# Databases Project
## 2024/2025

(2025/02/18: versão 1.1 do enunciado, pode ser usada o psycopg3)
(2025/02/17: versão 1.0 do enunciado)

## Introduction

The goal of this project is to provide students with experience in developing database-centric systems.

## Objectives

After completing this project, students should be able to:

- Understand how a database application development project is organized, planned, and executed
- Master the creation of conceptual and physical data models for supporting application data
- Design, implement, test, and deploy a database system
- Install, configure, manage, and tune a modern relational DBMS
- Understand client and server-side programming in SQL and PL/pgSQL

## Groups

The project is to be done in **groups of 3** students.
**IMPORTANT:** the members of each group **must** be enrolled in PL classes with the same Professor.

## Quality attributes for a good project

Your application must make use of:

(a) A transactional relational DBMS (e.g., PostgreSQL)
(b) A modelling tool (i.e., ONDA)
(c) A distributed database application architecture, providing a REST API
(d) SQL and PL/pgSQL
(e) Adequate triggers and functions/procedures running on the DBMS side
(f) Good strategies for **managing transactions** and **concurrency conflicts**, and **database security**
(g) Good **error avoidance, detection, and mitigation strategies**
(h) Good documentation

Your application must also respect the functional requirements defined in **Annex A** and execute without "visible problems" or "crashes". To fulfill the objectives of this assignment, you can be as creative as you want, provided you implement the list of required features.

Do not start coding right away - take time to think about the problem and structure your development plan and design.

**Plagiarism or any other form of fraud will not be tolerated. Content generated by AI-based systems (e.g., ChatGPT, Gemini, etc.) does not qualify as student-produced work. Any use of AI assistance must be explicitly disclosed; failure to do so will be regarded as an attempt of fraud.**

## Milestones and deliverables

**Midterm delivery (35% of the grade) –** 23:55 of March 21st

The group must sign up for an available time slot for the defense in Inforestudante, before the delivery due date (the list of available slots will be released before the deadline). **All students must be present, but the defense is individual, ordered by name.** The following artifacts must be uploaded at Inforestudante until the deadline (each group must select a member for performing this task):

- **Report** with the following information:
  - Team members and contacts
  - Brief description of the project
  - Definition of:
    - Transactions (where the concept of transactions is particularly relevant, multiple operations that must succeed or fail as a whole)
    - Potential concurrency conflicts (those that are not directly handled by the DBMS) and the strategy to avoid them
  - Development plan: planned tasks, initial work division per team member, timeline
- **ER diagram**
  - Use ONDA to design the ER, and include a description of entities, attributes, integrity rules, etc.
- **Relational data model**
  - Use ONDA to output the physical model of the database (i.e., the tables)
- **The ONDA project, exported as a JSON file.**

**Quick checkpoint 1 (5% of the grade) –** During PL class on the 12th week; everyone's presence in class is mandatory, slots will be made available in Inforestudante

- **DBMS** installed and configured with the database of the project
- **REST API** (provided in the demo) working, connected to the project DB, with a basic retrieval endpoint (available in the demo)

**Quick checkpoint 2 (5% of the grade) –** During PL class on the 14th week; everyone's presence in class is mandatory, slots will be made available in Inforestudante

- **Authentication** functionality implemented (login endpoint & authorization example in a separate endpoint)

**Final delivery (55% of the grade) –** 23:55 of May 22nd

The project outcomes must be submitted to *Inforestudante* by the deadline. Each group must select a member for performing this task.

- **Final report** with (the following items in a single document):
  - **Installation manual** describing how to deploy and run the software you developed
  - **User manual** describing the submitted Postman collection requests to test the application
  - **Final ER and relational data models (ONDA jpeg outputs)**
  - **Execution report:** make sure you specify which tasks were done by each team member and the effort involved (e.g., hours)
  - **All the information, details, and design decisions you consider relevant to understand how the application was built and how it satisfies the requirements of the project (do not make generic descriptions of the endpoints, include only relevant information regarding your process and implementation)**
- **Video demo of project (Max. 5-min)**
- **Source code and Scripts:**

- Include the source code, scripts, executable files, and libraries necessary for compiling and running the software
- DB creation scripts containing the definitions of tables, constraints, sequences, users, roles, permissions, triggers, functions, and procedures
- Postman collection with all the requests required to test the application
- The ONDA project, exported as a JSON file.

**Defense – June 2nd to June 6th**

- The group must sign up for any available time slot for the defense in *Inforestudante,* before the delivery due date (the list of available slots will be released before the deadline for the final delivery)
- **All students must be present, but the defense will be individual, ordered by name**

## Assessment

- This project accounts for 6 points (out of 20) of the total grade in the Databases course
- Midterm delivery corresponds to 35% of the grade of the project
- Checkpoints account for 10% (5%+5%) of the grade of the project
- Final submission accounts for the remaining 55 % of the grade of the project
- The minimum grade is 35%; failure to meet this still allows you to attend the exam but the project grade will be 0
- **Endpoints or functionalities submitted but not developed by the group will be considered as an attempt of fraud**

## Technical constraints

- The Entity-relationship model should be created using **ONDA**
- The project is to be developed in **Python**
- The project must use the **PostgreSQL** DBMS
- The interaction with the REST API should be done using **Postman** tool
- The use of ORMs is **not allowed**
- SQL queries should be created using the operators taught in the course**; other operators/functionalities are allowed but need to be explicitly justified!**

## Annex A: University Management System – Functional Description

The University Management System is a comprehensive platform designed to streamline and manage academic and extracurricular activities within a university. It handles course administration, student enrollment, class scheduling, financial tracking, and evaluation processes. The system ensures efficient management of student records, supports instructors in grading and reporting, and facilitates extracurricular participation while maintaining an up-to-date record of financial transactions and academic performance.

Courses are uniquely identified by course codes. Each edition of a course is managed by a coordinator instructor and may include multiple assistant instructors. The edition is scheduled with a defined timetable and can include multiple classes, such as Theory (T) and Practical Lab (PL). These classes are assigned to specific classrooms within campus buildings, with details such as capacity and location. Each course includes prerequisites, which are other courses that must be completed before enrollment, and each edition is subject to capacity limits. Additionally, courses may be associated with multiple degree programs.

Students in the system are identified by a unique identifier and are described by their personal information, academic records, and financial details. Student profiles are created by administrative staff, who are also responsible for enrolling students into degree programs. This enrollment generates a debt entry in the student's financial account (**this process must be implemented using triggers**). When enrolling in a course, students must select the specific classes they wish to attend. The system then adjusts the available capacity of the course edition and verifies that all prerequisites are met (a warning must be provided to the user when trying to enroll in an already full course). Students are restricted to enrolling in courses associated with their active degree program, although they may be linked to multiple degrees over time. The system accommodates hundreds of students per course and tracks individual attendance for each class.

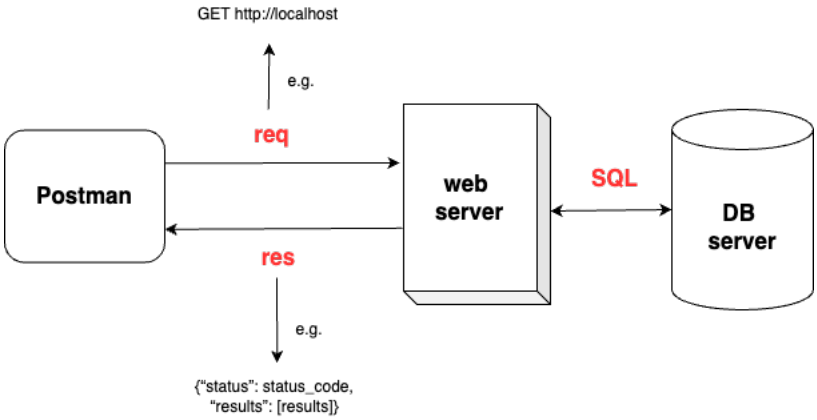
When instructors submit grades for a course, the system updates the approved students' academic averages within their respective degree programs and adjusts the count of approved courses for each student. The system supports multiple evaluation periods, and summaries of course edition evaluation results, such as average grades and the number of approved students, should be easily accessible for each period.

Beyond academic management, the system supports student participation in extracurricular activities, including clubs and sports. When a student registers for an activity, the associated fees are reflected in their financial account. If the student withdraws from the activity, the system reverses these updates to ensure accurate financial records.

The financial account of each student tracks tuition, fees, and other charges. Financial interactions, such as withdrawing from a degree program or an extracurricular activity, trigger recalculations of charges to maintain accurate and up-to-date records.

## Technical Description

The figure represents a simplified view of the system to be developed. The system must be made available through a REST API that allows the user to access it through HTTP requests (when content is needed, JSON must be used). As it can be seen, the user interacts with the web server through a REST request/response exchange, and in turn, the web server interacts with the database server through an SQL interface (e.g., psycopg2/psycopg3 in the case of Python). This is one of the most used architectures today and supports many web and mobile applications we use daily. Since the course focuses on the design and operation of a data management system and associated functionalities, the development of web or mobile applications is outside the scope of this work. To use or test the REST API, you must use the REST client Postman (postman.com).




To facilitate the development of the project and focus on the data management system, groups should follow/use the demo code that will be made available. An example of how to test the REST API endpoints using Postman is also provided. In the event of missing details, groups should explicitly identify them in the report.

HTTP works as a request-response protocol. For this work, three main methods might be necessary:

- **GET**: used to request data from a resource
- **POST**: used to send data to create a resource
- **PUT**: used to send data to update a resource
- **DELETE**: used to delete a resource

REST API responses should follow a simple but rigid structure, always returning a status code (this is a very simplified version of how a real REST API would work):

- 200 – success
- 400 – error: bad request (request error)
- 500 – error: internal server error (API error)

If there are errors, they must be returned in *errors*, and if there are data to be returned, they must be returned in *results* (as can be seen in the details of the endpoints that follow).

## Functionalities to be developed

The *endpoints* of the REST API should be **strictly followed**. Any details that are not defined should be explained in the report.

When the user starts using the platform, he/she can choose between registering a new account or logging in using an existing account. The following *endpoints* should be used.

***NOTICE**: This is a simplified scenario. In a real application, a secure/encrypted connection would be used to send the credentials (e.g., HTTPS).*

**User Authentication.** Login using the *username* and the *password* and receive an *authentication token* (e.g., JSON Web Token (JWT), https://jwt.io/introduction ) in case of success. This *token* should be included in the *header* of the remaining requests.

**req** **PUT** **http://localhost:8080/dbproj/user**
`{"username": username, "password": password}`

**res** `{"status": status_code, "errors": errors (if any occurs), "results": auth_token (if it succeeds)}`

After the authentication, the user can perform the following operations using the obtained *token* during the user authentication (the token should always be passed in every request, either in the body or in authentication headers). **Some operations are restricted to certain types of users.**

**Add Student, Staff, Instructor.** Create a new individual, inserting the data required by the data model. Take into consideration the individual type and attributes to be consider. Do not forget that different types of users have different attributes. Avoid duplicated code. Remember that user details may contain sensitive data. Only staff can add new individuals (one account should be added by default when creating the database).

**req** **POST** http://localhost:8080/dbproj/register/student
**POST** http://localhost:8080/dbproj/register/staff
**POST** http://localhost:8080/dbproj/register/instructor
{"username": username, "email": email, "password": password, (…)}

**res** {"status": status_code, "errors": errors (if any occurs), "results": user_id (if it succeeds)}

**Enroll Degree**. Enroll student in degree program, inserting the data required by the data model. Only a staff user can use this endpoint.

**req** **POST** http://localhost:8080/dbproj/enroll_degree/{degree_id}
{"student_id": student_id, "date": date}

**res** {"status": status_code, "errors": errors (if any occurs)}

**Enroll Activity**. Enroll student in activity, inserting the data required by the data model. Only a student can use this endpoint.

**req** **POST** http://localhost:8080/dbproj/enroll_activity/{activity_id}

**res** {"status": status_code, "errors": errors (if any occurs)}

**Enroll Course Edition**. Enroll course edition, inserting the data required by the data model. Only a student can use this endpoint.

**req** **POST** http://localhost:8080/dbproj/enroll_course_edition/{course_edition_id}
{"classes": [class_id1, class_id2, (…)], (…)}

**res** {"status": status_code, "errors": errors (if any occurs)}

**Submit Grades**. Submit course edition grades. Only the coordinator instructor of the respective course can do this.

**req** **POST** http://localhost:8080/dbproj/submit_grades/{course_edition_id}
{"period": evaluation_period, "grades": [[student_id1, grade1], [student_id2, grade2], [student_id3, grade3]], (…)}

**res** {"status": status_code, "errors": errors (if any occurs)}

**List Student Course Details**. List all student enrolled courses' details. Most recent results should be returned first. Only staff and the target user can use this endpoint.

**req** **GET** http://localhost:8080/dbproj/student_details/{student_id}

**res** {"status": status_code, "errors": errors (if any occurs), "results": [{"course_edition_id": course_edition_id1, "course_name": course_name1, "course_edition_year": year, "grade": grade}, …] (if it succeeds)}

**List Degree Courses Details**. List all the courses and editions and respective details for a given degree. Most recent results should be returned first. Only staff can use this endpoint. Just **one** *SQL* query should be used to obtain the information.

**req** **GET** http://localhost:8080/dbproj/degree_details/{degree_id}

**res** {"status": status_code, "errors": errors (if any occurs), "results": [{"course_id", course_id, "course_name": course_name1, "course_edition_id": course_edition_id1, "course_edition_year": year, "capacity": capacity, "enrolled_count", enrolled, "approved_count": approved, "coordinator_id": instructor_id, "instructors": [instructor_id1, instructor_id2, (…)]}, (…)] (if it succeeds)}

**List Top 3 students**. Get the top 3 students considering the average grades for the current academic year. The result should discriminate the grades by course and the activities each student is enrolled in. Just **one** *SQL* query should be used to obtain the information. Only staff can use this endpoint.

**req** **GET** http://localhost:8080/dbproj/top3

```
res   {"status": status_code, "errors": errors (if any occurs), "results": [
         {"student_name": student_name1, "average_grade": average_grade, "grades":
      [{"course_edition_id": course_edition_id1, "course_edition_name": course_edition_name1,
      "grade": grade, "date": date}, (…)], "activities": [activity1, activity2, (…)]},
         {"student_name": student_name2, "average_grade": average_grade, "grades":
      [{"course_edition_id": course_edition_id1, "course_edition_name": course_edition_name1,
      "grade": grade, "date": date}, (…)], "activities": [activity2, activity4, (…)]},
      ] (if it succeeds)}
```

**List Best Student by District**. List only the students that have the highest average of their district. Just **one** *SQL* query should be used to obtain the information. Only staff can use this endpoint.

```
req   GET http://localhost:8080/dbproj/top_by_district/
res   {"status": status_code, "errors": errors (if any occurs), "results": [{"student_id":
      student_id1, "district": district, "average_grade": average_grade}, (…)]}
```

**Generate a monthly report**. Get a list of the courses (editions) with more approved students each month in the last 12 months. Just **one** *SQL* query should be used to obtain the information. Only staff can use this endpoint.

```
req   GET http://localhost:8080/dbproj/report
res   {"status": status_code, "errors": errors (if any occurs), "results": [
         {"month": "month_0", "course_edition_id": course_edition_id3, "course_edition_name":
      course_edition_name3, "approved": count_approved, "evaluated": count_evaluated},
         {"month": "month_1", "course_edition_id": course_edition_id2, "course_edition_name":
      course_edition_name2, "approved": count_approved, "evaluated": count_evaluated },
         {"month": "month_2", "course_edition_id": course_edition_id1, "course_edition_name":
      course_edition_name1, "approved": count_approved, "evaluated": count_evaluated },
      (…)
      ]}
```

**Remove Student Data**. Remove all data related to a student. Only staff can use this endpoint.

```
req   DELETE http://localhost:8080/dbproj/delete_details/{student_id}
res   {"status": status_code, "errors": errors (if any occurs)}
```

## FINAL REMARKS

- As it is defined in some *endpoints*, the implemented solution should obtain the information using **a single SQL *query*** on the server side. This does not include potential steps to validate the user authentication. **Nevertheless**, in case you cannot solve it with a single query, it is preferable to use more *queries* than not implementing the *endpoint*.
- The data processing (e.g., order, restrictions) should be done in the *queries* whenever possible (and not in code).
- It is important that you can master PL/pgSQL procedures and triggers (at least some endpoints should make use of them), but you do not have to implement every functionality using procedures.
- The transaction control, concurrency, and security will be considered during the evaluation.