Católica
Universidade Católica de Brasília

Pró-Reitoria Acadêmica
Curso de Engenharia de Software
e Ciência da Computação
Matéria Laboratório de Banco de Dados

Documentação do Trabalho
Programação Laboratorial de Banco de Dados
BANCO MALVADER

Autor: Isacc Victor de Siqueira Santos,
Marcos Almeida de Souza,
Felipe Jayme Padilha
Professor: William Roberto Malvezzi

1. Visão Geral do Sistema

O **Banco Malvader** é uma aplicação bancária desenvolvida para a disciplina de Laboratório de Banco de Dados. O objetivo foi criar um sistema robusto onde a integridade dos dados e as regras de negócio críticas não dependem apenas da aplicação, mas são garantidas diretamente pelo banco de dados MySQL.

A aplicação simula um ambiente real com três perfis de conta (Corrente, Poupança e Investimento) e implementa fluxos complexos como cálculo dinâmico de score de crédito e autenticação multifator (MFA).

Stack Tecnológica

- **Backend:** Java 25 com Spring Boot 3
- **Frontend:** React.js (Interface web responsiva)
- **Banco de Dados:** MySQL 8.0
- **Segurança:** Spring Security com JWT e Google Authenticator

2. Detalhamento das Regras de Negócio no Banco de Dados

Optamos por uma arquitetura Database-Centric para as regras críticas. Isso significa que a integridade financeira e as validações de segurança não dependem exclusivamente do código Java, mas são garantidas nativamente pelo motor do MySQL. Abaixo, detalhamos como cada mecanismo foi implementado:

2.1. Garantia de Integridade Financeira (Gatilhos de Sincronização)

No sistema bancário, a consistência entre o extrato e o saldo é vital. Para evitar "dinheiro fantasma", removemos a responsabilidade de cálculo de saldo da aplicação Java.

- **Sincronização Automática de Saldo (atualizar_saldo_after_insert):** Ao invés de a aplicação emitir um comando UPDATE no saldo do cliente, ela apenas registra a transação (INSERT na tabela transacao). O banco detecta essa inserção e, através de um gatilho, calcula se deve somar ou subtrair o valor da conta de destino/origem. Isso garante que **jamais** existirá uma transação registrada sem a respectiva alteração no saldo .

- **Geração de Contas com Algoritmo de Luhn (conta_before_insert):** Para evitar erros de digitação e garantir unicidade, o número da conta é gerado automaticamente pelo banco no momento da criação. Utilizamos uma Stored Function personalizada que implementa o **Algoritmo de Luhn** (o mesmo usado em cartões de crédito) para gerar um dígito verificador matematicamente válido.

2.2. Controles de Segurança Ativa (Regras de Bloqueio)

Estas regras funcionam como uma "firewall" de negócios, impedindo operações fraudulentas ou fora do padrão.

- **Travamento de Limites Diários (validar_limite_diario):** Antes de aceitar qualquer movimentação, um gatilho analisa o histórico de transações do dia corrente para aquela conta. Se a nova transação somada ao histórico ultrapassar os limites definidos (R$ 10.000,00 para depósitos, R$ 5.000,00 para saques), o banco aborta a operação e retorna uma exceção SQL personalizada, impedindo a movimentação.
- **Protocolo de Encerramento de Conta (encerrar_conta):** Implementamos uma regra rígida onde o encerramento de uma conta não é um simples UPDATE de status. Uma Procedure verifica atomicamente se o saldo é estritamente **zero** e se a conta não possui pendências. Só então o status é alterado e o evento é logado na tabela de auditoria.

2.3. Inteligência de Negócio e Rotinas Batch

Lógicas complexas que exigiriam muito processamento de memória no Java foram movidas para o banco para ganhar performance.

- **Cálculo Dinâmico de Score (calcular_score_credito):** Criamos um algoritmo dentro do banco que analisa o comportamento financeiro do cliente. A procedure varre todo o histórico de transações, calcula o ticket médio e a antiguidade da conta, atribuindo uma nota de 0 a 100. Isso permite que o banco "decida" automaticamente se um cliente é elegível para melhores condições.
- **Aplicação Automática de Rendimentos (aplicar_rendimento_poupanca):** Uma rotina desenhada para execução diária (Batch) que identifica contas poupança fazendo "aniversário". Ela calcula os juros baseados na taxa contratada e insere os rendimentos sem intervenção manual.

2.4. Camada de Abstração de Dados (Views)

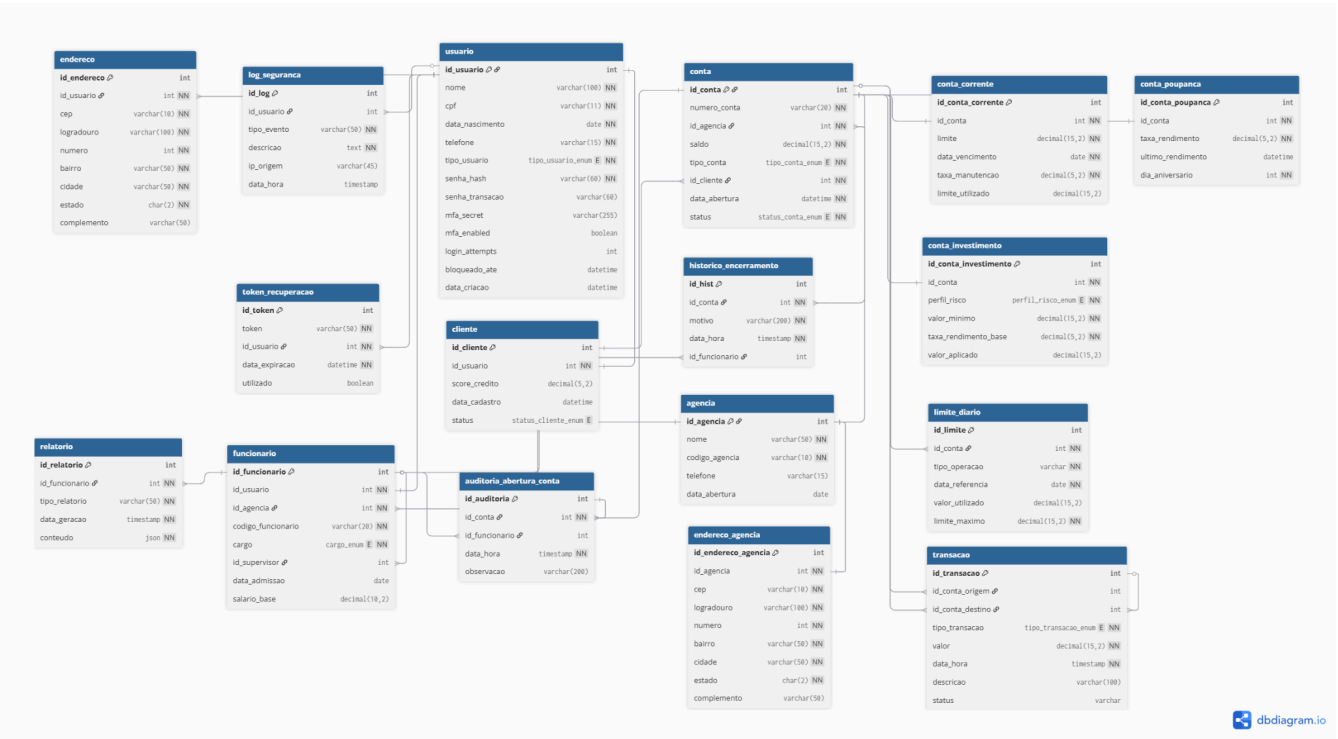
Para simplificar as consultas do Backend e aumentar a segurança dos dados sensíveis:

- **Visão de Movimentações Enriquecida (vw_movimentacoes_recentes):** Ao invés de o Java realizar múltiplos joins complexos toda vez que um extrato é solicitado, criamos uma View que já entrega os dados denormalizados (com nomes de clientes, tipos de conta e descrições formatadas). Isso torna a API mais rápida e o código Java mais limpo.
- **Radar de Inadimplência (vw_clientes_inadimplentes):** Uma visão estratégica que filtra em tempo real clientes com saldo negativo ou uso excessivo de limite, servindo de base para os relatórios gerenciais do sistema.

---

3. Estrutura do Banco de Dados (DER)

A estrutura segue o modelo relacional normalizado:



1. Entidades de Acesso e Perfil

USUARIO

- **Atributos:** id_usuario (PK), nome, cpf (Unique), senha_hash, mfa_secret, tipo_usuario.
- **Relacionamentos:**
  - Possui 1 ou N **ENDERECO**
  - Pode ser 1 **CLIENTE** (Especialização)
  - Pode ser 1 **FUNCIONARIO** (Especialização)
  - Possui 0 ou N **LOG_SEGURANCA**

ENDERECO

- **Atributos:** id_endereco (PK), cep, logradouro, cidade, estado
- **Relacionamentos:** Pertence a 1 **USUARIO**

2. Entidades de Negócio (Atores)

CLIENTE

- **Atributos:** id_cliente (PK), id_usuario (FK), score_credito, status
- **Relacionamentos:**
  - É uma extensão de **USUARIO**
  - Possui 1 ou N **CONTA**

FUNCIONARIO

- **Atributos:** id_funcionario (PK), id_usuario (FK), codigo_funcionario, cargo, id_agencia (FK), id_supervisor (FK)
- **Relacionamentos:**
  - É uma extensão de **USUARIO**
  - Trabalha em 1 **AGENCIA**
  - Pode supervisionar N **FUNCIONARIO** (Auto-relacionamento)
- Gera 0 ou N **RELATORIO**
- Realiza auditoria em 0 ou N **AUDITORIA_ABERTURA_CONTA**

3. Entidades Bancárias (Contas e Agência)

AGENCIA

- **Atributos:** id_agencia (PK), nome, codigo_agencia
- **Relacionamentos:**
    - Possui 1 **ENDERECO_AGENCIA**
    - Administra N **CONTA**
    - Possui N **FUNCIONARIO**

CONTA (Entidade Pai)

- **Atributos:** id_conta (PK), numero_conta, saldo, tipo_conta, status, id_agencia (FK), id_cliente (FK)
- **Relacionamentos:**
    - Pertence a 1 **CLIENTE**
    - Vinculada a 1 **AGENCIA**
    - Pode ser especializada em: **CONTA_CORRENTE**, **CONTA_POUPANCA** ou **CONTA_INVESTIMENTO**
    - Realiza N **TRANSACAO** (como origem ou destino)

Especializações de Conta:

- **CONTA_CORRENTE:** limite, vencimento
- **CONTA_POUPANCA:** taxa_rendimento, aniversario
- **CONTA_INVESTIMENTO:** perfil_risco, valor_minimo

4. Operações

TRANSACAO

- **Atributos:** id_transacao (PK), valor, tipo, data_hora, id_conta_origem (FK), id_conta_destino (FK)
- **Relacionamentos:**
    - Vinculada a 0 ou 1 **CONTA** (Origem)
    - Vinculada a 0 ou 1 **CONTA** (Destino). (Nota: É 0 ou 1 pois Depósitos não têm origem e Saques não têm destino)

4. Como Rodar o Projeto (Instalação)

Passo 1: Banco de Dados

- Tenha o **MySQL 8** instalado.
- Pegue o arquivo Banco_Malvader.sql que está na raiz do projeto.
- Rode o script inteiro no seu cliente SQL (Workbench/DBeaver). Ele já cria o banco, as tabelas, as triggers e insere usuários de teste.

Passo 2: Backend (API)

- Precisa do **JDK 25** configurado.
- No arquivo src/main/resources/application.properties, coloque a senha do seu MySQL.

Na pasta do backend, rode o comando:
./mvnw spring-boot:run

- (O sistema vai subir na porta 8080).

Passo 3: Frontend (Telas)

- Precisa do **Node.js** instalado.
- Entra na pasta fron end.

Roda os comandos:
npm install
npm run dev

- Acesse http://localhost:5173

5. Usuários para Teste

O banco já vem populado para agilizar a apresentação:

- **Para testar como Cliente:**
    - **CPF:** 12345678901
    - **Senha:** Senha123!
- **Para testar como Gerente:**
    - **CPF:** 11111111111
    - **Senha:** Senha123!

6. APIs Desenvolvidas

A comunicação entre o Frontend (React) e o Backend (Spring Boot) é feita através de uma API RESTful. Abaixo estão listados os principais endpoints, seus métodos HTTP e os dados esperados.

Nota: Todos os endpoints protegidos (exceto /auth) exigem o envio do token JWT no cabeçalho Authorization: Bearer <token>.

6.1 Endpoints de Autenticação

Responsáveis pelo controle de acesso e gestão de sessões.

Método	Endpoint	Descrição	Dados Esperados (JSON)
POST	/api/auth/login	Realiza o login e retorna o Token JWT.	{ "cpf": "...", "senha": "..." }
POST	/api/auth/recuperar-senha	Envia email para reset de senha.	{ "email": "..." }
POST	/api/auth/redefinir-senha	Define nova senha com token.	{ "token": "...", "novaSenha": "..." }

6.2 Endpoints de Operações (Cliente)

Permitem a movimentação financeira. As validações de saldo e limite são feitas no banco de dados antes da confirmação.

Método	Endpoint	Descrição	Dados Esperados (JSON)
POST	/api/cliente/transferir	Realiza transferência entre contas.	{ "contaOrigem": 1, "contaDestino": 2, "valor": 100.00 }

GET	/api/cliente/extrato/{id}	Busca o histórico de movimentações.	Nenhum (ID na URL)
POST	/api/cliente/deposito	Realiza depósito em conta.	{ "contaDestino": 1, "valor": 500.00 }
POST	/api/cliente/saque	Realiza saque da conta.	{ "contaOrigem": 1, "valor": 200.00 }

6.3 Endpoints MFA (Autenticação Multifator)

Gerenciam a segurança adicional via Google Authenticator.

Método	Endpoint	Descrição	Dados Esperados
POST	/api/auth/mfa/ativar/{id}	Gera o segredo/QR Code para o app.	Nenhum (ID na URL)
POST	/api/auth/mfa/verificar	Valida o código de 6 dígitos do app.	{ "usuarioId": 1, "codigo": "123456" }
GET	/api/auth/mfa/status/{id}	Verifica se o usuário já ativou o 2FA.	Nenhum (ID na URL)