**Decisão Arquitetural para o Sistema de Controle de Caixa**

**1. Introdução**

O desenvolvimento de um sistema de controle de fluxo de caixa para um comerciante exige uma solução que seja confiável, escalável e resiliente. Neste contexto, optamos por uma abordagem baseada em arquitetura **Clean Architecture** combinada com padrões de microsserviços, utilizando **MongoDB e o Polly** para garantir robustez contra falhas. Este documento detalha as decisões arquiteturais e técnicas tomadas para atender aos requisitos funcionais e não funcionais do projeto.

**2. Escolha da Arquitetura**

**Clean Architecture**

A escolha pela **Clean Architecture** foi motivada pela necessidade de modularização, separação de responsabilidades e testabilidade do sistema. Esta abordagem permite:

* Independência das camadas de infraestrutura e framework.
* Facilidade na manutenção e extensão do sistema.
* Reutilização de código e padronização das dependências.

A estrutura do projeto está dividida em quatro camadas principais:

1. **Domain**: Contém as entidades e regras de negócio puras.
2. **Application**: Implementa os casos de uso e intermedia as interações entre domain e infraestrutura.
3. **Infrastructure**: Gerencia acessos a dados
4. **Presentation**: Expõe as APIs para interação com o sistema.

**Uso de Microsserviços**

O sistema foi segmentado em dois serviços principais:

* **Serviço de Transações**: Responsável por processar e armazenar transações de débito e crédito.
* **Serviço de Consolidação**: Responsável por calcular o saldo consolidado diário com tolerância a falhas.

Essa separação permite escalabilidade independente de cada componente e garante que o serviço de transações continue funcionando mesmo que o serviço de consolidação apresente problemas.

**3. Atendimento aos Requisitos de Resiliência e Disponibilidade**

Para garantir que o sistema continue operando mesmo diante de falhas, foram adotadas as seguintes estratégias:

**Uso do Polly para Resiliência**

O **Polly** foi utilizado para implementar **Circuit Breaker** e **Retry Pattern**, permitindo a recuperação de falhas transitórias e prevenindo o overload de serviços falhos.

**Garantia de Operacionalidade Mesmo em Caso de Falha**

Para atender ao requisito não funcional de **garantir que o serviço de controle de transação continue operando mesmo se o serviço de consolidação falhar**, aplicamos as seguintes abordagens:

1. **Banco de Dados NoSQL (MongoDB)**:
   * Utilizamos MongoDB para armazenar as transações devido à sua capacidade de escalar horizontalmente e sua flexibilidade para armazenar diferentes tipos de transações sem necessidade de esquemas rígidos.
2. **Minimal Api**:
   * **Leveza para Microservices**: Para serviços pequenos, independentes e de alto desempenho (como microservices), uma Minimal API é uma excelente escolha, pois oferece uma maneira eficiente e simplificada de construir APIs de baixo custo e fácil escalabilidade.
   * Cenários com menor complexidade: Se sua aplicação não precisa de grandes funcionalidades, como autenticação complexa ou validação extensiva, as Minimal APIs são ideais, pois elas focam no essencial sem sobrecarregar o código

**4. Segurança e Boas Práticas**

Para garantir a segurança do sistema, adotamos as seguintes medidas:

* **Autenticação e Autorizção**: Utilizamos **JWT (JSON Web Token)** para autenticar as requisições.
* **Proteção contra ataques**: Configurações para evitar **SQL Injection**, **Cross-Site Scripting (XSS)** e **Cross-Origin Resource Sharing (CORS)**.

**5. Conclusão**

A abordagem adotada equilibra escalabilidade, resiliência e segurança, garantindo um sistema capaz de operar mesmo em cenários adversos. O uso de **Clean Architecture** facilita a manutenção e expansão do sistema, enquanto **Polly,** proporcionam resiliência e disponibilidade. O resultado é um sistema robusto e pronto para lidar com picos de acessos e falhas sem comprometer sua operação principal.