Desenvolvimento de Aplicações Cliente-Servidor em Java com Uso de Sockets e Threads.

Missão Prática | Nível 5 | Mundo 3

- **Aluna:** Amanda Duque Kawauchi
- **Matrícula:** 202209170254
- **Campus:** Morumbi

- **Curso:** Desenvolvimento Full-Stack
- **Disciplina:** Nível 5: Por Que Não Paralelizar?
- **Turma:** 2023.3
- **Semestre Letivo:** 3º Semestre
- **Repositório GitHub:** Link do Repositório GitHub

Objetivo da Prática

O objetivo da prática é desenvolver um servidor Java baseado em Sockets e implementar clientes síncronos e assíncronos, utilizando Threads para processamento paralelo. A meta é integrar o servidor com um banco de dados via JPA e praticar a criação de aplicações cliente-servidor eficientes em Java.

Análise e Conclusão

1º Procedimento | Criando o Servidor e Cliente de Teste

a. Como funcionam as classes Socket e ServerSocket?

As classes `Socket` e `ServerSocket` são fundamentais para a implementação de comunicação de rede em Java. `ServerSocket` atua como um ponto de escuta para as solicitações de conexão dos clientes. Quando um `ServerSocket` é instanciado e vinculado a uma porta específica, ele aguarda as solicitações dos clientes. Uma vez que uma solicitação é recebida, `ServerSocket` aceita a conexão e cria um `Socket` para comunicar com o cliente. A classe `Socket`, por outro lado, é usada tanto pelo cliente para estabelecer uma conexão com o servidor quanto pelo servidor para comunicar-se com o cliente após a conexão ter sido estabelecida. Esta abordagem permite uma comunicação bidirecional entre cliente e servidor.

b. Qual a importância das portas para a conexão com servidores?

As portas são cruciais em comunicações de rede, pois atuam como pontos de extremidade na comunicação entre o servidor e os clientes. Cada serviço em um servidor escuta em uma porta específica, permitindo que os clientes saibam para onde direcionar suas solicitações. Por exemplo, um servidor pode ter vários serviços rodando, como um servidor web na porta 80 e um servidor de banco de dados em outra porta. Sem as portas, seria difícil para o cliente saber como se conectar a um serviço específico no servidor. Além disso, as portas ajudam na organização do tráfego de rede e na implementação de medidas de segurança.

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

`ObjectInputStream` e `ObjectOutputStream` são usadas para ler e escrever objetos em uma conexão de rede, respectivamente. Essas classes permitem a transmissão de objetos Java através de Sockets. Para que um objeto possa ser transmitido dessa maneira, ele deve ser serializável, o que significa que ele deve implementar a interface `Serializable`. Isso é necessário porque a serialização converte o estado do objeto em um formato que pode ser enviado através da rede e posteriormente desserializado no lado receptor, reconstruindo o objeto em sua forma original. Sem serialização, a complexidade e os riscos de corrupção de dados ao enviar objetos complexos pela rede aumentariam significativamente.

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

O isolamento do acesso ao banco de dados, mesmo utilizando classes de entidades JPA no cliente, foi possível devido à arquitetura do projeto. As classes de entidades JPA no cliente são usadas apenas como representações dos dados, sem qualquer conexão direta ao banco de dados. Toda a lógica de acesso ao banco de dados é manipulada no lado do servidor. Isso significa que o cliente apenas envia e recebe objetos, mas não interage diretamente com o banco de dados. Essa separação garante que o acesso ao banco de dados seja centralizado, seguro e gerenciado exclusivamente pelo servidor, evitando acessos não autorizados e mantendo a integridade dos dados.

2º Procedimento | Servidor Completo e Cliente Assíncrono

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

Threads são utilizadas para permitir que o servidor lide com múltiplas conexões de clientes simultaneamente, sem que uma bloqueie a outra. No contexto assíncrono, cada cliente é atendido por uma Thread separada no servidor. Isso significa que quando um cliente envia uma requisição, o servidor cria uma nova

Thread para lidar com essa requisição, permitindo que o servidor continue livre para aceitar outras requisições. Do lado do cliente, a utilização de Threads permite que a interface do usuário continue responsiva enquanto espera por respostas do servidor. Isso é crucial em aplicações onde a espera pela resposta do servidor pode ser longa e não deve bloquear a interação do usuário com a aplicação.

b. Para que serve o método invokeLater, da classe SwingUtilities?

O método `invokeLater` da classe `SwingUtilities` é utilizado para garantir que as alterações na interface gráfica do usuário (GUI) sejam feitas de maneira segura em aplicações Swing. Em Java Swing, todas as alterações na GUI devem ocorrer na Thread de despacho de eventos (EDT - Event Dispatch Thread) para evitar problemas de sincronização e inconsistências visuais. O `invokeLater` permite que um trecho de código seja executado na EDT, independentemente da Thread em que o código é chamado. Isso é especialmente útil em aplicações multithreaded, onde as Threads de background precisam atualizar a GUI sem causar erros ou bloqueios.

c. Como os objetos são enviados e recebidos pelo Socket Java?

No Java, objetos são enviados e recebidos através de Sockets utilizando as classes `ObjectOutputStream` e `ObjectInputStream`. Essas classes permitem a serialização e desserialização de objetos para um formato que pode ser transmitido através de uma conexão de rede. Quando um objeto é enviado, ele é primeiro serializado (convertido em uma sequência de bytes) pelo `ObjectOutputStream` e então transmitido através do Socket. No lado receptor, o `ObjectInputStream` lê a sequência de bytes recebida pelo Socket e desserializa-a, reconstruindo o objeto original. É importante que os objetos transmitidos implementem a interface `Serializable` para que essa operação seja possível.

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

Em clientes com comportamento síncrono, as requisições ao servidor bloqueiam a Thread até que a resposta seja recebida. Isso significa que, durante esse tempo, o cliente não pode realizar outras tarefas ou responder a eventos do usuário, o que pode levar a uma experiência de usuário ruim, especialmente em operações de rede demoradas. Por outro lado, em clientes assíncronos, as requisições são feitas em uma Thread separada, permitindo que a interface do usuário continue interativa e responsiva. O cliente assíncrono não bloqueia enquanto espera pela resposta do servidor, podendo continuar processando outras tarefas ou eventos. Isso resulta em uma melhor experiência do usuário, com interfaces mais fluidas e responsivas, mesmo durante operações de rede demoradas.