the hospital to store and process that information - having to delete the records. In order to ensure that data isn't just destroyed, a copy of the medical records should be returned to the patient.

```
CREATE OR REPLACE FUNCTION delete_medical_records(userid UUID)
RETURNS TABLE(record_id UUID, datetime TIMESTAMP, doctor_id UUID, notes TEXT)
LANGUAGE plpgsql AS
    $$
        BEGIN
            CALL check_is_patient(userid);
            RETURN QUERY
                DELETE FROM medical_record WHERE patient_id = userid
                RETURNING record_id, timestamp, doctor_id, notes;
        END
    $$;
```

Figure: function allowing a patient to delete their medical records [delete_medical_records]

All staff and the patient themself can access their medical records.

```
CREATE OR REPLACE FUNCTION retrieve_medical_records(patient UUID)
RETURNS TABLE(record_id UUID, datetime DATE, doctor_id UUID, notes TEXT)
LANGUAGE plpgsql AS
    $$
        BEGIN
            RETURN QUERY
                SELECT m.record_id, m.datetime, m.doctor_id, m.notes
                FROM medical_record AS m WHERE patient_id = patient;
        END
    $$;
```

Figure: function to obtain a patient's medical records [retrieve_medical_records]

Only a doctor can link a new medical record to a patient's account.

```
CREATE OR REPLACE FUNCTION add_medical_record(patient UUID, doctor UUID, notes TEXT)
RETURNS UUID LANGUAGE plpgsql AS
    $$
        DECLARE
            recordid UUID;
        BEGIN
            CALL check_is_doctor(doctor);
            INSERT INTO medical_record (patient_id, doctor_id, datetime, notes)
            VALUES (patient, doctor, NOW(), notes)
            RETURNING record_id INTO recordid;
            RETURN recordid;
        END
    $$;
```

Figure: function for a doctor to add a medical record to a patient [add_medical_record]

Invoices

After an appointment, it is the responsibility of the administrator to create an invoice for the appointment to bill the patient (or their insurance).

```
CREATE OR REPLACE FUNCTION create_invoice(staffid UUID, appt UUID, cost INT)
RETURNS UUID LANGUAGE plpgsql AS
    $$
        DECLARE
            invoiceid UUID;
        BEGIN
            CALL check_is_administrator(staffid);
            INSERT INTO invoices (appointmentID, cost) VALUES (appt, cost) RETURNING id INTO invoiceid;
            RETURN invoiceid;
        END
    $$;
```

Figure: function to create an invoice for an appointment [create_invoice]

When the invoice is paid, the administrator will need to log the payment.

```
CREATE OR REPLACE PROCEDURE log_payment(
staffid UUID, appt UUID, insured boolean)
LANGUAGE plpgsql AS
    $$
        BEGIN
            CALL check_is_administrator(staffid);
            UPDATE invoices SET paid = NOW(), insurance_covered = insured
            WHERE appointmentID = appt;
        END
    $$;
```

Figure: procedure to log the payment for an invoice [log_payment]

If the user wants to see their billing history, they can.

```
CREATE OR REPLACE FUNCTION retrieve_billing_history(patient UUID)
RETURNS TABLE(datetime TIMESTAMP, cost INT) LANGUAGE plpgsql AS
    $$
        BEGIN
            CALL check_is_patient(patient);
            RETURN QUERY
                SELECT paid, amount from invoices INNER JOIN appointments ON invoices.appointmentID = appointments.id
                WHERE appointments.userid = patient AND paid IS NOT NULL;
        END
    $$;
```

Figure: function for patient to retrieve their billing history [retrieve_billing history]

Insurance

Patients must themselves keep their insurance up-to-date and inform the company of any changes.

```
CREATE OR REPLACE PROCEDURE set_insurance_policy(patient UUID, provider UUID, active boolean)
LANGUAGE plpgsql AS
    $$
        BEGIN
            CALL check_is_patient(patient);
            INSERT INTO insurance_policy (uuid, company_id, active) VALUES (patient, provider, active)
            ON CONFLICT(uuid) DO UPDATE SET company_id=EXCLUDED.company_id, active=EXCLUDED.active;
        END
    $$;
```

Figure: procedure to set/update a patient's insurance information [set_insurance_policy]

Patients need to be able to remove their insurance policy if they choose to cancel it.

```
CREATE OR REPLACE PROCEDURE remove_insurance_policy(patient UUID)
LANGUAGE plpgsql AS
    $$
        DECLARE
            insuranceID UUID;
        BEGIN
            CALL check_is_patient(patient);
            SELECT insuranceID FROM patients WHERE uuid = patient INTO insuranceID;
            DELETE FROM insurance_policy WHERE uuid = insuranceID;
        END
    $$;
```

Figure: procedure to remove a patient's insurance information [remove_insurance_policy]

Staff and patients need to be able to check if a patient has active insurance at any time.

```
CREATE OR REPLACE FUNCTION insurance_check(patient UUID)
RETURNS boolean LANGUAGE plpgsql AS
    $$
        DECLARE
            insuranceID UUID;
            activeInsurance boolean;
        BEGIN
            SELECT insuranceID FROM patients WHERE uuid = patient INTO insuranceID;
            SELECT active FROM insurance_policy WHERE uuid = insuranceID INTO activeInsurance;
            RETURN activeInsurance;
        END
    $$;
```

Figure: function to check if a patient has active insurance [insurance_check]

Staff and other patients should also be able to retrieve information about the insurance company for a particular patient.

```
CREATE OR REPLACE FUNCTION get_insurer_details(patient UUID)
RETURNS TABLE(uuid UUID, name TEXT, phone TEXT, address TEXT) LANGUAGE plpgsql AS
    $$
        BEGIN
            RETURN QUERY
                SELECT uuid, name, phone, address FROM insurance_companies
                INNER JOIN insurance_policy ON insurance_companies.uuid = insurance_policy.company_id
                INNER JOIN patients ON insurance_policy.uuid = patients.insuranceID
                WHERE patients.uuid = patient;
        END
    $$;
```

Figure: function to return the insurer details for a specific patient [get_insurer_details]