Kanban

1. Proposta

O presente documento tem como objetivo fornecer uma visão abrangente da arquitetura de software proposta para o projeto acadêmico em questão. Ele descreve a estrutura global do sistema, seus componentes principais, interações e decisões de design que orientarão o desenvolvimento do software.

2. Os objetivos e filosofia

1. **Desempenho e Eficiência:** A arquitetura será otimizada para lidar eficientemente com as demandas de desempenho, garantindo uma resposta rápida mesmo em condições adversas. Será dada atenção especial à minimização de gargalos e à maximização da eficiência dos desenvolvedores;
2. **Manutenção a Longo Prazo:** Considerando a importância da manutenção a longo prazo, a arquitetura será projetada com a modularidade e clareza em mente. A documentação será elaborada de forma abrangente para facilitar a compreensão e a modificação do sistema;
3. **Facilidade de Integração e Extensibilidade:** A arquitetura será flexível e fácil de integrar com novos componentes ou serviços. A extensibilidade será uma prioridade, permitindo a incorporação eficiente de novos recursos e funcionalidades à medida que os requisitos evoluem;
4. **Princípios e boas práticas de código:** A arquitetura será elaborada de acordo com os princípios do livro Clean Code, buscando a clareza e simplicidade no design. Códigos legíveis e compreensíveis serão prioritários, garantindo que a base do software seja acessível e compreendida por todos os membros da equipe de desenvolvimento. Isso não apenas facilitará a manutenção contínua, mas também agilizará o processo de onboarding para novos colaboradores;
5. **Testabilidade:** Será enfatizada a testabilidade do código, com a implementação de testes unitários. A automação dos processos de teste garantirá a detecção precoce de possíveis falhas, proporcionando maior confiabilidade ao sistema. Essa abordagem contribuirá para a construção de um software robusto e estável.

3. Dependências

- **Backend:** Java 11 ou uma versão superior; Spring Framework na versão 5 ou superior – o uso de todo seu ecossistema está liberado, ou seja: ORMs, Sprint Data, Spring Cloud, Hibernate etc; e o sistema gerenciador de banco de dados **deve** ser o **MySQL**;
- **Frontend:** HTML 5, CSS3, JavaScript usando o TypeScript – em conjunto com o framework Angular 2, na versão 11 ou superior;
- **Cloud:** Docker e Docker Compose.

4. Requisitos para a prática da codificação

Para colocar em prática nossa arquitetura é importante ter em mente que nós trabalharemos de uma forma desacoplada e modularizada, isso implica em que teremos uma separação de camadas estilo MVC através do padrão de arquitetura hexagonal – que será melhor explicado nos capítulos seguintes. Entretanto, há regras a serem seguidas para manter a integridade da arquitetura, que são:

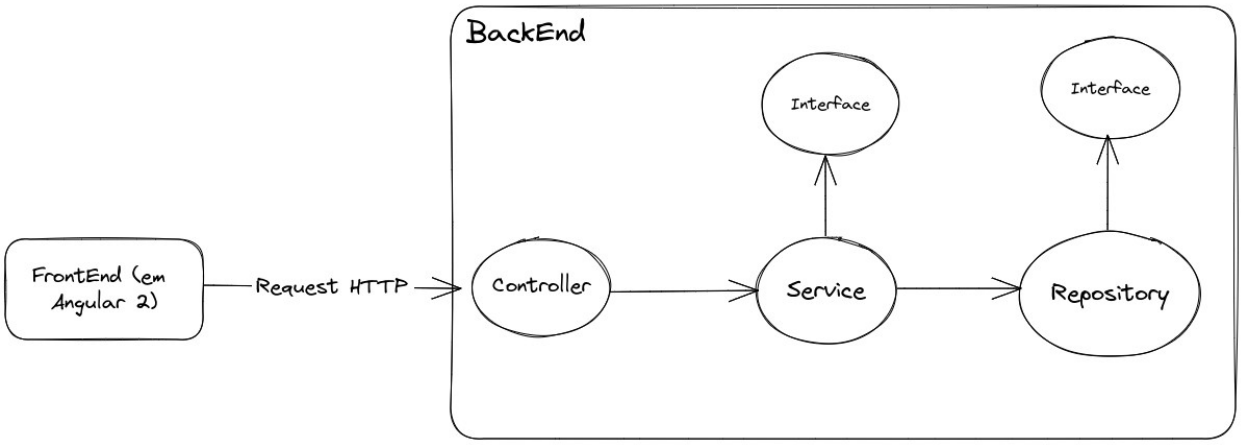
- Os nomes de variáveis, métodos, classes, pacotes etc devem ser significativos;
- Métodos pequenos, com responsabilidades únicas;
- Comente o necessário, nada de textos, seja objetivo;
- Refatorações continuas (para evitar repetições de código – utilize o princípio DRY: Don’t Repeat Yourself);

- Limite a quantidade de parâmetros dos métodos até 3, salvo raras exceções;
- Testabilidade: se tá difiícil para testar, pode ser melhorado;
- Complexidade: O código deve ser feito para todos entenderem, se para você estiver difícil então está ruim!

Caso, no decorrer do desenvolvimento haja alguma situação em que não esteja orientado nesse documento pense sempre no desenvolvedor do futuro. Se você voltar a ler o código que está escrevendo daqui a 1 ano seria de fácil entendimento e compreensão? Se a resposta for não, procure ajuda, pesquise, pergunte!

5. Camadas arquiteturais

O objetivo de uma arquitetura é para manter um código legível, padronizado e organizado, um sistema escalável e com valências para garantir o desacoplamento junto ao conceito de responsabilidades únicas. Portanto, adotamos a prática da arquitetura MVC, ou seja, a separação será em camadas onde cada uma possui sua responsabilidade e função. O workflow das camadas é bem simples: Controller, Service (que vai implementar uma interface com os métodos que possuirá as assinaturas) e Repository, exemplificando:



Controller

Essa é a camada que realizará o recebimento das requests – ou seja: onde será criado os endpoints da nossa API, será realizado a normalização dos dados (ou seja: transformação dos dados para um formato que o sistema entenda), ela também é a camada que deverá retornar o status code, o body e qualquer outra informação da request. Por fim, aqui também deve ser realizado o tratamento de erros que vão retornar da Service e/ou Repository, ou seja, o try/catch deve de um throw deve ser realizado aqui. A camada seguinte deve ser a Service e a bridge/ponte que ligará essas duas camadas deverá ser a interface criada para o service, isso vai garantir o desacoplamento – utilize os patterns de injeção de dependências e de singleton.

Exemplos de projetos que possui essa camada controller:

- https://github.com/Vinicius-92/order-api/blob/main/src/main/java/com/viniciusaugusto/orderapi/controllers/ClientController.java
- https://github.com/vitorvd/PortalSC/blob/master/backend/src/main/java/br/com/santacruz/portal/endpoint/UserEndPoint.java

Service

A camada de serviço encapsula e abstrai a lógica de negócios da aplicação. Isso significa que as regras e processos centrais do sistema são implementados e organizados nessa camada, por exemplo, a regra de negócio do usuário só conseguir deletar boards (quadros) criados por ele mesmo deve ser implementada aqui. A camada de serviço interage com a camada repository (camada de persistência no banco de dados) – a ponte entre o service e a repository também é a interface. Por fim, os testes unitários devem ser escritos em cima dos métodos da camada de serviço, ou seja, a escrita de testes para a controller é dispensável.

Exemplos de projetos que possui essa camada service:

- https://github.com/vitorvd/PortalSC/blob/master/backend/src/main/java/br/com/santacruz/portal/servico/UserServiceImpl.java
- https://github.com/Vinicius-92/order-api/blob/main/src/main/java/com/viniciusaugusto/orderapi/services/impl/ClientServiceImpl.java

Repository

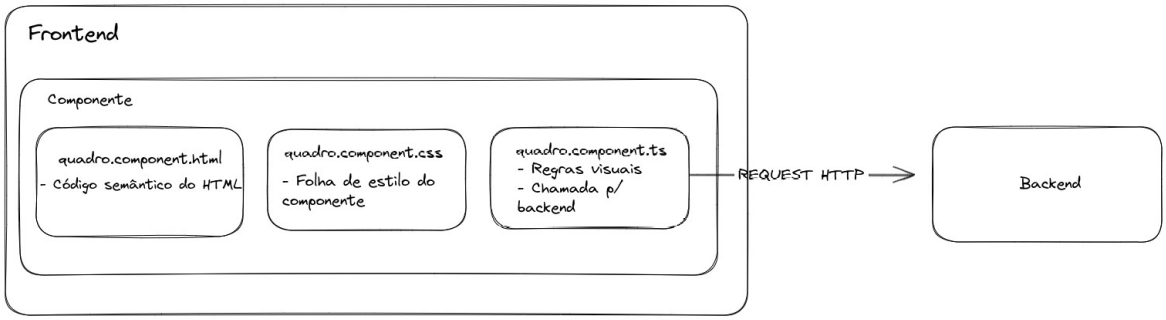
Sua função principal é abstrair as operações de persistência de dados, oferecendo uma interface consistente para as camadas superiores da aplicação, como a camada de service. Essa camada é crucial para garantir que as operações de leitura e gravação no banco de dados sejam eficientes, seguras e alinhadas com as regras de integridade definidas no nível de banco de dados. Além disso, a camada de Repositório desempenha um papel fundamental na ocultação de detalhes específicos do banco de dados, permitindo flexibilidade para mudanças na estrutura de armazenamento sem impactar outras partes do sistema. É dispensável a presença de testes unitários nessa camada. É notório nos exemplos que foi extendido uma abstração de criação, leitura, atualização e delete (CRUD) – que existe no Spring Framework, sinta-se livre para usar.

Exemplos de projetos que possui essa camada repository:

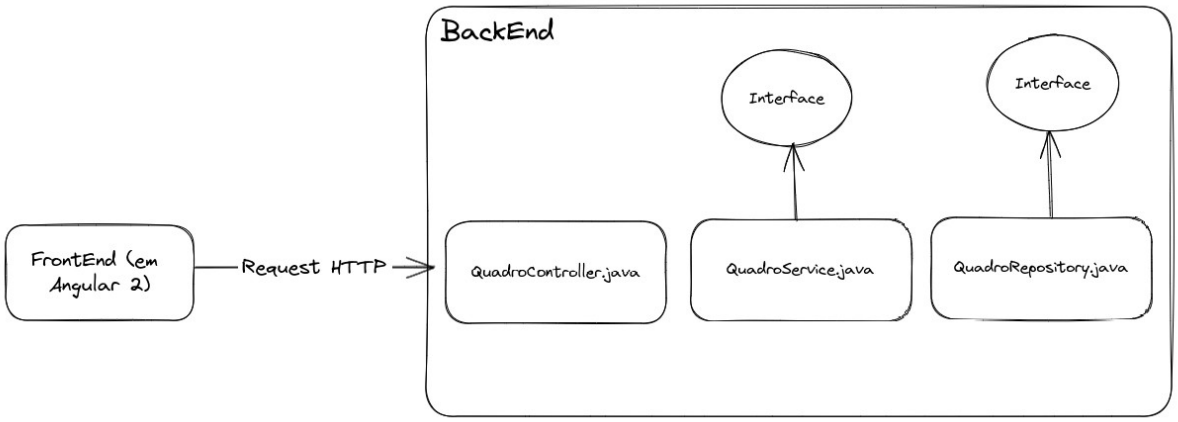
- https://github.com/Vinicius-92/order-api/blob/main/src/main/java/com/viniciusaugusto/orderapi/repositories/ClientRepository.java
- https://github.com/Vinicius-92/person-api/blob/main/src/main/java/com/viniciusaugusto/personapi/repository/PersonRepository.java

6. Architectural views

Um exemplo mais genérico já foi apresentado em 5**. Camadas arquiteturais**, segue abaixo um exemplo de um caso de uso real do sistema, que com toda certeza deverá ser desenvolvido:



Demonstrativo do Frontend



Demonstrativo do Backend