FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

departamento de engenharia informática

Plataforma de Gestão Académica

Projeto Final

Base de Dados

Anaísa Castela Rosa – 2023211854

Cecília Ernesto Silva Quaresma – 2024245307

Joana Correia Vilas Boas – 2023223186

Índice

Introdução

Este projeto consiste no desenvolvimento de uma plataforma de gestão académica que centraliza os dados de uma instituição de ensino. O objetivo é criar um sistema capaz de gerir registos de utilizadores (alunos, docentes e staff administrativo) e de dar suporte às funcionalidades académicas necessárias, tais como registo de alunos em cursos e disciplinas, matrícula em atividades extra-curriculares, submissão de classificações, consulta de resultados, entre outras.

A entidade principal é Utilizador, que inclui campos como *userID* (identificador único), *mail*, *palavra-passe*, *name* e *username*. Os diferentes tipos de utilizadores herdam desta tabela genérica: existem tabelas específicas **Admin**, **Student** e **Instructor** que se associam um-para-um com a tabela *Utilizador*, permitindo armazenar atributos particulares a cada papel. Por exemplo, *Student* possui campos adicionais como *balance* e *district*, enquanto *Admin* e *Instructor* podem ter campos próprios. Estas ligações refletem herança de dados, garantindo que cada aluno ou docente é também um registo na tabela *Utilizador* com credenciais e informações comuns.

O modelo de dados define ainda entidades académicas como Course, Version, Lesson, Classroom, Degree_program, Grades, Evaluation_period, Activities_extra, Transactions e Enrollment. Em conjunto, estas tabelas suportam as funcionalidades do sistema. Por exemplo, um *Course* identifica uma disciplina (campo *course_id* e *course_name*), enquanto *Version* armazena as diferentes edições dessa disciplina em semestres/anos (com *version_id*, *semester*, *year*, *capacity*, *coordinator*, etc.). A tabela *Enrollment* relaciona alunos (student_id) com cursos/versões ou atividades, permitindo registar o estado da matrícula (*status* e data).

As principais operações do sistema incluem (entre outras): registar novos utilizadores nos seus respetivos papéis (aluno, docente, staff), autenticar utilizadores, matricular alunos em graus académicos, cursos e atividades, submeter e consultar notas de disciplinas, e gerir informações académicas. A gestão das permissões é feita de acordo com o papel do utilizador: por exemplo, apenas um utilizador do tipo staff pode inscrever alunos nos cursos ou criar novos utilizadores, enquanto alunos podem aceder somente às suas funcionalidades (matricular-se em atividades, ver notas próprias, etc.). A arquitetura do sistema é distribuída, com uma API REST implementada em Python Flask no servidor e um SGBD PostgreSQL. O código Flask fornecido implementa estes endpoints consultando e atualizando as tabelas definidas no modelo JSON fornecido, garantindo que a base de dados segue o esquema especificado.

O vídeo de demonstração do projeto foi criado na plataforma LOOM e foi dividido em duas partes, disponíveis nos links a seguir:

Parte1:
https://www.loom.com/share/57c2d5aa9cf64cebac39e6fd8f27e481?sid=51cafae1-bcce-411a-a02f-7fc5ffd90cb6

Parte 2:

https://www.loom.com/share/ce3d353356674b2f96fff990451f7643

Manual de Instalação

Para instalar e correr a aplicação Flask desenvolvida, recomendamos seguir estes passos:

Pré-requisitos:

- Python 3.x: Deve estar instalado no sistema, já que a aplicação backend utiliza Python com Flask.
- pip: Gestor de pacotes do Python para instalar dependências. Geralmente já vem incluído nas distribuições de Python atuais.
- PostgreSQL: Versão 13 ou superior deve estar instalada e em execução, pois a aplicação utiliza PostgreSQL como SGBD.
- Ferramenta de linha de comandos do PostgreSQL (psql): Para criação/manutenção da base de dados.
- Editor de código / IDE: Para visualizar e modificar o código Python ou scripts.

Instalação de dependências

No diretório do projeto, instale as bibliotecas Python necessárias. Estas incluem, tipicamente:

- Flask: Framework web para Python.
- psycopg2: Biblioteca de ligação ao PostgreSQL.
- PyJWT: Para criar e validar tokens JWT.

Criação da Base de Dados:

1. Criação de uma base de dados nova na aplicação PgAdmin, essa com um usuário e senhas exclusivos para esse projeto.
2. Criação das tabelas de acordo com o modelo de dados fornecido. Pode ser feito executando scripts SQL. Com base no JSON do modelo, e no nosso diagrama ER desenvolvido, cria-se as tabelas com os campos e constraints definidos. Por exemplo:
    - Tabela Utilizador com campos *(userID SERIAL PRIMARY KEY, mail VARCHAR UNIQUE NOT NULL, palavra_passe VARCHAR NOT NULL, name VARCHAR NOT NULL, username VARCHAR UNIQUE NOT NULL)*.
    - Tabela Student com campos específicos (ex.: *balance NUMERIC NOT NULL, district VARCHAR NOT NULL* e *userID* como chave estrangeira).
    - Outras tabelas (Admin, Instructor, Course, Version, etc.) de acordo com o modelo JSON.

3. Criação de uma função create_default_staff() para definir um utilizador admin inicial para poder utilizar o sistema. Com isso, podemos aceder ao Postman e testar endpoints que funcionam apenas com a autenticação de admin/staff.

Configuração e execução da aplicação

- Configuração: No início do código Flask há relações para conectar-se à base de dados e aceder à chave para autenticação dos Tokens JWT. Configuramos o sistema para receber esses dados de um arquivo .env, que deve ser configurado de acordo com os dados do utilizador.
- Executar a aplicação: Com tudo configurado, inicie o servidor Flask. Por exemplo, na linha de comandos Python:
    - python app.py

Certifique-se de que o endereço e porta (por exemplo http://localhost:8080) estão corretos de acordo com o código. Uma vez executando, o servidor disponibilizará os endpoints da API REST (geralmente prefixados por /dbproj/, conforme o enunciado).

Após a configuração, teste se a aplicação inicia sem erros. Se tudo estiver correto, poderá consultar os logs no terminal e verificar que o Flask está a executar pedidos nas rotas definidas.

Manual do Utilizador

Este manual explica como interagir com a API REST da plataforma através de uma ferramenta de testes de API, como o Postman. A aplicação disponibiliza endpoints para autenticação e operações relacionadas com a gestão académica. Para aceder aos endpoints protegidos, deve primeiro efetuar *login* para obter um token de autenticação (JWT). Em seguida, utiliza-se esse token em cada pedido subsequente no campo de cabeçalho Authorization: Bearer <token>. O utilizador também deve conhecer o seu papel (aluno, staff ou instrutor), pois algumas operações só estão disponíveis a certos papéis.

Endpoints e Coleção de Testes

Fornece-se uma coleção Postman com todos os pedidos definidos. Para utilizá-la:

1. Importe a coleção no Postman: Abra o Postman e importe o ficheiro .json da coleção de pedidos.
2. Realize o pedido de login: Envie um POST para /dbproj/login com um JSON contendo o *username* (nome de utilizador) e *palavra_passe*. Se a autenticação for bem-sucedida, a resposta incluirá um token JWT. Copie-o.
3. Configure o token: No Postman, defina uma variável de ambiente ou adicione manualmente o header Authorization: Bearer <token> para os pedidos seguintes.
4. Execute os endpoints de teste: Use os pedidos já definidos na coleção. Cada endpoint está configurado com o método HTTP e o corpo JSON apropriados. Edite os parâmetros (por exemplo, IDs de cursos, utilizadores) conforme necessário.

Principais endpoints disponíveis (ilustrativos):

- Autenticação: POST /dbproj/login – recebe JSON com { "username": "...", "password": "..." } e retorna { "token": "<JWT>" }.
- Registo de utilizadores:
    - POST /dbproj/register/student – regista um novo aluno; só acessível a staff. Exemplo de corpo: { "username": "...", "email": "...", "password": "...", "name": "...", "district": "...", "balance": 0 }.
    - POST /dbproj/register/staff – regista um novo funcionário (admin). Corpo JSON semelhante (sem district/balance).
    - POST /dbproj/register/instructor – regista um novo instrutor (docente).
- Matricular em grau académico: POST /dbproj/enroll_degree/{degree_id} – matricula um aluno num programa de estudos. Enviar o *student_id* e data no corpo JSON. Reservado a utilizadores do tipo staff.
- Matricular em atividade extra: POST /dbproj/enroll_activity/{activity_id} – matricula o aluno autenticado na atividade de ID dado. Disponível apenas para estudantes.

- Matricular em curso/edição: POST /dbproj/enroll_course_edition/{course_edition_id} – matricula um aluno nas turmas de uma edição de curso. Enviar lista de IDs de turmas.
- Submeter Notas: POST /dbproj/submit_grades/{course_edition_id} – submeter as notas finais da edição de curso; só o instrutor coordenador pode executar. O corpo JSON inclui "period": <periodo_de_avaliação>, "grades": [[student_id, nota], ...].
- Consultas: GET /dbproj/courses – obtém lista de disciplinas; GET /dbproj/users/{id}, etc. Os endpoints GET geralmente estão disponíveis a vários perfis (estudante pode ver seus dados; staff pode listar).

Em cada pedido (exceto o de login), inclui-se sempre o token JWT no cabeçalho Authorization. Se o token não for válido ou faltar, o servidor responde com erro de autorização. As respostas variam conforme o endpoint, tipicamente retornando o status da operação e, quando aplicável, dados em JSON (como listas de cursos, detalhes de alunos, etc.).

Diagramas

Diagrama ER

O Diagrama Entidade-Relacionamento ilustra as entidades do modelo de dados e suas ligações conforme o esquema fornecido. Nele visualiza-se, por exemplo, como Utilizador está ligado a Student, Instructor e Admin (herança/conexões 1:1), como Course se relaciona com Version (curso possui várias versões/edições), e como Enrollment, Transactions, Grades e outras tabelas se interligam.

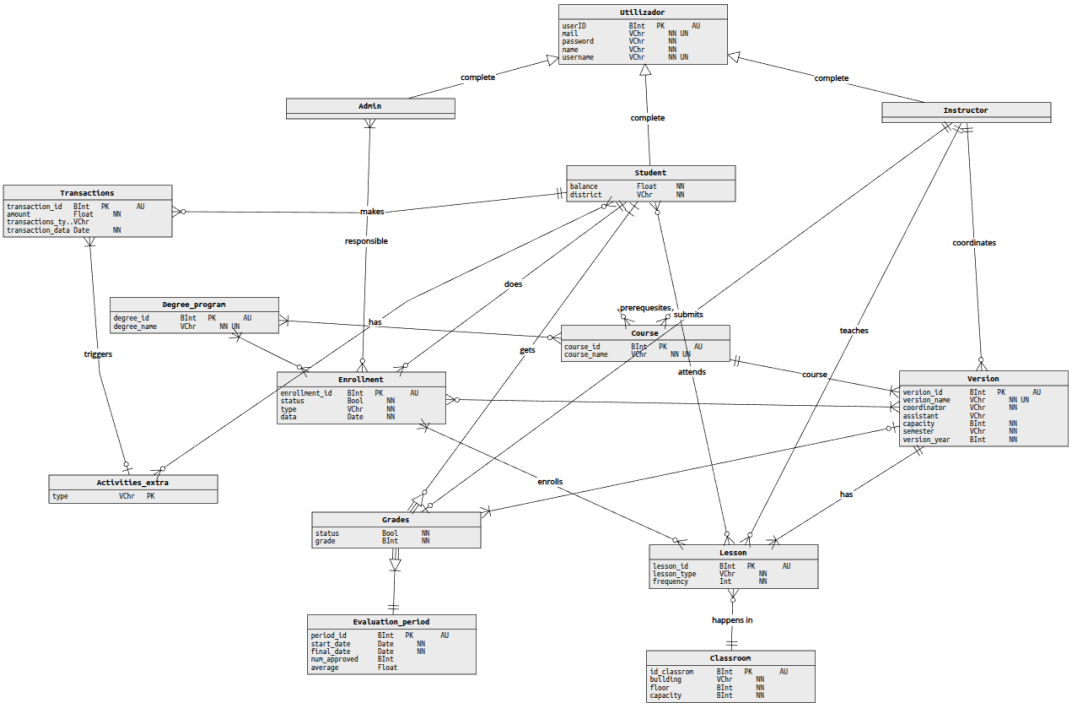
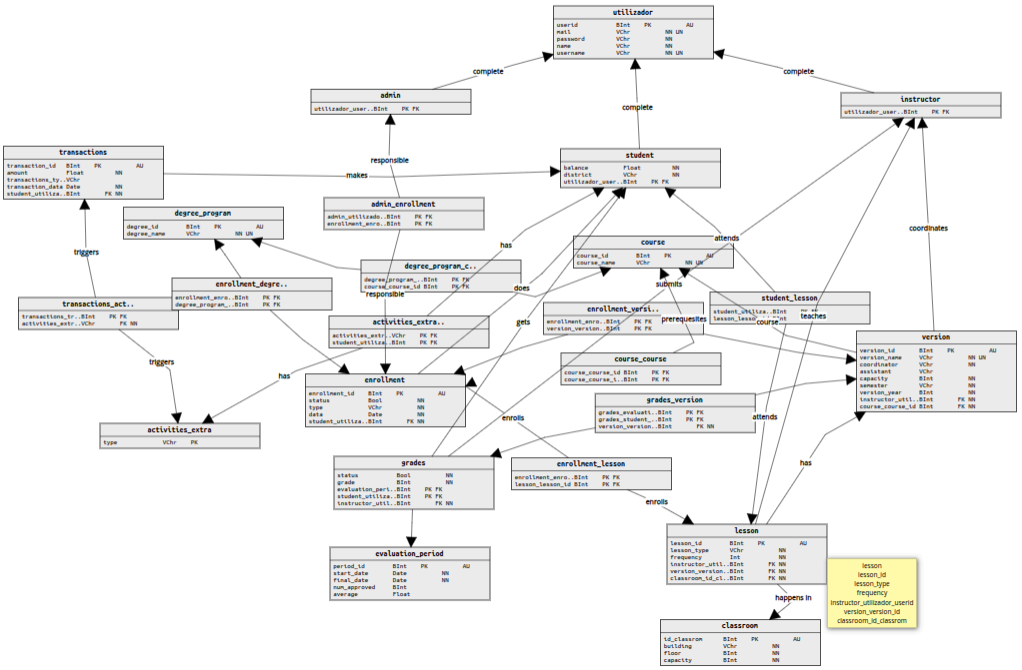


Diagrama Físico

O Diagrama Físico mostra a implementação concreta das tabelas no SGBD, incluindo nomes de colunas, tipos e constraints. Reflete o modelo anterior num esquema relacional concreto. Por exemplo, as chaves primárias (PK) estão marcadas, e as chaves estrangeiras (FK) indicam relações como *version_id* em Enrollment ou *student_id*, ligando cada matrícula ao seu aluno e à sua edição de curso.



Implementação

Autenticação JWT

A aplicação utiliza tokens JWT para autenticar utilizadores em cada pedido. No endpoint de login, após validar o *username* e a *palavra-passe* contra a base de dados, o servidor gera um token JWT contendo informações relevantes (por exemplo, o ID do utilizador e o seu papel – aluno, instrutor ou staff). Este token é assinado usando uma chave secreta definida na aplicação, garantindo que não pode ser falsificado. O token inclui ainda tempo de expiração para segurança. A cada acesso subsequente, o token é enviado no cabeçalho Authorization: Bearer .... No servidor, antes de processar o pedido, verifica-se a validade e assinatura do JWT. Se o token for inválido ou expirado, rejeita-se o pedido.

No código Python Flask, foi utilizada uma biblioteca para criar e decodificar os tokens. O token armazena uma *claim* com o ID do utilizador e o respetivo papel, o que permite ao servidor identificar quem está a pedir e que permissões tem. Por exemplo, numa função de rota protegida verifica-se jwt.decode(token, SECRET) para obter o payload. Esta estratégia assegura que não é necessário reenviar username/password a cada chamada, apenas um comprovativo assinado pelo servidor. Além dos endpoints pedidos, foi criado um endpoint extra par remover um aluno de uma atividade, o que gera um trigger que devolve o valor de sua matrícula na atividade.

Papéis e Autorização

O sistema distingue três papéis principais de utilizador: Staff (Admin), Student (Aluno) e Instructor (Docente). Cada operação só é permitida a utilizadores autorizados segundo o seu papel. No código fornecido, após descodificar o JWT, obtém-se o papel do utilizador e valida-se o acesso a cada endpoint. Por exemplo: apenas Admin pode criar novos utilizadores (/register) ou matricular alunos em cursos de grau (/enroll_degree). *Students* (autenticados) podem aceder a endpoints para se matricularem em atividades extras ou cursos disponíveis, consultar as suas próprias notas e informações pessoais. *Instructors* podem aceder a funcionalidades relacionadas às disciplinas que lecionam, como submeter notas (/submit_grades) para as turmas em que são coordenadores.

Esta separação de funções é implementada em nível de código. Por exemplo, cada rota verifica se o campo role no payload do JWT corresponde ao papel necessário, retornando erro caso contrário. Adicionalmente, implementou-se uma lógica de verificação: um instrutor que não seja coordenador daquela edição de curso não pode submeter notas. Assim, a aplicação impõe autorização estrita baseada em papéis e relações de dependência (como ligar o id do instrutor coordenador ao curso).

Concorrência e Transações

Para evitar problemas de concorrência, todas as operações de escrita críticas no banco de dados são executadas dentro de transações atómicas. A cada requisição que modifica o BD (INSERT, UPDATE, DELETE) usamos transações do PostgreSQL via psycopg. Por exemplo, ao matricular vários alunos concorrentemente em turmas limitadas, pode ocorrer conflito de vagas. Para mitigar isso, adotámos um nível de isolamento apropriado e usamos bloqueios de linha quando necessário.

No código Flask, isso é controlado por cursor/transações (ex.: with connection: e depois connection.commit() ou rollback() em caso de erro). Caso qualquer erro ocorra durante uma sequência de comandos, faz-se rollback para evitar estados inconsistentes. Esse cuidado assegura que transações simultâneas não deixem o sistema num estado incorreto. Além disso, estão definidas *foreign keys* e *check constraints* no BD (conforme descrito abaixo) que reforçam a integridade, de modo que qualquer tentativa de violar restrições seja rejeitada automaticamente pelo programa.

Também foram utilizados *triggers* e *funções armazenadas* no banco para automatizar tarefas e impor integridade extra. Por exemplo, um *trigger* pode atualizar automaticamente o saldo do estudante em Transactions ou verificar condições adicionais antes de inserir dados. Essas rotinas no lado do BD ajudam a manter o modelo lógico consistente e a evitar lógica duplicada na aplicação. Os principais triggers criados foram: **charge_student_on_enrollment, check_classrom_capacity_before_enrollment, update_evaluation_period_stats, charge_on_activity_signup e refund_on_activity_withdraw.** Os valores para se inscrever em um curso ou em uma atividade foram fixados em 100 e 50, respectivamente.

Views

Implementámos algumas visões (views) SQL para facilitar consultas complexas e reportes. Por exemplo, pode haver uma view que junte alunos e as suas matrículas em curso e disciplina (consolidação de Student, Enrollment e Course/Version) para rapidamente obter o histórico do aluno. Outra view pode resumir estatísticas por curso ou disciplina (média de notas, número de inscritos). Essas *views* ajudam a simplificar queries no código, permitindo que o Flask consulte uma única entidade ao invés de fazer JOIN complexos a cada pedido. As views também abstraem detalhes do esquema relacional, tornando a camada de aplicação mais limpa. As principais views criadas foram: **student_full_info, course_full_info e student_grades.**

Restrições (Constraints)

O modelo de dados contém várias restrições para garantir integridade. Todas as *chaves primárias* estão definidas (por exemplo, userID em *Utilizador*, course_id em *Course*, version_id em *Version*, etc.). Note-se que Utilizador.userID é *serial* (autoincrement) e é chave em tabelas como *Student*, *Admin* e *Instructor*.

Existem também *chaves estrangeiras* que ligam as tabelas. Por exemplo, *Student.userID* referencia *Utilizador.userID*, *Instructor.userID* referencia *Utilizador.userID*, garantindo que não existe estudante sem um correspondente utilizador geral. A tabela Enrollment possui campos como *student_id* (FK para *Student*) e *course_edition_id* (FK para *Version*), ligando cada matrícula ao aluno e à edição do curso. A Transactions inclui *student_id* (FK para *Student*) indicando qual aluno efetuou a transação.

Além disso, definimos restrições de *NOT NULL* em campos obrigatórios (por exemplo, mail, password e name de *Utilizador* não podem ficar vazios) e restrições de *UNIQUE* para atributos singulares. Em *Utilizador*, por exemplo, email e username são únicos, evitando duplicação de contas. Em *Course*, course_name é único. Em *Version*, version_name é único, além de estar ligado à disciplina.

Em resumo, as constraints de integridade no banco – PK, FK, UNIQUE, NOT NULL – são implementadas no esquema PostgreSQL de acordo com o modelo JSON. Elas garantem consistência automática: tentativas de inserir dados inválidos são rejeitadas, prevenindo erros antes mesmo do código da aplicação intervir.

Segurança

A segurança da aplicação foi considerada em múltiplos níveis. Todas as operações de base de dados no código Flask utilizam consultas parametrizadas, nunca concatenando diretamente strings do utilizador nas queries. Isso evita que inputs maliciosos corrompam os comandos SQL e protege contra SQL injection.

Na API, não expomos dados sensíveis nas respostas: por exemplo, ao obter informações de um utilizador via GET, nunca incluímos a palavra-passe nem o seu hash na resposta JSON. O token JWT gerado para autenticação contém apenas o username e o papel do utilizador, sem incluir a palavra-passe ou outros dados confidenciais.

Além disso, o acesso aos recursos é controlado por autenticação JWT e verificação de permissões, garantindo que apenas utilizadores legítimos e autenticados acedem às funcionalidades apropriadas.

Plano de Desenvolvimento

Tarefas e Cronograma

O desenvolvimento do projeto foi organizado em fases. O cronograma seguiu um processo incremental: primeiro modelámos e criamos o esquema do BD, depois implementámos o esqueleto do Flask e endpoints básicos (registro, login). Em seguida, expandimos as funcionalidades (matrículas, gestão de notas, etc.), testando cada recurso. Por fim, revisamos segurança, concorrência e concluímos a documentação e diagramas. Cada tarefa foi acompanhada de reuniões de equipa, controlo de versões e revisão de código.

O projeto foi feito em conjunto por todas, tendo-nos encontrado presencialmente sempre que possível para discutirmos ideias e resolver problemas. No entanto, por nos ser mais fácil, a maior parte do projeto foi sendo feito “à distância” e nós íamos comunicando online para nos entreajudar.

Conclusão

O desenvolvimento desta Plataforma de Gestão Académica permitiu-nos consolidar várias competências importantes. Aprendemos a projetar um modelo de dados complexo (apoiando-nos em ferramentas como o ONDA) e a traduzir esse modelo num esquema relacional implementado no PostgreSQL. No backend, aprofundámos conhecimentos em Python Flask e na criação de APIs RESTful seguras, usando JWT para autenticação. Foi desafiador garantir que cada papel de utilizador tinha exatamente as permissões definidas pelo enunciado, além de lidar com a concorrência e integridade transacional.

A experiência prática mostrou a importância de definir corretamente *constraints*, usar transações adequadamente e criar testes abrangentes (via Postman) para validar a aplicação. Além disso, a preparação deste relatório desenvolveu a nossa capacidade de documentar processos técnicos de forma clara e formal.

Em suma, alcançámos os objetivos do projeto: implementámos as funcionalidades solicitadas (registo de utilizadores, matrículas, gestão de notas, etc.) e cumprimos os requisitos não-funcionais de segurança e desempenho. A aprendizagem incluiu desde a conceção do modelo de dados até à codificação de endpoints e à gestão de uma aplicação full-stack.