UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS SOBRAL

CURSO DE ENGENHARIA DA COMPUTAÇÃO

CURSO DE GRADUAÇÃO EM ENGENHARIA DA COMPUTAÇÃO

FRANCISCO JAIRO ARAÚJO AGUIAR

APLICAÇÃO DE BOAS PRÁTICAS DE ENGENHARIA DE SOFTWARE NO DESENVOLVIMENTO DE UMA API REST PARA PLATAFORMA DE GESTÃO DE QUESTÕES E LISTAS DE EXERCÍCIOS

SOBRAL

2024

FRANCISCO JAIRO ARAÚJO AGUIAR

APLICAÇÃO DE BOAS PRÁTICAS DE ENGENHARIA DE SOFTWARE NO DESENVOLVIMENTO DE UMA API REST PARA PLATAFORMA DE GESTÃO DE QUESTÕES E LISTAS DE EXERCÍCIOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação do Campus Sobral da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia da Computação.

Orientador: Prof. Me. Fernando Rodrigues de Almeida Júnior.

SOBRAL

2024

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

A229a Aguiar, Francisco Jairo Araújo.
APLICAÇÃO DE BOAS PRÁTICAS DE ENGENHARIA DE SOFTWARE NO DESENVOLVIMENTO DE UMA API REST PARA PLATAFORMA DE GESTÃO DE QUESTÕES E LISTAS DE EXERCÍCIOS / Francisco Jairo Araújo Aguiar. – 2024.
65 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Sobral, Curso de Engenharia da Computação, Sobral, 2024.
Orientação: Prof. Me. Fernando Rodrigues de Almeida Júnior.

1. Plataforma Educacional. 2. API REST. 3. Engenharia de Software. 4. Desenvolvimento Orientado a Testes. 5. Arquitetura Limpa. I. Título.

CDD 621.39

FRANCISCO JAIRO ARAÚJO AGUIAR

APLICAÇÃO DE BOAS PRÁTICAS DE ENGENHARIA DE SOFTWARE NO DESENVOLVIMENTO DE UMA API REST PARA PLATAFORMA DE GESTÃO DE QUESTÕES E LISTAS DE EXERCÍCIOS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação do Campus Sobral da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia da Computação.

Aprovada em: 08/10/2024.

BANCA EXAMINADORA

Prof. Me. Fernando Rodrigues de Almeida Júnior (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dr. Wendley Souza da Silva
Universidade Federal do Ceará (UFC)

Prof. Dr. Iális Cavalcante de Paula Júnior
Universidade Federal do Ceará (UFC)

A Deus. À minha esposa, Thaís, e meus filhos,
Maria Júlia e João Guilherme e aos meus pais.
A todos vocês, minha eterna gratidão.

AGRADECIMENTOS

Agradeço à minha esposa, Thaís, por seu apoio incondicional e por estar ao meu lado em todos os momentos. Aos meus pais, pela motivação constante e por acreditarem em mim. E aos meus filhos, Maria Júlia e João Guilherme, por serem minha fonte diária de alegria e inspiração.

Agradeço à Universidade Federal do Ceará e aos meus professores por todo o conhecimento e dedicação durante minha formação. Em especial, ao Prof. Me. Fernando Rodrigues de Almeida Júnior, pela excelente orientação ao longo deste trabalho.

Também agradeço aos professores Dr. Iális Cavalcante de Paula Júnior e Dr. Wendley Souza da Silva, membros da banca examinadora, por seu tempo, valiosas colaborações e sugestões.

"A função de um bom software é fazer o complexo parecer simples." (Grady Booch. Atribuído a Booch em: Frank H. P. Fitzek et al. (2010). Qt for Symbian, p. xv)

RESUMO

Este trabalho aborda o desenvolvimento de uma API REST para uma plataforma de gestão de questões e listas de exercícios, enfatizando a aplicação de boas práticas de Engenharia de Software. A plataforma tem como objetivo fornecer uma ferramenta educacional flexível e intuitiva, permitindo a criação, administração e classificação de questões e listas de exercícios por meio de tags genéricas, como disciplina, tema e nível de dificuldade. A interface Web da plataforma se integra diretamente à API REST, concebida com uma Arquitetura Limpa e utilizando Desenvolvimento Orientado a Testes (TDD) para garantir a qualidade interna desde a fase inicial. Esta abordagem não só promove a flexibilidade e facilita a manutenção a longo prazo do sistema, mas também abre caminho para futuras integrações com diversas interfaces, incluindo aplicativos móveis, ampliando a versatilidade e usabilidade da plataforma. O estudo detalha não apenas a implementação técnica da API, mas também os processos e metodologias adotados para um desenvolvimento eficiente e sustentável, alinhado às exigências atuais do ambiente educacional e tecnológico.

Palavras-chave: plataforma educacional; API REST; Engenharia de Software; desenvolvimento orientado a testes; arquitetura limpa.

ABSTRACT

This paper explores the development of a REST API for a platform designed to manage questions and exercise lists, emphasizing the application of best practices in software engineering. The platform aims to provide a flexible and user-friendly educational tool, allowing users to create, manage, and categorize questions and exercise lists using generic tags like subject, topic, and difficulty level. The web interface is seamlessly integrated with the REST API, which is built on a Clean Architecture and employs Test-Driven Development (TDD) to ensure high-quality standards from the outset. This approach not only enhances flexibility and long-term maintainability but also sets the stage for future integrations with various interfaces, including mobile applications, thus increasing the platform's versatility and usability. The study covers not only the technical implementation of the API but also the processes and methodologies used to promote efficient and sustainable development, aligning with the current needs of the educational and technological landscape.

Keywords: educational platform; REST API; software engineering; test-driven development; clean architecture.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

API *Application Programming Interface* / Interface de Programação de Aplicação

JWT *JSON Web Token*

POC Prova de Conceito

REST *Representational State Transfer* / Transferência de Estado Representacional

SoC *Separação de Responsabilidades*

TDD *Test-Driven Development* / Desenvolvimento orientado a testes

XP *Extreme Programming* / Programacao Extrema

PDF *Portable Document Format*

SUMÁRIO

1 INTRODUÇÃO

A evolução da tecnologia ao longo das últimas décadas tem desempenhado um papel essencial na transformação do panorama educacional. Ela tem possibilitado a criação de novas formas de ensino e aprendizagem, tornando o processo educacional mais eficiente, eficaz e acessível. Para Kenski (KENSKI, 2007, p. 46), "Não há dúvida de que as novas tecnologias de comunicação e informação trouxeram mudanças consideráveis e positivas para a educação".

Inicialmente, os *softwares* educacionais surgiram com soluções simples, concentradas na criação de conteúdo educativo e no gerenciamento de informações acadêmicas. No entanto, com o decorrer do tempo, esses *softwares* evoluíram de maneira significativa, tornando-se ferramentas abrangentes, desde sistemas de gestão de aprendizagem até plataformas completas dedicadas ao ensino, como o Google Classroom.

No âmbito educacional, a criação de um serviço de gestão de questões e listas de exercícios voltado para professores representa um avanço significativo na eficiência e eficácia do processo pedagógico. Este serviço, que atuará como um banco de questões, tem como propósito oferecer aos educadores uma ferramenta dinâmica, intuitiva e personalizável, possibilitando-lhes criar, buscar e gerenciar questões de exercícios de forma simplificada, otimizando a gestão pedagógica e adaptando o sistema para atender às necessidades específicas de cada educador e disciplina.

O serviço proposto fornecerá funcionalidades para a criação de questões e listas de exercícios personalizadas, que poderão ser categorizadas de acordo com critérios diversos, como disciplina, tema ou nível de dificuldade, por meio de *tags* genéricas. O uso dessas *tags*, além de favorecer a organização, tornará a busca mais eficiente, otimizando o processo de criação de avaliações ou listas de exercícios.

Dada a complexidade e a necessidade de adaptação contínua das ferramentas educacionais modernas, é essencial que o desenvolvimento desse serviço siga boas práticas de Engenharia de Software, garantindo não apenas funcionalidades eficientes, mas também um *software* robusto e sustentável. A qualidade de um *Software* pode ser dividida em interna e externa: a interna, percebida principalmente por arquitetos, desenvolvedores e testadores, abrange o código-fonte, a documentação interna e as questões de arquitetura, sendo vital para a efetividade e longevidade do *software*, influenciando diretamente a facilidade de manutenção, a correção de *bugs* e a incorporação de novas funcionalidades. Por outro lado, a qualidade externa refere-se a atributos percebidos pelos usuários, como usabilidade, eficiência, confiabilidade e

funcionalidade, afetando diretamente a aceitação e a satisfação do usuário (PLöSCH R., 2008).

O *TDD* destaca-se como uma prática ágil que visa aprimorar a qualidade interna do software. Segundo Kent Beck (BECK, 2002), *TDD* é uma prática de desenvolvimento de software onde os testes de unidade automatizados são escritos de forma incremental, antes mesmo do código-fonte ser implementado. Uma revisão sistemática conduzida por BISSI et al. em 2016, analisando os efeitos do *TDD* na qualidade interna e externa do software, revelou que 76% dos estudos indicaram melhorias substanciais na qualidade interna, enquanto 88% destacaram aumento expressivo na qualidade externa, demonstrando o impacto positivo dessa abordagem em atributos percebidos pelos usuários (BISSI *et al.*, 2016). Esses achados corroboram a eficácia do *TDD* não apenas na promoção da robustez e eficiência interna do software, mas também na elevação dos padrões de qualidade observados e apreciados pelos usuários finais.

Frente à complexidade dos *softwares* modernos, que devem não apenas atender a requisitos funcionais, mas também adaptar-se eficientemente a mudanças contínuas, a Arquitetura de Software desempenha papel crucial. Nesse contexto, surge a Arquitetura Limpa, uma abordagem que emprega o princípio de inversão de dependência para separar componentes de alto nível daqueles de baixo nível (MARTIN, 2017). Essa arquitetura proporciona uma separação clara entre os componentes do sistema, facilitando a independência entre as camadas e a manutenibilidade ao longo do ciclo de vida do *software*.

1.1 Objetivos

O objetivo geral deste trabalho é desenvolver uma *API REST* para uma plataforma de gestão de questões e listas de exercícios, aplicando boas práticas de Engenharia de Software, como *TDD* e Arquitetura Limpa. Ao integrar essas práticas, buscamos criar uma ferramenta eficiente para educadores e demonstrar como essas abordagens melhoram a qualidade do software, facilitando sua evolução e manutenção.

1.2 Objetivos específicos

1. Planejar e implementar rotas da *API REST* para suportar funcionalidades básicas esperadas de uma aplicação de gerenciamento de questões e listas e exercícios.
2. Organizar o *software* utilizando uma Arquitetura Limpa, promovendo a separação clara de responsabilidades e isolamento das regras de negócio.

3. Adotar a metodologia de *TDD* para garantir a confiabilidade e funcionalidade correta de cada componente desde o início do desenvolvimento.
4. Incorporar uma camada de segurança robusta, incluindo autenticação e autorização, para proteger a aplicação contra ameaças e garantir a integridade dos dados sensíveis.
5. Adotar boas práticas de programação e documentação, garantindo a manutenibilidade do código e sua fácil compreensão por futuros desenvolvedores.
6. Projetar uma aplicação *Web* de *POC* para validar a integração e demonstrar as funcionalidades da *API REST* desenvolvida.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 API

Uma *API*, abreviação de Interface de Programação de Aplicações, é definida como a interface para uma entidade de *software* reutilizável utilizada por vários clientes fora da organização de desenvolvimento e que pode ser distribuída separadamente do código do ambiente (ROBILLARD *et al.*, 2013).

Em termos mais simples, uma *API* permite que um *software* utilize funcionalidades de outro *software* de maneira padronizada. Ela especifica os métodos de comunicação, os formatos de dados aceitos e outros detalhes importantes para que os desenvolvedores possam criar aplicações que interagem de forma consistente com um determinado serviço, biblioteca ou plataforma.

As *API*s são essenciais para garantir a interoperabilidade entre sistemas, promovendo modularidade e reutilização de componentes, ao simplificar a integração e o desenvolvimento de novas funcionalidades por meio da abstração da complexidade interna do *software*. Por exemplo, uma *API REST* segue princípios arquiteturais específicos e utiliza operações HTTP (GET, POST, PUT, DELETE) para permitir a comunicação entre cliente e servidor, sendo amplamente adotada em soluções modernas devido à sua flexibilidade, escalabilidade e compatibilidade com diferentes plataformas.

2.2 Arquitetura x Modelo Arquitetural

2.2.1 Arquitetura

Arquitetura de Software refere-se à estrutura fundamental de um sistema de *software*. Ela envolve a organização dos componentes principais do sistema, suas inter-relações, as diretrizes de design e os princípios que regem a sua evolução ao longo do tempo. A arquitetura define como os componentes do sistema interagem entre si, como são distribuídos, e como lidam com aspectos como desempenho, escalabilidade, segurança, e manutenção.

Um exemplo de arquitetura de sistemas é a arquitetura em camadas, onde há uma clara separação entre a camada de apresentação, camada de lógica de negócios e camada de dados. Cada camada possui suas responsabilidades e interfaces definidas.

2.2.2 Modelo Arquitetural

Um modelo arquitetural é uma representação abstrata ou um conjunto de regras que descreve um estilo ou padrão específico para organizar a Arquitetura de Software. Ele fornece um guia ou uma abordagem específica para implementar a arquitetura. O modelo arquitetural define as restrições, os padrões de interação, os componentes envolvidos e as diretrizes para a implementação de sistemas de acordo com esse modelo.

O *REST* é um exemplo de modelo arquitetural, que especifica como sistemas distribuídos devem ser organizados e interagir seguindo princípios como a separação cliente-servidor, a ausência de estado nas interações, a utilização de interfaces uniformes, entre outros.

2.3 Modelo Arquitetural REST

O Modelo Arquitetural *REST*, proposto por Roy Fielding em sua tese de doutorado em 2000, emerge como um paradigma fundamental na Engenharia de Software, especialmente em ambientes distribuídos, como a *World Wide Web*. *REST* enfatiza a escalabilidade das interações de componentes, generalidade de interfaces, implantação independente de componentes e componentes intermediários para reduzir a latência de interação, reforçar a segurança e encapsular sistemas legados (FIELDING, 2000).

O modelo *REST* é delineado por seis restrições, fornecendo uma estrutura robusta para a concepção e implementação de sistemas distribuídos:

1. Arquitetura Cliente/Servidor: Estabelece a separação entre cliente e servidor, permitindo que eles evoluam independentemente. Tal independência facilita a interoperabilidade e escalabilidade do sistema.
2. Sem Estado (*Stateless*): A *API* trata cada nova requisição de forma independente, sem depender do contexto das solicitações anteriores. Isso promove uma abordagem simplificada e eficiente na gestão do estado do sistema.
3. Capacidade de armazenamento em cache: Todas as respostas da *API* são projetadas para serem passíveis de armazenamento em cache. Essa característica não apenas otimiza o desempenho, mas também reduz a carga nos servidores, promovendo uma experiência mais eficiente para o usuário.
4. Interface Uniforme: A arquitetura *REST* preconiza uma interface uniforme, na qual a *API* retorna dados em um formato padronizado e completamente

utilizável. Isso facilita a compreensão e interação entre os componentes do sistema, promovendo uma consistência vital para a integridade do modelo.

5. Sistema em Camadas: Introduz a ideia de camadas intermediárias entre o cliente e o servidor, permitindo que o servidor tenha múltiplos intermediários. Essas camadas, embora invisíveis para o cliente, colaboram para atender às solicitações de forma eficiente e transparente.
6. Código sob demanda: Esta restrição opcional oferece a possibilidade de incluir trechos de código na resposta da *API*, proporcionando uma flexibilidade adicional. No entanto, seu uso é discricionário, dependendo das necessidades específicas do sistema.

2.3.1 Escalabilidade de APIs REST:

A escalabilidade é fundamental para garantir que um sistema possa crescer e atender a um número cada vez maior de usuários e requisições sem comprometer seu desempenho ou confiabilidade. Ela envolve a capacidade de processar mais dados e operações, mantendo a integridade e a disponibilidade do serviço à medida que a demanda aumenta.

Tipos de Escalabilidade

1. Escalabilidade Vertical:
   - Consiste em aumentar os recursos de um único servidor (mais CPU, memória, disco etc.). Esse método é limitado pelo hardware disponível e pelo custo, já que eventualmente há um limite físico para o aumento de capacidade de um servidor.
   - Em *API*s *REST*, aumentar a capacidade de um único servidor pode melhorar o desempenho para um número maior de requisições, mas não resolve problemas de redundância e disponibilidade.
2. Escalabilidade Horizontal:
   - Adicionar mais servidores ou instâncias que rodem a *API* em paralelo para dividir a carga. É o método mais comum para sistemas distribuídos e *API*s *REST* que precisam lidar com grandes volumes de tráfego.
   - Um balanceador de carga distribui as requisições entre as instâncias, garantindo que cada uma processe apenas uma parte do tráfego total. Isso permite

que o sistema cresça conforme necessário, adicionando novas instâncias de forma dinâmica.

Como APIs REST Promovem Maior Escalabilidade

As *API*s *REST* promovem maior escalabilidade principalmente devido à sua arquitetura *stateless* (sem estado). Isso significa que cada requisição do cliente contém todas as informações necessárias para ser processada, sem depender de dados armazenados no servidor entre as requisições. Essa característica facilita a escalabilidade horizontal, permitindo que múltiplos servidores tratem as requisições de forma independente.

Além disso, *REST* utiliza padrões padronizados e protocolos leves como HTTP, o que simplifica o balanceamento de carga e a distribuição de tráfego entre diferentes instâncias da *API*. O uso de caching também melhora a eficiência, reduzindo a carga nos servidores e diminuindo os tempos de resposta.

Esses elementos juntos permitem que uma *API REST* escale de maneira eficiente conforme o aumento de usuários e requisições, mantendo desempenho e confiabilidade elevados.

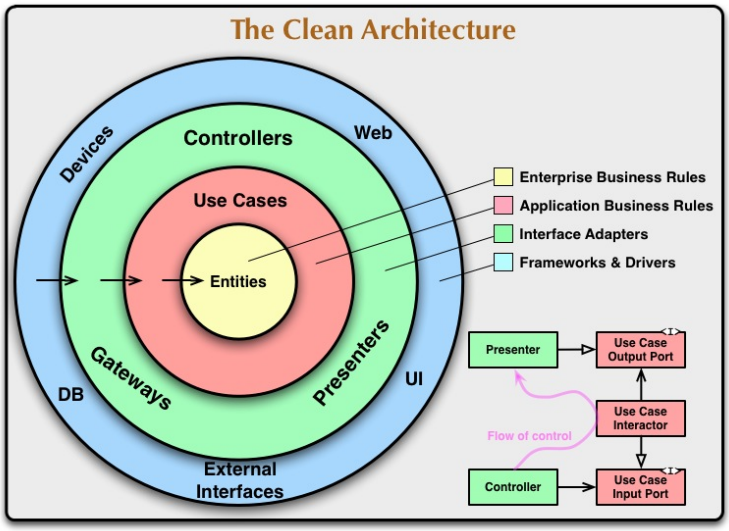
2.4 Arquitetura Limpa

A arquitetura limpa, do inglês *Clean Architecture*, desenvolvida por Robert C. Martin e detalhada em seu livro *"Clean Architecture: A Craftsman's Guide to Software Structure"*, é um modelo de organização de software que visa simplificar o desenvolvimento, implantação, manutenção e integração de regras de negócios (MARTIN, 2017). Ao aplicar a *Clean Architecture*, é possível dividir o *software* em camadas independentes, proporcionando uma arquitetura limpa, testável, escalável e de fácil manutenção.

As camadas são organizadas de forma concêntrica conforme a Figura 1, com as camadas mais internas abrigando políticas de alto nível, como regras de negócio, e as camadas mais externas concentrando-se nos detalhes de implementação. Essa disposição reflete a hierarquia de abstrações, onde o núcleo da aplicação, que contém as regras de negócio, é independente dos detalhes específicos das camadas externas.

Na *Clean Architecture*, as camadas de regras de negócio são mantidas completamente desacopladas de *frameworks*, tratando-os como ferramentas substituíveis sem impacto nas próprias regras de negócio. Essa abordagem também confere à aplicação uma testabilidade

Figura 1 – Conceito de Arquitetura Limpa



Fonte: Martin (2012).

eficiente, uma vez que torna possível a realização de testes isolados nas regras de negócios, mesmo na ausência de implementação nas camadas externas. Além disso, a aplicação mantém as regras de negócio independentes de interfaces de usuário, bancos de dados e quaisquer recursos externos (BOUKHARY; COLMENARES, 2019).

2.4.1 Regra da dependência

Segundo a regra da dependência, "as dependências do código-fonte devem apontar apenas para dentro, em direção a políticas de nível superior"(MARTIN, 2017). Isso implica que o código nas camadas internas não deve conhecer ou depender de funções, classes, variáveis ou qualquer entidade declarada nas camadas externas. Em outras palavras, as camadas internas devem ser totalmente independentes das camadas externas, não fazendo referência a nenhuma das suas implementações ou detalhes específicos.

2.5 Desenvolvimento Orientado a Testes

O desenvolvimento orientado a testes (*TDD*) desempenha um papel fundamental nas metodologias ágeis de desenvolvimento de *software*, sendo uma prática central no *Extreme Programming* / Programacao Extrema (*XP*) (BECK, 1999) e em diversas abordagens ágeis. *TDD* promove uma abordagem iterativa, onde o código é constantemente refinado e melhorado para atender aos requisitos específicos do cliente. A prática de TDD também ajuda a identificar problemas mais cedo no processo de desenvolvimento, o que pode economizar tempo e esforço

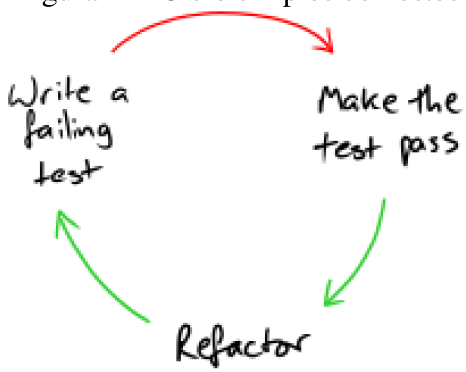
a longo prazo.

O papel que os testes desempenham no processo de desenvolvimento desafia as suposições da indústria de *software* sobre sua finalidade. Testar trata-se de ajudar a equipe a entender os recursos que os usuários precisam e a fornecer esses recursos de maneira confiável e previsível. Quando seguido até às suas conclusões, o *TDD* muda radicalmente a forma como desenvolvemos *software* e melhora drasticamente a qualidade dos sistemas que construímos, em particular a sua confiabilidade e a sua flexibilidade em resposta a novos requisitos (FREEMAN; PRYCE, 2009).

2.5.1 Ciclo do TDD

As etapas que compõem o ciclo do *TDD* são: escrever um teste para unidade isolada do código, escrever uma implementação mínima de código para fazer o teste passar e refatorar o código para tornar a implementação dos recursos testados a mais simples possível (FREEMAN; PRYCE, 2009). Essas etapas estão ilustradas na Figura 2. Note que as cores das setas indicam as etapas onde os testes falham em vermelho e onde passam em verde, por esse motivo esse ciclo é conhecido como ciclo *"Red-Green-Refactor"*.

Figura 2 – Ciclo simples de Testes



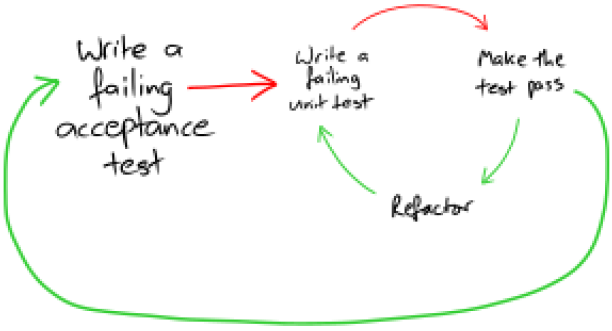
Fonte: Growing Object-Oriented Software by Freeman e Pryce (2009).

Ao iniciar o processo TDD, é comum a tentação de escrever testes unitários para classes na aplicação, capturando erros básicos de programação. Embora isso melhore a qualidade do código em unidades, projetos que dependem exclusivamente de testes unitários podem enfrentar desafios de integração no restante do sistema. A regra essencial é começar com um teste que falhe antes de implementar qualquer código (FREEMAN; PRYCE, 2009).

2.5.2 Testes de Aceitação

Quando se trata de implementar um recurso, a abordagem muda para testes de aceitação. Esses testes exercitam a funcionalidade desejada e, ao falharem, indicam a necessidade de implementação. O ciclo *"Red-Green-Refactor"* é então aplicado em nível de unidade, utilizando o teste de aceitação como guia (Figura 3). Esse processo garante que o código desenvolvido seja diretamente relevante para as exigências do sistema, evitando a criação de código desnecessário (FREEMAN; PRYCE, 2009).

Figura 3 – Ciclos de testes de Aceitação e de Unidade



Fonte: Growing Object-Oriented Software by Freeman e Pryce (2009).

3 DESENVOLVIMENTO

Conforme detalhado no Apêndice A, todo o código implementado para este projeto está disponível no repositório público do GitHub. Este repositório contém não apenas o código-fonte, mas também a documentação necessária para a execução e desenvolvimento contínuo da aplicação.

3.1 Escolha da Linguagem

A aplicação foi desenvolvida utilizando a linguagem Kotlin, uma linguagem multi-plataforma concisa, segura e moderna voltada para o desenvolvimento de aplicações escaláveis e de alto desempenho no lado do servidor. Kotlin minimiza significativamente a quantidade de código repetitivo necessário, o que resulta em um aumento da produtividade no desenvolvimento e na redução da probabilidade de ocorrência de erros.

Embora muito usada em aplicações móveis, também é uma ótima escolha para aplicações no lado do servidor, pois a linguagem possui total interoperabilidade com *frameworks* baseados em Java, permitindo a integração harmoniosa com o vasto ecossistema tecnológico existente e possibilitando a reutilização de várias bibliotecas (JETBRAINS, 2023).

3.2 Definições Iniciais

As definições iniciais estabelecem a base para o desenvolvimento e a utilização do software, garantindo que as funcionalidades atendam às necessidades dos usuários. Nessa sessão serão definidas as estrutura de usuários, as entidades principais e as funcionalidades de acesso e gerenciamento, visando garantir uma usabilidade otimizada.

3.2.1 Usuários e Papéis

O sistema categoriza os usuários em dois papéis distintos, cada um com permissões específicas que determinam seu nível de acesso e controle dentro da plataforma:

a) **Administrador (Admin):** Usuário com privilégios elevados, capaz de acessar todos os recursos do sistema. Os administradores podem gerenciar exercícios, listas de exercícios e usuários, incluindo outros administradores. Este papel é essencial para a manutenção, supervisão e configuração global da plataforma.

b) **Usuário Comum (User):** Usuário com permissões limitadas ao gerenciamento de seus próprios exercícios e listas de exercícios. Este papel é destinado a indivíduos que utilizam a plataforma para criar e organizar conteúdo de aprendizagem de forma personalizada.

3.2.2 Entidades Principais do Sistema

O software opera com duas entidades fundamentais que permitem a criação e organização de conteúdo educativo:

a) **Exercício:** Um exercício representa uma questão ou problema a ser resolvido e é composto pelos seguintes atributos:
   - **Título:** Identificação sucinta do exercício.
   - **Enunciado:** Descrição detalhada da questão proposta.
   - **Lista de Alternativas:** Conjunto de opções de resposta, com indicação clara da alternativa correta.
   - ***Tags*:** Palavras-chave que classificam e facilitam a busca e organização dos exercícios.
   - **Autor:** Usuário que criou o exercício, identificado como seu proprietário.

b) **Lista de Exercícios:** Uma lista de exercícios é um agrupamento de exercícios e possui as seguintes características:
   - **Título:** Identificação sucinta da lista.
   - **Conteúdo:** Conjunto de exercícios que podem ser adicionados ou removidos conforme a necessidade.
   - ***Tags*:** Palavras-chave que classificam e facilitam a busca e organização das listas.
   - **Autor:** Usuário que criou a lista, responsável por seu gerenciamento.

3.2.3 Políticas de Acesso e Controle

As políticas de acesso definem as permissões de visualização e manipulação das entidades dentro do sistema:

a) **Administradores:** Possuem acesso total a todos os exercícios e listas de exercícios, independentemente do autor. Podem também gerenciar usuários, incluindo outros administradores.

b) **Usuários Comuns:** Podem acessar e gerenciar apenas seus próprios exercícios e listas de exercícios. No entanto, as entidades de exercícios e listas possuem políticas de acesso configuráveis, permitindo futuras melhorias, como a possibilidade de tornar determinados exercícios ou listas públicos para outros usuários.

3.2.4 Funcionalidades de Indexação e Busca

Para melhorar a usabilidade e facilitar a navegação pelo conteúdo, o sistema oferece mecanismos avançados de indexação e busca:

a) **Exercícios e Listas de Exercícios:** Todos os usuários podem pesquisar exercícios utilizando termos genéricos e aplicar filtros baseados em *tags* para refinar os resultados da busca, visualizando apenas aqueles para os quais possuem permissão de acesso.

b) **Usuários:** Administradores têm a capacidade de listar e buscar usuários utilizando termos como nome de usuário ou e-mail, facilitando o gerenciamento de contas e permissões.

3.3 Implementando a Arquitetura Limpa

A arquitetura limpa, como apresentada no livro *Clean Architecture: A Craftsman's Guide to Software Structure* (MARTIN, 2017) é frequentemente representada com quatro camadas, mas isso não significa que essas camadas são fixas ou obrigatórias em todos os sistemas. O essencial é seguir a Regra de Dependência, onde as dependências sempre apontam para camadas de maior abstração.

Na aplicação desenvolvida, organizamos os componentes em três camadas: Domínio, Aplicação e Infraestrutura - ainda respeitamos esse princípio e alcançamos uma clara *Separação de Responsabilidades* (*SoC*). Embora o sistema possua múltiplas entidades, como *User* e *ExercisesList*, para facilitar o entendimento, focaremos na entidade *Exercise*, já que as demais seguem a mesma estrutura.

3.3.1 Camada de Domínio

Responsabilidades: Contém a lógica de negócio central e os modelos de domínio, independente de sistemas externos ou *frameworks*.

Componentes:

1. **Exercise**: Representa o objeto de negócio central para exercícios dentro do sistema. Encapsula atributos e comportamentos essenciais.
2. **ExerciseRepository <Interface>**: Define o contrato para operações de persistência de dados relacionadas aos exercícios, abstraindo a camada de acesso a dados.
3. **Requester**: Representa o usuário do sistema que pode acessar dados ou executar ações, contendo informações de identidade e papel.
4. **PermissionValidator**: Contém a lógica para validar se um usuário do sistema tem as permissões necessárias para realizar certas ações em uma entidade. Garante que as verificações de permissão façam parte da lógica de negócio central.

3.3.2 Camada de Aplicação

Responsabilidades: Orquestra regras de negócio específicas da aplicação e casos de uso, atuando como um mediador entre as camadas de domínio e infraestrutura.

Componentes:

1. **ExerciseController**: Serve como coordenador principal para lidar com casos de uso relacionados aos exercícios. Implementa as operações da aplicação (casos de uso) de forma simplificada. Depende de:
   - **Exercise**: Utiliza o modelo de domínio para realizar operações.
   - **ExerciseRepository**: Interage com a interface de persistência de dados para recuperar e armazenar instâncias da *Exercise*.
   - **Requester**: Identifica quem está fazendo a requisição para garantir o tratamento adequado.
   - **PermissionValidator**: Valida as permissões antes de executar operações para reforçar as restrições de segurança.

3.3.3 Camada de Infraestrutura

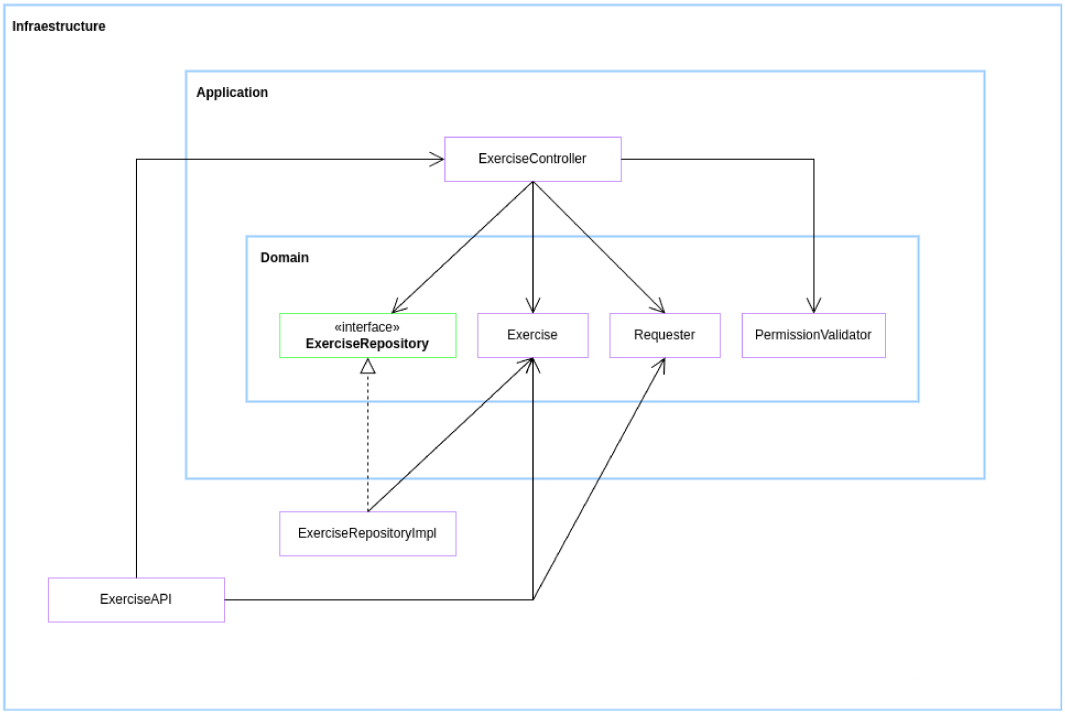
Responsabilidades: Contém os detalhes de implementação para sistemas externos, como bancos de dados, interfaces de usuário e *API*s. Interage com *frameworks* e ferramentas, fornecendo implementações concretas para interfaces definidas nas camadas internas.

Componentes:

1. **ExerciseAPI**: Expõe a funcionalidade da aplicação através de uma *API* via *endpoints REST*. Lida com requisições de entrada e envia respostas de volta ao cliente. Depende de:
   - **Requester**: Extrai informações do solicitante da requisição de entrada a partir do token de autenticação.
   - **Exercise**: Usada para converter objetos de exercício em formatos apropriados para requisições e respostas da *API*, permitindo que os dados sejam apresentados corretamente aos clientes.
   - **ExerciseController**: Delegar requisições de entrada ao controlador para processamento.
2. **ExerciseRepositoryImpl**: Uma implementação concreta da interface *ExerciseRepository* que lida com a lógica real de acesso a dados. É onde ficam os detalhes específicos relacionados a tecnologia de persistência adotada. Depende de:
   - **Exercise**: Mapeia registros do banco de dados para entidades de domínio e vice-versa.

A Figura 4 mostra um diagrama com as principais classes relacionadas a entidade *Exercise* organizadas em camadas.

Figura 4 – Camadas do Projeto



Fonte: elaborada pelo autor.

3.3.4 Fluxo de Dependência e Princípios da Arquitetura Limpa

1. **Regra de Dependência**: Todas as dependências apontam para dentro. Camadas externas podem depender de camadas internas, mas as camadas internas permanecem inconscientes das camadas externas.
   - A **Camada de Infraestrutura** depende da **Camada de Aplicação** e da **Camada de Domínio**.
   - A **Camada de Aplicação** depende da **Camada de Domínio**.
   - A **Camada de Domínio** não depende de nada externo a si mesma.
2. **Adaptadores de Interface**: O *ExerciseRepositoryImpl* na Camada de Infraestrutura implementa a interface *ExerciseRepository* da Camada de Domínio. Isso permite que a Camada de Domínio permaneça abstrata em relação à implementação real de acesso a dados.
3. **Abstração sobre Implementação**: Ao depender de interfaces, como *ExerciseRepository*, em vez de classes concretas, o sistema adere ao Princípio da Inversão de Dependência. Módulos de alto nível (Camada de Domínio) não dependem de módulos de baixo nível (Camada de Infraestrutura); ambos dependem de abstrações.
4. **Separação de Responsabilidades**: Cada camada tem uma responsabilidade clara, reduzindo o acoplamento e aumentando a coesão. As regras de negócio são isoladas na Camada de Domínio, a lógica da aplicação na Camada de Aplicação e as interações com sistemas externos na Camada de Infraestrutura.

3.3.5 Fluxo de uma Requisição Típica

Uma **requisição típica** refere-se a um fluxo padrão de processamento que uma solicitação (*request*) percorre dentro de um sistema ou aplicação. Nesse contexto específico, uma requisição típica descreve as etapas sequenciais que ocorrem desde o momento em que um cliente envia uma solicitação para a *API* (*ExerciseAPI*) até a geração e envio da resposta de volta ao cliente.

1. **Requisição de Entrada**:
   - Um cliente envia uma requisição para a *ExerciseAPI*.
   - A *ExerciseAPI* extrai as informações do *Requester* e quaisquer dados necessários

para processar a requisição.

2. **Delegação ao Controlador**:
   - A *ExerciseAPI* encaminha a requisição para o *ExerciseController*, passando o *Requester* e quaisquer dados necessários.
3. **Validação de Permissão**:
   - O *ExerciseController* usa o *PermissionValidator* para garantir que o *Requester* tenha as permissões necessárias.
4. **Execução da Lógica de Negócio**:
   - Após a validação bem-sucedida de permissão, o *ExerciseController* interage com o *Exercise*.
   - Pode usar o *ExerciseRepository* para recuperar ou armazenar instâncias de *Exercise*.
5. **Persistência de Dados**:
   - O *ExerciseController* chama métodos na interface *ExerciseRepository*.
   - O *ExerciseRepositoryImpl* (Camada de Infraestrutura) interage com o sistema de armazenamento adotado para lidar com operações de dados.
6. **Geração de Resposta**:
   - Os resultados são passados de volta para a *ExerciseAPI*, que formata a resposta ao cliente.

Por outro lado, uma **requisição não típica** refere-se a uma solicitação que desvia do fluxo padrão estabelecido para o processamento de requisições típicas dentro de um sistema ou aplicação. Enquanto uma requisição típica segue uma sequência previsível e linear de etapas para ser processada, uma requisição não típica envolve situações excepcionais ou condições especiais que exigem tratamento diferenciado. Por exemplo, se um cliente envia dados inválidos ou incompletos para uma operação, a requisição não típica resultante necessitará de validações adicionais e respostas de erro apropriadas.

3.4 Implementação da Persistência de Dados

A persistência de dados no sistema foi realizada utilizando o padrão de repositório, que abstrai as operações de acesso aos dados e promove uma separação clara entre a lógica de negócio e mecanismos de persistência (FOWLER *et al.*, 2002). Para a entidade *Exercise*, a classe principal responsável por essa implementação é a *JdbcExerciseRepository*, que utiliza o *JdbcTemplate* do *Spring Framework* para interagir com o banco de dados relacional MySQL e a

biblioteca *Gson* para a serialização e desserialização de objetos complexos em formato JSON.

A classe *JdbcExerciseRepository* implementa a interface *ExerciseRepository*, fornecendo métodos para operações de criação, leitura, atualização e exclusão (CRUD), além de funcionalidades adicionais como busca e recuperação de *tags* associadas aos exercícios. O uso do *JdbcTemplate* simplifica a execução de comandos SQL parametrizados, assegurando a proteção contra injeção de SQL e gerenciando conexões e recursos de forma eficiente (FRAMEWORK, 2024).

3.4.1 Serialização de Dados Complexos

Para acomodar a estrutura complexa dos dados da entidade *Exercise*, como descrições detalhadas e listas de respostas possíveis, foi utilizada uma coluna do tipo JSON na tabela exercises. A biblioteca *Gson* é empregada para converter objetos Java em representações JSON e vice-versa, permitindo armazenar e recuperar essas informações de maneira estruturada e flexível. Essa abordagem facilita a evolução do modelo de dados, permitindo a inclusão de novos campos sem a necessidade de alterações frequentes no esquema do banco de dados.

3.4.2 Persistência de Outras Entidades

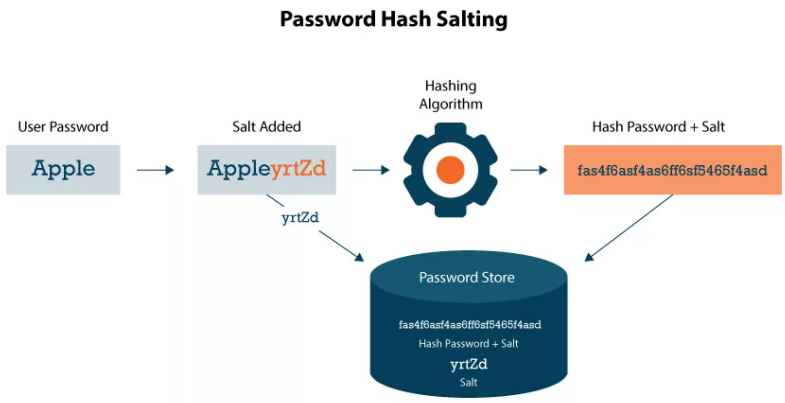
A mesma abordagem utilizada para a persistência da entidade *Exercise* foi aplicada a outras entidades do sistema, como *User* e *ExerciseList*. Essa consistência na implementação promove a reutilização de código, facilita a manutenção e assegura que diferentes partes do sistema sigam padrões semelhantes de design e arquitetura.

3.4.3 Hash Criptográfico sobre Dados Sensíveis

Na aplicação desenvolvida, a segurança das senhas dos usuários é garantida por meio de *hashing*, uma criptografia unidirecional que transforma a senha em texto plano em um *hash* irreversível. Utilizamos o algoritmo Argon2id, escolhido por sua resistência a ataques de hardware e de canal lateral, além de ser parametrizável para ajustar o uso de memória e desempenho (GREGóRIO; GOYA, 2019).

O processo de geração de hash ocorre ao criar ou alterar uma senha. O algoritmo Argon2id aplica um *salt* - um valor único e aleatório - à senha, gerando um hash que é armazenado no banco de dados (Figura 5). Isso garante que senhas iguais resultem em hashes diferentes.

Figura 5 – *Password Hash Salting*



Fonte: Mehta (2024).

No login, a senha fornecida pelo usuário é novamente processada e transformada em um hash para ser comparado com o hash armazenado. Se os hashes coincidirem, o usuário é autenticado. Essa abordagem protege as senhas contra vazamentos, pois nunca são armazenadas ou transmitidas em texto plano, assegurando conformidade com as melhores práticas de segurança.

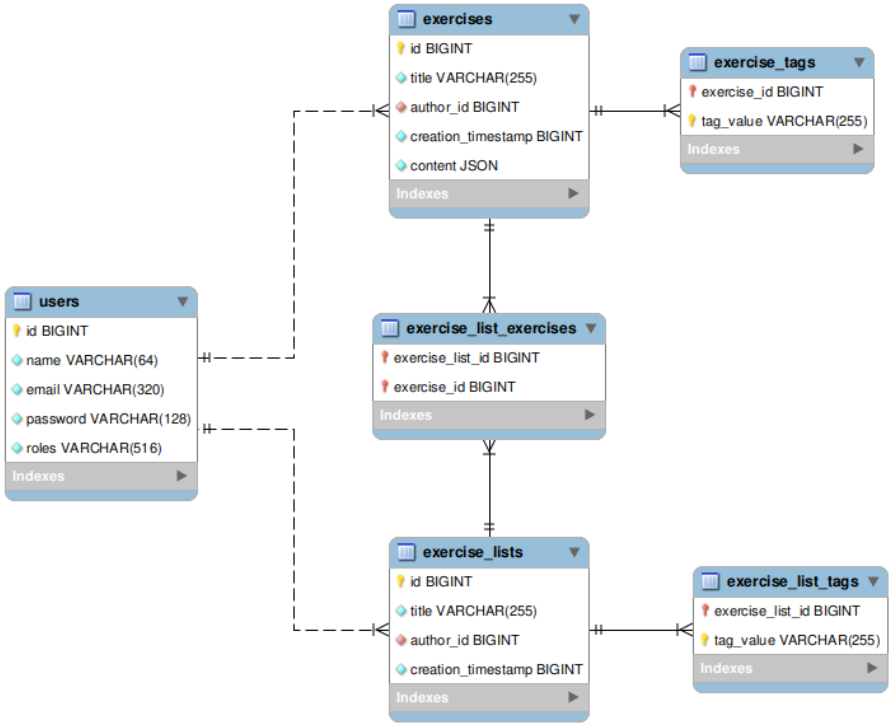
3.4.4 Controle de Versão do Esquema com Liquibase

O gerenciamento do esquema do banco de dados foi realizado utilizando o Liquibase, uma ferramenta de controle de versão para bancos de dados (LIQUIBASE, 2024). Isso possibilita a aplicação automatizada e consistente de mudanças no esquema, garantindo que diferentes ambientes de desenvolvimento, teste e produção mantenham a mesma estrutura de dados. Os arquivos de *changeset* definem a criação das tabelas e a configuração das constraints necessárias, como chaves primárias e estrangeiras, assegurando a integridade e a performance das operações de banco de dados.

Para declarar e gerenciar essas mudanças, foi utilizado o formato YAML nos *changesets* do Liquibase. O uso de arquivos YAML oferece uma sintaxe mais legível e concisa em comparação com outros formatos, como XML, facilitando a manutenção e a compreensão das mudanças realizadas no esquema. Cada *changeset* em YAML especifica uma operação a ser executada, como a criação de uma tabela ou a modificação de uma coluna, permitindo um controle granular das alterações.

O *changeset* YAML, que declara o esquema banco, pode ser visualizado no Apêndice B. A seguir, apresenta-se o esquema visual do banco de dados na Figura 6:

Figura 6 – Esquema do Banco da Dados



Fonte: elaborada pelo autor.

3.4.5 Testes de Integração com o Banco de Dados Utilizando Testcontainers

No projeto, utilizou-se a biblioteca Testcontainers (COMMUNITY, 2024) para garantir a qualidade dos testes de integração das classes que interagem com o banco de dados. Essa biblioteca permite rodar containers Docker no ambiente de testes, simulando um banco de dados real sem depender de um banco externo, assegurando testes consistentes e isolados.

A configuração da fonte de dados foi adaptada para usar um container MySQL durante os testes, como mostrado na classe *DataSourceTestContext* da Figura 7, que substitui o banco de dados real usado em produção.

Nesta configuração, o container MySQL é iniciado antes dos testes, e o Liquibase é utilizado para aplicar as mudanças necessárias para preparar o esquema do banco de dados. O container é destruído ao término dos testes, garantindo que o ambiente seja limpo após cada execução.

Os testes de integração foram implementados utilizando o JUnit 5. Eles verificam o comportamento correto das classes de repositório ao interagir com o banco de dados. Alguns exemplos de testes incluem a verificação de inserção, consulta, atualização e deleção de registros no banco de dados:

Figura 7 – Classe de Configuração da Fonte de Dados

```
@Configuration
open class DataSourceTestContext {
    private var mySQLContainer = MySQLContainer<Nothing>(
        DockerImageName.parse( fullImageName: "mysql:8.0")
    ).apply {
        withDatabaseName( databaseName: "exercises")
        withUsername( username: "root")
        withPassword( password: "root")
        start()
    }

    @Bean
    open fun springLiquibase(): SpringLiquibase {
        val springLiquibase = SpringLiquibase()
        springLiquibase.dataSource = dataSource()
        springLiquibase.changeLog = "classpath:db/changelog.yml"
        springLiquibase.isDropFirst = false
        return springLiquibase
    }

    @Bean
    open fun dataSource(): DataSource {
        val dataSource = DriverManagerDataSource()
        dataSource.setDriverClassName("com.mysql.jdbc.Driver")
        dataSource.url = mySQLContainer.jdbcUrl + "?allowPublicKeyRetrieval=true&useSSL=false"
        dataSource.username = mySQLContainer.username
        dataSource.password = mySQLContainer.password
        return dataSource
    }

    @Bean(destroyMethod = "stop")
    open fun stopMySQLContainer(): MySQLContainer<*> {
        return mySQLContainer
    }
}
```

Fonte: elaborada pelo autor.

No teste da Figura 8, um novo exercício é adicionado ao banco de dados simulado pelo container MySQL. Após a inserção, o teste verifica se o id foi inicializado corretamente e depois recupera o exercício e verifica se os dados persistidos correspondem aos dados esperados. Esse padrão se repete para outros casos de uso, como atualização, deleção e busca com filtros.

Figura 8 – Teste de Inserção de Dados no Banco

```
    @Test
    fun `add inserts exercise when it does not exist`() {
        val exercise = ExerciseFixture.anyExercise().copy(id = ExerciseId( value: null))

        val result = repository.add(exercise)

        assertNotNull(result.id)
        assertEquals(exercise.metadata.title, result.metadata.title)
        assertEquals(exercise.metadata.authorId, result.metadata.authorId)
        assertEquals(exercise.metadata.tags, result.metadata.tags)
        assertEquals(exercise.content, result.content)

        val insertedExercise = repository.get(result.id)
        assertEquals(result.copy(id = insertedExercise.id), insertedExercise)
    }
```

Fonte: elaborada pelo autor.

3.5 Implementação de APIs

O desenvolvimento da *API* seguiu um processo estruturado, utilizando boas práticas de design e implementação de rotas *REST* para interações com as entidades da aplicação. No caso da entidade *Exercise*, foi implementada uma classe responsável pela declaração de rotas específicas chamada *ExerciseRoutesDeclaration*. Esta classe define os *endpoints* necessários para criação, busca, atualização, exclusão e pesquisa de exercícios, todos com os devidos métodos HTTP (POST, GET, PUT e DELETE), garantindo a adesão aos padrões *REST*.

Durante a implementação, a autenticação foi tratada com um filtro de autorização, que se aplica a todas as rotas declaradas. Esse filtro valida a presença e autenticidade de tokens de autorização, permitindo que somente usuários autenticados possam interagir com os recursos protegidos. O filtro foi aplicado às rotas de exercício através do método ***authFilter.applyTo*** (Figura 9).

Figura 9 – Declaração de rotas da entidade *Exercise*

```
override fun declareRoutes(service: Service) {
    authFilter.applyTo(service, BASE_ENDPOINT)
    authFilter.applyTo(service, path: "$BASE_ENDPOINT/*")
    service.post(BASE_ENDPOINT, this::createExercise)
    service.get( path: "$BASE_ENDPOINT/:id", this::getExercise)
    service.put( path: "$BASE_ENDPOINT/:id", this::updateExercise)
    service.delete( path: "$BASE_ENDPOINT/:id", this::deleteExercise)
    service.post( path: "${BASE_ENDPOINT}/search", this::search)
}
```

Fonte: elaborada pelo autor.

Cada *endpoint* foi implementado de forma a extrair os dados necessários do corpo da requisição ou dos parâmetros de URL, validando adequadamente os dados antes de encaminhá-los ao controlador responsável (*ExerciseController*). Por exemplo, o método *createExercise* extrai os dados de criação da requisição, valida os valores passados e, em seguida, chama o método correspondente no controlador para persistir o novo exercício.

As *API*s implementadas estão devidamente documentadas, contendo informações sobre seus *endpoints*, métodos, tipos de dados aceitos, possíveis respostas e códigos de erro. A documentação completa pode ser consultada no Apêndice C deste trabalho.

3.5.1 Testes Automatizados de APIs

O desenvolvimento da API seguiu a metodologia de Desenvolvimento Orientado a Testes (*TDD*), assegurando que cada funcionalidade fosse testada de forma rigorosa antes de ser completamente integrada ao código da aplicação.

Por exemplo, o teste de criação de exercícios valida que, ao fornecer os dados corretos e um token de autorização válido, o *endpoint* responde com o código de status *201 Created* (Figura 10). Testes adicionais foram implementados para verificar casos de falhas, como a falta de um token de autorização, que resulta em um retorno de *401 Unauthorized* (Figura 11), e a tentativa de acesso a um exercício inexistente, que retorna *404 Not Found* (Figura 12).

Figura 10 – Teste de criação de exercícios com sucesso

```
@Test
fun `creates a new exercise successfully`() {
    val requestBody = """{"title": "Exercise title", "description": "This is the exercise description",
        |"tags": [{"value": "value_1"}], "possibleAnswers": ["Option A", "Option B", "Option C"],
        |"correctAnswerIndex": 1}""".trimMargin()
    every { exerciseController.create(any(), requester) } returns ExerciseFixture.anyExercise()

    given().contentType(ContentType.JSON)
        .header("Authorization", "Bearer valid-token")
        .body(requestBody)
    .`when`().post("/exercise")
        .then().assertThat()
        .statusCode(201)
}
```

Fonte: elaborada pelo autor.

Figura 11 – Teste de criação de exercícios sem token de autorização

```
@Test
fun `create exercise request with missing authorization token`() {
    val requestBody = """{"title": "Exercise title", "description": "This is the exercise description",
        |"tags": [{"value": "value_1"}], "possibleAnswers": ["Option A", "Option B", "Option C"],
        |"correctAnswerIndex": 1}""".trimMargin()

    given().contentType(ContentType.JSON)
        .body(requestBody)
    .`when`().post("/exercise")
    .then().assertThat()
        .statusCode(401)
        .contentType(ContentType.JSON)
        .body("message", equalTo( operand: "Unauthorized"))
}
```

Fonte: elaborada pelo autor.

Esses testes cobrem não apenas o sucesso das operações, mas também diversos cenários de erro, assegurando que a *API* se comporte de maneira robusta e previsível em diferentes situações.

Figura 12 – Teste de tentativa de acesso a exercício inexistente

```
@Test
fun `get exercise request with non existing exercise id`() {
    every {
        exerciseController.get(ExerciseId( value: "1"), requester)
    } throws NoSuchElementException("Exercise not found")

    given()
        .header("Authorization", "Bearer valid-token")
    .`when`().get("/exercise/1")
    .then().assertThat()
        .statusCode(404)
        .contentType(ContentType.JSON)
        .body("message", equalTo( operand: "Exercise not found"))
}
```

Fonte: elaborada pelo autor.

3.6 Implementando a Autenticação e Autorização

No contexto de uma *API REST*, uma das abordagens mais seguras e escaláveis para gerenciar autenticação e autorização de usuários é por meio do uso de *JWT*. Esta seção explica o fluxo de autenticação e autorização baseado em *JWT* adotado na *API*, incluindo uma explicação passo a passo do processo em que o usuário faz login, recebe um token de sessão e o utiliza para acessar recursos protegidos.

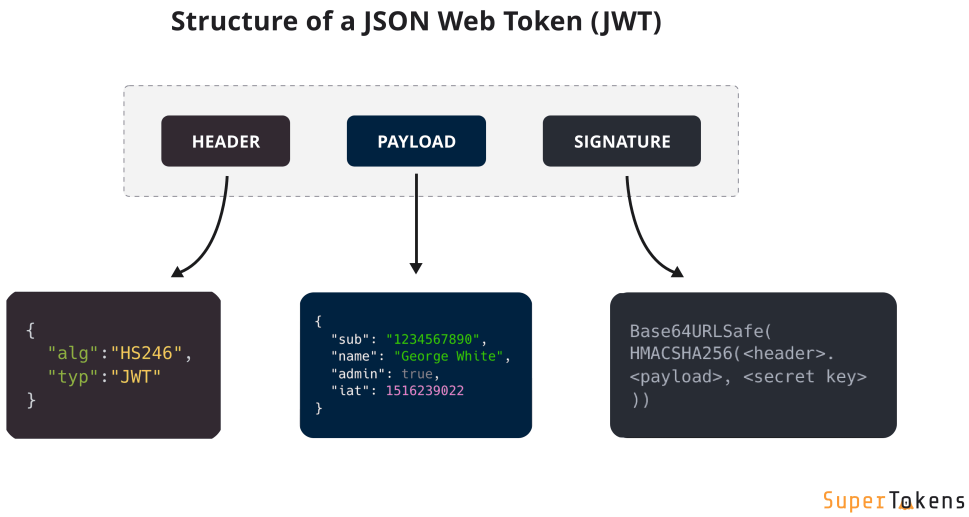
3.6.1 O que é JWT ?

JWT é um formato de token compacto e autocontido que pode ser usado para transmitir informações com segurança entre partes na forma de um objeto JSON (AUTH0, 2023). Ele é amplamente adotado em *API*s *REST* para autenticação sem estado, o que significa que nenhum dado de sessão é armazenado no servidor (RAI, 2021). Em vez disso, o cliente mantém o *JWT*, que inclui informações codificadas sobre o usuário, como as declarações de autenticação e permissões (roles). Este token é assinado usando uma chave secreta ou um par de chaves pública/privada, garantindo sua integridade e autenticidade.

A estrutura de um token *JWT* é composta por três partes, conforme mostrado na Figura 13:

1. **Header**: Especifica o tipo de token (*JWT*) e o algoritmo de assinatura (por exemplo SHA256).
2. **Payload**: Contém as declarações ou informações sobre o usuário (por exemplo, ID do usuário, papéis (roles), tempo de expiração).
3. **Signature**: Garante que o token não foi adulterado, gerado ao codificar o header

Figura 13 – Estrutura do token *JWT*



Fonte: Ibrahim (2024).

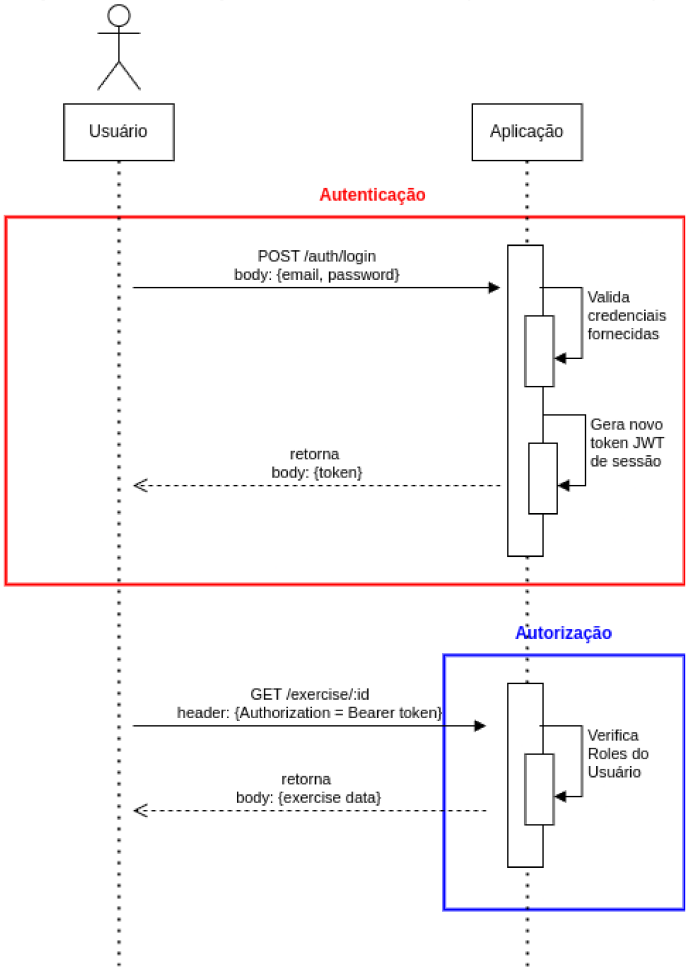
e o payload e assiná-los usando uma chave secreta.

3.6.2 Implementação

A seguir, está a explicação passo a passo do fluxo de autenticação e autorização implementado, conforme ilustrado na Figura 14:

1. Usuário autentica com email e senha
   - O usuário envia uma requisição POST para o *endpoint* /auth/login com seu email e senha.
   - A aplicação verifica as credenciais para validar a identidade do usuário.
   - Se as credenciais forem válidas, o servidor gera um token *JWT*. Este token inclui as informações e papeis (roles) do usuário, codificadas no payload, e é assinado com a chave secreta do servidor.
2. O servidor retorna um token de sessão *JWT*
   - Após a autenticação bem-sucedida, o servidor retorna o *JWT* no corpo da resposta ou no cabeçalho.
   - O cliente (por exemplo, um navegador ou aplicativo móvel) armazena o *JWT*, geralmente no *local storage* ou em um cookie seguro.
3. O usuário solicita um *endpoint* autenticado
   - O cliente envia uma requisição para um *endpoint* autenticado, como GET /exercise/:id, usado para recuperar dados específicos de um exercício.

Figura 14 – Diagrama de Autenticação e Autorização



Fonte: elaborada pelo autor.

- O token *JWT* é incluído como um *Bearer Token* no cabeçalho *'Authorization'* da requisição.

4. Aplicação verifica o token *JWT* e os papeis (roles) do usuário
   - Ao receber a requisição, o servidor extrai o token *JWT* do cabeçalho *'Authorization'*.
   - O servidor verifica a assinatura do token para garantir que ele não foi adulterado e verifica sua validade (por exemplo, se já expirou).
   - O payload do token contém os papeis (roles) e permissões do usuário. O servidor inspeciona esses papeis (roles) para determinar se o usuário tem o acesso necessário ao recurso solicitado.
5. Os dados do exercício são retornados
   - Se o token for válido e o usuário tiver as permissões necessárias, o servidor recupera os dados do exercício solicitado e os retorna ao cliente na resposta.
   - Se o token for inválido ou o usuário não tiver as permissões necessárias, o

servidor retorna uma resposta HTTP apropriada.

Essa abordagem baseada em *JWT* é eficiente porque o servidor não precisa manter o estado da sessão. As credenciais e permissões do usuário estão encapsuladas no token, que é passado com cada requisição, garantindo um processo de autenticação e autorização sem estado e escalável para a *API*.

4 RESULTADOS

Esta seção apresenta os resultados do desenvolvimento de uma *API REST* utilizando os princípios da arquitetura limpa e desenvolvimento orientado a testes *TDD*. Inclui os resultados de cobertura de testes e demonstra um aplicativo *frontend* de *POC* desenvolvido para testar a integração da *API*.

4.1 Desenvolvimento da API REST Usando Arquitetura Limpa e *TDD*

A *API REST* foi desenvolvida seguindo uma arquitetura limpa, enfatizando a separação de responsabilidades e independência de camadas para melhorar a manutenibilidade e escalabilidade da aplicação.

Ao adotar o *TDD*, cada unidade da *API* foi testada antes do desenvolvimento efetivo do código. Essa abordagem garantiu que o código atendesse aos requisitos especificados e facilitou a detecção precoce de defeitos. Testes unitários foram escritos para controladores, casos de uso e repositórios, garantindo cobertura abrangente das funcionalidades da *API*.

4.1.1 Resultados da Cobertura de Testes

Para avaliar a confiabilidade e robustez da aplicação desenvolvida, foram realizados testes extensivos através da suíte de testes que inclui testes unitários, testes de integração e testes de ponta a ponta. A alta cobertura de testes indica que a maior parte do código é exercitada durante os testes, reduzindo a probabilidade de falhas não detectadas e melhorando a qualidade do código. A Figura 15 ilustra o relatório de cobertura de código gerado pela ferramenta de testes.

Figura 15 – Relatório de Cobertura de Código

Element	Class, %	Method, %	Line, %	Branch, %
all	86% (97/112)	84% (263/313)	84% (702/832)	72% (138/190)
application	100% (4/4)	80% (16/20)	78% (47/60)	100% (2/2)
configuration	40% (4/10)	42% (8/19)	42% (8/19)	30% (8/26)
domain	89% (35/39)	84% (66/78)	86% (187/216)	81% (90/110)
infra	92% (53/57)	88% (168/189)	85% (426/499)	73% (38/52)
Application	100% (1/1)	100% (5/5)	100% (34/34)	100% (0/0)

Fonte: elaborada pelo autor.

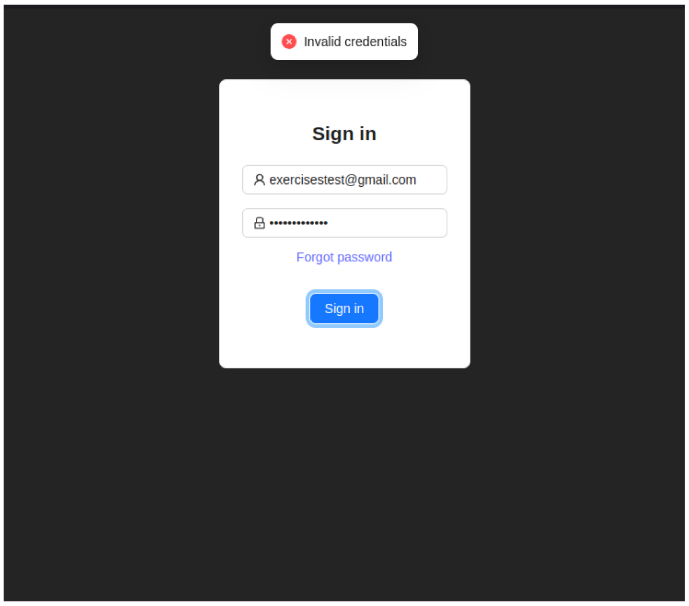
4.2 Aplicativo *Frontend* de *POC* para Integração com a *API*

Um aplicativo *frontend* de *POC* foi desenvolvido para validar a integração com a *API* e demonstrar algumas de suas funcionalidades. O *frontend* foi construído usando o *framework* moderno de JavaScript, React. Devido ao escopo limitado da *POC*, a funcionalidade de gestão de listas de exercícios não foi implementada no *frontend*.

4.2.1 Processo de Login

A interface de login permite que os usuários se autentiquem inserindo suas credenciais. Após um login bem-sucedido, os usuários recebem um token para gerenciamento de sessão. Mensagens de erro são exibidas para credenciais inválidas (Figura 16) ou erros do servidor, aprimorando a experiência do usuário.

Figura 16 – Tentativa de Login com Credenciais Inválidas



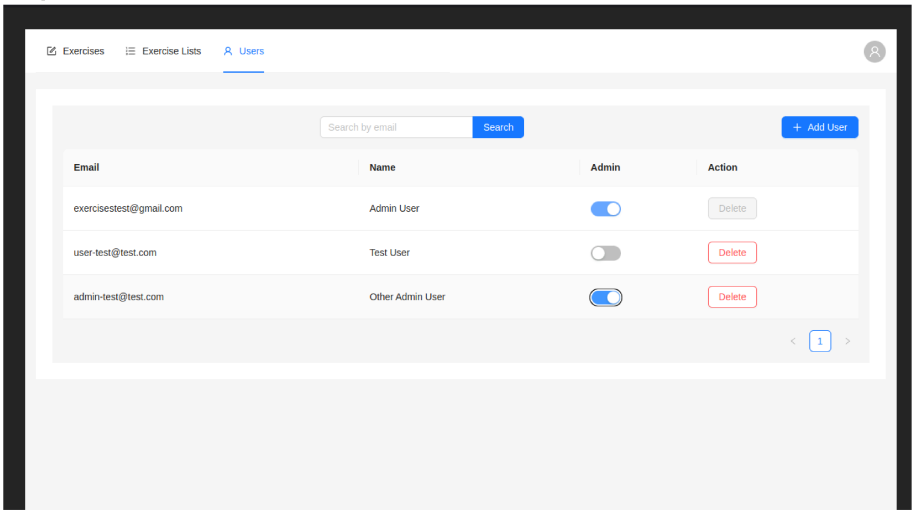
Fonte: elaborada pelo autor.

Na tela de login, os usuários também podem recuperar suas senhas ao clicar no botão *"Esqueci minha senha"*, onde um formulário é exibido para que forneça seu endereço de e-mail. Se o usuário existir, um e-mail contendo um token de sessão é enviado ao usuário, permitindo que ele redefina sua senha com segurança.

4.2.2 Painel de Gestão de Usuários (Visão Administrativa)

Acessível apenas para usuários administradores, o Painel de Gestão de Usuários (Figura 17) oferece funcionalidades para gerenciar contas de usuário:

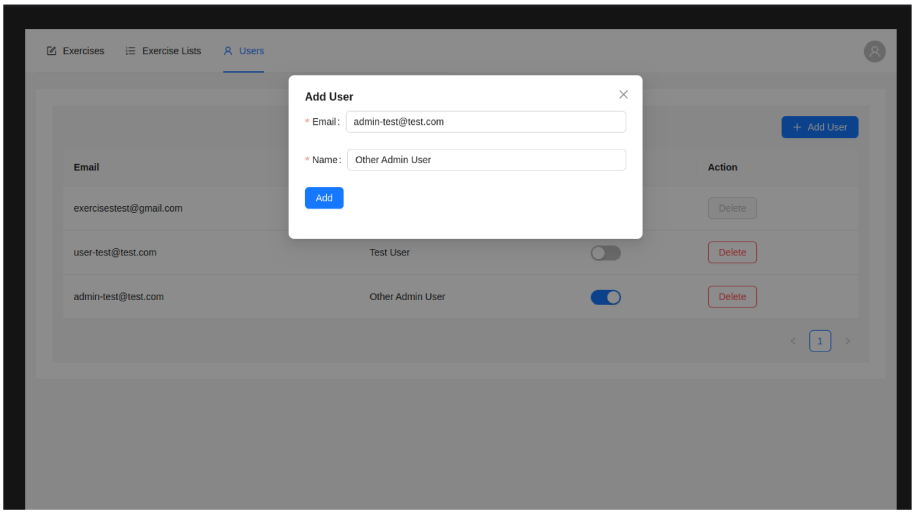
Figura 17 – Painel de Gestão de Usuários



Fonte: elaborada pelo autor.

– Visualizar Usuários: Exibe uma lista de usuários registrados com detalhes como nome de usuário, email e função.
– Criar Usuário: Adiciona novos usuários fornecendo as informações necessárias (Figura 18).

Figura 18 – Modal de Adição de Usuário



Fonte: elaborada pelo autor.

– Atualizar Papel do Usuário: Um botão permite que um usuário seja definido como administrador ou como usuário comum.

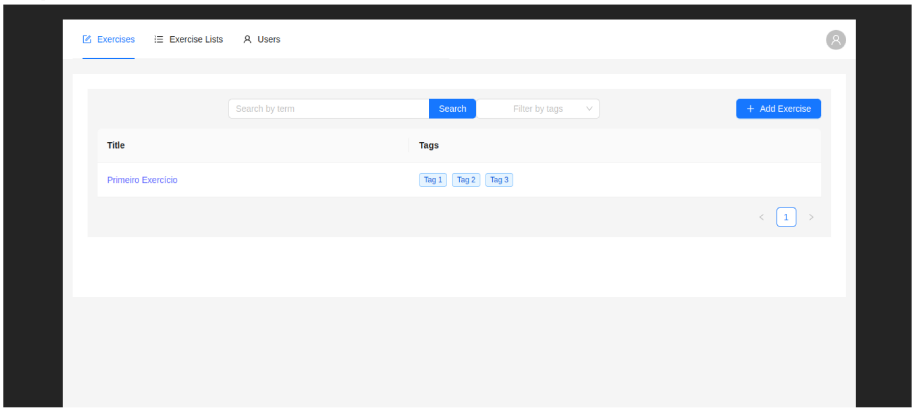
– Excluir Usuário: Remove contas de usuário do sistema.

As ações administrativas são protegidas para evitar acessos não autorizados.

4.2.3 Painel de Gestão de Exercícios

O Painel de Gestão de Exercícios (Figura 19) permite aos usuários gerenciar exercícios com as seguintes funcionalidades:

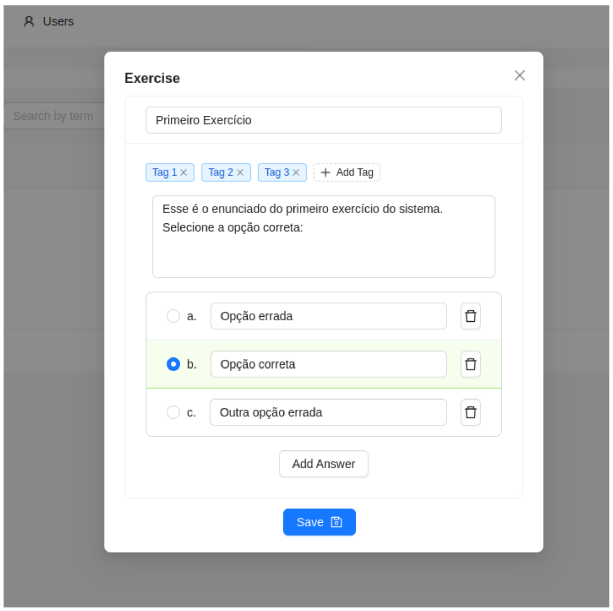
Figura 19 – Painel de Gestão de Exercícios



Fonte: elaborada pelo autor.

– Criar Exercício: Adiciona novos exercícios inserindo detalhes como título, descrição e tags (Figura 20).
– Atualizar Exercício: Edita exercícios existentes para atualizar informações (Figura 20).

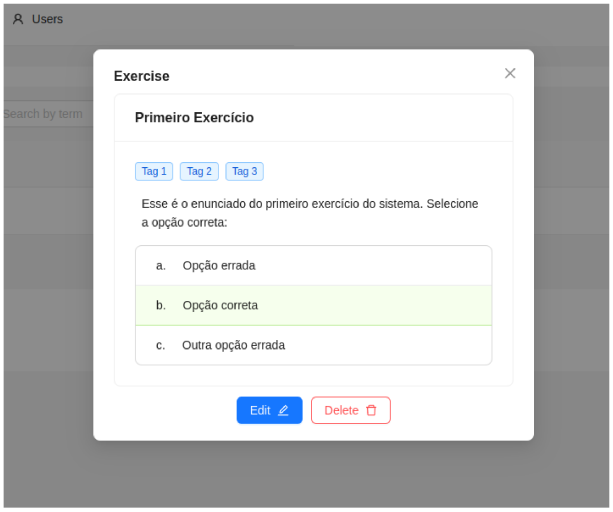
Figura 20 – Modal de Criação/Edição de Exercícios



Fonte: elaborada pelo autor.

– Excluir Exercício: Remove exercícios do sistema através do botão *Delete* (Figura 21).

Figura 21 – Modal de Visualização de Exercícios



Fonte: elaborada pelo autor.

– Buscar Exercícios: Encontra exercícios usando termos de pesquisa e filtra por tags.
– Paginação: Os resultados são paginados para melhorar o desempenho e a experiência do usuário, especialmente com grandes conjuntos de dados.

O desenvolvimento da *API REST* utilizando uma arquitetura limpa simplificada e *TDD* resultou em um sistema robusto e de fácil manutenção. A alta cobertura de testes confirma a confiabilidade do código. O aplicativo *frontend* de *POC* demonstra com sucesso as capacidades da *API* e fornece uma base sólida para futuras melhorias.

5 CONCLUSÕES E TRABALHOS FUTUROS

A aplicação desenvolvida neste trabalho atendeu com sucesso aos objetivos do projeto ao fornecer uma *API REST* para o gerenciamento de questões e listas de exercícios em ambientes educacionais. Ao empregar uma Arquitetura Limpa e o Desenvolvimento Orientado a Testes (*TDD*), entregamos um serviço robusto e de fácil manutenção, que auxilia efetivamente os educadores na organização e indexação de recursos educacionais.

Funcionalidades-chave, como a criação, leitura, atualização e exclusão de perguntas e listas de exercícios, foram implementadas. O uso de tags indexáveis permite uma classificação e recuperação eficiente com base em critérios como disciplina, tópico e nível de dificuldade. As funcionalidades de gerenciamento de usuários, incluindo controle de acesso baseado em funções, garantem acesso seguro e apropriado aos recursos.

5.1 Limitações

1. **Implementação das Interfaces *Frontend* para o Gerenciamento de Listas de Exercícios na *POC*:** O painel de gerenciamento de listas de exercícios não foi implementado no *frontend* para *POC* desenvolvida. Isso impede que os usuários interajam com as listas de exercícios por meio da interface.
2. **Falta de Testes com Usuários Finais:** A aplicação não foi testada por usuários finais, como educadores. Sem o *feedback* direto do público-alvo, questões de usabilidade ou necessidades não atendidas podem existir e não terem sido identificadas.

5.2 Trabalhos Futuros

Para superar essas limitações e aprimorar a aplicação, futuros trabalhos podem se concentrar em:

1. **Concluir a Aplicação *Frontend* da *POC* que Consome a *API*:** O desenvolvimento dos componentes de *frontend* para o gerenciamento de listas de exercícios permitirá que os usuários criem, modifiquem e organizem listas de exercícios diretamente pela interface. Isso proporcionará uma experiência completa e amigável, aproveitando totalmente as capacidades da *API*.
2. **Testes com Usuários Finais:** Realizar testes de usabilidade com educadores utilizando a aplicação *frontend* da *POC* para identificar possíveis problemas de usabilidade ou

requisitos adicionais. Incorporar as opiniões diretas dos usuários finais ajudará a adaptar a aplicação para melhor atender às suas necessidades e melhorar a satisfação geral dos usuários.

REFERÊNCIAS

AUTH0. **Introduction to JSON Web Tokens**. 2023. Disponível em: https://jwt.io/introduction. Acesso em: 10 nov. 2023.

BECK. **Test Driven Development: By Example**. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN 0321146530.

BECK, K. **Extreme Programming Explained: Embrace Change**. USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 0201616416.

BISSI, W.; Serra Seca Neto, A. G.; EMER, M. C. F. P. The effects of test driven development on internal quality, external quality and productivity: A systematic review. **Information and Software Technology**, v. 74, p. 45–54, 2016. ISSN 0950-5849. Disponível em: https://www.sciencedirect.com/science/article/pii/S0950584916300222.

BOUKHARY, S.; COLMENARES, E. A clean approach to flutter development through the flutter clean architecture package. In: **2019 International Conference on Computational Science and Computational Intelligence (CSCI)**. [*S. l.*: *s. n.*], 2019. p. 1115–1120.

COMMUNITY, T. **Testcontainers: Docker-based integration testing in Java**. 2024. Disponível em: https://www.testcontainers.org/. Acesso em: 2 set. 2024.

FIELDING, R. T. **Architectural Styles and the Design of Network-based Software Architectures**. Tese (Doutorado) – University of California, Irvine, 2000.

FOWLER, M.; RICE, D.; FOEMMEL, M.; HIEATT, E.; MEE, R.; STAFFORD, R. **Patterns of Enterprise Application Architecture**. 1st. ed. [*S. l.*]: Addison Wesley, 2002. 560 p. ISBN 0-321-12742-0.

FRAMEWORK, S. **JdbcTemplate Class Documentation**. 2024. Disponível em: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/JdbcTemplate.html. Acesso em: 2 set. 2024.

FREEMAN, S.; PRYCE, N. **Growing Object-Oriented Software, Guided by Tests**. 1st. ed. [*S. l.*]: Addison-Wesley Professional, 2009. ISBN 0321503627.

GREGóRIO, P.; GOYA, D. Hash criptográfico sobre senhas e aleatoriedade do argon2. In: **Anais Estendidos do XIX Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais**. Porto Alegre, RS, Brasil: SBC, 2019. p. 71–80. ISSN 0000-0000. Disponível em: https://sol.sbc.org.br/index.php/sbseg_estendido/article/view/14008.

IBRAHIM, M. **What is a JWT? Understanding JSON Web Tokens**. 2024. Disponível em: https://supertokens.com/blog/what-is-jwt. Acesso em: 2 set. 2024.

JETBRAINS. **Kotlin for Server Side**. 2023. Disponível em: https://kotlinlang.org/docs/server-overview.html. Acesso em: 2 set. 2024.

KENSKI, V. M. **Educação e Tecnologias: O Novo Ritmo da Informação.** 8st. ed. [*S. l.*]: Papirus, 2007. 144 p. ISBN 978-8530808280.

LIQUIBASE. **Introduction to Liquibase**. 2024. Disponível em: https://docs.liquibase.com/concepts/introduction-to-liquibase.html. Acesso em: 2 set. 2024.

MARTIN, R. C. **The Clean Code Blog - The Clean Architecture**. 2012. Disponível em: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. Acesso em: 10 nov. 2023.

MARTIN, R. C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. [*S. l.*]: Pearson, 2017. v. 2017. ISBN 0134494164.

MEHTA, J. **Why Password Salt and Hash Make for Better Security?** 2024. Disponível em: https://cheapsslweb.com/blog/why-password-salt-and-hash-make-for-better-security/. Acesso em: 2 set. 2024.

PLöSCH R., G. H. H. A. e. a. The emisq method and its tool support - expert-based evaluation of internal software quality. **Innovations Systems Software Eng**, v. 4, p. 3 – 15, 2008.

RAI, A. **Stateless web Architecture using REST API and JWT-cookie**. 2021. Disponível em: https://mraniketr.medium.com/stateless-web-architecture-using-rest-api-and-jwt-cookie-b1d6f26c0c65. Acesso em: 10 nov. 2023.

ROBILLARD, M. P.; BODDEN, E.; KAWRYKOW, D.; MEZINI, M.; RATCHFORD, T. Automated api property inference techniques. **IEEE Transactions on Software Engineering**, v. 39, n. 5, p. 613–637, 2013.

APÊNDICE A – REPOSITÓRIO DO PROJETO

O repositório disponibilizado através da URL https://github.com/jairoaraujo10/exercises contém todo o código desenvolvido neste trabalho. A estrutura do repositório está organizada da seguinte forma:

- **/backend/**: Código da *API REST* desenvolvido em Kotlin, com dependências gerenciadas via Maven, e seu Dockerfile.
- **/frontend/**: Código da aplicação *Frontend* de *POC* desenvolvido em React com TypeScript, utilizando o Vite como ferramenta de build, com arquivo Nginx de configuração de roteamento e seu Dockerfile.
- **docker-compose.yml**: Arquivo de orquestração de containers Docker, facilitando a execução simultânea dos serviços *backend* e *frontend*.
- **README.md**: Documentação de desenvolvimento que inclui uma visão geral do projeto instruções detalhadas para configuração, execução e desenvolvimento da aplicação.

APÊNDICE B – CHANGESET YAML DO LIQUIBASE

Este apêndice contém o código fonte YAML de *changeset* utilizado para o controle de versão do esquema do banco de dados com Liquibase. O arquivo define a criação de tabelas, colunas e restrições necessárias para a aplicação, assegurando a integridade e a consistência do banco de dados em diferentes ambientes de desenvolvimento, teste e produção.

Código-fonte 1 – Changeset YAML

```yaml
databaseChangeLog:
  - changeSet:
      id: 1
      author: jairo
      changes:
        - createTable:
            tableName: users
            columns:
              - column:
                  name: id
                  type: BIGINT AUTO_INCREMENT
                  constraints:
                    primaryKey: true
                    nullable: false
              - column:
                  name: name
                  type: VARCHAR(64)
                  constraints:
                    nullable: false
              - column:
                  name: email
                  type: VARCHAR(320)
                  constraints:
                    unique: true
                    nullable: false
```

```
26              - column:
27                  name: password
28                  type: VARCHAR(128)
29                  constraints:
30                    nullable: false
31              - column:
32                  name: roles
33                  type: VARCHAR(516)
34                  constraints:
35                    nullable: false
36
37  - changeSet:
38      id: 2
39      author: jairo
40      changes:
41        - createTable:
42            tableName: exercises
43            columns:
44              - column:
45                  name: id
46                  type: BIGINT AUTO_INCREMENT
47                  constraints:
48                    primaryKey: true
49                    nullable: false
50              - column:
51                  name: title
52                  type: VARCHAR(255)
53                  constraints:
54                    nullable: false
55              - column:
56                  name: author_id
57                  type: BIGINT
```

```
58                  constraints:
59                    nullable: false
60              - column:
61                  name: creation_timestamp
62                  type: BIGINT
63                  constraints:
64                    nullable: false
65              - column:
66                  name: content
67                  type: JSON
68                  constraints:
69                    nullable: false
70        - addForeignKeyConstraint:
71            baseTableName: exercises
72            baseColumnNames: author_id
73            constraintName: fk_exercises_users
74            referencedTableName: users
75            referencedColumnNames: id
76
77  - changeSet:
78      id: 3
79      author: jairo
80      changes:
81        - createTable:
82            tableName: exercise_tags
83            columns:
84              - column:
85                  name: exercise_id
86                  type: BIGINT
87                  constraints:
88                    nullable: false
89              - column:
```

```
90                  name: tag_value
91                  type: VARCHAR(255)
92                  constraints:
93                    nullable: false
94        - addPrimaryKey:
95            tableName: exercise_tags
96            columnNames: exercise_id, tag_value
97            constraintName: pk_exercise_tags
98        - addForeignKeyConstraint:
99            baseTableName: exercise_tags
100           baseColumnNames: exercise_id
101           constraintName: fk_exercise_tags_exercises
102           referencedTableName: exercises
103           referencedColumnNames: id
104
105   - changeSet:
106       id: 4
107       author: jairo
108       changes:
109         - createTable:
110             tableName: exercise_lists
111             columns:
112               - column:
113                   name: id
114                   type: BIGINT AUTO_INCREMENT
115                   constraints:
116                     primaryKey: true
117                     nullable: false
118               - column:
119                   name: title
120                   type: VARCHAR(255)
121                   constraints:
```

```
122                     nullable: false
123               - column:
124                   name: author_id
125                   type: BIGINT
126                   constraints:
127                     nullable: false
128               - column:
129                   name: creation_timestamp
130                   type: BIGINT
131                   constraints:
132                     nullable: false
133         - addForeignKeyConstraint:
134             baseTableName: exercise_lists
135             baseColumnNames: author_id
136             constraintName: fk_exercise_lists_users
137             referencedTableName: users
138             referencedColumnNames: id
139
140   - changeSet:
141       id: 5
142       author: jairo
143       changes:
144         - createTable:
145             tableName: exercise_list_exercises
146             columns:
147               - column:
148                   name: exercise_list_id
149                   type: BIGINT
150                   constraints:
151                     nullable: false
152               - column:
153                   name: exercise_id
```

```
154                   type: BIGINT
155                   constraints:
156                     nullable: false
157         - addPrimaryKey:
158             tableName: exercise_list_exercises
159             columnNames: exercise_list_id, exercise_id
160             constraintName: pk_exercise_list_exercises
161         - addForeignKeyConstraint:
162             baseTableName: exercise_list_exercises
163             baseColumnNames: exercise_list_id
164             constraintName:
                  fk_exercise_list_exercises_lists
165             referencedTableName: exercise_lists
166             referencedColumnNames: id
167         - addForeignKeyConstraint:
168             baseTableName: exercise_list_exercises
169             baseColumnNames: exercise_id
170             constraintName:
                  fk_exercise_list_exercises_exercises
171             referencedTableName: exercises
172             referencedColumnNames: id
173
174   - changeSet:
175       id: 6
176       author: jairo
177       changes:
178         - createTable:
179             tableName: exercise_list_tags
180             columns:
181               - column:
182                   name: exercise_list_id
183                   type: BIGINT
```

```
                constraints:
                  nullable: false
            - column:
                name: tag_value
                type: VARCHAR(255)
                constraints:
                  nullable: false
        - addPrimaryKey:
            tableName: exercise_list_tags
            columnNames: exercise_list_id, tag_value
            constraintName: pk_exercise_list_tags
        - addForeignKeyConstraint:
            baseTableName: exercise_list_tags
            baseColumnNames: exercise_list_id
            constraintName: fk_exercise_list_tags_lists
            referencedTableName: exercise_lists
            referencedColumnNames: id
```

APÊNDICE C – DOCUMENTAÇÃO DA *API*

Este apêndice contém a documentação completa da *API* em formato *Portable Document Format* (PDF) desenvolvida para a aplicação, incluindo *endpoints*, parâmetros, exemplos de requisições e respostas, entre outras informações técnicas. A documentação serve como referência para desenvolvedores que queiram integrar ou interagir com a *API* e está disponível no repositório do projeto (Apêndice A).

Documentação das APIs

Este documento fornece informações detalhadas sobre os endpoints da API de Autenticação, Gerenciamento de Usuários, Gerenciamento de Exercícios e Gerenciamento de **Listas de Exercícios**, incluindo suas funcionalidades, estruturas de requisição e resposta, e possíveis códigos de status.

Índice

Visão Geral da API de Autenticação

A API de Autenticação lida com os processos de autenticação de usuários, incluindo login, solicitação de redefinição de senha e redefinição de senha. Ela garante acesso seguro a recursos protegidos emitindo e validando tokens.

Visão Geral da API de Gerenciamento de Usuários

A API de Gerenciamento de Usuários permite gerenciar dados de usuários dentro do sistema. Fornece funcionalidades para recuperar informações de usuários, criar novos usuários, excluir usuários existentes e buscar usuários com base em critérios específicos.

Visão Geral da API de Gerenciamento de Exercícios

A API de Gerenciamento de Exercícios facilita a criação, recuperação, atualização, exclusão e busca de exercícios no sistema. Permite que usuários autorizados gerenciem o conteúdo de exercícios de maneira eficiente, garantindo que os exercícios estejam organizados, acessíveis e atualizados.

Visão Geral da API de Gerenciamento de Listas de Exercícios

A API de Gerenciamento de **Listas de Exercícios** permite a criação, recuperação, atualização, exclusão e busca de listas de exercícios, bem como a adição e remoção de exercícios dessas listas. Facilita a organização e o gerenciamento de conjuntos de exercícios para estudos ou avaliações.

Endpoints

Endpoints de Autenticação

1. Login de Usuário

Endpoint: `/auth/login`
Método: `POST`
Descrição: Autentica um usuário utilizando seu email e senha. Após a autenticação bem-sucedida, retorna um token para acesso autorizado.

Requisição:

- **Headers:**
    - `Content-Type: application/json`

- **Body:**

```
{
  "email": "string",
  "password": "string"
}
```

Respostas:

- **200 OK:**
  - **Body:**

```
{
  "token": "string"
}
```

- **400 Requisição Inválida:**
  - **Body:**

```
{
  "message": "string"
}
```

- **401 Não Autorizado:**
  - **Body:**

```
{
  "message": "Credenciais inválidas"
}
```

2. Solicitar Redefinição de Senha

Endpoint: `/auth/reset-password/request`
Método: `POST`
Descrição: Inicia o processo de redefinição de senha enviando um link de redefinição para o email fornecido.

Requisição:

- **Headers:**
  - `Content-Type: application/json`
- **Body:**

```
{
  "email": "string"
}
```

Respostas:

- **204 Sem Conteúdo**
- **400 Requisição Inválida:**
  - **Body:**

```
{
  "message": "Email ausente"
}
```

3. Redefinir Senha

Endpoint: `/auth/reset-password`
Método: `POST`
Descrição: Redefine a senha do usuário utilizando um token válido.

Requisição:

- **Headers:**
  - `Content-Type: application/json`
  - `Authorization: Bearer <token>`
- **Body:**

```
{
  "password": "string"
}
```

Respostas:

- **200 OK:**
  - **Body:**

```
{
  "message": "Senha redefinida com sucesso"
}
```

- **400 Requisição Inválida:**
- **401 Não Autorizado**
- **404 Não Encontrado**

Endpoints de Gerenciamento de Usuários

4. Obter Usuário por ID

Endpoint: `/user/{id}`
Método: `GET`
Descrição: Recupera um usuário pelo seu ID.

Requisição:

- **Headers:**
  - `Authorization: Bearer <token>`

Respostas:

- **200 OK:** Retorna os detalhes do usuário.
- **401 Não Autorizado**
- **404 Não Encontrado**

5. Criar Usuário

Endpoint: `/user`
Método: `POST`
Descrição: Cria um novo usuário.

Requisição:

- **Headers:**
  - `Content-Type: application/json`
  - `Authorization: Bearer <token>`
- **Body:**

```
{
  "name": "string",
  "email": "string"
}
```

Respostas:

- **201 Criado**
- **400 Requisição Inválida**
- **401 Não Autorizado**

6. Excluir Usuário

Endpoint: `/user/{id}`
Método: `DELETE`
Descrição: Exclui um usuário pelo ID.

Requisição:

- **Headers:**
  - `Authorization: Bearer <token>`

Respostas:

- **204 Sem Conteúdo**
- **401 Não Autorizado**
- **404 Não Encontrado**

7. Buscar Usuários

Endpoint: `/user/search`
Método: `POST`
Descrição: Busca usuários com base em critérios fornecidos. Suporta paginação via parâmetros de query.

Requisição:

- **Headers:**
  - `Content-Type: application/json`
  - `Authorization: Bearer <token>`
- **Query Parameters:**
  - `limit` (inteiro, opcional): O número máximo de usuários a serem retornados.
  - `offset` (inteiro, opcional): O número de usuários a serem ignorados antes de coletar o conjunto de resultados.
- **Body:**

```
{
  "searchTerm": "string"
}
```

Respostas:

- **200 OK:**
  - **Body:**

```
{
  "users": [
    {
      "id": "integer",
      "name": "string",
      "email": "string",
      "roles": ["string"]
    }
    // ... mais usuários
  ],
  "total": "integer"
}
```

- **400 Requisição Inválida**
- **401 Não Autorizado**

Endpoints de Gerenciamento de Exercícios

8. Criar Exercício

Endpoint: `/exercise`
Método: `POST`
Descrição: Cria um novo exercício.

Requisição:

- **Headers:**
  - `Content-Type: application/json`
  - `Authorization: Bearer <token>`
- **Body:**

```
{
  "title": "string",
  "description": "string",
  "tags": [{ "value": "string" }],
  "possibleAnswers": ["string"],
  "correctAnswerIndex": "integer"
}
```

Respostas:

- **201 Criado**
- **400 Requisição Inválida**
- **401 Não Autorizado**

9. Obter Exercício por ID

Endpoint: `/exercise/{id}`
Método: `GET`
Descrição: Recupera um exercício pelo ID.

Requisição:

- **Headers:**
  - `Authorization: Bearer <token>`

Respostas:

- **200 OK:** Retorna os detalhes do exercício.
  - **Body:**

```
{
  "id": "string",
  "title": "string",
  "description": "string",
  "tags": [
    {
      "value": "string"
    }
  ],
  "possibleAnswers": ["string"],
  "correctAnswerIndex": "integer"
}
```

- **401 Não Autorizado**
- **404 Não Encontrado**

10. Atualizar Exercício

Endpoint: `/exercise/{id}`
Método: `PUT`
Descrição: Atualiza um exercício pelo ID.

Requisição:

- **Headers:**
  - `Content-Type: application/json`
  - `Authorization: Bearer <token>`
- **Body:**

```
{
  "title": "string",
  "description": "string",
  "tags": [{ "value": "string" }],
  "possibleAnswers": ["string"],
  "correctAnswerIndex": "integer"
}
```

Respostas:

- **204 Sem Conteúdo**
- **400 Requisição Inválida**
- **401 Não Autorizado**
- **404 Não Encontrado**

11. Excluir Exercício

Endpoint: `/exercise/{id}`
Método: `DELETE`
Descrição: Exclui um exercício pelo ID.

Requisição:

- **Headers:**
  - `Authorization: Bearer <token>`

Respostas:

- **204 Sem Conteúdo**
- **401 Não Autorizado**
- **404 Não Encontrado**

12. Buscar Exercícios

Endpoint: `/exercise/search`
Método: `POST`
Descrição: Busca exercícios com base em critérios fornecidos. Suporta paginação via parâmetros de query.

Requisição:

- **Headers:**
  - `Content-Type: application/json`
  - `Authorization: Bearer <token>`
- **Query Parameters:**
  - `limit` (inteiro, opcional): O número máximo de exercícios a serem retornados.
  - `offset` (inteiro, opcional): O número de exercícios a serem ignorados antes de coletar o conjunto de resultados.
- **Body:**

```
{
  "searchTerm": "string",
  "tags": [
    {
      "value": "string"
    }
  ]
}
```

Respostas:

- **200 OK:**
  - **Body:**

```
{
  "exercises": [
    {
      "id": "string",
      "title": "string",
      "tags": [
        {
          "value": "string"
        }
      ]
    }
    // ... mais metadados de exercícios
  ],
  "total": "integer"
}
```

- **400 Requisição Inválida**
- **401 Não Autorizado**

Endpoints de Gerenciamento de Listas de Exercícios

13. Criar Lista de Exercícios

Endpoint: `/exercises-list`
Método: `POST`
Descrição: Cria uma nova lista de exercícios.

Requisição:

- **Headers:**
  - `Content-Type: application/json`
  - `Authorization: Bearer <token>`
- **Body:**

```
{
  "title": "string",
  "tags": [{ "value": "string" }]
}
```

Respostas:

- **201 Criado**
- **400 Requisição Inválida**
- **401 Não Autorizado**

14. Obter Lista de Exercícios por ID

Endpoint: `/exercises-list/{id}`
Método: `GET`
Descrição: Recupera uma lista de exercícios pelo ID.

Requisição:

- **Headers:**
  - `Authorization: Bearer <token>`

Respostas:

- **200 OK:** Retorna os detalhes da lista de exercícios.
  - **Body:**

```
{
  "id": "string",
  "title": "string",
  "tags": [{ "value": "string" }],
  "exerciseIds": ["integer"]
}
```

- **401 Não Autorizado**
- **404 Não Encontrado**

15. Atualizar Lista de Exercícios

Endpoint: `/exercises-list/{id}`
Método: `PUT`
Descrição: Atualiza uma lista de exercícios pelo ID.

Requisição:

- **Headers:**
  - `Content-Type: application/json`
  - `Authorization: Bearer <token>`
- **Body:**

```
{
  "title": "string",
  "tags": [{ "value": "string" }]
}
```

Respostas:

- **204 Sem Conteúdo**
- **400 Requisição Inválida**
- **401 Não Autorizado**
- **404 Não Encontrado**

16. Excluir Lista de Exercícios

Endpoint: `/exercises-list/{id}`
Método: `DELETE`
Descrição: Exclui uma lista de exercícios pelo ID.

Requisição:

- **Headers:**
  - `Authorization: Bearer <token>`

Respostas:

- **204 Sem Conteúdo**
- **401 Não Autorizado**
- **404 Não Encontrado**

17. Buscar Listas de Exercícios

Endpoint: `/exercises-list/search`
Método: `POST`
Descrição: Busca listas de exercícios com base em critérios fornecidos. Suporta paginação via parâmetros de query.

Requisição:

- **Headers:**
  - `Content-Type: application/json`

- `Authorization: Bearer <token>`

- **Query Parameters:**
  - `limit` (inteiro, opcional): O número máximo de listas a serem retornadas.
  - `offset` (inteiro, opcional): O número de listas a serem ignoradas antes de coletar o conjunto de resultados.

- **Body:**

```
{
  "searchTerm": "string",
  "tags": [{ "value": "string" }]
}
```

Respostas:

- **200 OK:**
  - **Body:**

```
{
  "exercisesLists": [
    {
      "id": "string",
      "title": "string",
      "tags": [{ "value": "string" }]
    }
    // ... mais metadados de listas
  ],
  "total": "integer"
}
```

- **400 Requisição Inválida**
- **401 Não Autorizado**

18. Adicionar Exercício à Lista

Endpoint: `/exercises-list/{id}/exercises`
Método: `POST`
Descrição: Adiciona um exercício à lista de exercícios especificada.

Requisição:

- **Headers:**
  - `Content-Type: application/json`
  - `Authorization: Bearer <token>`

- **Body:**

```
{
  "exerciseId": "integer"
}
```

Respostas:

- **204 Sem Conteúdo**
- **400 Requisição Inválida**
- **401 Não Autorizado**
- **404 Não Encontrado**

19. Remover Exercício da Lista

Endpoint: `/exercises-list/{id}/exercises/{exerciseId}`
Método: `DELETE`
Descrição: Remove um exercício da lista de exercícios especificada.

Requisição:

- **Headers:**
  - `Authorization: Bearer <token>`

Respostas:

- **204 Sem Conteúdo**
- **401 Não Autorizado**
- **404 Não Encontrado**

Respostas de Erro

A API utiliza códigos de status HTTP padrão:

- **400 Requisição Inválida:** Parâmetros ausentes ou inválidos.
- **401 Não Autorizado:** Token inválido ou ausente.
- **404 Não Encontrado:** Recurso não encontrado.
- **204 Sem Conteúdo:** Requisição bem-sucedida, mas sem conteúdo para enviar na resposta.

Autenticação

Para acessar endpoints protegidos, inclua o seguinte header:

```
Authorization: Bearer <token>
```

Certifique-se de substituir `<token>` pelo token de acesso válido fornecido após a autenticação.

Notas

- Todos os endpoints que modificam dados (como `POST`, `PUT`, `DELETE`) requerem autenticação.
- Use os parâmetros de query `limit` e `offset` para controlar a paginação nos endpoints de busca.
- As respostas de erro fornecem mensagens detalhadas para facilitar o diagnóstico de problemas.