Nome: Matheus de Amorim Favero, RA: 23111800 - Ciência da Computação

Aplicações Digitais: Planejamento e Produção – Prof. Robson Fernandes

**Atividade: Relatório sobre as Camadas Arquiteturais do Clean Architecture**

O conceito de Arquitetura de Software foi introduzido previamente como um dos pilares para o sucesso de grandes sistemas com vários contextos de operação. Essa prática se baseia em dividir o software entre camadas, e cada uma sessas partições são responsáveis por suas devidas preocupações. A principal função da arquitetura de software é garantir que o sistema seja modular, escalável, flexível e de fácil manutenção, o que é fundamental para a longevidade e sucesso do projeto.

Diagrama

Descrição gerada automaticamente

*Clean Architecture – Acesso em 21/10/2024.* [*https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html*](https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html)

O *Clean Architecture*, proposto por Robert C. Martin (no livro Clean Architecture: A Craftsman's Guide to Software Structure and Design), é uma abordagem que enfatiza a separação de responsabilidades por meio de camadas bem definidas, com o objetivo de manter a independência do código e a facilidade de manutenção. Ele organiza o software em camadas circulares, onde as regras de negócio ficam no centro e as dependências externas (como bancos de dados, frameworks, interfaces de usuário) ficam nas camadas mais externas.

As camadas do Clean Architecture são:

1. **Entidades (Entities)**: representam os objetos de domínio e as regras de negócio mais fundamentais do sistema. Elas são responsáveis por manter o estado e o comportamento central do negócio, sendo independentes de qualquer tecnologia, banco de dados ou interface de usuário. Essas entidades podem ser objetos simples, como classes ou estruturas de dados;
2. **Casos de Uso (Use Cases)**: representam as regras de negócios da aplicação, ou seja, como o sistema deve se comportar em termos de fluxos de trabalho e funcionalidades. Ou seja, definido o comportamento das entidades
3. **Adaptadores de Interface (Interface Adapters)**: tem o papel de adaptar dados entre as camadas internas (casos de uso e entidades) e a interface de usuário, banco de dados, APIs externas etc. Ela contém adaptadores e controladores responsáveis por converter dados no formato que a interface espera e no formato que a camada de negócios requer.
4. **Frameworks & Drivers**: contém os detalhes externos e tecnologias específicas do sistema, como bancos de dados, frameworks web, bibliotecas de interface de usuário, sistemas de armazenamento e APIs. Essa camada está na borda do círculo do Clean Architecture e, portanto, é a mais volátil, podendo ser substituída ou modificada sem afetar as camadas internas.

A arquitetura é construída de forma que as camadas internas (como Entities e Use Cases) não dependem das camadas externas (Interface Adapters e Frameworks & Drivers). As dependências sempre apontam para o centro, garantindo que a lógica de negócios seja isolada dos detalhes de implementação. Isso permite maior flexibilidade e facilidade de manutenção, pois mudanças externas (como trocar o banco de dados ou atualizar uma biblioteca de UI) não afetam o núcleo do sistema.

## Exemplo Aplicado

Neste exemplo, utilizaremos um sistema de cadastro de usuários para ilustrar como cada camada do Clean Architecture pode ser implementada e como elas interagem entre si. O foco será demonstrar a funcionalidade de cada camada, desde as entidades (regras de negócio essenciais), passando pelos casos de uso (lógica de aplicação), até os adaptadores de interface e a camada de frameworks e drivers. Essa estrutura modular permite que o sistema seja escalável e flexível, proporcionando uma base sólida para o crescimento da aplicação.

**Entidade (User.js)**

A classe User representa a entidade central da aplicação e define a estrutura básica do usuário, incluindo seus atributos como nome, email, senha, CPF, país, estado e cidade. Esse bloco de código encapsula as regras de negócio essenciais que se aplicam diretamente ao objeto "usuário", mantendo essa lógica independente de outras partes do sistema.

Texto

Descrição gerada automaticamente

**Casos de Uso (UserUseCases.js)**

A classe UserUseCases implementa as operações de manipulação de usuários, como criar, listar, atualizar e excluir usuários. Cada método realiza a lógica de negócio específica para cada funcionalidade e interage com o UserRepository para persistir as mudanças no banco de dados. Essa camada atua como intermediária entre a interface (controlador) e o repositório, garantindo que todas as regras de negócio sejam aplicadas antes de qualquer operação com os dados.

Texto

Descrição gerada automaticamente

**Adaptador de Interface (UserController.js)**

A classe **UserController** adapta as requisições HTTP (como GET, POST, DELETE e PUT) e interage com os casos de uso para manipular os dados do usuário. Ela recebe as requisições do cliente, passa as informações para os casos de uso e retorna as respostas HTTP com os resultados ou mensagens de erro. Essa camada garante que os dados enviados e recebidos pela interface (HTTP) estejam formatados corretamente, além de delegar a lógica de manipulação de usuários para a camada de casos de uso.

Texto

Descrição gerada automaticamente

**Repositório (UserRepository.js)**

A classe **UserRepository** é responsável por interagir diretamente com o banco de dados utilizando o Prisma ORM. Ela oferece métodos como findAll, save, delete e update para realizar operações de leitura, criação, exclusão e atualização de dados no banco. Essa camada encapsula a lógica de persistência e permite que as outras camadas (casos de uso e controladores) manipulem dados sem precisar conhecer os detalhes da implementação do banco de dados.

Texto

Descrição gerada automaticamente

## Conclusão

A escolha da arquitetura correta é fundamental para garantir a qualidade, a manutenção e a escalabilidade de um sistema ao longo do tempo. No exemplo apresentado, a implementação de operações CRUD usando o **Prisma ORM** permite que as interações com o banco de dados sejam gerenciadas de forma eficiente, mas, com o crescimento da aplicação e o aumento de sua complexidade, uma arquitetura bem definida se torna essencial para evitar problemas de acoplamento, dificuldade de manutenção e rigidez no sistema.

**Fontes**

**PITALIYA, Sarrah.** *Understanding software architecture: a complete guide.* Medium, 2023. Disponível em: <https://sarrahpitaliya.medium.com/understanding-software-architecture-a-complete-guide-cb8f05900603>. Acesso em: 18 out. 2024.

**PALMA, Rodrigo Otávio.** *ICMC-USP.* GitHub, 2024. Disponível em: <https://github.com/ropalma/ICMC-USP>. Acesso em: 20 out. 2024.

**MARTIN, Robert C.** *The clean architecture.* Blog Clean Coder, 2012. Disponível em: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>. Acesso em: 21 out. 2024.

**MARTIN, Robert C.** *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* 1. ed. [S.l.]: Prentice Hall, 2017.