



POLO CHAPADA - MANAUS - AM/UNIVERSIDADE ESTÁCIO DE SÁ

Missão Prática | Nível 5 | Mundo 3

Curso: Desenvolvimento Full Stack

Disciplina Nível 5: RPG0018 - Por que não paralelizar

Número da Turma: 2024.2

Semestre Letivo: Mundo-3

Aluno: Gilvan Júnior Nascimento Gonçalves

Matrícula: 202304560188

URL GIT: <https://github.com/Kakarotox10/Mundo3-MissaoPratica-Nivel5.git>

1º Título da Prática: Por que não paralelizar

Servidores e clientes baseados em Socket, com uso de Threads tanto no lado cliente quanto no lado servidor, acessando o banco de dados via JPA.

2º Objetivo da Prática:

1. Criar servidores Java com base em Sockets.
2. Criar clientes síncronos para servidores com base em Sockets.
3. Criar clientes assíncronos para servidores com base em Sockets.
4. Utilizar Threads para implementação de processos paralelos.
5. No final do exercício, o aluno terá criado um servidor Java baseado em Socket, com acesso ao banco de dados via JPA, além de utilizar os recursos nativos do Java para implementação de clientes síncronos e assíncronos. As Threads serão usadas tanto no servidor, para viabilizar múltiplos clientes paralelos, quanto no cliente, para implementar a resposta assíncrona.

1º Procedimento | Criando o Servidor e Cliente de Teste

1. Criar o projeto do servidor, utilizando o nome **CadastroServer**, do tipo console, no modelo Ant padrão, para implementar o protocolo apresentado a seguir:

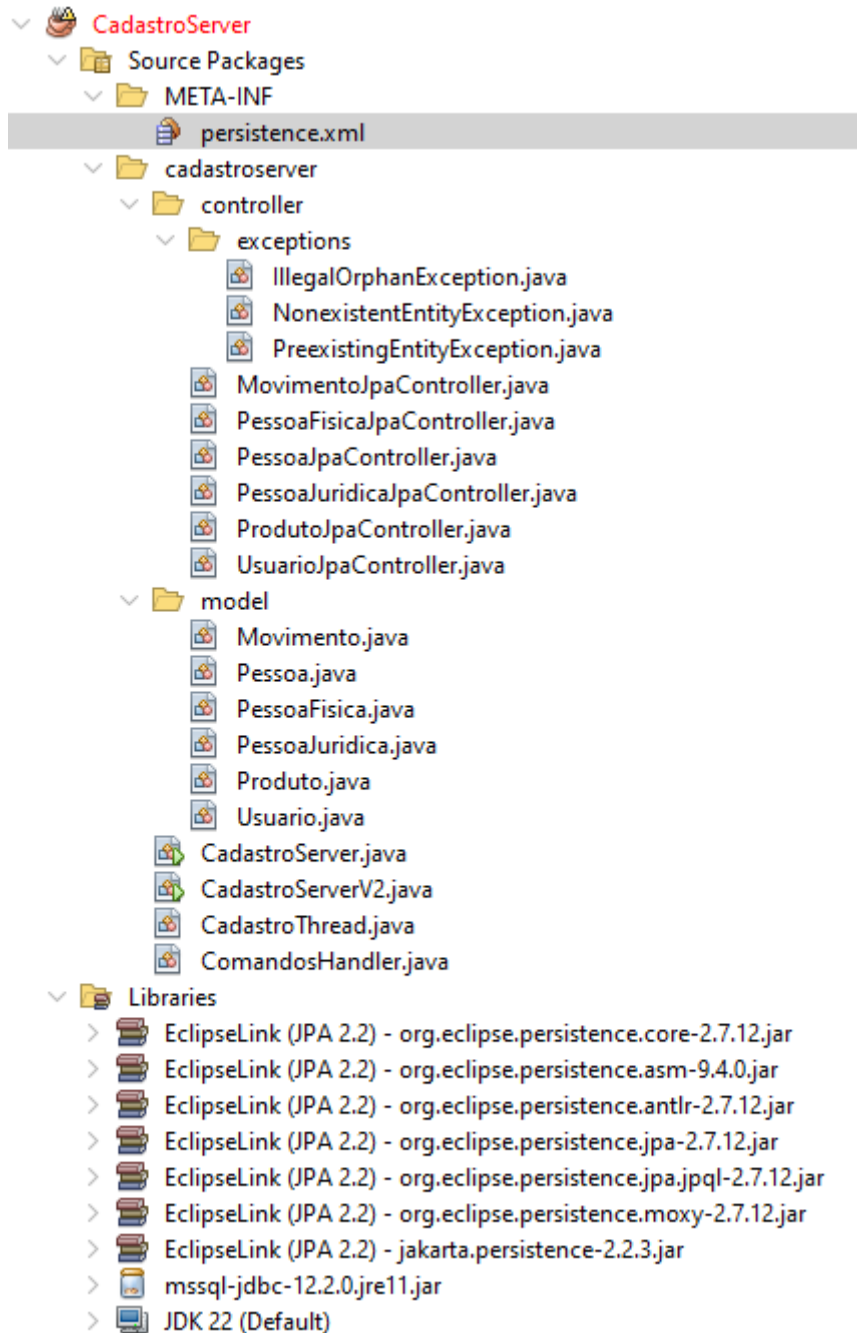


Figura 1. Estrutura das classes e bibliotecas do Projeto CadastroServer.

a) Cliente conecta e envia login e senha.

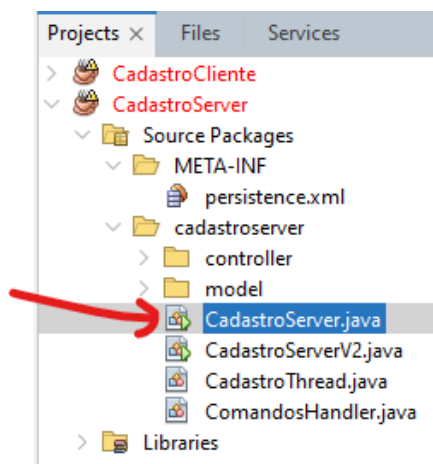


Figura 2. Start da aplicação CadastroServer

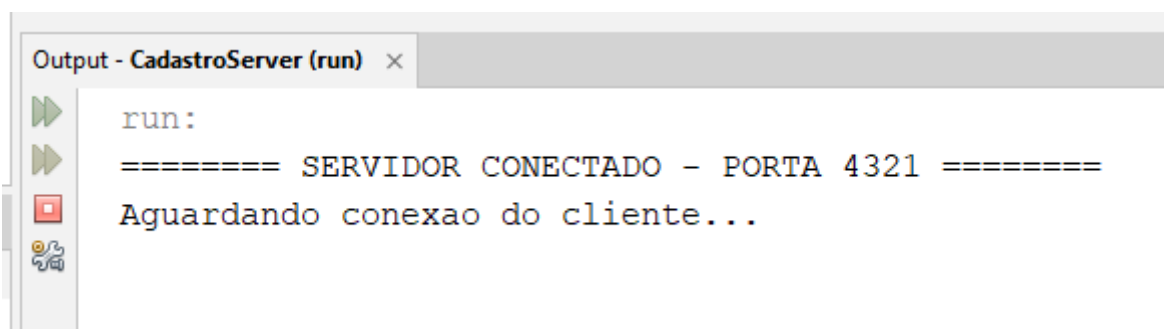


Figura 3. Resultado do Start da aplicação CadastroServer, aguardando Cliente digitar Login e Senha.

b) Servidor valida credenciais e, se forem inválidas, desconecta.

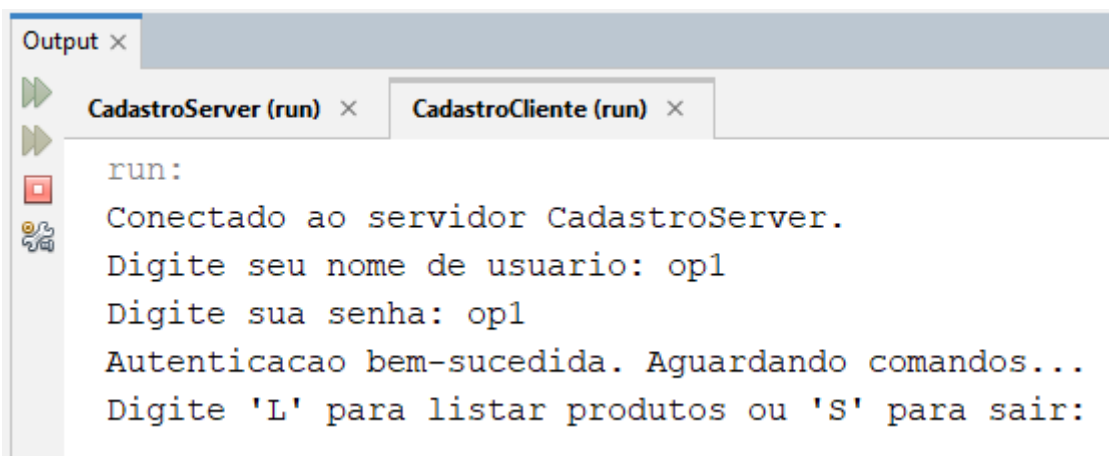
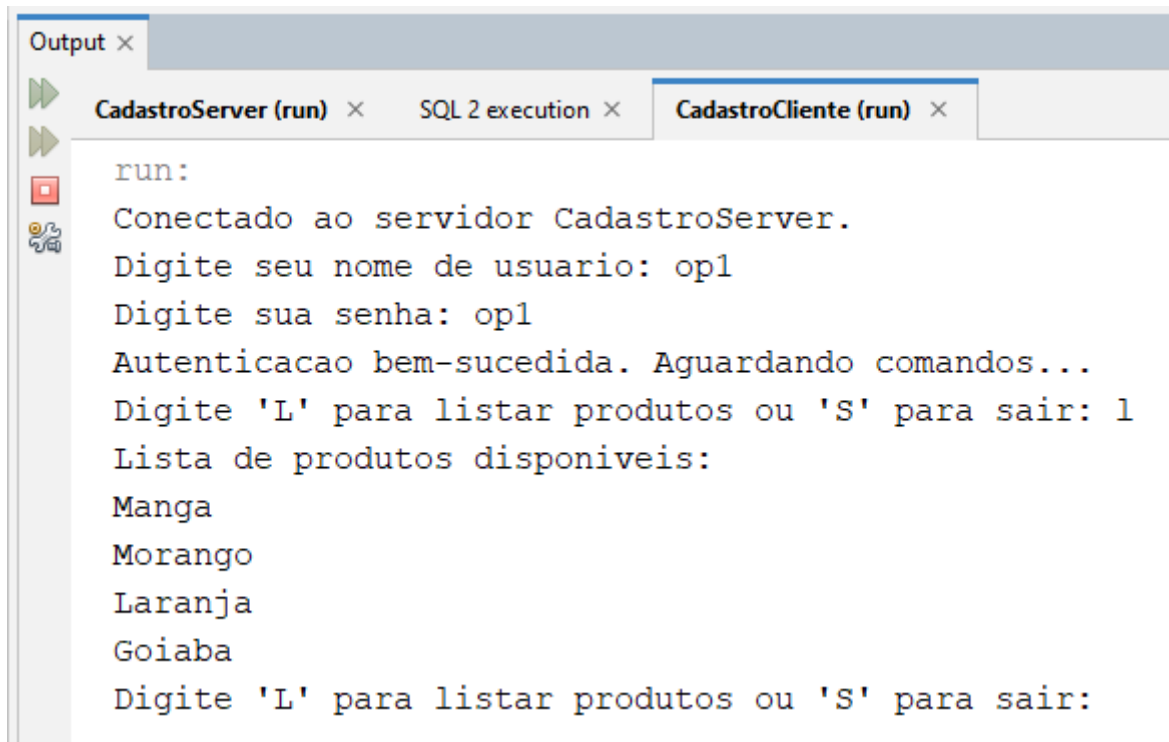


Figura 4. Resultado da validação das credenciais

- c) Com credenciais válidas, fica no ciclo de resposta.
- Cliente envia letra L.
 - Servidor responde com o conjunto de produtos.



```
run:
Conectado ao servidor CadastroServer.
Digite seu nome de usuario: opl
Digite sua senha: opl
Autenticacao bem-sucedida. Aguardando comandos...
Digite 'L' para listar produtos ou 'S' para sair: l
Lista de produtos disponiveis:
Manga
Morango
Laranja
Goiaba
Digite 'L' para listar produtos ou 'S' para sair:
```

Figura 5. Resultado cliente envia letra L, servidor responde listando o conjunto de produtos.

2. Os demais códigos solicitados estão disponíveis no repositório do github do projeto:

<https://github.com/Kakarotox10/Mundo3-MissaoPratica-Nivel5.git>

Análise e Conclusão do 1º Procedimento:

a. Como funcionam as classes Socket e ServerSocket?

Socket e ServerSocket são classes fundamentais em Java para a criação de aplicações de rede. Elas permitem a comunicação entre diferentes programas em máquinas distintas, estabelecendo um canal de comunicação bidirecional.

O que é um Socket?

- **Um ponto de conexão:** Um Socket pode ser visualizado como um ponto de conexão entre duas aplicações, similar a uma tomada elétrica que conecta um aparelho a uma fonte de energia.
- **Duas partes:** Um Socket é composto por um endereço IP e um número de porta, que juntos identificam de forma única uma aplicação em uma rede.
- **Bidirecional:** A comunicação através de um Socket é bidirecional, ou seja, dados podem ser enviados e recebidos pelas duas aplicações conectadas.

O que é um ServerSocket?

- **Um "ouvidor":** Um ServerSocket é como um "ouvidor" que fica aguardando conexões de clientes em uma determinada porta.
- **Criação de Sockets:** Quando um cliente se conecta ao ServerSocket, um novo Socket é criado para gerenciar a comunicação específica com aquele cliente.

Como funciona a comunicação entre Socket e ServerSocket?

1. **Criação do ServerSocket:** O servidor cria um ServerSocket em uma porta específica.
2. **Espera por conexões:** O ServerSocket fica em modo de espera, aguardando que um cliente se conecte.
3. **Conexão do cliente:** Um cliente cria um Socket e tenta se conectar ao ServerSocket do servidor, especificando o endereço IP e a porta.
4. **Aceitação da conexão:** O ServerSocket aceita a conexão e cria um novo Socket para gerenciar a comunicação com o cliente.
5. **Comunicação:** A partir daí, os dois Sockets podem trocar dados através dos seus fluxos de entrada e saída.

Exemplo básico em Java:

```
ExemploSockets.java
1  import java.net.*;
2  import java.io.*;
3
4  public class Servidor {
5      public static void main(String[] args) throws IOException {
6          ServerSocket serverSocket = new ServerSocket(12345); // Cria um ServerSocket na porta 12345
7          System.out.println("Servidor iniciado na porta 12345");
8
9          while (true) {
10             Socket clientSocket = serverSocket.accept(); // Aceita uma nova conexão
11             System.out.println("Novo cliente conectado: " + clientSocket.getInetAddress().getHostAddress());
12
13             // Cria fluxos de entrada e saída para comunicação
14             BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
15             PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
16
17             // Lê a mensagem do cliente
18             String mensagem = in.readLine();
19             System.out.println("Cliente: " + mensagem);
20
21             // Envia uma resposta para o cliente
22             out.println("Servidor: Mensagem recebida!");
23
24             // Fecha os fluxos e o socket
25             in.close();
26             out.close();
27             clientSocket.close();
28         }
29     }
30 }
```

Tipos de aplicações de Socket e ServerSocket:

- **Chat:** Criação de aplicativos de chat em tempo real.
- **Serviços de arquivos:** Servidores de arquivos para transferência de arquivos entre computadores.
- **Serviços web:** A base de muitos serviços web, como HTTP, FTP, etc.
- **Jogos online:** Comunicação entre clientes e servidores em jogos multiplayer.

Conceitos adicionais importantes:

- **Threads:** Para lidar com múltiplos clientes simultaneamente, é comum utilizar threads para cada conexão.
- **Protocolos:** A comunicação através de Sockets geralmente segue um protocolo, como TCP ou UDP, que define as regras de comunicação.
- **Segurança:** A segurança é um aspecto crucial em aplicações de rede. É importante implementar medidas de segurança para proteger a comunicação.

Socket e ServerSocket são ferramentas poderosas para construir aplicações de rede em Java. Entender como elas funcionam é fundamental para desenvolver aplicativos que se comuniquem de forma eficiente e segura.

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

As portas são como os números dos apartamentos em um prédio: elas permitem que diferentes serviços (como web, email, FTP, etc.) coexistam no mesmo endereço IP de um servidor. **Cada porta está associada a um serviço específico**, e quando você se conecta a um servidor, você não apenas especifica o endereço IP, mas também a porta que deseja acessar.

Por que as portas são tão importantes?

- **Organização do tráfego:** Imagine um prédio com vários apartamentos. Sem os números dos apartamentos, como saberíamos para qual apartamento entregar uma carta? As portas fazem o mesmo para o tráfego de rede, direcionando os dados para o serviço correto.
- **Segurança:** As portas permitem que você controle quais serviços estão disponíveis em um servidor. Ao fechar portas não utilizadas, você diminui a superfície de ataque para possíveis invasores.
- **Múltiplos serviços:** Um único servidor pode oferecer vários serviços diferentes, cada um utilizando uma porta distinta. Por exemplo, um servidor web normalmente utiliza a porta 80, enquanto um servidor de email pode utilizar a porta 25.

Como as portas funcionam?

- **Endereço IP + Porta:** Quando você digita um endereço web em seu navegador, você está, na verdade, especificando um endereço IP e a porta 80 (a porta padrão para o protocolo HTTP).
- **Protocolos:** Cada protocolo de rede (HTTP, FTP, SMTP, etc.) utiliza uma porta padrão. No entanto, você pode configurar um servidor para utilizar portas diferentes, se necessário.
- **Firewall:** Um firewall pode ser configurado para permitir ou bloquear o tráfego em determinadas portas, proporcionando uma camada adicional de segurança.

Exemplos de portas comuns:

- **Porta 80:** HTTP (serviços web)
- **Porta 443:** HTTPS (serviços web seguros)
- **Porta 25:** SMTP (envio de emails)
- **Porta 21:** FTP (transferência de arquivos)
- **Porta 22:** SSH (conexões seguras)

As portas são essenciais para a comunicação em redes, permitindo que múltiplos serviços coexistam em um mesmo servidor e que o tráfego de rede seja direcionado de forma eficiente e segura. Ao entender o conceito de portas, você terá uma melhor compreensão de como a internet funciona e como os servidores se comunicam.

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 classes fundamentais em Java para a serialização e desserialização de objetos. Isso significa que elas permitem converter objetos em um formato de bytes para que possam ser transmitidos através de redes, armazenados em arquivos ou simplesmente passados como argumentos para outros métodos.

Serialização

- **O que é:** É o processo de converter um objeto em uma sequência de bytes.
- **Por que:** Permite que objetos sejam transmitidos através de redes, armazenados em arquivos ou passados como argumentos para métodos remotos.
- **Como funciona:** O `ObjectOutputStream` percorre o grafo de objetos, escrevendo cada objeto e suas referências em um fluxo de saída.

Desserialização

- **O que é:** É o processo inverso da serialização, ou seja, reconstruir um objeto a partir de uma sequência de bytes.
- **Como funciona:** O `ObjectInputStream` lê a sequência de bytes e recria os objetos na mesma ordem em que foram serializados.

Por que os objetos devem ser serializáveis?

Para que um objeto possa ser serializado e desserializado, ele precisa implementar a interface `Serializable`. Isso indica ao mecanismo de serialização que o objeto e suas instâncias podem ser convertidas em um fluxo de bytes e restauradas posteriormente.

Motivos:

- **Marca objetos para serialização:** Ao implementar `Serializable`, você está explicitamente dizendo ao compilador que o objeto pode ser serializado.
- **Controle sobre o processo:** A interface `Serializable` não define nenhum método, mas serve como um marcador para o mecanismo de serialização.
- **Mecanismo de serialização padrão:** O mecanismo de serialização padrão do Java percorre recursivamente o grafo de objetos, serializando cada objeto e suas referências.

Quando usar `ObjectInputStream` e `ObjectOutputStream`?

- **Comunicação entre processos:** Para enviar objetos entre diferentes processos em uma máquina ou em máquinas diferentes.
- **Persistência de objetos:** Para salvar o estado de um objeto em um arquivo e restaurá-lo posteriormente.
- **Passagem de objetos por parâmetro:** Para passar objetos complexos como parâmetros para métodos remotos.

Exemplo:


```
SerializacaoExemplo.java 1 X
SerializacaoExemplo.java > SerializacaoExemplo > main(String[])
1  import java.io.*;
2
3
4  class Pessoa implements Serializable {
5      private String nome;
6      private int idade;
7
8  > public Pessoa(String nome, int idade) { ...
12
13 > public String getNome() { ...
16 > public void setNome(String nome) { ...
19 > public int getIdade() { ...
22 > public void setIdade(int idade) { ...
25 }
26
27
28 public class SerializacaoExemplo {
    Run | Debug
29     public static void main(String[] args) throws IOException {
30         Pessoa pessoa = new Pessoa(nome:"João", idade:30);
31
32         // Serialização
33         FileOutputStream fos = new FileOutputStream(name:"pessoa.ser");
34         ObjectOutputStream oos = new ObjectOutputStream(fos);
35         oos.writeObject(pessoa);
36         oos.close();
37
38         // Desserialização
39         FileInputStream fis = new FileInputStream(name:"pessoa.ser");
40         ObjectInputStream ois = new ObjectInputStream(fis);
41         Pessoa pessoaLida = (Pessoa) ois.readObject();
42         ois.close();
43
44         System.out.println(pessoaLida.getNome() + ", " + pessoaLida.getIdade());
45     }
46 }
```

Considerações importantes:

- **Não serializar tudo:** Nem todos os objetos devem ser serializados. Objetos que contêm referências a recursos externos (como sockets ou bancos de dados) podem causar problemas durante a desserialização.
- **Customizar a serialização:** É possível personalizar o processo de serialização usando a interface `Externalizable`.
- **Segurança:** A serialização pode ser um vetor de ataques, como a injeção de objetos. É importante tomar precauções para evitar esses problemas.

ObjectInputStream e ObjectOutputStream são ferramentas poderosas para trabalhar com objetos em Java. Ao entender como a serialização funciona e quais são suas limitações, poderemos criar aplicações mais robustas e flexíveis.

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

A premissa de que as classes de entidades JPA podem garantir o isolamento do acesso ao banco de dados diretamente no cliente é, em geral, incorreta.

Entendendo o Porquê:

- **JPA e o Contexto de Persistência:**
 - O JPA (Java Persistence API) é projetado para gerenciar a persistência de objetos Java em um banco de dados relacional.
 - O coração do JPA é o **contexto de persistência**, um objeto que mantém o estado de um conjunto de entidades e gerencia suas operações de persistência.
 - **O contexto de persistência geralmente reside no servidor**, onde o banco de dados está localizado.
- **Isolamento e Transações:**
 - O isolamento de transações é um mecanismo que garante a consistência dos dados em um banco de dados, mesmo quando múltiplas transações estão sendo executadas simultaneamente.
 - **O isolamento é geralmente implementado no nível do banco de dados** e gerenciado pelo sistema de gerenciamento de banco de dados (SGBD).
 - O JPA fornece mecanismos para definir o nível de isolamento desejado para as transações, mas a execução e o controle das transações ocorrem no contexto do SGBD.
- **Cliente vs. Servidor:**
 - **Cliente:** Normalmente, o cliente possui apenas as classes de entidade mapeadas para as tabelas do banco de dados. Ele não tem controle direto sobre a conexão com o banco de dados ou sobre as transações.
 - **Servidor:** O servidor, por outro lado, possui o contexto de persistência e o gerenciador de entidade, que são responsáveis por interagir com o banco de dados e garantir o isolamento.

Como o Isolamento é Garantido:

1. Contexto de Persistência no Servidor:

- Quando o cliente envia uma requisição para o servidor, o servidor cria um contexto de persistência e carrega as entidades necessárias.

- As operações de persistência (inserção, atualização, exclusão) são realizadas dentro de uma transação gerenciada pelo contexto de persistência.

2. Gerenciamento de Transações:

- O servidor define o nível de isolamento da transação (por exemplo, Read Committed, Repeatable Read, Serializable).
- O SGBD garante que as regras de isolamento sejam respeitadas durante a execução da transação.

3. Comunicação com o Cliente:

- Após a conclusão da transação, o servidor envia os resultados para o cliente.
- O cliente recebe os dados atualizados, mas não tem controle sobre o processo de atualização no banco de dados.

Cenários em que o Cliente Pode Influenciar o Isolamento:

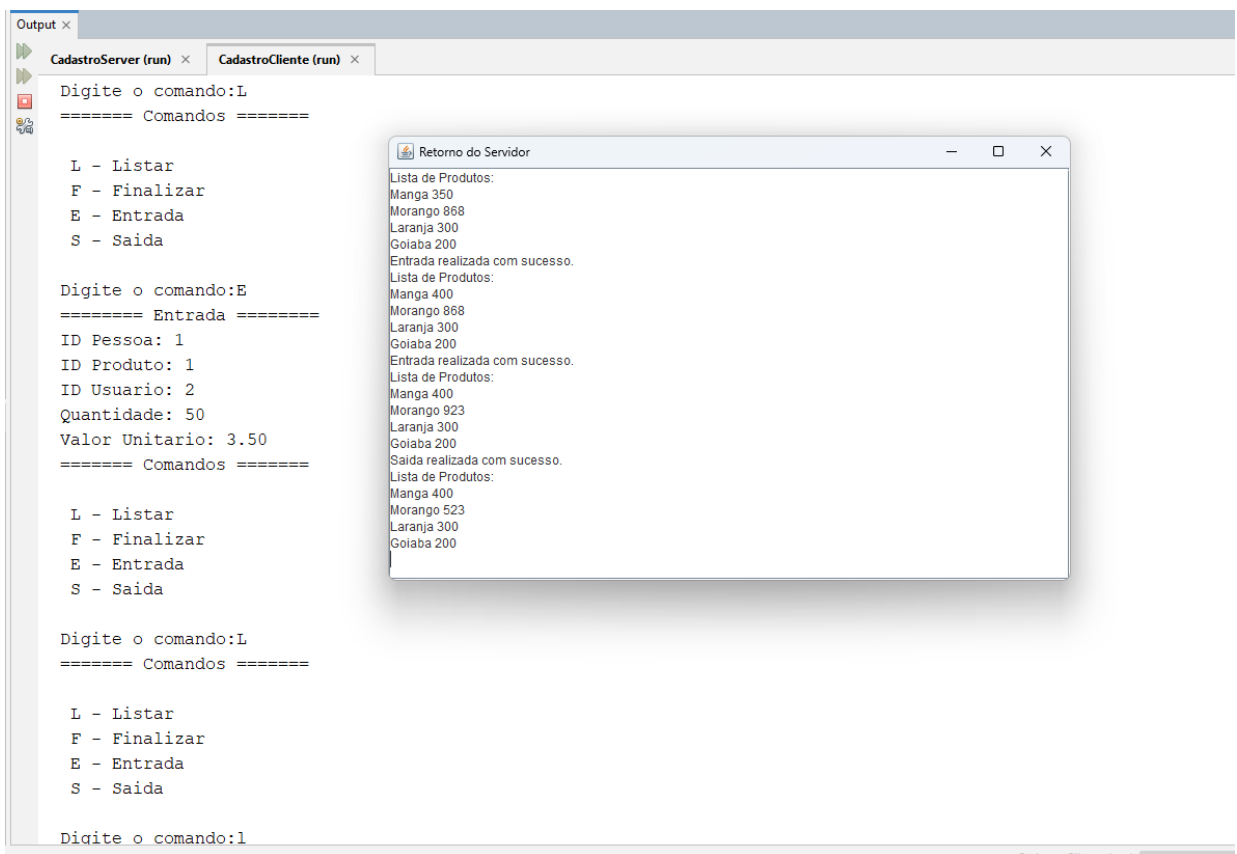
- **Pessimistic Locking:** Em alguns casos, é possível configurar o JPA para utilizar bloqueios pessimistas, onde o sistema de banco de dados bloqueia os dados enquanto uma transação está em andamento. Isso pode afetar o desempenho, mas pode ser necessário em cenários específicos.
- **Optimistic Concurrency Control:** O JPA também suporta o controle de concorrência otimista, onde as alterações são verificadas antes de serem confirmadas no banco de dados. Isso pode ajudar a evitar bloqueios, mas pode levar a conflitos de concorrência.

Embora as classes de entidades JPA sejam utilizadas no cliente, o isolamento do acesso ao banco de dados é garantido principalmente pelo servidor, através do contexto de persistência e do gerenciamento de transações. O cliente apenas interage com o servidor, enviando requisições e recebendo respostas.

O isolamento do acesso ao banco de dados é um conceito fundamental para garantir a integridade dos dados em sistemas multiusuários. Ao entender como o JPA e o servidor trabalham em conjunto para garantir o isolamento, você poderá desenvolver aplicações mais robustas e confiáveis.

2º Procedimento | Servidor Completo e Cliente Assíncrono

1. Criar uma segunda versão da Thread de comunicação, no projeto do servidor, com o acréscimo da funcionalidade apresentada a seguir:
 - a) Servidor recebe comando **E**, para entrada de produtos, ou **S**, para saída.
 - b) Gerar um objeto Movimento, configurado com o usuário logado e o tipo, que pode ser E ou S.
 - c) Receber o Id da pessoa e configurar no objeto Movimento.
 - d) Receber o Id do produto e configurar no objeto Movimento.
 - e) Receber a quantidade e configurar no objeto Movimento.
 - f) Receber o valor unitário e configurar no objeto Movimento.
 - g) Persistir o movimento através de um MovimentoJpaController com o nome ctrlMov.
 - h) Atualizar a quantidade de produtos, de acordo com o tipo de movimento, através de ctrlProd.
2. Os Códigos solicitados neste quesito estão disponíveis no repositório do github:
<https://github.com/Kakarotox10/Mundo3-MissaoPratica-Nivel5.git>
3. Resultado da Execução dos códigos:



```
Output x
CadastroServer (run) x CadastroCliente (run) x

Digite o comando:L
===== Comandos =====

L - Listar
F - Finalizar
E - Entrada
S - Saída

Digite o comando:E
===== Entrada =====
ID Pessoa: 1
ID Produto: 1
ID Usuario: 2
Quantidade: 50
Valor Unitario: 3.50
===== Comandos =====

L - Listar
F - Finalizar
E - Entrada
S - Saída

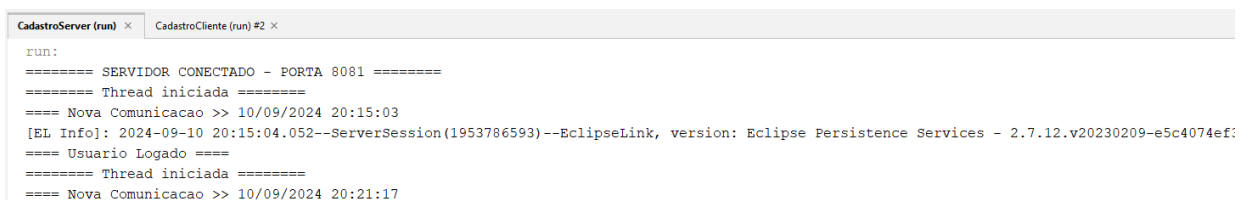
Digite o comando:L
===== Comandos =====

L - Listar
F - Finalizar
E - Entrada
S - Saída

Digite o comando:l

Retorno do Servidor
Lista de Produtos:
Manga 350
Morango 868
Laranja 300
Goiaba 200
Entrada realizada com sucesso.
Lista de Produtos:
Manga 400
Morango 868
Laranja 300
Goiaba 200
Entrada realizada com sucesso.
Lista de Produtos:
Manga 400
Morango 923
Laranja 300
Goiaba 200
Saída realizada com sucesso.
Lista de Produtos:
Manga 400
Morango 523
Laranja 300
Goiaba 200
```

Figura 6. Classe CadastroClienteV2 em Execução, com Exemplos de Movimentação (Entrada/Saída)



```
CadastroServer (run) x CadastroCliente (run) #2 x

run:
===== SERVIDOR CONECTADO - PORTA 8081 =====
===== Thread iniciada =====
==== Nova Comunicacao >> 10/09/2024 20:15:03
[EL Info]: 2024-09-10 20:15:04.052--ServerSession(1953786593)--EclipseLink, version: Eclipse Persistence Services - 2.7.12.v20230209-e5c4074ef:
==== Usuario Logado ====
===== Thread iniciada =====
==== Nova Comunicacao >> 10/09/2024 20:21:17
```

Figura 7. Classe CadastroClienteV2 em Execução.

Análise e Conclusão do 2º Procedimento:

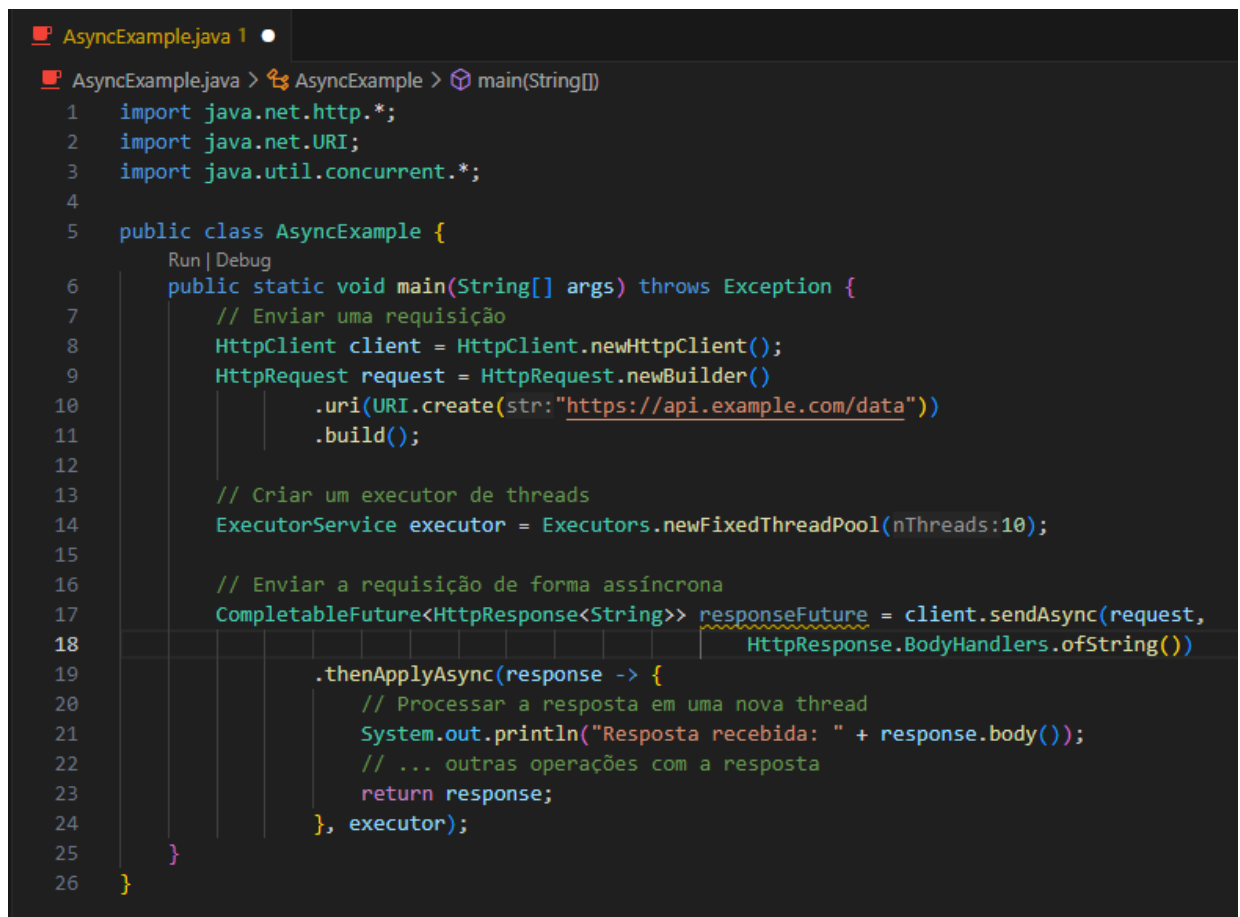
a) Como as Threads podem ser utilizadas para o tratamento assíncrono das respostas enviadas pelo servidor?

Em Java, threads são unidades de execução independentes dentro de um processo. Cada thread possui seu próprio contador de programa e pilha, permitindo que múltiplas tarefas sejam executadas concorrentemente.

Como as Threads são utilizadas para o tratamento assíncrono?

- **Tarefas Independentes:** Ao receber uma resposta do servidor, uma nova thread pode ser criada para processar essa resposta. Enquanto isso, a thread principal pode continuar com outras tarefas, como enviar novas requisições ou atualizar a interface do usuário.
- **Evita Bloqueios:** Se o processamento da resposta levar algum tempo, a thread principal não será bloqueada, evitando que a aplicação fique lenta ou unresponsive.
- **Melhora a Responsividade:** Ao distribuir as tarefas entre múltiplas threads, a aplicação como um todo se torna mais responsiva, pois diferentes partes do código podem ser executadas em paralelo.

Exemplo Prático:



```
1  import java.net.http.*;
2  import java.net.URI;
3  import java.util.concurrent.*;
4
5  public class AsyncExample {
6      public static void main(String[] args) throws Exception {
7          // Enviar uma requisição
8          HttpClient client = HttpClient.newHttpClient();
9          HttpRequest request = HttpRequest.newBuilder()
10             .uri(URI.create(str:"https://api.example.com/data"))
11             .build();
12
13          // Criar um executor de threads
14          ExecutorService executor = Executors.newFixedThreadPool(nThreads:10);
15
16          // Enviar a requisição de forma assíncrona
17          CompletableFuture<HttpResponse<String>> responseFuture = client.sendAsync(request,
18             HttpResponse.BodyHandlers.ofString())
19             .thenApplyAsync(response -> {
20                 // Processar a resposta em uma nova thread
21                 System.out.println("Resposta recebida: " + response.body());
22                 // ... outras operações com a resposta
23                 return response;
24             }, executor);
25      }
26  }
```

Explicação do Código da figura acima:

- **CompletableFuture:** É usado para representar o resultado de uma operação assíncrona.
- **thenApplyAsync:** Executa uma função (lambda) quando o resultado estiver disponível, criando uma nova thread para isso.
- **ExecutorService:** Gere threads para executar as tarefas.

Vantagens do Tratamento Assíncrono:

- **Melhor desempenho:** Permite que a aplicação continue a responder a outras solicitações enquanto aguarda respostas do servidor.
- **Maior escalabilidade:** Facilita o tratamento de um grande número de requisições simultâneas.
- **Código mais limpo:** Separa as diferentes tarefas em funções distintas, tornando o código mais organizado e fácil de entender.

Considerações Importantes:

- **Gerenciamento de Threads:** É fundamental gerenciar as threads corretamente para evitar vazamentos de memória e deadlocks.
- **Sincronização:** Se múltiplas threads acessarem os mesmos dados, é necessário utilizar mecanismos de sincronização para evitar problemas de concorrência.
- **Overhead:** A criação e gerenciamento de threads envolvem um certo overhead, por isso é importante avaliar o custo-benefício em cada caso.

Em resumo, as threads em Java são uma ferramenta poderosa para implementar o tratamento assíncrono de respostas do servidor, melhorando o desempenho, a responsividade e a escalabilidade das aplicações.

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

O método **`invokeLater`** da classe **`SwingUtilities`** é utilizado na programação de interfaces gráficas em Java, especificamente no Swing framework.

Este método é importante para garantir que as alterações na interface do usuário sejam feitas de maneira segura no contexto da Event Dispatch Thread (EDT), que é a thread responsável por gerenciar eventos e atualizações de interface gráfica em Swing.

Quando se deseja executar um bloco de código que altera a interface do usuário, como atualizar um componente gráfico ou responder a eventos de usuário, `invokeLater` é utilizado para enfileirar esse bloco de código na EDT. Isto assegura que as mudanças na interface sejam realizadas de maneira sequencial e segura, assim evita problemas de concorrência e inconsistências na interface gráfica.

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

Sockets em Java são pontos de extremidade de uma comunicação bidirecional entre processos, permitindo que diferentes programas em máquinas diferentes se comuniquem. Ao lidar com a troca de dados complexos, como objetos, é necessário serializar esses objetos em um formato que possa ser transmitido pela rede e depois desserializado no outro lado.

A comunicação de objetos via Socket em Java envolve a serialização, transmissão e desserialização. A escolha do método de serialização dependerá das suas necessidades específicas, como performance, segurança e compatibilidade.

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

Comparação entre Comportamento Assíncrono e Síncrono em Clientes com Socket Java

Comportamento Síncrono:

- **Funcionamento:**
 - Cada operação de entrada/saída (E/S) realizada pelo socket bloqueia a thread até que a operação seja concluída.
 - Isso significa que, enquanto uma thread está esperando por uma resposta do servidor, ela não pode realizar outras tarefas.
- **Características:**
 - **Simples de implementar:** O modelo síncrono é mais fácil de entender e implementar, pois o fluxo de execução é linear e sequencial.
 - **Bloqueio:** A thread que realiza a operação de E/S fica bloqueada até que a resposta seja recebida ou ocorra um erro.
 - **Menor consumo de recursos:** Geralmente, o modelo síncrono utiliza menos recursos do sistema, pois não requer a criação e gerenciamento de múltiplas threads.
- **Cenários de uso:**
 - Aplicações simples que não requerem alta concorrência ou responsividade.
 - Quando a ordem das operações é crítica e não se deseja lidar com a complexidade do paralelismo.

Comportamento Assíncrono:

- **Funcionamento:**
 - As operações de E/S são iniciadas e a thread é liberada para realizar outras tarefas.
 - Quando a operação é concluída, um mecanismo de notificação (como callbacks ou futures) é utilizado para informar a thread sobre o resultado.
- **Características:**
 - **Alta concorrência:** Permite que a aplicação processe múltiplas requisições simultaneamente, melhorando a responsividade.
 - **Não bloqueante:** A thread principal não fica bloqueada enquanto espera por uma resposta, podendo realizar outras tarefas.
 - **Complexidade:** Requer um gerenciamento mais cuidadoso das threads e dos mecanismos de sincronização para evitar problemas de concorrência.
 - **Maior consumo de recursos:** A criação e gerenciamento de múltiplas threads podem aumentar o consumo de memória e CPU.
- **Cenários de uso:**

- Aplicações com alta carga de trabalho e que exigem alta responsividade.
- Servidores que precisam atender a um grande número de clientes simultaneamente.
- Aplicações que realizam operações de E/S demoradas, como downloads de arquivos grandes.

Quando usar cada abordagem?

- **Síncrono:** Ideal para aplicações simples, onde a ordem das operações é crítica e a performance não é um fator determinante.
- **Assíncrono:** Indicado para aplicações com alta carga de trabalho, que precisam ser altamente responsivas e escaláveis.

A escolha entre o comportamento síncrono e assíncrono depende das características da sua aplicação e dos requisitos de desempenho. O modelo assíncrono oferece maior escalabilidade e responsividade, mas exige um cuidado maior na implementação. O modelo síncrono é mais simples de entender e implementar, mas pode limitar a performance da aplicação em cenários de alta carga.

Fatores a considerar na escolha:

- **Carga de trabalho:** Alta carga indica a necessidade de assincronismo.
- **Responsividade:** Aplicações que exigem alta responsividade se beneficiam do assincronismo.
- **Complexidade:** O assincronismo pode tornar o código mais complexo.
- **Recursos:** O assincronismo pode consumir mais recursos do sistema.

Ao escolher o modelo adequado, poderemos desenvolver aplicações mais eficientes e escaláveis.