Documentação do Sistema de Caixa Eletrônico (ATM)

Introdução

O Sistema de Caixa Eletrônico (ATM) é uma aplicação web desenvolvida para facilitar a gestão de transações bancárias, permitindo que usuários realizem operações financeiras. O projeto foi inspirado na necessidade de simplificar o acesso a serviços bancários, como depósitos, saques, transferências, e consultas, em uma interface intuitiva. Alinhado com a ideia de promover inclusão financeira, o sistema oferece uma solução para gerenciar contas e transações, com foco em usabilidade.

O principal desafio foi garantir a sincronização de transações financeiras, como evitar saques ou transferências com saldo insuficiente e registrar todas as operações no banco de dados. Isso foi resolvido com o uso de transações JDBC no backend, separando a lógica de negócios (Servlets), a interface (JSPs), e o estilo (CSS).

Sumário

Conceitos Aplicados

O projeto segue uma arquitetura em camadas, inspirada em boas práticas de desenvolvimento web, com separação de responsabilidades:

- **Servlets**: Responsáveis pela lógica de negócios, processando requisições HTTP e interagindo com o banco de dados via JDBC.
- **JSPs (JavaServer Pages)**: Definem a interface dinâmica, renderizando conteúdo personalizado com base em dados do servidor.

- **JDBC (Java Database Connectivity)**: Gerencia a conexão com o banco de dados Java DB, utilizando PreparedStatements para segurança.
- **CSS**: Estiliza as páginas JSP, garantindo um design responsivo e moderno.

A organização em pacotes (`com.mycompany.atmbancario.servlets` , `com.mycompany.atmbancario.db`) promove modularidade, facilitando a manutenção e escalabilidade, como ensinado em práticas de desenvolvimento Java.

Tecnologias Utilizadas

- **Java 19**: Linguagem principal para o backend, escolhida por sua robustez e suporte a Servlets.
- **Java Servlets**: Tecnologia para processar requisições HTTP no servidor GlassFish.
- **JSP (JavaServer Pages)**: Usado para criar interfaces dinâmicas, integradas com Servlets.
- **Java DB (Apache Derby)**: Banco de dados relacional embarcado, gerenciando dados de usuários, contas, e transações.
- **JDBC**: API para conexão com o Java DB, centralizada na classe `DatabaseConnection` .
- **CSS**: Estilização das páginas JSP, com design responsivo via media queries.
- **GlassFish 7.0.23**: Servidor de aplicação para hospedar a aplicação web.
- **NetBeans 25**: IDE para desenvolvimento, configuração, e execução do projeto.

Estrutura do Projeto

O projeto está organizado em pacotes e arquivos, cada um com uma responsabilidade específica. Abaixo, detalhamos os principais arquivos e suas funcionalidades.

Arquivos SQL

- **esquema.sql** (`com.mycompany.atmbancario`):
  - **Descrição**: Define a estrutura do banco de dados `ATMBancoDB` , criando as tabelas `usuarios` , `contas` , e `transacoes` .
  - **Funcionalidade**:
    - `usuarios` : Armazena dados de usuários (ID, nome, CPF, senha).
    - `contas` : Gerencia contas bancárias (ID, usuário, número da conta, saldo, tipo).
    - `transacoes` : Registra operações financeiras (ID, conta origem/destino, valor, tipo, data).
  - **Detalhes do Código**:
    - Usa `GENERATED ALWAYS AS IDENTITY` para IDs auto-incrementados.
    - Define chaves estrangeiras para relacionar tabelas (`id_usuario` em `contas` , `id_conta` em `transacoes`).

    - Garante unicidade em campos como CPF e número da conta com `UNIQUE`.

Arquivos Java

Pacote `com.mycompany.atmbancario.db`

- **DatabaseConnection.java**:
  - **Descrição**: Centraliza a conexão com o banco `ATMBancoDB` via JDBC.
  - **Funcionalidade**:
    - Define constantes para URL (`jdbc:derby://localhost:1527/ATMBancoDB`), usuário (`root`), e senha (`123`).
    - Fornece um método estático `getConnection()` que retorna uma conexão JDBC.
  - **Detalhes do Código**:
    - Usa `DriverManager.getConnection()` para estabelecer a conexão.
    - Evita duplicação de código, sendo reutilizado por todos os Servlets.

Pacote `com.mycompany.atmbancario.servlets`

- **LoginServlet.java** (`/login`):
  - **Descrição**: Autentica usuários verificando CPF e senha.
  - **Funcionalidade**:
    - Recebe CPF e senha via POST, consulta a tabela `usuarios` com `PreparedStatement`.
    - Armazena `id_usuario` na sessão HTTP se válido, redireciona para `atm.jsp`.
    - Retorna erro para `index.jsp` se inválido.
  - **Detalhes do Código**:
    - Usa JDBC para consulta segura (`SELECT id_usuario FROM usuarios WHERE cpf = ? AND senha = ?`).
    - Gerencia sessão com `request.getSession().setAttribute()`.
- **CadastroServlet.java** (`/cadastro`):
  - **Descrição**: Cadastra novos usuários e suas contas.
  - **Funcionalidade**:
    - Recebe nome, CPF, e senha via POST, insere na tabela `usuarios`.
    - Cria uma conta com número aleatório e tipo "Corrente" na tabela `contas`.
    - Usa transações JDBC para garantir consistência entre usuário e conta.
    - **Detalhes do Código**:
      - Usa `conn.setAutoCommit(false)` e `conn.commit()` para transações.
      - Gera número de conta com `Math.random()`.
      - Valida inserções com `PreparedStatement.RETURN_GENERATED_KEYS`.
- **DepositoServlet.java** (`/deposito`):

    - **Descrição**: Realiza depósitos em uma conta.
    - **Funcionalidade**:
        - Recebe valor via POST, atualiza o saldo na tabela `contas`.
        - Registra a transação na tabela `transações` com tipo "DEPOSITO".
        - Usa transações para consistência.
    - **Detalhes do Código**:
        - Verifica usuário logado via sessão (`id_usuario`).
        - Usa subconsulta SQL para inserir transação (`SELECT id_conta FROM contas WHERE id_usuario = ?`).
- **SaldoServlet.java** (`/saldo`):
    - **Descrição**: Consulta o saldo da conta do usuário logado.
    - **Funcionalidade**:
        - Busca o saldo na tabela `contas` via GET e passa para `saldo.jsp`.
        - Redireciona para erro se a conta não for encontrada.
    - **Detalhes do Código**:
        - Usa `request.setAttribute()` para passar o saldo ao JSP.
        - Consulta simples com `SELECT saldo FROM contas WHERE id_usuario = ?`.
- **ExtratoServlet.java** (`/extrato`):
    - **Descrição**: Lista as transações da conta do usuário.
    - **Funcionalidade**:
        - Busca transações na tabela `transações` via GET, vinculando com `contas`.
        - Formata transações em uma lista de strings e passa para `extrato.jsp`.
    - **Detalhes do Código**:
        - Usa JOIN SQL para buscar transações (`SELECT ... FROM transacoes t JOIN contas c ...`).
        - Armazena resultados em `ArrayList` para exibição dinâmica.
- **SaqueServlet.java** (`/saque`):
    - **Descrição**: Realiza saques, verificando saldo suficiente.
    - **Funcionalidade**:
        - Recebe valor via POST, verifica saldo na tabela `contas`.
    - **Detalhes do Código**:
        - Valida saldo com consulta prévia (`SELECT saldo FROM contas WHERE id_usuario = ?`).
        - Retorna erro se saldo insuficiente.
- **TransferenciaServlet.java** (`/transferencia`):
    - **Descrição**: Transfere valores entre contas.
    - **Funcionalidade**:
        - Recebe número da conta destino e valor via POST.
        - Verifica saldo da conta origem e existência da conta destino.

        - Atualiza saldos e registra transação ("TRANSFERENCIA").
    - **Detalhes do Código**:
        - Usa múltiplas consultas para validar contas (`SELECT id_conta, saldo ...`).
        - Transação JDBC garante atomicidade.
- **InvestimentoServlet.java** (`/investimento`):
    - **Descrição**: Simula investimentos com juros compostos.
    - **Funcionalidade**:
        - Recebe valor, taxa, e meses via POST, calcula montante e rendimento.
        - Registra transação ("INVESTIMENTO") e passa resultados para `investimento.jsp` .
    - **Detalhes do Código**:
        - Usa `Math.pow()` para cálculo de juros compostos.
        - Formata resultados com `String.format()` .

Arquivos JSP (Pasta `webapp`)

- **index.jsp**:
    - **Descrição**: Tela de login do usuário.
    - **Funcionalidade**:
        - Exibe formulário para CPF e senha, enviando para `/login` .
        - Mostra mensagem de erro se credenciais inválidas.
    - **Detalhes do Código**:
        - Usa `<form action="login" method="post">` para POST.
        - Condicional JSP (`<% if (request.getParameter("error") != null) %>`) para erros.
- **cadastro.jsp**:
    - **Descrição**: Tela de cadastro de novos usuários.
    - **Funcionalidade**:
        - Exibe formulário para nome, CPF, e senha, enviando para `/cadastro` .
        - Mostra mensagens de sucesso ou erro.
    - **Detalhes do Código**:
        - Formulário com `required` para validação básica.
        - Condicionais JSP para feedback.
- **atm.jsp**:
    - **Descrição**: Menu principal do ATM.
    - **Funcionalidade**:
        - Exibe formulários para depósito, saque, transferência, e investimento.
        - Links para consultar saldo e extrato.
        - Mostra mensagens de sucesso ou erro.
    - **Detalhes do Código**:

        - Múltiplos `<form>` para cada operação.
        - Links `<a href="saldo">` para GET.
- **saldo.jsp**:
    - **Descrição**: Exibe o saldo da conta.
    - **Funcionalidade**:
        - Mostra o valor do saldo retornado por `SaldoServlet`.
        - Link para voltar ao menu.
    - **Detalhes do Código**:
        - Usa `<%= request.getAttribute("saldo") %>` para exibir saldo.
- **extrato.jsp**:
    - **Descrição**: Lista as transações da conta.
    - **Funcionalidade**:
        - Exibe transações em uma lista `<ul>`, formatadas por `ExtratoServlet`.
        - Mostra mensagem se não houver transações.
    - **Detalhes do Código**:
        - Laço JSP (`<% for (String transacao : transacoes) %>`) para exibir lista.
- **investimento.jsp**:
    - **Descrição**: Exibe resultados de simulação de investimento.
    - **Funcionalidade**:
        - Mostra montante e rendimento calculados por `InvestimentoServlet`.
        - Link para voltar ao menu.
    - **Detalhes do Código**:
        - Usa `<%= request.getAttribute("montante") %>` para resultados.

Arquivo CSS (Pasta `webapp`)

- **styles.css**:
    - **Descrição**: Estiliza todas as páginas JSP.
    - **Funcionalidade**:
        - Define layout responsivo com Flexbox e media queries.
    - **Detalhes do Código**:
        - Classe `.container` centraliza o conteúdo.
        - Media query `@media (max-width: 600px)` ajusta para telas menores.
        - Estilos para `.error` (vermelho) e `.success` (verde).

Requisitos

- **JDK 19**: Necessário para compilar e executar a aplicação.

- **NetBeans 25**: IDE recomendada para desenvolvimento e execução.
- **GlassFish 7.0.23**: Servidor de aplicação para hospedar a aplicação web.
- **Java DB (Apache Derby)**: Banco de dados embarcado, incluído no NetBeans.

Configuração do Ambiente

1. **Configurar o Java DB**:
   - No NetBeans, vá para **Services** > **Databases** > **Java DB** > **Start Server**.
   - Conecte-se ao banco `jdbc:derby://localhost:1527/ATMBancoDB` com usuário `root` e senha `123`.
   - Execute o script `esquema.sql`:
     - Clique com o botão direito em `ATMBancoDB` > **Execute Command**.
     - Execute para criar as tabelas `usuarios`, `contas`, e `transacoes`.
2. **Configurar o Projeto no NetBeans**:
   - Abra o projeto `ATMBancario` no NetBeans.
   - Certifique-se de que o GlassFish 7.0.23 está configurado como servidor.

Como Executar

1. **Iniciar o GlassFish**:
   - No NetBeans, vá para **Services** > **Servers** > **GlassFish Server** > **Start**.
2. **Executar o Projeto**:
   - Clique com o botão direito em `ATMBancario` > **Run**.
   - O navegador abrirá em `http://localhost:8080/ATMBancario/index.jsp`.
3. **Testar Funcionalidades**:
   - Cadastre um usuário (ex.: Nome: "João Silva", CPF: "123.456.789-00", Senha: "1234").
   - Faça login.
   - Use o menu para realizar depósitos, saques, transferências, consultar saldo, ver extrato, ou simular investimentos.