# BD Intermediate Report

João Esteves, Pedro Lima, Sofia Leitão

Bachelor's in Informatics Engineering
Faculty of Sciences and Technology of the University of Coimbra

May 28, 2024

## Contents

# 1 Introduction

This report documents the work done by the group for the final review stage of the Databases course project.

The project name chosen is LifeLink, joining the concept of Healthcare and Databases through the chosen words: "*link*" - as in the relations between entities in a database, and *"life"* - self explanatory.

**Members and contacts:**

- João Esteves (2022230550@student.dei.uc.pt)
- Pedro Lima (pmlima@student.dei.uc.pt)
- Sofia Leitão (2022214134@student.dei.uc.pt)

# 2 Project Description

LifeLink is a simplified application for managing and storing processes for a Healthcare institution. It aims to provide different users a set of features specific to their role (patient, doctor, nurse or assistant) through an HTTP web server, allowing them to review and change information within the institution's database through simple HTTP requests.

The application layer in between the user and the database not only streamlines the functions from an end user perspective, but also allows a good developer the opportunity to protect the database, maintaining data integrity while maximizing the efficiency of its storing and accessing.

# 3 DB Architecture

Worthwhile points to mention for the DB architecture are how the hierarchies were implemented, creating an auxiliary table where we can insert into the foreign key the id above in hierarchy and in the primary key the id of the entity below. This works for both specializations and nurses, the latter of which wasn't utilized in the actual operations of the DB.

It's worth mentioning as well that bills and payments are related to patients by joining them on the event they're related to. Additionally, and upon final implementation of the daily summary endpoint, the group understood the potential need to have a created_at field for prescriptions, payments and bills, to more accurately obtain their dates for hospitalizations that can last more than a single day. This was not implemented due to time constraints.

For the structure of the person/employee/patient, the DB assumes employees can be patients, but employee types are unique, even though the endpoint specs for registering didn't allow adequate testing of this feature. This means in all queries that require a free doctor/nurse we accurately check if they are busy being a patient in any one event, and likewise when checking if the patient is free, we confirm they're not busy being a doctor/nurse. One potential flaw in this is that if a doctor/nurse that was busy requires

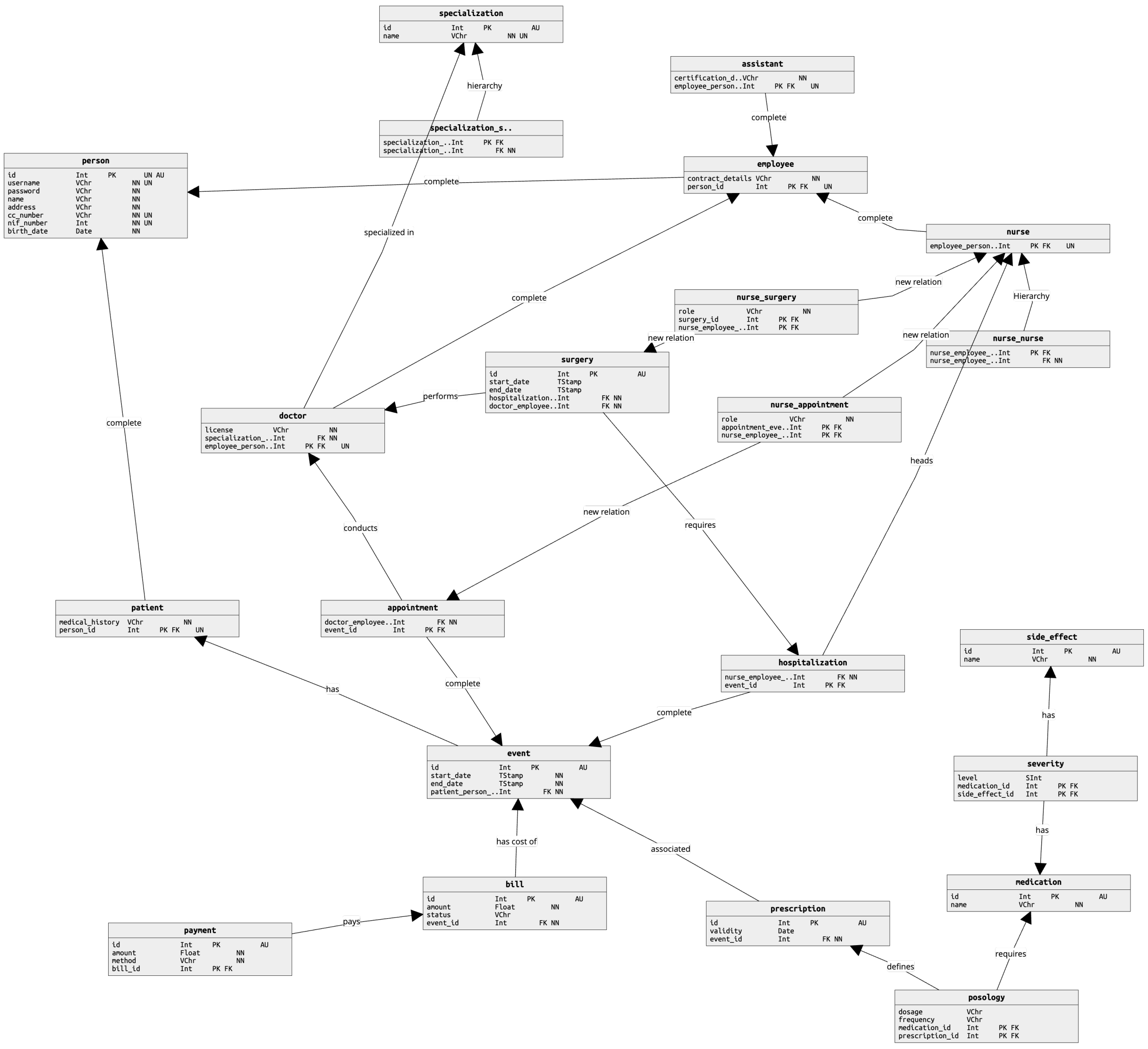

Figure 1: Model of the database architecture

an emergency surgery, the system will have trouble dealing with the situation. This could be resolved by adding a "forceful" flag to the create surgery endpoint and db function, that does not check if the patient is free. This forceful flag could be useful in other similar scenarios, but wasn't implemented.

# 4 Code structure

The code was divided in 5 files: "api.py", "validator.py", "endpoints.py", "db.py", "utils.py", all contained inside a /src directory, and finally "run.py" in the main directory.

The api code contains the initialization of the api flask object, as well as 4 wrapper functions (decorators), 2 of them leveraging flask specific features that enable a db_connection to be pooled from a pool of connections created and imported in "db.py", one "token_required" for endpoints that need to validate the token, and an error handling decorator, further discussed in the next section.

The utils code serves as a collection of useful initialization functions such as the logger initialization and the config initialization, that takes sensitive variables like the db credentials away from the source code and into a .env file, as well as some useful dictionaries/lists.

The validator code serves as the initial validation step of the endpoint, that makes sure all the fields are correct and raises an error if they aren't. The validation is surface level, and doesn't completely prevent ValueErrors down the line, but it serves as a good cornerstone.

The endpoints code serves as the actual implementation of each endpoint, configuring routes and making use of the decorators in api and validator functions. It then calls usually a single function from the db module, where all the interactions with the db are programmed.

The db code contains the actual interaction with the db. Worth noting that it implements a decorator "transactional" further discussed in the next section.

The run code imports the flask app object, as well as the config and logger objects, and initializes the application.

One final caveat on the structure, the requirements.txt enables the command "pip install -r requirements.txt" to easily setup a python environment capable of running the application.

# 5 Further nuances

## 5.1 Token handling and login flow

The token handling is done through the use of the pyjwt library. During the login, the user authenticates with username and password. The db checks this user exists and returns all the login_types (patient, doctor, nurse, assistant) as a list, as well as its id. This information is added to the token, which is then encoded.

The decoding of the token is done by the decorator, that adds the types and id of the login to the arguments of the function being decorated.

## 5.2 Connection pooling

Connection pooling was implemented in the project for optimization reasons. It creates a pool of db connections at the start of the program, which are then fetched and returned to the pool by the decorators defined in the api module we programmed, at the start and end of a request loop.

## 5.3 Transactions and concurrency handling

To handle transactions and concurrency in a way that's elegant and enables the programmer to more easily focus on the logic of the db, the decorator transactional opens a cursor and sets the auto-commit to false, initiating a transaction. This prevents other connections from accessing the same rows before the full transaction is commited to the db. This decorator is used only for the functions being called by the endpoints. If other sub functions are called by the parent db function, they do not require the transactional decorator. It's important to never commit any of our cursor executions as the decorator takes care of that at the end of the parent db function's execution, as well as the cursor closing. This also means the cursor needs to be passed down from function to function, and that the cursor is passed by the decorator to the main db function.

Here's the specific concurrency conflits identified:

#### Assigning employees to overlapping tasks simultaneously

While checking in appointments and surgeries if a doctor/nurse is already assigned at that hour, two transactions could simultaneously find that entity to be "free", creating the issue of "same person, in different places, at the same time".

This can be avoided by locking the task table we are modifying, ensuring that other transactions don't simultaneously modify the same record. While our transactional decorator already helps prevent these issues, in register functions we adequately lock the tables needed to perform the registration. Further locking could lead to deadlocks, so it is avoided.

#### Accessing bills, prescriptions and hospitalization information that has been altered but not yet committed, resulting in dirty Reads

By choosing an appropriate isolation level, we can achieve the required data consistency and integrity we need to avoid Dirty Reads. In PostgreSQL, the default isolation level is READ COMMITTED, so we did not implement further isolation levels. The one case where this could lead to problems are phantom reads, that don't seem likely to occur.

## 5.4 Error handling

Error handling is done by the decorator defined in the api module. This decorator abstracts a try: catch block on all endpoint functions that use it, to enable the actual function code from being more readable, and to protect the whole execution from errors. This last decorator handles ValueError as api errors, and other errors as internal errors.

During the code of functions where we want to throw an api error, we simply raise a ValueError. This could be done in more detail creating multiple different types of error, but for the purposes of this project this sufficed.

## 5.5 Indexing

Not a lot of indexes were created. In the tables person, patient, event, hospitalization, surgery, appointment, nurse appointment, nurse surgery, prescription, payment, and posology it was not done since these tables are constantly being inserted into, meaning the db would need to be constantly reindexing these fields, which is less than ideal, even if those fields would otherwise make sense to index.

In the tables severity, side effect and specialization, it did not make sense to create indexes since the entries in these tables are not expected to be large enough to gain anything from those indexes.

In the medication table the id and name were indexed, since in theory, this table would have hundreds of entries that are not changed regularly, and have enough variation that it would have some efficiency gain to the db.

In the tables nurse, doctor, assistant the foreign key for the id was indexed, since the number of employees isn't expected to constantly change, and these fields are constantly being queried and joined into for a small quantity of ids (likely less than 10% of the total number of ids).

There is an argument to be made to index in appointment, hospitalization, surgery and event both the ids, and foreign key ids, as well as start/end_date. However, the large amounts of insertions makes them likely not feasible for indexing, even if the performance gain would be substantial. In truth, this should be tested during the implementation of the program in the particular hospital, monitored and checked if it's worthwhile.

## 5.6 Query optimization

The program is careful to parameterize all queries and not call functions or operators on the actual fields being compared, but to make those operations on the parameters. This means, for example, when finding if a timestamp falls within a certain day, we compare that timestamp to a timestamp starting at 00:00 of that day and a timestamp starting at 00:00 of the next day, rather than operating on the timestamp with the DATE() function. The program also tries to minimize joins where data is replicated in the query, although in some scenarios, query readability and maintainability is opted for in place of pure query optimization.

# 6 DB functions

## 6.1 Register User

Every type of user can register themselves, therefore, to insert a new user to the database we create the payload, to store all the information needed. First we check to see if the inserted user already exists, if it doesn't, it inserts the user and adds it's specific table. Then we build the query, execute it and handle any error (in case they occur), after we retrieve the user ID and return it. The password is passed as an argument and its transformed by a hash process and stored in the database in that way.

## 6.2 Login User

To login any type of user we prepare the query, then we hash the password and execute the query. Afterwards we check if the login is successful or not. If it is successful the user ID is retrieved to prepare the role query. Then we execute the query, check for any assigned roles and return the user ID and roles.

## 6.3 Schedule Appointment

This function performs several critical checks and operations to ensure that an appointment can be scheduled without any conflict and with all its details valid. Firstly, it converts and validates the appointment date, then we ensure that the appointment won't be scheduled to the past and that the patient won't schedule an appointment with themselves as doctors or nurses. After all these validations we check the availability of every employee involved, and only now an appointment is created and the transaction committed.

## 6.4 Schedule Surgery

To schedule a surgery the following operations are made, firstly, we ensure the time of the surgery isn't in the past and that the user isn't scheduling an operation for themselves. Next, depending on the user, we either create a hospitalisation or update the current one, checking for availability of everyone involved (patient, nurses and doctor). Only after these validations is the surgery created and uploaded into the database and commits the changes, and ensures the changes were made correctly.

## 6.5 Get Prescriptions

This function works by building and executing the query to fetch the prescription and posology details of the given patient, afterwards, it converts the query results into a structured formatted list of dictionaries, by doing that it processes the results. To finish, it handles the edge cases, in this case, raises an error if the patient doesn't have any prescription.

## 6.6 Add Prescriptions

This method starts by extracting and validating the input by checking the validity of the event ID and type to ensure the prescription is associated with a valid event. Afterwards, it inserts the new prescription into it's table and retrieves it's ID, it is now that the posology records are associated. To finish, the transaction is committed and new prescription ID is returned.

## 6.7 Execute Payment

To execute a payment we first convert the amount of the payload into a float, only now can we validate the payment. Now we verify that the extracted bill belongs to the logged user, afterwards the payment is made and the bill is updated. To finish, to provide feedback, a message is returned to indicate if a bill is totally paid or the amount left to pay.

## 6.8 Top 3

To generate the top 3 patients of the month we start by determining what are the starts and end dates of the current month. Now, by using a query, we retrieve the top 3 patients based on the total spending, including distinct procedures. To finish, we return, in a structured format, all the information regarding the top 3 users and raise an error if no users are found.

## 6.9 Daily Reports

To generate the daily reports we convert the input date to ensure that the date is in the correct format to calculate the range for the specified day. After this, the query is constructed and all the data from the database that follows the specified criteria is accounted for. To finish this transaction, the result is structured into a dictionary and presented to the user, an error is raised if no users are registered in the database.

One important caveat in this implementation, as discussed earlier, is the fact that it pertains to hospitalizations, and therefore payments and prescriptions, where the hospitalization is active in the current day. Meaning there's some carry over from day to day. This could easily be solved by doing something like the following code to the prescriptions/payments tables, enabling more precise querying:

```
ALTER TABLE prescriptions
ADD COLUMN created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP;
```

This solution was not implemented due to time constraints.

## 6.10 Monthly Reports

To generate the daily reports we convert the input date to ensure that the date is in the correct format to calculate the range for the specified year. After this, the query is constructed and all the surgery counts for each month and the top-doctor is selected. To finish this transaction, the result is structured into a dictionary and presented to the user, an error is raised if no users are registered in the database.

## 7 Development Plan

The group planned the development of the project more strictly for the available weeks, leaving later tasks more roughly outlined, already assuming the vision for these tasks and the group members' preferences would consolidate upon completion of the first round of tasks. Regardless, the following table aims to demonstrate the planning done.

### Timeline

| # | Week | Task Description | Group member |
|---|---|---|---|
| 1 | 1 | Start the database setup in PGAdmin | João Esteves |
| 2 | 1 | Explore Postman's professor example | Sofia Leitão |
| 3 | 2 | Start the application code (Rest API) in Python | Pedro Lima |
| 4 | 3 | Develop Endpoints in the application | Pedro Lima |
| 5 | 3 | Postman Collections | Sofia Leitão |
| 6 | 4-7 | Develop transactions/queries and database adjustment | Every member |
| 7 | 8 | Final adjustments and tests | Every member |
| 9 | 8 | Report conclusion and User Manual | Every member |

The report, user manual were done continuously throughout the development of the project.

## 8 Conclusion

With this report, the group aims to represent the work done to the docent. The group accepts that documentation is an important part of developing database applications and software as a whole, and concludes that the project was adequately completed, even if there's small room for improvements.