UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Pedro Eduardo Concon Silva

Modernização tecnológica de legados back-end: uma proposta de migração do sistema Classroom eXperience

Uberlândia, Brasil

2025

UNIVERSIDADE FEDERAL DE UBERLÂNDIA

Pedro Eduardo Concon Silva

Modernização tecnológica de legados back-end: uma proposta de migração do sistema Classroom eXperience

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Orientador: Rafael Dias Araújo

Universidade Federal de Uberlândia – UFU

Faculdade de Computação

Bacharelado em Sistemas de Informação

Uberlândia, Brasil

2025

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

S586 Silva, Pedro Eduardo Concon, 1999-
2025 Modernização tecnológica de legados back-end: uma proposta de migração do sistema Classroom eXperience [recurso eletrônico] : Modernização tecnológica de legados back-end / Pedro Eduardo Concon Silva. - 2025.

Orientador: Rafael Dias Araújo.
Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Uberlândia, Graduação em Sistemas de Informação.
Modo de acesso: Internet.
Inclui bibliografia.

1. Tecnologia da informação - Administração. I. Araújo, Rafael Dias,1986-, (Orient.). II. Universidade Federal de Uberlândia. Graduação em Sistemas de Informação. III. Título.

CDU: 658:681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:
Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

Pedro Eduardo Concon Silva

Modernização tecnológica de legados back-end: uma proposta de migração do sistema Classroom eXperience

Trabalho de conclusão de curso apresentado à Faculdade de Computação da Universidade Federal de Uberlândia, como parte dos requisitos exigidos para a obtenção título de Bacharel em Sistemas de Informação.

Trabalho aprovado. Uberlândia, Brasil, 05 de dezembro de 2025:

Rafael Dias Araújo (FACOM/UFU)
Orientador

Prof. Dr. Fabiano Azevedo Dorça (FACOM/UFU)

Prof. Dra. Maria Adriana Vidigal de Lima (FACOM/UFU)

Uberlândia, Brasil
2025

Dedico este trabalho à minha família, aos meus pais Maria Luiza e Rodrigo, e aos meus avós Terezinha e Valentin, que se dedicaram imensamente para que eu pudesse ter uma formação de qualidade e conquistar meus objetivos, sempre com muito apoio e carinho.

Agradecimentos

Agradeço, em primeiro lugar, aos meus pais, Maria Luiza e Rodrigo, por me proporcionarem uma base sólida de educação, valores e respeito. Sem o apoio, os ensinamentos e o amor de vocês, não seria possível chegar até aqui. Tudo o que sou carrego da base que vocês construíram com tanto trabalho, carinho e dedicação. Estendo minha gratidão aos meus irmãos, Júlia, Danilo e Lorenzo, pelo companheirismo constante, pelas risadas compartilhadas e por todo o carinho ao longo dessa jornada, vocês são um orgulho pra mim.

Agradeço à minha noiva, Gabriela, por estar ao meu lado em todos os momentos, especialmente nos mais difíceis. Com seu apoio, incentivo constante e torcida, a caminhada rumo aos meus objetivos tornou-se muito mais leve. Sou imensamente grato pelo seu amor, companheirismo e paciência, que me fortaleceram ao longo dessa jornada. Com você, cada conquista tem um sabor ainda mais especial, e cada passo adiante é sempre ao seu lado.

Aos meus avós, Valentin e Terezinha, agradeço por todos os ensinamentos e pelas lindas memórias de infância que guardo com tanto carinho. Com amor, exemplo e sabedoria, vocês me mostraram o valor da dedicação, da humildade e da busca constante por ser alguém melhor. Estendo minha gratidão aos meus tios, Gustavo, Eduardo, Milena, Juliana e Camila, pelo companheirismo, ajuda e pelas palavras de incentivo ao longo da caminhada.

Aos meus saudosos tios, Otávio e Lúcia, minha eterna gratidão por tudo o que fizeram por mim. Guardo com muito carinho as lembranças da infância, os ensinamentos transmitidos com afeto e os valores que me passaram com tanta simplicidade e por toda a generosidade. Sou muito grato por cada momento vivido ao lado de vocês, suas memórias continuam vivas em mim e me inspiram a seguir em frente com coragem e coração leve.

Agradeço aos meus amigos da UFU Luis Felipe, Luis Otávio, João Vitor, Cairo, Antônio, Carlos, Henrique, Arthur, Caio, Breno e Murilo, por todo o companheirismo e pela leveza que trouxeram ao longo dessa caminhada. Seja dividindo um copo de cerveja ou encarando juntos uma matéria que parecia impossível, cada momento ao lado de vocês tornou essa trajetória mais leve, divertida e memorável. Minha profunda gratidão a todos.

Aos meus amigos de longa data, Gabriel Soares, Gabriel Torres, Guilherme Silva, Cassiel, Matheus de Camargo, João Pedro, Antônio, Eduardo Ribeiro, Paulo Vitor, Iago, Gabriel Wanzeler, Caio Ferreira, Luiz Augusto e Ana Julia. Obrigado pelo apoio constante ao longo da jornada, vocês são parte essencial da minha história, e a cada um de vocês sou profundamente grato pela amizade, pela lealdade e por fazerem parte da minha vida de forma tão especial.

Resumo

Este trabalho apresenta o desenvolvimento de um sistema Web com ênfase na modernização de aplicações legadas e na aplicação de boas práticas de arquitetura back-end. O principal objetivo foi construir uma aplicação robusta e escalável utilizando tecnologias consolidadas como Java, Spring Boot, PostgreSQL e Docker. O processo envolveu desde o entendimento do sistema legado, passando pela modelagem da arquitetura da aplicação, até a implementação de funcionalidades como autenticação com JSON Web Token (JWT), testes unitários e de integração, e a configuração de um pipeline de Integração Contínua (CI) via GitLab. Como resultado, obteve-se uma aplicação funcional, bem documentada e com cobertura de testes automatizados, validada por meio da execução contínua em ambiente de CI. Conclui-se que a escolha das tecnologias e a abordagem arquitetural adotadas contribuíram diretamente para a qualidade, segurança e manutenibilidade do sistema.

Palavras-chave: Desenvolvimento Web, Java, Spring Boot, Docker, Testes Automatizados, Integração Contínua.

Lista de ilustrações

Lista de Abreviaturas e Siglas

ACID Atomicity, Consistency, Isolation, Durability

API Application Programming Interface

C&A Captura e Acesso

CX Classroom eXperience

DER Diagrama Entidade-Relacionamento

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

ILS Intelligent Learning System

JDBC Java Database Connectivity

JPA Java Persistence API

JSP Java Server Pages

JWT JSON Web Token

ORM Object-Relational Mapping

POO Programação Orientada a Objetos

REST Representational State Transfer

SGBD Sistema Gerenciador de Banco de Dados

SQL Structured Query Language

WWW World Wide Web

YAML YAML Ain't Markup Language

Sumário

1 Introdução

Os sistemas de ambientes educacionais são uma realidade no mundo moderno, onde o conhecimento é disseminado com muita praticidade, organização e flexibilidade para o estudante. Sendo assim, é necessário cada vez mais a criação de plataformas robustas que suportem funcionalidades de multimídia, arquivos e personalização para cada tipo de aluno, de acordo com preferências e dificuldades.

A versatilidade do ensino remoto gerou mais oportunidade para as pessoas estudarem, aumentando muito a procura por ferramentas de aprendizado que proporcionem um estudo de qualidade, com isso esses modelos tornaram-se imprescindíveis para a sociedade. Além disso, existem melhorias evidentes que, em alguns casos, a qualidade para os estudantes que possuem acesso a essas plataformas pode apresentar um aumento de desempenho de até 50% para aqueles que possuem notas ruins ou estão insatisfeitos (LIU; LOMOVTSEVA; KOROBEYNIKOVA, 2020).

No entanto, a utilização do ensino remoto indica algumas desvantagens. Em alguns modelos de estudo, os alunos sentem falta das interações da sala de aula com o professor e os próprios colegas. Outro aspecto negativo é manter a motivação, uma vez que um estudante sem autorregulação e disciplina tem tendência a não atribuir tempo suficiente para as tarefas, e isso acontece devido à independência que o aprendizado online proporciona (RAWASHDEH et al., 2021).

A plataforma Classroom eXperience tem como objetivo a Captura e Acesso (C&A) de conteúdos para registrar as informações das atividades feitas pelo professor durante a aula, fornecendo aos alunos conteúdos de forma simples e objetiva. A meta do trabalho proposto é modernizar esse sistema *Web* aplicando melhorias no desenvolvimento *Back-End*, arquiteturais e principalmente atualizando a tecnologia do projeto, sendo fundamental sua migração do Java Server Pages (JSP).

No projeto será utilizada a linguagem Java com o *framework Spring Boot*, construindo *API's (Application Programming Interface)* para comunicação entre os componentes do sistema, aplicando uma arquitetura de microsserviços. Será utilizado o banco de dados *PostgreSQL* para gerenciamento e manipulação dos dados. Além disso, será feita toda a parte de conteinerização do projeto utilizando o *docker*.

1.1 Objetivos

O objetivo geral deste trabalho é realizar a modernização tecnológica e a reestruturação do *backend* da plataforma Classroom eXperience (CX), com foco na adoção de boas

práticas de engenharia de software, maior qualidade estrutural e aderência às exigências técnicas contemporâneas. A proposta consiste em transformar uma aplicação legada, monolítica e fortemente acoplada em uma solução moderna, modular e de fácil manutenção, utilizando ferramentas amplamente reconhecidas na indústria, como contêineres *Docker*, integração contínua com *GitLab CI*, testes automatizados e documentação interativa por meio do *Swagger*.

Para atingir esse objetivo geral, foram definidos os seguintes objetivos específicos:

- Identificar as principais limitações técnicas do sistema legado e propor uma nova arquitetura baseada em princípios de modularização, baixo acoplamento e alta coesão;
- Implementar os principais componentes da aplicação utilizando o *framework Spring Boot*, com foco em uma arquitetura em camadas.
- Aumentar a confiabilidade do sistema por meio de testes automatizados e práticas de integração contínua com GitLab CI;
- Viabilizar a implantação reprodutível do sistema utilizando contêineres Docker e documentar a Application Programming Interface (API) com Swagger/OpenAPI;
- Registrar as decisões técnicas tomadas ao longo do projeto, facilitando a manutenção e a evolução futura da plataforma.

1.2 Organização do texto

O trabalho está distribuído em cinco capítulos, permitindo ao leitor compreender desde os fundamentos teóricos até a implementação prática e os resultados alcançados. No Capítulo 1 (Introdução), são apresentados o contexto do projeto, seus objetivos gerais e específicos, além de uma visão geral da estrutura do texto. Esta seção tem como propósito situar o leitor sobre a motivação do trabalho e os caminhos metodológicos adotados ao longo do desenvolvimento da solução proposta.

O Capítulo 2 (Fundamentação Teórica) contempla os conceitos essenciais que embasam a execução do projeto. São abordados temas como a computação ubíqua e o conceito de Captura e Acesso C&A, sistemas web, fundamentos do desenvolvimento *back-end*, processos de migração de sistemas legados, princípios de arquitetura de software e a estruturação de *APIs RESTful*. Esses tópicos sustentam as escolhas técnicas realizadas durante a reestruturação da plataforma, fornecendo o embasamento necessário para sua modernização.

No Capítulo 3 (Trabalhos Relacionados), são exploradas publicações acadêmicas e estudos técnicos que tratam da modernização de sistemas, refatoração arquitetural,

evolução de sistemas legados e práticas modernas de engenharia de software. Os trabalhos citados servem de referência para decisões tomadas no projeto e reforçam a relevância da proposta no contexto atual de transformação digital e evolução de aplicações web.

O Capítulo 4 (Desenvolvimento) descreve de forma detalhada o processo prático de execução do projeto. Inicialmente, são apresentados o entendimento do sistema legado e a análise dos requisitos funcionais e não funcionais. Em seguida, é realizada a modelagem dos dados e definida a arquitetura da nova aplicação. Também são discutidas as escolhas tecnológicas como uso de Java, Spring Boot, PostgreSQL e Docker, além da organização dos testes e da documentação. Essa seção evidencia a transição do sistema legado para uma arquitetura moderna, escalável e de fácil manutenção. Por fim, o Capítulo 5 (Conclusão) resume os resultados obtidos, destaca as dificuldades enfrentadas, apresenta considerações finais e propõe caminhos para a continuidade e evolução do projeto.

2 Fundamentação Teórica

2.1 Captura e Acesso

A área de Captura e Acesso C&A faz parte da computação ubíqua ou pervasiva, que tem como objetivo auxiliar as pessoas nas tarefas cotidianas por meio da coleta de informações importantes. A C&A tem o propósito de resgatar esses dados, principalmente de multimídia, para serem disponibilizados posteriormente aos usuários, visando uma melhora na experiência e proporcionando flexibilidade no acesso (RIBEIRO et al., 2014).

Neste projeto a captura dos dados será modernizada, melhorando o acesso dos usuários às informações da plataforma CX. Além disso, também será validado caso necessite da adição de novos dados para disponibilização, aperfeiçoando os benefícios de personalização do conteúdo educacional, recomendações e restrições de mídia por usuário.

2.2 Sistemas Web

Durante a história da *Web* o seu início foi caracterizado por *hyperlinks* e páginas estáticas, a primeira versão do World Wide Web (WWW) tinha como finalidade a interligação de documentos de hipermídia na internet. Berners-Lee desenvolveu esse sistema permitindo que fossem compartilhados esses documentos eficientemente por meio de uma rede de computadores, surgindo assim a *Web 1.0* (BERNERS-LEE et al., 2023).

Com a progressão da internet e o acesso expandido das pessoas, gradualmente surgiram aprimoramentos nas páginas e documentos online. Dessa forma, em 2004 a O'Reilly Media utilizou pela primeira vez o termo *Web 2.0* em que as páginas poderiam ter conteúdos dinâmicos e interativos entre os usuários, melhorando a usabilidade e, principalmente, tornando a internet como uma plataforma de desenvolvimento de aplicações (O'REILLY, 2009).

A ideia da terceira geração, conhecida como *Web 3.0*, visa utilizar de forma mais inteligente os dados, definindo automações de atividades cotidianas e apoiando a acessibilidade da internet móvel. Sendo assim, ela é conhecida como a *Web* Semântica ou *Web* Inteligente, de modo que, de acordo com a estrutura das informações, é possível reutilizar e compartilhar dados em aplicativos, empresas e comunidades (CHOUDHURY, 2014).

Por fim, a *Web 4.0* conhecida como *Web* Obíqua ou *Web* Simbíotica surge para garantir transparência, governança dos dados e distribuição global dos sistemas (CHOUDHURY, 2014). Trazendo evoluções no setor de telecomunicações, avançando em nanotecnologia

e aumentando a interação entre humanos e máquinas, atingindo um nível, por exemplo, de inteligência em ler o conteúdo da Web, permitindo uma experiência mais intuitiva e imersiva (CHOUDHURY, 2014).

Além dos avanços na geração da *Web*, é importante ressaltar que sua arquitetura é por natureza distribuída e baseada no modelo cliente-servidor. Desse modo, significa que os clientes são os dispositivos de acesso as páginas através dos navegadores, já a responsabilidade dos servidores é hospedar, disponibilizar e responder os clientes fornecendo todos os dados necessários para a exibição da página.

2.3 Fundamentos do desenvolvimento Back-end

O desenvolvimento *Back-End* é fundamental para a implementação de regras de negócios, mecanismos de segurança, manipulação e gerenciamento de dados no lado do servidor (SANTOS, 2023). Sendo assim, o *Back-End* é a parte invisível do sistema, que proporciona a interação dos dados, permitindo que o usuário consiga salvar, manipular e consultar informações.

Existem diversas opções de linguagens e *frameworks* para desenvolver, em tese deve-se avaliar fatores de escalabilidade, desempenho, expertise da equipe desenvolvedora e suporte da comunidade. Um dos passos iniciais para a escolha é comparar os recursos das linguagens, os fatores que são levados em consideração são a sintaxe, tipos de dados, gerenciamento de memória, simultaneidade, bibliotecas e compatibilidade (ZAPALOWSKI, 2011).

A programação *Back-End* pode ser implementada com diferentes paradigmas, por exemplo, Programação Orientada a Objetos (POO) ou funcional. Quando se utiliza POO, a aplicação é construída em torno de classes de objetos com propriedades de dados e métodos que podem ser invocados por outros objetos, definindo as interações entre as classes (ZAPALOWSKI, 2011). Já a programação funcional é um paradigma concentrado em expressões e funções matemáticas que tem como base o uso de cálculo lambda (CHURCH, 1985), indicada para *inputs* e *outputs* simples, sem mudanças de estado (THOMPSON, 2011).

Neste projeto será utilizado a linguagem Java e o framework Spring Boot. Essa escolha está atrelada aos benefícios da POO para criar um código modular, reutilizável e de fácil manutenção. Além disso, a tecnologia JSP utiliza o Java, colaborando na migração para atingir o objetivo do projeto.

2.4 Migração de Sistemas Legados

A migração de sistemas legados consiste na transição de aplicações antigas para novas plataformas tecnológicas, com o objetivo de melhorar desempenho, escalabilidade e manutenção. Esse processo pode ser realizado de diferentes formas, como reescrita total, encapsulamento, reengenharia ou substituição gradual. No contexto corporativo, a decisão de migrar um sistema geralmente ocorre devido a problemas críticos, seja pela obsolescência da tecnologia utilizada ou por falhas nas regras de negócio, comprometendo a entrega das funcionalidades esperadas pela aplicação.

A migração deve ocorrer de forma incremental (BRODIE; STONEBRAKER, 1995), reduzindo riscos e custos em comparação com a reescrita total do sistema. O uso de *gateways*, interfaces ou *middleware* para converter requisições HyperText Transfer Protocol (HTTP) ou chamadas Structured Query Language (SQL) direcionadas a bancos de dados obsoletos é altamente recomendado, pois permite preservar certas interações do sistema legado de forma paliativa que seriam complexas de migrar inicialmente.

No contexto deste projeto, a migração não exige a preservação de interações críticas com regras de negócio altamente complexas. O principal desafio está na obsolescência tecnológica, pois a aplicação atual foi desenvolvida em JSP, um modelo ultrapassado para a construção de aplicações web dinâmicas devido ao forte acoplamento entre a camada de apresentação e a lógica de negócios, dificultando a manutenção e escalabilidade. Com o avanço de frameworks modernos, como Spring Boot e API Representational State Transfer (REST), tecnologias mais eficientes e modulares substituíram o JSP. Neste projeto, adotamos a abordagem de reengenharia, refatorando o código existente para uma arquitetura mais moderna e sustentável (FOWLER, 2012).

2.5 Arquitetura de Software

A arquitetura de software é um pilar extremamente importante para garantir escalabilidade, sustentabilidade e desempenho. Existem diversos modelos, sendo os mais utilizados a arquitetura de microsserviços, monolítica e hexagonal (BASS, 2012).

A arquitetura de microsserviços, ilustrada na Figura 1, prevê a segregação da aplicação em pequenos serviços independentes, ou seja, cada um com sua própria instância, definido por uma responsabilidade específica e com sua própria base de dados. Esse modelo oferece excelentes benefícios em termos de escalabilidade, independência de desenvolvimento e implantação. No entanto, a utilização dessa arquitetura introduz complexidades que podem ser desnecessárias para alguns tipos de projetos, uma vez que adiciona complicações na comunicação entre serviços, orquestração, problemas de monitoramento, custos de infraestrutura e desafios de depuração (NEWMAN, 2021).

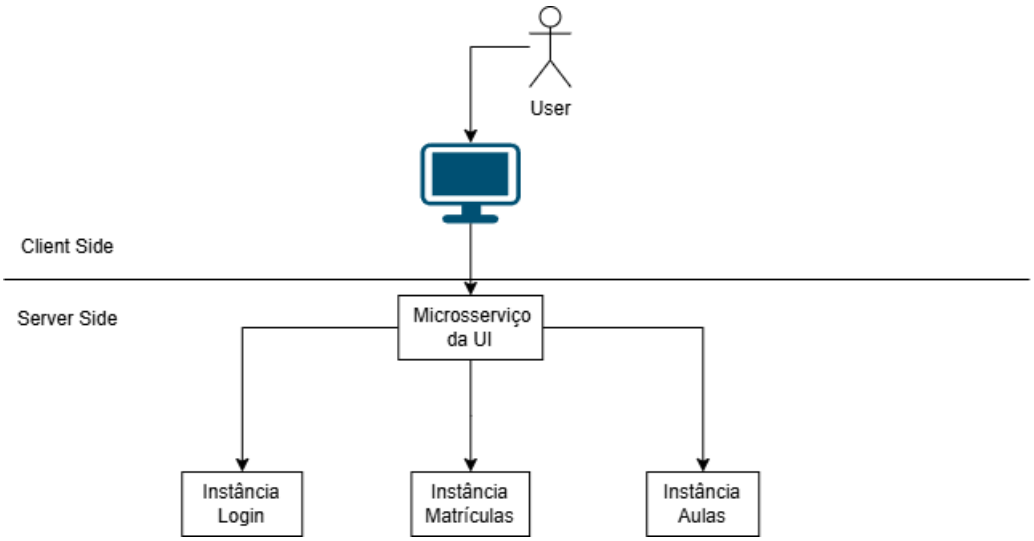

Figura 1 – Arquitetura de Microsserviços

Fonte: Próprio autor.

A arquitetura monolítica, ilustrada na Figura 2, apresenta características de uma aplicação única e coesa, ou seja, com apenas uma instância, em que as funcionalidades compartilham o mesmo código-base e banco de dados, facilitando o desenvolvimento inicial, a implantação e o gerenciamento em pequenos sistemas. No entanto, com o aumento da quantidade de usuários e o crescimento do projeto, tornam-se mais evidentes as dificuldades de escalabilidade, os problemas de concorrência nas modificações entre os desenvolvedores, entre outros desafios (LAURETIS, 2019).

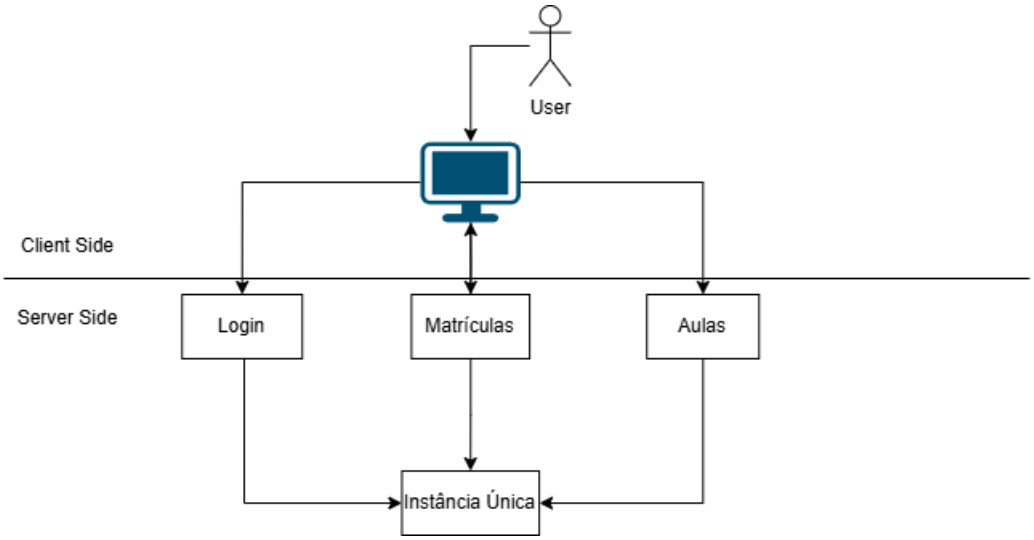

Figura 2 – Arquitetura Monolítica

Fonte: Próprio autor.

Diante disso, neste projeto foi definida a criação de uma única API, adotando uma abordagem mais próxima da arquitetura monolítica. Essa decisão se justifica pelo fato de o sistema utilizar uma única base de dados, possuir um número reduzido de *endpoints* e não apresentar regras de negócio complexas. As arquiteturas possuem *trade-offs*, cada uma com seus pontos positivos e negativos. Cabe ao desenvolvedor reconhecer o cenário mais adequado para cada contexto.

2.6 API REST

API é uma interface que permite a comunicação entre diferentes sistemas ou aplicações com o back-end de forma padronizada, independentemente da tecnologia utilizada no lado do cliente. Trata-se de um conjunto de definições, rotinas e protocolos que facilitam a troca de informações e a execução de funcionalidades específicas de maneira segura e eficiente (MASSE, 2011).

Nos anos 2000, (FIELDING, 2000) definiu um estilo arquitetural para Web chamado REST, esse modelo utiliza o protocolo HTTP como base para a comunicação entre cliente e servidor. Na dissertação, (FIELDING, 2000) propôs o REST como uma alternativa mais escalável e padronizada para a construção de aplicações distribuídas, destacando seis restrições arquiteturais fundamentais: cliente-servidor, ausência de estado (stateless), cache, interface uniforme, sistema em camadas e código sob demanda (opcional). Essas restrições visam melhorar a escalabilidade, a simplicidade e a independência dos componentes do sistema.

Sendo assim, optou-se pelo desenvolvimento de uma única API REST centralizada utilizando *Spring Boot*, responsável por prover todos os endpoints da aplicação. Essa abordagem simplifica a organização do código, facilita a manutenção e reduz a complexidade arquitetural. Como o sistema não possui um número elevado de funcionalidades nem apresenta regras de negócio altamente especializadas, a separação em múltiplos microsserviços foi considerada desnecessária neste contexto (RICHARDSON, 2018).

Para a segurança da API será utilizado no projeto o Spring Security, que é um dos módulos do ecossistema *Spring* que oferece uma estrutura robusta para autenticação e autorização em aplicações Java. Ele permite a configuração de múltiplos métodos de autenticação, como *login* baseado em formulários, autenticação via *tokens JWT* e integração com provedores *OAuth2*. No contexto do sistema CX, o *Spring Security* é fundamental para garantir que apenas usuários autenticados possam acessar os recursos protegidos da API. Adotar esse módulo aumenta significativamente a segurança da aplicação, atendendo a requisitos essenciais de confidencialidade e integridade dos dados dos usuários (SPILCA, 2020).

Para manipulação de dados em bancos relacionais, será utilizado o Spring Data

Java Persistence API (JPA), que é uma biblioteca que simplifica o acesso por meio da abstração da camada de persistência. Ele reduz a necessidade de escrever código repetitivo ao permitir a criação de repositórios com base em interfaces, utilizando métodos que seguem convenções de nomenclatura padronizadas. No projeto CX, essa abordagem garante maior produtividade no desenvolvimento, facilita testes e promove um código mais limpo e coeso.

Para a camada de testes unitários e de integração da API, adotou-se o *JUnit 5* devido à sua extensibilidade e ao conjunto rico de anotações que facilitam a estruturação de cenários de teste claros e expressivos (TUDOSE, 2020). Para garantir o isolamento de dependências externas como repositórios, clientes HTTP e componentes de segurança recorreu-se ao Mockito, cuja API possibilita a criação de representações comportamentais. Essa infraestrutura de testes permite validar com precisão o comportamento dos serviços em diferentes condições, verificar a geração de tokens JWT e assegurar que os endpoints REST respondam conforme o esperado JWT.

Portanto, manter uma única API monolítica segue o princípio da simplicidade e da eficiência, alinhando-se à real necessidade do projeto. Explorando o *framework Spring Boot*, que oferece suporte nativo à construção de *APIs RESTful*, além de integração com ferramentas como Swagger para documentação, Spring Security para autenticação, testes unitários e Spring Data para acesso a dados. Essa escolha tecnológica proporciona agilidade no desenvolvimento e garante uma estrutura robusta para atender às demandas do sistema atual, sem a necessidade de fragmentação em múltiplos serviços.

3 Trabalhos Relacionados

A computação Ubíqua, conforme tratada no trabalho de Ferreira et al. (2012), refere-se a um paradigma no qual a tecnologia se integra de forma invisível no ambiente ao nosso redor, integrando-se nas nossas atividades cotidianas. No sistema Classroom eXperience, a infraestrutura de captura e acesso é projetada para se adaptar ao ambiente educacional de forma não intrusiva. Isso significa que o sistema é capaz de capturar automaticamente artefatos produzidos durante as aulas, como anotações, apresentações e interações aluno-professor. Esses dados capturados são então disponibilizados aos alunos de forma simples e objetiva, oferecendo uma experiência de aprendizado mais dinâmica e interativa com o intuito de aumentar o desempenho dos alunos nas aulas.

A importância do CX é indiscutível no contexto educacional moderno. Conforme estudado no trabalho por Ribeiro et al. (2014), o sistema Web CX oferece uma experiência de aprendizado dinâmica e flexível, permitindo que os alunos acessem e revisem os conteúdos em momentos convenientes para eles. Dessa forma, os alunos têm a liberdade de estudar o material em qualquer momento viável, sem se preocupar com o detalhamento das anotações durante as aulas. Isso não apenas facilita o processo de aprendizagem, mas também permite mais concentração no conteúdo apresentado, contribuindo assim para o aumento do desempenho acadêmico. Além disso, o CX encoraja a participação e a colaboração entre os alunos, criando um ambiente de aprendizado mais engajador e estimulante.

A modernização de um sistema legado web é um desafio para os desenvolvedores. Inicialmente, muitos programadores enfrentam dificuldades relacionadas à obsolescência da tecnologia. No entanto, existem outros obstáculos, como regras de negócio internas e questões de configuração de ambiente. O trabalho realizado por Raksi et al. (2017), publicado pela Aalto University na Finlândia, visa abordar esses impasses e trazer estratégias para uma migração moderna do sistema legado, garantindo uma transição tranquila para os usuários atuais. Ao aplicar métodos atualizados de desenvolvimento e lidar com os desafios técnicos e organizacionais associados à modernização, os desenvolvedores podem garantir que o sistema CX continue a desempenhar um papel fundamental no avanço da educação digital.

O estudo conduzido por Gnoyke, Schulze e Krüger (2024) apresenta uma análise profunda da evolução das arquiteturas em sistemas e analisando os seus problemas, *architecture smells*, destacando a importância de identificar e mitigar problemas recorrentes na arquitetura de software ao longo do tempo. Inspirado por essas descobertas, o desenvolvimento da nova arquitetura do sistema CX buscou aplicar boas práticas de engenharia

de software para evitar a formação de dependências cíclicas e instabilidades estruturais. A arquitetura proposta foi desenhada com foco na modularização, uso adequado de camadas e princípios de baixo acoplamento, de modo a prevenir a degradação arquitetural observada em sistemas analisados no estudo. Assim, além de melhorar a manutenibilidade e escalabilidade do sistema, essas decisões arquiteturais contribuem diretamente para a sustentabilidade do projeto a longo prazo.

O trabalho de Durelli (2016) propõe uma abordagem sistemática para a criação, reúso e aplicação de refatorações com foco na modernização arquitetural de sistemas legados. A abordagem descrita no artigo fornece mecanismos para identificar *bad smells* em arquiteturais de sistemas legados e aplicar refatorações reutilizáveis que melhoram a qualidade estrutural do software. Ao adotar conceitos dessa metodologia, foi possível conduzir a modernização do sistema de forma mais segura e orientada, mantendo a integridade das funcionalidades existentes e promovendo uma evolução arquitetural sustentável.

O artigo de Leon e Horita (2020) discute a importância da modernização de arquiteturas de sistemas como um pilar fundamental para viabilizar a transformação digital nas organizações. Entre os principais pontos abordados estão a adoção de arquiteturas baseadas em microsserviços, a virtualização, o uso de contêineres e a automação de processos de integração (CI). Esses elementos estão diretamente alinhados com as decisões tomadas no desenvolvimento do sistema CX, que incorporou práticas modernas como a utilização de Docker para isolamento e portabilidade, configuração de pipelines de CI no GitLab para garantir qualidade e agilidade no ciclo de desenvolvimento. Dessa forma, o projeto não apenas atualiza uma solução existente, mas também a posiciona estrategicamente dentro de um cenário de transformação digital, preparando a aplicação para escalabilidade, resiliência e evolução contínua.

Diante dos trabalhos analisados, é possível perceber que a modernização de sistemas legados, especialmente no contexto educacional, demanda uma abordagem multidimensional que abrange desde a infraestrutura ubíqua até a aplicação de boas práticas de arquitetura de software. A identificação e mitigação de *architecture smells*, bem como a aplicação sistemática de refatorações, foram fundamentais para garantir a qualidade estrutural da aplicação. Dessa forma, os trabalhos relacionados não apenas embasaram teoricamente o desenvolvimento, como também nortearam decisões práticas cruciais para o sucesso da solução proposta.

4 Desenvolvimento

Este capítulo descreve o processo de desenvolvimento do novo sistema, abordando desde a análise inicial até as etapas finais de testes e validação. A modernização da aplicação teve início com a compreensão do funcionamento do sistema legado, seguida pela identificação das funcionalidades essenciais que precisavam ser migradas. A partir disso, foi possível levantar os requisitos do novo sistema, definindo claramente os objetivos da migração.

Com os requisitos em mãos, foi realizada a escolha das tecnologias mais adequadas para a construção de uma API moderna, segura e escalável. A modelagem de dados foi elaborada considerando as necessidades atuais do sistema e a estrutura existente no banco de dados legado.

Além disso, foram incluídas práticas de documentação e observabilidade para garantir maior transparência, rastreabilidade e facilidade de manutenção do sistema. Por fim, foram realizados testes e coletas de métricas com o objetivo de avaliar o desempenho, a confiabilidade e a qualidade da solução desenvolvida.

4.1 Entendimento do sistema legado

A partir da interface do sistema CX antigo[1], foi realizada uma análise detalhada do funcionamento da aplicação baseada em JSP, observando como as rotas são invocadas, como ocorre a transação de informações entre as páginas e o servidor, e quais funcionalidades estão presentes na interface.

Durante esse processo, foi possível identificar as principais páginas utilizadas, como login, cadastro e visualização de dados, bem como mapear o fluxo de navegação e os pontos de integração com o banco de dados. Essa etapa foi fundamental para compreender as dependências existentes no sistema legado e garantir que as funcionalidades essenciais fossem corretamente migradas e preservadas na nova solução.

No início, para entender como as requisições HTTP são enviadas, foi analisada a tela de login do sistema, mostrado na Figura 3. No modelo tradicional com JSP e Servlets, quase todo o processamento é feito no servidor. Em tempo de execução, o servidor de aplicações Web (Apache Tomcat) compila os arquivos .jsp em servlets Java. Esses servlets são responsáveis por processar as requisições HTTP, executar lógica de negócio e devolver uma resposta HyperText Markup Language (HTML) ao cliente (navegador).

[1] http://cx.facom.ufu.br/

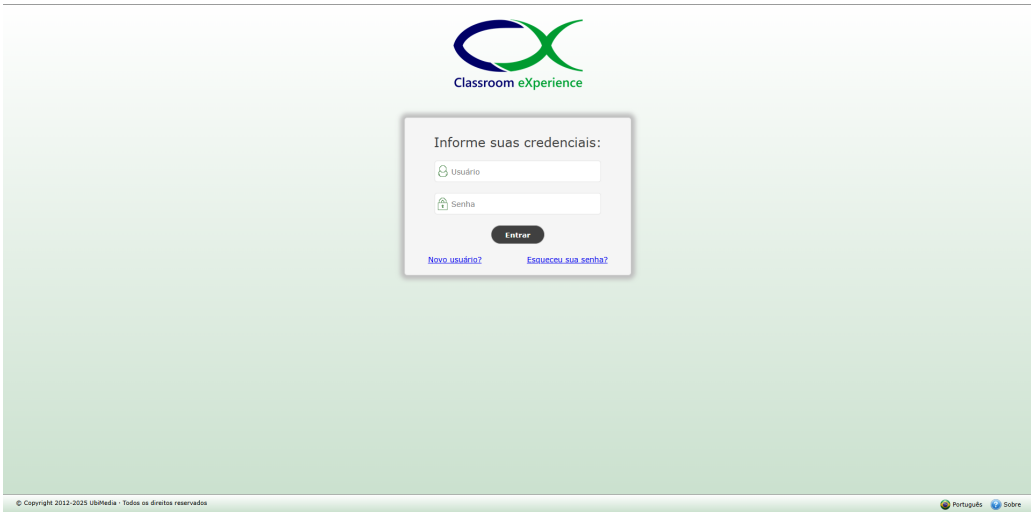

Figura 3 – Login CX antigo.

O usuário, ao preencher a página de login e clicar em "Entrar", aciona um formulário que realiza uma requisição do tipo POST para o servidor, enviando as informações digitadas, como nome de usuário e senha. O servidor, então, processa essa requisição, realiza as devidas validações e retorna a resposta ao cliente. Nesse fluxo, o endpoint utilizado é o endereço http://cx.facom.ufu.br/CxController, que corresponde a um servlet responsável pelo tratamento dessa lógica, conforme ilustrado pela Figura 4.

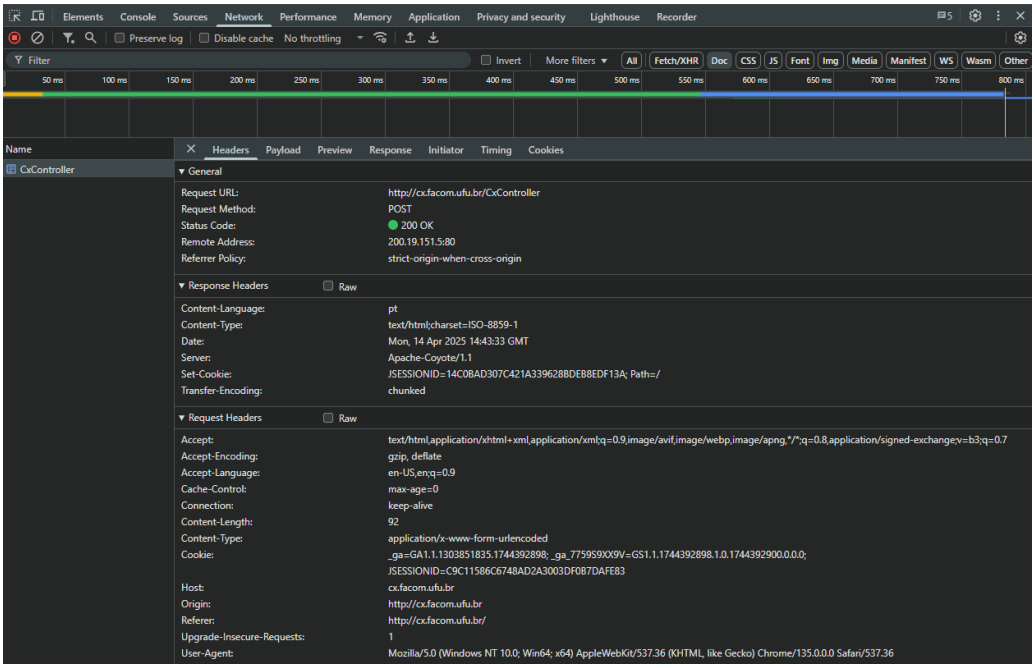

Figura 4 – Endpoint para efetuar o login.

O *payload* enviado na requisição define diretamente qual lógica de negócio será executada no sistema. Em aplicações JSP, é comum a utilização do parâmetro "cmd" para

esse fim. A partir desse valor, o servidor identifica qual funcionalidade deve ser processada. No caso apresentado, o valor "loginUsuario" indica que se trata de uma requisição POST relacionada ao processo de autenticação do usuário, conforme ilustrado pela Figura 5.

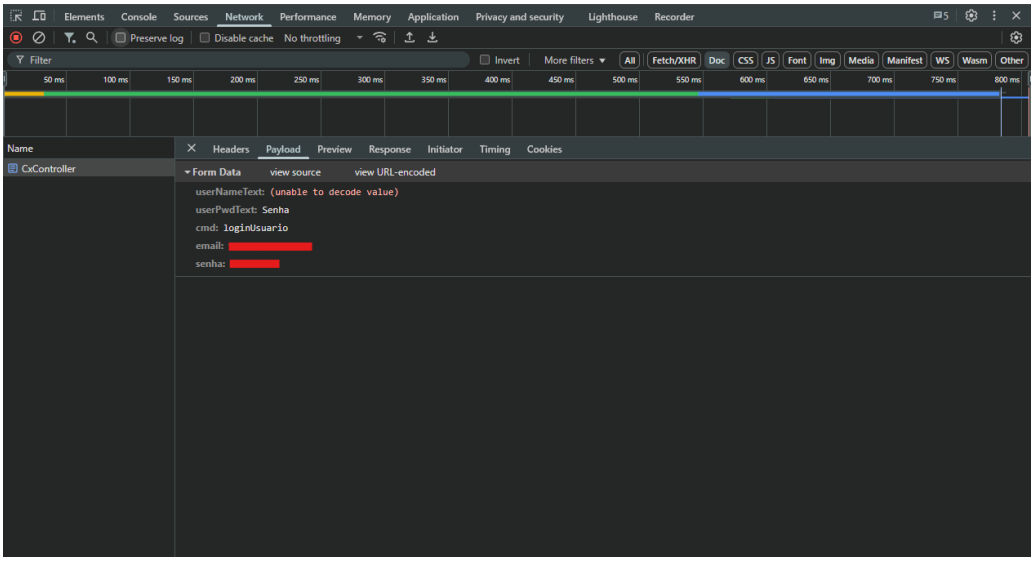

Figura 5 – Payload do parâmetro loginUsuario.

Por fim, o Servlet processa os dados enviados no payload, realizando a validação do login no banco de dados por meio de consultas utilizando Java Database Connectivity (JDBC). Caso as credenciais estejam corretas, o Servlet gera dinamicamente um arquivo HTML com o conteúdo da área logada, que é então enviado como resposta para o navegador. Dessa forma, a navegação entre páginas ocorre por meio da renderização do conteúdo no lado do servidor, caracterizando o modelo tradicional de aplicações web baseadas em JSP.

4.2 Metodologia de Desenvolvimento

Durante a execução do projeto, foi adotada uma abordagem iterativa e colaborativa, alinhada com boas práticas da Engenharia de Software (PRESSMAN, 2011). A equipe optou por um processo leve, com foco em entregas contínuas e melhoria progressiva do sistema.

As atividades foram organizadas em ciclos quinzenais, nos quais eram realizadas reuniões de alinhamento para definição de prioridades, avaliação das funcionalidades já implementadas e planejamento das próximas etapas. Esse processo favoreceu a comunicação entre os membros da equipe, facilitando o acompanhamento da evolução do projeto e a rápida adaptação a mudanças de requisitos.

Embora não tenha sido aplicada uma metodologia ágil formal como Scrum ou Kanban, diversos princípios dessas abordagens foram incorporados de forma prática, como a

colaboração contínua, a entrega incremental e o feedback recorrente. Isso permitiu garantir maior controle sobre o progresso e a qualidade do software, além de manter o foco nos objetivos definidos para cada ciclo.

4.3 Análise de Requisitos

Os requisitos de um sistema são elementos fundamentais que definem o que o software deve fazer e quais limitações ele deve respeitar para satisfazer os objetivos do projeto. Em geral, esses requisitos são divididos em dois grupos: os funcionais, que descrevem os comportamentos esperados da aplicação, e os não-funcionais, que abrangem aspectos como desempenho, segurança e usabilidade.

Neste projeto, a definição dos requisitos foi realizada a partir de uma investigação cuidadosa das operações já implementadas no sistema legado, além de reuniões com o professor orientador, em que a orientação foi essencial para destacar as funcionalidades prioritárias no processo de modernização.

4.3.1 Requisitos Funcionais

- RF01: Cadastro de Usuário — Permitir que o aluno realize seu registro no sistema, informando os dados necessários para criação de uma conta.
- RF02: *Login* — Disponibilizar um mecanismo de autenticação para que o usuário acesse sua conta utilizando e-mail e senha.
- RF03: Redefinição de Senha — Possibilitar a recuperação de senha por meio do envio de instruções ao e-mail previamente cadastrado.
- RF04: Logout — Implementar funcionalidade para que o usuário encerre sua sessão e retorne à tela de *login*.
- RF05: Alteração de Idioma — Oferecer ao usuário a opção de modificar o idioma da interface de acordo com sua preferência.
- RF06: Carregamento de Dados — Carregar automaticamente os dados do usuário ao iniciar a sessão, garantindo uma experiência personalizada.
- RF07: Notificação de Intelligent Learning System (ILS) Diário — Enviar lembrete diário ao usuário incentivando o preenchimento do formulário ILS.
- RF08: Registro de ILS Diário — Permitir que o usuário responda ao formulário ILS diário diretamente a partir da notificação exibida.

- RF09: Notificação de Comentários — Informar o usuário sobre comentários recentes realizados em conteúdos vinculados ao seu perfil.
- RF10: Notificação de Quizzes — Alertar o usuário quando houver novos quizzes disponíveis no sistema.
- RF11: Listagem de Disciplinas — Exibir ao usuário uma lista organizada de disciplinas, categorizadas como Matriculadas, Meu Curso e Públicas.
- RF12: Matrícula em Disciplina — Permitir que o usuário se inscreva em disciplinas disponíveis para matrícula.
- RF13: Visualização de Disciplina — Apresentar os detalhes de uma disciplina acessada pelo usuário, com informações completas.
- RF14: Acesso às Aulas — Oferecer acesso ao conteúdo das aulas relacionadas à disciplina, incluindo vídeo, slides ou versão completa.
- RF15: Edição de Perfil — Disponibilizar uma interface para que o usuário possa atualizar seus dados pessoais cadastrados.
- RF16: Registro de Contexto — Permitir que o usuário cadastre novos contextos de uso conforme suas necessidades.
- RF17: Consulta de Contexto — Possibilitar que o usuário visualize os contextos de acesso registrados anteriormente.
- RF18: Registro de Restrições — Habilitar o usuário a criar restrições de acesso para determinados recursos ou funcionalidades.
- RF19: Visualização de Restrições — Permitir que o usuário consulte todas as restrições que definiu no sistema.
- RF20: Exclusão de Restrições — Oferecer a opção de remover restrições cadastradas anteriormente.
- RF21: Registro de Preferências — Viabilizar que o usuário configure suas preferências de uso na plataforma.
- RF22: Visualização do ILS Completo — Permitir o acesso ao questionário ILS em sua versão completa.
- RF23: Resposta ao ILS Completo — Habilitar o usuário a responder todas as questões do formulário ILS integralmente.
- RF24: Listagem de Badges — Exibir todas as conquistas (*badges*) obtidas pelo usuário durante o uso do sistema.

4.3.2 Requisitos Não-Funcionais

- RNF01: Conectividade — O sistema deverá funcionar plenamente quando houver conexão ativa com a internet, sendo esta essencial para o uso completo da aplicação.
- RNF02: Compatibilidade Multiplataforma — A aplicação precisa ser executável em diversos navegadores, sistemas operacionais e dispositivos, garantindo portabilidade e bom desempenho em diferentes ambientes.
- RNF03: Interface Acessível — A interface do usuário deve ser responsiva, adaptando-se a vários tamanhos de tela, além de seguir diretrizes de acessibilidade para atender usuários com diferentes necessidades.
- RNF04: Segurança de Sessão — Deve-se assegurar que os dados dos usuários sejam mantidos em sigilo, utilizando mecanismos de autenticação, gerenciamento de sessões e proteção contra acessos indevidos.
- RNF05: Persistência e Confiabilidade — Os dados gerados e armazenados no sistema devem estar sempre disponíveis e protegidos contra alterações não autorizadas, assegurando a integridade da informação.
- RNF06: Manutenibilidade do Código — O sistema deve ser desenvolvido com base em uma arquitetura modular e bem organizada, com código documentado de acordo com padrões técnicos que favoreçam futuras manutenções e extensões.
- RNF07: Suporte a Idiomas (Internacionalização) — A plataforma deve ser internacionalizável, com possibilidade de exibir os conteúdos em mais de um idioma conforme as preferências do usuário.

4.4 Modelagem de dados

A modelagem de dados representa uma etapa essencial na construção de sistemas, pois define a estrutura lógica das informações que serão manipuladas ao longo da aplicação. Ela permite organizar, relacionar e garantir a integridade dos dados, facilitando a manutenção e evolução do sistema. Para esse projeto, a modelagem foi baseada nas funcionalidades do sistema legado, buscando preservar os dados relevantes e aprimorar a estrutura quando necessário.

Foi utilizado software MySQL Workbench[2], uma ferramenta visual amplamente utilizada para criação do Diagrama Entidade-Relacionamento (DER), o que facilitou a representação gráfica das tabelas, atributos, relacionamentos e suas respectivas chaves primárias e estrangeiras. Através dessa abordagem, foi possível compreender melhor o

[2] <https://www.mysql.com/products/workbench/>

domínio do sistema, padronizar nomenclaturas e garantir um modelo mais consistente e alinhado às regras de negócio.

Durante a modelagem, foram mapeadas todas as entidades principais do sistema, incluindo usuários, disciplinas, instituições, respostas ao ILS, entre outras. Essas entidades foram normalizadas para evitar redundâncias e garantir uma boa performance do banco de dados. Com isso, foi possível representar de forma clara como os dados estão interligados, o que facilitará as consultas, inserções e futuras evoluções da aplicação. A Figura 6 apresenta o DER do sistema CX.

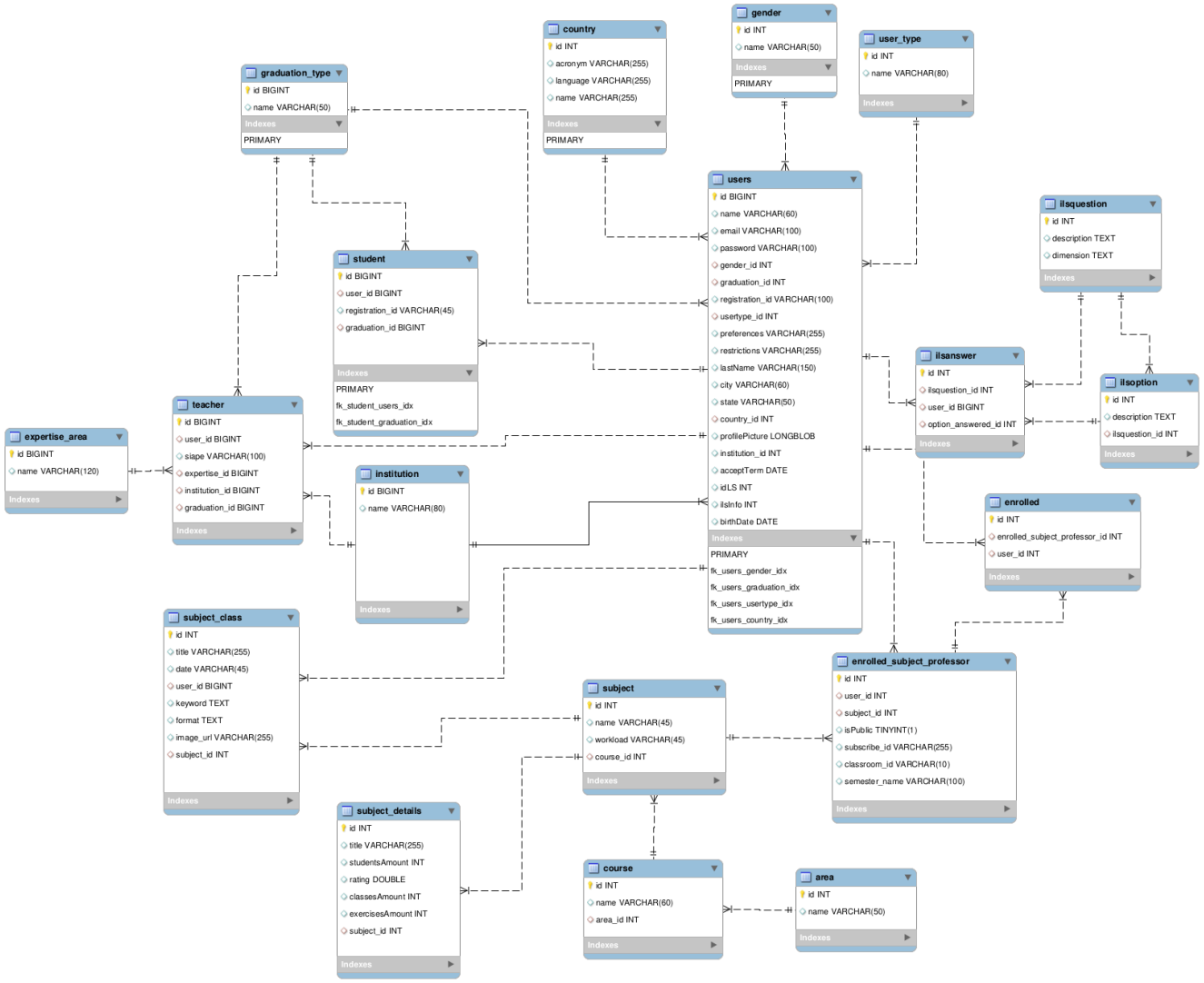

Figura 6 – Diagrama Entidade-Relacionamento (DER) do Classroom eXperience.

Para este projeto, foram criadas 19 entidades que representam os principais componentes e informações do sistema. Uma entidade, no contexto de banco de dados, é um objeto ou conceito que pode ser identificado de forma única dentro do domínio da aplicação e sobre o qual se deseja armazenar informações. Em um modelo relacional, cada entidade é representada por uma tabela, cujas colunas representam os atributos e as linhas representam os registros dessa entidade. Esse conceito permite estruturar os dados de maneira lógica e eficiente. A seguir, a listagem de todas as entidades criadas no projeto, acompanhadas de suas respectivas descrições.

- **Area**: representa uma área de conhecimento ou atuação. Utilizada para categorizar cursos, disciplinas ou especializações disponíveis no sistema.
- **Context**: define um conjunto de informações contextuais associadas ao perfil ou momento de acesso do usuário, como localização, horário, ou ambiente.
- **Country**: armazena os dados dos países cadastrados. Essa entidade é utilizada em formulários institucionais ou nos perfis de usuários.
- **Course**: representa os cursos acadêmicos disponíveis no sistema, com informações como nome, duração e vínculo institucional.
- **Enrolled**: registra as matrículas dos alunos nas disciplinas, vinculando estudantes às turmas em que estão ativos.
- **EnrolledSubjectProfessor**: realiza a associação entre estudantes, disciplinas e os professores responsáveis, permitindo o controle da relação acadêmica entre eles.
- **ExpertiseArea**: identifica a área de especialização dos usuários, geralmente utilizada para qualificar professores ou profissionais do sistema.
- **Gender**: define os gêneros disponíveis no sistema, sendo utilizado no cadastro e personalização de perfil dos usuários.
- **GraduationType**: classifica os tipos de formação acadêmica, como bacharelado, licenciatura ou tecnólogo, aplicáveis aos cursos ofertados.
- **IlsAnswer**: armazena as respostas fornecidas pelos usuários ao questionário ILS, utilizado para análise de perfil de aprendizagem.
- **IlsOption**: contém as opções de resposta disponíveis para as questões do ILS, garantindo padronização na coleta de dados.
- **IlsQuestion**: lista as perguntas que compõem o questionário ILS, sendo cada item vinculado às respectivas opções.

- **Institution**: representa as instituições de ensino cadastradas no sistema, responsáveis por ofertar cursos e associar usuários.
- **Restrictions**: registra restrições de acesso configuradas pelos usuários, permitindo personalização do uso da plataforma.
- **Student**: representa os usuários do tipo aluno, armazenando suas informações acadêmicas e de acesso ao sistema.
- **Subject**: refere-se às disciplinas disponíveis no sistema, incluindo nome, descrição, conteúdo e vinculação a cursos e professores.
- **SubjectDetails**: representa informações complementares sobre uma disciplina, incluindo palavras-chave e formatos de apresentação, permitindo uma caracterização mais rica e detalhada dos conteúdos oferecidos.
- **SubjectClass**: modela uma aula específica vinculada a uma disciplina, contendo dados como título, data, professor responsável e material de apoio, sendo essencial para organizar e apresentar o cronograma das aulas da plataforma.
- **Teacher**: representa os usuários do tipo professor, com suas credenciais e vinculações às disciplinas e áreas de atuação.
- **User**: entidade genérica que define os dados comuns a todos os tipos de usuários da plataforma, como login, senha e informações básicas.
- **UserType**: define os papéis possíveis dos usuários no sistema (aluno, professor, administrador), determinando permissões e acessos.

A criação das entidades foi automatizada por meio do uso do JPA, uma especificação Java para Object-Relational Mapping (ORM). Com o auxílio de anotações diretamente no código das classes Java, foi possível definir tabelas, chaves primárias, relacionamentos e demais restrições de forma integrada ao desenvolvimento da aplicação. Essa abordagem promove agilidade na modelagem de dados, facilita a manutenção futura e garante alinhamento entre a estrutura do banco e a lógica de negócio.

A Figura 7 apresenta um exemplo prático de classe de entidade implementada com JPA, onde são especificadas suas colunas, chave primária e a cardinalidade dos relacionamentos com outras entidades. O uso dessas anotações permite mapear com precisão o modelo relacional no código, proporcionando maior controle sobre o comportamento das entidades e favorecendo a consistência entre o sistema e o banco de dados.

```java
package com.ufu.classroom_api.entities;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.util.List;

@Getter
@Setter
@Entity(name = "ilsquestion")
public class IlsQuestion {

    @Id
    private Integer id;

    @Column(name = "description")
    private String description;

    @Column(name = "dimension")
    private String dimension;

    @OneToMany(mappedBy = "ilsQuestion", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<IlsOption> options;

    @OneToMany(mappedBy = "ilsQuestion", cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.LAZY)
    private List<IlsAnswer> ilsAnswer;
}
```

Figura 7 – Entidade de exemplo.

4.5 Escolha de Tecnologias

4.5.1 Java

A linguagem Java foi escolhida para o projeto devido a sua extensa documentação, robustez e ampla adoção no mercado. Por ser uma linguagem orientada a objetos, com uma vasta biblioteca padrão que auxilia no desenvolvimento e manutenção de aplicações. Além disso, a linguagem possui excelente suporte para integração com banco de dados, autenticação e persistência de dados, aspectos fundamentais para o projeto.

O Java proporciona um ecossistema confiável e estável, sustentado por uma comunidade ativa e ferramentas consolidadas de desenvolvimento, IDEs como IntelliJ e Eclipse, sistemas de build (Maven, Gradle) e ferramentas de testes. A grande disponibilidade de materiais de estudo e o suporte a boas práticas de desenvolvimento tornam o Java uma opção adequada para aplicações de médio a grande porte, como o sistema proposto neste trabalho.

4.5.2 Spring Boot

O Spring Boot é um framework que simplifica o desenvolvimento de aplicações Java, especialmente aplicações web e APIs REST. Integrado ao ecossistema Spring, amplamente adotado em soluções corporativas, o Spring Boot destaca-se por sua abordagem de configuração automática e por eliminar grande parte do código repetitivo necessário em configurações tradicionais, acelerando assim o processo de desenvolvimento.

Com ele, é possível criar aplicações estruturadas e organizadas de maneira eficiente, seguindo boas práticas de arquitetura de software, como a separação em camadas e o

uso da injeção de dependência. O Spring Boot também oferece suporte nativo a diversos módulos fundamentais, como o Spring Security (para autenticação e autorização), o Spring Data JPA, o Spring Actuator (para monitoramento) e ferramentas de teste integradas, proporcionando maior robustez e facilidade de manutenção ao sistema.

A utilização do Spring Boot neste projeto permitiu o desenvolvimento ágil e padronizado da aplicação, com foco na escalabilidade e segurança. A seguir, serão apresentados os principais aspectos estruturais do sistema, incluindo a organização das camadas, a implementação da segurança e o funcionamento dos endpoints construídos com base na arquitetura proposta.

4.5.2.1 Estrutura e Camadas do Projeto

A estrutura do projeto foi organizada com base em uma estrutura por *features*, em vez da tradicional separação estrita por camadas (como *controller*, *service* e *repository* em diretórios globais). Essa decisão foi tomada com o objetivo de facilitar uma possível migração futura para uma arquitetura de microsserviços, permitindo maior modularidade e isolamento das funcionalidades do sistema.

Dentro da estrutura do projeto, algumas pastas foram criadas para organizar melhor os componentes, como mostra a Figura 8. A pasta *commons* concentra as classes comuns que são compartilhadas por toda a aplicação, um exemplo são as classes de exceção utilizadas de forma compartilhada, promovendo reutilização e padronização no tratamento de erros. Já a pasta config contém configurações essenciais para o funcionamento da aplicação, como a integração com o *Swagger* para documentação da API e os arquivos de configuração relacionados à segurança do sistema (como autenticação e autorização via *Spring Security*).

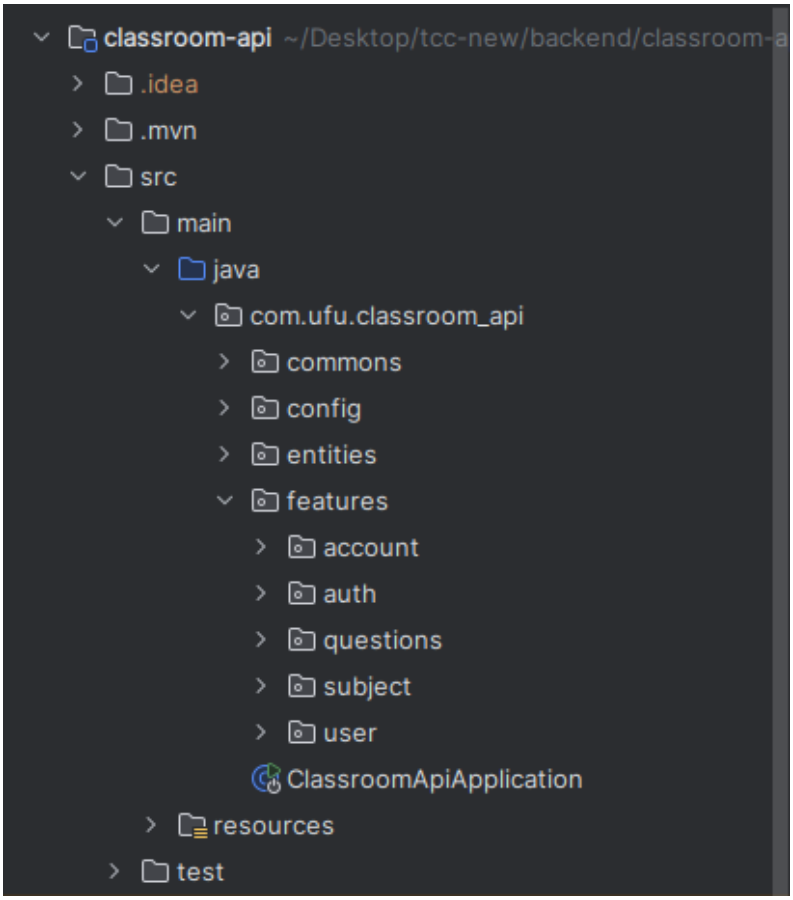

Figura 8 – Estrutura do projeto.

Além disso, há a pasta *entities*, responsável por agrupar todas as classes de entidade que representam as tabelas do banco de dados, mapeadas com anotações do JPA. Por fim, a pasta *features* organiza as funcionalidades principais da aplicação. Dentro de cada feature, encontram-se suas respectivas classes de *controller*, *service* e *repository*, o que garante coesão entre os componentes relacionados a uma mesma funcionalidade. Essa abordagem facilita o entendimento do sistema, melhora a manutenção do código e abre caminho para uma separação natural das responsabilidades em módulos ou serviços independentes no futuro.

4.5.2.2 Segurança

A segurança da aplicação foi implementada utilizando os recursos do *Spring Security* juntamente com autenticação via JWT. Essa combinação permite proteger os *endpoints* da API, garantindo que apenas usuários autenticados tenham acesso a determinadas funcionalidades, sem a necessidade de sessões persistentes no servidor (*Stateless Authentication*).

A diferença entre uma autenticação *stateful* e *stateless* está na forma como as informações de sessão do usuário são armazenadas. Na autenticação *stateful*, os dados de

estado do usuário são mantidos no servidor, podendo ser em memória ou em um banco de dados. Esse modelo não é recomendado para aplicações baseadas em *APIs*, devido à complexidade de escalabilidade. Já na autenticação *stateless*, o servidor não armazena informações de sessão. Em vez disso, após o *login*, o servidor gera um *token* como, por exemplo, o JWT e o envia ao cliente. Esse *token* contém todas as informações necessárias para autenticar o usuário e deve ser enviado junto com cada requisição, geralmente por meio do cabeçalho *Authorization*.

O processo de autenticação começa com a classe UserDetailsServiceImpl, mostrada pela Figura 9, que implementa a lógica para carregar um usuário com base no e-mail. Ela consulta o banco de dados através do UserService e constrói um objeto CustomUserDetails, que contém as informações necessárias para autenticação.

```java
package com.ufu.classroom_api.config.security;

import com.ufu.classroom_api.entities.User;
import com.ufu.classroom_api.features.user.service.UserService;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import java.util.Optional;

@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    private final UserService authService;

    public UserDetailsServiceImpl(UserService authService) {
        this.authService = authService;
    }

    @Override
    public CustomUserDetails loadUserByUsername(String email) throws UsernameNotFoundException {

        Optional<User> user = authService.findByEmail(email);

        CustomUserDetails customUserDetails = new CustomUserDetails();
        if (user.isPresent()) {
            customUserDetails.setUsername(user.get().getEmail());
            customUserDetails.setPassword(user.get().getPassword());
            customUserDetails.setId(user.get().getId());

            return customUserDetails;
        } else {
            throw new UsernameNotFoundException("User not found.");
        }
    }
}
```

Figura 9 – UserDetailsServiceImpl.

A classe SecurityConfig, mostrada pela Figura 10, configura toda a segurança da aplicação, incluindo regras de acesso às rotas, permissões públicas (como /login e /swagger-ui), e a criação do SecurityFilterChain, responsável por interceptar e proteger as requisições. Também é nela que ocorre a configuração do AuthenticationManager, além da definição do PasswordEncoder (BCrypt) e da chave de decodificação dos tokens.

```java
@Bean
public PasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}

@Bean
public JwtDecoder jwtDecoder() {
    SecretKey secretKey = jwtConfig.getKey();
    return NimbusJwtDecoder.withSecretKey(secretKey).build();
}

@Bean
public AuthenticationManager authenticationManager(
        AuthenticationConfiguration authConfig) throws Exception {
    return authConfig.getAuthenticationManager();
}

@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws
        Exception{

    http.cors(AbstractHttpConfigurer::disable)
            .csrf(AbstractHttpConfigurer::disable)
            .sessionManagement((sessionManagement) -> sessionManagement.
                    sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests(authorizeHttpRequests ->
                    authorizeHttpRequests
                            .requestMatchers("/swagger-ui/**", "/v3/api-docs/**").permitAll()
                            .requestMatchers(HttpMethod.POST, "/login/**").permitAll()
                            .requestMatchers("/register/**").permitAll()
                            .anyRequest().authenticated())
            .cors(c -> c.configurationSource(customCorsConfiguration))
            .addFilterBefore(authenticationFilter, UsernamePasswordAuthenticationFilter.class);
    return http.build();
}
```

Figura 10 – SecurityConfig

O JwtService, mostrado na Figura 11, é o componente responsável por gerar e validar os tokens JWT. Ele define o tempo de expiração, assina os tokens com uma chave secreta e extrai informações como o ID do usuário autenticado. Quando o cliente envia uma requisição com o token, o serviço valida sua integridade e devolve os dados contidos no token.

```java
@Component
public class JwtService {

    private final long expirationTime;

    static final String PREFIX = "Bearer ";

    private final JwtConfig jwtConfig;

    public JwtService(JwtConfig jwtConfig, @Value("${jwt.expiration.time}") Long expirationTime){
        this.jwtConfig = jwtConfig;
        this.expirationTime = expirationTime;
    }

    public String getToken(String username, Long idUser) {

        return Jwts.builder()
                .subject(idUser.toString())
                .expiration(new Date(System.currentTimeMillis() +
                        expirationTime))
                .signWith(jwtConfig.getKey())
                .compact();
    }

    public String getAuthUser(HttpServletRequest request) {
        String token = request.getHeader
                (HttpHeaders.AUTHORIZATION);
        if (token != null) {

                return Jwts.parser()
                        .verifyWith(jwtConfig.getKey())
                        .build()
                        .parseSignedClaims(token.replace(PREFIX, ""))
                        .getPayload()
                        .getSubject();
        }
        return null;
    }

    public Claims getAuthClaims(HttpServletRequest request) {
        String token = request.getHeader
                (HttpHeaders.AUTHORIZATION);
        if (token != null) {
            return Jwts.parser()
                        .verifyWith(jwtConfig.getKey())
                        .build().parseSignedClaims(token.replace("Bearer ", ""))
                        .getPayload();
        }
        return null;
    }
}
```

Figura 11 – JwtService

A classe AuthenticationFilter, mostrada na Figura 12, intercepta todas as requisições antes de chegarem aos controladores. Ela extrai o token do cabeçalho Authorization, valida com o JwtService, e se tudo estiver correto, define o usuário autenticado no contexto de segurança do Spring.

```java
@Component
public class AuthenticationFilter extends OncePerRequestFilter{

    private final JwtService jwtService;

    public AuthenticationFilter(JwtService jwtService) {
        this.jwtService = jwtService;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response, FilterChain filterChain)
            throws ServletException, java.io.IOException {

        String jws = request.getHeader(HttpHeaders.AUTHORIZATION);

        if (jws != null) {

            String user = jwtService.getAuthUser(request);

            Claims claims = jwtService.getAuthClaims(request);

            Authentication authentication =
                    new UsernamePasswordAuthenticationToken(user, claims.get("id"),
                            java.util.Collections.emptyList());
            SecurityContextHolder.getContext()
                    .setAuthentication(authentication);
        }
        filterChain.doFilter(request, response);
    }
}
```

Figura 12 – AuthenticationFilter

No sistema desenvolvido, há um exemplo prático de autenticação utilizando JWT por meio do endpoint de */login*, onde o usuário fornece suas credenciais e, em caso de autenticação bem-sucedida, recebe um *token* JWT como resposta, mostrado na Figura 13. Esse *token* contém as informações necessárias para validar o usuário e deve ser enviado pelo cliente nas próximas requisições, no cabeçalho *Authorization*, utilizando o prefixo *Bearer*. Em um endpoint protegido, o sistema verifica automaticamente a presença e a validade do *token* antes de conceder o acesso, garantindo que apenas usuários autenticados possam acessar recursos sensíveis ou restritos. Esse fluxo reforça a segurança e a escalabilidade da aplicação, seguindo boas práticas para APIs RESTful.

4.5.2.3 Funcionamento do Framework

O funcionamento do *framework* Spring Boot baseia-se na separação de responsabilidades por camadas, facilitando a organização e manutenção do código. A primeira camada é a *Controller*, responsável por receber as requisições HTTP e retornar as respostas apropriadas. Nessa camada, definem-se os endpoints da aplicação e o mapeamento entre as rotas e os métodos que executam a lógica de controle. As classes *Controller* atuam como ponto de entrada da aplicação, delegando as responsabilidades para as camadas subsequentes. A Figura 15 mostra a classe *Controller* responsável pelo gerenciamento de assuntos de uma aula.

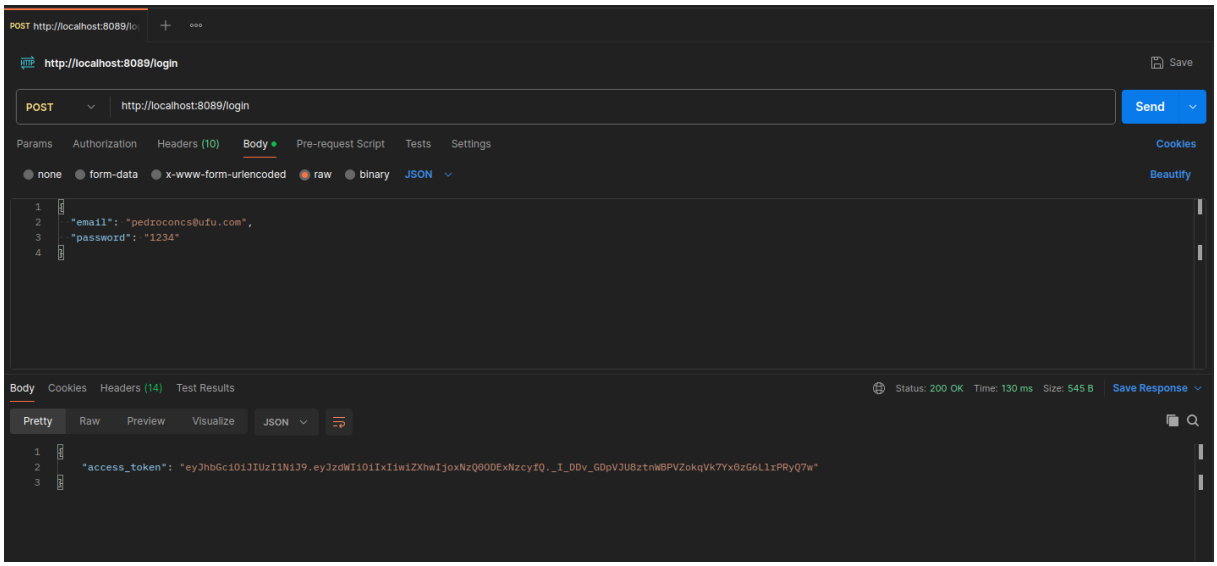

Figura 13 – Geração do JWT a partir do login

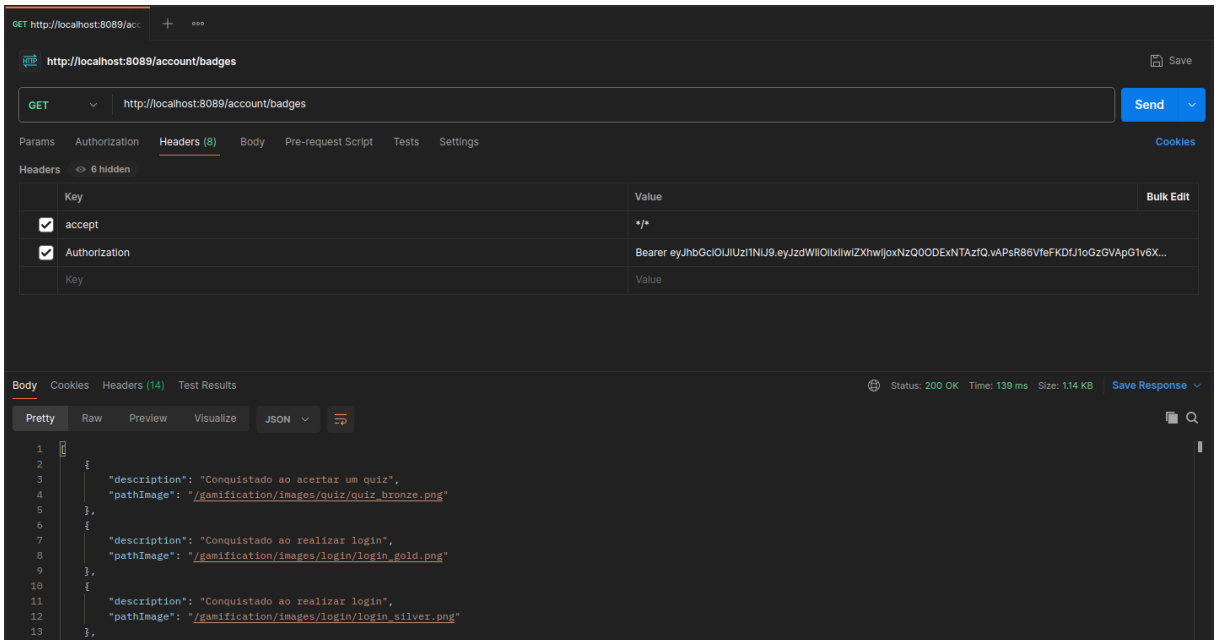

Figura 14 – Endpoint teste enviando Authorization

```java
@RestController
@RequestMapping("/subjects")
public class SubjectController {

    @Autowired
    private SubjectService subjectService;

    @GetMapping
    @Operation(summary = "Get all enrolled subjects",
            description = "List all subjects enrolled by user ID with teacher info")
    private ResponseEntity<List<SubjectResponse>> getAllSubjectsWithTeacher(Authentication authentication){

        Long user_id =  Long.valueOf(authentication.getPrincipal().toString());

        return ResponseEntity.ok(subjectService.getAllSubjectsWithTeacher(user_id));
    }

    @GetMapping("/public")
    @Operation(summary = "Get all public subjects",
            description = "List all public subjects with teacher info")
    private ResponseEntity<List<SubjectResponse>> getAllPublicSubjects(){

        return ResponseEntity.ok(subjectService.getAllPublicSubjects());
    }

    @GetMapping("/{course_id}")
    @Operation(summary = "Get all subjects",
            description = "List all subjects registered with course info")
    private ResponseEntity<List<SubjectResponse>> getAllSubjects(@PathVariable Long course_id) {

        return ResponseEntity.ok(subjectService.findAllSubjectsByCourse(course_id));
    }
```

Figura 15 – SubjectController

A camada *Service* contém a lógica de negócio da aplicação. É nela que ocorrem os processamentos principais, validações e regras específicas do domínio do sistema. A *Service* é chamada pela *Controller* e, por sua vez, comunica-se com a camada de persistência de dados (*Repository*). Essa separação permite uma maior reutilização de código e facilita a aplicação de testes unitários, pois encapsula as regras de negócio em um único ponto, sem depender diretamente das requisições externas. A Figura 16 mostra a classe *SubjectService* responsável pelas regras de negócio relacionadas aos assuntos de uma aula.

Por fim, a camada *Repository* é responsável pela comunicação direta com o banco de dados. Utilizando o Spring Data JPA, os repositórios são interfaces que herdam funcionalidades prontas, como salvar, buscar, atualizar e deletar registros, eliminando a necessidade de implementar consultas SQL básicas manualmente. Além disso, é possível definir métodos personalizados utilizando convenções de nomenclatura, tornando a persistência de dados simples, eficiente e integrada com as entidades do sistema. A Figura 17 mostra a interface *SubjectRepository* utilizada para executar consultas no banco de dados relacionadas à assuntos das aulas.

```java
@Service
public class SubjectService {

    @Autowired
    private SubjectRepository subjectRepository;

    @Autowired
    private EnrollRepository enrollRepository;

    @Autowired
    private EnrolledSubjectProfessorRepository enrolledSubjectProfessorRepository;

    @PersistenceContext
    private EntityManager entityManager;

    @Autowired
    private UserRepository userRepository;

    public List<SubjectResponse> getAllSubjectsWithTeacher(Long user_id){

        return subjectRepository.findAllEnrolledSubjectsWithTeacher(user_id);
    }

    public void enrollStudent(Long subject_id, Long user_id){

        EnrolledSubjectProfessor enrolledSubjectProfessor = enrolledSubjectProfessorRepository
                .findBySubject_Id(subject_id)
                .orElseThrow(() -> new EntityNotFoundException("EnrolledSubjectProfessor not found for subject_id: " + subject_id));

        User user = userRepository
                .findById(user_id)
                .orElseThrow(() -> new EntityNotFoundException("User not found"));

        enrollRepository.save(new Enrolled(enrolledSubjectProfessor, user));
    }
```

Figura 16 – SubjectService

```java
@Repository
public interface SubjectRepository extends JpaRepository<Subject, Long> {

    @Query("SELECT new com.ufu.classroom_api.features.subject.response.SubjectResponse(" +
            "sub.id," +
            "sub.name as title, " +
            "new com.ufu.classroom_api.features.subject.response.TeacherResponse(use.id," +
             "(select name || ' ' || lastName from User where id = enrpro.user.id))) " +
            "FROM User use " +
            "INNER JOIN Enrolled enr " +
            "ON use.id = enr.user.id " +
            "INNER JOIN EnrolledSubjectProfessor enrpro " +
            "ON enr.enrolledSubjectProfessor.id = enrpro.id " +
            "INNER JOIN Subject sub " +
            "ON enrpro.subject.id = sub.id " +
            "WHERE use.id = :user_id")
    List<SubjectResponse> findAllEnrolledSubjectsWithTeacher(Long user_id);

}
```

Figura 17 – SubjectRepository

4.5.3 PostgreSQL

O PostgreSQL é um Sistema Gerenciador de Banco de Dados (SGBD) open-source que é amplamente utilizado em aplicações que demandam consistência, confiabilidade e alto desempenho. Ele adota o modelo relacional tradicional, organizando os dados em tabelas compostas por linhas e colunas, além de permitir a criação de chaves primárias, estrangeiras, índices e restrições para assegurar a integridade dos dados. Com suporte completo às propriedades Atomicity, Consistency, Isolation, Durability (ACID) (Atomi-

cidade, Consistência, Isolamento e Durabilidade), o PostgreSQL é uma solução indicada para sistemas que exigem controle rigoroso sobre a persistência das informações.

No contexto deste projeto, o PostgreSQL foi mantido como sistema de banco de dados, uma vez que já era utilizado na versão anterior do sistema legado, desenvolvida em JSP. Contudo, foi necessário realizar a atualização da versão do PostgreSQL para garantir compatibilidade com as tecnologias mais recentes adotadas, além de melhorar a performance e a segurança. Essa atualização também contribuiu para uma estrutura de dados mais moderna e eficiente, facilitando a integração com as novas ferramentas da aplicação.

4.5.4 Docker

A virtualização é uma técnica consolidada na área da computação que permite a execução de múltiplos sistemas operacionais em uma mesma máquina física. Esse modelo tende a consumir mais recursos, devido à necessidade de simular um sistema operacional completo para cada aplicação. Em projetos modernos, como aplicações web simples ou microsserviços, esse nível de sobrecarga pode ser evitado com o uso de contêineres.

Os contêineres são uma evolução da virtualização, eles oferecem ambientes isolados e leves para a execução de aplicações, utilizando diretamente o sistema operacional do host. A principal ferramenta utilizada nesse contexto é o Docker, que permite empacotar uma aplicação e todas as suas dependências em uma imagem portátil, garantindo que ela funcione da mesma forma em qualquer ambiente. Essa padronização facilita a integração contínua, testes automatizados e implantação em escala, além de reduzir o risco de falhas relacionadas ao ambiente.

No contexto deste projeto, que possui uma única API em Spring Boot e um banco de dados, os contêineres oferecem uma solução prática e eficiente. Com o uso de Docker e Docker Compose, é possível configurar ambos os serviços de maneira simplificada, garantindo o isolamento entre a aplicação e o banco, além de facilitar a replicação do ambiente por outros desenvolvedores. Esse modelo também torna a aplicação mais preparada para futuras evoluções, como escalabilidade e separação de serviços.

Sendo assim, a utilização de contêineres contribui significativamente para a portabilidade, automação de ambientes e facilidade de manutenção do projeto. Mesmo em sistemas que não exigem alta complexidade de infraestrutura, essa tecnologia agrega muito valor por proporcionar confiabilidade na execução da aplicação em diferentes etapas do desenvolvimento e produção, reforçando as boas práticas de DevOps e engenharia de software moderna. A Figura 18 ilustra a arquitetura de contêiner utilizada neste projeto.

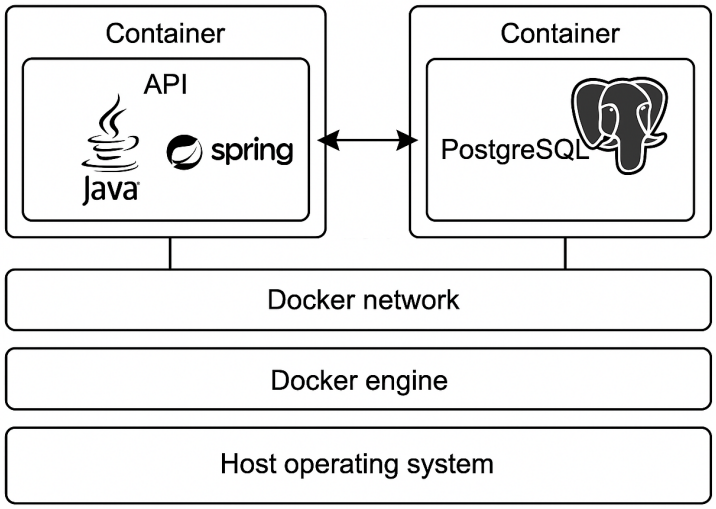

Figura 18 – Arquitetura de contêiner.

Fonte: Próprio autor.

4.6 Documentação

A documentação da API é uma parte essencial no desenvolvimento de sistemas, pois permite que outros desenvolvedores compreendam de forma clara como interagir com os serviços disponíveis, quais são os endpoints existentes, os parâmetros esperados, os tipos de retorno e os códigos de resposta. Para isso foi utilizado o *Swagger*, uma ferramenta amplamente adotada para geração de documentação interativa de APIs REST. Com ele, é possível testar as requisições diretamente na interface gráfica, o que facilita a verificação do funcionamento dos endpoints durante o desenvolvimento e também serve como referência para futuras manutenções e integrações.

Neste projeto, o Swagger foi configurado para documentar cinco controladores principais: o *account-controller* (Figura 19), responsável por operações de manipulação de contexto, restrições e preferencias; o *auth-controller* (Figura 20), que trata dos processos de autenticação, como login e geração de tokens; o *question-controller* (Figura 21), encarregado de lidar com as perguntas utilizadas em quizzes ou avaliações do sistema; o *subject-controller* (Figura 22), que gerencia as funcionalidades ligadas às disciplinas cadastradas; e o *user-controller* (Figura 23), que centraliza as ações voltadas à manipulação e visualização dos dados dos usuários. Com essa documentação automatizada, é possível visualizar claramente as rotas da aplicação, os métodos suportados (GET, POST, PUT, DELETE), além das estruturas de dados envolvidas, promovendo maior transparência, produtividade e organização no desenvolvimento da aplicação.

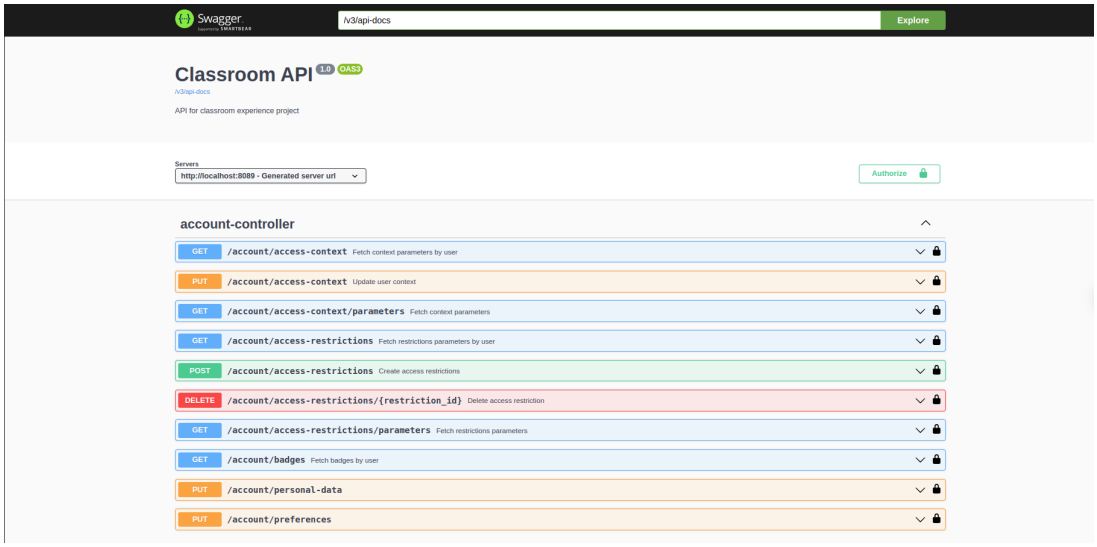

Figura 19 – Swagger Classroom API AccountController.

Fonte: Próprio autor.

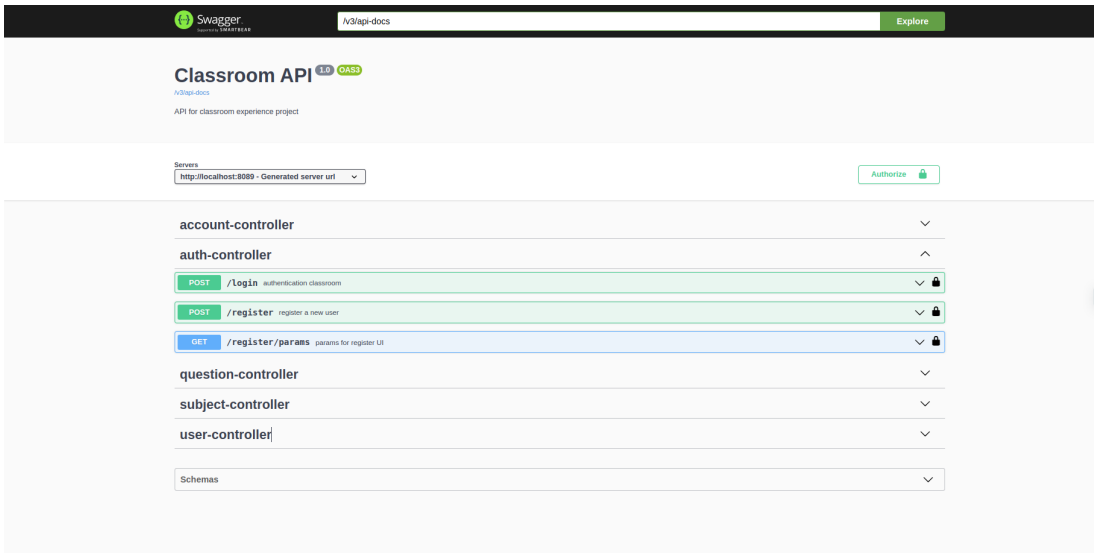

Figura 20 – Swagger Classroom API AuthController.

Fonte: Próprio autor.

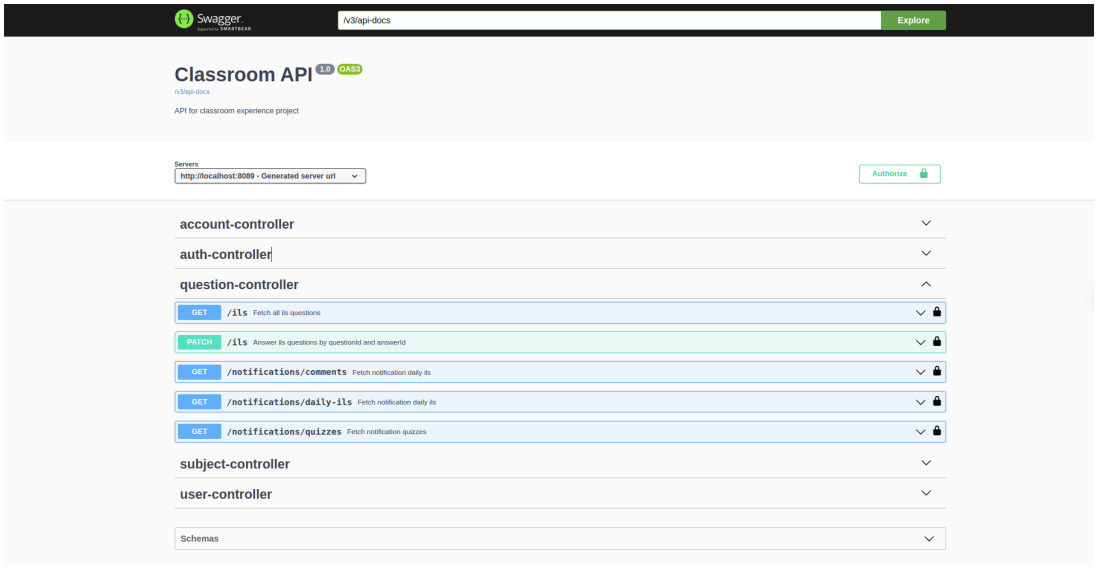

Figura 21 – Swagger Classroom API QuestionController.

Fonte: Próprio autor.

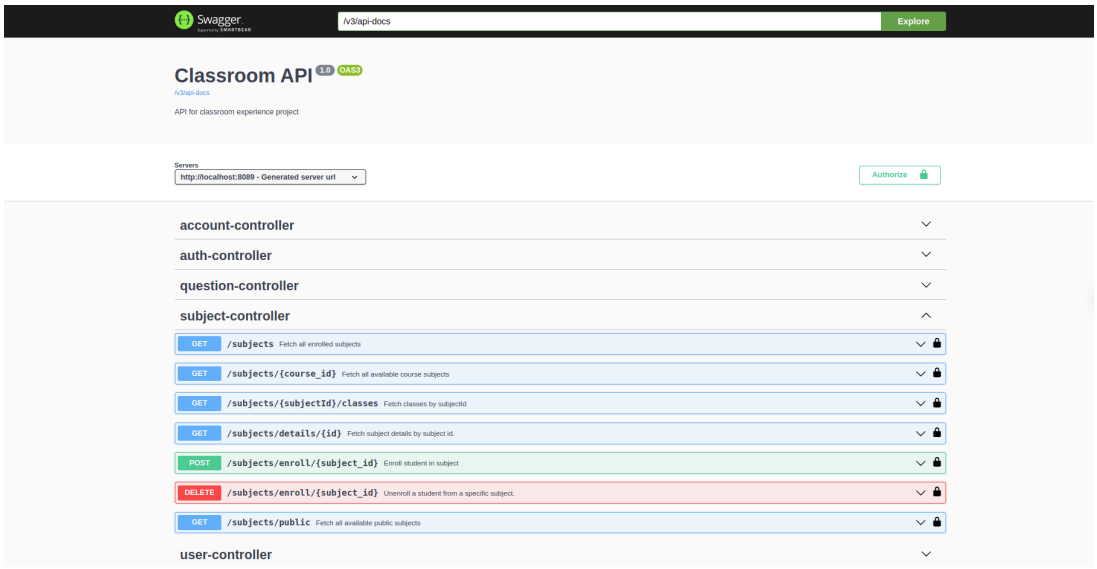

Figura 22 – Swagger Classroom API SubjectController.

Fonte: Próprio autor.

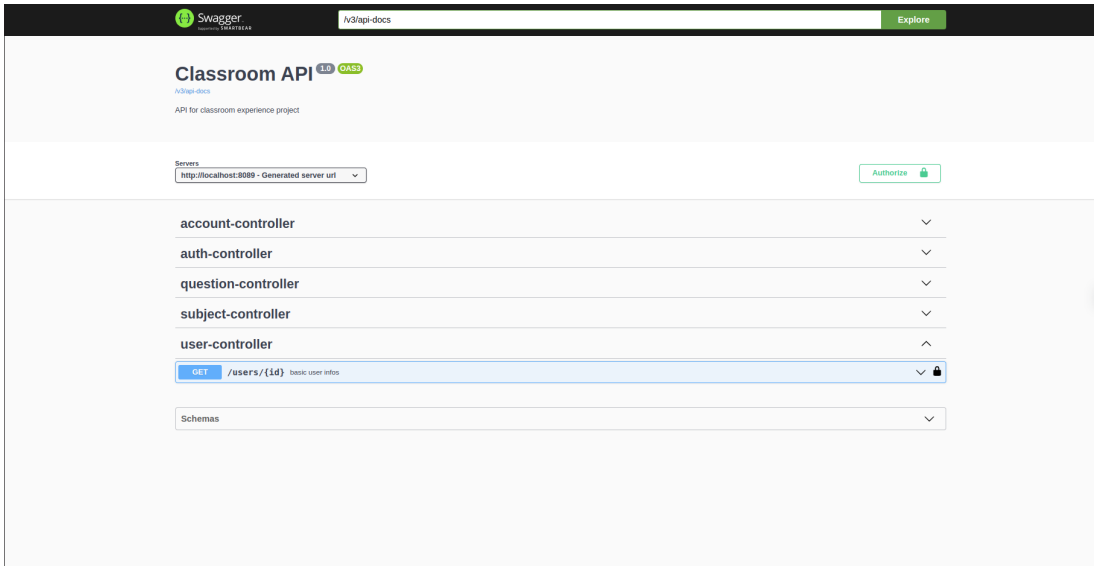

Figura 23 – Swagger Classroom API UserController.

Fonte: Próprio autor.

4.7 Diagrama de Componentes

A arquitetura do sistema foi modelada seguindo o padrão de camadas proposto pelo Spring Boot, e está representada no diagrama de componentes da Figura 24. Essa representação permite visualizar a estrutura modular do sistema, destacando os principais pacotes e suas responsabilidades.

A aplicação foi organizada em pacotes funcionais que representam features do sistema, como account, auth, questions, subject e user. Cada feature possui três camadas principais: Controller, responsável por lidar com as requisições HTTP; Service, onde está concentrada a lógica de negócio; e Repository, que abstrai o acesso aos dados utilizando Spring Data JPA.

Além das features, o sistema também conta com módulos de suporte como:

- **security**: responsável pela autenticação e autorização, integrando-se ao Spring Security.
- **doc**: responsável pela documentação da API, utilizando o Swagger.
- **config e commons**: que centralizam configurações gerais e componentes reutilizáveis em toda a aplicação.
- **entities**: camada que representa os modelos persistentes mapeados com JPA.

Os componentes interagem com uma camada de persistência que se conecta ao banco de dados PostgreSQL. A comunicação entre os módulos é feita via injeção de dependência do Spring, promovendo baixo acoplamento e alta coesão entre as partes.

Essa arquitetura foi pensada para garantir manutenibilidade, clareza na separação de responsabilidades e escalabilidade futura, caso haja a necessidade de transformação para uma arquitetura baseada em microsserviços.

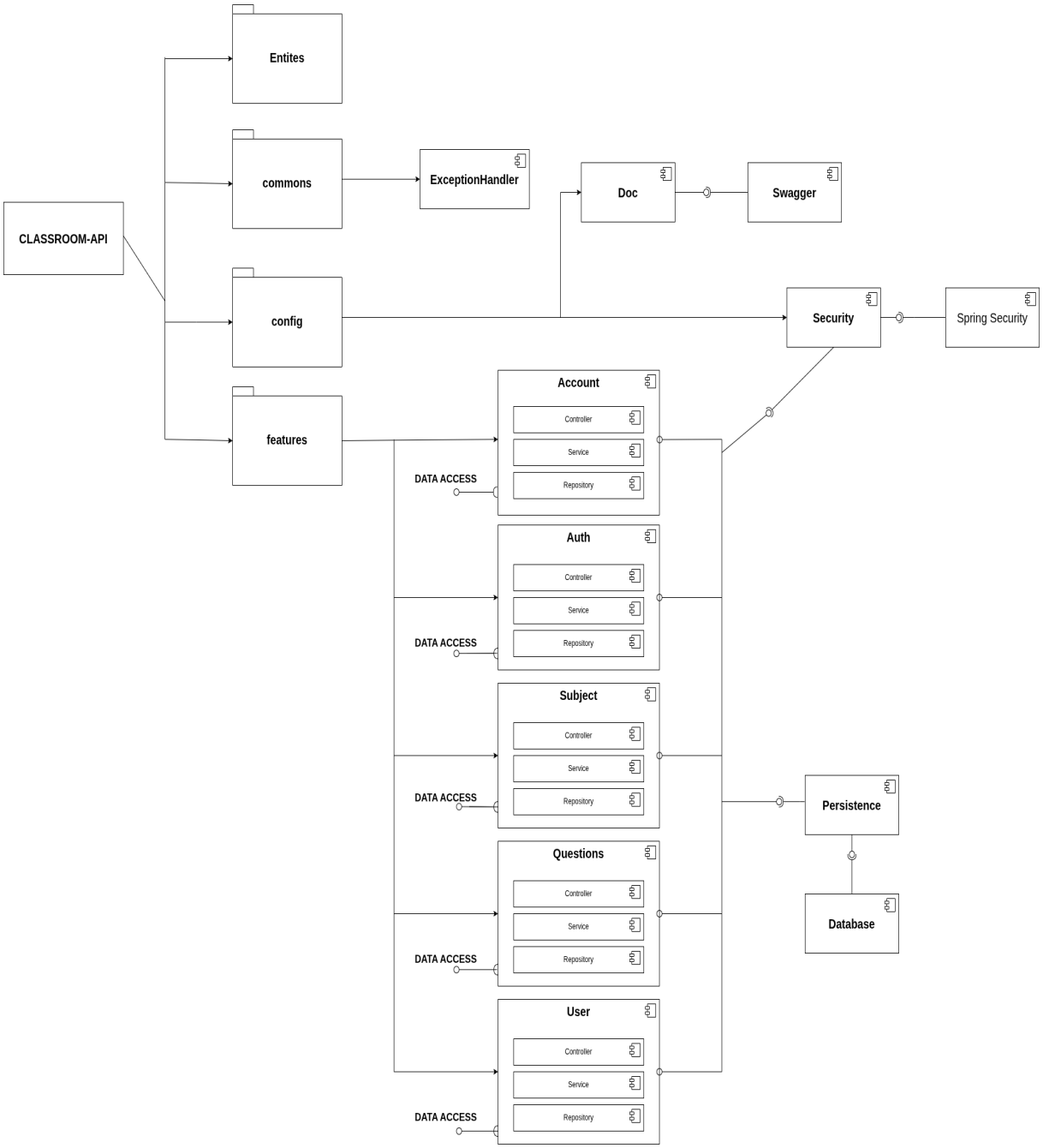

Figura 24 – Diagrama de Componentes.

Fonte: Próprio autor.

4.8 Testes e Métricas

A realização de testes automatizados é uma etapa essencial no desenvolvimento de sistemas, pois garante que o comportamento da aplicação atenda aos requisitos esperados ao longo do tempo, mesmo com a evolução contínua do código. No contexto deste trabalho, foram aplicados testes unitários e de integração para verificar o correto funcionamento dos serviços e controladores da aplicação. Essa prática não apenas contribui para a identificação precoce de falhas, mas também reforça a confiança na estabilidade do sistema, tornando-o mais robusto e confiável para o ambiente de produção.

Os testes foram implementados utilizando a biblioteca JUnit em conjunto com o framework Mockito, possibilitando a simulação de dependências e o isolamento de componentes. Para os testes de integração, foi utilizado o suporte do Spring Boot, permitindo simular requisições HTTP e testar os endpoints de forma realista, sem a necessidade de um servidor externo. Com essas abordagens, foi possível verificar não apenas a lógica interna das classes, mas também o fluxo completo das requisições REST, incluindo autenticação e manipulação de dados.

Como métrica para avaliar a eficácia dos testes, foi utilizada a porcentagem de cobertura de código (code coverage), gerada por ferramentas como JaCoCo. Essa métrica apresenta o percentual de código que foi efetivamente executado durante a execução dos testes. Os principais indicadores analisados foram a cobertura de linhas, que mostra quantas instruções foram executadas; métodos, que indica quantos métodos foram invocados; e branches, que revela quantas bifurcações condicionais (como if/else ou switch) foram testadas. Uma boa cobertura não garante ausência de bugs, mas é um indicativo importante de que o sistema foi amplamente validado. A Figura 25 mostra os resultados obtidos.

Figura 25 – Métricas de cobertura dos testes da API.

Fonte: Próprio autor.

4.9 Integração Contínua (CI)

A Integração Contínua (*Continuous Integration*) é uma prática fundamental no desenvolvimento moderno de software. Ela consiste na automação do processo de construção, teste e validação contínua do código a cada alteração realizada no repositório. Essa abordagem tem como objetivo detectar rapidamente falhas, reduzir o tempo de entrega e melhorar a qualidade geral do software.

Durante o desenvolvimento do projeto, foi configurado um pipeline de CI utilizando o GitLab. Para isso, foi criado um arquivo de configuração chamado `.gitlab-ci.yml`, mostrado na Figura 26, que define as etapas do pipeline, mostradas na Figura 27, como compilação do projeto e execução dos testes automatizados. Com essa configuração, a cada novo commit ou merge realizado na branch principal, o pipeline é automaticamente iniciado, garantindo que o código esteja sempre validado e funcional. Além disso, o pipeline serve como uma documentação executável do processo de build (Figura 28) e testes (Figura 29), tornando o fluxo de desenvolvimento mais transparente e confiável.

Figura 26 – Trecho do arquivo `.gitlab-ci.yml` com as configurações do pipeline.

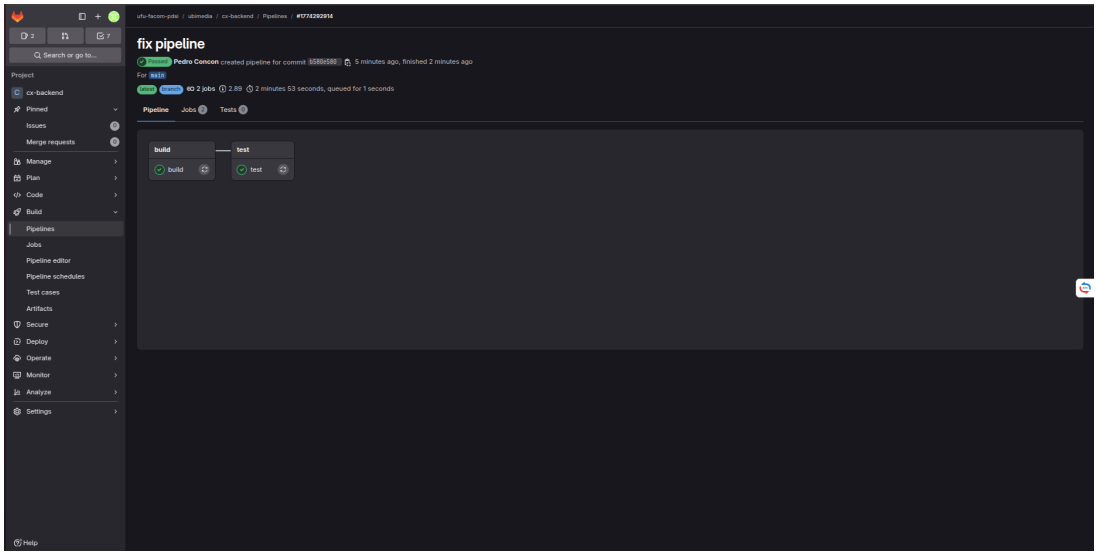

Figura 27 – Etapas do pipeline de CI configurado no GitLab.

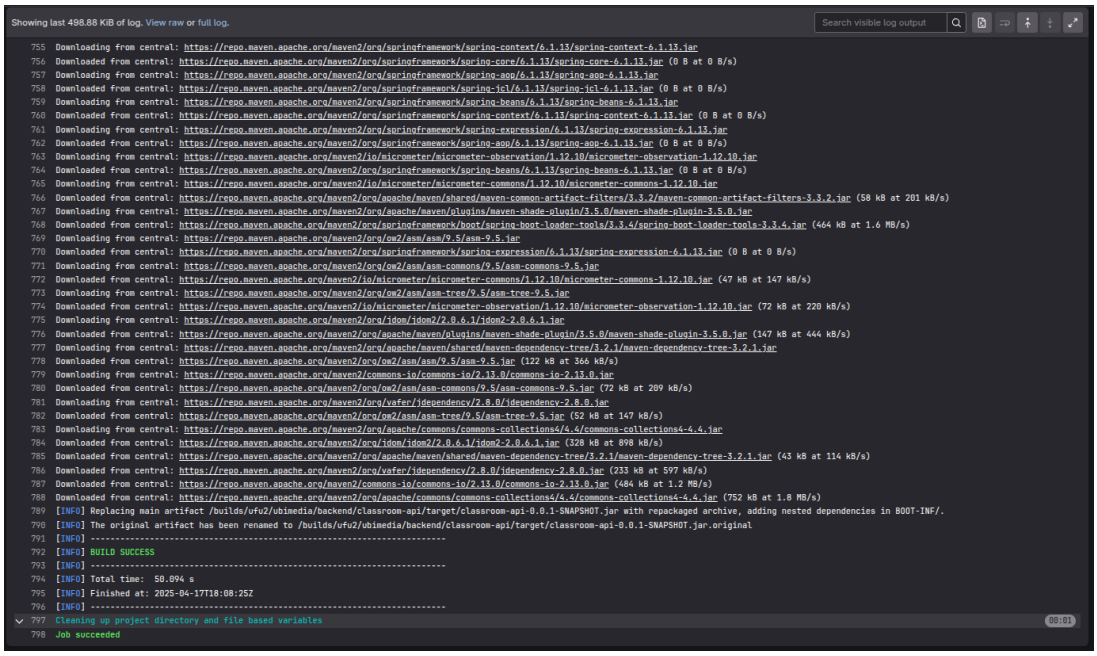

Figura 28 – Log da execução do build realizado com sucesso no GitLab CI.

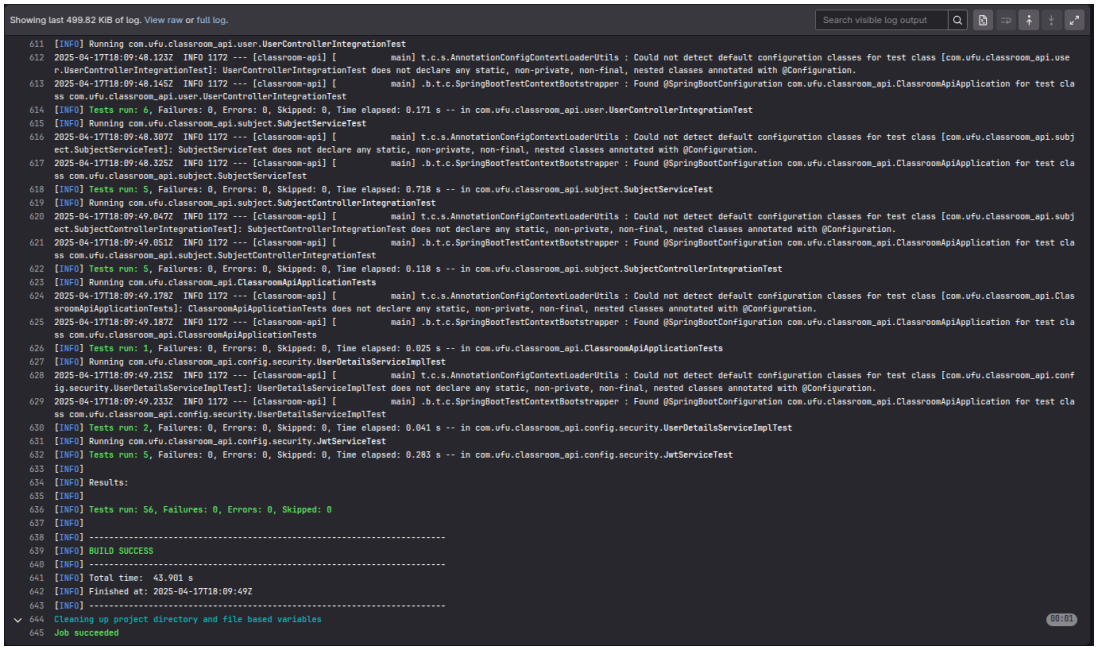

Figura 29 – Resultado dos 56 testes automatizados executados com sucesso.

5 Conclusão

Este trabalho teve como objetivo desenvolver uma solução back-end moderna e robusta para um sistema educacional, utilizando tecnologias consolidadas como Java, Spring Boot, PostgreSQL e Docker. Durante o desenvolvimento, foram abordados conceitos fundamentais relacionados a arquitetura de software, desenvolvimento web, autenticação e segurança, testes automatizados, integração contínua e boas práticas de desenvolvimento. A proposta visou oferecer uma base sólida para aplicações escaláveis e manuteníveis, garantindo flexibilidade para evoluções futuras.

A construção da API foi realizada com foco na separação de responsabilidades, reutilização de componentes e observabilidade do sistema. A escolha das tecnologias foi guiada pela estabilidade, suporte da comunidade e compatibilidade entre os recursos, garantindo assim uma solução eficiente e alinhada com os padrões do mercado.

5.1 Dificuldades encontradas

Durante o desenvolvimento do projeto, algumas dificuldades foram encontradas. Uma das principais foi a correta configuração dos testes automatizados, especialmente em cenários que envolviam autenticação com JWT e o uso de contextos de segurança do Spring Security. Além disso, a configuração da integração contínua com o GitLab CI exigiu ajustes nos ambientes de teste e no arquivo de configuração YAML Ain't Markup Language (YAML) para que o build fosse executado corretamente e os testes fossem validados automaticamente.

Outras dificuldades envolveram a configuração de ambientes isolados com Docker, ajustes finos no mapeamento das entidades no banco de dados e a simulação de contextos reais para testes com serviços que dependem de autenticação ou chamadas externas.

5.2 Resultados

Apesar dos desafios, os resultados obtidos foram bastante satisfatórios. A API desenvolvida atende aos requisitos definidos, oferece endpoints bem estruturados, documentação integrada com Swagger e testes unitários e de integração cobrindo os principais casos de uso. A integração com o GitLab CI permite que todo push ou merge dispare automaticamente a execução do pipeline de build e testes, garantindo qualidade contínua ao longo do desenvolvimento.

Além disso, a utilização de contêineres com Docker facilitou a criação de ambientes

reprodutíveis, tornando o deploy mais confiável e previsível. A estrutura modular da aplicação e o uso de boas práticas de desenvolvimento tornam o sistema extensível e de fácil manutenção.

5.3 Trabalhos Futuros

Como trabalhos futuros, é possível expandir a integração com ferramentas de monitoramento e observabilidade em tempo real (como Prometheus e Grafana), além da implementação de métricas e logs estruturados para acompanhamento contínuo da saúde da aplicação.

Outra frente importante para continuidade do projeto é a realização do deploy da aplicação em um ambiente de produção, incluindo a orquestração dos contêineres com ferramentas como Docker Compose ou Kubernetes. Essa etapa envolve desafios como a configuração de variáveis de ambiente sensíveis, controle de acesso, definição de escalabilidade e exposição segura da API. Além disso, a criação e manutenção de uma instância confiável do banco de dados PostgreSQL em produção exige atenção especial quanto à persistência de dados, segurança e estratégias de backup e recuperação.

Este projeto, portanto, não apenas alcançou os objetivos propostos, como também estabeleceu uma base sólida para futuras evoluções, tanto do ponto de vista técnico quanto acadêmico.

Referências

BASS, L. **Software architecture in practice**. Pearson Education India, 2012. Disponível em: <https://dl.acm.org/doi/abs/10.5555/773239>. Citado na página 16.

BERNERS-LEE, T.; CAILLIAU, R.; LUOTONEN, A.; NIELSEN, H. F.; SECRET, A. The world-wide web. In: **Linking the World's Information: Essays on Tim Berners-Lee's Invention of the World Wide Web**. [s.n.], 2023. p. 51–65. Disponível em: <https://doi.org/10.1145/3591366.3591373>. Citado na página 14.

BRODIE, M. L.; STONEBRAKER, M. Migrating legacy systems: gateways, interfaces & the incremental approach. **(No Title)**, 1995. Disponível em: <https://cir.nii.ac.jp/crid/1130000796385698432>. Citado na página 16.

CHOUDHURY, N. World wide web and its journey from web 1.0 to web 4.0. **International Journal of Computer Science and Information Technologies**, v. 5, n. 6, p. 8096–8100, 2014. Disponível em: <https://down.documentine.com/378117255549eb529d3ae4353f05766f.pdf>. Citado 2 vezes nas páginas 14 e 15.

CHURCH, A. **The calculi of lambda-conversion**. Princeton University Press, 1985. Disponível em: <https://doi.org/10.1515/9781400881932>. Citado na página 15.

DURELLI, R. S. Uma abordagem para criação, reúso e aplicação de refatorações no contexto da modernização dirigida a arquitetura. 2016. Disponível em: <https://bdtd.ibict.br/vufind/Record/USP_b0035bbaa6badf3dc1e8a27aaed31361>. Citado na página 21.

FERREIRA, H. N. M. et al. Captura multimídia em ambientes educacionais instrumentados: aspectos arquiteturais, modelo de comunicação e interface de acesso contextual. Universidade Federal de Uberlândia, 2012. Disponível em: <https://repositorio.ufu.br/handle/123456789/12525>. Citado na página 20.

FIELDING, R. T. **Architectural styles and the design of network-based software architectures**. University of California, Irvine, 2000. Disponível em: <https://www.proquest.com/openview/fc2d064044b971dda476dfb429a2b344/1?pq-origsite=gscholar&cbl=18750&diss=y>. Citado na página 18.

FOWLER, M. **Patterns of enterprise application architecture**. Addison-Wesley, 2012. Disponível em: <https://scholar.google.com.br/scholar?hl=pt-BR&as_sdt=0%2C5&q=FOWLER%2C+M.+Patterns+of+enterprise+application+architecture.+%5BS.l.%5D%3A+Addison-Wesley%2C+2012.&btnG=>. Citado na página 16.

GNOYKE, P.; SCHULZE, S.; KRÜGER, J. Evolution patterns of software-architecture smells: An empirical study of intra-and inter-version smells. **Journal of Systems and Software**, Elsevier, v. 217, p. 112170, 2024. Disponível em: <https://doi.org/10.1016/j.jss.2024.112170>. Citado na página 20.

LAURETIS, L. D. From monolithic architecture to microservices architecture. In: IEEE. **2019 IEEE International Symposium on Software Reliability**

Engineering Workshops (ISSREW). 2019. p. 93–96. Disponível em: <https://ieeexplore.ieee.org/abstract/document/8990350/>. Citado na página 17.

LEON, P. L.; HORITA, F. Modernização de arquiteturas de sistemas para suporte à transformação digital. In: SBC. **Simpósio Brasileiro de Sistemas de Informação (SBSI)**. 2020. p. 61–66. Disponível em: <https://doi.org/10.5753/sbsi.2020.13128>. Citado na página 21.

LIU, Z.-Y.; LOMOVTSEVA, N.; KOROBEYNIKOVA, E. Online learning platforms: Reconstructing modern higher education. **International Journal of Emerging Technologies in Learning (iJET)**, International Journal of Emerging Technology in Learning, v. 15, n. 13, p. 4–21, 2020. Disponível em: <https://doi.org/10.3991/ijet.v15i13.14645>. Citado na página 11.

MASSE, M. **REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces**. O'Reilly Media, 2011. ISBN 9781449319908. Disponível em: <https://books.google.com.br/books?id=eABpzyTcJNIC>. Citado na página 18.

NEWMAN, S. **Building microservices: designing fine-grained systems**. "O'Reilly Media, Inc.", 2021. Disponível em: <https://books.google.com.br/books?hl=pt-BR&lr=&id=ZvM5EAAAQBAJ&oi=fnd&pg=PT16&dq=NEWMAN,+S.+Building+microservices:+designing+fine-grained+systems.+%5BS.l.%5D:+%22O%E2%80%99Reilly+Media,+Inc.%22,+2021.&ots=ui8n6C7BQo&sig=c6S31rBNfLIa5vomE2IK-JQ0BDA#v=onepage&q&f=false>. Citado na página 16.

O'REILLY, T. **What is web 2.0**. "O'Reilly Media, Inc.", 2009. Disponível em: <https://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html>. Citado na página 14.

PRESSMAN, R. S. **Engenharia de Software: uma abordagem profissional**. 7. ed. São Paulo: McGraw-Hill, 2011. Disponível em: <https://books.google.com.br/books?hl=pt-BR&lr=&id=FSE3EAAAQBAJ&oi=fnd&pg=PT34&dq=PRESSMAN,+R.+S.+Engenharia+de+Software:+uma+abordagem+profissional.&ots=kAKSgUsCDH&sig=hktnU8jzQKH_mgJ6NL9YVZNo9lo#v=onepage&q=PRESSMAN%2C%20R.%20S.%20Engenharia%20de%20Software%3A%20uma%20abordagem%20profissional.&f=false>. Citado na página 24.

RAKSI, M. et al. **Modernizing web application: case study**. Dissertação (Mestrado), 2017. Disponível em: <https://aaltodoc.aalto.fi/items/1f03f049-5d3b-441d-ba30-777d7bbe2b54>. Citado na página 20.

RAWASHDEH, A. Z. A.; MOHAMMED, E. Y.; ARAB, A. R. A.; ALARA, M.; AL-RAWASHDEH, B. Advantages and disadvantages of using e-learning in university education: Analyzing students' perspectives. **Electronic Journal of E-learning**, v. 19, n. 3, p. 107–117, 2021. Disponível em: <https://doi.org/10.34190/ejel.19.3.2168>. Citado na página 11.

RIBEIRO, T. B. et al. Formalização e validação de aplicações de captura e acesso em ambientes educacionais ubíquos. Universidade Federal de Uberlândia, 2014. Disponível em: <https://repositorio.ufu.br/handle/123456789/12562>. Citado 2 vezes nas páginas 14 e 20.

RICHARDSON, C. **Microservices patterns: with examples in Java**. Simon and Schuster, 2018. Disponível em: <https://books.google.com.br/books?hl=pt-BR&lr=&id=QTgzEAAAQBAJ&oi=fnd&pg=PT21&dq=RICHARDSON,+C.+Microservices+patterns:+with+examples+in+Java.+%5BS.l.%5D:+Simon+and+Schuster,+2018&ots=95ea_EOFB9&sig=Jd8U08DxMvpJDIfOyWhpF4gn3H8#v=onepage&q&f=false>. Citado na página 18.

SANTOS, G. S. d. O. Análise de frameworks back-end escritos em java: uma perspectiva em critérios. 2023. Disponível em: <https://repositorio.ufc.br/handle/riufc/75542>. Citado na página 15.

SPILCA, L. **Spring security in action**. Manning, 2020. Disponível em: <https://books.google.com.br/books?hl=pt-BR&lr=&id=CF4BEAAAQBAJ&oi=fnd&pg=PA1&dq=SPILCA,+L.+Spring+security+in+action.&ots=_ULG3A1Xez&sig=Tf0XaNKHjs9P44k5XVXRExp7Ua4#v=onepage&q=SPILCA%2C%20L.%20Spring%20security%20in%20action.&f=false>. Citado na página 18.

THOMPSON, S. **Haskell: the craft of functional programming**. Addison-Wesley, 2011. Disponível em: <https://kar.kent.ac.uk/30749/1/craft3e.pdf>. Citado na página 15.

TUDOSE, C. **JUnit in action**. Manning, 2020. Disponível em: <https://books.google.com.br/books?hl=pt-BR&lr=&id=mhgIEAAAQBAJ&oi=fnd&pg=PR15&dq=TUDOSE,+C.+JUnit+in+action.+%5B&ots=g0il7M4at8&sig=x8SXbKRBOyBDF3rgSHCqbIuzkxc#v=onepage&q=TUDOSE%2C%20C.%20JUnit%20in%20action.%20%5B&f=false>. Citado na página 19.

ZAPALOWSKI, V. Análise quantitativa e comparativa de linguagens de programação. 2011. Disponível em: <https://lume.ufrgs.br/handle/10183/31036>. Citado na página 15.