1. Objetivo

Fornecer uma aplicação para controle de fluxo de caixa diário, permitindo lançamentos de débitos e créditos e a geração de relatórios consolidados de saldo.

1. Funcionalidades

* Cadastro de Lançamentos: Inclusão de receitas (créditos) e despesas (débitos).
* Relatório Diário: Exibição do saldo consolidado por dia.
* Histórico de Movimentações: Permite visualizar lançamentos passados.
* Categorias Financeiras: Organização dos lançamentos por tipo (Ex.: Vendas, Aluguel, Fornecedores).
* Autenticação e Segurança: Login seguro via JWT, garantindo controle de acesso.
* Exportação de Relatórios: Opção para exportar os dados em PDF/Excel.

1. Arquitetura e Tecnologia

* Backend: .NET 8 + Entity Framework + SQL Server
* Frontend: Angular
* Banco de Dados: SQL Server (Docker)
* Autenticação: JWT
* Desdobramento Futuro: API aberta para integrações (ex.: integração com PIX)

1. Serviço de Controle de Lançamentos

• Criar um endpoint para cadastrar lançamentos (crédito/débito).

• Validar os campos obrigatórios (valor, tipo, data).

• Associar os lançamentos a um comerciante (caso tenha multiusuário no futuro).

1. Serviço de Consolidação Diária

• Criar um serviço que soma os lançamentos do dia e calcula o saldo.

• Implementar um endpoint para retornar o saldo diário consolidado.

• Preparar para filtros futuros (ex.: período, categorias).

1. Requisitos técnicos obrigatórios

* Desenho da solução
* Deve ser feito usando C#
* Testes
* Boas praticas são bem vindas (Design Patterns, Padrões de Arquitetura, SOLID e etc)
* Readme cominstruções claras de como a aplicação funciona, e como rodar localmente
* Hospedar em repositório publico (GitHub)
* Todas as documentações de projeto devem estar no repositório

1. Requisitos não funcionais

O serviço de controle de lançamento não deve ficar indisponível se o sistema de consolidado diário cair.

Em dias de picos, o serviço de consolidado diário recebe 50 requisições por segundo, com no máximo 5% de perda de requisições.

* Arquitetura: Podemos usar Clean Architecture para organizar bem as camadas e facilitar testes.
* Resiliência: O serviço de consolidação pode ser assíncrono (ex.: fila com RabbitMQ) para evitar sobrecarga.
* Escalabilidade: O uso de um banco de dados otimizado (indexação, particionamento) ajudará no desempenho.
* Testes: Testes unitários com xUnit e testes de carga para validar os 50 req/s.
* Observabilidade: Log e monitoramento via Serilog + Health Checks na API.

1. Arquitetura Geral

A solução será baseada em Clean Architecture, garantindo separação de responsabilidades.

Os serviços serão desacoplados para evitar impacto caso um deles fique indisponível.

1. Componentes Principais

* Serviço de Controle de Lançamentos
  + Responsável por registrar créditos e débitos.
  + Garante a persistência dos lançamentos no banco de dados.
  + Disponível via API REST.
* Serviço de Consolidação Diária
  + Processa os lançamentos do dia e gera um saldo consolidado.
  + Funciona de forma assíncrona para garantir performance.
  + Deve suportar 50 requisições por segundo.
* Banco de Dados
  + SQL Server com tabelas otimizadas (índices, particionamento).
  + Tabelas principais: Lançamentos, Saldos Consolidados, Comerciantes.
* Mensageria (RabbitMQ ou outro)
  + Separa o processamento dos lançamentos da consolidação.
  + O Serviço de Lançamentos publica eventos para o Serviço de Consolidação processar.
* Monitoramento e Logs
  + Serilog para rastrear logs.
  + Health Checks para verificar disponibilidade dos serviços.
* Autenticação
  + JWT para garantir que apenas usuários autenticados acessem os serviços.

1. Fluxo de Processamento

1. Comerciante faz um lançamento (crédito/débito).
2. O Serviço de Controle de Lançamentos grava no banco e envia evento para o Serviço de Consolidação.
3. O Serviço de Consolidação processa os dados e armazena o saldo consolidado.
4. Comerciante consulta o saldo diário consolidado via API.

1. Arquitetura da solução:

Fluxo estruturado e modular.

[Cliente] → [API Gateway] → [Serviço de Lançamentos] ←→ [Banco de Dados Lançamentos]

↓ (Eventos via RabbitMQ)

[Serviço de Consolidado] ←→ [Banco de Dados Consolidado]

↑

[Serviço de Relatórios]

Explicação do Fluxo

1. Cliente: Interage com o sistema via uma aplicação frontend (ex.: Angular).
2. API Gateway: Controla o tráfego, autenticação (JWT) e roteia requisições.
3. Serviço de Lançamentos:

• Processa débitos e créditos.

• Armazena no Banco de Dados de Lançamentos.

• Publica eventos no RabbitMQ para processamento assíncrono.

1. Serviço de Consolidação:

• Consome eventos do RabbitMQ.

• Calcula o saldo diário e armazena no Banco de Dados Consolidado.

1. Serviço de Relatórios:

• Consulta o Banco de Dados Consolidado.

• Retorna informações para o usuário via API.

Esse modelo garante que o serviço de lançamentos funcione independentemente da consolidação, e que o sistema suporte alta carga sem comprometer o desempenho.

Visão Geral

A solução será desenvolvida seguindo os princípios da Clean Architecture, garantindo separação de preocupações, testabilidade e manutenibilidade. A aplicação será dividida em camadas bem definidas com responsabilidades claras.

Fluxo de Dados

1. Frontend envia requisições para a API

2. API processa e persiste dados no banco

3. Para operações assíncronas, usa-se RabbitMQ

4. Dados são recuperados para relatórios

Diagrama de Componentes

[Frontend Angular] ← HTTP

↑

↓

[API .NET 8]

↑

↓

[SQL Server] ← Entity Framework Core

FluxoDeCaixa/

FluxoDeCaixa.Application # Regras de negócio (caso de uso, DTOs, validações)

FluxoDeCaixa.Domain # Entidades e interfaces

FluxoDeCaixa.Infra # Persistência (EF Core), RabbitMQ e serviços externos

FluxoDeCaixa.API # Controllers, validação de requisições e injeção de dependências

Tests # Testes unitários e de integração

Docs # Documentação arquitetural