1 2 9 0

UNIVERSIDADE Ð COIMBRA

Faculty of Sciences and Technology

DATABASES - BACHELOR'S DEGREE IN INFORMATICS ENGINEERING

HOSPITAL MANAGEMENT SYSTEM

Project by:

Nuno Batista 2022216127

Miguel Martins 2022213951

May, 2024

1 - Introduction

1.1 - Members and Contacts

Nuno Batista - nunomarquesbatista@gmail.com
Miguel Martins - uc2022213951@student.uc.pt

1.2 - Brief Description of the Project

This project's objective is to develop an **efficient and secure** system to manage information in a hospital's database.

Additionally, this system will also serve as a platform for the patients to schedule medical services and pay their bills. It will also take into account the possibility of multiple users accessing the system simultaneously and **solve potential concurrency conflicts**. Such conflicts will be discussed further ahead.

In this database, there will be information about doctors, nurses, assistants, patients, medications and their side effects, hospitalizations, appointments, bills, medical specializations, the hierarchical levels of the nurses, possible roles that nurses can enroll in medical services and the employee contracts.

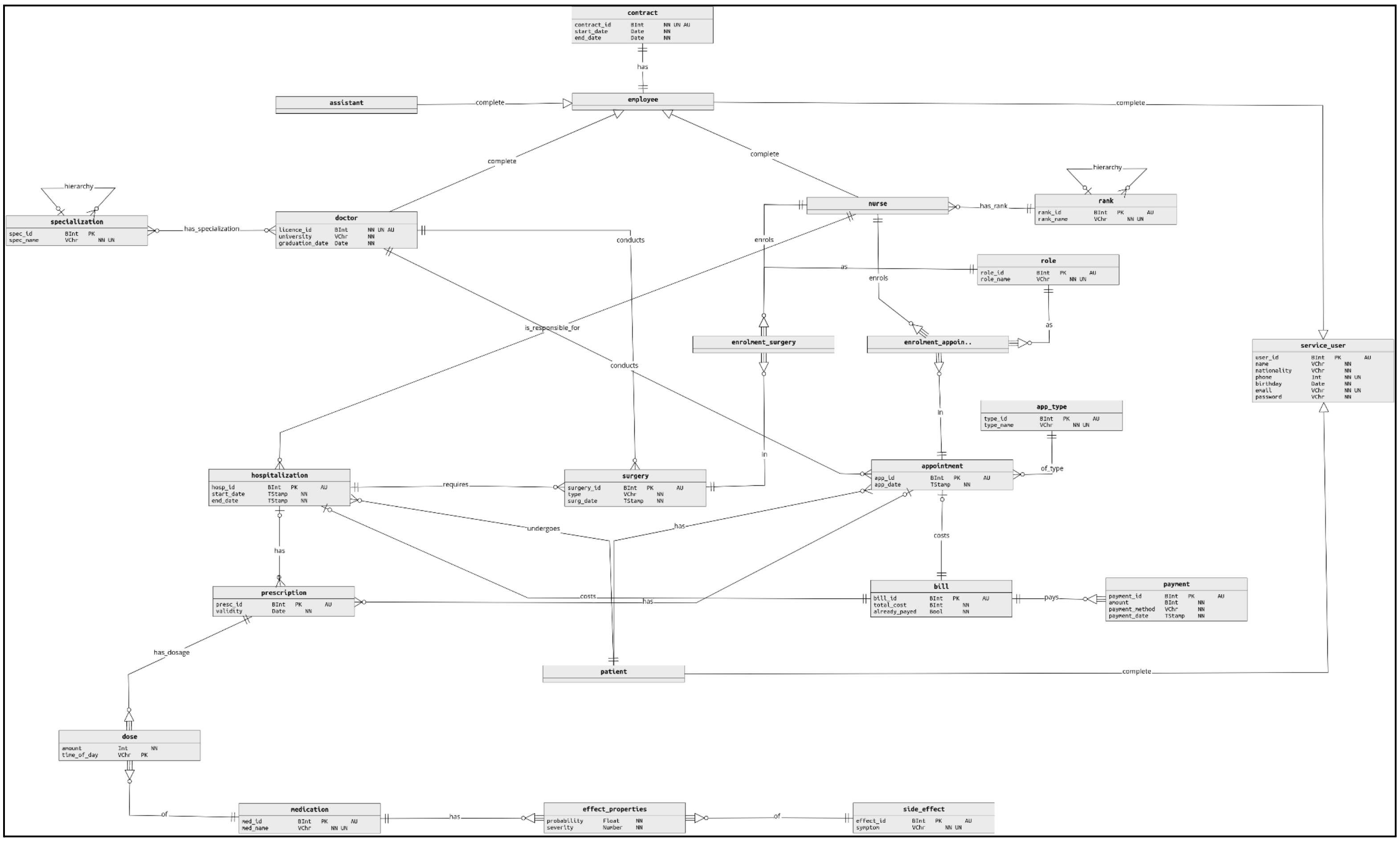
There will be 4 types of users, each with their own functionalities and traits.

1.3 - User Functionalities

- **Patient**
  - Schedule appointment;
  - Check their own appointments;
  - Check their own prescriptions;
  - Pay their bills;
- **Assistant**
  - Check any patient's appointments;
  - Check any patient's prescriptions;
  - Schedule surgeries associated to a given hospitalization (or create a new hospitalization);
  - Check the top 3 patients in terms of money spent in the current month;
  - Get a summary with details about surgeries, payments and prescriptions from a given day;
  - Generate a report containing the doctor with more surgeries performed in the last 12 months;
- **Doctor**
  - Check any patient's prescriptions
  - Add a prescription to an appointment or hospitalization;
- **Nurse**
  - Check any patient's prescriptions

2 - Entity-Relationship Model

2.1 - E-R Diagram



2.2 - Entity Attributes

➔ **service_user**

- **user_id** : *BigInt - Primary key* | Unique identifier for each user
- **name** : *VarChar - Not Null* | Name of the user
- **nationality** : *VarChar - Not Null* | User's nationality
- **phone** : *Int - Not Null, Unique* | User's phone number
- **birthday** : *Date - Not Null* | User's birth date
- **email** : *VarChar - Not Null, Unique* | User's email address
- **password**: *VarChar - Not Null* | User's password

```
--  'name' field only contains letters and spaces
name ~ '^[a-zA-Z ]+$';
--  phone - Phone number has to comprise of 9 digits
LENGTH(CAST(phone AS text)) = 9 AND CAST(phone AS text) ~ '^[0-9]+$'
-- birthday - Birthday can't be earlier than 1904
birthday < '1904-01-01'
-- email - Email has to be in the format of '<string>@<string>.<string (len>2))>'
email ~ '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
-- password - Password has to be at least 4 characters long
LENGTH(password) >= 4
```

- **patient**
- **contract**
    - **contract_id** : *BigInt - Not Null, Unique* | Unique identifier for the contract
    - **start_date** : *Date - Not Null* | Contract's start date
    - **end_date** : *Date - Not Null* | Contract's enddate

      -- start_date must be previous to end_date

      start_date >= end_date;
- **employee**
    - **nurse**
        - **rank**
            - **rank_id** : *BigInt - Primary Key* | Unique identifier for the hierarchical level
            - **rank_name** : *VarChar - Not Null, Unique* | name of the hierarchical level

              -- 'rank_name' field only contains letters and spaces

              rank_name ~ '^[a-zA-Z]+$';
    - **assistant**
    - **doctor**
        - **university**: *VarChar - Not Null* | University where the doctor graduated
        - **license_id** : *VarChar - Not Null, Unique* | Unique identifier for the doctor's license
        - **graduation_date** : *Date - Not Null* | Date when the doctor graduated

          -- 'university' field only contains letters and spaces

          university~ '^[a-zA-Z]+$';
- **specialization**
    - **spec_id**: *BigInt - Primary Key* | Unique identifier of the specialization
    - **spec_name** : *VarChar - Not Null, Unique* | Name of the specialization

      -- 'spec_name' field only contains letters and spaces

      spec_name~ '^[a-zA-Z]+$';
- **appointment**
    - **app_id** : *BigInt - Primary Key* | Unique identifier for the appointment
    - **app_date** : *Timestamp - Not Null* | Date of the appointment
- **app_type**
    - **type_id** : *BigInt - Primary Key* | Unique identifier for the appointment's type
    - **type_name** : *VarChar - Not Null, Unique* | Name of the appointment's type

      -- 'type_name' field only contains letters and spaces

      type_name ~ '^[a-zA-Z]+$';
- **bill**
    - **bill_id** : *BigInt - Primary Key* | Unique identifier for the bill
    - **total_cost** : *BigInt - Not Null* | How much the bill costs
    - **already_payed**: *BigInt - Not Null* | Amount already paid

      -- 'already_paid' must be <= than 'total_cost'

      already_paid <= total_cost

      -- 'already_paid' must be positive

      already_paid >= 0

      -- 'total_cost' must be positive

      total_cost >= 0
- **payment**
    - **payment_id** : *BigInt - Not Null, Unique* | Unique identifier for the payment
    - **amount** : *BigInt - Not Null* | How much is being payed
    - **payment_date** : *Timestamp - Not Null* | Date of the payment
    - **payment_method** : *VarChar - Not Null* | What method is being used to pay

-- 'amount' must be positive
amount > 0

- **surgery**
  - **surgery_id** : BigInt - *Primary Key* | Unique identifier for the surgery
  - **type** : *VarChar - Not Null* | Surgery's type
  - **surg_date** : *Date - Not Null* | Surgery's date

  -- 'type' field only contains letters and spaces
  type~ '^[a-zA-Z]+$';
- **role**
  - **role_id**: *BigInt - Primary Key* | Unique identifier for the role
  - **name** - *VarChar - Not Null, Unique* | Name of the role

  -- 'role_name' field only contains letters and spaces
  role_name ~ '^[a-zA-Z]+$';
- **enrolment_surgery**
- **enrolment_appointment**
- **hospitalization**
  - **hosp_id** : *BigInt - Primary Key* | Unique identifier for the hospitalization date
  - **start_date** : *Timestamp - Not Null* | Start date of the hospitalization
  - **end_date** : *Timestamp - Not Null* | End date of the hospitalization

  -- start_date must be previous to end_date
  start_date >= end_date;
- **prescription**
  - **presc_id** : *BigInt - Primary Key* | Unique identifier for the prescription
  - **validity** : *Timestamp - Not Null* | Date of the prescription
- **medication**
  - **med_id** : *BigInt - Primary Key* | Unique identifier for the medication
  - **med_name** : *VarChar - Not Null, Unique* | Name of the medication

  -- 'med_name' field only contains letters and spaces
  med_name ~ '^[a-zA-Z]+$';
- **dose**
  - **amount** : *Integer - Not Null* | Dosage
  - **time_of_day** : *VarChar - Primary Key* | Time of the day to take the medicine

  -- 'amount' must be positive
  amount > 0
  -- 'time_of_the_day' has the domain morning, afternoon or evening
  time_of_the_day IN ('morning', 'afternoon', 'evening');
- **effect_properties**
  - **probability** : *Float - Not Null* | Probability of the medicine to take effect
  - **severity** : *Number - Not Null* | Severity of the effect

  -- 'severity' must be positive
  severity> 0
  -- 'probability' must be between 0 and 1
  probability> 0 AND probability < 1
- **side_effects**
  - **effect_id** : *BigInt - Primary Key* | Unique identifier for the symptom
  - **symptom** : *VarChar - Not Null, Unique* | Symptom's name

  -- 'symptom' field only contains letters and spaces
  symptom~ '^[a-zA-Z]+$';

2.3 - Main E-R Decisions

This section aims to explain some of the **less obvious** decisions made during the creation of the ER.

2.3.1 - Reason for the service_user Entity Set

For categorization purposes and as employees and patients have several attributes in common, the set of entities *service_user* was created, making employees and patients specializations of *service_user* that inherit its attributes.

2.3.2 - Association of nurses with different roles in appointments and surgery

In order to associate a nurse with a surgery or appointment with a given role, the nurse entity set is linked to *enrolment_surgery* and *enrolment_appointment*, which, in turn, are linked to role (role contains entries for all possible nurse roles) as well as surgery and appointment. For example, if an entity nurse **N** has the **role R** in a surgery, then the entry in the *enrollment_surgery* table will have one of the foreign keys pointing to **N**, another pointing to **S** and the third pointing to **R**.

2.3.3 - Reason for using "forced" primary keys in certain tables

Taking *specialization* entity set as an example, it has the spec_name attribute denoting the name of the specialization, which, obviously, can never be repeated, however, this attribute is a string of characters, which **makes SQL queries slower** than that with integers, therefore, in the name of efficiency, the attribute *spec_id* was added, this being an integer. Similar situations are verified in the medication, *side_effect*, role and *app_type* entity sets.

2.3.4 - Hierarchy of nurse ranks and medical specializations

These hierarchies were implemented with recursive relationships, from 0..1 to 0..n, that is, seeing these relationships as trees, an entity <u>can have</u> several children as well as it <u>can have</u> a parent. Thus, *doctor* is associated with *specialization* with 0..n to 0..n (since a *doctor* entity may not have specialization, but may also have several, and different *doctor* entities may have the same specialization) and the *specialization* entities in themselves are organized hierarchically as described previously. In turn, *nurse* is associated with *rank* with 0..n to 1..1, with 0..in the cardinality of *nurse* (since a *nurse* entity must have a level of hierarchy within the hospital, and several *nurse* entities can be on the same level).

2.3.5 - Details about the prescription entity set

The *Prescription* entity can have several medications and must also define their respective doses, to describe the dose of a given medication, the set of *dose* entities was created which is weak for *prescription* and *medication*, this will define the quantity of a *medication* entity that contains a *prescription* entity, as well as the time of day and for how long the medicine should be taken.

2.3.6 - Details about the medication and side_effect entity sets

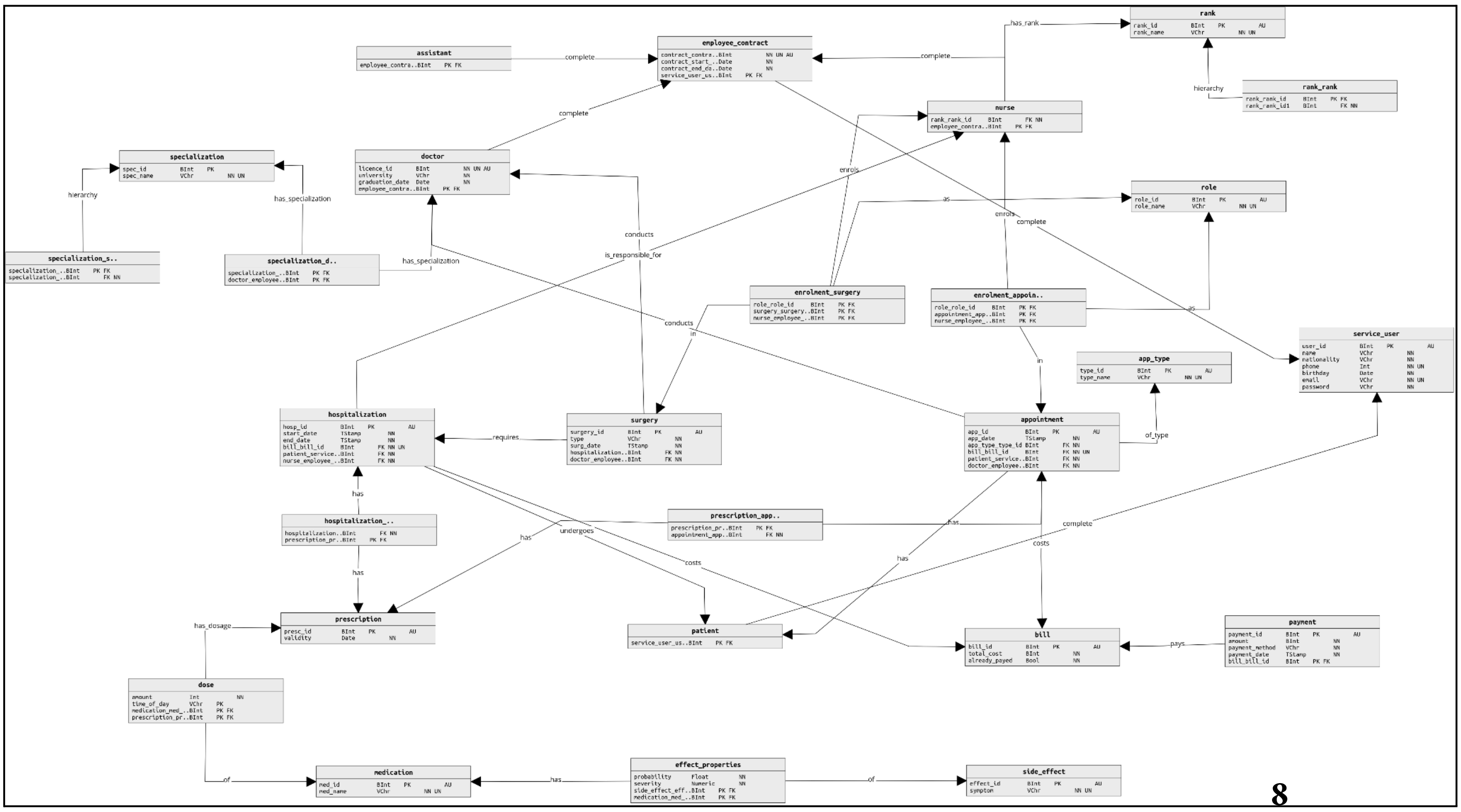
A medication can have several side effects with different levels of severity and probability of occurrence, therefore, to describe these characteristics for a given medication, the entity set *effect_properties* was created, which is weak for *medication* and *side_effect*, this has attributes that describe the severity and probability of occurrence of the *side_effect* entity to which it is associated, when taking the *medication* entity to which it is also associated.

2.3.7 - time_of_day attribute in the dose table

In a prescription, it is common that the same medicine might be taken at different times throughout the day, to make this possible in the relational model, *time_of_day* primary key was added, which defines the time of day at which the dose defined by the *amount* attribute should be taken. This way, we can have the **same medicine more than once in the same prescription** at different times of the day. Furthermore, this will be treated as the *posology_frequency* in the endpoints.

2.4 - Physical Diagram

In the end, the aforementioned E-R model translates to the following relational data model:



3 - Transaction Definitions

A transaction is a logical unit of work that must necessarily be completed. If that is not possible, it is imperative to abort the transaction and rollback the work done so far to the nearest checkpoint.

This section will describe the main, more complex transactions, that is, transactions that involve more than one table, although some of them would be possible to deduce from the endpoints section ahead, we will now be viewing the transactions from a more abstract point of view, not how they are executed.

3.1 - Create a New User

This transaction will affect different tables depending on the type of registered user. In either case, an entry is added to *service_user*.

If it is an employee, an entry will be added to *employee*, as well as to the table corresponding to the specific type of employee that was added, that is: *nurse*, *doctor* or *assistant*. If it's a *doctor*, one or more entries can also be added to the table linking *doctor* to *specialization* to denote the specializations of the added *doctor*. It is also worth noting that if it is nurse, it is necessary to add an entry to the table that links *nurse* to *rank* in order to denote the level in the hierarchy of the newly added *nurse*.

If it is a patient, an entry is added to *patient*.

3.2 - Schedule Surgery

When a surgery is scheduled, an entry is added to *surgery* and the entry in the *bill* table corresponding to the hospitalization associated with the surgery is updated (using triggers). When there is no hospitalization associated with the surgery, new entries are also added to *hospitalization* and *bill*. Additionally, there may also be updates to the *enrolment_surgery* table if the involvement of *nurse* entities is necessary.

3.3 - Make an Appointment

When an appointment is made, an entry is added to the *appointment* and *bill* table. Additionally, there may also be updates to the *enrolment_appointment* table if the involvement of *nurse* entities is necessary.

3.4 - Add Prescriptions

When a prescription is added, in addition to adding an entry to *prescription*, it is also necessary to add entries both to the table that links *prescription* to *hospitalization*, and to the table that links prescription to *medication* (dose), in order to describe the doses of the prescribed medicines.

3.5 - Create a Daily Summary

To create a daily summary, you need to read the details of the *prescription*, *surgery* and *payment* tables for a given day.

3.6 - Generate a Monthly Report

In order to generate a monthly report, we will need to read data from the *surgery* table to know which doctor performed each surgery in a given month.

3.7 - Execute Payment

When a patient makes a payment, an entry is added to the *payment* table, it is also checked whether the sum of payments for a given *bill* is equal to the total cost of that *bill*, in this case, the *already_paid* attribute of the bill is updated and changed to true.

3.8 - List of the Top 3 Patients of the Month

To get a list of the 3 customers who paid the most in the current month, we will need to consult the *payment* table, to check all payments made in the last 30 days, to know which patient made a given payment, you will need to consult the *bill* entry at which the payment is associated with, as well as the *appointment* or *hospitalization* entry to which the *bill* is associated, finally, we check which entry in the *patient* table is linked to the *appointment* or *hospitalization*.

4 - Potential Concurrency Conflicts

Evidently, the fact that multiple users are allowed to access the system at the same time will lead to some concurrent transactions, thus, **not every concurrency conflict will be handled by the DBMS**, those less obvious cases will now be addressed.

4.1 - Schedule Surgery or Appointment

Whenever concurrent transactions try to schedule surgeries at the same time, it's important to make sure that the same employee is not involved in more than one medical service at the same time, to make those verifications, we first need to apply an **update lock** to the *appointment* and *enrolment_appointment* tables (or the *surgery* and *enrolment_surgery* tables) tables and check them for appointments or surgeries already taking place at that time. It's assumed that a surgery takes **2 hours** and an appointment takes **30 minutes**, so, we can neither insert a surgery 2 hours before another service nor an appointment 30 minutes before another service (analogously, we can't insert services 30 minutes after an appointment or 2 hours after a surgery).

The update lock ensures that no doctor or nurse is inserted by another transaction in the time frame between the availability check and the actual insertion by the "main" transaction.

Needless to say, when implementing this approach, to avoid **deadlocks**, the availability verifications are **always** made in the same order (doctors -> nurses).

4.2 - Top 3 Patients, Monthly Report and Daily Summary

An important aspect about the creation of the top 3 patients list are **dirty reads** if we neglect the transaction isolation. For example, the following scenário might take place:

Current Order:	Empty List	A	A	A	B > A
Current patient A's amount:	10€	10€	30€	30€	30€
Current patient B's amount:	5€	5€	5€	15€	15€
Program State:	Start Query	Read A	A pays 20€	B pays 10 €	Read B
Current Moment:	T0	T1	T2	T3	T4

In this case, even though patient B has never had a superior amount to A's, in the final list, B's amount ended up in the first place, which is clearly wrong.

To avoid these dirty reads, the transaction is isolated on a **serializable level**. The same principle is applied in the monthly report and the daily summary

4.3 - Execute Payments

To ensure that an already paid bill cannot be paid again, when a payment is made, a **row-level lock** is applied to the *bill* table entry associated with the payment being made.

5 - Endpoints

The endpoints related to registering users into the database, all using the POST method, take the following payload format:

5.1 - Register Patient - http://localhost:8080/dbproj/register/patient

```
{
    "email": email, // Instead of username
    "password": password,

    // Extra arguments, not defined in the project statement
    "name": name,
    "birthday": birthday,
    "phone": phone_number,
    "nationality": nationality
}
```

5.2 - Register Assistant - http://localhost:8080/dbproj/register/assistant

```
{
     "email": email,
     "password": password,

     // Extra arguments, not defined in the project statement
     "name": name,
     "birthday": birthday,
     "phone": phone_number,
     "nationality": nationality,
     "contract_start_date": contract_start_date,
     "contract_end_date": contract_end_date,
}
```

5.3 - Register Nurse - http://localhost:8080/dbproj/register/nurse

```
{
     "email": email,
     "password": password,

     // Extra arguments, not defined in the project statement
     "name": name,
     "birthday": birthday,
     "phone": phone_number,
     "nationality": nationality,
     "contract_start_date": contract_start_date,
     "contract_end_date": contract_end_date,
     "rank_id": rank_id
}
```

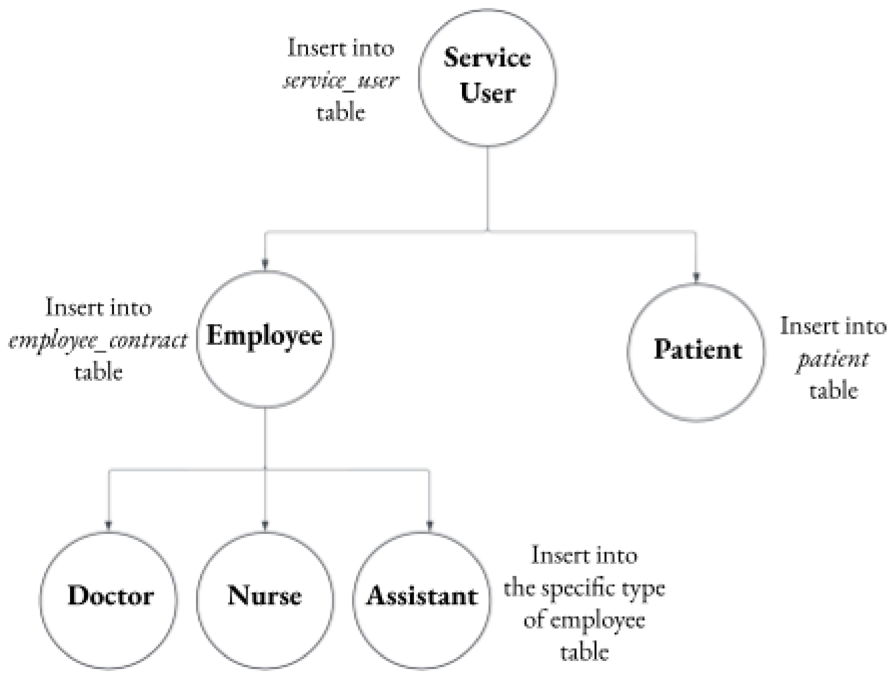
5.4 - Register Doctor - http://localhost:8080/dbproj/register/doctor

```
{
     "email": email,
     "password": password,

     // Extra arguments, not defined in the project statement
     "name": name,
     "birthday": birthday,
     "phone": phone_number,
     "nationality": nationality,
     "contract_start_date": contract_start_date,
     "contract_end_date": contract_end_date,
     "university": university,
     "graduation_date": graduation_date,
     // Possible specializations are stored hierarchically in the
     // specialization table
     "specializations":
         [
             specialization1_name,
             specialization2_name,
             (...)
         ]
  }
```

For all of the registration endpoints, an *email* is asked instead of the *username*, as the *email* has an unique constraint in the database, as stated previously.

The user creation transaction, for the different types of user, was made according to the following flow tree, each node being a different path that the *register_service_user* function can take, as a way to make the code more modular:



Insert into *service_user* table
Service User
Insert into *employee_contract* table
Employee
Patient
Insert into *patient* table
Doctor
Nurse
Assistant
Insert into the specific type of employee table

5.5 - Authenticate User - http://localhost:8080/dbproj/user

The *authenticate_user* endpoint uses the PUT method and takes the following arguments:

```
 {
     "email": email, // Instead of username
     "password": password,
 }
```

Similarly to the registration endpoints, there is no *username* as it was fully replaced with *email*. Upon retrieving the email and verifying the password against the stored hash (discussed later), the role tables are queried to retrieve the user's role. The access token is then generated and set as a **cookie**, alternatively it can be set manually as a request header in Postman, the token contains information about the user's id and their type.

5.6 - Schedule Appointment - http://localhost:8080/dbproj/appointment

The *schedule appointment* endpoint uses the POST method and takes the following arguments:

```
{
    "doctor_id" : doctor_user_id,
    "date" : date,
    // Extra arguments, not defined in the project text
    "type" : type,
    "nurses" :[
        {"nurse_id": nurse_user_id1, "role": role_name},
        {"nurse_id": nurse_user_id2, "role": role_name},
        (...)
    ]
}
```

Upon retrieving the information, the doctor and nurses availability is verified and, if everyone is available, the appointment is inserted into the *appointment* table and the nurses (if applicable) are associated via the *enrolement_appointments* table and a **trigger** inserts a new bill into the *bill* table for that appointment.

5.7 - See Appointments - http://localhost:8080/dbproj/appointments/{patient_user_id}

The *see appointments* endpoint uses the GET method and takes no arguments. Since a patient can only view their own appointments, if a patient accesses this endpoint, their token's identity is verified against the *user_id* before fetching the information from the *appointment* table.

5.8 - Schedule Surgery - http://localhost:8080/dbproj/surgery/{hospitalization_id}

The *schedule surgery* endpoint uses the POST method and takes the following arguments:

```
{
    "patient_id" : patient_user_id,
    "doctor": doctor_user_id,
    "date": date,
    "type": type,
    "nurses": [
          {"nurse_id": nurse_user_id1, "role": role},
          {"nurse_id": nurse_user_id1,  "role": role},
    ]
}
```

As there's no way to have a surgery without a hospitalization, when a hospitalization_id is not provided, a new hospitalization is created, taking into account the fact that an hospitalization needs a responsible nurse, the first nurse provided in the payload is considered responsible for the hospitalization, which makes it mandatory to provide at least one nurse **when scheduling a surgery without associating it with an hospitalization**, if that's not the case, the inclusion of nurses in the payload is facultative.

The new hospitalization (if applicable) is then inserted into the *hospitalization* table, the surgery into the surgery table and the nurses (if applicable) are associated to the surgery via de *enrolment_surgery* table.

Furthermore, a **trigger** also updates or creates the hospitalization bill accordingly.

5.9 - Prescribe Medication - http://localhost:8080/dbproj/prescription

The *prescribe medication* endpoint uses the POST method and takes the following arguments:

```
{
    "type": "hospitalization/appointment",
    "event_id": id,
    "validity": date,
    "medicines":[
        {
            "medicine":medicine1_name,
            "posology_dose":value,
            "posology_frequency": value
        },
      (...)
    ]
}
```

The prescription is added into the *prescription_table*. Then, in the table that links prescription to medication, inserts the *prescription_id* and *medication_id* as well as the *posology_dose* and *posology_frequency* (*time_of_day*). This happens for every medicine provided in the payload.

5.10 - Get Prescription - http://localhost:8080/dbproj/prescriptions/{person_id}

The *get prescription* endpoint uses a GET method and takes no arguments. As the patient can only get their own prescriptions, the token's identity is verified if the user is a patient, just like in the *see appointments* endpoint.

Since any given prescription can be associated to an appointment or a hospitalization, the SQL query used to get the prescriptions from a given user, must join results from the prescriptions associated to that user's hospitalizations and appointments. Moreover, the prescriptions must be grouped by id and validity (a prescription might have multiple medications and posology information, but only one id and one validity date).

5.11 - Execute Payment - http://localhost:8080/dbproj/bills/{bill_id}

The *execute payment* endpoint uses the POST method and takes the following arguments:

```
{
    "amount": value,
    "payment_method": value
}
```

A patient can only pay their own bill, therefore it is verified if the *bill_id* is associated with the user. Then the transaction starts and the payment is processed. A **trigger** then takes place to update the bill, according to the amount paid.

5.12 - Get Top 3 - http://localhost:8080/dbproj/top3

The *get top 3* endpoint uses the GET method and takes no arguments, this endpoint is implemented in a single SQL query, it returns a list of the 3 patients who spent the most money in the last month, as long as the procedures where they spent that money.

To achieve that, the query selects the sum of the *already_paid* (grouped by *patient_id*) attribute from the payment table rows whose *payment_date* is in the last 30 days. To know which patient executed each payment, we need to check the associated *bill* entry, which itself is associated with the patient's appointment or hospitalization, these services and their details will also be fetched. After ordering by the sum of *already_played* in decreasing order, the results are limited to 3 results.

5.13 - Daily Summary - http://localhost:8080/dbproj/daily/{year-month-day}

The *daily summary* endpoint uses the GET method and takes no arguments, this endpoint is implemented in a single SQL query that returns the amount spent, the surgeries made and the prescriptions whose validity starts on a given date.

In order to get those results, the query counts the rows obtained when getting that day's surgeries and the prescriptions whose validity starts on that date along with the sum of the payments made on that day. The results are coalesced in order to display 0 when no results are found.

5.14 - Monthly Report - http://localhost:8080/dbproj/report

The *monthly report* endpoint uses the GET method and takes no arguments, this endpoint is implemented in a single SQL query that returns a list with every month, alongside the doctor who performed more surgeries in each month.

To carry out this goal, the query generates a list of the last 11 months alongside the current one, to ensure we get a row for each month, even if there are no surgeries performed. It then ranks the doctors over each month's partition, ordering them by the amount of surgeries. Only the doctors in the first place of the rank are displayed and when there are no surgeries in a given month, the result is coalesced to “NO SURGERIES”.

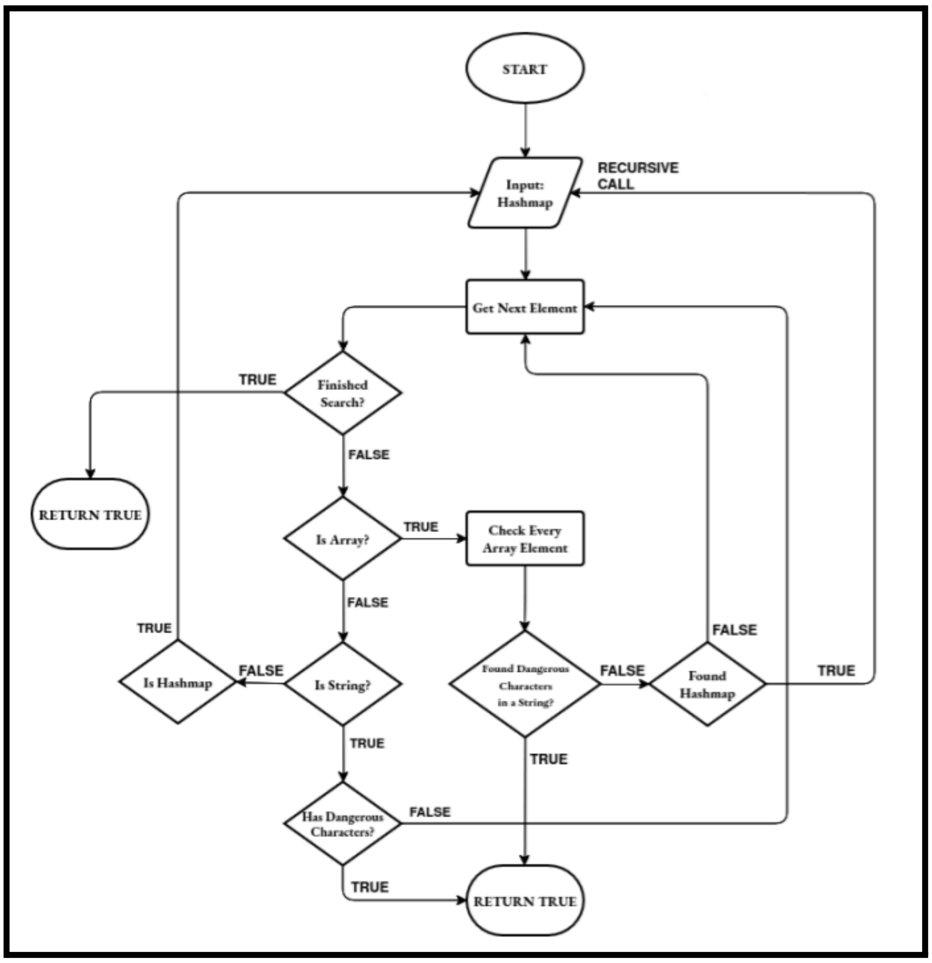
6 - Security Measures

6.1 - Password Hashing

Before being stored, every password goes through 100,000 iterations of the **SHA-256** algorithm. We chose to do a significant amount of iterations to increase the computational cost of brute-force attacks. Moreover, a salt is being applied to make sure the algorithm's output stays nondeterministic, to prevent rainbow table attacks. This process is applied to an inserted password during the login sequence, thus guaranteeing the user’s authentication upon inserting a valid email-password combination.

6.2 - SQL Injection Avoidance

Every time a payload is received from a user, we need to make sure to minimize the risk of **SQL Injections** as much as possible, we need to check the payloads for any dangerous characters. This is made with a recursive function, which handles the case where the attacker inputs an hashmap or an array inside of the payload by recursively calling itself. This behavior can be seen in this flowchart and in the function’s code.



```
# Search dangerous characters in a user input string
def string_contains_dangerous_chars(input_str):
    # Check for SQL injection characters
    dangerous_chars = [';', '--', '/*', '*/']
    for char in dangerous_chars:
        if char in input_str:
            return True
    return False

# Check if a payload contains dangerous characters
def payload_contains_dangerous_chars(payload):
    for key, value in payload.items():
        # If it's a dictionary, recursively call the function
        if isinstance(value, dict):
            if payload_contains_dangerous_chars(value):
                return True
        # If it's a list or tuple, iterate over the elements and check each one
        elif isinstance(value, (list, tuple)):
            for element in value:
                if isinstance(element, str) and string_contains_dangerous_chars(element):
                    return True
                elif isinstance(element, (dict, list, tuple)) and
↪payload_contains_dangerous_chars({0: element}):
                    return True
        # If it's a string, check for dangerous characters
        elif isinstance(value, str):
            if string_contains_dangerous_chars(value):
                return True
    return False
```

As another layer of security against injections, the queries are **highly** parameterized, never using the user input directly in the query's string.

7 - Database Tuning

7.1 - Indexing

To make queries more efficient, indexes were added to some fields, however, as indexes can make insertions, updates and deletions slower, we only added indexes to fields commonly used by WHERE clauses. Those being the user emails, names, people ID's, bill ID's and dates.

7.2 - Denormalization of the *bill* Table

The *already_paid* field, although redundant, makes verifying if a bill has already been paid significantly faster as there is no need to check every payment associated with a given bill every time a user wants to make a payment, so, we considered this tradeoff between redundancy and efficiency a positive trade.

8 - Non-Explicit Design Choices

8.1 - Hospitalization's Responsible Nurse

Whenever a new surgery is scheduled, in case no hospitalization is associated with it, one must be created and a responsible nurse has to be defined. Therefore, the 'nurses' parameter is mandatory in the *shedule_surgery* endpoint when hospitalization_id is None.

```
extra_required_key = 'patient_id'
    if hospitalization_id is None:
        extra_required_key = 'nurses'

    missing_keys = check_required_fields(payload, ['patient_id', ↪'doctor',
↪extra_required_key, 'date', 'type'])
    if (len(missing_keys) > 0):
        response = {
            'status': StatusCodes['bad_request'],
            'errors': f'Missing required field(s): {", ".join(missing_keys)}'
        }
        return flask.jsonify(response)
```

8.2 - Hospitalization's End and Start Dates

As the start and end are not specified in the payloads, the hospitalization's end and start date adapt to the first and last surgeries performed in said hospitalization, i.e, the *start_date* is the date of the earliest surgery and the *end_date* is the date of the latest surgery + the duration of the surgery.

8.3 - Responsible Nurses

Even though nurses cannot be in multiple surgeries or appointments at the same time, they are allowed to be responsible for multiple hospitalizations at the same time, as that responsibility is not considered a task that a nurse must be actively doing

8.4 - Authorization Token Cookies

To automate the process of including the authorization token in the request headers, we store it in the cookies and check them every time a request that needs authorization is made.

9 - Program Files Structure

9.1 - Python Files

- **rest_api.py**
  - Contains every endpoint route and initiates the application
- **global_functions.py**
  - Contains functions used throughout the program, those being: database connection, required fields verifications, SQL script execution, dangerous character verification and database setup.
- **register_user.py**
  - Responsible for inserting each type of user in their respective tables.
- **authentication.py**
  - Has the function used for authentication of a user and providing the authorization token, this file also has a decorator to verify if the user has the correct permissions for a given endpoint.
- **scheduling.py**
  - This file contains the functions responsible for inserting new entries into the hospitalization, surgery and appointment tables.
- **get_appointments.py**
  - Holds the function for consulting a certain user's appointments.
- **prescriptions.py**
  - Responsible for both inserting new prescriptions as well as getting a user's prescription information.
- **payment.py**
  - Executes a bill's payment.
- **top3.py**
  - Runs the single top 3 query.
- **daily_summary.py**
  - Runs the single daily summary query.
- **monthly_report.py**
  - Runs the single monthly report query.

9.2 - SQL Files

- **tables_creation.sql**
  - Creates the tables, the primary key constraints and the indexes.
- **tables_constraints.sql**
  - Adds constraints to the table's attributes

- **drop_tables.sql**
  - Drops every table.
- **populate_tables.sql**
  - Populates tables with information such as nurse ranks, medical specializations, appointment types, medicine names, and side effects.
- **disconnect_users.sql**
  - Removes the connection privileges from every user currently connected.
- **check_setup.sql**
  - Verifies every table is set up properly before starting the program.
- **add_payment.sql**
  - Adds an entry to the *payment* table .
- **create_bill_triggers.sql**
  - Defines and sets triggers for surgery, hospitalization, appointment and payment insertions.
- **daily_summary.sql**
  - Single daily summary query,
- **get_prescriptions.sql**
  - Gets a user's prescriptions.
- **monthly_report.sql**
  - Single monthly report query.
- **top3.sql**
  - Single top 3 query.

10 - Installation Manual

10.1 - Prerequisites

- **Python 3.X**
  - Go to www.python.org and follow the installation instructions for your operating system.

- **Docker or Docker Desktop**
  - Go to www.docker.com and follow the installation instructions for your

operating system.

10.2 - Database Setup

- **Navigate to the project's folder**

```
$ cd hms_db
```

- **Build the Docker image (Run with root permissions if needed)**

```
$ docker build -t hms_postgres .
```

- **Run the Docker container (Run with root permissions if needed)**

```
$ docker run -p 5432:5432 hms_postgres
```

10.3 - Python Environment Setup

- **Navigate to the program's folder on another terminal window**

```
$ cd hms_db
```

- **Create a virtual environment**

```
$ python -m venv env
```

- **Activate the virtual environment**
    - **Linux/MacOS**

```
$ source env/bin/activate
```

    - **Windows**

```
$ .\env\Scripts\activate
```

- **Install the dependencies**

```
$ pip install -r requirements.txt
```

11 - User Manual

11.1 - Running the Program for the First Time

In order to set up the tables for the first time, you have to run the program with a specific flag, this will not be needed in posterior executions, unless you want to reset the table contents.

- **Navigate to the program's source folder**

```
$ cd src
```

- **Execute rest_api.py using the --setup flag if it's your first time running**

```
$ python rest_api.py --setup
```

11.2 - Postman Collection

In the project root folder there is also a Postman collection JSON file, (*hms_collection.postman_collection.json*), you can import this collection into your Postman client to get every request needed to test the endpoints.

- **Registering Users:**

  The specific syntax of each individual type of user's registration is as follows, keep in mind that the **email** attribute must be in the form `<string>@<string>.<string>`, the **phone number** must be 9 digits long and the **birthday** must be in the form `YYYY-MM-DD`, additionally, the birthday can't be a date before 1904 (as there is no one alive older than 120).

  - **[POST] Register Patient:**

http://localhost:8080/dbproj/register/patient

```
{
    "email": email,
    "password": password,
    "name": name,
    "birthday": birthday,
    "phone": phone_number,
    "nationality": nationality
}
```

- **[POST] Register Assistant:**
  http://localhost:8080/dbproj/register/assistant

```
{
    "email": email,
    "password": password,
    "name": name,
    "birthday": birthday,
    "phone": phone_number,
    "nationality": nationality,
    "contract_start_date": contract_start_date,
    "contract_end_date": contract_end_date,
    "rank_id": rank_id
}
```

- **[POST] Register Nurse:**
  http://localhost:8080/dbproj/register/nurse
  The possible *rank_id* values range from 1 to 3, corresponding to Junior Nurse, Senior Nurse and Head Nurse respectively.

```
{
    "email": email,
    "password": password,
    // Extra arguments, not defined in the project text
    "name": name,
    "birthday": birthday,
    "phone": phone_number,
    "nationality": nationality,
    "contract_start_date": contract_start_date,
    "contract_end_date": contract_end_date,
    "rank_id": rank_id
}
```

- **[POST] Register Doctor:**
  http://localhost:8080/dbproj/register/doctor
  The possible specializations/sub-specializations are:
  - ➔ Neurology
    - ◆ Pediatric Neurology
    - ◆ Neurophysiology
    - ◆ Neuropathology
  - ➔ Radiology
    - ◆ Thoracic Radiology
    - ◆ Interventional Radiology

        - Head and Neck Radiology
    - ➔ Cardiology
        - Pediatric Cardiology
        - Interventional Cardiology
        - Cardiac Electrophysiology

```
{
    "email": email,
    "password": password,

    // Extra arguments, not defined in the project text
    "name": name,
    "birthday": birthday,
    "phone": phone_number,
    "nationality": nationality,
    "contract_start_date": contract_start_date,
    "contract_end_date": contract_end_date,
    "university": university,
    "graduation_date": graduation_date,
    // Possible specializations are stored hierarchically in the
    // specialization table
    "specializations":
        [
            specialization1_name,
            specialization2_name,
            (...)
        ]
 }
```

To make the process of populating the tables with users faster, the registration requests have a Pre-Request javascript scripts included to automatically generate credentials, in order to use them, you have to write each request attribute with the following syntax: {`"attribute": "{{attribute}}"`}. For example, to generate a random nurse, one would input:

```
{
   "name": "{{name}}",
   "nationality": "{{nationality}}",
   "phone": "{{phone}}",
   "birthday": "{{birthday}}",
   "email": "{{email}}",
   "password": "{{password}}",
   "contract_start_date": "{{contract_start_date}}",
   "contract_end_date": "{{contract_end_date}}",
   "rank_id": "{{rank_id}}"
}
```

- **[PUT] User Authentication**
  http://localhost:8080/dbproj/user
  To log into the system as some user, you can input the user's email and password as follows:

```
{
    "email": email,
    "password": password,
}
```

After the credentials are validated, an authentication token will be added to your cookies and the access to the rest of the endpoints will be unlocked for 30 minutes (when the session expires).

- **[POST] Schedule Appointment**
  http://localhost:8080/dbproj/appointment

  To schedule an appointment, you must be logged in as a **patient** and input the appointment's details as follows, keep in mind that **date** must be in the form `YYYY-MM-DDTHH-MM-SSZ` and the possible **types** of appointments are:
  - ➔ Primary Care
  - ➔ Psychotherapy
  - ➔ Specialist Consultation
  - ➔ Physiotherapy

And the possible **roles** of a nurse are:
  - ➔ Instrument
  - ➔ Preoperative
  - ➔ Patient Advocacy
  - ➔ Euthanasist

```
{
    "doctor_id" : doctor_user_id,
    "date" : date,
    // Extra arguments, not defined in the project text
    "type" : type,
    "nurses" :[
        {"nurse_id": nurse_user_id1, "role": role_name},
        {"nurse_id": nurse_user_id2, "role": role_name},
        (...)
    ]
}
```

- **[GET] See Appointments**
  http://localhost:8080/dbproj/appointments/{patient_user_id}

  To check a patient's appointments you have to be logged in as an **assistant** or the **target patient**, to select a patient, it's user id must be included in the request's URL, for example: http://localhost:8080/dbproj/appointments/1 to check the appointments for the patient with id 1.

- **[POST] Schedule Surgery**
  http://localhost:8080/dbproj/surgery/{hospitalization_id}

  To schedule a surgery and associate it to a hospitalization, you need to be logged in as an **assistant** and input the surgery's details as follows, keep in mind that this request's behavior will depend on whether an hospitalization id was included in the request's URL, as a new hospitalization will be created if no id is provided, also, the **date** must be in the format YYYY-MM-DDTHH-MM-SSZ and the available nurse **roles** are the same as mentioned in the **Schedule Appointment** section.

```
{
    "patient_id" : patient_user_id,
    "doctor": doctor_user_id,
    "date": date,
    "type": type,
    "nurses": [
           {"nurse_id": nurse_user_id1, "role": role},
           {"nurse_id": nurse_user_id1,  "role": role},
    ]
}
```

- **[POST] Prescribe Medication**
  http://localhost:8080/dbproj/prescription

  To prescribe medication and associate it to an appointment or hospitalization, you must be logged in as a **doctor** and input the prescription details as follows, keep in mind that the possible **posology_frequency** values are "Morning", "Afternoon" and "Evening" (not case sensitive), the **date** must be in the format YYYY-MM-DD and the possible **medicines** are:

  ➔ Paracetamol
  ➔ Ibuprofen
  ➔ Aspirin
  ➔ Morphine
  ➔ Codeine
  ➔ Quetiapine

```
{
    "type": "hospitalization/appointment",
    "event_id": id,
    "validity": date,
    "medicines":[
        {
            "medicine":medicine1_name,
            "posology_dose":value,
            "posology_frequency": value
        },
      (...)
    ]
}
```

- **[GET] Get Prescriptions**
  http://localhost:8080/dbproj/prescriptions/{person_id}

  To check a patient's prescriptions, you must be logged in as the **target patient** or an **employee**, to select a patient, it's user id must be included in the request's URL, for example: http://localhost:8080/dbproj/prescriptions/1 to check the prescriptions for the patient with id 1.

- **[POST] Execute Payment**
  http://localhost:8080/dbproj/bills/{bill_id}
  To execute a payment, you must be logged in as the patient whose bill belongs to and include said bill's id in the URL, for example: http://localhost:8080/dbproj/bills/1 to pay the bill with id 1, and input the details as follows:

```
{
    "amount": value,
    "payment_method": value
}
```

- **[GET] Get Top 3**
  http://localhost:8080/dbproj/top3
  To get a list of the patients who paid the most in the last 30 days, you must be logged in as an **assistant**.

- **[GET] Daily Summary**
  http://localhost:8080/dbproj/daily/{year-month-day}
  To get a list of the money made, the amount of surgeries performed and prescriptions whose validity starts on a given date, you must be logged in as an **assistant** and include that date in the URL, for example: http://localhost:8080/dbproj/daily/2024-05-23 to check the summary from 2024-05-23.

- **[GET] Monthly Report**
  http://localhost:8080/dbproj/report
  To get a list of the doctors who performed more surgeries in each month of the last year, you must be logged in as an **assistant**.

12 - Development Plan

Nuno - 🟦 Miguel - 🟥

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9
User Registration		🟥							
User Authentication		🟦							
Service Scheduling				🟦					
Payment Execution				🟥					
Add Prescriptions					🟦				
Check Prescriptions/Appointments					🟥				
Security						🟦			
Tunning						🟥			
Top 3 Patients						🟦			
Monthly Report						🟦			
Daily Summary						🟥			
User Manual						🟥			
Installation Manual						🟦			
Presentation Slides							🟪		
Final Report								🟪	🟪

Estimated effort involved per team member: 20h