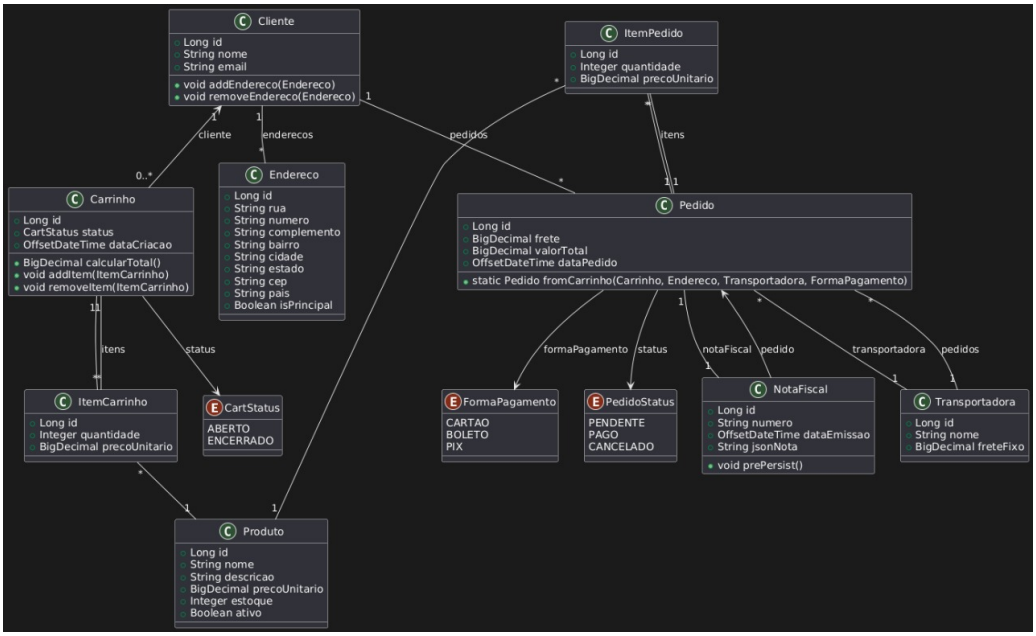
Documentação Técnica

Sistema de Checkout e Faturamento

Diagrama de Classes

Durante o desenvolvimento do sistema, criei um diagrama de classes para representar de forma clara as entidades que compõem nosso domínio. Esse diagrama foi essencial para entender as responsabilidades de cada classe e as relações entre elas.



Descrição das Classes

Ao longo da implementação, foi definida a classe **Cliente** como o ponto de entrada para o usuário do sistema. Ela guarda informações básicas como `id`, `nome` e `email`, e possui métodos que facilitam a adição e remoção de endereços.

A **Endereco** foi pensada para armazenar detalhes de localização, é usada uma flag `isPrincipal` para indicar o endereço padrão.

A classe **Produto** representa os itens do do catálogo, com atributos como nome, descrição, valor e estoque. O campo `ativo` permite ativar ou inativar produtos com facilidade.

O **Carrinho** agrupa os itens escolhidos antes do checkout. Implementado o cálculo de total e controle de itens diretamente nele.

ItemCarrinho representa a relação entre carrinho e produto, reforçando boas práticas de encapsulamento.

Ao finalizar a compra, o sistema cria um **Pedido** via `fromCarrinho`, consolidando todos os dados.

Cada **ItemPedido** representa um item dentro do pedido.

A **NotaFiscal** é gerada automaticamente, com UUID e data. Foram criadas exceções como `ResourceNotFoundException` para regras de negócio.

A **Transportadora** define o frete aplicado ao pedido.

Enumerações

- **CartStatus**: ABERTO, ENCERRADO
- **FormaPagamento**: CARTAO, BOLETO, PIX
- **PedidoStatus**: PENDENTE, PAGO, CANCELADO

Relações Principais

- Cliente possui vários endereços e pedidos.
- Carrinho e Pedido agregam itens.
- Pedido gera uma Nota Fiscal.
- Transportadora aplica o frete.

Arquitetura e Decisões Tomadas

O sistema foi desenvolvido utilizando uma arquitetura em camadas, visando clareza, separação de responsabilidades e manutenibilidade. As camadas foram organizadas da seguinte forma:

- **Controller** — recebe as requisições HTTP e repassa para os serviços.
- **Service** — contém as regras de negócio da aplicação.
- **Model** — representa o modelo de domínio (entidades e enums).
- **Repository** — responsável pela persistência dos dados via Spring Data JPA.

Principais decisões técnicas adotadas:

- Uso de `ModelMapper` para conversão entre entidades e DTOs, com `Assemblers` específicos por recurso.
- Separação clara entre modelos de entrada (`InputModel`) e saída (`Model`) para evitar acoplamento com as entidades.
- Implementação de HATEOAS para enriquecer as respostas da API com links navegáveis.
- Exceções customizadas como `ResourceNotFoundException` e `InvalidQuantityException` para tratamento de regras de negócio.
- Versionamento do banco com `Flyway`, garantindo controle sobre a evolução do schema.
- Utilização de enums para representar estados fixos, como forma de pagamento e status do pedido.
- Ambiente de testes isolado com banco em memória (H2) e perfil dedicado.

As decisões foram baseadas em boas práticas consolidadas no desenvolvimento de APIs REST com Spring Boot e na familiaridade com técnicas aplicadas em sistemas utilizados em produção.

Diagrama de Sequência — Checkout

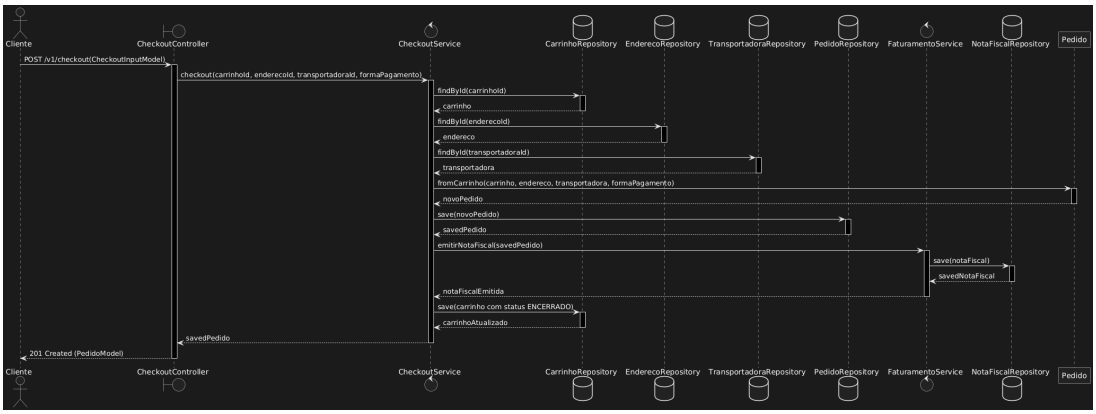
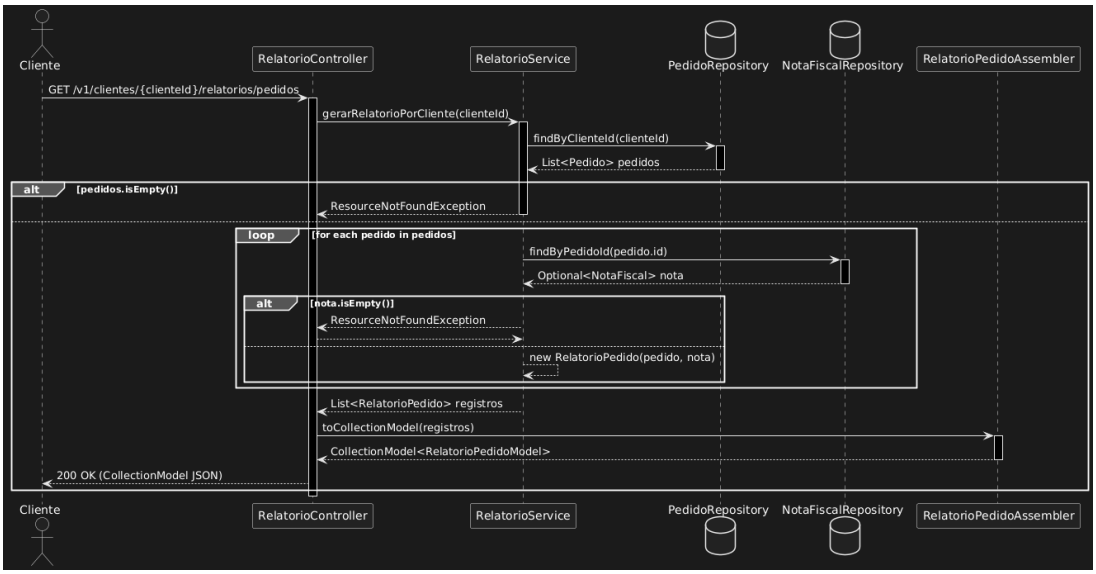


Diagrama de Sequência — Relatório



O JSON completo da resposta pode ser visto no Swagger.

Endpoints Disponíveis

Os principais endpoints da API estão disponíveis nas seguintes rotas:

- **Swagger UI (documentação interativa)**:
  http://localhost:8080/swagger-ui-custom.html
- **Base da API REST (versão 1)**:
  http://localhost:8080/v1

Exemplos de recursos acessíveis:

- `POST /v1/carrinhos?clienteId={id}` — Cria um novo carrinho para o cliente.
- `POST /v1/carrinhos/{id}/itens` — Adiciona um item ao carrinho.
- `POST /v1/checkout` — Finaliza o pedido e gera a nota fiscal.
- `GET /v1/relatorios/pedidos` — Lista os pedidos por cliente com nota fiscal.

Configuração application.properties

Listing 1: application.properties

```
spring.application.name=teste_pulse
server.port=${SERVER_PORT:8080}
spring.datasource.url=${DB_URL:jdbc:postgresql://localhost:5432/teste_pulse}
spring.datasource.username=${POSTGRES_USER:database_user}
spring.datasource.password=${POSTGRES_PASSWORD:database_password}
spring.datasource.driver-class-name=org.postgresql.Driver
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=${JPA_DDL_AUTO:validate}
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.show-sql=${JPA_SHOW_SQL:false}
spring.flyway.enabled=${FLYWAY_ENABLED:true}
spring.flyway.locations=classpath:db/migration
springdoc.api-docs.path=/api-docs
springdoc.swagger-ui.path=/swagger-ui-custom.html
```

Configuração via .env

Listing 2: .env de exemplo

```
SERVER_PORT=8080
DB_URL=jdbc:postgresql://db:5432/teste_pulse
POSTGRES_USER=postgres
POSTGRES_PASSWORD=your_secure_password
POSTGRES_DB=teste_pulse
JPA_DDL_AUTO=update
JPA_SHOW_SQL=true
FLYWAY_ENABLED=true
```

Docker e Healthcheck

Listing 3: docker-compose.yml

```
services:
  db:
    image: postgres:16-alpine
    env_file: .env
    environment:
      - POSTGRES_USER:${POSTGRES_USER}
      - POSTGRES_PASSWORD:${POSTGRES_PASSWORD}
      - POSTGRES_DB:${POSTGRES_DB}
    volumes:
      - db_data:/var/lib/postgresql/data
    ports:
      - "5432:5432"
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U $${POSTGRES_USER}"]
```

```
    interval: 10s
    retries: 5

  api:
    build:
      context: .
      dockerfile: Dockerfile
      args:
        SERVER_PORT: ${SERVER_PORT}
    env_file: .env
    environment:
      - SERVER_PORT=${SERVER_PORT}
      - DB_URL=jdbc:postgresql://db:5432/${POSTGRES_DB}
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - JPA_DDL_AUTO=${JPA_DDL_AUTO}
      - JPA_SHOW_SQL=${JPA_SHOW_SQL}
      - FLYWAY_ENABLED=${FLYWAY_ENABLED}
    ports:
      - "${SERVER_PORT}:${SERVER_PORT}"
    depends_on:
      db:
        condition: service_healthy

volumes:
  db_data:
```

Perfil de Testes

Listing 4: application-test.properties

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=false
spring.flyway.enabled=false
```

Testes Automatizados

Serviços

Utilizado JUnit5 e Mockito para cobrir cenários de sucesso e exceção. Repositórios foram mockados.

Controladores

Usando `@WebMvcTest` e `MockMvc` para simular requisições e validar JSON, status HTTP e tratamento de exceções.

Ambiente de Testes

- Banco em memória H2
- Swagger e Flyway desabilitados
- Perfil `test` com configuração isolada