UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Proposta de arquitetura e desenvolvimento de API do sistema Bot Educacional utilizando os princípios de Domain-Driven Design e Arquitetura Limpa

Pedro Henrique Wilfride de Lima Boussiengui

Orientador: Prof. Dr. Orivaldo Vieira de Santana Júnior

Natal, RN, 13 de dezembro de 2023

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE TECNOLOGIA
GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

Proposta de arquitetura e desenvolvimento de API do sistema Bot Educacional utilizando os princípios de Domain-Driven Design e Arquitetura Limpa

Pedro Henrique Wilfride de Lima Boussiengui

Orientador: Prof. Dr. Orivaldo Vieira de Santana Júnior

Trabalho de Conclusão de Curso de Graduação na modalidade Monografia, submetido como parte dos requisitos necessários para conclusão do curso de Engenharia de Computação pela Universidade Federal do Rio Grande do Norte (UFRN/CT).

Natal, RN, 13 de dezembro de 2023

Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI
Catalogação de Publicação na Fonte. UFRN - Biblioteca Central Zila Mamede

Boussiengui, Pedro Henrique Wilfride de Lima.
Proposta de arquitetura e desenvolvimento de API do sistema Bot Educacional utilizando os princípios de Domain-Driven Design e Arquitetura Limpa / Pedro Henrique Wilfride de Lima Boussiengui. - 2023.
41 f.: il.

Monografia (graduação) - Universidade Federal do Rio Grande do Norte, Curso de Engenharia da COmputação, Natal, RN, 2023.
Orientação: Prof. Dr. Orivaldo Vieira de Santana Júnior.

1. Bot Educacional - Monografia. 2. Domain-Driven Design - Monografia. 3. Arquitetura limpa - Monografia. 4. TDD - Monografia. I. Santana Júnior, Orivaldo Vieira de. II. Título.

RN/UF/BCZM CDU 004.41

Elaborado por Ana Cristina Cavalcanti Tinoco - CRB-15/262

Proposta de arquitetura e desenvolvimento de API do sistema Bot Educacional utilizando os princípios de Domain-Driven Design e Arquitetura Limpa

Pedro Henrique Wilfride de Lima Boussiengui

Monografia aprovada em 13 de dezembro de 2023, pela banca examinadora composta pelos seguintes membros:

Prof. Dr. Orivaldo Vieira de Santana Júnior (Orientador) . . . . . ECT/UFRN

Prof. Dr. Eduardo De Lucena Falcão (Examinador) . . . . . DCA/UFFN

Prof. Dr. Marconi Câmara Rodrigues (Examinador) . . . . . ECT/UFRN

Ao meu orientador e todos os desenvolvedores que contribuíram e aos que ainda vão contribuir com o desenvolvimento desse projeto.

Agradecimentos

A Deus em primeiro lugar, por me permitir chegar até onde cheguei.

Ao meu orientador, professor Orivaldo, sou grato pela orientação e pela confiança de desenvolver esse trabalho.

Aos demais colegas de graduação, pelas críticas e sugestões.

À minha família pelo apoio durante esta jornada.

Resumo

Recentemente, tornou-se frequente a utilização de robôs de conversação, conhecidos como *bots*, para interações baseadas em texto, executando comandos específicos. Nesse contexto, surge o Bot Educacional, um sistema inovador projetado para atuar como uma plataforma de *bots*, proporcionando suporte tanto para alunos quanto para professores nas disciplinas iniciais de Lógica de Programação. A arquitetura proposta visa modularizar os componentes do sistema, buscando um desacoplamento eficiente que favoreça a flexibilidade e escalabilidade. Durante o desenvolvimento do Bot Educacional, foram adotadas práticas sólidas de engenharia de software, incorporando os princípios do Domain-Driven Design e da Arquitetura Limpa. Esse enfoque resultou em um domínio expressivo e bem definido, refletindo claramente as regras de negócio e os casos de uso do sistema. A introdução de Design Patterns, por meio de padrões como *Repository*, contribuiu significativamente para uma estrutura de código mais adaptável e extensível, estimulando a reutilização de componentes e simplificando a manutenção contínua do sistema. Adicionalmente, a aplicação da abordagem TDD garantiu a robustez e confiabilidade do código, estabelecendo uma base sólida para a evolução futura do Bot Educacional. Este projeto serve como um exemplo notável da aplicação bem-sucedida de práticas de engenharia modernas, resultando em uma API coesa e adaptável. A visão de futuro do Bot Educacional é tornar-se uma ferramenta amplamente utilizada por diversos professores e turmas, solidificando seu papel como uma solução eficaz principalmente nos cursos iniciais da graduação.

Palavras-chave: Bot Educacional, Domain-Driven Design, Arquitetura Limpa, TDD.

Abstract

Recently, the use of chatbots, commonly known as bots, for text-based interactions and specific command executions has become increasingly prevalent. In this context, the Educational Bot's emerges as an innovative system designed to function as a bot platform, providing support for both students and teachers in introductory Programming Logic courses. The proposed architecture aims to modularize system components, striving for efficient decoupling that enhances flexibility and scalability. Throughout the development of the Educational Bot's, robust software engineering practices were employed, incorporating principles from Domain-Driven Design and Clean Architecture. This approach resulted in an expressive and well-defined domain, transparently reflecting the system's business rules and use cases. The introduction of Design Patterns, such as the Repository pattern, significantly contributed to a more adaptable and extensible code structure, encouraging component reuse and simplifying continuous system maintenance. Furthermore, the ap- plication of Test-Driven Development ensured the robustness and reliability of the code, establishing a solid foundation for the Educational Bot's future evolution. This project stands as a notable example of the successful application of modern engineering practices, yielding a cohesive and adaptable API. The Educational Bot's future vision is to become a widely utilized tool across various classrooms and by numerous educators, solidifying its role as an effective solution, especially in the early undergraduate courses.

Keywords: Educational Bot's, Domain-Driven Design, Clean Architecture, TDD.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Os *bots*, uma forma curta para Software Robots, são programas de computador projetados para interagir e conduzir conversas com usuários de forma automatizada. Elas utilizam, em sua maioria, técnicas de inteligência artificial e processamento de linguagem natural. Os bots são capazes de compreender e responder a perguntas, fornecer informações, executar tarefas específicas e até simular diálogos humanos. Segundo (Ferrara et al. 2016), os bots existem desde os primórdios da computação e sempre houve esforços no desenvolvimento de algoritmos que conseguissem manter uma conversa com um humano e que passassem no teste de Turing.

Nos dias atuais, testemunhamos um notável crescimento e difusão dos *bots*, impulsionados pela crescente demanda por interações instantâneas e eficientes em diversos setores. Desde o atendimento ao cliente até assistentes pessoais virtuais, os *bots* têm se tornado uma presença onipresente, oferecendo respostas rápidas, soluções automatizadas e uma experiência mais fluida para os usuários. Esse fenômeno reflete a busca contínua por tecnologias que otimizem a comunicação digital, proporcionando não apenas eficácia operacional, mas também uma resposta ágil às expectativas modernas de conveniência e acessibilidade.

Tendo isso em vista, o sistema Bot Educacional surgiu exatamente como uma solução de software que consiste de um robô interativo por texto que auxilia alunos e professores nas disciplinas iniciais do curso de BCT (Bacharelado em Ciências e Tecnologia), sendo, a princípio, direcionado para os alunos da disciplina de LOP (Lógica de Programação). A interface gráfica de comunicação escolhida em um primeiro momento foi o *Discord*, que é uma plataforma para comunicação individual ou em grupos que permite enviar mensagens de texto e criar chamadas de áudio ou vídeo.

O código do sistema Bot Educacional, que é objeto de pesquisa desse trabalho, em um primeiro momento apresentava em sua primeira implementação um forte acoplamento, sendo composto por um único monólito que integrava tanto as funcionalidade de robô dos Discord, como as regras de negócio da aplicação, a API (*Application Programming Interface*), e a parte referente ao motor NLP (Neural Language Processing), que do ponto de vista da Engenharia de Software vai em oposição as boas de projeto e desenvolvimento.

Como mencionado antes, um *bot* é um software, e como software é necessário usar melhores práticas para obter um sistema mais coeso e desacoplado, que seja de fácil manutenção e entendimento. Nesse sentido, o DDD (Domain-Driven Design) entra com o

intuito de desenvolvimento de um domínio muito bem definido e que expresse claramente a intenção da aplicação. A Arquitetura Limpa reutiliza muitos conceitos do próprio DDD, porém ele expressa melhor como deve ser a aplicação do ponto de vista da arquitetura, ou seja, as camadas do sistema, os padrões de design que podem ser utilizados e como os testes são fundamentais para o desenvolvimento seguro do sistema.

Em resumo, vale destacar que o desenvolvimento deste sistema não adota uma implementação "ao pé da letra"das metodologias DDD e Arquitetura Limpa, mas sim uma inspiração consciente desses princípios. Ao adaptar e integrar conceitos dessas abordagens, o propósito é construir um sistema que, embora flexível e eficiente, atenda de forma dinâmica e eficaz às necessidades específicas do contexto em que está inserido.

1.1 Objetivos gerais

O objetivo geral do presente trabalho é projetar e implementar uma arquitetura flexível para o sistema Bot Educacional, que visa auxiliar no aprendizado dos alunos nas disciplinas iniciais do curso de Ciências e Tecnologia. A implementação desse sistema se dará pela refatoração do sistema existente, separando o código em três partes: A primeira é a API, que engloba grande parte das regras de negócio do sistema. O motor NLP, que é inteligencia artificial por trás de responder as perguntas dos alunos e por último o orquestrador de *bots*, que controla as instancias de *bots* do Discord executando como servidor.

Este estudo busca aplicar as boas práticas de desenvolvimento, criando um sistema desacoplado com um domínio rico, para isso será utilizado técnicas de Domain-Driven Design e Arquitetura Limpa. A expectativa é que, ao final do processo, o sistema seja de fácil manutenção e se ajuste facilmente aos novos requisitos que possam surgir.

1.2 Objetivos específicos

- Documentar os requisitos do assistente virtual educacional.
- Projetar um arquitetura adequada para os requisitos do sistema.
- Desenvolver a API principal para o assistente virtual educacional seguindo práticas de Domain-Driven Design e Arquitetura limpa.

No próximo capítulo será explorado o referencial teórico, aprofundando-nos nos conceitos de Domain-Driven Design e Arquitetura Limpa discutidos anteriormente. Além disso, serão analisados outros elementos que desempenham um papel crucial nas boas práticas de desenvolvimento de sistemas. Este mergulho teórico fornecerá uma compreensão mais detalhada das metodologias adotadas, contextualizando-as dentro do panorama mais amplo de práticas recomendadas na engenharia de software.

Capítulo 2

Fundamentação Teórica

Nesse capítulo será feita uma introdução teórica acerca dos tópicos mais relevantes para uma boa compreensão do problema bem como a implementação de sua solução. A princípio será comentado a respeito de arquitetura de software em geral e então partindo para o tema central do trabalho, que é Domain-Driven Design e Arquitetura limpa. Após isso será tratado de clean code, seguindo do banco de dados MongoDB, usado na implementação e por último sobre assistentes virtuais.

2.1 Domain-Driven Design

O Domain-Driven Design (DDD), em português Design Orientado a Domínio, é uma abordagem de design de software que se concentra na compreensão aprofundada do domínio do problema em questão (Evans 2004). Desenvolvido por Eric Evans, o DDD oferece um conjunto de princípios e padrões para criar modelos de domínio robustos e eficazes. Dentro desse contexto, três conceitos fundamentais destacam-se: Entidades, Objetos de Valor e Agregados.

As Entidades representam objetos únicos identificáveis por meio de um identificador único, os Objetos de Valor são estruturas imutáveis que descrevem características importantes, e os Agregados são agrupamentos de Entidades e Objetos de Valor que são tratados como uma única unidade coesa no sistema. Estes padrões fornecem diretrizes valiosas para a criação de modelos de domínio flexíveis e compreensíveis, promovendo uma melhor expressividade e alinhamento entre o código e as complexidades do domínio em questão.

2.1.1 Entidades

As entidades desempenham um papel crucial na modelagem de domínios complexos, proporcionando uma representação rica e significativa dos elementos essenciais para o negócio. O conceito de Entidades vai além da mera encapsulação de dados; ele aborda a ideia de objetos que possuem uma identidade única e persistente ao longo do tempo, sendo capazes de evoluir e refletir o estado do domínio.

Uma característica fundamental das entidades é sua distinção por meio de identificadores únicos. Cada entidade é diferenciada por um identificador que permanece constante

durante sua existência, mesmo que outros atributos possam ser modificados. Essa identidade fornece um meio crucial de rastrear e referenciar objetos no contexto do domínio, facilitando a comunicação e a consistência dentro do sistema.

2.1.2 Objetos de valor

Os objetos de valor constituem um elemento fundamental na modelagem de domínios complexos, proporcionando uma representação rica e imutável de conceitos importantes para o negócio. Ao contrário das Entidades, os objetos de valor não possuem uma identidade única e são definidos inteiramente pelos seus atributos. Sua imutabilidade é uma característica distintiva, o que significa que, uma vez criados, os objetos de valor não são alterados, mas sim substituídos por novas instâncias em caso de modificação. Essa característica os torna particularmente úteis para expressar conceitos no domínio que não requerem rastreamento de identidade, contribuindo para uma modelagem mais precisa e compreensível.

Além disso, eles desempenham um papel crucial na criação de modelos de domínio mais expressivos, encapsulando comportamentos e regras específicos do negócio de forma coesa. Eles são frequentemente utilizados para representar informações que, embora fundamentais para o domínio, não necessitam de uma identidade única para serem eficazes. A ênfase na imutabilidade dos objetos de valor promove uma abordagem mais segura para a manipulação de dados, evitando efeitos colaterais indesejados e contribuindo para a construção de sistemas mais robustos e alinhados com as nuances do negócio.

2.1.3 Agregados

Conforme esclarece (Evans 2004), os agregados representam um conceito fundamental que visa organizar e estruturar entidades e objetos de valor de maneira coesa e consistente, fornecendo uma abordagem estratégica para lidar com a complexidade e as interrelações entre diferentes elementos do sistema.

Em essência, um Agregado é um grupo composto por entidades e objetos de valor que são tratados como uma única unidade transacional. Ele é definido por uma raiz de Agregado, geralmente uma Entidade, que atua como o ponto de entrada para a manipulação de todo o conjunto. A raiz do Agregado encapsula a lógica de negócios e garante a consistência das operações dentro do escopo definido pelo Agregado.

A principal motivação por trás do conceito de Agregado é proporcionar uma fronteira clara e encapsulada para garantir a integridade e consistência dos dados. Isso é particularmente crucial em sistemas distribuídos, nos quais a manutenção da consistência pode se tornar um desafio. Ao tratar o Agregado como uma unidade coesa, as operações que afetam os elementos internos são realizadas de forma atômica, garantindo que o sistema permaneça em um estado válido.

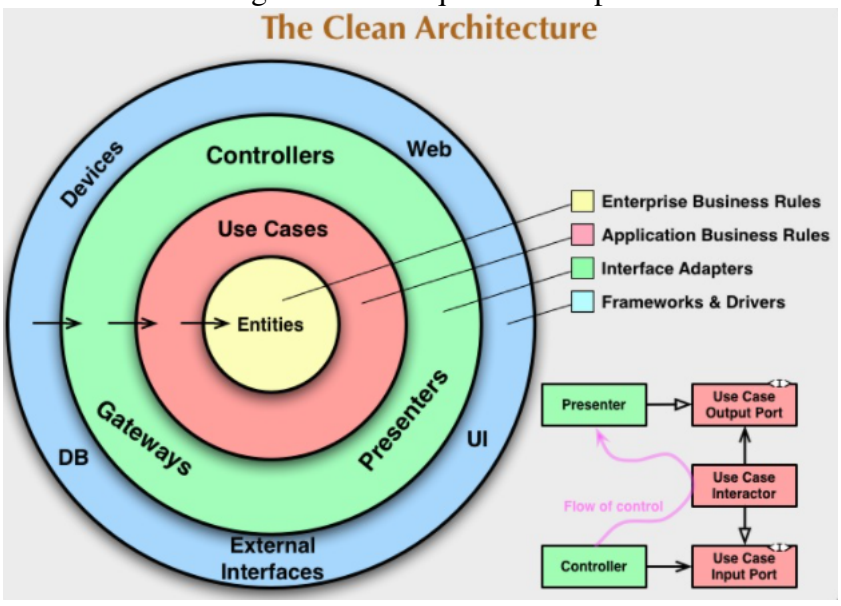
2.2 Arquitetura limpa

A Arquitetura Limpa, proposta por Robert C. Martin, também conhecido como "Uncle Bob", é uma abordagem de design de software que busca estabelecer uma estrutura modular e escalável para sistemas, promovendo a manutenibilidade, testabilidade e flexibilidade ao longo do tempo. A essência da Arquitetura Limpa reside na priorização da separação de preocupações e na organização hierárquica do código em camadas bem definidas, cada uma com responsabilidades claramente delimitadas.

Ao adotar a Arquitetura Limpa, o código é estruturado em anéis concêntricos, representando diferentes níveis de abstração e dependência. O núcleo interno contém as entidades de negócio e as regras de negócio, isoladas de qualquer detalhe de implementação externo. À medida que se expande para camadas mais externas, como os casos de uso, interfaces e frameworks, a dependência é direcionada para dentro, preservando a integridade e coesão do núcleo do sistema.

Essa abordagem visa mitigar a dependência de detalhes externos, como frameworks e bibliotecas, garantindo que a lógica de negócios permaneça independente e que as mudanças nas tecnologias externas não comprometam a estabilidade do sistema. A Arquitetura Limpa busca criar sistemas que não apenas atendam aos requisitos funcionais imediatos, mas que também sejam projetados para resistir às inevitáveis mudanças e evoluções no ciclo de vida do software. Essa introdução à Arquitetura Limpa destaca seu foco na simplicidade, na clareza e na construção de sistemas que transcendem a implementação técnica para atender efetivamente aos objetivos de negócios de maneira sustentável.

Figura 2.1: A arquitetura limpa



A Figura 2.1 mostra exatamente cada um desses anéis, são eles: enterprise business rules, application business rules, interface adapters e frameworks & drivers.

- **Enterprise Business Rules**: É a camada mais interna e fundamental da Clean Architecture, também chamada de camada de domínio. Esse domínio está intrinse-

camente relacionado com o domínio do DDD, uma vez que essa camada abriga as entidades da aplicação e essas entidades contém as regras de negócio centrais do sistema. Essas entidades e regras são independentes de qualquer detalhe técnico e não devem depender de *frameworks* ou tecnologias externas.

- **Application Business Rules**: É a camada que vem logo após a "Enterprise Business"e ela é responsável por coordenar a execução dos casos de uso. Os casos de uso podem ser compreendidos como as funcionalidades do sistema e são responsáveis por orquestrar a interação entre as entidades do domínio.
- **Interface Adapters**: A camada Interface Adapters desempenha um papel crucial na Clean Architecture, servindo como a fronteira que separa os componentes internos dos externos. Essa camada atua como a interface de entrada e saída do domínio do sistema, proporcionando uma clara demarcação entre a lógica interna e as interfaces externas. Essencialmente, ela facilita a interação entre o núcleo da aplicação e os elementos externos, assegurando uma arquitetura flexível e adaptável a mudanças sem comprometer a integridade do domínio.
- **Framweworks & Drivers**: A camada Framework & Drivers, situada mais externamente na Clean Architecture, é incumbida de gerenciar aspectos técnicos que não estão diretamente associados ao domínio específico da aplicação. Dentro desse escopo, englobam-se decisões relacionadas a elementos como banco de dados, servidor HTTP, protocolos de comunicação, entre outros. Essa camada serve como uma interface adaptadora, garantindo a integração eficaz entre a lógica de negócios interna e os detalhes de implementação externos, mantendo uma clara separação entre as preocupações do domínio e os elementos técnicos da aplicação.

2.2.1 Princípios S.O.L.I.D

O acrônimo SOLID representa um conjunto de cinco princípios de design de software que foram formulados por Robert C. Martin, visando criar sistemas mais compreensíveis, flexíveis e sustentáveis. Cada letra do SOLID corresponde a um princípio específico:

Princípio da Responsabilidade Única

De acordo com o Princípio da Responsabilidade Única (SRP), cada componente de um sistema tenha uma única razão para mudar. Cada classe, módulo ou componente deve ter apenas uma responsabilidade claramente definida, evitando a sobrecarga com múltiplas tarefas ou funcionalidades.

Segundo (Martin 2017), o SRP promove a manutenção e a evolução do código de forma mais eficiente. Ao designar responsabilidades específicas para cada componente, a base do software torna-se mais resiliente a mudanças, permitindo que ajustes ou melhorias em uma área não impactem outras de maneira inesperada. Isso resulta em um código mais modular, compreensível e fácil de manter ao longo do tempo.

Princípio do Aberto/Fechado

O Princípio do Aberto/Fechado (OCP) diz que as entidades de software, ou seja, classes, módulos, funções, etc devem ser abertas para extensão, mas fechadas para modificação, promovendo a capacidade de adicionar novos recursos sem alterar o código existente. Seu objetivo consiste em fazer com que o sistema seja fácil de estender sem que a mudança cause um alto impacto. Para concretizar esse objetivo, particionamos o sistema em componentes e organizamos esses componentes em uma hierarquia de dependência que proteja os componentes de nível mais alto das mudanças em componentes de nível mais baixo (Martin 2017).

Princípio da Substituição de Liskov

O Princípio da Substituição de Liskov (LSP) destaca a importância da consistência de comportamento entre as classes derivadas e suas classes base. Em termos simples, se uma classe base é substituída por uma classe derivada, o código que utiliza a classe base deve continuar funcionando corretamente sem necessidade de alterações.

Princípio da Segregação de Interface

O Princípio da Segregação de Interface (ISP) sugere que uma classe não deve ser forçada a implementar interfaces que ela não utiliza. Em outras palavras, a segregação de interface propõe que uma interface deve conter apenas os métodos que são relevantes para a implementação da classe que a utiliza.

Segundo (Martin 2017), o ISP contribui para a construção de componentes mais especializados e coesos, evitando a criação de interfaces "gordas"que obrigam as classes a implementarem métodos desnecessários, reduzindo a coesão e aumentando a complexidade do sistema.

Princípio da Inversão de Dependência

Segundo o Princípio da Inversão de Dependência (DIP), os sistemas mais flexíveis são aqueles em que as dependências de código-fonte se referem apenas a abstrações e não a itens concretos (Martin 2017).

2.2.2 Padrões de projeto

Design patterns, ou padrões de projeto, são soluções generalizadas para problemas comuns que ocorrem durante o desenvolvimento de software. Eles representam boas práticas de design, baseadas em experiências anteriores, que visam fornecer soluções reutilizáveis para desafios específicos. Design patterns não são implementações concretas ou códigos prontos, mas sim modelos ou *templates* que podem ser aplicados a problemas similares em diferentes contextos.

De acordo com (Gamma et al. 1994), foram identificados 23 padrões de design que podem ser aplicados para resolver recorrentemente desafios específicos no desenvolvimento

de software. Esses padrões são organizados em três categorias principais: Padrões de Criação, Padrões Estruturais e Padrões Comportamentais. seguir será abordado a respeito dos principais padrões que foram utilizados no decorrer da implementação do sistema.

Padrão *Command*

O padrão de projeto *Command* é um padrão comportamental que encapsula uma solicitação como um objeto, permitindo que os clientes parametrizem os clientes com diferentes solicitações, enfileirem solicitações e suportem operações reversíveis.

Ao aplicar o *Command* na arquitetura limpa, ele é comumente utilizado para estruturar as interações entre os Casos de Uso e as Entidades no núcleo da arquitetura, permitindo assim o desacoplamento de quem solicita a ação (Invoker) da implementação da ação (Receiver). A

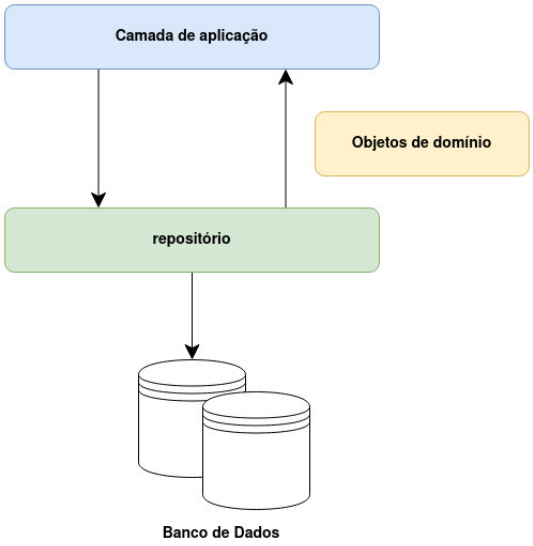
Padrão *Factory*

O padrão de projeto *Factory*

2.2.3 Padrão *Repository*

O padrão de projeto *Repository* é uma abordagem que visa separar a lógica de acesso a dados da lógica de negócios em um sistema. Essencialmente, o *Repository* atua como uma camada intermediária entre o código da aplicação e a fonte de dados, como mostrado na Figura 2.2, seja um banco de dados, serviço web ou outro mecanismo de armazenamento. Ele fornece uma interface consistente para a manipulação de dados, abstraindo os detalhes de implementação do acesso ao banco de dados e permitindo que a lógica de negócios trabalhe com objetos de domínio em vez de diretamente com consultas e comandos SQL ou APIs específicas.

Figura 2.2: Diagrama do padrão *Repository*

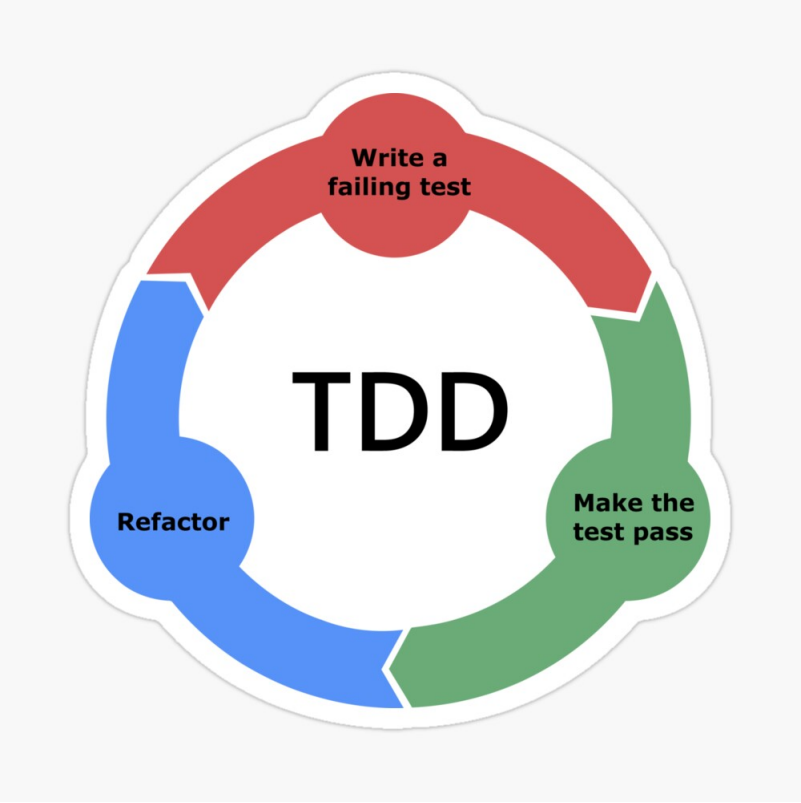


Ao adotar o padrão *Repository*, benefícios significativos são alcançados, como a melhoria da testabilidade, pois é possível criar implementações de repositório que simulam o comportamento do banco de dados durante os testes. Além disso, o *Repository* facilita a substituição de tecnologias de armazenamento de dados sem impactar a lógica de negócios, promovendo uma arquitetura mais flexível e adaptável a mudanças nos requisitos e na infraestrutura de dados.

2.2.4 Test-Driven Development

Segundo (Beck 2022), o Desenvolvimento Orientado a Testes (TDD) é uma abordagem de desenvolvimento de software que coloca a criação de testes automatizados no centro do processo de codificação. Na prática do TDD, os desenvolvedores escrevem testes antes mesmo de implementar o código funcional, seguindo um ciclo iterativo de "Red, Green, Refactor"(Vermelho, Verde, Refatorar), conforme a Figura 2.3.

Figura 2.3: Diagrama Red Green Refactor



Em um primeiro momento, o teste é escrito para uma pequena unidade de funcionalidade que ainda não foi implementada, o que resulta em uma execução falha (vermelha). Em seguida, o código é desenvolvido para passar no teste, transformando a execução para bem-sucedida (verde). Por fim, é realizada a etapa de refatoração, na qual o código é aprimorado sem alterar o comportamento, mantendo os testes verdes. Essa abordagem

promove a criação de código mais robusto, modular e testável, além de fornecer *feedback* instantâneo sobre a qualidade do código durante todo o processo de desenvolvimento.

Essa abordagem é bastante útil não apenas para o desenvolvedor atual do software, mas também para futuros programadores que venham a ter contato com a implementação, uma vez que quanto novos requisitos forem integrados ao sistema, os testes já escritos darão a garantia e a confiabilidade necessária que o sistema ainda funciona.

2.3 MongoDB

O MongoDB é um sistema de gerenciamento de banco de dados (SGBD) NoSQL amplamente utilizado, conhecido por sua flexibilidade e escalabilidade. Diferentemente dos bancos de dados relacionais, o MongoDB é orientado a documentos, armazenando dados em formato Binary JSON (BSON), o que facilita a representação de estruturas complexas.

Como mencionado em (Banker et al. 2016), a arquitetura do MongoDB é baseada em documentos elimina a necessidade de esquemas rígidos, permitindo que os desenvolvedores adicionem ou removam campos facilmente, proporcionando agilidade no desenvolvimento. Além disso, o MongoDB oferece recursos poderosos de consulta, indexação e suporte a operações distribuídas, tornando-o uma escolha popular para aplicativos que demandam armazenamento de dados flexível, rápido acesso e escalabilidade horizontal eficiente.

Nesse capítulo foi abordado acerca dos principais tópicos que são fundamentais para compreensão do problema bem como da implementação da solução. No próximo será abordado os problemas relacionado a versão inicial do sistema Bot Educacional, antes da refatoração e aplicações das técnicas de DDD e Clean Architecture e as boas práticas de programação.

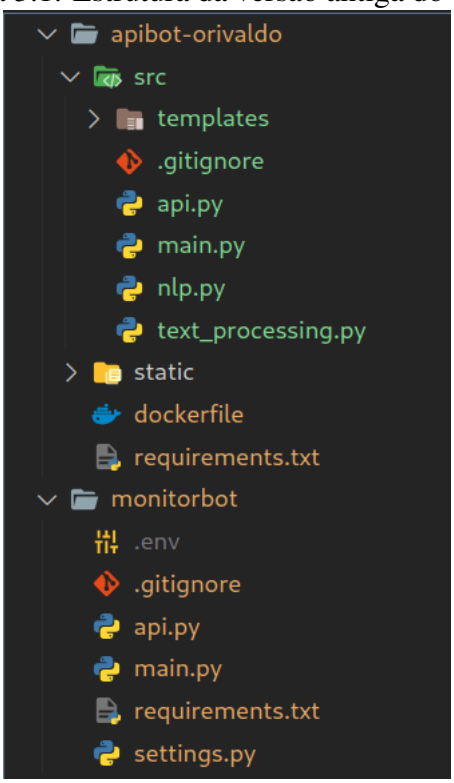
Capítulo 3

Problema

Neste capítulo tem objetivo de apresentar os principais problemas relacionado a versão antiga do sistema Bot Educacional, pontuando cada um e explicando o porquê é uma má implementação.

Em um primeiro momento o sistema Bot Educacional era um sistema composto de um único monólito simples que era responsável por manter em execução todos os módulos dos sistema, como os Bot, a API e parte responsável pelo NLP. Como pode-se observar na Figura 3.1, a raiz do projeto contém os diretórios `src` e `monitorbot`, sendo a `src` contendo as implementação da API para casos de uso e as regras de negócio da aplicação e `monitorbot` responsável por manter o bot executando como um servidor.

Figura 3.1: Estrutura da versão antiga do código



Porém essa implementação possui diversos pontos negativos, dentre eles pode-se des-

tacar:

- **Implementação puramente funcional**: O código dessa versão do sistema educacional Bot é um código puramente funcional, isso significa que não é Programação Orientada à Objetos (POO) e portanto não pode usufruir dos benefícios deste paradigma de programação como classes, objetos, métodos, herança.
- **Forte acoplamento**: Um sistema de software com forte acoplamento é, em muitos casos, considerado inadequado em termos de design e manutenção. O código da versão apresentada na Figura 3.1 possui alta coesão porque há um grau de interdependência entre módulos ou componentes de um sistema, isso significa que mudanças em uma parte do sistema têm um impacto significativo em outras partes, o que exige um esforço grande de manutenção.
- **Baixa coesão**: Assim como o forte acoplamento, a baixa coesão também traz impactos negativos para o software. Ela está relacionada à medida em que os elementos dentro de um módulo ou componente estão agrupados logicamente, ou seja, um módulo realiza diversas funcionalidades distintas que podem não ter uma conexão direta.
- **Domínio diluído entre as diversas partes do código**: Outra evidência que o código não bem projetado se dá pela dificuldade de entender o que sistema faz, uma vez que o domínio não está disposto de maneira expressiva, mas sim diluído entre os vários módulos e funções.
- **Ausência de testes automatizados**: Outro ponto notável na implementação antiga é ausência testes automatizados, o que torna aplicação pouco confiável, visto que não se tem garantia de funcionamento. Também é complicado de adicionar funcionalidades, visto que é necessário, a cada mudança, executar manualmente o código e checar se está tudo funcionando.

Nesse capítulo foram apresentados alguns problemas relacionados ao sistema antigo, tais quais servirão de motivação para o desenvolvimento desse trabalho. O próximo capítulo será abordado a implementação da nova arquitetura do sistema, com ênfase na API.

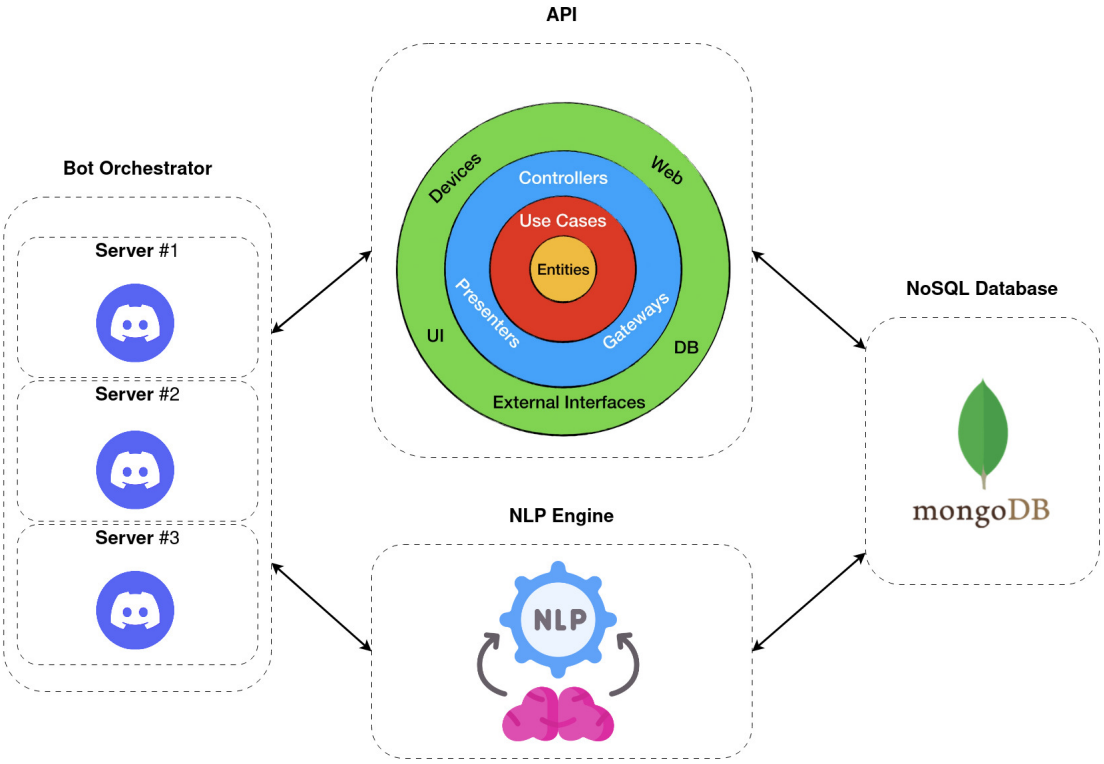
Capítulo 4

Implementação

Neste capítulo será dedicado a explicar a implementação da nova versão do sistema Bot Educacional, partindo de uma visão geral da arquitetura, explicando cada um dos serviços. Em seguida será dado um foco maior na API do sistema, que é o principal objeto de pesquisa deste trabalho, destacando detalhes da implementação do ponto de vista do DDD e da Arquitetura Limpa.

4.1 Arquitetura

Figura 4.1: Proposta de arquitetura do sistema Bot Educacional



4.1.1 Orquestrador de Bots

O orquestrador é um serviço responsável por manter os bots em execução no servidor. Com o orquestrador é possível criar, iniciar e parar um bot. Cada bot executa as mesmas funções, sendo a criação do bot uma abstração, criando uma nova instância do cliente usando parâmetros diferentes. A implementação é realizada utilizando a biblioteca do python `discord.py`, que ao executar o bot ele cria um conexão websocket, dessa forma é possível utilizá-lo na forma de chat.

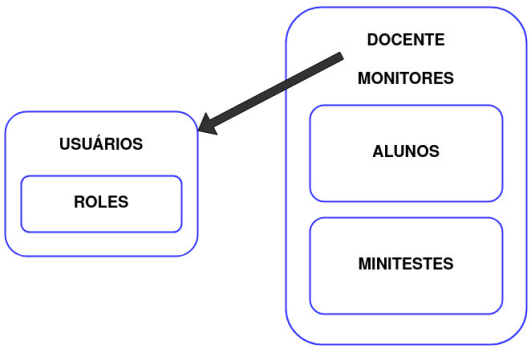
4.1.2 NLP Engine

A NLP Engine é um sistema pequeno responsável por processar a entrada de texto dos usuários do Bot e partir disso realizar uma busca no banco de dados e tentar buscar uma resposta e devolver para o usuário.

4.1.3 API

A API na arquitetura do sistema Bot Educacional é o sistema mais complexo de ser implementado, visto que ele é responsável pela maior parte das funcionalidades. Através da API os professores podem fazer controle de suas turmas, juntamente aos monitores. Os Alunos, por sua vez, utilizam a API para registrar presença e responder minitestes.

Figura 4.2: Diagrama em alto nível de abstração da API



4.2 Detalhando a API

4.2.1 Entidades de domínio

As entidades de domínio são as principais entidades da modelagem de domínio na Clean Architecture. Nessa implementação, todas as entidades são *dataclasses*, que facilitam as implementação de classes de dados. Todas as entidades possuem um Id, que as identificam unicamente durante o fluxo de execução do código. Além disso cada entidade possuem um método de validação, pois na arquitetura cada entidade deve ser responsável por auto-validar e métodos expressivos que fazem com que os objetivos se comportem como esperado de acordo com as requisitos.

Figura 4.3: Entidade Usuario em usuario.py

```python
@dataclass
class Usuario:
    id: str
    discord_id: str = field(default=None)
    roles: Set[Role] = field(default_factory=set)

    def __post_init__(self):
        self.validate()

    1 usage
    def validate(self):
        if self.id == '':
            raise Exception('Id nao pode ser vazio')

    def grant_role(self, role: Role):
        self.roles.add(role)

    2 usages
    def revoke_role(self, role: Role):
        if role in self.roles:
            self.roles.remove(role)

    4 usages
    def subscribe(self, discord_id):
        if discord_id == '' or discord_id is None:
            raise Exception('Discord Id invalido')
        self.discord_id = discord_id
```

Figura 4.4: Objeto de valor Role em role.py

```python
class Role(Enum):
    DOCENTE = 1
    DISCENTE = 2
    MONITOR = 3
    ADMIN = 4
```

Na Figura 4.3 mostra a implementação da entidade Usuário com seus atributos e respectivos métodos. O usuário do sistema são aqueles que possuem algum tipo de papel ou autorização no sistema e, via de regra, está associado a um ID do Discord de um usuário. O entidade `Usuário` possui o objeto de valor `Role`, que define um conjunto de permissões a qual o usuário está submetido.

Figura 4.5: Entidade Turma em turma.py

```python
@dataclass
class Turma:
    id: str
    denominacao: str
    ano: int
    periodo: int
    data_inicio: date
    data_fim: date
    esta_consolidada: bool = False
    docente: Usuario = None
    alunos: List[Aluno] = field(default_factory=list)
    monitores: List[Usuario] = field(default_factory=list)
    minitestes: List[Miniteste] = field(default_factory=list)

    def __post_init__(self):
        self.validate()

    1 usage
    def validate(self):
        if self.id == '':
            raise Exception('Id nao pode ser vazio')
        if self.denominacao == '':
            raise Exception('Denominacao nao permitida')
        if self.data_fim < self.data_inicio:
            raise Exception('Data fim deve ser apos a Data inicio da turma')
```

Figura 4.6: Métodos da entidade Turma em turma.py

```python
    def add_monitor(self, monitor: Usuario):
        if Role.MONITOR in monitor.roles:
            self.monitores.append(monitor)
        else:
            raise Exception(f'O usuario nao possui a role {Role.MONITOR}')

    4 usages
    def add_miniteste(self, minitestes: List[Miniteste]):
        for miniteste in minitestes:
            self.minitestes.append(miniteste)

    1 usage
    def consolidar(self):
        self.esta_consolidada = True

    6 usages (1 dynamic)
    def definir_docente(self, docente: Usuario):
        if Role.DOCENTE in docente.roles:
            self.docente = docente
        else:
            raise Exception(f'O usuario nao possui a role {Role.DOCENTE}')
```

A entidade Turma, como evidenciado na Figura 4.5, é uma das classes mais importantes, pois ela engloba outras entidades como Aluno e Minitestes, de tal forma que forma o agregado Turma. Além de Id e atributos gerais para controle e identificação, a classe possui uma lista de alunos que fazem parte da turma, uma lista de monitores, uma lista de minitestes e um docente, que é responsável pela turma. Os métodos são `add_monitor`, para adicionar monitores a turma, `add_miniteste` para adicionar minitestes à turma, `consolidar` para fechar a turma, a tornando inacessível para os alunos e `definir_docente` para definir o professor da turma. Vale ressaltar que para um usuário ser capaz de ser adicionado como monitor ele deve possui a *role* `MONITOR`, de forma semelhante os professores devem possuir a *role* `DOCENTE`.

Figura 4.7: Entidade Aluno em aluno.py

```python
@dataclass
class Aluno:
    id: str
    nome: str
    matricula: str
    turma: str
    frequencias: List[date] = field(default_factory=list)
    minitestes: List[MinitesteResult] = field(default_factory=list)

    def __post_init__(self):
        self.validate()

    1 usage
    def validate(self):
        if self.id == '':
            raise Exception('Id nao pode ser vazio')
        if self.nome == '':
            raise Exception('Nome nao pode ser vazio')
        if self.matricula == '':
            raise Exception('Matricula nao pode ser vazio')
        if self.turma == '':
            raise Exception('Turma nao pode ser vazio')

    2 usages
    def registrar_presenca(self, today: date) -> None:
        if today in self.frequencias:
            raise InvalidDateRegistrarPresencaException(today)
        self.frequencias.append(today)

    1 usage
    def responder_minitest(self, minitest_result: MinitesteResult):
        self.minitestes.append(minitest_result)
```

De forma semelhante, a entidade aluno possui os atributos de alunos e os métodos para registrar presença, `registrar_presenca` e para responder minitestes, `responder_miniteste`. A frequência possui uma regra que não pode ser registrada mais de uma vez no mesmo.

Figura 4.8: Entidade Miniteste em miniteste.py

```python
@dataclass
class Miniteste:
    id: str
    teste_id: str
    pergunta: str
    resposta: str
    alternativas: Alternativas
    total_respostas: int = 0
    total_acertos: int = 0

    def __post_init__(self):
        self.validate()

    1 usage
    def validate(self):
        if self.id == '':
            raise Exception('Id nao pode ser vazio')
        if self.teste_id == '':
            raise Exception('Teste Id nao pode ser vazio')
        if self.pergunta == '':
            raise Exception('Pergunta nao pode ser vazio')
        if self.resposta == '':
            raise Exception('Resposta nao pode ser vazio')

    1 usage
    def responder(self, opcao: str):
        if opcao == self.resposta:
            self.total_acertos += 1
            self.total_respostas += 1
            return True
        else:
            self.total_respostas += 1
            return False

    def get_taxa_acertos(self):
        return self.total_acertos / self.total_respostas
```

Figura 4.9: Objeto de valor Alternativas em alternativas.py

```python
@dataclass
class Alternativas:
    A: str
    B: str
    C: str
    D: str

    def __post_init__(self):
        self.validate()

    1 usage
    def validate(self):
        if self.A == '' or self.B == '' or self.C == '' or self.D == '':
            raise Exception('Alternativa nao pode ser vazia')
```

4.2.2 Repositórios

No domínio do Arquitetura Limpa, ou seja, o centro, além das entidades de negócio também estão situados as interfaces abstratas repositório. Elas definem um contrato, especificando quais métodos devem ser implementados para comunicação com um banco de dados. As interfaces de repositórios independem de banco de dado ou tipo de banco de dados. A linguagem Python não possui suporte a nenhum tipo de estrutura de dados do tipo Interface, como no Java e no Typescript. A solução para isso é utilizar o pacote padrão `abc`, que refere-se a "Abstract Base Classes"(Classes Abstratas de Base). O módulo `abc` fornece funcionalidades para a definição de classes abstratas.

Figura 4.10: AbstractRepository em *abstract_repository.py*

```python
from abc import ABC, abstractmethod
from typing import Optional, TypeVar, Sequence

T = TypeVar('T')


4 usages
class AbstractRepository(ABC):
    @abstractmethod
    def insert(self, entity: T) -> Optional[T]:
        pass

    @abstractmethod
    def update(self, entity: T) -> None:
        pass

    @abstractmethod
    def get_by_id(self, id) -> Optional[T]:
        pass

    @abstractmethod
    def delete(self, id) -> None:
        pass

    @abstractmethod
    def list(self) -> Optional[Sequence[T]]:
        pass
```

As classes abstratas de repositório para Turma e Usuário entendem a classe genérica `AbstractRepository`, mas ainda não implementam. Essas classes devem apenas acrescentar funcionalidade que for requerida pela classe, como em `AbstractUsuarioRepository` que adiciona o método `get_by_discord_id()`, esse método só é necessário para o repositório de usuário.

Figura 4.11: AbstractUsuarioRepository em *abstract_usuario_repository.py*

```python
class AbstractUsuarioRepository(AbstractRepository):

    2 usages
    @abstractmethod
    def insert(self, user: Usuario) -> Optional[Usuario]:
        pass

    @abstractmethod
    def get_by_id(self, user_id: str) -> Optional[Usuario]:
        pass

    2 usages
    @abstractmethod
    def get_by_discord_id(self, user_discord_id: str) -> Optional[Usuario]:
        pass

    2 usages
    @abstractmethod
    def update(self, user: Usuario) -> None:
        pass

    @abstractmethod
    def list(self) -> List[Usuario]:
        pass
```

Figura 4.12: AbstractTurmaRepository em *abstract_turma_repository.py*

```python
class AbstractTurmaRepository(AbstractRepository):

    1 usage
    @abstractmethod
    def insert(self, entity: Turma) -> Optional[Turma]:
        pass

    @abstractmethod
    def get_by_id(self, turma_id) -> Optional[Turma]:
        pass

    5 usages
    @abstractmethod
    def update(self, turma: Turma) -> None:
        pass

    @abstractmethod
    def delete(self, turma_id):
        pass

    @abstractmethod
    def list(self) -> List[Turma]:
        pass
```

4.2.3 Infraestrutura

A camada de infraestrutura na Arquitetura Limpa é responsável por abrigar detalhes técnicos e implementações específicas, como acesso a bancos de dados, frameworks, serviços externos e outros componentes externos ao núcleo da aplicação. Essa camada encapsula todas as preocupações relacionadas à tecnologia, permitindo que a lógica de negócios permaneça independente e desacoplada.

O código da Figura 4.13 é responsável por lidar com a conexão com banco de dados MongoDB. Utiliza a biblioteca *pymongo* para estabelecer uma conexão ao banco, obtendo a string de conexão de variáveis de ambiente definidas em um arquivo .env. A configuração do ambiente é feita usando os módulos os e dotenv. O código segue boas práticas ao isolar informações sensíveis, como a string de conexão, em variáveis de ambiente para maior segurança. Essa abordagem permite a modularização e a separação adequada de responsabilidades, facilitando a manutenção e evolução do sistema.

Figura 4.13: Implementação de *mongodb_instance.py*

```python
import os
import dotenv
from pymongo import MongoClient

dotenv_file = dotenv.find_dotenv()
dotenv.load_dotenv(dotenv_file)

CONNECTION_STRING = os.environ.get("MONGODB_STRING_CONNECTION")

connection = MongoClient(CONNECTION_STRING)
```

Na camada de infraestrutura também se situam as Implementações concretas dos repositórios abstratos na camada de domínio. As implementações concretas de repositório foram desenvolvidas usando o banco de dados NoSql MongoDB. O repositório recebe uma instância de conexão com MongoDB e então define qual a coleção que será utilizada para a persistência dos dados.

Figura 4.14: Implementação de *mongodb_usuario_repository.py*

```python
class MongodbUsuarioRepository(AbstractUsuarioRepository):

    def __init__(self, client):
        self.client = client
        self.db = client.apibotv2
        self.collection = self.db.usuarios

    6 usages
    def insert(self, user: Usuario) -> Optional[Usuario]:
        user_dict = user.to_dict()
        self.collection.insert_one(user_dict)

    def get_by_id(self, user_id: str) -> Optional[Usuario]:
        result = self.collection.find_one({'_id': user_id})
        if result is None:
            return None
        else:
            return Usuario.from_dic(result)

    def get_by_discord_id(self, user_discord_id: str) -> Optional[Usuario]:
        result = self.collection.find_one({'discord_id': user_discord_id})
        if result is None:
            return None
        else:
            return Usuario.from_dic(result)

    4 usages
    def update(self, user: Usuario):
        self.collection.update_one({"_id": user.id}, {"$set": user.to_dict()})

    def list(self):
        users = self.collection.find()
        return [Usuario.from_dic(user) for user in users]

    def delete(self, user_id) -> None:
        pass
```

Figura 4.15: Implementação de *mongodb_turma_repository.py*

```python
class MongodbTurmaRepository(AbstractTurmaRepository):

    def __init__(self, client):
        self.client = client
        self.db = client.apibotv2
        self.collection = self.db.turmas
        self.usuario_repository = MongodbUsuarioRepository(client)

    3 usages
    def insert(self, turma: Turma) -> Optional[Turma]:
        turma_dict = turma.to_dict()
        self.collection.insert_one(turma_dict)

    def get_by_id(self, turma_id) -> Optional[Turma]:
        turma_opt = self.collection.find_one({'_id': turma_id})
        if turma_opt is None:
            return None
        turma = Turma.from_dict(turma_opt)
        if turma_opt['docente']:
            docente = self.usuario_repository.get_by_id(turma_opt['docente'])
            turma.definir_docente(docente)
        for id in turma_opt.get('monitores', []):
            monitor = self.usuario_repository.get_by_id(id)
            turma.add_monitor(monitor)
        return turma

    1 usage
    def update(self, turma: Turma) -> None:
        self.collection.update_one({"_id": turma.id}, {"$set": turma.to_dict()})

    def delete(self, turma_id) -> None:
        pass

    def list(self) -> List[Turma]:
        pass
```

4.2.4 Casos de uso

Os casos de uso representam as funcionalidades do sistema, ou seja, uma intenção do usuário dentro do sistema. Eles são responsáveis por coordenar e orquestrar as operações necessárias para atender a um objetivo específico do usuário ou do sistema. Os casos de uso são implementados na camada de aplicação, que fica no núcleo da arquitetura e é independente de detalhes de infraestrutura ou tecnologia. Foram implementados 12 casos de uso, como explicitados no diagrama de casos de uso na Figura 4.16:

Figura 4.16: Diagrama de casos de uso

- **Caso de uso Cadastrar Turma**: Um usuário com autorização de docente pode cadastrar uma turma enviando os dados da turma e uma lista inicial de alunos. Cada aluno da turma terá o papel de discente no sistema.
- **Caso de uso Cadastrar Alunos em Turma**: Um usuário com autorização de docente pode adicionar novos alunos em uma turma previamente cadastrada. Cada aluno da turma terá o papel de discente no sistema.
- **Caso de uso Adicionar Docente em Turma**: Um usuário administrador do sistema pode cadastrar um usuário como docente de uma turma, caso haja necessidade.
- **Caso de uso Adicionar Monitor em Turma**: Um usuário docente pode adicionar usuário de papel monitor em sua turma, esses por sua vez poderão realizar ações de caráter restrito na turma que monitoram.
- **Caso de uso Adicionar Minitestes em Turma**: Um usuário docente pode adicionar

miniteste em uma turma previamente cadastrada. Os miniteste funcionam como uma forma de obter dados sobre a retenção de conteúdos dos alunos e ajudar o professor a tomar decisões.
- **Caso de uso Registrar Presença de Aluno**: Um usuário discente pode registrar presença no sistema.
- **Caso de uso Responder Miniteste**: Um usuário discente pode responder um miniteste que estiver cadastrado em sua turma.
- **Caso de uso Obter Frequência de Turma**: O docente da turma pode obter a frequência de todos os alunos de sua turma.
- **Caso de uso Criar Usuário**: Um usuário administrador pode criar um novo usuário no sistema.
- **Caso de uso Conceder Papel para Usuário**: Um usuário administrador pode conceder um papel para um usuário no sistema.
- **Caso de uso Revogar Papel para Usuário**: Um usuário administrador pode revogar um papel para um usuário no sistema.
- **Caso de uso Auto-registro de Usuário**: Um discente pode realizar auto-registro no sistema para associar sua matrícula ao seu identificador usuário de Discord – ou simplesmente Discord ID. Dessa maneira ele está apto para realizar ações na turma que está cadastrado.

As seguintes figuras consistem na implementação de classe correspondente a cada caso de uso do sistema. Cada caso de uso foi implementado usando o *command pattern*, que encapsula toda a lógica do caso de uso, orquestrando os repositórios e as entidades de domínio para garantir o devido funcionamento da funcionalidade.

Vale ressaltar que, o método construtor de cada caso de uso recebe como argumento uma abstração de repositório, como representado na Figura 4.17. No momento de instanciação do caso de uso é passada uma implementação de repositório, que pode ser para qualquer banco de dados, seja relacional ou não-relacional, ou seja, os casos de usos são independentes de tecnologia.

Caso de uso Cadastrar Turma

Figura 4.17: Código para CadastrarTurmaUseCase em *turma_usecase.py*

```
class CadastrarTurmaUseCase():
    def __init__(self, turma_repository: AbstractTurmaRepository,
                 usuario_repository: AbstractUsuarioRepository):
        self.turma_repository = turma_repository
        self.usuario_repository = usuario_repository

    11 usages
    def execute(self, turma_dto_input: TurmaDtoInput) -> TurmaDtoOutput:
        turma = Turma(id=str(uuid4()),
                      denominacao=turma_dto_input.denominacao,
                      ano=turma_dto_input.ano,
                      periodo=turma_dto_input.periodo,
                      data_inicio=date.fromisoformat(turma_dto_input.data_inicio),
                      data_fim=date.fromisoformat(turma_dto_input.data_fim))
        alunos = [Aluno(str(uuid4()), aluno.nome, aluno.matricula, aluno.turma)
                  for aluno in turma_dto_input.alunos]
        turma.add_alunos(alunos)

        self.turma_repository.insert(turma)
        for aluno in turma.alunos:
            user_fetched = self.usuario_repository.get_by_id(aluno.id)
            if user_fetched is None:
                new_user = Usuario(id=aluno.id)
                new_user.grant_role(Role.DISCENTE)
                self.usuario_repository.insert(new_user)
            else:
                user_fetched.id = aluno.id
                user_fetched.grant_role(Role.DISCENTE)
                self.usuario_repository.update(user_fetched)

        return TurmaDtoOutput(id=turma.id, denominacao=turma.denominacao)
```

Caso de uso Cadastrar Aluno Em Turma

Figura 4.18: Código para CadastrarAlunoEmTurmaUseCase em *turma_usecase.py*

```
class CadastrarAlunoEmTurmaUseCase():
    def __init__(self, turma_repository: AbstractTurmaRepository,
                 usuario_repository: AbstractUsuarioRepository):
        self.turma_repository = turma_repository
        self.usuario_repository = usuario_repository

    3 usages
    def execute(self, turma_id: str, alunos_dto: List[AlunoDto]) → None:
        turma_fetched = self.turma_repository.get_by_id(turma_id)
        if turma_fetched is None:
            raise TurmaNotFoundException

        alunos = [Aluno(str(uuid4()), aluno.nome, aluno.matricula, aluno.turma)
                  for aluno in alunos_dto]
        turma_fetched.add_alunos(alunos)
        self.turma_repository.update(turma_fetched)

        for aluno in alunos:
            user_fetched = self.usuario_repository.get_by_id(aluno.id)
            if user_fetched is None:
                new_user = Usuario(id=aluno.id)
                new_user.grant_role(Role.DISCENTE)
                self.usuario_repository.insert(new_user)
            else:
                user_fetched.id = aluno.id
                user_fetched.grant_role(Role.DISCENTE)
                self.usuario_repository.update(user_fetched)
```

Caso de uso Adicionar Docente em Turma

Figura 4.19: Código para AddDocenteToTurmaUseCase em *turma_usecase.py*

```python
class AddDocenteToTurmaUseCase():
    def __init__(self, turma_repository: AbstractTurmaRepository,
                 usuario_repository: AbstractUsuarioRepository):
        self.turma_repository = turma_repository
        self.usuario_repository = usuario_repository

    2 usages
    def execute(self, turma_id: str, docente_discord_id: str) -> None:
        turma_fetched = self.turma_repository.get_by_id(turma_id)
        if turma_fetched is None:
            raise TurmaNotFoundException

        user_fetched = self.usuario_repository.get_by_discord_id(docente_discord_id)
        if user_fetched is None:
            raise UserNotFoundException

        turma_fetched.definir_docente(user_fetched)
        self.turma_repository.update(turma_fetched)
```

Caso de uso Adicionar Monitor em Turma

Figura 4.20: Código para AddMonitorToTurmaUseCase em *turma_usecase.py*

```python
class AddMonitorToTurmaUseCase():
    def __init__(self, turma_repository: AbstractTurmaRepository,
                 usuario_repository: AbstractUsuarioRepository):
        self.turma_repository = turma_repository
        self.usuario_repository = usuario_repository

    2 usages
    def execute(self, turma_id: str, monitor_discord_id: str) -> None:
        turma_fetched = self.turma_repository.get_by_id(turma_id)
        if turma_fetched is None:
            raise TurmaNotFoundException

        user_fetched = self.usuario_repository.get_by_discord_id(monitor_discord_id)
        if user_fetched is None:
            raise UserNotFoundException

        turma_fetched.add_monitor(user_fetched)
        self.turma_repository.update(turma_fetched)
```

Caso de uso Adicionar Minitestes em Turma

Figura 4.21: Código para AddMinitesteToTurma em *turma_usecase.py*

```
class AddMinitesteToTurma():
    def __init__(self, turma_repository: AbstractTurmaRepository):
        self.turma_repository = turma_repository

    4 usages
    def execute(self, turma_id: str, minitestes_dto: List[MinitestesDto]) → None:
        turma_fetched = self.turma_repository.get_by_id(turma_id)
        if turma_fetched is None:
            raise TurmaNotFoundException

        minitestes = []
        for miniteste in minitestes_dto:
            minitestes.append(Miniteste(id=str(uuid4()),
                                        teste_id=miniteste.teste_id,
                                        pergunta=miniteste.pergunta,
                                        resposta=miniteste.resposta,
                                        alternativas=Alternativas(
                                            A=miniteste.alternativas.A,
                                            B=miniteste.alternativas.B,
                                            C=miniteste.alternativas.C,
                                            D=miniteste.alternativas.D)
                                        )
                              )
        turma_fetched.add_miniteste(minitestes)
        self.turma_repository.update(turma_fetched)
```

Caso de uso Registrar Presença

Figura 4.22: Código para RegistrarPresencaUseCase em *turma_usecase.py*

```python
class RegistrarPresencaUseCase():
    def __init__(self, turma_repository: AbstractTurmaRepository,
                 usuario_repository: AbstractUsuarioRepository):
        self.turma_repository = turma_repository
        self.usuario_repository = usuario_repository

    5 usages
    def execute(self, turma_id: str, discord_id: str) -> None:
        turma_fetched = self.turma_repository.get_by_id(turma_id)
        if turma_fetched is None:
            raise TurmaNotFoundException

        aluno_fetched = self.usuario_repository.get_by_discord_id(discord_id)
        if aluno_fetched is None:
            raise UserNotFoundException

        for aluno in turma_fetched.alunos:
            if aluno.id == aluno_fetched.id:
                aluno.registrar_presenca(date.today())
                break

        self.turma_repository.update(turma_fetched)
```

Caso de uso Responder Miniteste

Figura 4.23: Código para ResponderMinitesteUseCase em *turma_usecase.py*

```
class ResponderMinitesteUseCase():
    def __init__(self, turma_repository: AbstractTurmaRepository,
                 usuario_repository: AbstractUsuarioRepository):
        self.turma_repository = turma_repository
        self.usuario_repository = usuario_repository

    3 usages
    def execute(self, turma_id: str, discord_id: str,
                resposta_miniteste_dto: ResponderMinitesteDto) -> bool:
        turma_fetched = self.turma_repository.get_by_id(turma_id)
        if turma_fetched is None:
            raise TurmaNotFoundException

        aluno_fetched = self.usuario_repository.get_by_discord_id(discord_id)
        if aluno_fetched is None:
            raise Exception('Aluno nao foi encontrado')

        aluno_to_answer = None
        minitest_to_answer = None

        for miniteste in turma_fetched.minitestes:
            if miniteste.teste_id == resposta_miniteste_dto.teste_id:
                minitest_to_answer = miniteste
                break

        for aluno in turma_fetched.alunos:
            if aluno.id == aluno_fetched.id:
                aluno_to_answer = aluno
                break

        miniteste_status = minitest_to_answer.responder(resposta_miniteste_dto.opcao)
        result = MinitesteResult(teste_id=resposta_miniteste_dto.teste_id,
                                 opcao=resposta_miniteste_dto.opcao,
                                 status=miniteste_status)
        aluno_to_answer.responder_minitest(result)
        self.turma_repository.update(turma_fetched)
        return miniteste_status
```

Caso de uso Obter Frequência de Turma

Figura 4.24: Código para ObterFrequenciaTurmaUseCase em *turma_usecase.py*

```python
class ObterFrequenciaTurmaUseCase():
    def __init__(self, turma_repository: AbstractTurmaRepository):
        self.turma_repository = turma_repository

    2 usages
    def execute(self, turma_id: str) -> List[PresencaAlunoDto]:
        turma_fetched = self.turma_repository.get_by_id(turma_id)

        if turma_fetched is None:
            raise TurmaNotFoundException

        dados = []
        for aluno in turma_fetched.alunos:
            freq = [date.isoformat(freq) for freq in aluno.frequencias]
            dado = PresencaAlunoDto(nome=aluno.nome,
                                    matricula=aluno.matricula,
                                    frequencia=freq)
            dados.append(dado)
        return dados
```

Caso de uso Criar Usuário

Figura 4.25: Código para CreateUserUsecase em *usuario_usecase.py*

```python
class CreateUserUseCase():
    def __init__(self, usuario_repository: MongodbUsuarioRepository):
        self.usuario_repository = usuario_repository

    3 usages
    def execute(self, user_dto_input: UserDtoInput) -> UserDtoOutput:
        user = Usuario(id=str(uuid4()), discord_id=user_dto_input.discord_id)
        self.usuario_repository.insert(user)
        return UserDtoOutput(id=user.id, discord_id=user.discord_id)
```

Caso de uso Conceder Papel para Usuário

Figura 4.26: Código para GrantRoleToUsuario em *usuario_usecase.py*

```
class GrantRoleToUsuario():
    def __init__(self, usuario_repository: MongodbUsuarioRepository):
        self.usuario_repository = usuario_repository

    3 usages
    def execute(self, user_id: str, rolename: str):
        user_fetched = self.usuario_repository.get_by_id(user_id)
        if user_fetched is None:
            raise Exception('Usuario nao encontrado')

        user_fetched.grant_role(Role[rolename])
        self.usuario_repository.update(user_fetched)
```

Caso de uso Revogar Papel para Usuário

Figura 4.27: Código para RevokeRoleFromUsuario em *usuario_usecase.py*

```
class RevokeRoleFromUsuario():
    def __init__(self, usuario_repository: MongodbUsuarioRepository):
        self.usuario_repository = usuario_repository

    1 usage
    def execute(self, user_id: str, rolename: str):
        user_fetched = self.usuario_repository.get_by_id(user_id)
        if user_fetched is None:
            raise Exception('Usuario nao encontrado')

        user_fetched.revoke_role(Role[rolename])
        self.usuario_repository.update(user_fetched)
```

Caso de uso Auto-registro de Usuário

Figura 4.28: Código para RegistrarAlunoUseCase em *usuario_usecase.py*

```python
class RegistrarAlunoUseCase():
    def __init__(self, usuario_repository: MongodbUsuarioRepository,
                 turma_repo: MongodbTurmaRepository):
        self.usuario_repository = usuario_repository
        self.turma_repo = turma_repo

    4 usages
    def execute(self, turma_id: str, discord_id: str, matricula: str):
        aluno_id = None
        turma_fetched = self.turma_repo.get_by_id(turma_id)
        for aluno in turma_fetched.alunos:
            if matricula == aluno.matricula:
                aluno_id = aluno.id
                break

        user_fetched = self.usuario_repository.get_by_id(aluno_id)
        if user_fetched is None:
            raise Exception('Usuario nao encontrado')

        user_fetched.subscribe(discord_id)
        self.usuario_repository.update(user_fetched)
```

4.3 Routers e *Endpoins*

Uma parte importante da API é o servidor HTTP que será utilizado quando a aplicação for executada. Como tudo foi implementado em python a escolha de servidor web foi a *FastAPI*, que é um moderno e rápido (alta performance) framework web para construção de APIs com Python 3.8 ou superior, baseado nos type hints padrões do Python.

A melhor maneira organizar os endpoints da API é utilizando os *routers*, que são uma forma de organizar diferentes partes da sua API em módulos separados e encapsulados. Cada rota pode ser definida em um roteador específico, permitindo uma estrutura modular e facilitando a manutenção.

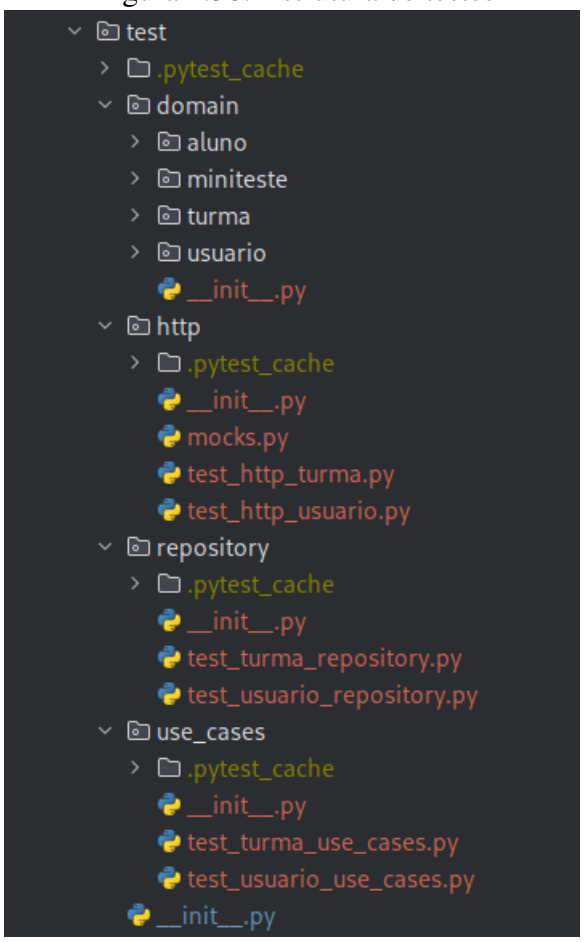
Figura 4.29: Código exemplo para do endpoint adicionar_alunos

```python
router = APIRouter(prefix='/turmas', tags=['turma'])

turma_reposity = MongodbTurmaRepository(connection)
usuario_repository = MongodbUsuarioRepository(connection)


@router.post("/{turma_id}/alunos", status_code=200)
async def adicionar_alunos(turma_id: str, alunos: List[AlunoDto]):
    try:
        usecase = CadastrarAlunoEmTurmaUseCase(turma_reposity, usuario_repository)
        usecase.execute(turma_id, alunos)
        return {'response': 'Os alunos foram cadastrados com sucesso na turma'}
    except TurmaNotFoundException as e:
        raise HTTPException(status.HTTP_404_NOT_FOUND, detail=str(e))
```

Na Figura 4.29 mostra um exemplo de endpoint organizado no router turma. O exemplo é realizado usando o endpoint *adicionar_turma*, os demais endpoints funcionam de maneira muito semelhante. No inicio é chamado a classe da APIRouter, que define o prefixo do agrupamento de endpoints e as tag. Após isso, são injetadas as implementações concretas de repositório do MongoDB e Turma e Usuário, passando uma conexão com banco de dados como argumento. O único trabalho o endpoint é criar uma instância do caso de uso passando os argumentos de requisição e devolver uma resposta, se alguma exceção for lançada na execução é devolvido o erro.

4.4 Testes

Figura 4.30: Estrutura de testes



Na implementação da API, a abordagem de Desenvolvimento Orientado a Testes (TDD) foi aplicada, com os testes estruturados em módulos distintos, conforme apresentado na Figura 4.30. Estes módulos incluem: domain, repository, usecases, e http.

Os testes de domínio focam na validação direta dos métodos das entidades, garantindo a conformidade com as *Enterprise Business Rules*. Já os testes de repositório avaliam as implementações concretas de repositório, assegurando a funcionalidade dos métodos de CRUD (*Create, Read, Update, Delete*). Para esta finalidade, foi adotado o pacote `mongomock` para criar uma conexão com o banco de dados MongoDB em memória.

Os testes de casos de uso verificam as implementações dos casos de uso, validando as *Application Business Rules*. Este módulo também utiliza o banco de dados em memória e testa se os resultados são conforme esperado quando os DTOs (*Data Transfer Objects*) são passados como argumentos.

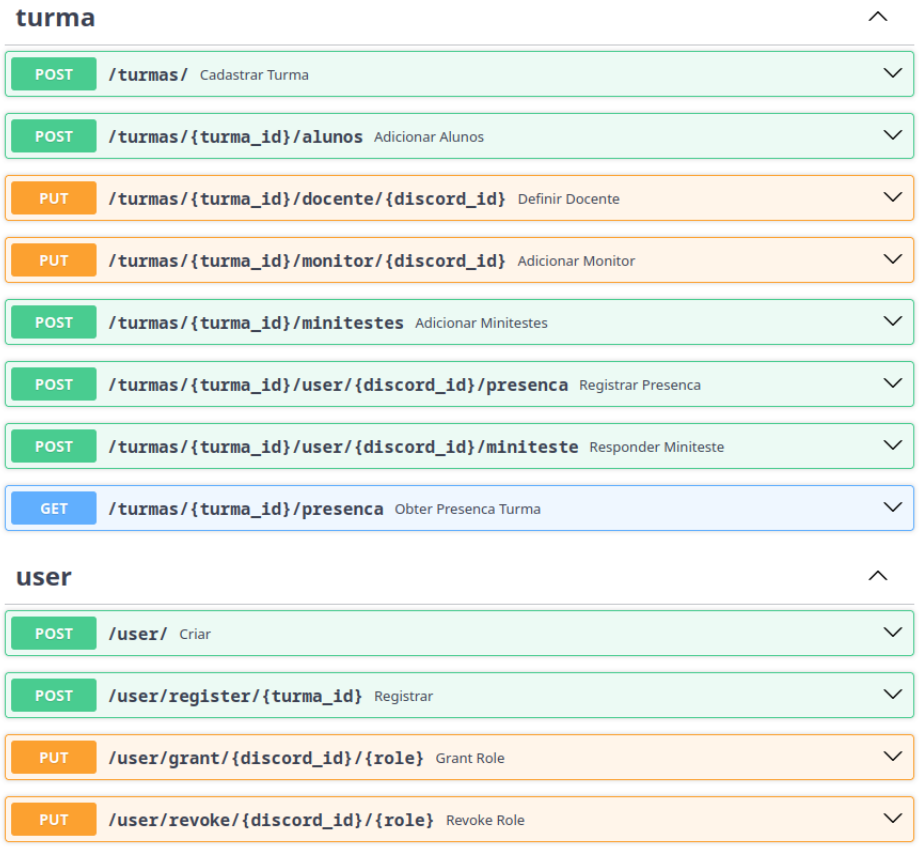
O último módulo compreende os testes HTTP, que se diferenciam dos testes unitários anteriores por se tratarem de testes de integração. Estes simulam a comunicação efetiva com a API através de requisições HTTP aos endpoints, verificando se as respostas estão

de acordo com o esperado.

Vale salientar que os testes unitários foram realizados utilizando o pacote `unittest` e os teste de integração para endpoints do FastAPI foi utilizado o pacote `httpx` e `pytest`

4.5 Documentação

Figura 4.31: Documentação da API



O FastAPI inclui a ferramenta de documentação automática que utiliza o Swagger UI. Essa documentação é gerada automaticamente com base nos metadados das suas funções de manipulação de rotas, aproveitando a tipagem de Python e as anotações de tipo para criar uma documentação precisa e rica em informações. Na Figura 4.31 mostra a página web referente a documentação que está dividida em duas seções: turma e user.

Capítulo 5

Conclusão

O presente trabalho teve como foco a implementação da arquitetura do sistema Bot Educacional com o objetivo de refatorar um código antigo que apresentava diversos problemas sob a óticas das boas práticas de programação e projeto de software, propondo então uma nova arquitetura usando os conceitos de Domain-Driven Design e Arquitetura Limpa, com o objetivo de implementar um domínio rico e consistente, com boa manutenibilidade e testabilidade.

A utilidade do DDD no projeto é justamente o desenvolvimento de um domínio isolado e que contém as regras de negócio da aplicação, enquanto os frameworks e tecnologias, também chamados de detalhes técnicos, podem variar e isso não deve impactar do domínio do sistema. A Arquitetura Limpa engloba muita coisa do DDD, porém é aplicado em um nível mais amplo, propondo um modelo de camadas e detalhando o que deve conter em cada camada, funcionando como uma bússola no desenvolvimento do código.

No decorrer do projeto foi constatado que a aplicação do DDD e da Arquitetura Limpa tem diversos impactos no código. De um lado, a implementação se torna mais expressiva, pois o isolamento do domínio permite se ter uma clareza maior acerca do sistema. Os padrões de projeto corretamente aplicados tornam o desenvolvimento mais intuitivo. Por outro lado as implementações de Arquitetura Limpa levam a um código grande, pelo motivo que é necessário escrever mais código quando o objetivo é separar o domínio do sistema das tecnologias e frameworks que a envolve.

Em última análise, o projeto de implementação utilizando os padrões previamente citados tem o propósito de contribuir com um estudo mais detalhado dos sistema Bot Educacional, ressaltando a importância da correta aplicação dos padrões de projeto para o desenvolvimento sustentável de software, entendendo que problema vem antes da tecnologia e não o contrário.

Referências Bibliográficas

Banker, Kyle, Douglas Garrett, Peter Bakkum & Shaun Verch (2016), *MongoDB in action: covers MongoDB version 3.0*, Simon and Schuster.

Beck, Kent (2022), *Test driven development: By example*, Addison-Wesley Professional.

Evans, Eric (2004), *Domain-driven design: tackling complexity in the heart of software*, Addison-Wesley Professional.

Ferrara, Emilio, Onur Varol, Clayton Davis, Filippo Menczer & Alessandro Flammini (2016), 'The rise of social bots', *Communications of the ACM* **59**(7), 96–104.

Gamma, Erich, Richard Helm, Ralph Johnson & John M. Vlissides (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, 1ª edição, Addison-Wesley Professional.
URL: *http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt$_a$t$_e$p$_d$pi$_1$*

Martin, Robert C (2017), 'Clean architecture'.