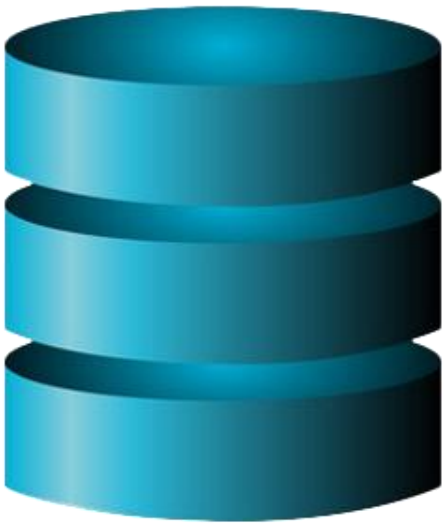
Final Report

Hospital Management System

Instruction Manual

Installation Manual | User Manual | Development Plan



- Bruno Ricardo Madeira Tavares Mascarenhas

  Nº de aluno: 2020196055
- Carlos Miguel Matos Soares

  Nº de aluno: 2020230124
- Miguel Proença Simões Aragão Machado

  Nº de aluno: 2020222874

Base de Dados
Docente: Adorilson Araújo

Indíce

Installation Manual

Prerequisites:

- **Python:** Python 3.7 or higher.
- **PostgreSQL:** PostgreSQL 12 or higher.
- **Python Packages:** Flask, psycopg2, pyjwt (for our implementation).
- **pgAdmin 4**
- **Postman**

Database:

- Create or access the database. This can be done via terminal/cmd using psql or through pgAdmin. In this case, the database is called projeto2024.

- Run the provided scripts to create and populate the tables with some sample data to facilitate testing. These are the updated versions of the ER model provided in the first project deliverable.

API Server:

- To run the API Server, we first need to install the packages described in the Prerequisites section. These packages can be installed using pip or conda. It is recommended to use a virtual environment to avoid cluttering the main Python installation with unnecessary packages.

- Before running the "demo-api.py" file, update it (if necessary) with the correct PostgreSQL credentials as shown in Figure 1. After this update, start the application by executing the Python file.

- The server will start at http://127.0.0.1:8080.

```
def db_connection():
    db = psycopg2.connect(
        user='aulaspl',
        password='aulaspl',
        host='127.0.0.1',
        port='5432',
        database='projeto2024'
    )

    return db
```

Figure 1: Example of connection setup.

User Manual

This API implements the features that were discussed conceptually in the 1st Project Delivery Report. The interaction and testing will be done through the Postman application.

To make it more readable we organized it in the following way:

- Authentication
  - Login
- User Registration
  - Register User
- Appointments
  - Schedule Appointment
  - View Appointments
- Surgeries
  - Schedule Surgery
- Prescriptions
  - Get Prescriptions
  - Add Prescription
- Billing
  - Execute Payment
- Reports
  - Get Top 3 Patients
  - Get Daily Summary
  - Generate Monthly Report

The test and results structure follow the written project guideline provided by the teachers.

Now, we will show the features, one by one, explaining the method, the expected response, and some errors you can encounter while testing.

Authentication

1. **Login:**
   - **Endpoint:** /dbproj/user
   - **Method:** PUT
   - **Headers:** Content-Type: application/json

```
{
    "username": "paciente1",
    "password": "senha123"
}
```

Figure 2: Body.

```
{
    "status": 200,
    "results": "auth_token"
}
```

Figure 3: Expected Response.

```
{
    "status": 400,
    "errors": "Username or password missing"
}
```

Figure 4: Possible Error – Missing username or password.

```
{
    "status": 401,
    "errors": "Invalid Credentials"
}
```

Figure 5: Possible Error – Invalid Credentials.

User Registration

1. **Register user:**
   - **Endpoint:** /dbproj/register/<user_type>
   - **Method:** POST
   - **Headers:** Content-Type: application/json

Figure 6: Body.

Figure 7: Expected Response.

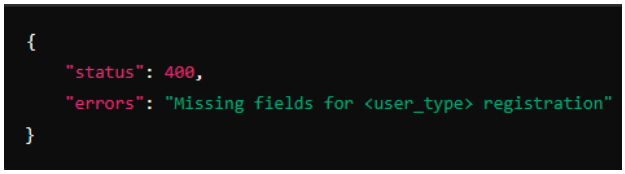


Figure 8: Possible Error – Missing fields for <user_type> registration.

Figure 9: Possible Error – Database error message.

Appointments

1. Schedule Appointment:

- **Endpoint:** /dbproj/appointment
- **Method:** POST
- **Headers:** Content-Type: application/json | Authorization: Bearer <token>

```
{
    "doctor_id": 1,
    "date": "2024-06-01",
    "purpose": "Consulta de rotina"
}
```

Figure 10: Body.

```
{
    "status": 200,
    "results": "<appointment_id>"
}
```

Figure 11: Expected Response.

```
{
    "status": 400,
    "errors": "Missing required fields"
}
```

Figure 12: Possible Error – Missing required fields.

```
{
    "status": 401,
    "errors": "Unauthorized access"
}
```

Figure 13: Possible Error –Unauthorized access.

2. View Appointment:

- **Endpoint:** /dbproj/appointments/<patient_user_id>
- **Method:** GET
- **Headers:** Authorization: Bearer <token>

```
{
    "status": 200,
    "results": [
        {
            "id": "<appointment_id>",
            "doctor_id": "<doctor_id>",
            "date": "<date>",
            "doctor_name": "<doctor_name>"
        }
    ]
}
```

Figure 14: Expected Response.

```
{
    "status": 403,
    "errors": "You are not authorized to view these appointments"
}
```

Figure 15: Possible Error – Unauthorized access.

```
{
    "status": 404,
    "errors": "No data found"
}
```

Figure 16: Possible Error – No data found.

```
{
    "status": 500,
    "errors": "Database error message"
}
```

Figure 17: Possible Error – Database error message.

Surgeries

1. Schedule surgery:

- **Endpoint:** /dbproj/surgery or /dbproj/surgery/<hospitalization_id>
- **Method:** POST
- **Headers:** Content-Type: application/json | Authorization: Bearer<token>

```
{
    "patient_id": 1,
    "doctor": 2,
    "nurses": [[3, "chefe"], [4, "assistente"]],
    "date": "2024-06-15"
}
```

Figure 18: Body.

```
{
    "status": 200,
    "results": {
        "hospitalization_id": "<hospitalization_id>",
        "surgery_id": "<surgery_id>",
        "patient_id": "<patient_id>",
        "doctor_specialization_employee_employee_id":
        "date_surgery": "<date_surgery>",
        "bill_id": "<bill_id>"
    }
}
```

Figure 19: Expected Response.

```
{
    "status": 404,
    "errors": "No hospitalization found with the given ID"
}
```

Figure 20: Possible Error – No hospitalization found with the given ID.

Other errors:

- And some more like the database error, the missing fields, unauthorized access, etc.
- Will try to show only errors that we haven´t seen in others.

Prescriptions

1. **Get prescriptions:**
   - **Endpoint:** /dbproj/prescriptions/<person_id>
   - **Method:** GET
   - **Headers:** Authorization: Bearer<token>

```
{
    "status": 200,
    "results": [
        {
            "id": "<prescription_id>",
            "validity": "<validity>",
            "posology": {
                "dose": "<dose>",
                "frequency": "<frequency>",
                "medicine": "<medicine>"
            }
        }
    ]
}
```

Figure 21: Expected Response.

```
{
    "status": 404,
    "errors": "No prescriptions found"
}
```

Figure 22: Possible Error – No prescription found.

Other errors:

- And some more like the database error, the missing fields, unauthorized access, etc.
- Will try to show only errors that we haven´t seen in others.

2. Add prescriptions:

- **Endpoint:** /dbproj/prescriptions/
- **Method:** POST
- **Headers:** Content-Type: application/json | Authorization: Bearer <token>

```
{
    "type": "appointment",
    "event_id": 1,
    "validity": "2024-07-01",
    "medicines": [
        {
            "medicine": "Aspirina",
            "posology_dose": "1 comprimido",
            "posology_frequency": "diário"
        }
    ]
}
```

Figure 23: Body.

```
{
    "status": 200,
    "results": "<prescription_id>"
}
```

Figure 24: Expected Response.

```
{
    "status": 403,
    "errors": "Unauthorized access, only doctors can use this endpoint"
}
```

Figure 25: Possible Error – If a person that is not a doctor tries to access.

```
{
    "status": 404,
    "errors": "Medicine not found in the database: <medicine>"
}
```

Figure 26: Possible Error – Medicine does not exist.

Billing

1. **Execute payment:**
   - **Endpoint:** /dbproj/bills/<bill_id>
   - **Method:** POST
   - **Headers:** Content-Type: application/json | Authorization: Bearer <token>

```
{
    "amount": 100,
    "payment_method": "cartão de crédito"
}
```

Figure 27: Body.

```
{
    "status": 200,
    "results": {
        "remaining_value": "<remaining_value>"
    }
}
```

Figure 28: Expected Response.

```
{
    "status": 403,
    "errors": "Unauthorized access, only patients can pay their bills"
}
```

Figure 29: Possible Error – If someone, not a patient, tries to pay a bill.

```
{
    "status": 400,
    "errors": "This bill is already paid"
}
```

Figure 30: Possible Error – Bill is already paid.

Reports

1. **Get top 3 patients:**
   - **Endpoint:** /dbproj/top3
   - **Method:** GET
   - **Headers:** Authorization: Bearer<token>

2. Get daily summary:

- **Endpoint:** /dbproj/daily/<year_month_day>
- **Method:** GET
- **Headers:** Authorization: Bearer<token>

3. **Generate monthly report:**
   - **Endpoint:** /dbproj/report
   - **Method:** GET
   - **Headers:** Authorization: Bearer<token>

ER Diagram

Now, we will provide the physical and conceptual version of our ER Diagram.

First, let´s look at the changes we made after the first deliver.

1. Additions

- **New Tables**
  - Two new tables were added to the diagram to improve the representation of data related to medicine posology and side effects.

- **Table “Posology”**
  - Fields:
    - dosage: String type, not null, up to 512 characters.
      - Example: "1 tablet"
    - frequency: String type, not null, up to 512 characters.
      - Example: "Twice a day"

  - Justification: The addition of this table aims to improve the control and specification of how drugs should be administered to patients, providing a detailed structure for storing dosage and frequency of use.

- **Table “Medicine_side_effects”**
  - Fields:
    - severity: String type, not null, up to 512 characters.
      - Example: "Moderate"

  - Justification: This table was introduced to better manage data on drug side effects, allowing the recording of the severity of these effects in a detailed and structured manner.

2. Modifications to Existing Tables

Significant changes were made to the following tables to correct inconsistencies, add new attributes, or modify existing ones to better meet system requirements:

Table “patient”

- Modification: Added the insurance_number field (String).
    - Example:
        - Added: insurance_number: "123456789"
- Justification: This field was added to record patient insurance information, facilitating the management of health insurance data.

Table “Appointment”

- Modification: Changed the date field from String to Date.
    - Example:
        - Current: date: 2023-01-01
- Justification: Changing the date field type improves data integrity and facilitates date comparison and manipulation operations.

Table “Medicine”

- Modification: Added the expiry_date field (Date).
    - Example:
        - Added: expiry_date: 2024-12-31
- Justification: This field was added to record the expiration date of drugs, allowing more rigorous control over the use of drugs within their expiration date.

Table “side_effects”

- Modification: Removed the description field.
    - Example:
        - Removed: description: "May cause nausea"
- Justification: This field was removed to avoid redundancy, as detailed descriptions of side effects are now managed in the Medicine_side_effects table.

3. Additions and Removals of Links

3.1 Links Added

New relationships between tables were established to reflect the newly added tables and to improve the referential integrity of the database.

- Link between “Prescription” and “Posology”

- Example:
  - Prescription.prescription_id -> Posology.prescription_id
  - Justification: This link was added to associate each prescription with its specific posology, allowing a clear and detailed record of how each drug should be administered.

-Link between Medicine and Medicine_side_effects

- Example:
  - Medicine.medicine_id -> Medicine_side_effects.medicine_id
  - Justification: This link was established to connect drugs to their respective side effects, facilitating the consultation and management of this information.

3.2 Links Removed

Some relationships between tables were removed to eliminate redundancies and correct incorrect relationships.

Removal of Link between “Patient” and “Insurance”

- Example:
  - Removed: Patient.patient_id -> Insurance.patient_id
  - Justification: This link was removed because insurance information is now directly stored in the patient table through the new insurance_number field, making the link redundant.

Removal of Link between Appointment and Doctor

- Example:
    - Removed: Appointment.doctor_id -> Doctor.doctor_id
    - Justification: The link was removed to simplify the model, as the relationship can be inferred through a separate relationship table or an additional field if necessary.

4. Justifications for the Changes

Addition of the Posology and Medicine_side_effects Tables:

- These tables were added to improve the management of information on posology and side effects, allowing a more detailed and accurate representation of this data.

Modifications to Existing Tables:

- Changes to existing tables were necessary to correct inconsistencies and adapt the model to new functional and business requirements identified during the project review.

Additions and Removals of Links:

- New relationships established between tables aim to increase the model's coherence, while the removal of redundant or incorrect links aims to simplify the diagram and improve database performance and maintenance.

Physical Version of ER Diagram:

Conceptual Version of ER Diagram:

Development Plan:

The goal of this project was to develop a database for a hospital management system in two main phases:

The first one consisted in the creation and implementation of the ER model and was the conceptual design stage of the project. During the midterm presentation we received feedback regarding some relations in the ER model and the first step was to address it and make small corrections - these were done during the midterm defense with the teacher's help.

The second phase, corresponding to this delivery, focused on API development and testing. Since not all group members had the same level of proficiency with the tools required—such as Postman, REST, and JSON, which were outside the core database course content—we needed to put in extra effort to study, understand, and teach each other these technologies. [1]

The first step was to implement the initial functionalities to connect to the database and to setup the API by following the teacher's tutorial, after that, we proceeded to setup the main JWT handling functions:

- **token_required()**
- **check_user()**
- **login()**
- **decode_token()**

The most important goal for us was to have the first working and complete version of the API by May 15th to give leeway to perform adjustments and corrections. This goal was achieved. The report was to be finished by at least May 21st but since the project due date was postponed, we also postponed the report to be finished on the 23rd, which was also successful. The only thing we did after that date was altering some of the formatting to make it more readable and replacing the API examples bellow with pictures taken from postman instead of text.

The following functionalities were done by the three members separately - each one did their own version - and discussed together afterwards to pick the best implementation. We found these to be the most difficult ones, hence the chosen approach.

- **get_top_three_patients()**
- **get_daily_summary(year_month_day)**
- **generate_monthly_report()**

In the next section, we outline the task distribution we agreed upon. While the tasks were assigned to individual team members, we maintained constant communication and provided feedback to each other throughout the process. No task was completed in isolation. For instance, although the presentation and report were officially assigned to one person each, they were developed as team efforts, with continuous revisions and improvements based on other team member's input and suggestions.

[1] *Unfortunately, one team member was ill during the entirety of April, which made us face the possibility of being short one member as the recovery time was not certain, however, it all turned out well and the team was able to overcome the challenge and reorganize.*

Task Distribution:

Carlos Soares:

1. Authentication and Authorization:
   - register_user(user_type)
   - insert_user(), insert_doctor(), insert_nurse(), insert_assistant(), insert_patient()

2. Scripts for Data Insertion:
   - Create scripts to fill the employee, patient, appointment, and prescription tables.

3. Presentation:
   - Write the presentation of the project.

Miguel Machado:

1. Appointments and Surgeries:
   - schedule_appointment()
   - see_appointments(patient_user_id)
   - schedule_surgery(hospitalization_id)

2. Prescriptions:
   - get_prescriptions(person_id)
   - add_prescription()

3. Scripts for Data Insertion:
   - Create scripts to fill the doctor_specialization, nurse, assistant, medicine and hospitalization tables.

Bruno Mascarenhas:

1. Billing and Payments:
   - execute_payment(bill_id)

2. Scripts for Data Insertion:
   - Create scripts to fill the surgery, bill, payment, and medicine tables.

3. Report and Development Plan:
   - Write the Report that includes the instruction manual and the development plan.

Overview of some API functionalities and code:

In this final section we present a summarized overview of some of the implemented functions we deemed more notable, to make it easier to understand the code in the context of the **SQL** logic. All of the functionalities that require token authentication are using JWT (JSON Web Token) logic to ensure the correct access.

Register User:

- Adds new users (doctors, nurses, assistants, or patients) to the database using **INSERT** statements.
- The **transaction** starts with a user registration request. All related operations (inserting into specific tables) are in a try-except block. The transaction begins with the first insert statement and commits at the end of the try block with conn.commit(). If any insert fails, a rollback happens with the except block - conn.rollback()- ensuring that all operations either succeed or fail.

Schedule Appointment:

- Adds a new appointment to the database with **INSERT**.
- Verifies the patient's identity using their token to ensure that the appointment is being scheduled by an authenticated user.
- The transaction starts with the first insert statement and if any part of the insertion process fails the transaction is rolled back, if it passes all the checks of the try block, it's commited.

See Appointments:

- Allows checking appointment details using **SELECT** combined with JOIN (inner) to get related data from multiple tables - inner **JOIN** ensures only matching data from tables is selected, in this case we want to get appointments and the corresponding doctor details.
- The appointment table is joined with the doctor_specialization table to link each appointment to the doctor's specialization.
- The doctor_specialization table is joined with the employee table to include all the doctor's details.
- In the end we verify the patient's info using **WHERE** to match with the patient id.

Get Prescriptions:

- Retrieves prescription details using SELECT statements combined with **INNER** and **LEFT JOINs** to get data from multiple tables. Like the previous point, this data is related.
- **INNER JOIN** is used for Posology and Medicine tables because each prescription must have a defined dosage, frequency, and corresponding medicine which ensures only elements with complete dosage and medicine information are included.
- **LEFT JOIN** is used for Appointment_Prescription, Appointment, Hospitalization_Prescription and Hospitalization to guarantee that all prescriptions are selected, even if they are only linked to appointments or hospitalizations. If there are prescriptions without a corresponding appointment or hospitalization (or if some details are missing in the tables), they are still included in the results.

Execute Payment:

- Retrieves bill details and updates the bill's status using **SELECT** and **UPDATE**. The payment is recorded with **INSERT**.
- We handle the concurrency with **Row-Level Locking**, **SELECT** lock the rows for the duration of the transaction which prevents other transactions from modifying the same bill.

Get Top Three Patients:

- Gets the top three patients based on their monthly payments.
- We use **WITH** to organize one query into smaller parts. **MonthlyPayments** keeps total monthly payments for each patient (uses SUM to calculate the total amount spent by each patient and GROUP BY to organize the data by patient ID) and **ProcedureDetails** saves information about each patient's appointments and hospitalizations.
- To link payments with appointments or hospitalizations - to cover all payment records - we use **LEFT JOIN** in **MonthlyPayments** and we also use it in **ProcedureDetails** to include all of that information for each patient.
- **INNER JOIN** is used between **MonthlyPayments** and **Patient** so that only patients with payment records are included in the results and another **INNER JOIN** is used between **MonthlyPayments** and **ProcedureDetails** to link the payment data with patient information.

Get Daily Summary:

- Gets a summary of daily activities (amount spent, number of surgeries, and prescriptions).
- **LEFT JOIN** is used between **Hospitalization** and **Payment** tables to calculate the total amount spent on the specified date and we collect the payment ammounts with SUM.
- We then do three subqueries with **SELECT COUNT**, one to count the surgeries (by querying the **Surgery** table) and two to count the perscriptions related to apointment (inner joining Perscription with **Appointment_Perscription** and **Appointment tables**) and the ones related to hospitalizations(**joining Perscription** with **Hospitalization_Perscription** and **Hospitalization tables**).

Conclusion on API functionalities and code:

In conclusion, the API functionalities implemented in the Hospital Management System provide a robust and secure solution for managing hospital operations. The use of **JWT** for authentication ensures that only authorized users can access and perform operations, **enhancing data security**. By leveraging advanced **SQL** techniques and comprehensive transaction management, the system maintains data integrity and consistency, even during complex operations.

The key functionalities, such as user registration, appointment scheduling, and prescription retrieval, demonstrate the system's ability to handle detailed and interconnected data efficiently. The use of **INNER** and **LEFT JOINs** in **SQL queries** ensures that related information is accurately retrieved, providing a complete view of the data necessary for effective hospital management.

Additionally, the API supports efficient payment processing with secure transaction handling and offers valuable analytical insights through functions like **Get Top Three Patients** and **Get Daily Summary**. These features support informed decision-making and resource allocation, ultimately improving the operational efficiency of the hospital and the quality of care provided to patients.

Overall, the system's design prioritizes security, data integrity, and user experience, making it a reliable and scalable solution for hospital management needs.

Some final considerations

During the development of the Hospital Management System, our team encountered several challenges that allowed us to enhance our technical skills and teamwork. Below, we present some of our opinions and final considerations about the project.

Technical challenges

Integration of Technologies: Integrating different technologies such as Python, PostgreSQL, and various Python libraries was a significant initial challenge. However, this process provided us with a deeper understanding of how these tools can work together to create a robust application.

Data Modeling: Creating the ER (Entity-Relationship) model was a complex task due to the need to accurately represent the structure and operations of a hospital. It was necessary to review and adjust the model multiple times to ensure it met all functional requirements.

Security and Authentication: Implementing a secure authentication system using JWT (JSON Web Tokens) was essential to protect sensitive patient data. This aspect taught us a lot about best security practices in web applications.~

Lessons learned

Importance of Documentation: Maintaining clear and up-to-date documentation was crucial for the project's success. Documentation not only helped integrate new team members but was also vital for the system's maintenance and updates.

Collaboration and Communication: We learned that effective communication and collaboration among team members are key to overcoming obstacles and meeting deadlines. Project management and communication tools like Trello and Slack were indispensable.

Testing and Debugging: Establishing a rigorous testing process, both automated and manual, was vital to ensure the system's functionality and reliability. Quickly identifying and fixing bugs helped maintain the quality of the final product.

Final considerations

The development of the Hospital Management System was an enriching experience that provided us with valuable insights into the software development lifecycle. The practical application of classroom knowledge better prepared us for future challenges in our professional careers.

We believe that the project not only meets the defined requirements but also serves as a solid foundation for future expansions and improvements. We are proud of the work accomplished and confident that this system can provide significant benefits for hospital management.