Transaction Module - Equipe 4 Nullbank

Gabriel Ramos Magalhães
David Souza Santos
Matheus Henrique da Silva
Rinaldo Lira de Albuquerque Lima
Gustavo Henrique de Jesus da Silva

Table of Contents

Visão Geral do Módulo de Transações

O Módulo de Transações é um componente central do nosso sistema bancário, responsável por gerenciar todas as transações bancárias. Esta documentação fornece informações abrangentes sobre a arquitetura, configuração e uso do módulo.

Funcionalidades Principais

- Transferências entre contas
- Depósitos e saques
- Histórico de transações
- Extrato bancário
- Gerenciamento de beneficiários

Stack Tecnológica

- Java 17
- JavaFX (Interface gráfica)
- JPA (Persistência)
- MySQL (Banco de dados)
- Maven (Gerenciamento de dependências)
- Docker (Containerização)

Estrutura do Módulo

O projeto segue os princípios da Clean Architecture com as seguintes camadas:

```
src/
├── main/
│   ├── java/
│   │   └── org/jala/university/
│   │       ├── application/     # Casos de uso e regras de
aplicação
│   │       ├── domain/         # Regras de negócio e entidades
│   │       ├── infrastructure/ # Implementações técnicas
│   │       └── presentation/   # Interface do usuário e
controladores
```

Começando

Consulte o guia Primeiros Passos (Primeiros Passos) para instruções de configuração.

Primeiros Passos

Este guia fornece instruções detalhadas para configurar e executar o Módulo de Transações em seu ambiente de desenvolvimento.

Pré-requisitos

Software Necessário

- Java Development Kit (JDK) 17 ou superior
  - Recomendamos o Amazon Corretto ou OpenJDK
  - Variável JAVA_HOME deve estar configurada
- Docker Desktop (Windows/Mac) ou Docker Engine (Linux)
  - Docker Compose v2.0 ou superior
- Maven 3.8+
- Git 2.x
- IDE (recomendamos uma das opções abaixo):
  - IntelliJ IDEA (recomendado)
  - Eclipse com plugin e(fx)clipse

Requisitos de Hardware

- Mínimo de 8GB de RAM
- 10GB de espaço em disco
- Processador dual-core ou superior

Configuração do Ambiente

1. Preparação do Ambiente

Windows

```
# Verifique as instalações
java --version
docker --version
mvn --version
git --version
```

Linux/MacOS

```
# Instale as dependências (Ubuntu/Debian)
sudo apt update
sudo apt install openjdk-17-jdk maven git docker.io docker-compose

# Adicione seu usuário ao grupo docker
sudo usermod -aG docker $USER
```

2. Clonando o Repositório

```
# Clone o repositório
git clone <url-do-repositório>
cd transaction-module

# Instale as dependências Maven
mvn clean install
```

3. Configuração do Banco de Dados

Usando Docker Compose

```
# Inicie os serviços
docker-compose up -d

# Verifique os logs
docker-compose logs -f mysql
```

Configurações do MySQL

```
Host: localhost
Porta: 3306
Banco: transaction_db
Usuário: transaction_user
Senha: password123
```

Verificação do Banco

```
# Teste a conexão
docker exec -it mysql-container mysql -utransaction_user -
ppassword123 transaction_db
```

4. Configuração da IDE

IntelliJ IDEA

1. Abra o projeto: File → Open → Selecione a pasta do projeto
2. Configure o JDK: File → Project Structure → Project SDK → Add SDK → JDK
3. Importe as dependências Maven: Clique no ícone "Load Maven Changes"
4. Configure o Run Configuration:
   - Main class: `org.jala.university.presentation.MainApplication`
   - JRE: Java 17
   - VM options: `--module-path /path/to/javafx-sdk/lib --add-modules javafx.controls,javafx.fxml`

Eclipse

1. Import → Maven → Existing Maven Projects
2. Configure o Java Build Path
3. Adicione as dependências do JavaFX

5. Executando o Projeto

Via IDE

1. Localize a classe principal:

```
src/main/java/org/jala/university/presentation/controller/MainAppl
ication.java
```

2. Execute como Java Application

Via Maven

```
# Execução padrão
./mvnw clean javafx:run

# Execução com perfil de desenvolvimento
./mvnw clean javafx:run -Pdev

# Execução com debug
./mvnw clean javafx:run -X
```

Estrutura do Projeto

O projeto segue uma arquitetura em camadas. Abaixo está a estrutura principal:

Estrutura de Diretórios

```
src/
├── main/
│   ├── java/
│   │   └── org/jala/university/
│   │       ├── application/     # Casos de uso
│   │       ├── domain/         # Entidades e regras de negócio
│   │       ├── infrastructure/ # Implementações técnicas
│   │       └── presentation/   # Controllers e Views
│   └── resources/
```

```
│       ├── css/                # Arquivos de estilo
│       └── fxml/              # Layouts das telas
```

Solução de Problemas

Problemas Comuns

1. Erro de Conexão com o Banco

```
# Verifique o status do container
docker ps
docker logs mysql-container

# Teste a conexão
telnet localhost 3306
```

2. Erro de Compilação

```
# Limpe o cache Maven
mvn clean

# Atualize as dependências
mvn dependency:purge-local-repository
mvn clean install -U
```

3. Problemas com JavaFX

- Verifique o PATH do JavaFX SDK
- Confirme as dependências no pom.xml
- Teste com diferentes versões do JDK

Logs e Debugging

```
# Ative logs detalhados
mvn clean javafx:run -X
```

Próximos Passos

1. Interface do Usuário (Interface do Usuário)
    - Fluxo de navegação
    - Componentes principais
    - Personalização de temas
2. Arquitetura (Arquitetura)
    - Clean Architecture
    - Padrões de projeto
    - Boas práticas
3. Guia de Desenvolvimento (Guia de Desenvolvimento)
    - Convenções de código
    - Processo de build
    - Testes

Suporte e Recursos

Recursos Adicionais

- JavaFX Documentation (https://openjfx.io/)
- Maven Guide (https://maven.apache.org/guides/)
- Docker Documentation (https://docs.docker.com/)

Os 4P's do Módulo de Transações

Este documento apresenta uma visão geral do módulo de transações organizada segundo o framework dos 4P's: Product, Process, People e Project.

Diagrama Conceitual

Visão Integrada

O módulo de transações é um componente crítico do sistema bancário, desenvolvido com foco em segurança e usabilidade. A integração dos 4P's garante uma abordagem holística:

- **Product**: Define o que será construído, com foco nas necessidades do usuário
- **Process**: Estabelece como o desenvolvimento será conduzido
- **People**: Identifica quem está envolvido e suas responsabilidades
- **Project**: Organiza os recursos e ferramentas necessários

Para mais detalhes sobre cada aspecto, consulte as seções específicas:

- Visão Geral do Produto (Visão Geral do Produto)
- Processo de Desenvolvimento (Processo de Desenvolvimento)
- Equipe e Stakeholders (Equipe e Stakeholders)
- Visão Geral do Projeto (Visão Geral do Projeto)

Visão Geral do Produto

Especificações

Descrição do problema

É necessário ter um módulo específico para gerenciar as transações a fim de garantir segurança, consistência e confiabilidade na hora de realizar transações dentro da rede bancária.

Diagrama de casos de uso

O diagrama de casos de uso do módulo de transações ilustra as interações entre os usuários (clientes) e o sistema, destacando as principais funcionalidades oferecidas. Abaixo estão os casos de uso previstos:

- Realizar transação via QR Code
- Realizar transação via NFC
- Consultar histórico de transações
- Gerenciar beneficiários (adicionar, editar, remover)
- Agendar transferência
- Validar dados do beneficiário
- Realizar transação entre contas
- Autenticar transferência com senha
- Autenticação multifator (MFA) para transações de alto valor

Tecnologias

- Java
- JFX

- Spring Boot
- MySQL
- Docker

Alcance

O módulo em desenvolvimento será uma extensão de um sistema financeiro digital voltado para a realização de transações seguras e práticas por meio de tecnologias modernas como QR Code e NFC. Ele visa oferecer ao cliente uma experiência de transferência simplificada, segura e personalizada, com funcionalidades essenciais de autenticação, gestão e histórico.

Funcionalidades Principais

- **Transações via QR Code e NFC:** O módulo permitirá que os clientes realizem pagamentos e transferências de forma rápida, usando leitura de QR Codes e aproximação por tecnologia NFC.
- **Gestão de Beneficiários:** Os usuários poderão adicionar, editar e remover beneficiários, com validação automática dos dados para maior segurança.
- **Agendamento de Transferências:** O cliente poderá programar transferências para datas futuras, com possibilidade de visualização e cancelamento.
- **Histórico de Transações:** O sistema manterá um histórico completo de todas as transações realizadas, acessível ao cliente a qualquer momento.
- **Segurança Avançada:**
  - Requisição de senha para efetivação de transferências.
  - Autenticação multifator (MFA) para transações de alto valor, como medida de proteção adicional.
  - Validação dos dados do beneficiário antes da conclusão da transação.
- **Operações em Contas:** Os clientes poderão realizar transações entre contas cadastradas, com suporte a diferentes instituições bancárias.

Limitações

- O módulo não cobre funcionalidades de investimento, crédito ou suporte a moedas estrangeiras.
- A autenticação multifator será aplicada apenas em transações acima de um valor pré-configurado (a definir pelo negócio).

Processo de Desenvolvimento

Boas práticas

- **Revisão de código**: Todo código passa por revisão por pares antes de ser integrado
- **Integração contínua**: Uso de pipelines automatizados para garantir qualidade
- **Modelagem de branches**: Seguindo o GitFlow para organização do repositório
- **Testes unitários**: Cobertura mínima de 80% do código
- **Documentação**: Manutenção de documentação técnica e de usuário atualizada

Metodologia

O projeto segue a metodologia Ágil, com os seguintes princípios:

- **Colaboração**: Trabalho em equipe e comunicação constante
- **Transparência**: Visibilidade do progresso e impedimentos
- **Retroalimentação**: Ciclos curtos de feedback para melhorias contínuas
- **Melhoria contínua**: Retrospectivas regulares para aprimoramento do processo
- **Adaptabilidade**: Capacidade de ajustar prioridades conforme necessário

Equipe e Stakeholders

Stakeholders

- **Professor:** José Paulo Rodrigues

PO / Arquitetura / Liderança

- **Tutor:** Karem Huacota Saavedra

Equipe de Desenvolvimento (Grupo 4)

- Rinaldo Lira
- Gustavo Jesus
- Matheus Henrique
- David Souza
- Gabriel Ramos

Responsabilidades

Product Owner

- Definição de requisitos
- Priorização do backlog
- Validação de entregas

Arquiteto

- Definição da estrutura técnica

- Garantia de padrões de qualidade
- Suporte técnico à equipe

Desenvolvedores

- Implementação de funcionalidades
- Testes unitários
- Revisão de código
- Documentação técnica

Visão Geral do Projeto

Ferramentas

Ambiente de Desenvolvimento

- **IDE**: IntelliJ IDEA / Eclipse
- **Controle de Versão**: Git
- **Repositório**: GitLab
- **Documentação**: Writerside

Infraestrutura

- **Containerização**: Docker
- **Banco de Dados**: MySQL
- **CI/CD**: GitLab CI

Organização do Projeto

Estrutura de Diretórios

```
src/
├── main/
│   ├── java/
│   │   └── org/jala/university/
│   │       ├── application/     # Casos de uso
│   │       ├── domain/         # Entidades e regras
│   │       ├── infrastructure/ # Implementações
│   │       └── presentation/   # UI e controllers
│   └── resources/
│       ├── META-INF/
│       │   └── persistence.xml
```

```
│       └── fxml/                # Layouts JavaFX
└── test/
    └── java/                   # Testes unitários
```

Cronograma

O projeto segue um cronograma ágil com sprints de duas semanas, incluindo:

- Sprint Planning
- Daily Standups
- Sprint Review
- Sprint Retrospective

Métricas de Acompanhamento

- Velocidade da equipe
- Cobertura de testes
- Bugs por funcionalidade
- Tempo médio de resolução de problemas

Arquitetura

Esta seção documenta as arquiteturas de software utilizadas no Módulo de Transações, explicando como foram implementadas e quais benefícios trazem ao projeto.

Clean Architecture

Visão Geral

O Módulo de Transações implementa os princípios da Clean Architecture, uma abordagem proposta por Robert C. Martin (Uncle Bob) que organiza o código em camadas concêntricas com regras de dependência claras.

Implementação no Projeto

A estrutura do projeto reflete as camadas da Clean Architecture:

```
src/
├── main/
│   ├── java/
│   │   └── org/jala/university/
│   │       ├── domain/         # Entidades e regras de negócio
│   │       ├── application/    # Casos de uso e regras de
aplicação
│   │       ├── infrastructure/ # Implementações técnicas
│   │       └── presentation/   # Interface do usuário
```

1. **Domain Layer (Camada de Domínio)**
   - Contém as entidades de negócio e regras de domínio
   - Independente de frameworks e tecnologias
   - Exemplo: `Account.java`, `Transaction.java`
2. **Application Layer (Camada de Aplicação)**
   - Implementa casos de uso específicos
   - Orquestra o fluxo entre entidades e infraestrutura

- Exemplo: `TransactionService.java`, `UserService.java`

3. **Infrastructure Layer (Camada de Infraestrutura)**
   - Implementa interfaces definidas nas camadas internas
   - Contém adaptadores para frameworks e tecnologias
   - Exemplo: `UserRepositoryImpl.java`, `JPAConfig.java`
4. **Presentation Layer (Camada de Apresentação)**
   - Interface com o usuário
   - Controladores e views
   - Exemplo: `LoginController.java`, `login-view.fxml`

Benefícios no Projeto

1. **Testabilidade**
   - Camadas internas podem ser testadas independentemente
   - Mocks podem substituir implementações externas
2. **Independência de Frameworks**
   - Regras de negócio não dependem de frameworks
   - Facilita a troca de tecnologias (ex: mudar de JPA para outro ORM)
3. **Manutenibilidade**
   - Separação clara de responsabilidades
   - Código mais organizado e previsível

MVC (Model-View-Controller)

Visão Geral

O padrão MVC separa a aplicação em três componentes principais: Model (dados), View (interface) e Controller (lógica de controle). No contexto do JavaFX, este padrão é

adaptado para a estrutura específica do framework.

Implementação no Projeto

1. **Model**
   - Entidades de domínio e DTOs
   - Exemplo: `User.java`, `TransactionDTO.java`
2. **View**
   - Arquivos FXML que definem a interface
   - Exemplo: `login-view.fxml`, `transfer-view.fxml`
3. **Controller**
   - Classes que controlam a interação entre Model e View
   - Exemplo: `LoginController.java`, `TransferController.java`

Benefícios no Projeto

1. **Separação de Responsabilidades**
   - Interface separada da lógica de negócio
   - Facilita alterações na UI sem afetar a lógica
2. **Reutilização de Código**
   - Views podem ser reutilizadas com diferentes controllers
   - Models podem ser usados em diferentes contextos
3. **Desenvolvimento Paralelo**
   - Equipes podem trabalhar simultaneamente em diferentes componentes

Arquitetura Monolítica

Visão Geral

Uma arquitetura monolítica encapsula todos os componentes da aplicação em um único processo ou artefato de implantação. Embora contraste com arquiteturas distribuídas como microserviços, pode ser apropriada para aplicações de escopo definido.

Implementação no Projeto

O Módulo de Transações é implementado como uma aplicação monolítica:

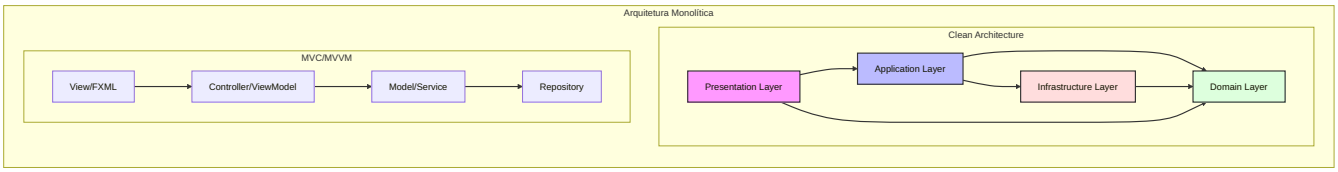
1. **Empacotamento Único**
   - Toda a aplicação é empacotada em um único JAR
   - Implantação como uma unidade coesa
2. **Banco de Dados Compartilhado**
   - Todas as funcionalidades acessam o mesmo banco de dados
   - Transações ACID garantidas
3. **Processo Único**
   - Execução em um único processo JVM
   - Comunicação interna via chamadas de método diretas

Benefícios no Projeto

1. **Simplicidade**
   - Desenvolvimento e implantação simplificados
   - Menor complexidade operacional
2. **Performance**
   - Comunicação eficiente entre componentes
   - Menor overhead de rede
3. **Consistência de Dados**
   - Transações atômicas entre diferentes partes do sistema

- Integridade referencial garantida

Diagrama de Arquitetura



Considerações de Design

1. **Por que Clean Architecture?**
   - Promove a separação de responsabilidades
   - Facilita testes unitários
   - Protege regras de negócio de mudanças em frameworks
2. **Por que MVC?**
   - Adequado para aplicações desktop com JavaFX
   - Facilita a manutenção da interface do usuário
   - Permite binding de dados eficiente
3. **Por que Monolítico?**
   - Apropriado para o escopo atual do projeto
   - Simplifica o desenvolvimento e implantação
   - Reduz a complexidade operacional

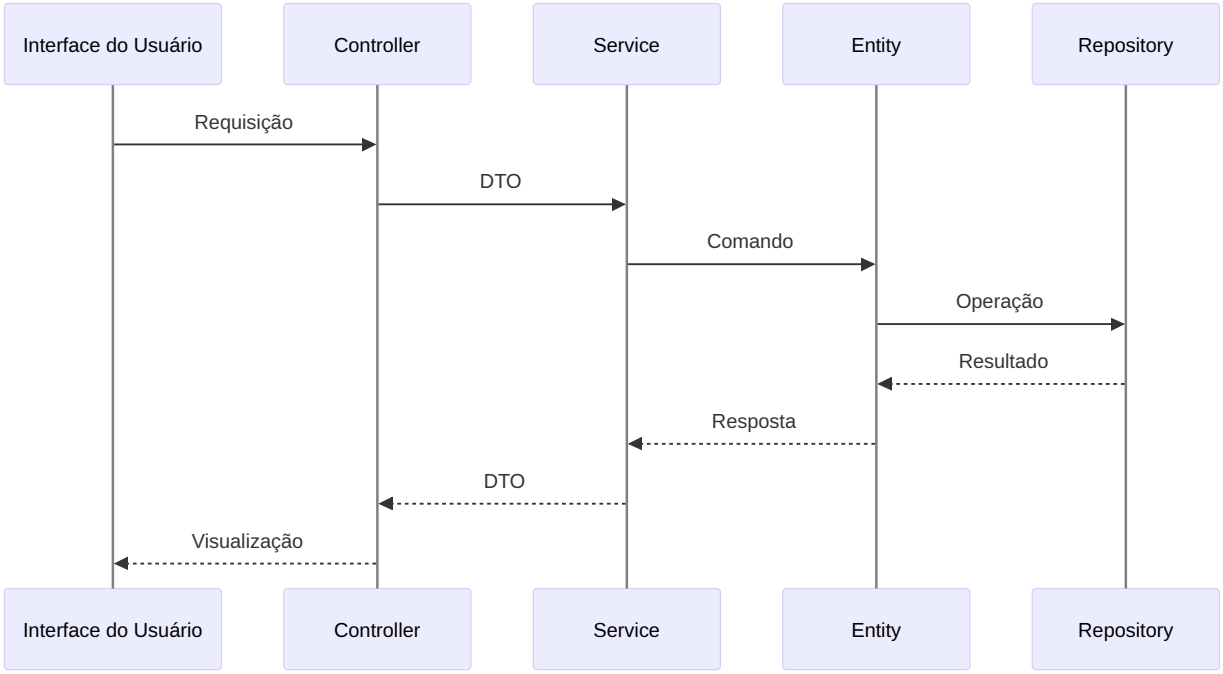
Princípios Arquiteturais

1. **Dependency Rule**
   - Dependências apontam para dentro

- Camadas internas não conhecem externas
- Inversão de dependência quando necessário

2. **SOLID**
   - Single Responsibility Principle
   - Open/Closed Principle
   - Liskov Substitution Principle
   - Interface Segregation Principle
   - Dependency Inversion Principle
3. **Clean Code**
   - Nomes significativos
   - Funções pequenas e focadas
   - Comentários apenas quando necessário
   - Formatação consistente

Fluxo de Dados



Para mais detalhes sobre cada camada, consulte:

- Domain Layer (Camada de Domínio)
- Application Layer (Application Layer)
- Infrastructure Layer (Camada de Infraestrutura)
- Presentation Layer (Camada de Apresentação)

Padrões de Projeto Utilizados

O Módulo de Transações implementa diversos padrões de design para melhorar a qualidade do código e facilitar a manutenção. Para uma visão detalhada, consulte a seção de Padrões de Design (Padrões de Design).

Alguns dos principais padrões utilizados incluem:

1. **Repository Pattern**
   - Abstrai acesso a dados
   - Permite troca de implementação
2. **Factory Pattern**
   - Criação de objetos complexos
   - Centraliza lógica de instanciação
3. **MVVM Pattern**
   - Separa lógica de apresentação
   - Facilita testes unitários
4. **Adapter Pattern**
   - Converte interfaces incompatíveis
   - Facilita integração com sistemas externos
5. **Facade Pattern**

- Simplifica subsistemas complexos
- Fornece interface unificada

Decisões Arquiteturais

Banco de Dados

- MySQL para persistência
- Migrations para controle de schema
- Connection pool para performance

Interface do Usuário

- JavaFX para GUI
- FXML para layouts
- CSS para estilização

Segurança

- Autenticação baseada em tokens
- Criptografia de senhas
- Validação de inputs

Considerações de Performance

1. **Otimizações**
    - Cache em memória
    - Queries otimizadas
    - Paginação de resultados

2. **Monitoramento**
    - Logs estruturados
    - Métricas de performance
    - Rastreamento de erros

Evolução e Manutenção

1. **Versionamento**
    - Controle de versão com Git
    - Branches por feature
    - Semantic versioning
2. **Testes**
    - Testes unitários
    - Testes de integração
    - Testes end-to-end
3. **Documentação**
    - Javadoc para APIs
    - README atualizado
    - Diagramas atualizados

Camada de Domínio

A camada de domínio é o núcleo da aplicação, contendo as regras de negócio e entidades principais. Esta camada é independente de frameworks e tecnologias externas.

Estrutura

```
domain/
├── entities/       # Entidades de domínio
├── repositories/   # Interfaces de repositórios
├── services/       # Serviços de domínio
└── exceptions/     # Exceções de domínio
```

Princípios

1. **Independência**
   - Sem dependências externas
   - Regras de negócio puras
   - Interfaces bem definidas
2. **Imutabilidade**
   - Objetos imutáveis quando possível
   - Estado consistente
   - Thread-safety
3. **Validações**
   - Invariantes de domínio
   - Regras de negócio
   - Estado válido

Value Objects

```
public record Money(BigDecimal amount, Currency currency) {
    public Money {
        Objects.requireNonNull(amount, "Amount cannot be null");
        Objects.requireNonNull(currency, "Currency cannot be
null");
        if (amount.scale() > 2) {
            throw new IllegalArgumentException("Amount cannot have
more than 2 decimal places");
        }
    }

    public Money add(Money other) {
        if (!this.currency.equals(other.currency)) {
            throw new IllegalArgumentException("Cannot add
different currencies");
        }
        return new Money(this.amount.add(other.amount),
this.currency);
    }
}
```

Entidades

Entidade Base

```
@Data
@MappedSuperclass
@NoArgsConstructor
@AllArgsConstructor
public abstract class BaseEntity<ID> {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private ID id;

    @Column(updatable = false)
    @CreationTimestamp
    private LocalDateTime createdAt;

    @UpdateTimestamp
    private LocalDateTime updatedAt;
}
```

Entidades Principais

User

```
@Data
@Entity
@Table(name = "users")
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class User extends BaseEntity<Long> {
    @Column(nullable = false, length = 150)
    private String fullName;
```

```
    @Column(nullable = false, length = 100, unique = true)
    private String email;

    @Column(nullable = false, length = 250)
    private String password;

    @Column(nullable = false, length = 13, unique = true)
    private String cpf;

    @Column(nullable = false, length = 15)
    private String phoneNumber;

    @Enumerated(EnumType.STRING)
    private Role roles;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private Set<Account> accounts;
}
```

Transaction

```
@Data
@Entity
@Table(name = "transactions")
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class Transaction extends BaseEntity<Long> {
    @ManyToOne
    @JoinColumn(name = "source_account_id", nullable = false)
    private Account source;

    @ManyToOne
    @JoinColumn(name = "destination_account_id", nullable = false)
    private Account destination;

    @Column(nullable = false, precision = 10, scale = 2)
```

```
    private BigDecimal amount;

    @Enumerated(EnumType.STRING)
    @Column(nullable = false)
    private TransactionStatus status;

    @Column(nullable = false)
    private LocalDateTime scheduledFor;

    @PrePersist
    @PreUpdate
    private void validateTransaction() {
        if (source.equals(destination)) {
            throw new InvalidTransactionException("Source and
destination accounts cannot be the same");
        }
        if (amount.compareTo(BigDecimal.ZERO) <= 0) {
            throw new InvalidTransactionException("Amount must be
positive");
        }
    }
}
```

Anotações Lombok

- `@Data`: Gera getters, setters, equals, hashCode e toString
- `@Builder`: Implementa o padrão Builder
- `@NoArgsConstructor`: Gera construtor sem argumentos
- `@AllArgsConstructor`: Gera construtor com todos os argumentos

Anotações JPA

- `@Entity`: Marca a classe como entidade JPA

- `@Table`: Define o nome da tabela
- `@Column`: Configura propriedades da coluna
- `@ManyToOne`: Define relacionamento muitos-para-um
- `@OneToMany`: Define relacionamento um-para-muitos
- `@Enumerated`: Configura persistência de enums

Repositórios

Interfaces Base

```
public interface Repository<T extends BaseEntity<ID>, ID> {
    Optional<T> findById(ID id);
    T save(T entity);
    void delete(T entity);
    boolean exists(ID id);
}
```

Repositórios de Domínio

UserRepository

```
public interface UserRepository extends Repository<User, UUID> {
    Optional<User> findByEmail(Email email);
    boolean existsByEmail(Email email);
    List<User> findByRole(Role role);
}
```

TransactionRepository

```
public interface TransactionRepository extends
Repository<Transaction, UUID> {
    List<Transaction> findByAccount(Account account);
    List<Transaction> findByStatus(TransactionStatus status);
    List<Transaction> findScheduledBetween(LocalDateTime start,
LocalDateTime end);
}
```

Boas Práticas

1. **Interfaces Claras**

- Métodos bem nomeados
- Parâmetros tipados
- Documentação quando necessário

2. **Consultas Específicas**
   - Métodos para casos de uso
   - Critérios de busca
   - Ordenação quando relevante
3. **Performance**
   - Paginação quando necessário
   - Consultas otimizadas
   - Carregamento sob demanda

Serviços de Domínio

Serviços Base

```
public abstract class DomainService {
    protected final EventPublisher eventPublisher;

    protected DomainService(EventPublisher eventPublisher) {
        this.eventPublisher =
Objects.requireNonNull(eventPublisher);
    }

    protected void publishEvent(DomainEvent event) {
        eventPublisher.publish(event);
    }
}
```

Implementações

TransactionService

```
public class TransactionService extends DomainService {
    private final TransactionRepository transactionRepository;
    private final AccountService accountService;

    public void executeTransaction(Transaction transaction) {
        validateTransaction(transaction);

        try {
            accountService.debit(transaction.source(),
transaction.amount());
            accountService.credit(transaction.destination(),
transaction.amount());

            transaction.complete();
```

```
            transactionRepository.save(transaction);

            publishEvent(new
TransactionCompletedEvent(transaction));
        } catch (Exception e) {
            transaction.fail();
            transactionRepository.save(transaction);
            publishEvent(new TransactionFailedEvent(transaction,
e));
            throw new TransactionExecutionException("Failed to
execute transaction", e);
        }
    }

    private void validateTransaction(Transaction transaction) {
        if (transaction.status() != TransactionStatus.PENDING) {
            throw new InvalidTransactionStateException(
                "Transaction must be in PENDING state");
        }

        if
(transaction.scheduledFor().isAfter(LocalDateTime.now())) {
            throw new InvalidTransactionStateException(
                "Transaction is scheduled for future execution");
        }
    }
}
```

Boas Práticas

1. **Regras de Negócio**
   - Validações de domínio
   - Invariantes preservados
   - Transações atômicas

2. **Eventos de Domínio**
    - Notificação de mudanças
    - Auditoria
    - Integração desacoplada
3. **Tratamento de Erros**
    - Exceções específicas
    - Estados consistentes
    - Rollback quando necessário

Application Layer

A camada de aplicação é responsável por orquestrar o fluxo de dados e implementar os casos de uso do sistema, atuando como intermediária entre a interface do usuário e o domínio.

Estrutura da Camada

DTOs (Data Transfer Objects)

- Objetos para transferência de dados entre camadas
- Previne exposição das entidades de domínio
- Otimiza a transferência de dados
- Define contratos de entrada e saída

Mappers

- Conversão entre DTOs e entidades de domínio
- Mapeamento bidirecional de objetos
- Isolamento das transformações de dados
- Prevenção de acoplamento entre camadas

Services

- Implementação dos serviços de aplicação
- Orquestração de regras de negócio
- Gerenciamento de transações
- Integração com serviços externos

Responsabilidades

1. **Orquestração**
    - Coordenar fluxo entre camadas
    - Gerenciar transações
    - Controlar acesso a recursos
2. **Transformação de Dados**
    - Converter DTOs em entidades
    - Preparar dados para apresentação
    - Validar entrada de dados
3. **Segurança**
    - Autenticação de usuários
    - Autorização de operações
    - Validação de permissões
4. **Integração**
    - Comunicação entre camadas
    - Gerenciamento de dependências
    - Tratamento de erros

Padrões Utilizados

1. **DTO Pattern**
    - Transferência de dados
    - Encapsulamento

- Versionamento de APIs

2. **Mapper Pattern**
    - Conversão de objetos
    - Isolamento de transformações
    - Manutenção da coesão
3. **Service Layer Pattern**
    - Orquestração de operações
    - Abstração de complexidade
    - Reutilização de lógica

Boas Práticas

1. **Validação**
    - Validar dados de entrada
    - Verificar regras de negócio
    - Tratar exceções apropriadamente
2. **Performance**
    - Otimizar transformações
    - Minimizar overhead
    - Gerenciar recursos
3. **Manutenibilidade**
    - Código limpo e organizado
    - Documentação clara

- Testes unitários

Data Transfer Objects (DTOs)

DTOs são objetos simples usados para transferir dados entre as camadas da aplicação, especialmente nas operações de entrada e saída do sistema.

Tipos Principais

Request DTOs

Usados para receber dados de entrada:

```
CreateTransactionRequestDTO
├── amount: BigDecimal
├── description: String
└── accountId: UUID
```

Response DTOs

Usados para retornar dados processados:

```
TransactionResponseDTO
├── id: UUID
├── amount: BigDecimal
├── status: TransactionStatus
└── createdAt: LocalDateTime
```

Boas Práticas

1. **Estrutura**
   - Manter campos simples
   - Usar tipos apropriados
   - Incluir apenas dados necessários
2. **Nomenclatura**
   - Sufixo `DTO`

- Nomes descritivos
- Padrão consistente

3. **Validação**
   - Campos obrigatórios
   - Formatos válidos
   - Regras básicas

Benefícios

- Separa dados de regras de negócio
- Protege entidades do domínio
- Simplifica transferência de dados
- Facilita manutenção da API

Mappers

Mappers são responsáveis pela conversão entre DTOs e entidades do domínio, mantendo a separação entre as camadas da aplicação.

Interface Base

```
public interface Mapper<T, I> {
    I mapTo(T input);
    T mapFrom(I input);
}
```

Implementações

UserMapper

Converte entre `User` e `UserDTO`:

```
UserMapper
├── mapTo: User -> UserDTO
└── mapFrom: UserDTO -> User
```

Responsabilidades

1. **Conversão**
   - Transformação entre objetos
   - Mapeamento bidirecional
   - Tratamento de tipos
2. **Isolamento**
   - Centraliza lógica de conversão
   - Evita duplicação de código

- Facilita manutenção

Boas Práticas

- Implementar interface `Mapper`
- Manter mapeamentos simples
- Tratar casos nulos
- Validar dados convertidos

Services

Services implementam a lógica de negócio da aplicação, orquestrando operações entre diferentes componentes do sistema.

Tipos de Serviços

AuthenticationService

Gerencia autenticação e autorização:

```
AuthenticationService
├── signup(SignupDTO)
├── login(LoginDTO)
└── validateToken(String)
```

UserService

Gerencia operações de usuários:

```
UserService
├── create(UserDTO)
├── update(UserDTO)
├── delete(Long)
└── findById(Long)
```

Responsabilidades

1. **Regras de Negócio**
   - Validações complexas
   - Orquestração de operações
   - Controle transacional
2. **Integração**
   - Comunicação com repositórios

- Uso de mappers
- Chamadas externas

Boas Práticas

1. **Estrutura**
   - Interface definindo contrato
   - Implementação separada
   - Injeção de dependências
2. **Operações**
   - Métodos claros e coesos
   - Tratamento de erros
   - Logging apropriado

Camada de Infraestrutura

A camada de infraestrutura implementa os detalhes técnicos e interfaces definidas nas camadas internas, seguindo o princípio de inversão de dependências.

Responsabilidades

1. **Persistência**
   - Implementação dos repositórios
   - Configuração do JPA/Hibernate
   - Gerenciamento de transações
2. **Configuração**
   - Setup do ambiente
   - Gestão de dependências
   - Configurações do sistema

Estrutura Principal

```
infrastructure/
├── persistence/      # Implementações de repositórios
├── config/          # Configurações do sistema
└── exceptions/      # Tratamento de erros
```

Princípios Adotados

- Inversão de Dependências
- Injeção de Dependências
- Separação de Responsabilidades

- Configuração Centralizada

Persistência

Implementação Base

O projeto utiliza uma implementação base de repositório que fornece operações CRUD comuns:

```
public abstract class CrudRepository<T extends BaseEntity<ID>, ID>
{
    protected final EntityManager entityManager;

    // Operações básicas: save, findById, delete, etc.
}
```

Configuração JPA

```
Principais Configurações:
├── Hibernate como provedor JPA
├── Connection pool com HikariCP
└── Transações gerenciadas por JPA
```

Boas Práticas

1. **Performance**
   - Consultas otimizadas
   - Paginação quando necessário
   - Índices apropriados
2. **Transações**
   - Escopo bem definido
   - Isolamento adequado

- Tratamento de deadlocks

3. **Segurança**
   - Prevenção de SQL Injection
   - Validação de inputs
   - Controle de acesso

Exemplo de Uso

```
@Repository
public class UserRepositoryImpl extends CrudRepository<User, Long>
{
    // Implementações específicas para User
}
```

Configuração

Configurações do Sistema

Banco de Dados

```
# application.properties
spring.datasource.url=jdbc:mysql://localhost:3306/transaction_db
spring.datasource.username=transaction_user
spring.datasource.password=password123
```

JPA/Hibernate

```
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQ
LDialect
spring.jpa.open-in-view=false
```

Gestão de Ambiente

Docker

```yaml
version: '3.8'
services:
  mysql:
    image: mysql:8.0
    ports:
      - "3306:3306"
    environment:
      MYSQL_DATABASE: transaction_db
      MYSQL_USER: transaction_user
      MYSQL_PASSWORD: password123
```

Boas Práticas

1. **Segurança**
    - Senhas em variáveis de ambiente
    - HTTPS em produção
    - Logs seguros
2. **Performance**
    - Connection pool otimizado
    - Cache configurado
    - Timeouts apropriados
3. **Manutenção**
    - Configurações externalizadas
    - Perfis de ambiente
    - Logs estruturados

Camada de Apresentação

A camada de apresentação implementa a interface gráfica usando JavaFX e segue o padrão MVVM (Model-View-ViewModel) para separação de responsabilidades.

Estrutura

```
presentation/
├── controllers/     # Controladores JavaFX
├── viewmodels/     # ViewModels para binding
└── views/          # Arquivos FXML
```

Padrão MVVM

1. **View (FXML)**
   - Layout da interface
   - Estilos CSS
   - Bindings declarativos
2. **ViewModel**
   - Estado da tela
   - Lógica de apresentação
   - Comandos e bindings
3. **Controller**
   - Inicialização da View
   - Injeção de dependências
   - Configuração de bindings

Boas Práticas

1. **Separação de Responsabilidades**
    - View para apresentação
    - ViewModel para estado
    - Controller para coordenação
2. **Reatividade**
    - Properties observáveis
    - Bindings bidirecionais
    - Atualizações automáticas
3. **Validação**
    - Feedback visual
    - Mensagens de erro
    - Estados de loading

Controllers

Estrutura Base

```
public abstract class BaseController {
    @FXML
    private Parent root;

    protected final ValidationSupport validationSupport;
    protected final ErrorHandler errorHandler;

    // Métodos comuns
}
```

Implementações

LoginController

```
public class LoginController extends BaseController {
    @FXML
    private TextField emailField;
    @FXML
    private PasswordField passwordField;
    @FXML
    private Button loginButton;

    private final LoginViewModel viewModel;

    @Override
    public void initialize() {
        setupValidation();
        setupBindings();
    }
}
```

Boas Práticas

1. **Inicialização**
   - `@FXML` para injeção
   - Setup no `initialize()`
   - Validação inicial
2. **Tratamento de Erros**
   - Mensagens amigáveis
   - Logging apropriado
   - Recuperação graceful
3. **Navegação**
   - Transições suaves
   - Estado preservado
   - Loading indicators

Exemplo de Uso

```
@FXML
private void handleLogin() {
    if (!validationSupport.isValid()) {
        return;
    }

    viewModel.loginCommand()
        .thenAccept(this::navigateToHome)
        .exceptionally(this::handleError);
}
```

Views

Estrutura FXML

Login View

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.layout.VBox?>
<?import javafx.scene.control.*?>

<VBox spacing="10" alignment="CENTER" styleClass="login-
container">
    <TextField fx:id="emailField"
               promptText="Email"/>
    <PasswordField fx:id="passwordField"
                   promptText="Senha"/>
    <Button fx:id="loginButton"
            text="Entrar"
            onAction="#handleLogin"/>
</VBox>
```

Estilos CSS

```
.login-container {
    -fx-padding: 20;
    -fx-background-color: white;
}

.text-field, .password-field {
    -fx-pref-width: 250px;
}

.button {
    -fx-background-color: #2196F3;
```

```
    -fx-text-fill: white;
}
```

Boas Práticas

1. **Layout**
   - Responsivo
   - Acessível
   - Consistente
2. **Estilos**
   - Reutilizáveis
   - Temáveis
   - Manuteníveis
3. **Usabilidade**
   - Feedback visual
   - Tab navigation
   - Atalhos de teclado

Componentes Comuns

1. **Dialogs**
   - Confirmação
   - Erro
   - Progresso
2. **Forms**

- Validação visual
- Campos obrigatórios
- Auto-complete

3. **Tables**
    - Paginação
    - Ordenação
    - Filtros

Padrões de Design

Visão Geral

O Módulo de Transações implementa diversos padrões de design para promover código limpo, manutenível e extensível. Esta seção documenta os principais padrões utilizados, suas implementações e benefícios para o projeto.

Padrões Principais

1. **Adapter Pattern**
   - Converte interfaces de classes em outras interfaces esperadas pelo cliente
   - Permite que classes com interfaces incompatíveis trabalhem juntas
   - Utilizado principalmente na camada de infraestrutura para adaptar serviços externos
2. **Dependency Inversion Principle (DIP)**
   - Módulos de alto nível não dependem de módulos de baixo nível, ambos dependem de abstrações
   - Abstrações não dependem de detalhes, detalhes dependem de abstrações
   - Implementado através de interfaces e injeção de dependências
3. **Facade Pattern**
   - Fornece uma interface unificada para um conjunto de interfaces em um subsistema
   - Simplifica o uso de subsistemas complexos
   - Utilizado para encapsular operações complexas em interfaces simples
4. **Factory Pattern**
   - Centraliza a criação de objetos complexos
   - Encapsula a lógica de instanciação

- Facilita a criação de objetos com configurações específicas

Benefícios no Projeto

- **Desacoplamento**: Redução de dependências diretas entre componentes
- **Testabilidade**: Facilidade para criar mocks e stubs para testes
- **Manutenibilidade**: Código mais organizado e com responsabilidades bem definidas
- **Extensibilidade**: Facilidade para adicionar novas funcionalidades sem modificar código existente

Implementação

Cada padrão é implementado de acordo com as necessidades específicas do módulo, seguindo as melhores práticas e adaptando-se à arquitetura geral do sistema. As seções a seguir detalham cada padrão, com exemplos concretos de implementação no projeto.

Adapter Pattern

Visão Geral

O padrão **Adapter** é um padrão de projeto estrutural que permite que classes com interfaces incompatíveis trabalhem juntas, convertendo a interface de uma classe em outra esperada por um cliente. No contexto da arquitetura hexagonal, ele é comumente aplicado para adaptar implementações técnicas (como UI e persistência) às interfaces de domínio (ports).

Implementação no Projeto

BeneficiaryController como Adapter

A classe `BeneficiaryController` atua como um **adaptador da interface do usuário (JavaFX)** para os casos de uso da aplicação (`BeneficiaryUseCases`). Ela converte ações do usuário (eventos de botão, preenchimento de formulário) em chamadas aos métodos da camada de aplicação.

```
public class BeneficiaryController {
    private final BeneficiaryUseCases beneficiaryUseCases;

    public BeneficiaryController() {
        this.beneficiaryUseCases = new BeneficiaryUseCases(
            new BeneficiaryRepositoryImp(entityManager),
            new AccountRepositoryImp(entityManager),
            new AccountBeneficiaryRepositoryImp(entityManager),
            new ValidationService()
        );
    }

    public void saveBeneficiary() {
        BeneficiaryDTO dto = ...
        beneficiaryUseCases.addBeneficiary(dto, currentAccountId);
```

```
    }
}
```

Repositórios como Adapters

As classes `BeneficiaryRepositoryImp`, `AccountRepositoryImp` e `AccountBeneficiaryRepositoryImp` são **adapters da infraestrutura (JPA)** para o domínio. Elas implementam interfaces (`BeneficiaryRepository`, etc.) que definem operações esperadas no domínio, mas usando tecnologia específica (JPA).

```
public class BeneficiaryRepositoryImp implements
BeneficiaryRepository {
    private final EntityManager em;

    public void addBeneficiary(Beneficiary b) {
        em.persist(b);
    }
}
```

Estrutura do Adapter no Projeto

Papel do Adapter Pattern	Elemento no Projeto	Descrição
Target (Interface esperada)	`BeneficiaryRepository`, `AccountRepository`, etc.	Interfaces do domínio que definem o contrato esperado.
Adapter (classe adaptadora)	`BeneficiaryRepositoryImp`, `AccountRepositoryImp`	Implementam as interfaces usando JPA.
Adaptee (API/tecnologia real)	`EntityManager`, `JPA`, `JavaFX`, etc.	Ferramentas externas que estão sendo adaptadas.
Cliente	`BeneficiaryUseCases`, `BeneficiaryController`	Usam as interfaces e não conhecem as implementações técnicas.

Diagrama Estrutural

Benefícios no Projeto

1. **Desacoplamento de Tecnologia**
   - As camadas superiores (casos de uso e controladores) usam apenas interfaces, sem conhecimento de JPA, JavaFX, etc.
2. **Testabilidade**
   - Pode-se substituir facilmente os adapters por mocks em testes, já que dependem apenas de interfaces.
3. **Facilidade de Substituição**
   - É possível trocar o mecanismo de persistência ou interface gráfica sem afetar a lógica de negócios.
4. **Organização por Camadas**
   - Fica claro que os adapters pertencem à camada de infraestrutura (persistência) ou apresentação (UI), mantendo o domínio isolado.

Uso no Código

Interface (Target)

```
public interface BeneficiaryRepository {
    void addBeneficiary(Beneficiary beneficiary);
    void removeBeneficiary(Long id);
}
```

Adapter de Persistência (Implementação JPA)

```
public class BeneficiaryRepositoryImp implements
BeneficiaryRepository {
    private final EntityManager em;
```

```
    public void addBeneficiary(Beneficiary beneficiary) {
        em.persist(beneficiary);
    }
}
```

Adapter de Apresentação (Controller)

```
public class BeneficiaryController {
    private final BeneficiaryUseCases useCases;

    public void saveBeneficiary() {
        BeneficiaryDTO dto = ...
        useCases.addBeneficiary(dto, currentAccountId);
    }
}
```

Considerações de Design

1. **Quando Usar**
   - Quando há necessidade de **conectar uma camada desacoplada** (domínio) com implementações concretas (JPA, UI, APIs).
   - Quando você quer **separar lógica de negócio da infraestrutura técnica**.
2. **Vantagens**
   - Código mais limpo, modular, substituível e testável.
   - Redução do acoplamento entre camadas do sistema.
3. **Complementaridade com outros padrões**
   - O padrão Adapter se integra bem com os padrões **Hexagonal Architecture**, **Ports and Adapters**, e **Facade**, todos presentes nesse projeto.

Dependency Inversion Principle (DIP)

Visão Geral

Injeção de dependência é uma forma de escrever código mais flexível e desacoplado. Em vez de uma classe criar diretamente os objetos que precisa para funcionar, ela recebe esses objetos de fora, geralmente por meio do construtor, de um método ou de um framework.

Implementação no Projeto

A interface `AuthenticationService` atua como uma abstração da lógica de sua classe onde é implementada, servindo para desacoplamento, onde uma classe pode usar métodos de outra classe sem que saiba a sua lógica ou dependa da sua implementação.

```
/**
 * Service that encapsulates the authentication of users.
 */
public interface AuthenticationService {

    /**
     * Authenticates a user and returns a JWT token that can be
used to access protected resources.
     * @param email the user email
     * @param password the user password
     * @return a JWT token
     */
    String authenticate(String email, String password);

    /**
     * Retrieves a user from a given JWT token.
     * @param token the JWT token
     * @return the user associated with the token
     */
```

```
    UserDTO getUserFromToken(String token);
}
```

AuthenticationServiceImpl como implementação

Onde consegue ter uma ou mais implementações, sendo a principal dela a `AuthenticationServiceImpl` onde fica sua real lógica.

```
/**
 * Implementation of the authentication service that handles user
authentication and token management.
 * This service is responsible for validating user credentials and
managing JWT tokens.
 *
 * @version 1.0
 * @see AuthenticationService
 * @see SignupService
 * @see JwtUtil
 */
@Service
public class AuthenticationServiceImpl implements
AuthenticationService {

    /** Service for user registration and retrieval operations */
    private final SignupService signupService;

    /**
     * Constructs a new AuthenticationServiceImpl.
     * Initializes the required signup service for user
operations.
     */
    public AuthenticationServiceImpl() {
        this.signupService = new SignupServiceImp();
    }

    /**
     * Authenticates a user with their email and password.
```

```
     * If authentication is successful, generates and returns a
JWT token.
     *
     * @param email    the user's email address
     * @param password the user's password in plain text
     * @return JWT token string if authentication is successful
     * @throws RuntimeException if the user is not found or
credentials are invalid
     */
    @Override
    public String authenticate(String email, String password) {
        UserDTO userByEmail =
signupService.findUserByEmail(email);
        if (!isValidUser(userByEmail, password)) throw new
RuntimeException("Invalid user or password");
        return JwtUtil.generateToken(userByEmail);
    }

    /**
     * Retrieves user information from a JWT token.
     * Validates the token and extracts the user details.
     *
     * @param token the JWT token string
     * @return UserDTO containing the user information from the
token
     * @throws RuntimeException if the token is blank or invalid
     */
    @Override
    public UserDTO getUserFromToken(String token) {
        if (token.isBlank()) throw new RuntimeException("Invalid
token");
        return JwtUtil.getUserFromToken(token);
    }

    /**
     * Validates a user's password against the stored hash.
     * Uses BCrypt for secure password verification.
     *
```

```
     * @param userDTO  the user DTO containing the hashed password
     * @param password the plain text password to verify
     * @return true if the password is valid, false otherwise
     */
    private boolean isValidUser(UserDTO userDTO, String password)
{
        return BCrypt.checkpw(password, userDTO.getPassword());
    }
}
```

Uso No Código

```
private final AuthenticationService authenticationService;

public LoginController() {
    this.authenticationService = new AuthenticationServiceImpl();
}
```

O construtor do `LoginController` recebe a injeção de dependência onde a interface recebe a instância de sua implementação, conseguindo assim acessar seus métodos de forma indireta através da interface.

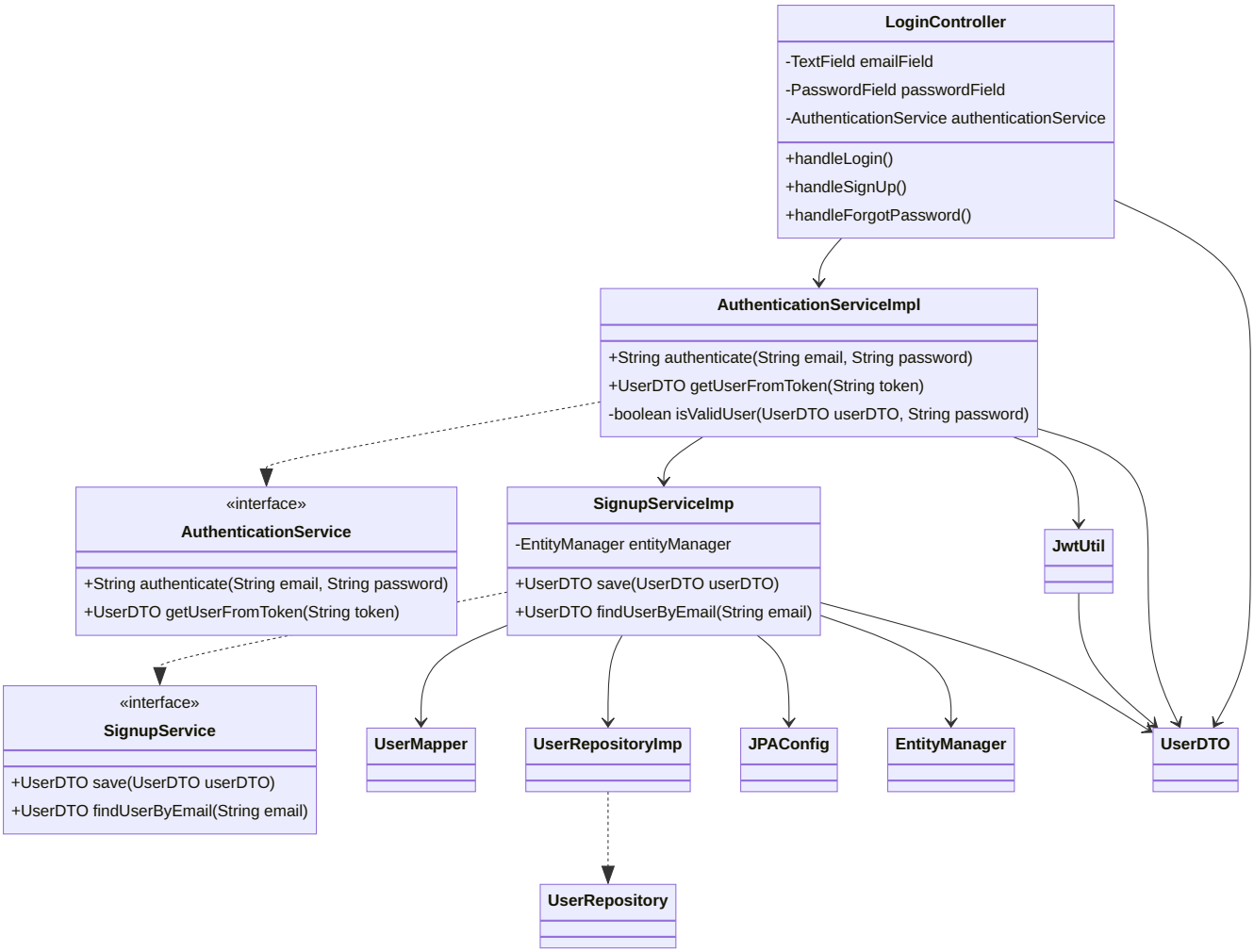
```
private void handleLogin() {
    String email = emailField.getText();
    String password = passwordField.getText();

    try {
        // Tenta autenticar
        String token = authenticationService.authenticate(email,
password);
        UserDTO userFromToken =
this.authenticationService.getUserFromToken(token);
```

No método `handleLogin` onde ele acessa seus métodos.

Benefícios no Projeto

1. **Desacoplamento entre componentes**
   - O `LoginController` depende apenas da interface `AuthenticationService`, e não da sua implementação concreta. Isso reduz o acoplamento entre as classes e torna o código mais modular e fácil de entender.
2. **Facilidade para trocar implementações**
   - Como o controlador depende de uma abstração, é possível substituir a implementação por outra (como uma versão mockada ou uma integração com um serviço externo) sem modificar o código do consumidor.
3. **Aumento da testabilidade**
   - Durante os testes, é possível injetar implementações falsas ou controladas da interface, simulando diferentes comportamentos e cenários sem depender da lógica real de autenticação.
4. **Centralização da lógica de construção**
   - O ponto onde a implementação concreta é instanciada pode ser isolado (como em um framework de injeção ou classe de configuração), mantendo o restante do sistema limpo e sem responsabilidade de criar objetos.
5. **Flexibilidade e extensibilidade**
   - Novas formas de autenticação podem ser adicionadas ao sistema simplesmente criando novas classes que implementam a interface, sem precisar modificar os consumidores já existentes.

Diagrama



Considerações de Design

1. **Quando Usar**

- Quando há necessidade de conectar camadas de alto nível (como serviços e controladores) com implementações de baixo nível (como repositórios JPA ou utilitários de criptografia) sem acoplamento direto.
- Quando se deseja separar claramente as regras de negócio da infraestrutura técnica, mantendo o foco do sistema na lógica de domínio.

2. **Vantagens**

- Torna o código mais limpo, modular, substituível e testável, pois as dependências são baseadas em abstrações e não em implementações concretas.

- Facilita a troca de implementações (por exemplo, trocar um repositório real por um mock) sem impacto nas camadas superiores.
- Permite maior facilidade na injeção de dependência e no uso de frameworks como Spring, promovendo boas práticas de teste e manutenção.
- Reduz significativamente o acoplamento entre camadas, promovendo uma arquitetura mais estável e evolutiva.

3. **Complementaridade com outros padrões**

- Integra-se bem com o padrão Adapter, que converte interfaces para atender as expectativas das camadas superiores.
- É um dos pilares da Arquitetura Hexagonal (Ports and Adapters), reforçando a separação entre domínio e infraestrutura.
- Trabalha em conjunto com o padrão Facade ao esconder complexidades de infraestrutura atrás de interfaces simples.
- Facilita a implementação de testes unitários com mocks ou stubs ao permitir a substituição das dependências por versões controladas.

Facade Pattern

Visão Geral

O padrão **Facade** é um padrão estrutural que tem como objetivo fornecer uma interface simplificada para um conjunto de interfaces em um subsistema. Em outras palavras, ele oferece uma camada de abstração que centraliza e simplifica o acesso a diferentes componentes ou serviços de um sistema. No contexto do Módulo de Beneficiários, este padrão é utilizado para encapsular a complexidade dos diferentes repositórios e serviços, oferecendo uma interface simplificada e organizada para os casos de uso e o controlador.

Implementação no Projeto

BeneficiaryUseCases

A classe `BeneficiaryUseCases` atua como uma fachada, centralizando a lógica de negócios relacionada ao gerenciamento de beneficiários. Ela coordena o acesso aos repositórios e serviços necessários para realizar operações de adição, edição, listagem e remoção de beneficiários.

```
public class BeneficiaryUseCases {
    private final BeneficiaryRepository beneficiaryRepository;
    private final AccountRepository accountRepository;
    private final AccountBeneficiaryRepository
accountBeneficiaryRepository;
    private final ValidationService validationService;

    public BeneficiaryUseCases(
            BeneficiaryRepository beneficiaryRepository,
            AccountRepository accountRepository,
            AccountBeneficiaryRepository
accountBeneficiaryRepository,
            ValidationService validationService) {
        this.beneficiaryRepository = beneficiaryRepository;
        this.accountRepository = accountRepository;
        this.accountBeneficiaryRepository =
```

```
accountBeneficiaryRepository;
        this.validationService = validationService;
    }
}
```

BeneficiaryUseCases como Facade

A classe `BeneficiaryUseCases` fornece métodos simplificados para os casos de uso que envolvem as operações de beneficiários. Através dessa interface, outras partes do sistema podem realizar operações complexas sem se preocupar com a lógica interna de cada repositório ou serviço utilizado.

Exemplos de Métodos no BeneficiaryUseCases

- **Adição de Beneficiário:** O método `addBeneficiary` centraliza a lógica de validação, verificação de existência de beneficiários e criação da relação entre conta e beneficiário.

```
public void addBeneficiary(BeneficiaryDTO beneficiaryDTO, Long
accountId) {
    if (!validationService.validateBeneficiary(beneficiaryDTO)) {
        throw new IllegalArgumentException("Dados do beneficiário
inválidos");
    }

    // Verificação e criação do beneficiário
    Account account = accountRepository.findById(accountId);
    Beneficiary beneficiary = new Beneficiary(beneficiaryDTO);
    beneficiaryRepository.addBeneficiary(beneficiary);

    // Criação da relação
    AccountBeneficiary accountBeneficiary = new
AccountBeneficiary(account, beneficiary);
    accountBeneficiaryRepository.save(accountBeneficiary);
}
```

- **Remoção de Beneficiário:** O método `removeBeneficiary` simplifica o processo de remoção, abstraindo a necessidade de lidar diretamente com repositórios de dados.

```
public void removeBeneficiary(Long beneficiaryId) {
    beneficiaryRepository.removeBeneficiary(beneficiaryId);
}
```

RemoveBeneficiaryController

O controlador `RemoveBeneficiaryController` funciona como uma fachada no contexto da camada de apresentação. Ele simplifica a interação com a camada de serviço, oferecendo uma interface simplificada para o usuário final ao remover um beneficiário.

```
public class RemoveBeneficiaryController {
    private final BeneficiaryUseCases beneficiaryUseCases;
    private Long accountId;

    public void setBeneficiaryUseCases(BeneficiaryUseCases
beneficiaryUseCases) {
        this.beneficiaryUseCases = beneficiaryUseCases;
    }

    public void handleRemoveBeneficiary() {
        if (beneficiarySelected()) {

beneficiaryUseCases.removeBeneficiary(selectedBeneficiaryId());
            reloadBeneficiariesList();
        }
    }
}
```

Benefícios no Projeto

1. **Centralização da Lógica de Negócio**

- A classe `BeneficiaryUseCases` centraliza a lógica relacionada ao gerenciamento de

beneficiários e suas relações com contas bancárias.

- Esse design facilita a manutenção e evolução da lógica de negócios, pois todas as operações complexas são realizadas em um único lugar.

2. **Facilidade de Manutenção**

- Mudanças na lógica de negócios (como validações ou regras de criação de beneficiários) podem ser feitas sem impactar diretamente os controladores ou outras partes do sistema que consomem essa fachada.

3. **Desacoplamento**

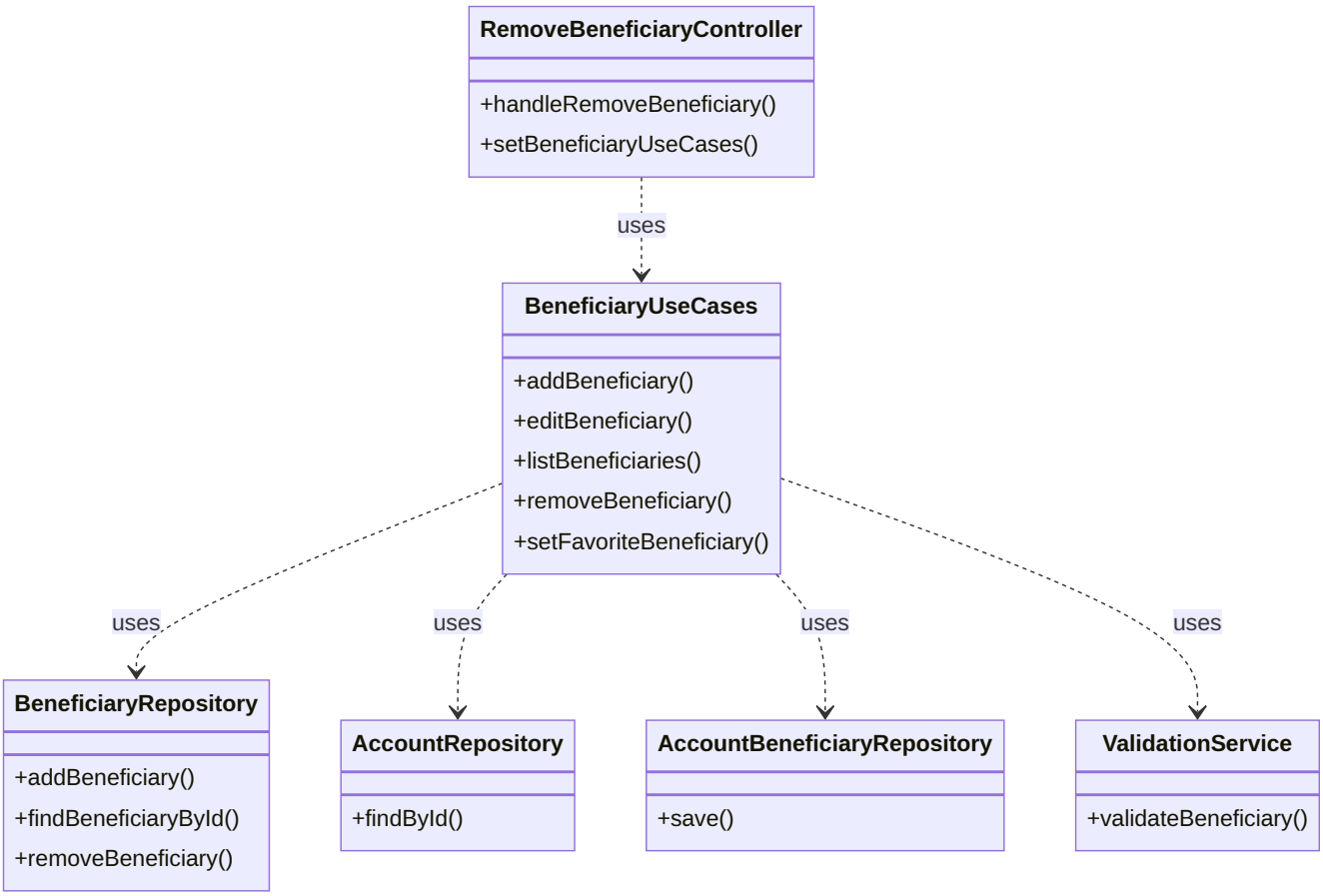
- O controlador `RemoveBeneficiaryController` não precisa entender a complexidade das operações de remoção de beneficiário. Ele apenas invoca a fachada para realizar a ação.

- Esse desacoplamento facilita a testabilidade e a escalabilidade do sistema.

4. **Simples Interface para os Clientes**

- O uso do padrão **Facade** oferece uma interface simples para interagir com os repositórios e serviços necessários, sem a necessidade de lidar com as complexidades internas.

Diagrama



Uso no Código

Exemplo de Uso no Controlador

```
public class RemoveBeneficiaryController {
    private BeneficiaryUseCases beneficiaryUseCases;
    private Long accountId;

    public void handleRemoveBeneficiary() {
        if (beneficiarySelected()) {

beneficiaryUseCases.removeBeneficiary(selectedBeneficiaryId());
            reloadBeneficiariesList();
        }
    }
}
```

Exemplo de Uso no Caso de Uso

```
public void removeBeneficiary(Long beneficiaryId) {
    beneficiaryRepository.removeBeneficiary(beneficiaryId);
}
```

Considerações de Design

1. **Quando Usar**

- Quando você precisa de uma interface simplificada para acessar funcionalidades complexas ou distribuídas entre diferentes componentes.
- Quando deseja centralizar a lógica de negócios, facilitando manutenção e evolução.
- Quando deseja desacoplar a camada de apresentação (controladores) da lógica de negócio.

2. **Alternativas Consideradas**

- **Service Layer**: Embora útil, o padrão Service Layer pode ser excessivamente detalhado em sistemas simples, enquanto o Facade oferece uma interface mais simplificada.
- **Adapter**: O Adapter é mais voltado para converter interfaces, enquanto o Facade oferece uma interface simplificada para um conjunto de funcionalidades.

3. **Evolução Futura**

- Avaliar a introdução de mais fachadas para outros casos de uso no sistema, especialmente se houver mais operações complexas a serem realizadas.
- Considerar a implementação de uma **Abstract Factory** para gerenciar a criação de objetos mais complexos dentro da fachada.

Factory Pattern

Visão Geral

O Factory Pattern é um padrão de design criacional que fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados. No contexto do Módulo de Transações, este padrão é utilizado para encapsular a lógica de criação de objetos complexos e desacoplar a instanciação da implementação.

Implementação no Projeto

MockDataFactory

A principal implementação do Factory Pattern no projeto é a classe `MockDataFactory`, que centraliza a criação de provedores de dados mockados para testes e desenvolvimento:

```
public class MockDataFactory {

    private static final TransactionHistoryMockProvider
transactionHistoryMockProvider = new
TransactionHistoryMockProvider();
    private static final TransactionDetailsMockProvider
transactionDetailsMockProvider = new
TransactionDetailsMockProvider();

    public static TransactionHistoryMockProvider
getTransactionHistoryMockProvider() {
        return transactionHistoryMockProvider;
    }

    public static TransactionDetailsMockProvider
getTransactionDetailsMockProvider() {
        return transactionDetailsMockProvider;
    }
```

```
    public static boolean shouldUseMockData() {
        // Lógica para determinar se deve usar dados mockados
        return true;
    }
}
```

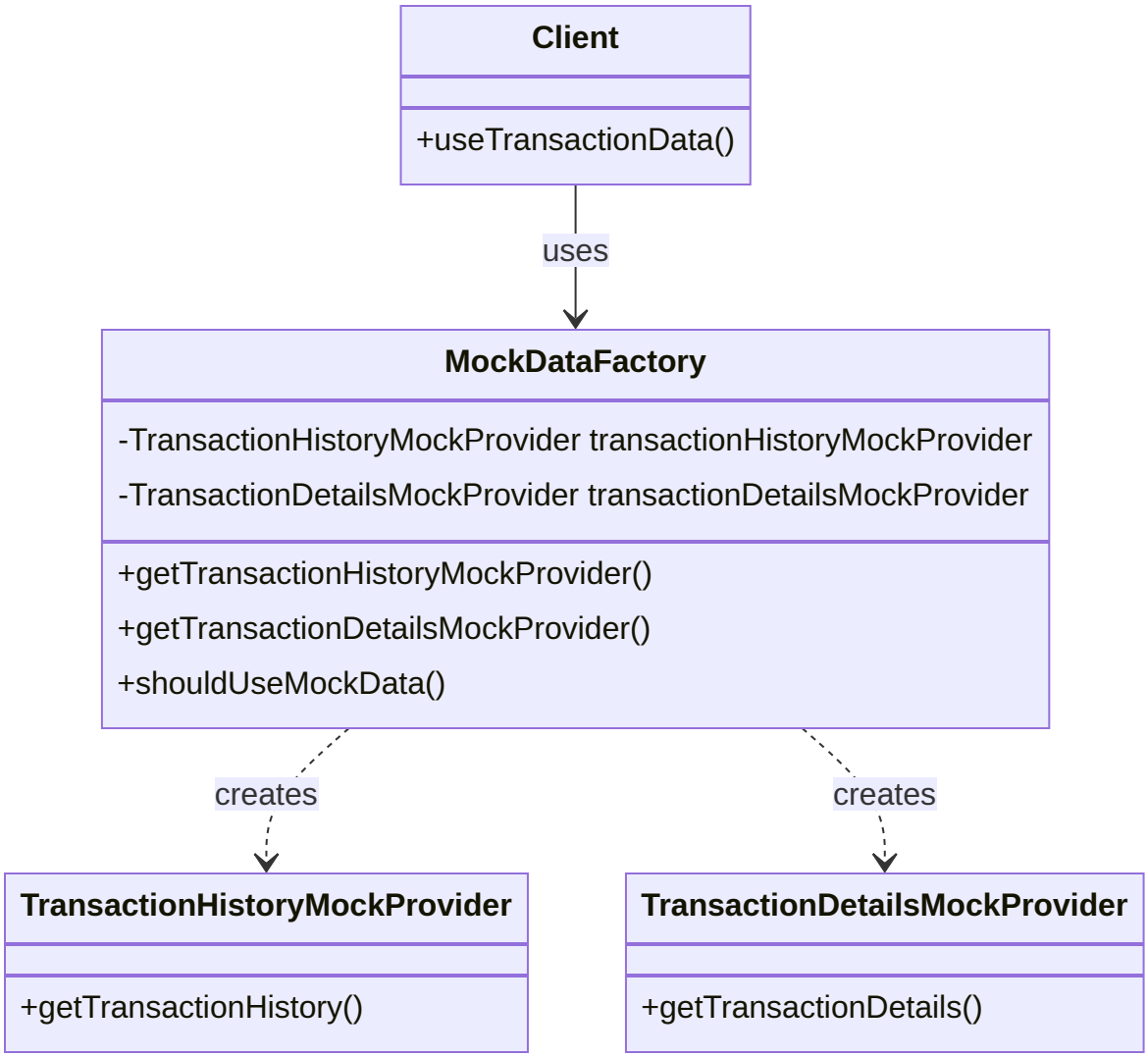
Esta implementação:

1. Encapsula a criação de provedores de dados mockados
2. Fornece métodos estáticos para acessar instâncias únicas (combinando com Singleton)
3. Oculta os detalhes de implementação dos clientes

Benefícios no Projeto

1. **Centralização da Criação de Objetos**
   - Toda a lógica de criação de provedores de dados mockados está em um único lugar
   - Facilita a manutenção e alterações futuras
2. **Desacoplamento**
   - Os clientes não precisam conhecer os detalhes de implementação dos provedores
   - Permite trocar implementações sem afetar o código cliente
3. **Flexibilidade**
   - Facilita a adição de novos tipos de provedores de dados
   - Permite controlar quando usar dados reais ou mockados

Diagrama



Uso no Código

```
// Exemplo de uso do Factory Pattern no código cliente
public class TransactionService {

    public List<TransactionHistoryDTO> getTransactionHistory(Long
userId) {
        if (MockDataFactory.shouldUseMockData()) {
            // Usa o factory para obter o provider de dados
mockados
            return
MockDataFactory.getTransactionHistoryMockProvider().getTransaction
History();
```

```
        } else {
            // Usa dados reais do repositório
            return transactionRepository.findByUserId(userId);
        }
    }
}
```

Variações no Projeto

Além da implementação principal, o padrão Factory também aparece em outras formas no projeto:

1. **Factory Methods em Serviços**
   - Métodos que encapsulam a criação de objetos complexos
   - Exemplo: `createUser()` em `UserService`
2. **Builder Pattern (uma variação do Factory)**
   - Usado com Lombok para criar objetos com muitos parâmetros opcionais
   - Exemplo: `UserDTO.builder().name("John").email("john@example.com").build()`

Considerações de Design

1. **Quando Usar**
   - Para encapsular a criação de objetos complexos
   - Quando a criação de objetos requer lógica que não deve ser exposta aos clientes
   - Para facilitar testes com dados mockados
2. **Alternativas Consideradas**
   - Dependency Injection: Útil para injetar implementações, mas o Factory oferece mais controle sobre a criação

- Service Locator: Mais complexo e pode introduzir acoplamento global

3. **Evolução Futura**
    - Considerar a implementação de uma Abstract Factory para famílias de objetos relacionados
    - Explorar o uso de Providers configuráveis para diferentes ambientes (dev, test, prod)

Diagramas UML

Esta seção apresenta os diagramas UML (Unified Modeling Language) utilizados para modelar o sistema de transações. Os diagramas UML fornecem uma representação visual da arquitetura, comportamento e estrutura do sistema, facilitando a compreensão e comunicação entre os membros da equipe de desenvolvimento.

Tipos de Diagramas

O projeto utiliza diversos tipos de diagramas UML para documentar diferentes aspectos do sistema:

1. **Diagramas de Caso de Uso**: Representam as interações entre os usuários (atores) e o sistema
2. **Diagramas de Classe**: Mostram a estrutura estática do sistema, incluindo classes, atributos e relacionamentos
3. **Diagramas de Sequência**: Ilustram a interação entre objetos ao longo do tempo
4. **Diagramas de Estado**: Representam os diferentes estados de um objeto durante seu ciclo de vida
5. **Diagramas de Atividade**: Mostram o fluxo de controle entre atividades

Ferramentas Utilizadas

Para a criação e manutenção dos diagramas UML, o projeto utiliza as seguintes ferramentas:

- **PlantUML**: Para diagramas baseados em texto que podem ser versionados
- **Draw.io/diagrams.net**: Para diagramas mais complexos e detalhados
- **Mermaid**: Para diagramas incorporados na documentação

Convenções

Para manter a consistência nos diagramas UML, o projeto segue estas convenções:

1. **Nomenclatura**: Nomes de classes em PascalCase, métodos em camelCase
2. **Cores**: Atores em azul, casos de uso em verde, classes de domínio em amarelo
3. **Detalhamento**: Nível apropriado de detalhes para o público-alvo
4. **Versionamento**: Diagramas são versionados junto com o código-fonte

Diagramas de Casos de Uso

Visão Geral

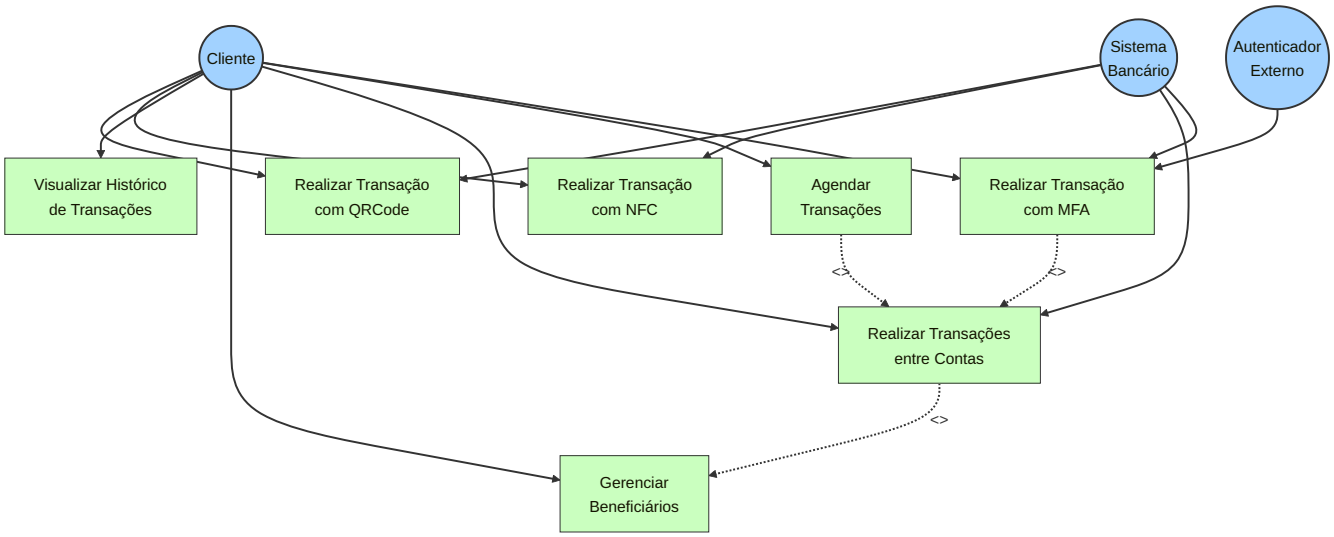
Os diagramas de casos de uso representam as interações entre os usuários (atores) e o sistema. Eles são utilizados para capturar os requisitos funcionais do sistema e ilustrar como os diferentes atores interagem com as funcionalidades disponíveis.

Atores Principais

- **Cliente**: Usuário final que utiliza o sistema para realizar transações financeiras
- **Autenticador Externo**: Sistema externo responsável pela autenticação multifatorial
- **Sistema Bancário**: Sistema externo que processa as transações financeiras

Diagrama Geral

O diagrama abaixo apresenta uma visão geral de todos os casos de uso do sistema e suas relações:



Casos de Uso Detalhados

1. Realizar Transação com QRCode (RN-01)

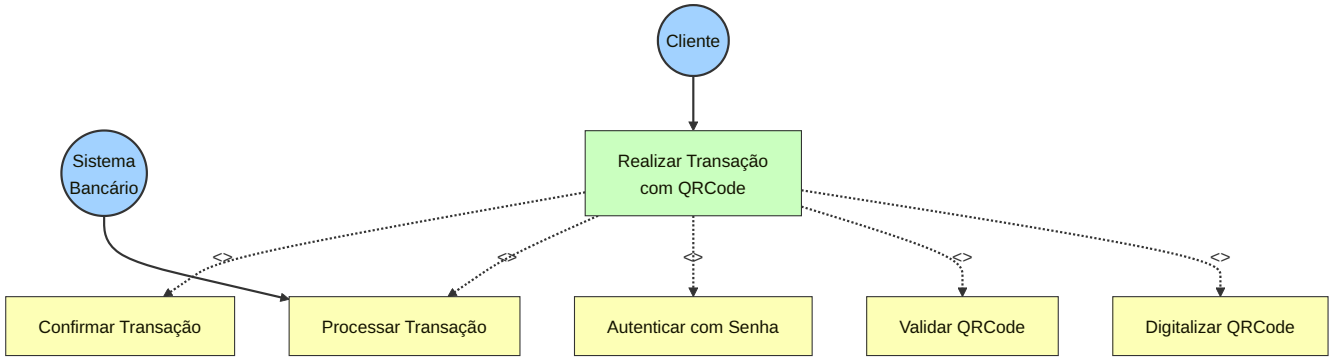
Ator Principal: Cliente

Fluxo Principal:

- Seleciona a opção de transação com QRCode
- Digitaliza o QRCode
- Valida o QRCode
- Exige senha

- Insere senha
- Valida a senha
- Processa a transação
- Recebe confirmação

Diagrama:



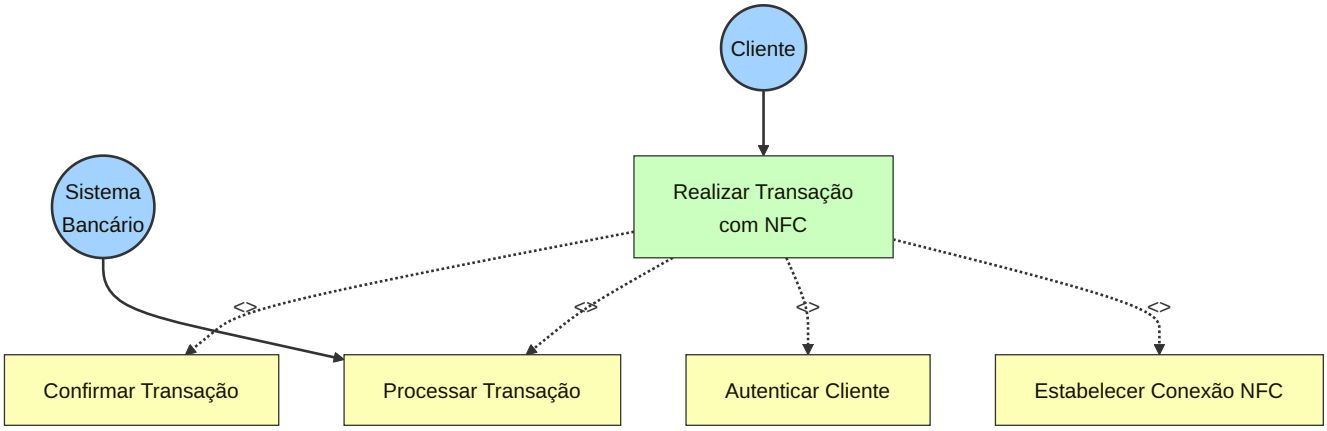
2. Realizar Transação com NFC (RN-02)

Ator Principal: Cliente

Fluxo Principal:

- Seleciona opção de transação NFC
- Estabelece conexão NFC
- Autentica o cliente
- Processa a transação
- Fornece confirmação

Diagrama:



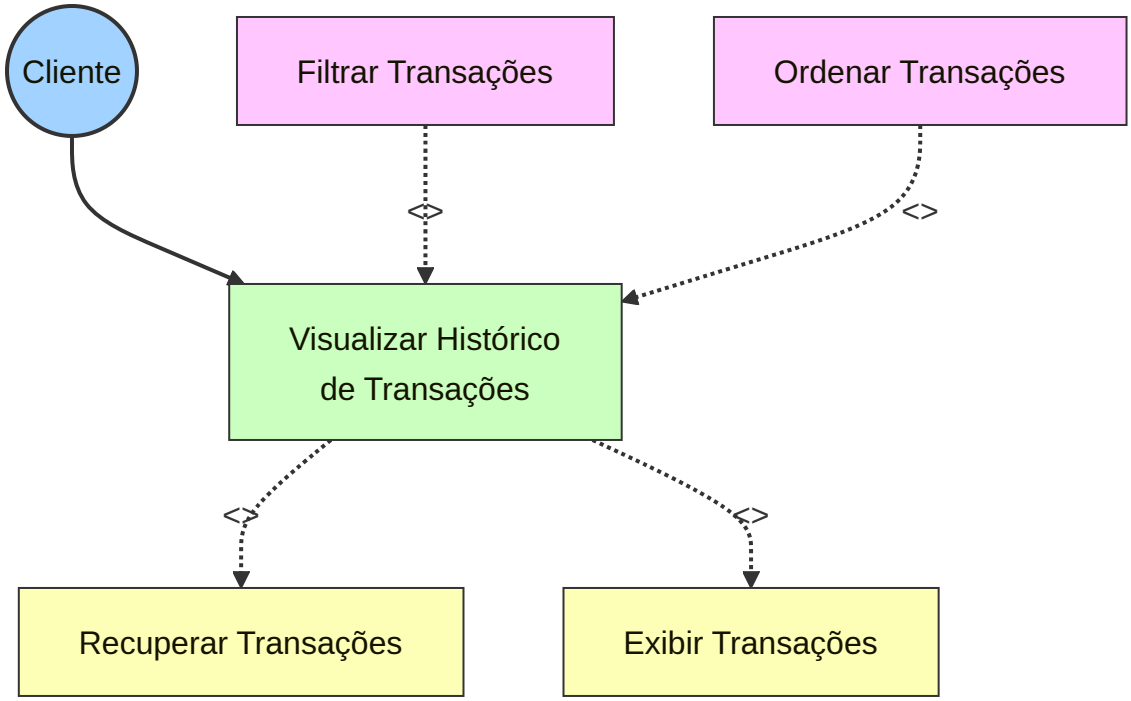
3. Histórico de Transações (RN-03)

Ator Principal: Cliente

Fluxo Principal:

- Fornece histórico de transações
- Recupera transações
- Exibe transações
- Filtra ou ordena as transações

Diagrama:



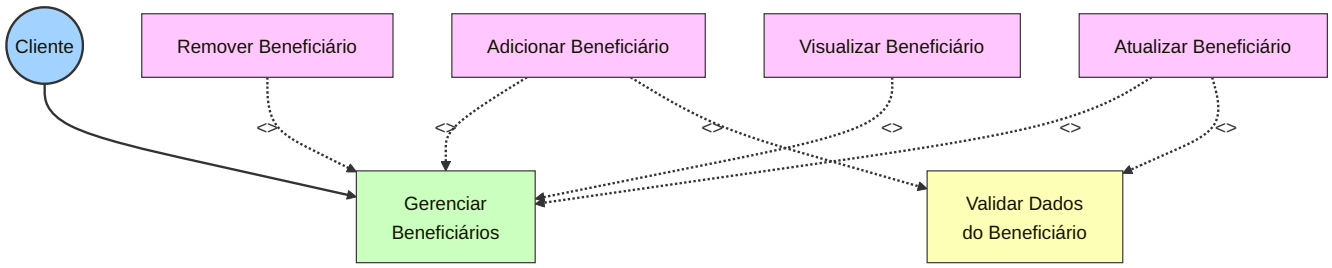
4. Gestão de Beneficiários (RN-04)

Ator Principal: Cliente

Fluxo Principal:

- Adiciona beneficiário
- Valida dados do beneficiário
- Remove beneficiário
- Atualiza beneficiário
- Visualiza beneficiário

Diagrama:



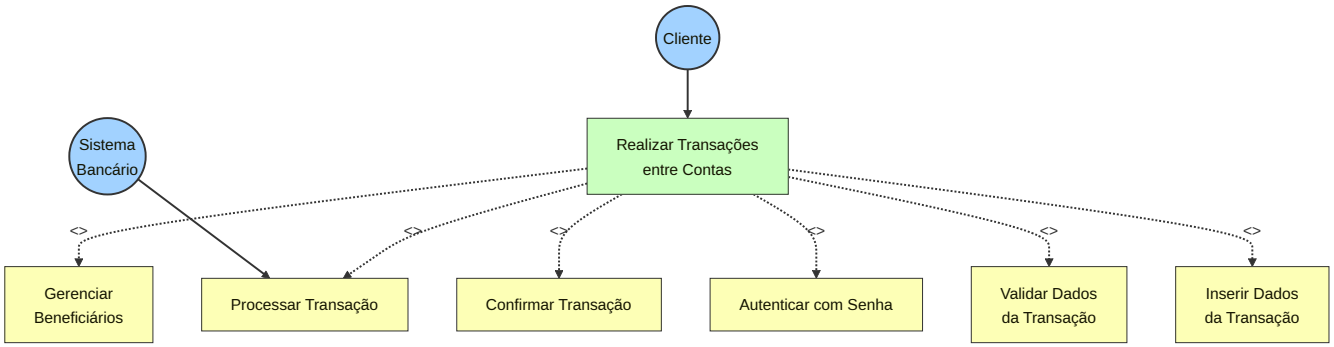
5. Transações entre Contas (RN-05)

Ator Principal: Cliente

Fluxo Principal:

- Seleciona opção de transação
- Insere dados da transação
- Valida dados da transação
- Exibe senha
- Insere senha
- Valida a senha
- Processa a transação
- Exibe a confirmação

Diagrama:



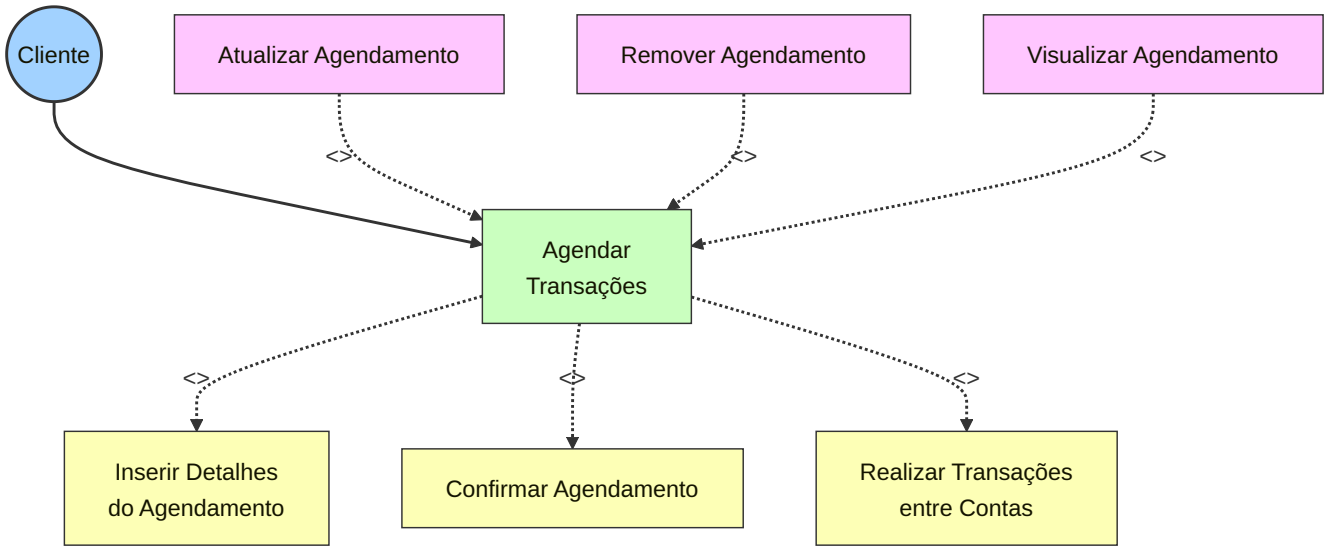
6. Agendamento de Transação (RN-06)

Ator Principal: Cliente

Fluxo Principal:

- Agenda transação
- Insere detalhes
- Confirma agendamento
- Atualiza ou remove agendamento
- Exibe agendamento

Diagrama:



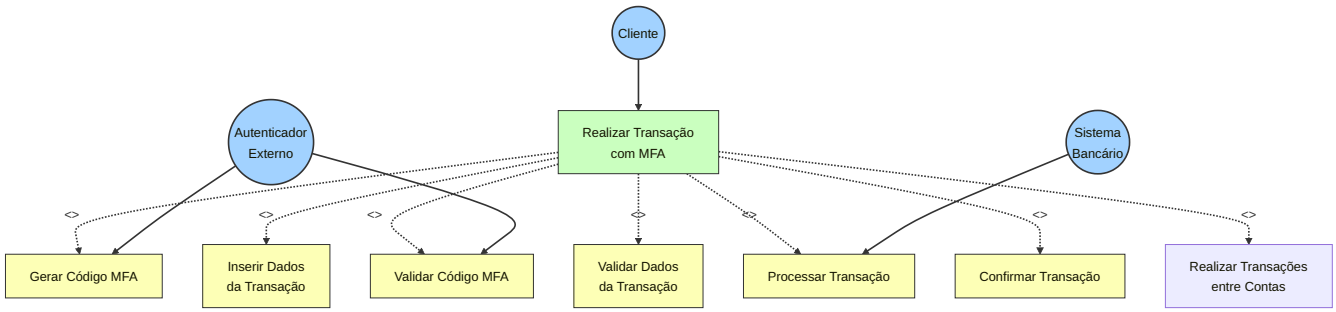
7. Transações com MFA (RN-09)

Atores: Cliente, Autenticador Externo

Fluxo Principal:

- Seleciona a opção de transação com MFA
- Insere dados para transação
- Valida os dados
- Exibe código
- Insere o código
- Valida a senha
- Processa a transação
- Mostra confirmação

Diagrama:



Relacionamentos entre Casos de Uso

Alguns casos de uso possuem relacionamentos entre si:

1. **Inclusão (< >)**: Um caso de uso inclui outro caso de uso como parte de seu fluxo
   - "Realizar Transações entre Contas" inclui "Gerenciar Beneficiários"
   - "Agendar Transações" inclui "Realizar Transações entre Contas"
2. **Extensão (< >)**: Um caso de uso estende outro caso de uso com comportamento adicional
   - "Transações com MFA" estende "Realizar Transações entre Contas" para valores elevados
   - "Filtrar Transações" estende "Visualizar Histórico de Transações"

Requisitos Não-Funcionais

Além dos casos de uso funcionais, o sistema também deve atender a requisitos não-funcionais:

1. **Segurança**: Todas as transações devem ser criptografadas
2. **Desempenho**: Transações devem ser processadas em menos de 3 segundos
3. **Disponibilidade**: O sistema deve estar disponível 99,9% do tempo
4. **Usabilidade**: Interface intuitiva para todos os casos de uso ``

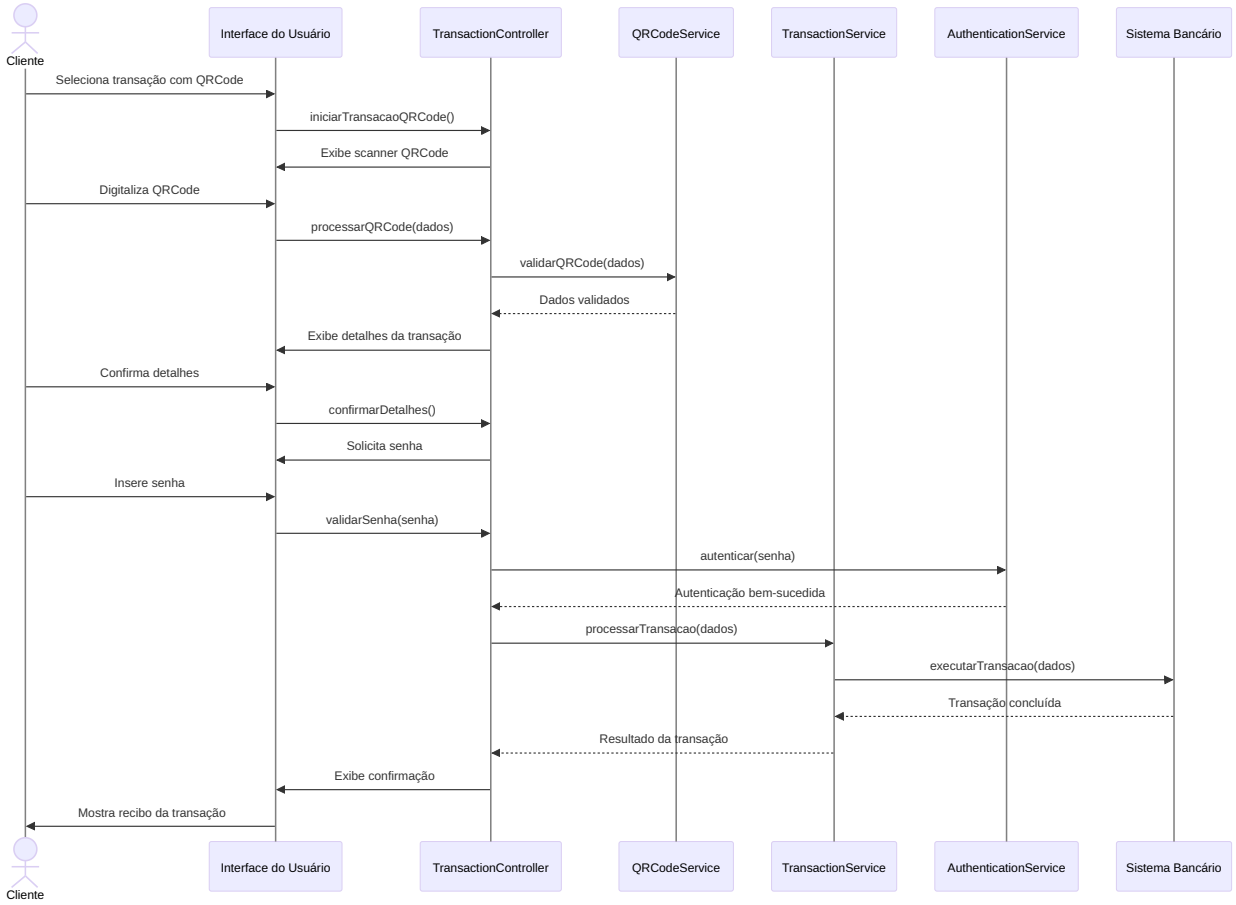
Diagramas de Sequência

Visão Geral

Os diagramas de sequência ilustram como os objetos interagem em um cenário específico de um caso de uso. Eles mostram a ordem cronológica das mensagens trocadas entre os objetos e são úteis para entender o fluxo de controle em um sistema orientado a objetos.

1. Realizar Transação com QRCode

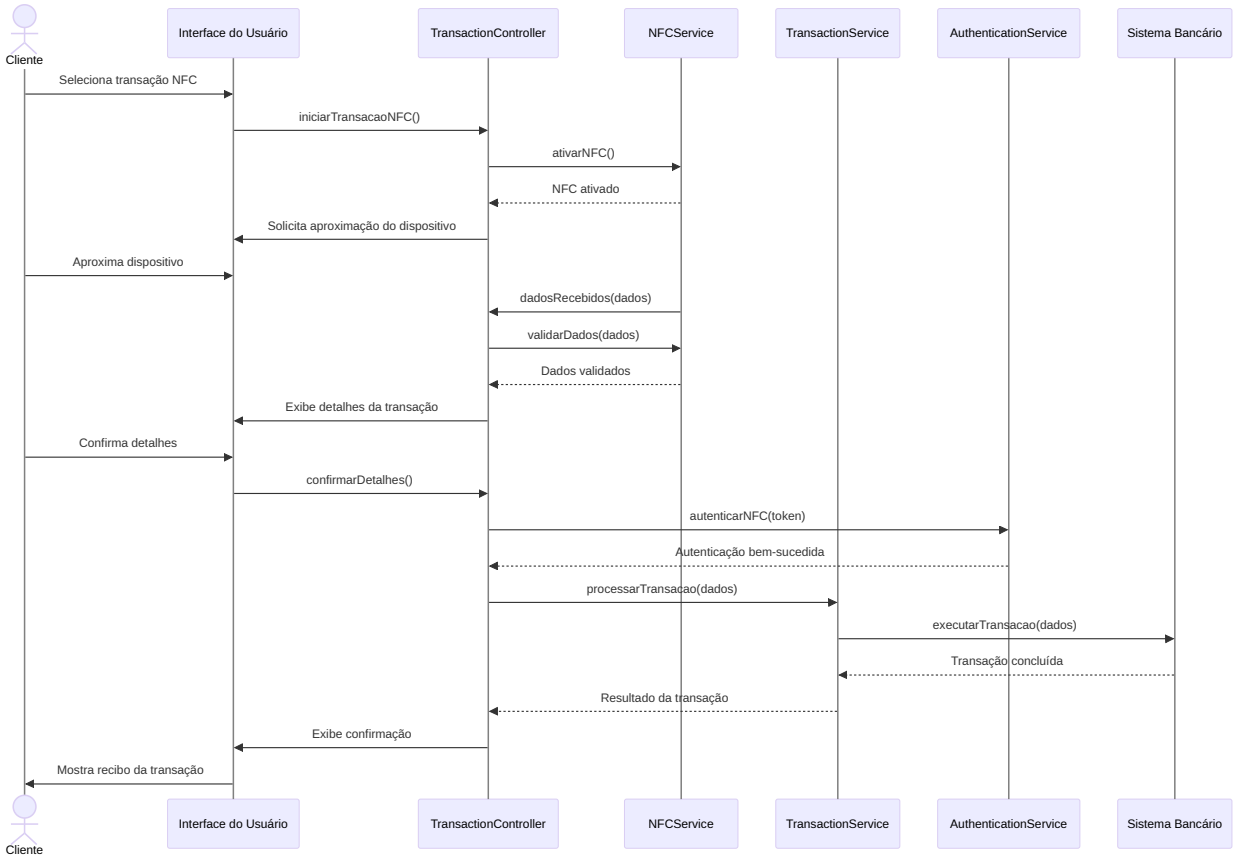
Este diagrama mostra a sequência de interações quando um cliente realiza uma transação usando QRCode.



2. Realizar Transação com NFC

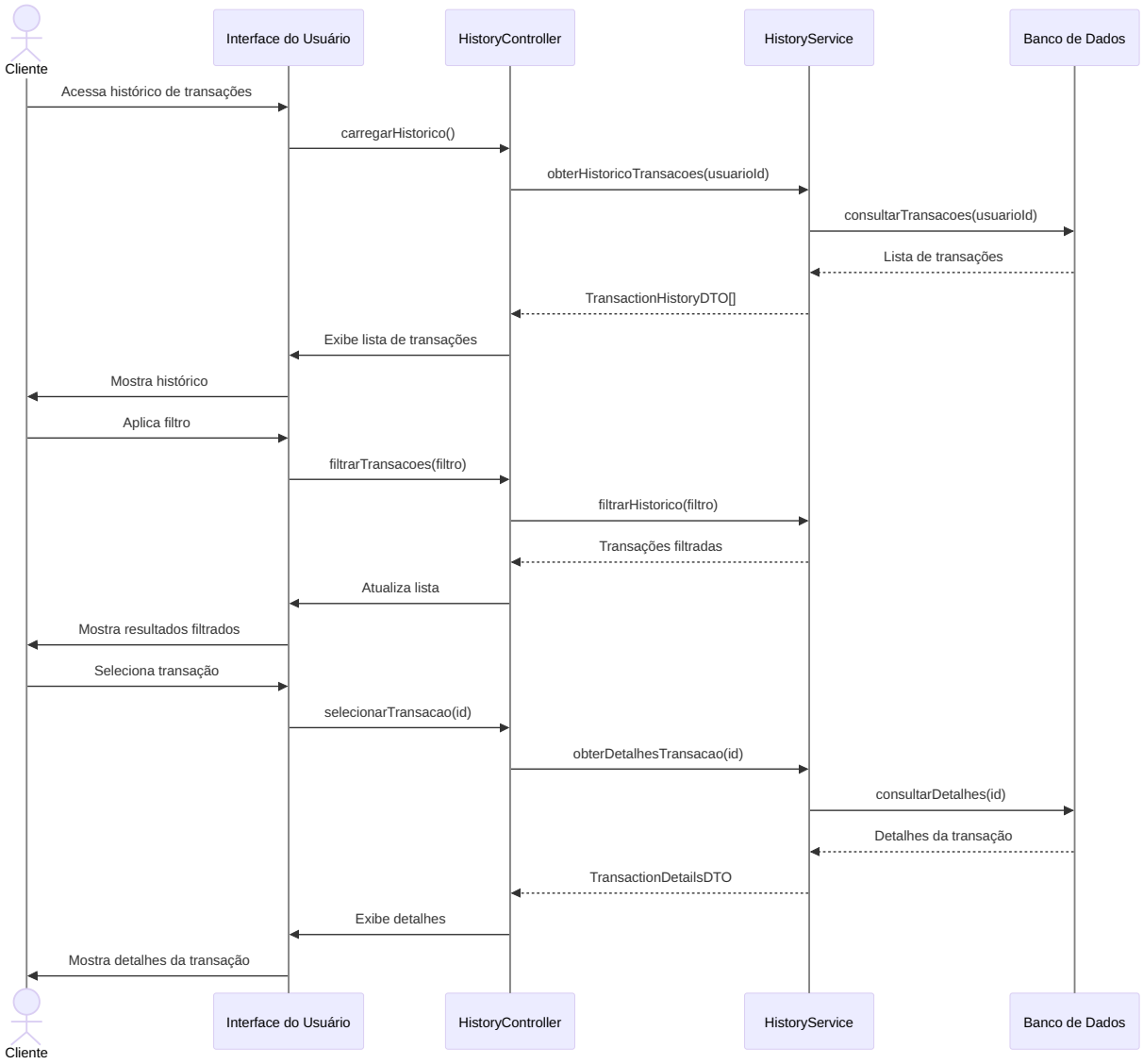
Este diagrama mostra a sequência de interações quando um cliente realiza uma

transação usando NFC.



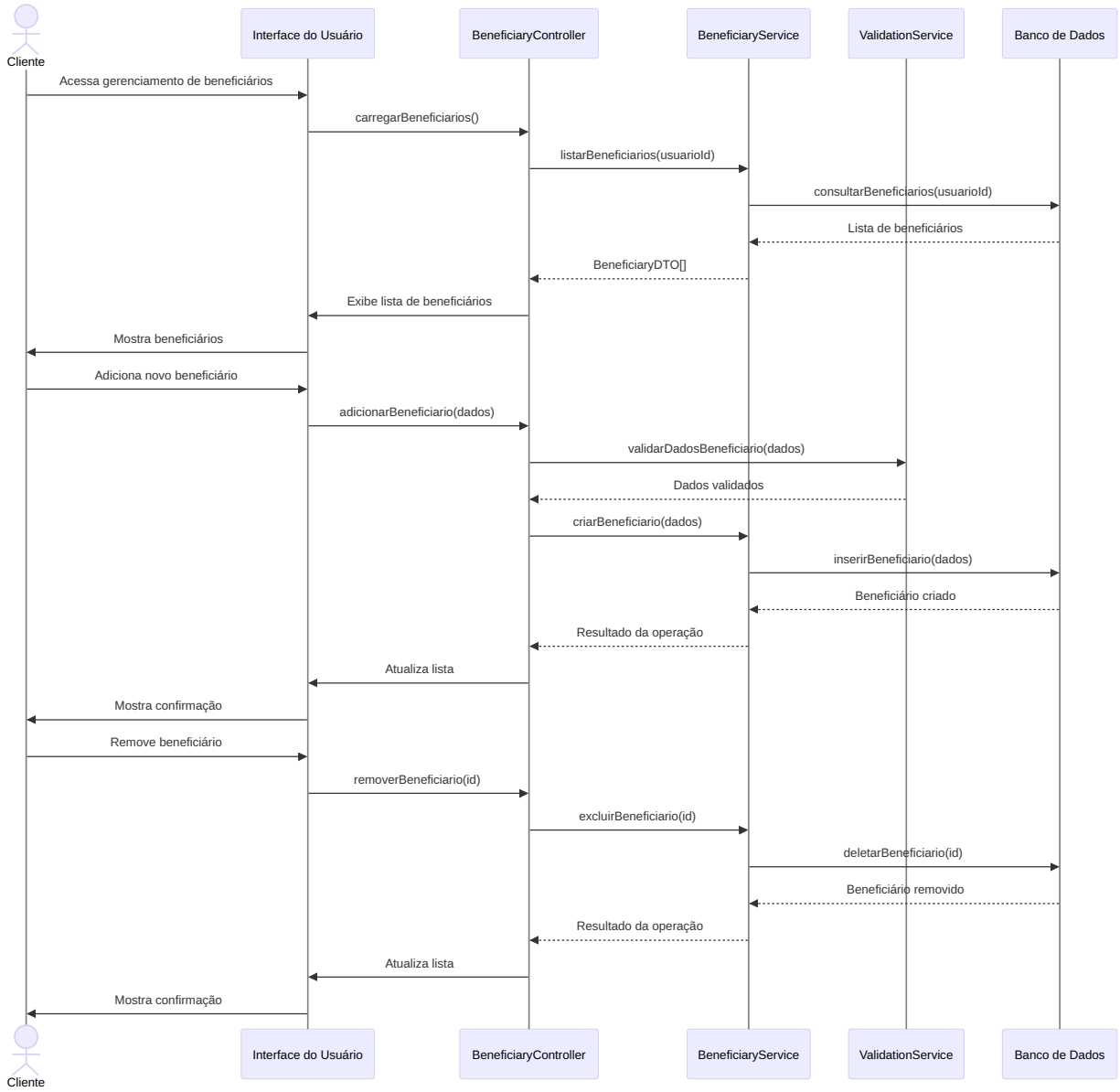
3. Visualizar Histórico de Transações

Este diagrama mostra a sequência de interações quando um cliente visualiza seu histórico de transações.



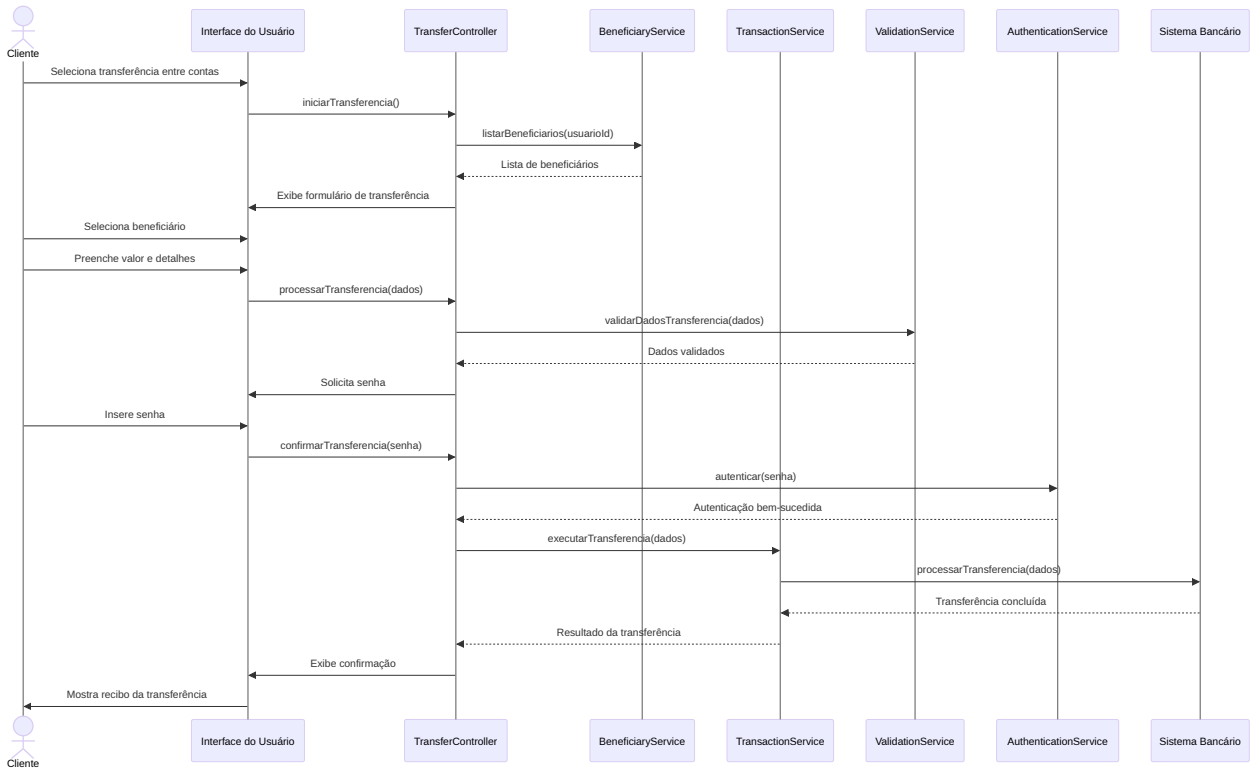
4. Gerenciar Beneficiários

Este diagrama mostra a sequência de interações quando um cliente gerencia seus beneficiários.



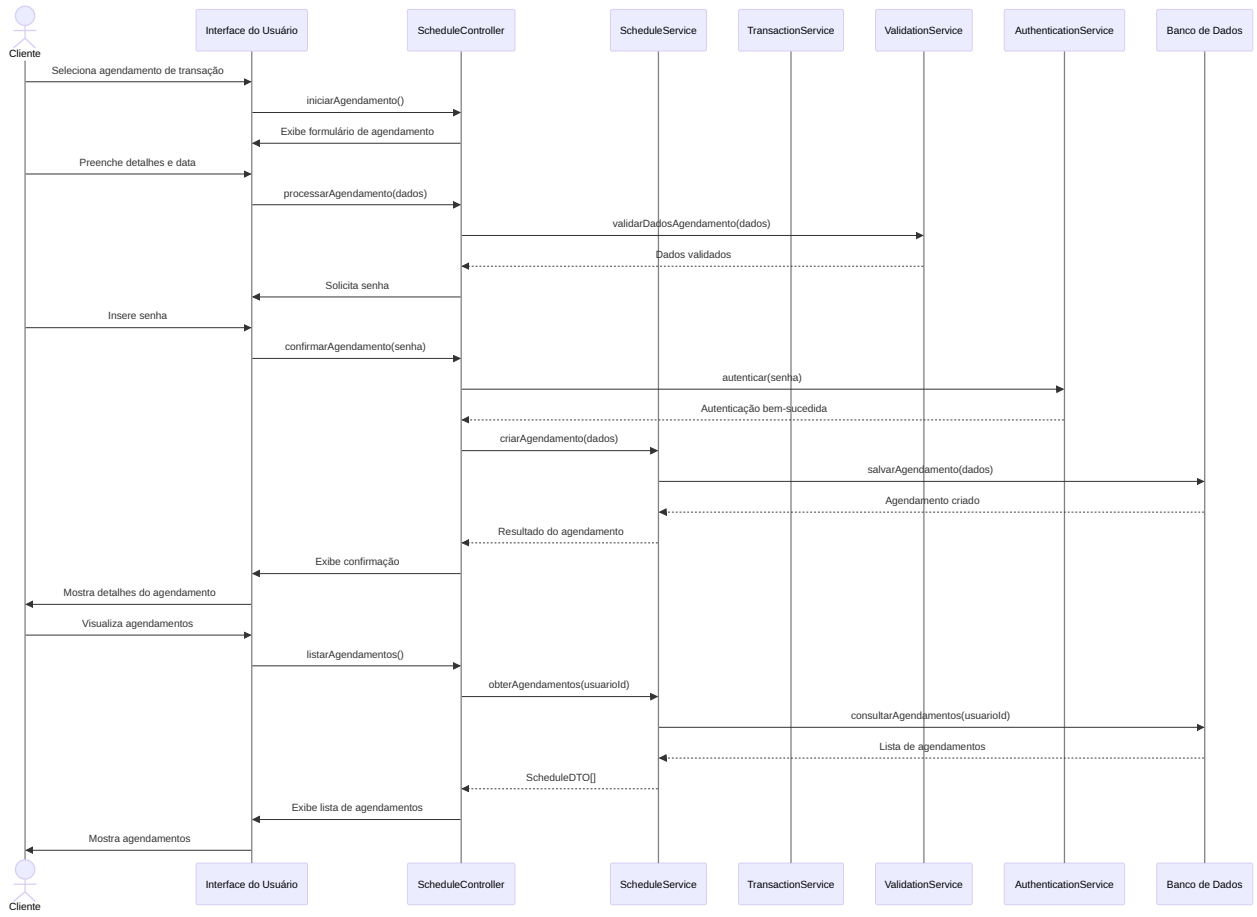
5. Realizar Transações entre Contas

Este diagrama mostra a sequência de interações quando um cliente realiza uma transferência entre contas.



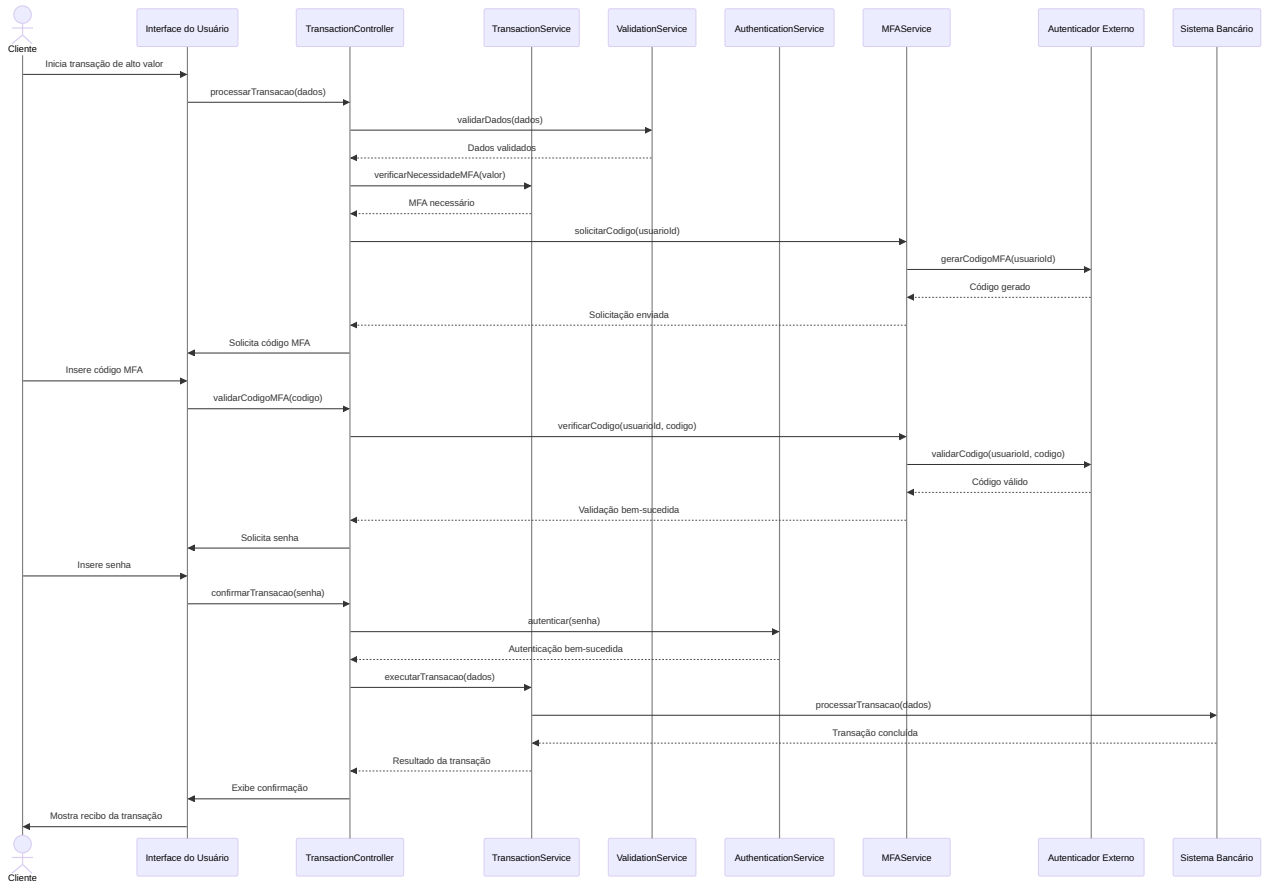
6. Agendar Transação

Este diagrama mostra a sequência de interações quando um cliente agenda uma transação.



7. Realizar Transação com MFA

Este diagrama mostra a sequência de interações quando um cliente realiza uma transação com autenticação multifatorial.



Padrões Comuns

Nos diagramas de sequência acima, podemos observar alguns padrões comuns:

1. **Validação de Dados**: Antes de processar qualquer operação, os dados são validados
2. **Autenticação**: Operações sensíveis exigem autenticação do usuário
3. **Padrão MVC/MVVM**: Separação clara entre interface do usuário, controladores e serviços
4. **Confirmação de Operações**: Após cada operação, o usuário recebe uma confirmação

Estes padrões garantem a segurança, consistência e usabilidade do sistema.

Interface do Usuário

A interface do usuário foi desenvolvida usando JavaFX, seguindo um design moderno e responsivo. O sistema utiliza FXML para separação entre layout e lógica, e controllers para gerenciar as interações.

Estrutura de Arquivos

```
src/main/
├── resources/
│   ├── fxml/
│   │   ├── login-view.fxml
│   │   ├── signup-view.fxml
│   │   ├── dashboard-view.fxml
│   │   ├── transfer-view.fxml
│   │   ├── history-view.fxml
│   │   ├── transaction-details-screen.fxml
│   │   └── beneficiary-view.fxml
│   └── css/
│       └── styles.css
└── java/org/jala/university/presentation/
    ├── controller/
    │   ├── LoginController.java
    │   ├── SignUpController.java
    │   ├── DashboardController.java
    │   ├── TransferController.java
    │   ├── TransactionHistoryController.java
    │   ├── TransactionHistoryDetailsController.java
    │   └── BeneficiaryController.java
    └── util/
        ├── ViewSwitcher.java
        └── SessionManager.java
```

Padrões de Design

MVC (Model-View-Controller)

- **Model**: Entidades e DTOs que representam os dados
- **View**: Arquivos FXML que definem a interface
- **Controller**: Classes Java que controlam a interação entre Model e View

FXML

- Layouts declarativos em XML
- Separação clara entre design e lógica
- Facilidade de manutenção e atualização

CSS

- Estilização separada da estrutura
- Temas consistentes em toda a aplicação
- Personalização de componentes

Principais Telas

Tela de Login

Interface de autenticação com campos para email e senha, além de opção para cadastro de novos usuários.

Dashboard

Tela principal que exibe saldo, transações recentes e acesso rápido às principais funcionalidades.

Transferências

Interface para realização de transferências entre contas, com validação de dados e confirmação.

Histórico de Transações

Visualização detalhada do histórico de transações com filtros e opções de busca.

Detalhes da Transação

Tela que apresenta informações detalhadas sobre uma transação específica.

Gerenciamento de Beneficiários

Interface para adicionar, editar e remover beneficiários para transferências.

Navegação entre Telas

O sistema utiliza o padrão ViewSwitcher para gerenciar a navegação entre telas, permitindo:

- Transição suave entre diferentes interfaces
- Manutenção do estado da aplicação durante a navegação
- Gerenciamento centralizado de rotas

Responsividade

A interface foi projetada para se adaptar a diferentes tamanhos de tela:

1. **Layout Fluido**
   - Uso de containers que se ajustam ao tamanho da janela
   - Elementos que redimensionam proporcionalmente
2. **Media Queries em CSS**
   - Estilos específicos para diferentes tamanhos de tela
   - Ajustes de fonte e espaçamento
3. **Componentes Adaptáveis**
   - Tabelas com colunas que se ajustam
   - Formulários que reorganizam campos em telas menores

Acessibilidade

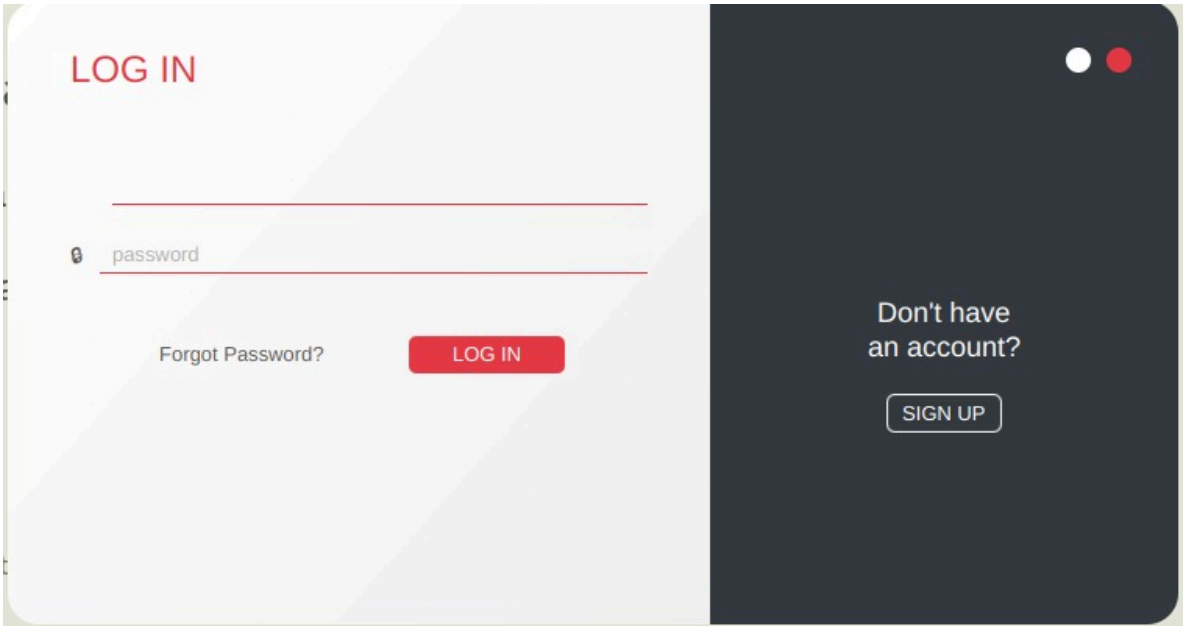
O sistema implementa as seguintes práticas de acessibilidade:

1. **Navegação por Teclado**
   - Todos os elementos podem ser acessados via teclado
   - Ordem de tabulação lógica
2. **Textos Alternativos**
   - Descrições para elementos visuais
   - Mensagens de erro claras e específicas
3. **Contraste Adequado**
   - Cores com contraste suficiente para leitura
   - Opção de tema de alto contraste

Tela de Login

Visão Geral

A tela de login é o ponto de entrada principal do sistema, oferecendo uma interface minimalista e intuitiva para autenticação de usuários. Projetada com foco na experiência do usuário, esta tela combina simplicidade visual com funcionalidades robustas de segurança.



Tela de Login

Funcionalidades

1. **Validação de Campos**
    - Verificação de campos obrigatórios
    - Validação de formato de email
    - Feedback visual para campos inválidos
2. **Autenticação**
    - Verificação de credenciais contra o backend

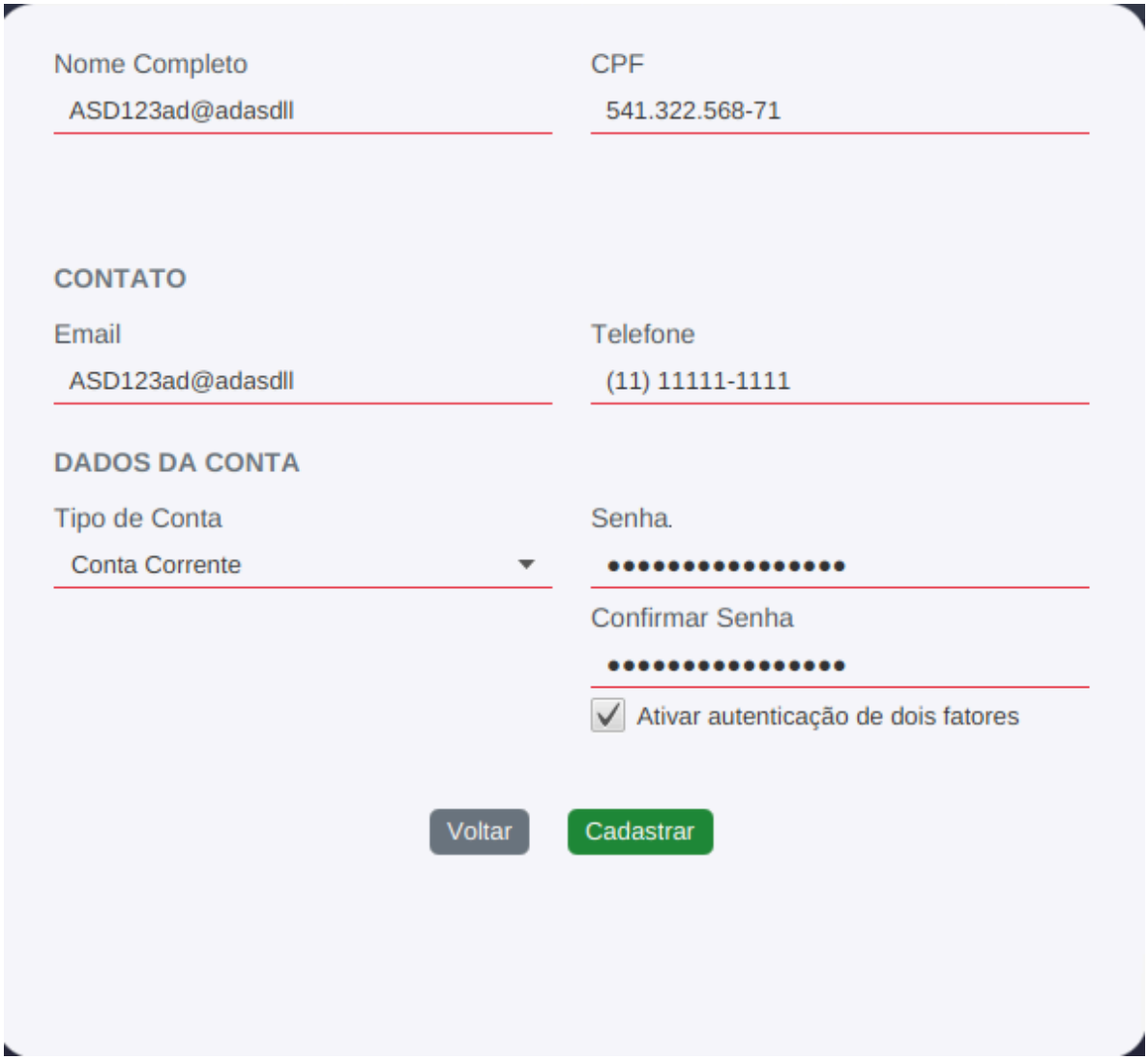
- Tratamento de erros de autenticação
- Suporte a autenticação de dois fatores

3. **Persistência de Sessão**
   - Opção "Lembrar-me" para salvar email
   - Gerenciamento de token de autenticação
   - Redirecionamento baseado no estado da autenticação
4. **Navegação**
   - Redirecionamento para cadastro
   - Acesso à recuperação de senha
   - Transição para dashboard após login bem-sucedido

Tela de Cadastro

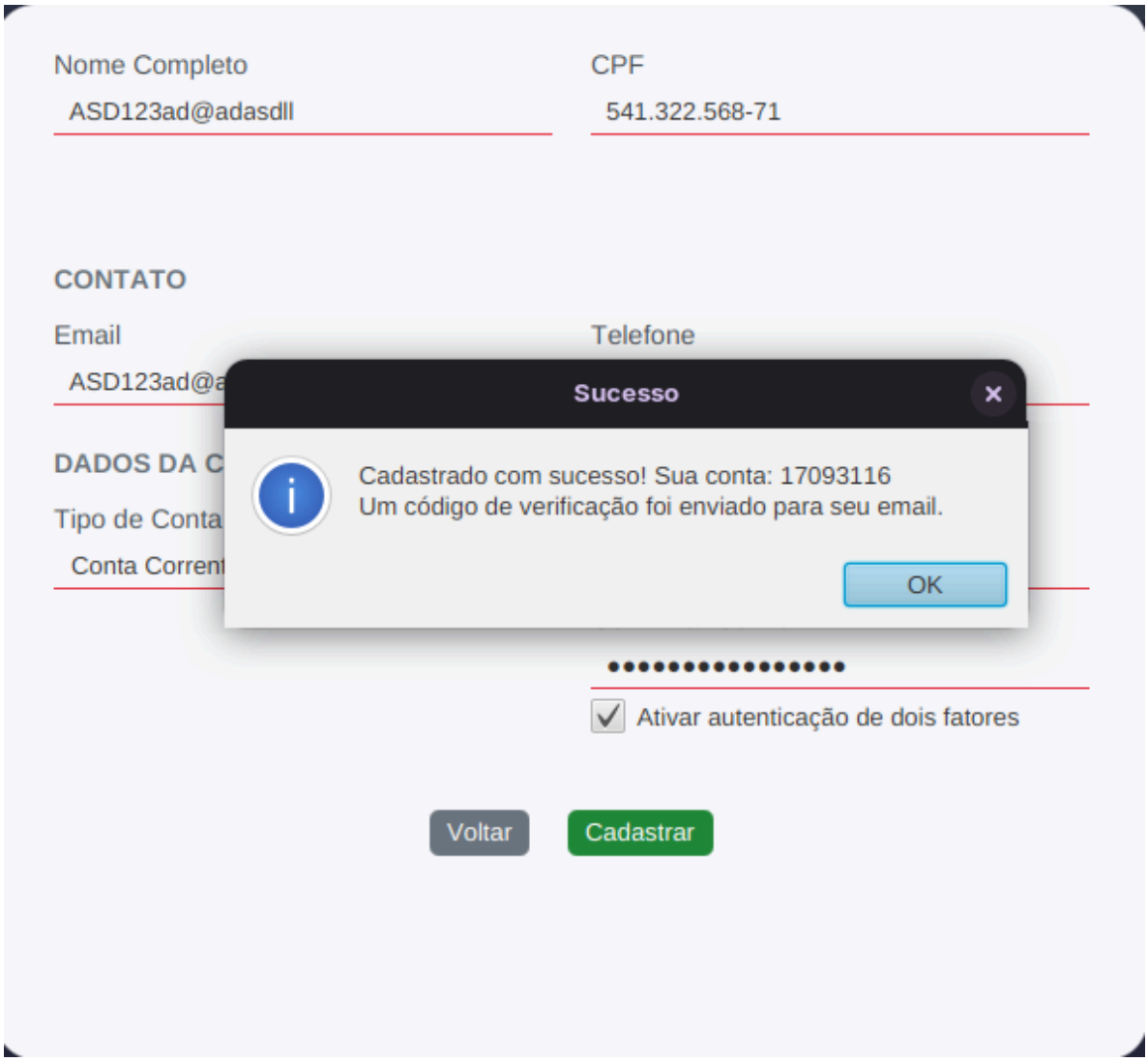
Layout

- Tela de cadastro:



Cadastro

- Mensagem depois de se cadastrar e os dados serem válidos:

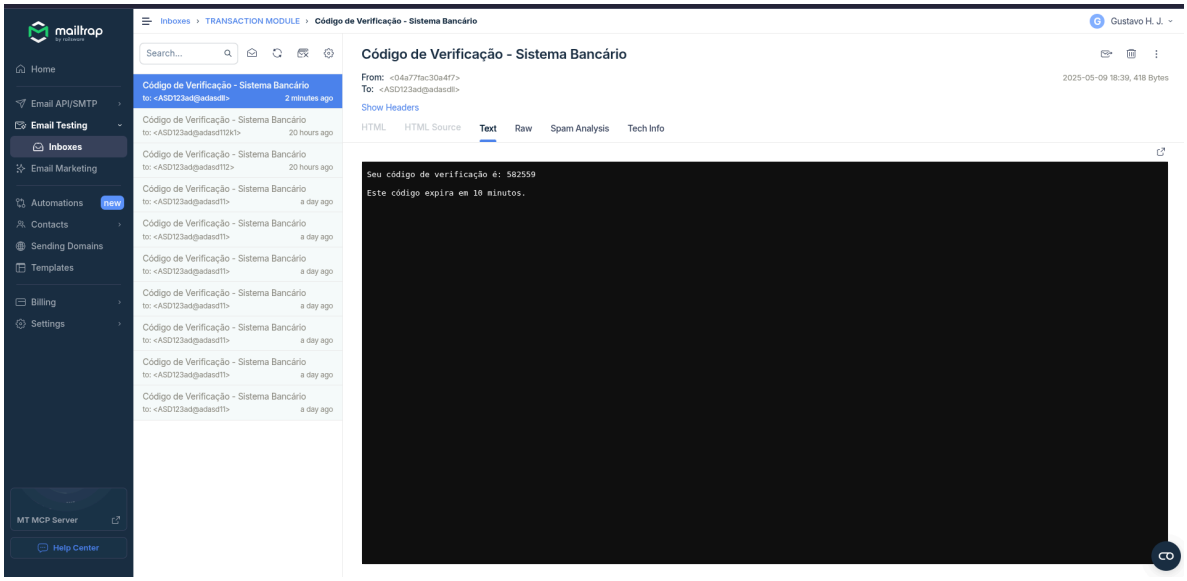


Mensagem depois de se cadastrar

- Tela de Autenticação:

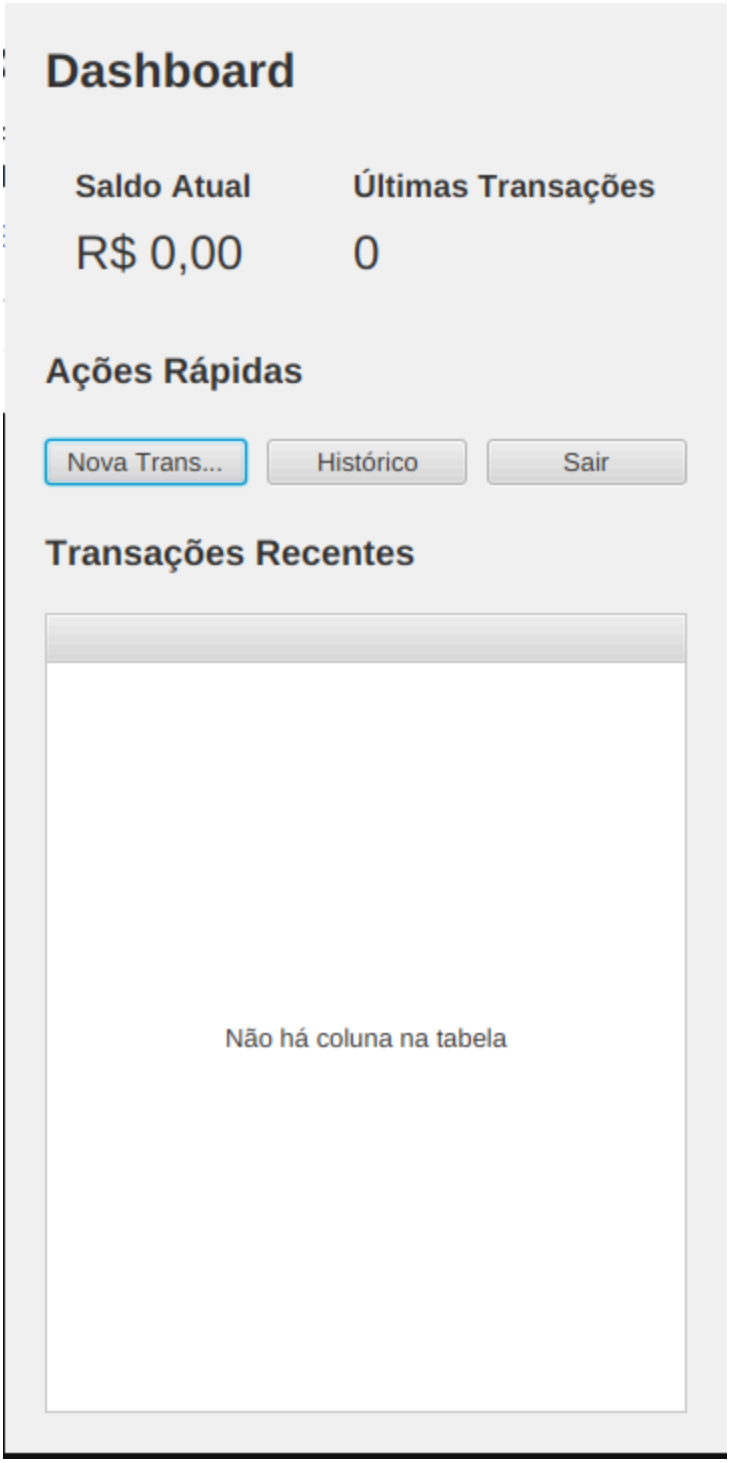
Autenticação

- Tela do serviço de email:



Tela do serviço de email

- Ao ser autenticado ele é levado para a dashboard do usuário:



Dashboard

Componentes FXML

```
<!-- Campos do formulário -->
<TextField fx:id="fullNameField" promptText="Digite seu nome
```

```
completo"/>
<TextField fx:id="cpfField" promptText="000.000.000-00"/>
<TextField fx:id="emailField" promptText="seu@email.com"/>
<TextField fx:id="phoneField" promptText="(00) 00000-0000"/>
<ComboBox fx:id="accountTypeCombo"/>
<PasswordField fx:id="passwordField" promptText="********"/>
```

Controller

```
public class SignUpController {
    @FXML private TextField fullNameField;
    @FXML private TextField cpfField;
    @FXML private TextField emailField;
    @FXML private TextField phoneField;
    @FXML private ComboBox<String> accountTypeCombo;

    @FXML
    private void initialize() {
        // Inicialização do ComboBox
        accountTypeCombo.getItems().addAll(
            "Conta Corrente",
            "Conta Poupança",
            "Conta Salário"
        );

        // Máscaras para campos
        TextFieldFormatter.applyMask(cpfField, "###.###.###-##");
        TextFieldFormatter.applyMask(phoneField, "(##) #####-
####");
    }

    @FXML
    private void handleRegister() {
        // Lógica de registro
```

```
    }
}
```

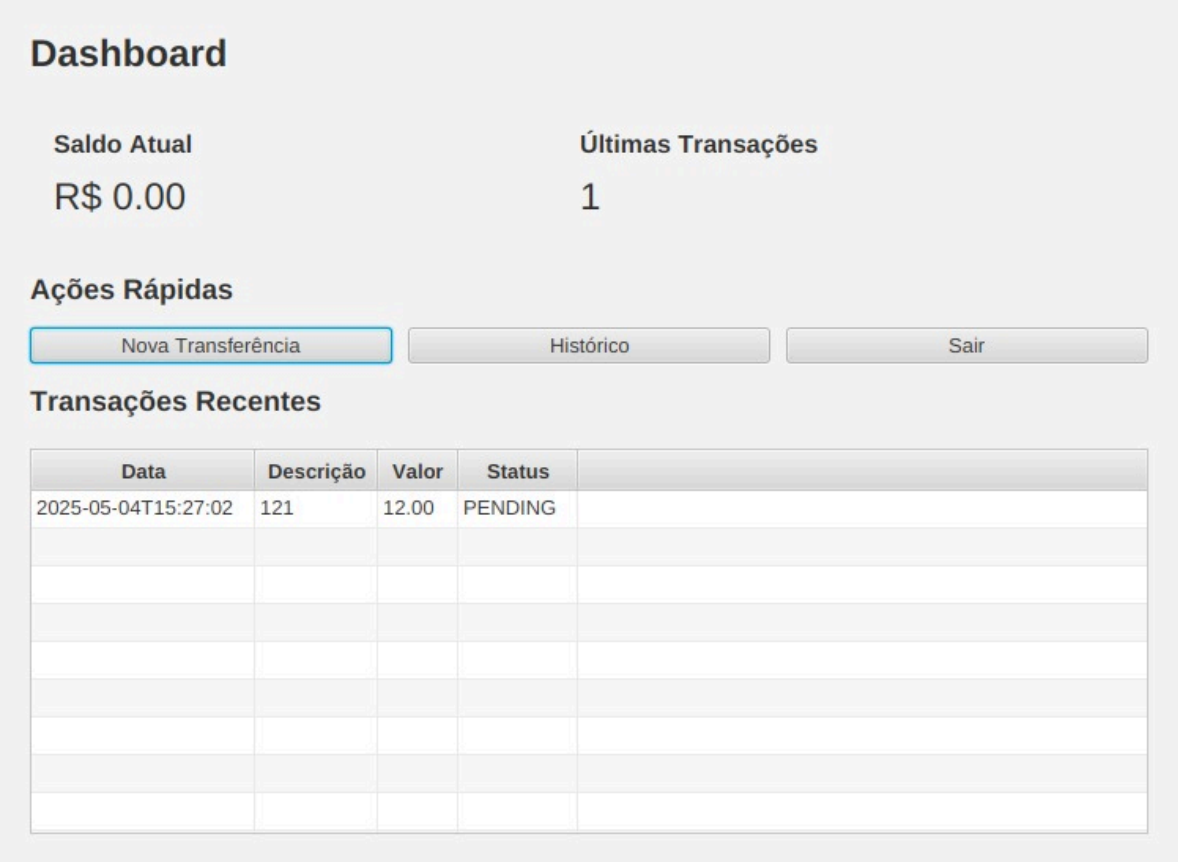
Funcionalidades

1. **Validação de Dados**
   - Máscaras para CPF e telefone
   - Validação de campos obrigatórios
   - Verificação de formato de email
2. **Tipos de Conta**
   - Seleção via ComboBox
   - Opções pré-definidas
3. **Navegação**
   - Botão "Voltar" para tela de login
   - Botão "Cadastrar" para finalizar registro

Tela de Dashboard

Visão Geral

A tela de dashboard é o centro de controle principal do usuário após o login, apresentando uma visão consolidada das informações financeiras e oferecendo acesso rápido às principais funcionalidades do sistema. Esta interface foi projetada para fornecer uma experiência intuitiva e informativa, permitindo que o usuário visualize seu status financeiro e realize ações comuns com facilidade.



Tela de Cadastro

Componentes da Interface

1. **Cabeçalho**

- Saudação personalizada ao usuário

- Botões de acesso rápido a configurações e logout
- Indicador de notificações

2. **Card de Saldo**
   - Exibição destacada do saldo disponível
   - Botão para visualização detalhada do extrato
   - Indicador visual de variação do saldo
3. **Transações Recentes**
   - Lista das últimas transações realizadas
   - Indicadores visuais para diferentes tipos de transação
   - Acesso direto aos detalhes de cada transação
4. **Menu de Ações Rápidas**
   - Botões para nova transferência, histórico e gerenciamento de beneficiários
   - Acesso direto às funcionalidades mais utilizadas
   - Organização visual intuitiva

Funcionalidades

1. **Visualização de Dados Financeiros**
   - Saldo atual da conta
   - Histórico recente de transações
   - Indicadores de entradas e saídas
2. **Navegação do Sistema**
   - Acesso às principais funcionalidades
   - Transição para telas específicas

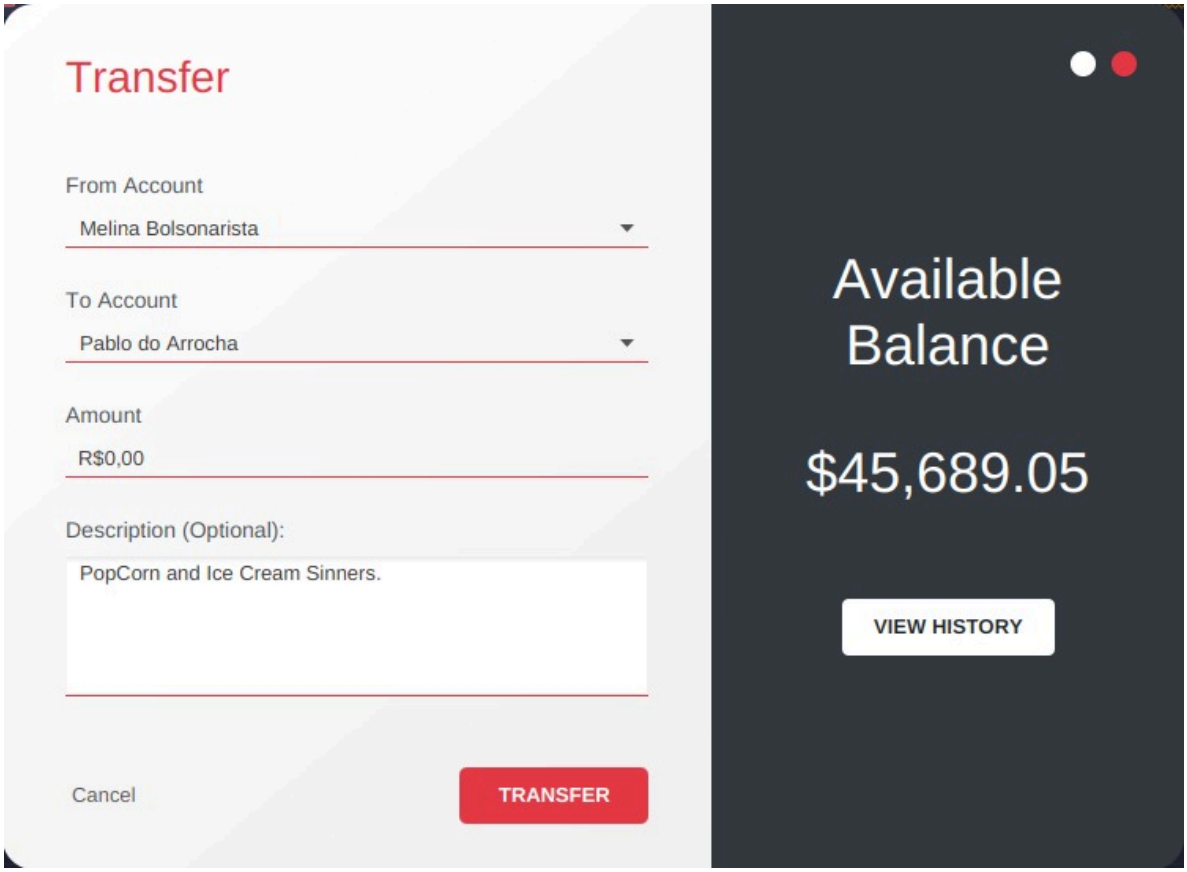
- Logout seguro do sistema

3. **Interação com Transações**
    - Visualização rápida de transações recentes
    - Acesso aos detalhes de cada transação
    - Filtros básicos para visualização

Tela de Transferência

Visão Geral

A tela de transferência é uma interface especializada para a realização de transferências financeiras entre contas. Projetada com foco na segurança e usabilidade, esta tela guia o usuário através do processo de transferência de forma clara e intuitiva, garantindo que todas as informações necessárias sejam fornecidas corretamente antes da confirmação da operação.



Tela de transfer

Componentes da Interface

1. **Formulário de Transferência**

- Seleção de conta de destino

- Campo para valor da transferência
- Campo opcional para descrição/comentário
- Botões de ação (Transferir e Cancelar)

2. **Painel de Informações**
    - Exibição do saldo disponível
    - Histórico resumido de transferências recentes
    - Acesso rápido ao histórico completo
3. **Área de Confirmação**
    - Resumo dos dados da transferência
    - Opções de confirmação ou cancelamento
    - Requisitos adicionais de segurança para valores elevados

Funcionalidades

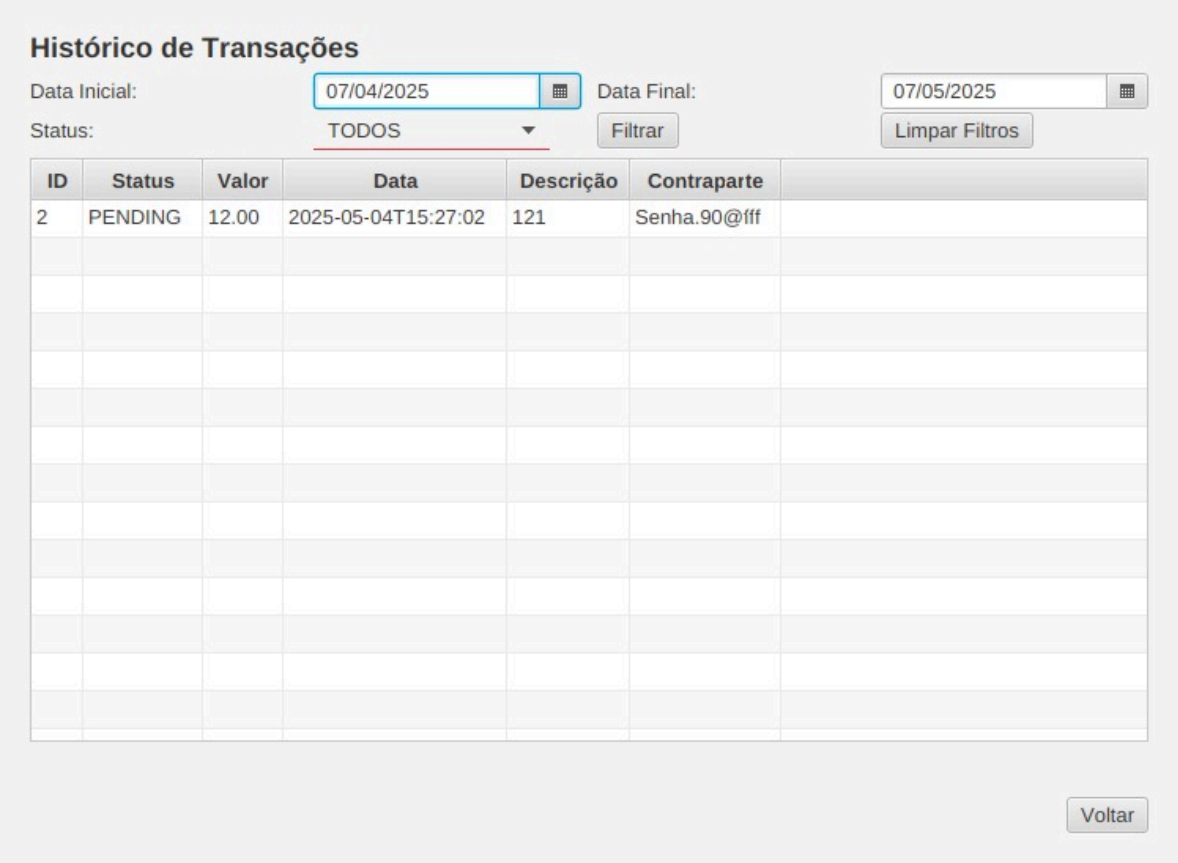
1. **Seleção de Destinatário**
    - Lista de beneficiários cadastrados
    - Opção para adicionar novo beneficiário
    - Busca rápida por nome ou número de conta
2. **Definição de Valor**
    - Entrada de valor com validação de limites
    - Verificação automática contra o saldo disponível
    - Formatação monetária em tempo real
3. **Confirmação e Processamento**
    - Verificação de segurança adicional para valores elevados

- Processamento da transação com feedback visual
- Geração de comprovante após conclusão

Tela de Histórico de Transações

Visão Geral

A tela de histórico de transações oferece uma visão abrangente e detalhada de todas as movimentações financeiras realizadas na conta do usuário. Esta interface foi projetada para permitir a fácil visualização, filtragem e análise das transações, auxiliando o usuário no controle e acompanhamento de suas finanças.



Tela de History Transfer

Componentes da Interface

1. **Barra de Filtros**

- Seleção de período (data inicial e final)
- Filtros por tipo de transação (entrada, saída)

- Filtros por status (concluída, pendente, cancelada)
- Campo de busca textual

2. **Tabela de Transações**
   - Colunas para data, descrição, valor e status
   - Indicadores visuais para diferentes tipos de transação
   - Paginação para navegação entre resultados
   - Ordenação por coluna
3. **Painel de Resumo**
   - Total de transações no período selecionado
   - Somatório de entradas e saídas
   - Balanço do período

Funcionalidades

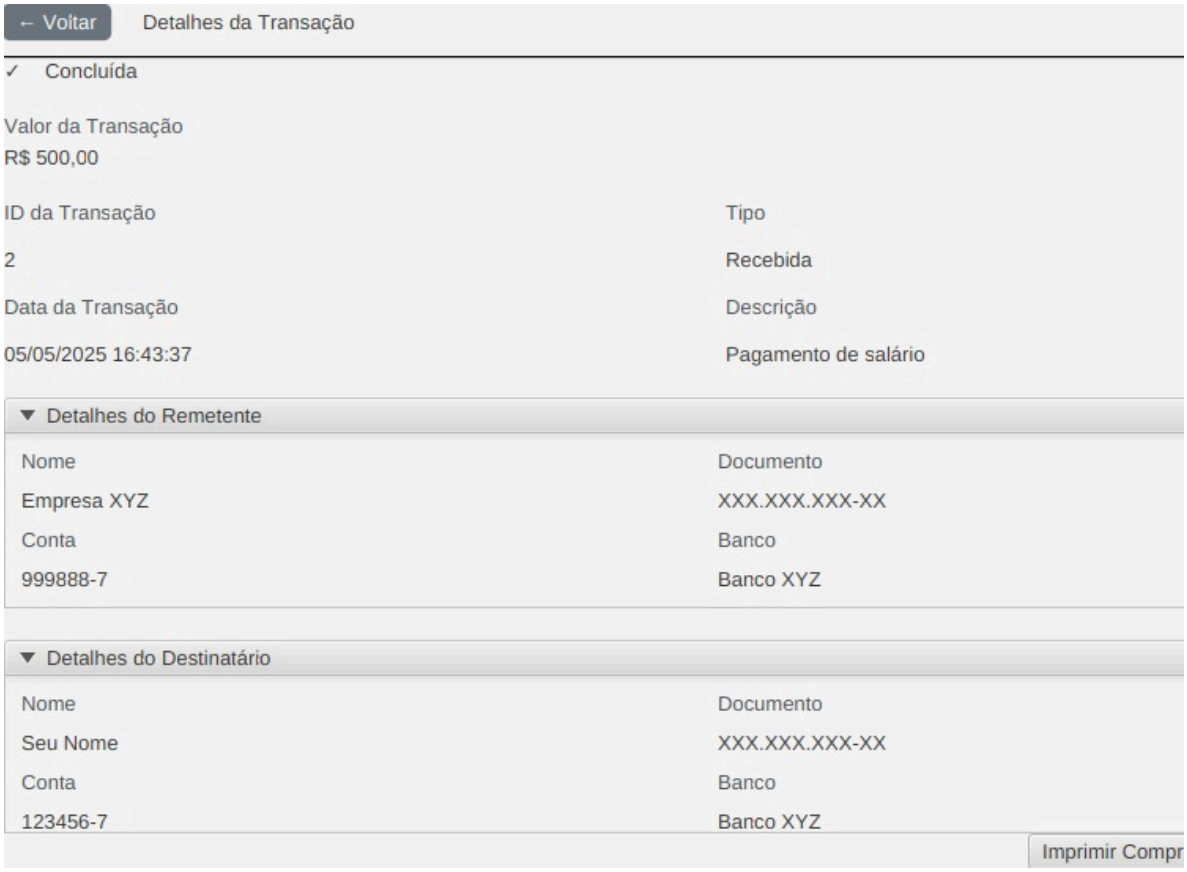
1. **Visualização de Transações**
   - Listagem cronológica de todas as transações
   - Detalhamento ao clicar em uma transação específica
   - Indicadores visuais para diferentes tipos e status
2. **Filtragem e Busca**
   - Aplicação de múltiplos filtros simultaneamente
   - Busca por texto em descrições e nomes
   - Filtros por período personalizado
3. **Análise de Dados**
   - Visualização de totais e resumos

- Identificação de padrões de gastos
- Acompanhamento de transações recorrentes

Tela de Detalhes da Transação

Visão Geral

A tela de detalhes da transação apresenta informações completas e detalhadas sobre uma transação específica selecionada pelo usuário. Esta interface foi projetada para fornecer transparência total sobre cada operação financeira, incluindo todos os dados relevantes, status atual e histórico de processamento.



Tela de Details History Transfer

Componentes da Interface

1. **Cabeçalho Informativo**
    - Tipo e status da transação com indicador visual
    - Data e hora da operação

- Identificador único da transação

2. **Detalhes Financeiros**
    - Valor da transação em destaque
    - Taxas aplicadas (quando existentes)
    - Valor líquido processado
3. **Informações das Partes**
    - Dados do remetente (conta de origem)
    - Dados do destinatário (conta de destino)
    - Instituições financeiras envolvidas
4. **Informações Adicionais**
    - Descrição/comentário da transação
    - Categoria (quando aplicável)
    - Referências externas
5. **Ações Disponíveis**
    - Impressão de comprovante
    - Compartilhamento via email
    - Contestação (quando aplicável)

Funcionalidades

1. **Visualização de Informações**
    - Exibição detalhada de todos os aspectos da transação
    - Formatação adequada para diferentes tipos de dados
    - Destaque para informações críticas

2. **Geração de Comprovantes**

   - Criação de comprovante oficial em formato PDF
   - Opções para impressão direta
   - Compartilhamento via email ou outros meios

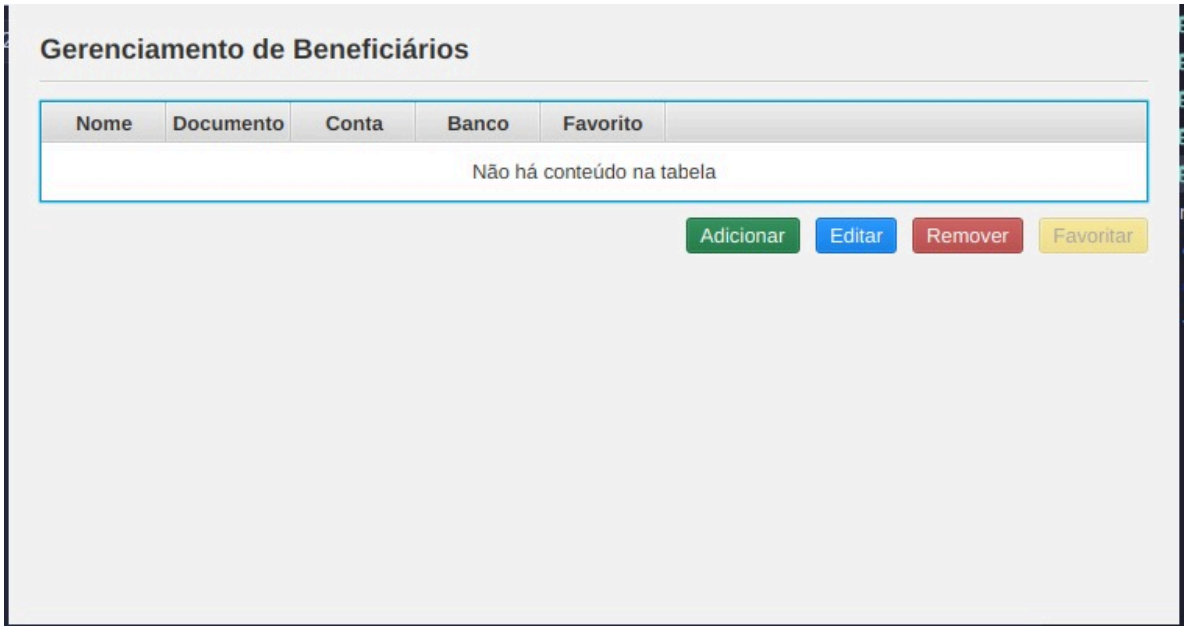
3. **Ações Relacionadas**

   - Repetição da transação com os mesmos dados
   - Contestação em caso de problemas
   - Navegação para transações relacionadas

Tela de Gerenciamento de Beneficiários

Visão Geral

A tela de gerenciamento de beneficiários permite ao usuário visualizar, adicionar, editar e remover destinatários para transferências financeiras. Esta interface centraliza todas as operações relacionadas aos beneficiários, simplificando o processo de transferências futuras ao manter um catálogo organizado de destinatários frequentes.



Tela de Gerenciamento de Beneficiarios

Características Principais

- **Listagem Organizada**: Visualização clara de todos os beneficiários cadastrados
- **Adição Simplificada**: Processo intuitivo para cadastro de novos beneficiários
- **Edição Rápida**: Atualização fácil de informações de contatos existentes
- **Favoritos**: Marcação de beneficiários frequentes para acesso prioritário
- **Categorização**: Organização por tipo de beneficiário ou relacionamento

- **Busca Eficiente**: Localização rápida por nome, banco ou número de conta

Banco de Dados

O sistema utiliza MySQL como banco de dados principal, com JPA/Hibernate para mapeamento objeto-relacional.

Configuração

```
<persistence-unit name="transaction-pu" transaction-
type="RESOURCE_LOCAL">
    <properties>
        <property name="jakarta.persistence.jdbc.url"

value="jdbc:mysql://localhost:3306/transaction_db"/>
        <property name="hibernate.dialect"
                  value="org.hibernate.dialect.MySQLDialect"/>
        <property name="hibernate.show_sql" value="true"/>
        <property name="hibernate.format_sql" value="true"/>
    </properties>
</persistence-unit>
```

Estrutura Principal

```
transaction_db
├── users
├── accounts
├── transactions
└── account_beneficiaries
```

Boas Práticas

1. **Nomenclatura**
   - Tabelas em plural (users, accounts)
   - Colunas em snake_case

- Chaves estrangeiras com sufixo _id

2. **Integridade**
   - Constraints de chave estrangeira
   - Índices para performance
   - Unicidade onde necessário
3. **Auditoria**
   - Campos created_at e updated_at
   - Tracking de modificações

Esquema do Banco de Dados

Tabelas Principais

Users

```
CREATE TABLE users (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    full_name VARCHAR(150) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    password VARCHAR(250) NOT NULL,
    cpf VARCHAR(13) NOT NULL UNIQUE,
    phone_number VARCHAR(15) NOT NULL,
    roles VARCHAR(20) NOT NULL,
    created_at DATETIME,
    updated_at DATETIME
);
```

Accounts

```
CREATE TABLE accounts (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    user_id BIGINT NOT NULL,
    agency VARCHAR(20) NOT NULL,
    account_number VARCHAR(20) NOT NULL,
    account_type VARCHAR(20) NOT NULL,
    balance DECIMAL(10,2) DEFAULT 0.00,
    created_at DATETIME,
    updated_at DATETIME,
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Transactions

```
CREATE TABLE transactions (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
```

```
    sender_id BIGINT NOT NULL,
    receiver_id BIGINT NOT NULL,
    amount DECIMAL(10,2) NOT NULL,
    status VARCHAR(20) NOT NULL,
    description VARCHAR(255) NOT NULL,
    created_at DATETIME,
    FOREIGN KEY (sender_id) REFERENCES accounts(id),
    FOREIGN KEY (receiver_id) REFERENCES accounts(id)
);
```

Account Beneficiaries

```
CREATE TABLE account_beneficiaries (
    account_id BIGINT NOT NULL,
    beneficiary_id BIGINT NOT NULL,
    favorite BIT(1),
    PRIMARY KEY (account_id, beneficiary_id),
    FOREIGN KEY (account_id) REFERENCES accounts(id),
    FOREIGN KEY (beneficiary_id) REFERENCES accounts(id)
);
```

Índices

```
-- Users
CREATE INDEX idx_users_email ON users(email);
CREATE INDEX idx_users_cpf ON users(cpf);

-- Accounts
CREATE INDEX idx_accounts_user ON accounts(user_id);
CREATE INDEX idx_accounts_number ON accounts(account_number);

-- Transactions
CREATE INDEX idx_transactions_sender ON transactions(sender_id);
CREATE INDEX idx_transactions_receiver ON
```

```
transactions(receiver_id);
CREATE INDEX idx_transactions_created ON transactions(created_at);
```

Migrações de Banco de Dados

Estratégia de Migração

O sistema utiliza a estratégia de migração automática do Hibernate com `hibernate.hbm2ddl.auto=update` em desenvolvimento.

Configuração

```
<property name="hibernate.hbm2ddl.auto" value="update"/>
```

Scripts de Migração

V1 - Estrutura Inicial

```
-- Criação das tabelas principais
CREATE TABLE users ( ... );
CREATE TABLE accounts ( ... );
CREATE TABLE transactions ( ... );
CREATE TABLE account_beneficiaries ( ... );

-- Índices iniciais
CREATE INDEX idx_users_email ON users(email);
CREATE INDEX idx_users_cpf ON users(cpf);
```

Boas Práticas

1. **Versionamento**
   - Scripts incrementais
   - Nomenclatura clara (V1, V2, etc.)
   - Documentação de mudanças
2. **Segurança**

- Backup antes de migrações
- Scripts idempotentes
- Rollback planejado

3. **Ambiente**
    - Teste em desenvolvimento
    - Validação em staging
    - Execução em produção

Configuração do Projeto

Este guia detalha os passos necessários para configurar o ambiente de desenvolvimento do módulo de transações.

Requisitos

- Java Development Kit (JDK) 17 ou superior
- Maven 3.8+
- Docker e Docker Compose
- IDE compatível com JavaFX (recomendado: IntelliJ IDEA)

Estrutura do Projeto

```
./
├── docker/
│   └── docker-compose.yml
├── src/
│   └── main/
│       ├── java/
│       └── resources/
├── pom.xml
└── .env
```

Primeiros Passos

1. Clone o repositório
2. Configure as variáveis de ambiente
3. Inicie o banco de dados com Docker

4. Execute o build com Maven

Configuração do Docker

Docker Compose

O arquivo `docker-compose.yml` define os serviços necessários para o ambiente de desenvolvimento.

```yaml
version: '3.8'

services:
  mysql:
    image: mysql:8.0
    container_name: transaction_mysql
    ports:
      - "3306:3306"
    environment:
      MYSQL_DATABASE: transaction_db
      MYSQL_USER: transaction_user
      MYSQL_PASSWORD: password123
      MYSQL_ROOT_PASSWORD: root
    volumes:
      - mysql_data:/var/lib/mysql

volumes:
  mysql_data:
```

Comandos Principais

```
# Iniciar os serviços
docker-compose up -d

# Verificar status
docker-compose ps

# Parar os serviços
docker-compose down
```

```
# Logs
docker-compose logs -f
```

Verificação

1. Banco MySQL disponível na porta 3306
2. Credenciais configuradas corretamente
3. Volume persistente criado

Configuração do Maven

POM.xml

Principais dependências e plugins configurados no `pom.xml`:

```
<dependencies>
    <!-- Local Dependency -->
    <dependency>
        <groupId>org.jala.university</groupId>
        <artifactId>commons-module</artifactId>
        <version>${commons.module.version}</version>
    </dependency>

    <!-- Spring Boot -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
        <version>${spring.boot.version}</version>
    </dependency>

    <!-- Jakarta Persistence -->
    <dependency>
        <groupId>jakarta.persistence</groupId>
        <artifactId>jakarta.persistence-api</artifactId>
        <version>${jakarta.persistence.version}</version>
    </dependency>

    <!-- Lombok -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>${lombok.version}</version>
        <optional>true</optional>
        <scope>provided</scope>
    </dependency>
```

```
    <!-- JavaFX -->
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-controls</artifactId>
        <version>${javafx.version}</version>
    </dependency>
    <dependency>
        <groupId>org.openjfx</groupId>
        <artifactId>javafx-fxml</artifactId>
        <version>${javafx.version}</version>
    </dependency>

    <!-- Testing -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.10.0-M1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>5.5.0</version>
        <scope>test</scope>
    </dependency>

    <!-- Database -->
    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <version>9.1.0</version>
        <scope>runtime</scope>
    </dependency>

    <!-- Security -->
    <dependency>
        <groupId>com.auth0</groupId>
```

```
        <artifactId>java-jwt</artifactId>
        <version>4.5.0</version>
    </dependency>
</dependencies>
```

Versões das Dependências

```
<properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <commons.module.version>1.0-SNAPSHOT</commons.module.version>
    <spring.boot.version>3.2.4</spring.boot.version>

<jakarta.persistence.version>3.1.0</jakarta.persistence.version>
    <lombok.version>1.18.38</lombok.version>
    <javafx.version>22</javafx.version>

<javafx.maven.plugin.version>0.0.8</javafx.maven.plugin.version>
</properties>
```

Plugins

```
<build>
    <plugins>
        <!-- JavaFX -->
        <plugin>
            <groupId>org.openjfx</groupId>
            <artifactId>javafx-maven-plugin</artifactId>
            <version>${javafx.maven.plugin.version}</version>
            <configuration>

<mainClass>org.jala.university.presentation.LoginApplicationorg.ja
la.university.presentation.LoginApplication</mainClass>
            </configuration>
        </plugin>
```

```
        <!-- Lombok -->
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.11.0</version>
            <configuration>
                <annotationProcessorPaths>
                    <path>
                        <groupId>org.projectlombok</groupId>
                        <artifactId>lombok</artifactId>
                        <version>${lombok.version}</version>
                    </path>
                </annotationProcessorPaths>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Comandos Maven

```
# Limpar e compilar
mvn clean compile

# Executar testes
mvn test

# Executar aplicação
mvn javafx:run

# Gerar pacote
mvn package
```

Repositórios

Para dependências do GitLab, adicione:

```
<repositories>
    <repository>
        <id>gitlab-maven</id>

<url>https://gitlab.com/api/v4/projects/{project_code}/packages/ma
ven</url>
    </repository>
</repositories>
```

Boas Práticas

1. Mantenha as versões das dependências atualizadas
2. Use variáveis para versões no `properties`
3. Configure corretamente os escopos das dependências
4. Mantenha os plugins necessários para o build
5. Documente alterações significativas no `pom.xml`

Variáveis de Ambiente

Arquivo .env

```
# Database
DB_HOST=localhost
DB_PORT=3306
DB_NAME=transaction_db
DB_USER=transaction_user
DB_PASS=password123

# Application
APP_PORT=8080
APP_ENV=development
```

Configuração no IDE

IntelliJ IDEA

1. Edit Configurations

2. Add VM Options:

```
-Ddb.host=${DB_HOST}
-Ddb.port=${DB_PORT}
-Ddb.name=${DB_NAME}
-Ddb.user=${DB_USER}
-Ddb.pass=${DB_PASS}
```

Variáveis Necessárias

Variável	Descrição	Valor Padrão
DB_HOST	Host do banco de dados	localhost
DB_PORT	Porta do MySQL	3306
DB_NAME	Nome do banco	transaction_db
DB_USER	Usuário do banco	transaction_user
DB_PASS	Senha do banco	password123
APP_PORT	Porta da aplicação	8080
APP_ENV	Ambiente atual	development

Segurança

- Nunca commite o arquivo `.env`
- Use `.env.example` como template
- Mantenha senhas seguras em produção

Guias

Esta seção contém guias detalhados para desenvolvimento, testes e implantação do módulo de transações.

Visão Geral

- Guia de Desenvolvimento (Guia de Desenvolvimento): Práticas e padrões de código
- Guia de Testes (Guia de Testes): Estratégias e ferramentas de teste
- Guia de Implantação (Guia de Implantação): Processos de build e deploy

Fluxo de Trabalho

1. Desenvolvimento local seguindo o guia de desenvolvimento
2. Testes conforme guia de testes
3. Implantação seguindo guia de deploy

Guia de Desenvolvimento

Configuração do Ambiente

1. Instale as ferramentas necessárias:
   - JDK 17
   - Maven 3.8+
   - Docker
   - IDE (recomendado: IntelliJ IDEA)
2. Clone o repositório:

```
git clone <repository-url>
cd transaction-module
```

Estrutura do Projeto

```
src/
├── main/
│   ├── java/
│   │   └── org/jala/university/
│   │       ├── application/     # Casos de uso
│   │       ├── domain/         # Entidades e regras
│   │       ├── infrastructure/ # Implementações
│   │       └── presentation/   # UI e controllers
│   └── resources/
│       ├── META-INF/
│       │   └── persistence.xml
│       └── fxml/               # Layouts JavaFX
└── test/
    └── java/                  # Testes unitários
```

Padrões de Código

Nomenclatura

- Classes: PascalCase (ex: `TransactionService`)
- Métodos/Variáveis: camelCase (ex: `getUserBalance`)
- Constantes: SNAKE_CASE (ex: `MAX_AMOUNT`)

Arquitetura

- Seguir Clean Architecture
- Usar injeção de dependência
- Manter camadas desacopladas

Documentação

- Javadoc em classes públicas
- README atualizado
- Comentários claros quando necessário

Git Flow

1. Branch principal: `main`
2. Branch de desenvolvimento: `develop`
3. Branches de feature: `feature/*`
4. Branches de correção: `hotfix/*`

Commits

```
# Formato
<tipo>(<escopo>): <descrição>

# Exemplos
feat(transaction): adiciona validação de saldo
fix(auth): corrige verificação de token
```

Boas Práticas

1. **SOLID**
   - Single Responsibility
   - Open/Closed
   - Liskov Substitution
   - Interface Segregation
   - Dependency Inversion
2. **Clean Code**
   - Métodos pequenos
   - Nomes significativos
   - DRY (Don't Repeat Yourself)
   - KISS (Keep It Simple)
3. **Tratamento de Erros**
   - Usar exceções apropriadas
   - Logging adequado
   - Validações de entrada

Desenvolvimento Local

```
# Build
mvn clean install

# Executar
mvn javafx:run

# Testes
mvn test
```

Dicas

1. Use Lombok para reduzir boilerplate
2. Mantenha testes atualizados
3. Revise código antes do commit
4. Atualize documentação quando necessário

Guia de Testes

Estrutura de Testes

Tipos de Testes

1. **Testes Unitários**
   - JUnit 5
   - Mockito
   - Testes isolados
2. **Testes de Integração**
   - Banco de dados
   - APIs externas
   - Componentes integrados
3. **Testes de UI**
   - TestFX para JavaFX
   - Cenários de usuário

Configuração

```
<!-- Dependências de teste -->
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.10.0-M1</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
```

```
    <artifactId>mockito-core</artifactId>
    <version>5.5.0</version>
    <scope>test</scope>
</dependency>
```

Padrões de Teste

Nomenclatura

```
@Test
void deveRetornarSaldoCorreto_QuandoContaExiste() {
    // ...
}

@Test
void deveLancarExcecao_QuandoContaNaoExiste() {
    // ...
}
```

Estrutura AAA

```
@Test
void deveRealizarTransacao() {
    // Arrange
    var conta = new Conta("123", 1000.0);
    var valor = 100.0;

    // Act
    service.realizarTransacao(conta, valor);

    // Assert
    assertEquals(900.0, conta.getSaldo());
}
```

Mocks e Stubs

```
@Test
void deveValidarUsuario() {
    // Arrange
    var userRepo = mock(UserRepository.class);
    when(userRepo.findById("123"))
        .thenReturn(Optional.of(new User("123")));

    var service = new UserService(userRepo);

    // Act & Assert
    assertTrue(service.validarUsuario("123"));
}
```

Cobertura de Código

- Meta mínima: 80% de cobertura
- Foco em código de negócio
- Usar JaCoCo para relatórios

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.8</version>
</plugin>
```

Execução de Testes

```
# Todos os testes
mvn test

# Testes específicos
mvn test -Dtest=TransactionServiceTest
```

```
# Com cobertura
mvn verify
```

Boas Práticas

1. **Independência**
   - Testes isolados
   - Sem dependências entre testes
   - Setup e teardown apropriados
2. **Legibilidade**
   - Nomes descritivos
   - Comentários quando necessário
   - Organização clara
3. **Manutenibilidade**
   - DRY nos testes
   - Fixtures reutilizáveis
   - Constantes compartilhadas
4. **Confiabilidade**
   - Testes determinísticos
   - Sem dependências externas
   - Timeouts apropriados

CI/CD

- Testes automatizados no pipeline

- Verificação de cobertura
- Relatórios de teste

Guia de Implantação

Preparação

1. **Verificações**
   - Testes passando
   - Cobertura adequada
   - Dependências atualizadas
2. **Versionamento**
   - Semantic Versioning
   - Changelog atualizado
   - Tags no Git

Build

Maven Build

```
# Limpar e compilar
mvn clean compile

# Gerar JAR
mvn package

# Build completo
mvn clean install
```

Artefatos

- JAR executável
- Dependências

- Recursos

Ambientes

Desenvolvimento

```
# application-dev.properties
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Produção

```
# application-prod.properties
spring.jpa.hibernate.ddl-auto=none
spring.jpa.show-sql=false
```

Deploy

Pré-requisitos

- JDK 17+
- MySQL 8.0+
- Configurações de ambiente

Passos

1. **Banco de Dados**

```
# Backup
mysqldump -u root -p transaction_db > backup.sql

# Migrations
mysql -u root -p transaction_db < migrations.sql
```

2. **Aplicação**

```
# Executar JAR
java -jar transaction-module.jar --spring.profiles.active=prod
```

Monitoramento

- Logs centralizados
- Métricas de performance
- Alertas configurados

Rollback

1. **Procedimento**

```
# Reverter versão
git checkout v1.2.3

# Rebuild
mvn clean install

# Deploy anterior
java -jar previous-version.jar
```

2. **Banco de Dados**

```
# Restaurar backup
mysql -u root -p transaction_db < backup.sql
```

Checklist de Deploy

- [ ] Testes executados
- [ ] Versão atualizada
- [ ] Backup realizado
- [ ] Configurações verificadas
- [ ] Documentação atualizada
- [ ] Logs configurados