Documentação do Projeto – Sistema Bancário

1. Introdução

O projeto Sistema Bancário foi originalmente desenvolvido em Java, com estrutura e toda a lógica de negócio implementada no próprio código-fonte. Para alterar sua arquitetura e facilitar expansões, o sistema foi refatorado para adotar uma separação entre frontend (cliente) e backend (servidor).

A nova versão segue o modelo cliente/servidor, no qual o backend expõe uma API REST desenvolvida em Node.js + Express, responsável pela persistência e regras de negócio, enquanto o frontend (em Java) atua apenas como consumidor dessa API.

2. Arquitetura do Sistema

A arquitetura foi dividida em duas camadas principais:

Backend (API em Node.js)

- Tecnologias: Node.js, Express.js, PostgreSQL, biblioteca pg, dotenv.
- Estrutura da API: rotas /clientes e /contas para CRUD de clientes e contas correntes.
- Segurança: dados de conexão ao banco são isolados no arquivo .env.
- Banco de dados foi criado no Render (dashboard.render.com), logo é possível acessar o mesmo banco em qualquer ambiente durante o desenvolvimento.

Frontend (app Console em Java)

- Tecnologias: Java 8+, biblioteca Gson para JSON, HttpURLConnection para HTTP.
- Classes principais: Cliente, ContaCorrente, GerenciadoraClientes, GerenciadoraContas, ApiClient.
- Tratamento de exceções feito internamente nas classes gerenciadoras, mantendo a classe main inalterada.
- Implementado funcionalidades faltantes.

3. Fluxo de Execução

1. O usuário acessa o sistema em Java.
2. O menu da classe main permite cadastrar clientes, abrir contas, consultar saldos, depositar, sacar e efetuar uma transferência.
3. As operações são enviadas para a API Node.js em vez de listas locais em memória.
4. A API processa as requisições e persiste os dados no PostgreSQL.

5. A resposta em JSON é retornada e tratada pelo cliente Java com Gson.

4. Vantagens da Nova Arquitetura

- Manutenção: separação entre lógica de apresentação (Java) e lógica de negócio (Node.js).
- Flexibilidade: o frontend pode ser substituído por outro cliente (ex.: Android ou Web).

5. Testes implementados

Tipos de teste

- Caixa preta: foca em entradas/saídas (AVL e Fuzz).
- Caixa branca: foca em caminhos/ramificações e efeitos internos.
- Caixa cinza: combina entradas/saídas com parte da lógica interna (Matriz de Decisão e Matriz Ortogonais).

Métodos

- AVL (Análise de Valor Limite) (clienteController.avl.test.js):
  - Verifica idades 17, 18, 19 em criarCliente.
  - Espera sucesso para >= 18 e erro para < 18.
- Ramificação (clienteController.ramificacao.test.js):
  - Cobre sucesso e erro de validação.
  - Assegura que queries para o banco não são chamadas em entradas inválidas.
- Loops (contaController.loops.test.js):
  - depositar: aceita valor positivo, rejeita zero/negativo com erro e sem executar query.
  - sacar: sucesso com saldo suficiente, erro para saldo insuficiente ou para conta inexistente.
- Fuzzing (clienteController.fuzz.js):
  - Gera entradas aleatórias para criarCliente (nome, idade, email, id_conta).
  - Busca revelar exceções e comportamentos inesperados fora dos casos determinísticos.
- Matrizes de Decisão (contaController.matriz.test.js):
  - Define combinações de condições (ex.: saldo suficiente/insuficiente, conta válida/inexistente, valor positivo/negativo).
  - Garante cobertura de múltiplos cenários de regras de negócio com menos casos redundantes.
- Matrizes Ortogonais (contaController.matrizOrtogonal.test.js):
  - Seleciona subconjunto representativo de combinações possíveis (idade, email válido/inválido, conta vinculada/nula).
  - Reduz a explosão combinatória mantendo boa cobertura de interações entre variáveis.

Ferramentas

- Jest: framework de testes unitários e de integração para JavaScript, que fornece ambiente de execução isolado, mocks, asserções e relatórios para validação automatizada do código.

- Jazzer: fuzzer para aplicações JavaScript e Node.js, gerando entradas aleatórias ou mutadas para explorar diferentes caminhos de execução e detectar falhas, vulnerabilidades e exceções de forma automatizada.
- Supertest: biblioteca para Node.js usada em conjunto com frameworks como Express, que permite testar requisições HTTP de forma programática, validando respostas de APIs e endpoints em testes automatizados.

Por que esses testes?

- Validar regras de negócio críticas (idade mínima, valores positivos).
- Garantir decisões corretas e evitar efeitos colaterais indevidos (não acessar DB quando inválido).
- Verificar a ordem correta das operações em transações financeiras.
- Aumentar robustez contra inputs inesperados via fuzzing.

6. Observação

Durante o processo, foram utilizadas ferramentas de apoio baseadas em Inteligência Artificial. O objetivo foi sugerir trechos de código e auxiliar na usabilidade das ferramentas de testes. Todas as implementações foram revisadas manualmente pelos desenvolvedores para garantir a qualidade do projeto.

7. Repositório

O repositório do projeto está no GitHub:

- https://github.com/GuiNegretto/Projeto_Teste_Sistema_Nodejs