▶️ Procedimento 2 - Servidor Completo e Cliente Assíncrono

✅ Objetivos

O projeto tem como objetivo desenvolver uma aplicação cliente-servidor para cadastro e movimentação de produtos em estoque. Os usuários autenticados podem listar produtos, registrar movimentações de entradas e saídas de produtos, bem como consultar informações diretamente de um banco de dados relacional, utilizando comunicação em rede baseada em sockets Java.

🛠️ Tecnologias Utilizadas

- **Linguagem:** Java (JDK 17+)
- **IDE:** NetBeans
- **Comunicação:** Sockets TCP/IP (Object Streams)
- **Persistência:** JPA (Java Persistence API)
- **Banco de dados:** Microsoft SQL Server. Base configurada via persistence.xml
- **Modelo de dados:** Entidades JPA. Usuario e Produto

🧱 Estrutura do Projeto

- **Projeto Servidor (cadastroserver)**
  - Camada de controle (controllers): ProdutoJpaController, UsuarioJpaController, PessoaJpaController, OperacaoJpaController
  - Modelo de dados (entidades JPA): Produto, Pessoa, Usuario, Operacao
  - Servidor principal: CadastroServer
  - Thread de atendimento por cliente: CadastroThreadV2
- **Projeto Cliente (cadastroclient)**
  - Aplicação de console: CadastroClient e CadastroClientV2 (versão assíncrona)
  - Comunicação via Socket com servidor
  - Envio de comandos e leitura de respostas

Funcionalidades

- Login de usuários com verificação via banco de dados.
- Listagem de todos os produtos com informações de estoque e preço.
- Registro de entrada (compra) e saída (venda) de produtos, com atualização de estoque.
- Armazenamento de operações no banco, com rastreamento de comprador, vendedor, data, quantidade e valor unitário.
- Interface gráfica para exibir mensagens e operações recebidas do servidor.
- Armazenamento de movimentações com data, quantidade e valor unitário.
- Controle de estoque com atualização automática.
- Atendimento concorrente de múltiplos clientes via Thread.

Atividades Realizadas

- Criação das entidades JPA com anotações e relacionamentos
- Implementação dos controladores (JpaControllers) para cada entidade
- Desenvolvimento do servidor multithreaded (CadastroServer e CadastroThreadV2)
- Implementação do cliente console com menu de comandos
- Testes de conexão, persistência e atualização de dados
- Tratamento de exceções e mensagens de erro

Desafios Enfrentados

- Tratamento de exceções silenciosas e erros de sincronização nos fluxos de rede.
- Compatibilização entre os tipos de dados enviados pelo cliente e esperados pelo servidor.
- Configuração e testes da unidade de persistência com JPA.
- Gerenciamento correto de ObjectInputStream e ObjectOutputStream.
- Controle de concorrência e sincronização das threads.
- Validação de comandos e consistência de dados no banco.
- Conversão e manipulação de tipos como BigDecimal e Date.
- Criação correta de associações entre entidades como Operacao, Produto e Pessoa.

📄 Códigos

Os códigos foram desenvolvidos com a IDE NetBeans e se encontram em repositório no GitHub onde podem ser acessados pelos links:

Servidor:

https://github.com/CarlosCatao/Mundo_3_Nivel_5_Missao_Pratica/tree/main/Procedimento-2/CadastroServer

Cliente:

https://github.com/CarlosCatao/Mundo_3_Nivel_5_Missao_Pratica/tree/main/Procedimento-2/CadastroClientV2

🧪 Testes

- Testes manuais de todas as operações (L, E, S, X) com dados reais.
- Simulação de múltiplos clientes se conectando simultaneamente ao servidor.
- Verificação de mensagens exibidas na interface gráfica para feedback em tempo real.
- Teste de persistência e consistência de estoque após entrada/saída.
- Simulação de entradas inválidas (ID inexistente, valor negativo, string ao invés de número) para validação de robustez.
- Verificação de atualização do estoque após entradas e saídas.

📜 Resultados

Os resultados da execução dos códigos se encontram ilustrados no arquivo *Resultados.pdf* que se encontra em repositório no GitHub onde podem ser acessados pelo link: https://github.com/CarlosCatao/Mundo_3_Nivel_5_Missao_Pratica/blob/main/Procedimento-2/RESULTADOS%20PROCEDIMENTO-2.pdf

📌 Conclusão

O sistema desenvolvido atinge com sucesso o objetivo de registrar movimentações comerciais em uma estrutura cliente-servidor robusta, segura e modular. A aplicação oferece tanto uma interação via console quanto uma interface gráfica útil para mensagens, integrando os conceitos de rede, banco de dados relacional, programação orientada a objetos e interface com o usuário. Os desafios encontrados ao longo do desenvolvimento foram valiosos para consolidar conhecimentos técnicos e fortalecer práticas de depuração, tratamento de exceções e modelagem de dados.

❓ Questionamentos

▶️ Como as Threads podem ser utilizadas para o tratamento assíncrono das respostas enviadas pelo servidor?

As **threads** podem ser utilizadas para o tratamento assíncrono das respostas enviadas pelo servidor com o objetivo de evitar que a aplicação cliente fique bloqueada enquanto aguarda por uma resposta; isto é especialmente importante em aplicações que precisam continuar responsivas ou lidar com múltiplas tarefas simultaneamente.

▶️ Para que serve o método invokeLater, da classe SwingUtilities?

O método *invokeLater()* da classe *SwingUtilities* é utilizado para agendar a execução de um trecho de código na ***Event Dispatch Thread*** **(EDT)**, a *thread* responsável por atualizar a interface gráfica (GUI) em aplicações que usam **Swing**.

▶️ Como os objetos são enviados e recebidos pelo Socket Java?

No Java, os objetos podem ser enviados e recebidos através de *Sockets* usando os fluxos de entrada e saída com suporte à *serialização de objetos*, por meio das classes:

- *ObjectOutputStream* (para enviar objetos),
- *ObjectInputStream* (para receber objetos).

▶ Compare a utilização de comportamento assíncrono ou síncrono nos clientes com Socket Java, ressaltando as características relacionadas ao bloqueio do processamento.

A comparação entre comportamento assíncrono e síncrono em clientes que utilizam *Socket* em Java envolve principalmente a forma como o processamento lida com bloqueios durante comunicação de rede.

Característica	Comportamento Síncrono	Comportamento Assíncrono (com Threads ou NIO)
Execução do código	Linha por linha, bloqueia até receber resposta	Continua executando, não bloqueia enquanto espera
Bloqueio (blocking I/O)	Sim, métodos como read() ou readObject() bloqueiam	Não, a comunicação é tratada em segundo plano
Responsividade da aplicação	Baixa — se a rede for lenta, tudo "trava"	Alta — a interface ou lógica principal continua fluindo
Uso de threads	Opcional (mas recomendável em GUIs)	Necessário (Threads, ExecutorService ou NIO)
Complexidade do código	Simples	Mais complexo (gerenciamento de threads ou callbacks)
Controle de concorrência	Baixo	Requer controle (sincronização, locks, filas, etc.)
Escalabilidade	Limitada — cada cliente precisa de uma thread dedicada	Melhor — com NIO ou thread pool, escala bem
Exemplo típico de uso	Programas de linha de comando simples	Chats, servidores multiplayer, GUIs, apps interativas