▶️ Procedimento 1 - Criando o Servidor e Cliente de Teste

✅ Objetivos

O objetivo principal deste projeto foi desenvolver uma aplicação cliente/servidor em Java, com foco em autenticação de usuários e gerenciamento de produtos, utilizando comunicação via sockets e persistência de dados com JPA. O sistema foi pensado para ser modular, escalável e aplicável a contextos reais.

🛠️ 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

O projeto foi dividido em dois módulos principais:

1. **Módulo Servidor (cadastroserver):**
    - CadastroServer: ponto de entrada principal, escuta conexões.
    - CadastroThread: thread dedicada a cada cliente conectado.
    - UsuarioJpaController e ProdutoJpaController: classes para acesso ao banco de dados via JPA.
2. **Módulo Cliente (cadastroclient):**
    - CadastroClient: aplicação que conecta ao servidor, envia comandos, recebe e exibe dados.
    - Utiliza ObjectInputStream e ObjectOutputStream para comunicação binária.

⚙️ Funcionalidades

- Autenticação de usuários via login e senha.
- Validação de credenciais contra o banco de dados.
- Retorno de mensagens de autenticação ao cliente.
- Comando L (listar): retorna todos os produtos cadastrados.
- Comando S: encerra a sessão do usuário.
- Comunicação segura entre cliente e servidor via objetos serializados.

📝 Atividades Realizadas

- Modelagem das entidades Usuario e Produto.
- Implementação dos controladores JPA para acesso ao banco.
- Criação da estrutura de sockets para comunicação bidirecional.
- Controle de múltiplas conexões simultâneas no servidor com threads.
- Tratamento de exceções e encerramento seguro de conexões.

🚧 Desafios Enfrentados

- Gerenciamento correto de fluxos de entrada e saída com ObjectInputStream e ObjectOutputStream.
- Evitar exceções como EOFException, causadas por falhas de sincronização entre cliente e servidor.
- Manutenção de conexões persistentes sem travar o servidor.
- Garantia de integridade dos dados durante a serialização/deserialização.

📄 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-1/CadastroServer

Cliente:

https://github.com/CarlosCatao/Mundo_3_Nivel_5_Missao_Pratica/tree/main/Procedimento-1/CadastroClient

🧪 Testes

- Testes manuais com múltiplos usuários e comandos.
- Simulação de falhas de autenticação.
- Verificação de resposta do servidor em tempo real (mensagens e lista de produtos).
- Teste de robustez com entrada de comandos inválidos.

📜 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-1/RESULTADOS%20PROCEDIMENTO%201.pdf

📌 Conclusão

O projeto alcançou todos os seus objetivos: foi possível implementar uma comunicação cliente-servidor robusta com autenticação, listagem de dados e controle de sessões. A integração de Java com JPA e sockets mostrou-se eficiente e adequada ao contexto proposto. O aprendizado com os desafios enfrentados, especialmente relacionados à comunicação de rede, reforçou conceitos fundamentais de programação distribuída e persistência de dados.

❓ Questionamentos

▶️ Como funcionam as classes Socket e ServerSocket?

As classes *Socket* e *ServerSocket* fazem parte da API de redes (networking) e são usadas para implementar comunicação entre computadores (ou processos)

A Classe *ServerSocket* cria um **servidor** TCP que escuta conexões de clientes e a Classe *Socket* cria um **cliente** TCP ou lida com a conexão de um cliente no servidor.

Estas Classes funcionam segundo o seguinte fluxo:

1. O **servidor** cria um ServerSocket e fica esperando conexões.
2. O **cliente** cria um Socket e se conecta ao IP/porta do servidor.

3. O ServerSocket aceita essa conexão e retorna um Socket para se comunicar com o cliente.
4. Ambos os lados trocam dados usando *InputStream* e *OutputStream*.

▶️ Qual a importância das portas para a conexão com servidores?

As **portas** são importantes para a comunicação entre computadores na rede, pois elas funcionam como **"canais"** que permitem que múltiplas aplicações ou serviços se comuniquem simultaneamente em um mesmo dispositivo.

Quando o cliente se conecta ao IP do servidor, ele também precisa saber , senão, não saberá qual serviço usar.

▶️ Para que servem as classes de entrada e saída ObjectInputStream e ObjectOutputStream, e por que os objetos transmitidos devem ser serializáveis?

As classes *ObjectInputStream* e *ObjectOutputStream* são utilizadas para ler e escrever objetos inteiros (com estado e atributos) em fluxos de entrada/saída entre arquivos, redes ou memória, por exemplo. Elas fazem parte do mecanismo de **serialização de objetos** em Java.

Objetos em Java precisam ser **"serializáveis"** (ou seja, implementar a interface *Serializable*) para que possam ser convertidos em uma sequência de bytes em um processo denominado de **serialização**.

▶️ Por que, mesmo utilizando as classes de entidades JPA no cliente, foi possível garantir o isolamento do acesso ao banco de dados?

Porque as classes de entidade JPA no cliente são apenas representações de dados, não têm capacidade de acessar o banco diretamente. O isolamento é garantido porque a lógica de acesso ao banco (via *EntityManager*, transações, queries etc.) permanece no servidor.