API RESTFUL: APLICAÇÃO DE CONTROLE FINANCEIRO PESSOAL

[Cauê Silva Cabral]

[Kauã Gabriel da Silva Antunes]

API RESTFUL: APLICAÇÃO DE CONTROLE FINANCEIRO PESSOAL

Relatório Técnico apresentado como requisito de avaliação da disciplina de Desenvolvimento de Sistemas.

Orientador/Professor: [Daniel Lim-Apo]

SUMÁRIO

1 INTRODUÇÃO
2 ANÁLISE DE REQUISITOS E PLANEJAMENTO
3 ARQUITETURA DE SOFTWARE E MODELAGEM
4 IMPLEMENTAÇÃO DA INFRAESTRUTURA E CÓDIGO
5 EVIDÊNCIAS FUNCIONAIS E VALIDAÇÃO
6 QUALIDADE E TESTES AUTOMATIZADOS
7 CONCLUSÃO

1 INTRODUÇÃO

Este documento técnico visa apresentar e comprovar a implementação da **API RESTful de Controle Financeiro Pessoal**. O projeto demonstrou a aplicação prática de conceitos de arquitetura de software, segurança (JWT), e persistência de dados (Docker/EF Core), conforme as exigências mínimas do plano de ensino.

2 ANÁLISE DE REQUISITOS E PLANEJAMENTO

2.1 Requisitos de Negócio (BRD)

O sistema foi modelado para atender às seguintes regras de negócio (RNs) centrais:

RN001 (Segurança): Toda a gestão de dados é protegida por autenticação JWT (Token).

RN002 (Integridade): Toda transação é obrigatoriamente vinculada a um `UserId` válido (Chave Estrangeira).

RN005 (Lógica): O Saldo Total deve ser calculado pela diferença entre a soma das Receitas e a soma das Despesas.

2.2 Requisitos Funcionais (ERS) e Cronograma

O planejamento seguiu um cronograma estruturado, focado em entregar os requisitos funcionais básicos (CRUD completo e Login) antes das validações de qualidade:

Fase	Descrição
Análise	Levantamento de Requisitos e Diagramas.
Infraestrutura	Configuração do Docker e ORM.
Core	Implementação do Módulo de Autenticação.
Negócio	Implementação do CRUD de Transações e Categorias.
Final	Testes Automatizados, Tratamento de Erros e Documentação.

3 ARQUITETURA DE SOFTWARE E MODELAGEM

O backend adota a **Clean Architecture**, promovendo alta modularidade e organização.

3.1 Camadas da Arquitetura

A estrutura do projeto é dividida nas seguintes camadas:

FinanceiroPessoal.Domain: Modelos de entidades (`User` , `Transaction` , `Category`) e regras de negócio.

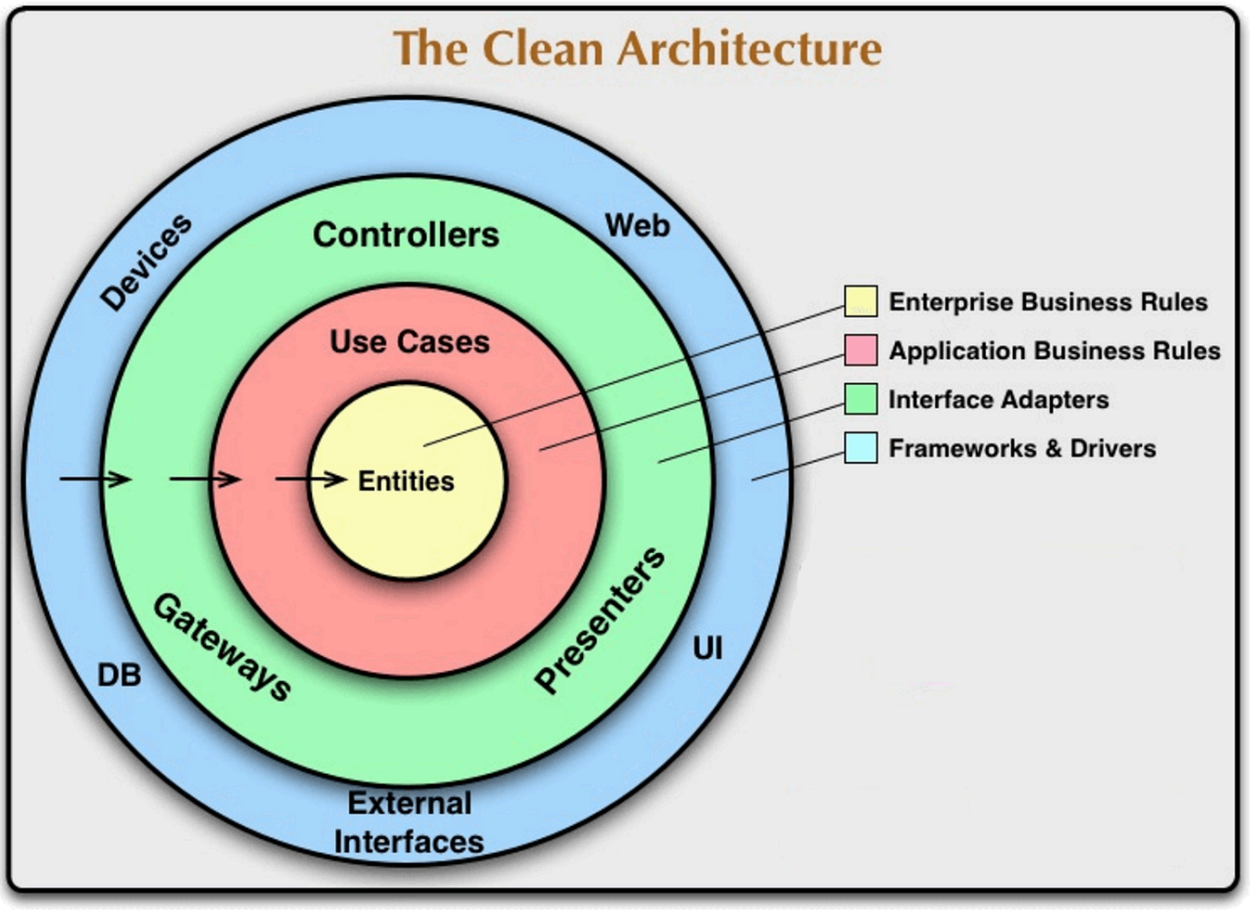
FinanceiroPessoal.Application: Contratos (`ITransactionService`) e modelos de comunicação (DTOs).

FinanceiroPessoal.Infrastructure: Implementação dos serviços e acesso ao banco (EF Core).

FinanceiroPessoal.Api: Camada de entrada, Controllers e injeção de dependência.

Evidência 3.1: Diagrama de Camadas da Clean Architecture

Figura 1: Diagrama de Camadas da Arquitetura Limpa



Cor	Elemento Conceitual	Seu Projeto (Localização)
Amarelo (Centro)	Regras de Negócio Corporativas	`FinanceiroPessoal.Domain`
Rosa/Vermelho	Regras de Negócio da Aplicação	`FinanceiroPessoal.Application`
Verde	Adaptadores de Interface	`FinanceiroPessoal.Infrastructure`
Azul (Borda)	Frameworks & Drivers	`FinanceiroPessoal.Api` e Docker

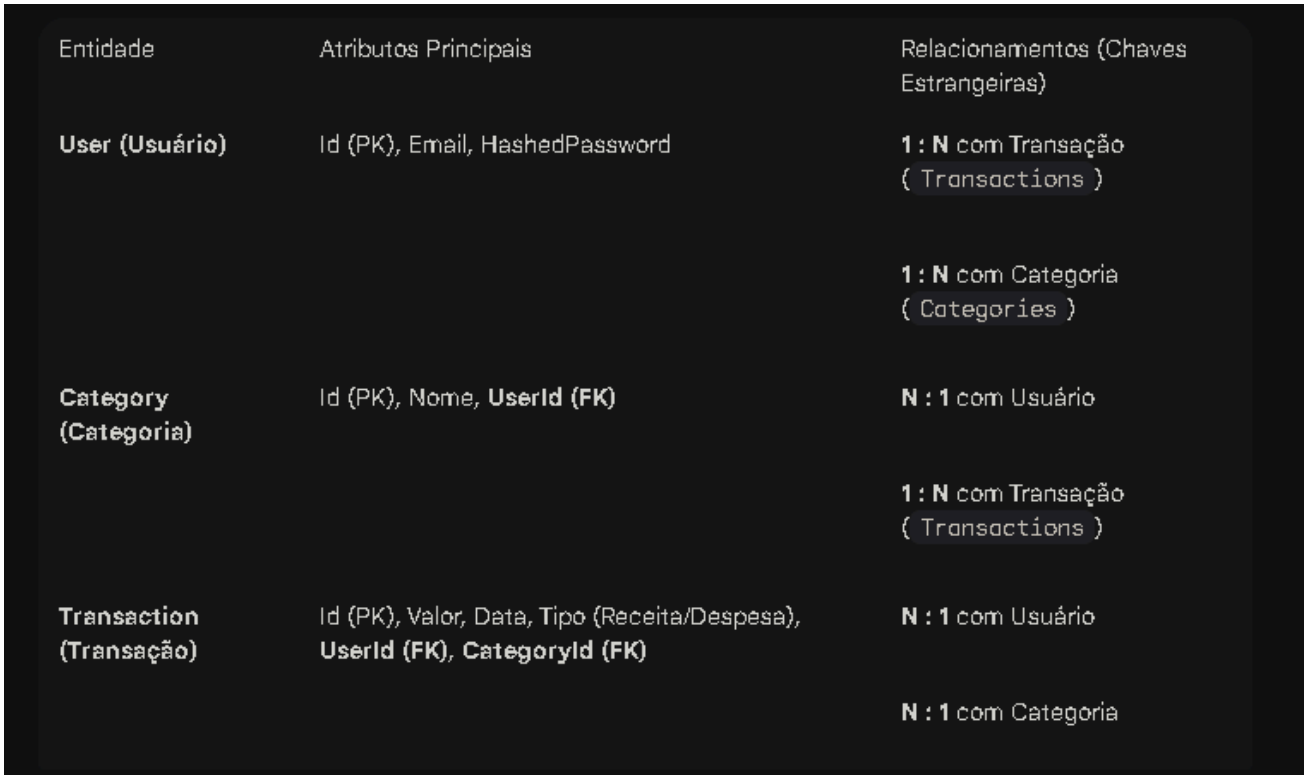
3.2 Diagrama de Entidade-Relacionamento (ER)

O modelo relacional é simples, centrado no usuário e nos seus dados financeiros, estabelecendo as seguintes relações:

- `User` (1) para N `Transaction` (N)
- `User` (1) para N `Category` (N)
- `Category` (1) para N `Transaction` (N)

Evidência 3.2: Diagrama de Classes e Relacionamento (UML/ER)

Figura 2: Diagrama de Classes UML para User, Category, e Transaction



Entidade	Atributos Principais	Relacionamentos (Chaves Estrangeiras)
User (Usuário)	Id (PK), Email, HashedPassword	**1 : N** com Transação (`Transactions`)
		1 : N com Categoria (`Categories`)
Category (Categoria)	Id (PK), Nome, **UserId (FK)**	**N : 1** com Usuário
		1 : N com Transação (`Transactions`)
Transaction (Transação)	Id (PK), Valor, Data, Tipo (Receita/Despesa), **UserId (FK)**, **CategoryId (FK)**	**N : 1** com Usuário
		N : 1 com Categoria

4 IMPLEMENTAÇÃO DA INFRAESTRUTURA E CÓDIGO

4.1 Persistência e Docker

O banco de dados **PostgreSQL** foi configurado para rodar em um container **Docker**, garantindo o isolamento do ambiente. As migrações foram aplicadas com sucesso para criar o esquema de tabelas.

Evidência 4.1.1: Status do Container PostgreSQL

Figura 3: Saída do Terminal mostrando o container Docker ‘POSTGRES_FINANCEIRO’ com status ‘UP’

```
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS          PORTS                                       NAMES
a934057a91e9   postgres:latest   "docker-entrypoint.s…"   About an hour ago   Up About an hour   0.0.0.0:5433->5432/tcp, [::]:5433->5432/tcp   postgres_financeiro
```

CONTAINER ID	IMAGE	COMMAND	STATUS	PORTS	NAMES
a934057a91e9	postgres:latest	"docker-entrypoint.s..."	Up About an hour	0.0.0.0:5433->5432/tcp	postgres_financeiro

4.2 Configuração de Segurança

A API implementa autenticação segura, conforme o requisito **RN001**:

> **Hashing:** Senhas armazenadas com algoritmo BCrypt (na camada Infrastructure).
>
> **Autenticação:** Uso de **JWT** configurado como esquema Bearer no `Program.cs` e no Swagger.

5 EVIDÊNCIAS FUNCIONAIS E VALIDAÇÃO

A validação foi realizada via **Swagger UI**, demonstrando o fluxo completo de uso.

5.1 Validação do Login e Token

O sistema aceita credenciais válidas e retorna o Token JWT, que é a chave para acessar as rotas

protegidas. O código HTTP de sucesso é `200 OK`.

5.2 Validação do CRUD: Criação de Categoria

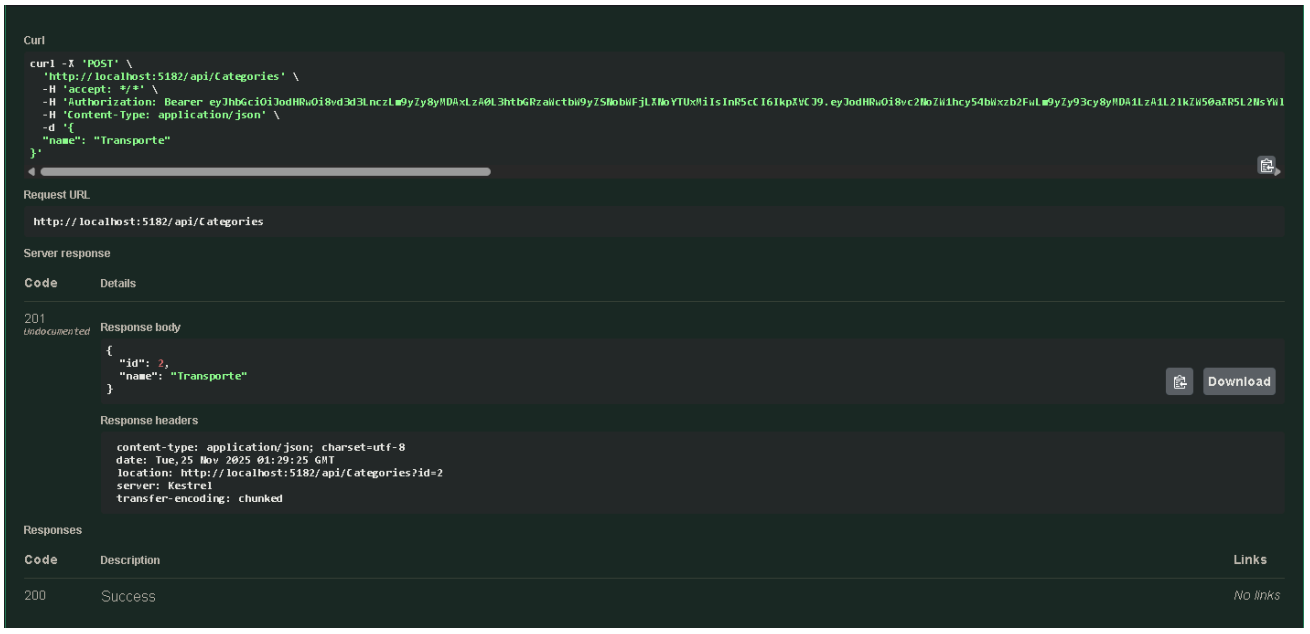
A rota de criação de categoria, que é protegida, foi acessada com sucesso após a autorização via Token.

Evidência 5.2.1: Criação de Categoria (POST)

Status: *201 Created*

Resultado: O banco devolve a entidade recém-criada (Ex: *{"id": 2, "name": "Transporte"}*).

Figura 4: Print da tela do Swagger mostrando o código 201 e a categoria criada



5.3 Validação da Lógica de Negócio (Saldo)

O principal requisito de negócio (**RN005**) foi validado, provando que o sistema realiza o cálculo financeiro corretamente (Receita - Despesa).

Evidência 5.3.1: Cálculo de Saldo (RF005)

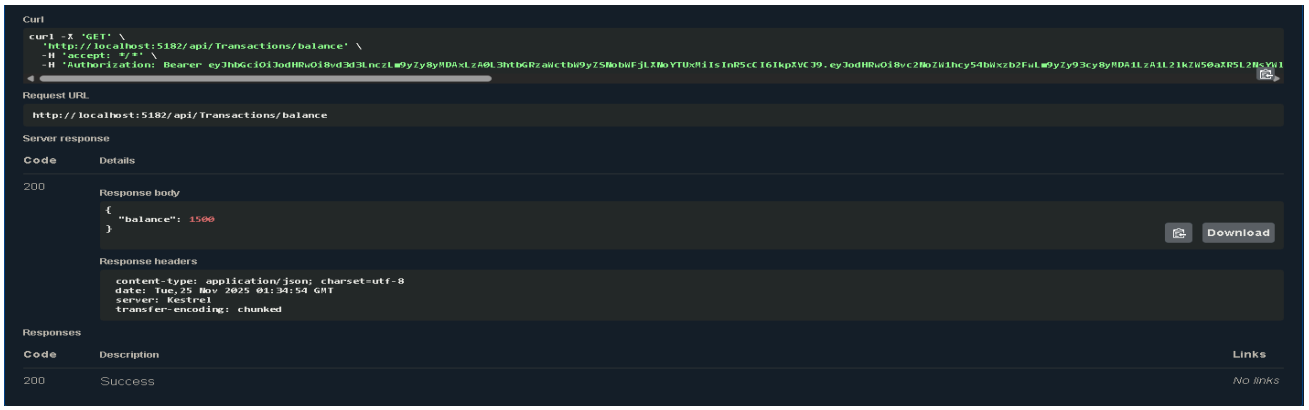
Endpoint: *GET /api/transactions/balance*

Status: *200 OK*

Resultado: Confirmação do valor (Ex: Saldo de R$ 1500,00).

Figura 5: Print da tela do Swagger mostrando o código 200 e o resultado

{"balance": 1500.0}



6 QUALIDADE E TESTES AUTOMATIZADOS

O requisito de qualidade foi atendido com a implementação de testes unitários para a camada de serviço.

6.1 Implementação de Testes (xUnit)

Foram implementados 5 testes automatizados , focados nos fluxos críticos do serviço de autenticação (`AuthService`).

Evidência 6.1.1: Resultado do *dotnet test*

O teste foi executado com sucesso, provando a qualidade da implementação da segurança.

Status: Bem-sucedido: 5

Falhas: 0

```
Resumo do teste: total: 5; falhou: 0; bem-sucedido: 5; ignorado: 0; duração: 2,8s
```

7 CONCLUSÃO

O projeto foi concluído com sucesso, demonstrando a correta aplicação de padrões de arquitetura (Clean Architecture), segurança (JWT), e ferramentas modernas (Docker, EF Core). Todos os requisitos funcionais e de qualidade foram atendidos, resultando em uma API RESTful robusta e funcional.

REFERÊNCIAS

ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS (ABNT). **NBR 6023**: informação e documentação: referências. Rio de Janeiro, 2018.

DOCKER. **Docker Documentation**. Disponível em: https://docs.docker.com/. Acesso em: [20/11/25].

MICROSOFT. **ASP.NET Core Documentation**. Disponível em: https://learn.microsoft.com/en-us/aspnet/core/. Acesso em: [20/11/25].

MICROSOFT. **Entity Framework Core Documentation**. Disponível em: https://learn.microsoft.com/en-us/ef/core/. Acesso em: [20/11/25].

MICROSOFT. **JSON Web Tokens (JWT) in ASP.NET Core**. Disponível em: https://learn.microsoft.com/en-us/aspnet/core/security/authentication/jwt-auth. Acesso em: [20/11/25].

POSTGRESQL GLOBAL DEVELOPMENT GROUP. **PostgreSQL Documentation**. Disponível em: https://www.postgresql.org/docs/. Acesso em: [20/11/25].

XUNIT.NET. **xUnit.net Documentation**. Disponível em: https://xunit.net/docs/. Acesso em: [20/11/25].