



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

MISSÃO PRÁTICA

Nível 5

OBJETIVO

Este trabalho visa a criação de um sistema robusto e eficiente, baseado em um servidor Java que utiliza Sockets para comunicação com clientes, JPA para acesso a bancos de dados e Threads para processamento paralelo.

O objetivo principal é desenvolver um sistema que permita a comunicação simultânea de múltiplos clientes, garantindo a integridade e segurança dos dados armazenados no banco de dados. Para isso, serão exploradas as funcionalidades das classes Socket e ServerSocket, bem como a implementação de clientes síncronos e assíncronos, utilizando Threads para otimizar o desempenho e a responsividade do sistema.

CÓDIGOS

A seguir, apresentamos os códigos dos procedimentos realizados neste trabalho.

Destaque-se que este trabalho envolveu a criação de 3 (dois) projetos Java: o projeto Servidor, o projeto Cliente e o projeto ClienteV2.

Projeto Servidor

O projeto Servidor é destinado ao processamento das requisições do lado do servidor.



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

No lado do servidor, além do pacote com as classes de thread e da função main, temos pacotes da camada model, responsável por processar as informações, e controller, responsável por intermediar as requisições que chegam ao servidor e se destinam à camada model. Os códigos das camadas model e controller foram gerados com o auxílio da IDE NetBeans.

Classe CadastroThread

A classe CadastroThread estende a classe Thread, nativa do Java, e se destina a tratar as threads no servidor, permitindo que vários clientes possam se conectar e solicitar o serviço.

```
package servidor;

import controller.MovimentoJpaController;
import controller.PessoaFisicaJpaController;
import controller.ProdutoJpaController;
import controller.UsuarioJpaController;
import controller.exceptions.NonexistentEntityException;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import model.Movimento;
import model.Pessoa;
import model.Produto;
import model.Usuario;

public class CadastroThread extends Thread {
```



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

```
private ProdutoJpaController ctrl;  
private UsuarioJpaController ctrlUsu;  
private MovimentoJpaController ctrlMov;  
private PessoaFisicaJpaController ctrlPessoa;  
private Socket s1;  
  
    public CadastroThread(ProdutoJpaController ctrl,  
UsuarioJpaController ctrlUsu, MovimentoJpaController ctrlMov,  
PessoaFisicaJpaController ctrlPessoa, Socket s1) {  
    this.ctrl = ctrl;  
    this.ctrlUsu = ctrlUsu;  
    this.ctrlMov = ctrlMov;  
    this.ctrlPessoa = ctrlPessoa;  
    this.s1 = s1;  
}  
  
    public CadastroThread(ProdutoJpaController ctrl,  
UsuarioJpaController ctrlUsu, Socket s1) {  
    this.ctrl = ctrl;  
    this.ctrlUsu = ctrlUsu;  
    this.s1 = s1;  
}  
  
@Override  
public void run() {  
    try (  
        // A ordem primeiro ObjectOutputStream e depois  
ObjectInputStream evita deadlocks.  
        ObjectOutputStream saida = new  
ObjectOutputStream(s1.getOutputStream());  
        ObjectInputStream entrada = new  
ObjectInputStream(s1.getInputStream());  
    ) {
```



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

```
// Obter login e senha.
String login = (String) entrada.readObject();
String senha = (String) entrada.readObject();

// Verificar login.
Usuario usuario = ctrlUsu.findUsuario(login, senha);
if (usuario == null) {
    saida.writeObject("Login inválido.");

    return; // Termina a conexão.
}

// Loop de resposta para comandos.
while (true) {
    String comando = (String) entrada.readObject();

    // Se o comando fo "L", listar os produtos.
    if ("L".equalsIgnoreCase(comando)) {
        List<Produto> produtos =
ctrl.findProdutoEntities();
        saida.writeObject(produtos);
    }
    else if ("E".equalsIgnoreCase(comando) ||
"S".equalsIgnoreCase(comando)) {
        // Gerar um objeto Movimento.
        Movimento movimento = new Movimento();
        movimento.setIdUsuario(usuario);
        movimento.setTipoMovimento(comando);

        Pessoa pessoa = (Pessoa) entrada.readObject();
        movimento.setIdPessoa(pessoa);
    }
}
```



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

```

                                Produto idProduto = (Produto)
entrada.readObject();
                                Produto produto =
ctrl.findProduto(idProduto.getIdProduto());
                                movimento.setIdProduto(idProduto);

                                int quantidade = (int) entrada.readObject();
                                movimento.setQuantidade(quantidade);

                                double valorUnitario = (double)
entrada.readObject();
                                movimento.setValorUnitario(valorUnitario);

                                // Persistir o movimento.
                                ctrlMov.create(movimento);

                                // Atualizar estoque.
                                if (comando.equals("E")) {
                                    produto.setQuantidade(produto.getQuantidad
e() + quantidade);
                                }
                                else {
                                    produto.setQuantidade(produto.getQuantidad
e() - quantidade);
                                }

                                ctrl.edit(produto);

                                saida.writeObject("Movimento registrado com
sucesso!");
                                }
                                else {
                                    saida.writeObject("Comando inválido.");
                                }

```

```
        }  
    }  
    }  
    catch (IOException | ClassNotFoundException e) {  
        System.err.println("Erro na conexão: " +  
e.getMessage());  
    } catch (NonexistentEntityException ex) {  
        Logger.getLogger(CadastroThread.class.getName()).log(L  
evel.SEVERE, null, ex);  
    } catch (Exception ex) {  
        Logger.getLogger(CadastroThread.class.getName()).log(L  
evel.SEVERE, null, ex);  
    }  
    finally {  
        try {  
            s1.close();  
        }  
        catch (IOException e) {  
            System.err.println("Erro ao fechar conexão: " +  
e.getMessage());  
        }  
    }  
}  
}
```

Classe Servidor

Na classe Servidor se encontra a função main, responsável por dar início à aplicação no servidor.

Aqui o servidor permanecerá escutando uma porta determinada, esperando uma solicitação.

```
package servidor;
```



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

```
import controller.MovimentoJpaController;
import controller.PessoaFisicaJpaController;
import controller.ProdutoJpaController;
import controller.UsuarioJpaController;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class Servidor {

    public static void main(String[] args) {

        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("ServidorPU");
        ProdutoJpaController ctrl = new ProdutoJpaController(emf);
        UsuarioJpaController ctrlUsu = new
UsuarioJpaController(emf);
        MovimentoJpaController ctrlMov = new
MovimentoJpaController(emf);
        PessoaFisicaJpaController ctrlPessoa = new
PessoaFisicaJpaController(emf);

        try (ServerSocket serverSocket = new ServerSocket(4321)) {
            System.out.println("Servidor aguardando conexões na
porta 4321...");

            while (true) {
                // Aguardando conexão com o cliente.
                Socket socket = serverSocket.accept();

                CadastroThread cadastroThread = new
CadastroThread(ctrl, ctrlUsu, ctrlMov, ctrlPessoa, socket);
```



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

```
// Inicia um nova thread diante de uma requisição
de um cliente.
        cadastroThread.start();
    }
}
catch (IOException e) {
    System.err.println("Erro na conexão: " +
e.getMessage());
}
}
```

Projeto Cliente

O projeto Cliente é destinado a tratar da realização de requisições por parte do cliente, sem a utilização de threads, sendo utilizado primariamente para testar a resposta do servidor.

Classe Cliente

Na classe Cliente se encontra a função main, responsável por dar início à aplicação do cliente.

A aplicação cliente irá apenas enviar informações que serão processadas pelo servidor e validadas junto ao banco de dados.

```
package cliente;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.Collection;
```




Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

```
import model.Produto;

public class Cliente {

    public static void main(String[] args) {

        try (

            // Socket apontando para "localhost:4321".
            Socket socket = new Socket("localhost", 4321);

            // Encapsular os canais de entrada e saída.
            ObjectOutputStream saida = new
ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream entrada = new
ObjectInputStream(socket.getInputStream());
        ) {

            // Login e senha utilizando os dados da tabela
            Usuarios.

            String login = "op1";
            String senha = "op1";

            // Escrever o login e a senha na saída
            saida.writeObject(login);
            saida.writeObject(senha);
            saida.flush();

            System.out.println("Usuário conectado com sucesso!");

            // Enviar o comando L no canal de saída.
            saida.writeObject("L");
            saida.flush();

            // Receber a coleção de entidades no canal de entrada.
```



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

```
        Object reposta = entrada.readObject();

        // Processar a resposta recebida.
        if (reposta instanceof Collection<?>) {
            Collection<?> produtos = (Collection<Object>)
reposta;
            for (Object produto : produtos) {
                if (produto instanceof Produto) {
                    System.out.println("Produto: " +
((Produto) produto).getNome());
                }
            }
        }
        catch (IOException e) {
            System.err.println("Erro na conexão: " +
e.getMessage());
        }
        catch (ClassNotFoundException e) {
            System.err.println("Erro ao obter dados: " +
e.getMessage());
        }
    }
}
```

Projeto ClienteV2

O projeto Cliente é destinado a tratar da realização de requisições por parte do cliente, com a utilização de threads.



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

Classe ThreadClient

A classe ThreadClient estende a classe Thread, nativa do Java, e se destina a tratar as threads no lado do cliente, permitindo que a aplicação não fique congelada enquanto aguarda uma resposta do servidor.

```
package clientev2;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.util.List;
import javax.swing.JTextArea;
import javax.swing.SwingUtilities;
import model.Produto;

public class ThreadCliente extends Thread {

    private ObjectInputStream entrada;
    private JTextArea textArea;

    public ThreadCliente(ObjectInputStream entrada, JTextArea
textArea) {
        this.entrada = entrada;
        this.textArea = textArea;
    }

    @Override
    public void run() {

        try {
            while (true) {
                Object mensagem = entrada.readObject();

                if (mensagem instanceof String) {
```



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

```
SwingUtilities.invokeLater(() ->
textArea.append((String) mensagem + "\n"));
    }
    else if (mensagem instanceof List) {
        SwingUtilities.invokeLater(() -> {
            for (Object item : (List<?>) mensagem) {
                if (item instanceof Produto) {
                    Produto produto = (Produto) item;
                    textArea.append(produto.getNome()
+ " - Quantidade: " + produto.getQuantidade() + "\n");
                }
            }
        });
    }
}
}
catch (IOException | ClassNotFoundException e) {
    System.err.println("Erro na thread cliente: " +
e.getMessage());
    SwingUtilities.invokeLater(() ->
textArea.append("Conexão com servidor perdida."));
}
}
}
```

Classe SaidaFrame

A classe SaidaFrame implementará uma simples interface gráfica para receber as respostas do servidor.

```
package clientev2;

import javax.swing.JDialog;
import javax.swing.JTextArea;
```



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

```
public class SaidaFrame extends JDialog {

    public JTextArea texto;

    public SaidaFrame() {
        texto = new JTextArea();
        setBounds(100, 100, 400, 300);
        setModal(false);
        getContentPane().add(texto);
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        setTitle("Mensagens do servidor");
    }
}
```

Classe ClienteV2

Na classe ClienteV2 se encontra a função main, responsável por dar início à aplicação do cliente.

A aplicação cliente irá apenas enviar informações que serão processadas pelo servidor e validadas junto ao banco de dados.

```
package clientev2;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;

public class ClienteV2 {
```



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

```
public static void main(String[] args) {

    try {
        // Socket apontando para "localhost:4321".
        Socket socket = new Socket("localhost", 4321);

        // Encapsular os canais de entrada e saída.
        ObjectOutputStream saida = new
ObjectOutputStream(socket.getOutputStream());
        ObjectInputStream entrada = new
ObjectInputStream(socket.getInputStream());

        // Encapsular a leitura do teclado.
        BufferedReader teclado = new BufferedReader(new
InputStreamReader(System.in));

        // Login e senha utilizando os dados da tabela
        Usuarios.

        String login = "op1";
        String senha = "op1";

        // Escrever o login e a senha na saída
        saida.writeObject(login);
        saida.writeObject(senha);
        saida.flush();

        System.out.println("Usuário conectado com sucesso!");

        // Instanciar janela e thread assíncrona
        SaidaFrame saidaFrame = new SaidaFrame();
        saidaFrame.setVisible(true);

        ThreadCliente threadCliente = new
ThreadCliente(entrada, saidaFrame.texto);
```



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

```
threadCliente.start();

String comando;
do {
    System.out.println("\nMENU:");
    System.out.println("L - Lista | X - Finalizar | E
- Entrada | S - Saída");
    System.out.print("Opção: ");
    comando = teclado.readLine().toUpperCase();

    saida.writeObject(comando);

    switch (comando) {
        case "E":
        case "S":
            System.out.print("ID da pessoa: ");
            saida.writeObject(Integer.parseInt(teclado
.readLine()));
            System.out.print("ID do produto: ");
            saida.writeObject(Integer.parseInt(teclado
.readLine()));
            System.out.print("Quantidade: ");
            saida.writeObject(Integer.parseInt(teclado
.readLine()));
            System.out.print("Valor unitário: ");
            saida.writeObject(Double.parseDouble(tecla
do.readLine()));
            break;
        case "L":
            saida.writeObject("L");
            saida.flush();
            break;
        case "X":
```



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

```
                                System.out.println("Encerrando a
conexão.");
                                break;
                                default:
                                System.out.println("Comando inválido.");
                                break;
                                }

                                } while (!comando.equalsIgnoreCase("X"));

                                teclado.close();
                                entrada.close();
                                saida.close();
                                socket.close();
                                } catch (IOException | NumberFormatException e) {
                                System.err.println("Erro: " + e.getMessage());
                                }
                                }
}
```

1º PROCEDIMENTO | CRIAÇÃO DAS ENTIDADES E SISTEMA DE PERSISTÊNCIA

ANÁLISE E CONCLUSÃO

a) Como funcionam as classes Socket e ServerSocket?

ServerSocket:

- Essa classe é usada no lado do servidor. Ela cria um "ponto de escuta" em uma porta específica do servidor.



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

- Sua principal função é aguardar conexões de clientes. Quando um cliente tenta se conectar, o `ServerSocket` aceita a conexão e cria um objeto `Socket` para representar essa conexão.
- O `ServerSocket` age como um porteiro, controlando quem pode entrar no servidor.

Socket:

- Essa classe representa uma conexão entre o cliente e o servidor. Tanto o cliente quanto o servidor usam objetos `Socket` para se comunicar.
- Um objeto `Socket` permite que os dados sejam enviados e recebidos entre as duas partes da conexão.
- É através do `Socket` que se estabelece o canal de comunicação, permitindo o fluxo de dados entre as aplicações.

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

1. Identificação de serviços

- As portas são números que identificam serviços específicos em um servidor. Por exemplo, o HTTP geralmente usa a porta 80, e o HTTPS usa a porta 443.
- No contexto de um servidor `Socket`, a porta define onde o servidor está escutando as conexões dos clientes.

2. Comunicação direcionada

- As portas permitem que múltiplos serviços rodem no mesmo servidor sem conflitos. Cada serviço pode usar uma porta diferente.
- Quando um cliente se conecta a um servidor, ele precisa especificar a porta correta para se comunicar com o serviço desejado.

3. Segurança

- Portas bem configuradas ajudam a proteger o servidor, limitando o acesso a serviços específicos apenas para clientes autorizados.



CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE
CURSO: Desenvolvimento Full Stack
DISCIPLINA: Por Que Não Paralelizar
TURMA: 9001
SEMESTRE: 2024.4
ALUNO: Daniel Kilzer Brasil Dias

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`:

- Essas classes permitem que objetos Java sejam enviados e recebidos através de um fluxo de dados, como uma conexão Socket.
- `ObjectOutputStream` pega um objeto Java e o transforma em uma sequência de bytes que pode ser enviada pela rede.
- `ObjectInputStream` pega a sequência de bytes recebida e a reconstrói em um objeto Java.

Serialização:

- A serialização é o processo de converter um objeto Java em uma sequência de bytes.
- Os objetos transmitidos devem ser serializáveis porque a rede só pode transmitir dados em formato de bytes.
- A serialização garante que o objeto possa ser reconstruído corretamente no lado receptor.
- Para serializar um objeto, a classe do objeto deve implementar a interface `Serializable`.

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

1. Separação de responsabilidades

- As classes de entidades JPA (como `@Entity`) são usadas para mapear tabelas do banco de dados para objetos Java.



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

- No entanto, o acesso ao banco de dados é controlado pelo servidor, não pelo cliente. O cliente apenas envia solicitações ao servidor, que processa essas solicitações e acessa o banco de dados conforme necessário.

2. Transmissão de dados, não de acesso direto

- O cliente pode receber objetos que representam entidades JPA (por exemplo, um objeto `Usuario`), mas esses objetos são apenas cópias dos dados, não têm conexão direta com o banco de dados.
- O servidor é o único que interage diretamente com o banco de dados, garantindo que o cliente não tenha acesso físico ao banco.

3. Segurança e controle

- O servidor pode implementar regras de negócio, validações e controles de acesso antes de acessar o banco de dados.
- O cliente não tem permissão para executar operações diretamente no banco, o que previne vulnerabilidades como injeção de SQL ou acesso não autorizado.

4. Serialização e desserialização

- As entidades JPA são serializadas pelo servidor e enviadas ao cliente como objetos simples. O cliente não tem acesso ao contexto de persistência (como o `EntityManager`), o que garante o isolamento.

2º PROCEDIMENTO | CRIAÇÃO DO CADASTRO EM MODO TEXTO

ANÁLISE E CONCLUSÃO

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



CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE
CURSO: Desenvolvimento Full Stack
DISCIPLINA: Por Que Não Paralelizar
TURMA: 9001
SEMESTRE: 2024.4
ALUNO: Daniel Kilzer Brasil Dias

Em uma aplicação cliente-servidor, é comum que o cliente envie uma solicitação ao servidor e aguarde uma resposta. Se esse processo for realizado de maneira síncrona, o cliente ficará bloqueado até receber a resposta, o que pode afetar a responsividade da aplicação, especialmente em interfaces gráficas. Para evitar esse bloqueio, é possível utilizar threads para tratar as respostas de forma assíncrona.

No cliente:

- Thread de leitura: uma thread separada pode ser criada para aguardar e processar as respostas do servidor. Dessa forma, a thread principal do cliente permanece livre para outras tarefas, como atualizar a interface do usuário ou enviar novas solicitações.

No servidor:

- Thread por cliente: o servidor pode criar uma nova thread para cada conexão de cliente. Isso permite que múltiplos clientes sejam atendidos simultaneamente, garantindo que a comunicação com um cliente não bloqueie a comunicação com outros.

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

O método `invokeLater` da classe `SwingUtilities` é utilizado para garantir que um determinado trecho de código seja executado na Event Dispatch Thread (EDT), que é a thread responsável por manipular componentes gráficos no Swing. Como o Swing não é thread-safe, todas as atualizações na interface gráfica devem ocorrer na EDT para evitar condições de corrida e outros problemas de concorrência. O `invokeLater` recebe um objeto `Runnable` e o coloca na fila de eventos para ser executado assim que possível na EDT.

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



CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE
CURSO: Desenvolvimento Full Stack
DISCIPLINA: Por Que Não Paralelizar
TURMA: 9001
SEMESTRE: 2024.4
ALUNO: Daniel Kilzer Brasil Dias

Para enviar um objeto, usa-se a classe `ObjectOutputStream`. O objeto deve implementar a interface `Serializable`. Já para receber um objeto, usa-se a classe `ObjectInputStream`.

A classe `Serializable` serve para lidar com a serialização dos objetos. Na origem, o objeto é convertido em uma sequência de bytes (serialização) para ser transmitido pela rede; no destino, os bytes são convertidos de volta em um objeto (desserialização).

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

Comportamento Síncrono:

- **Bloqueio de Processamento:** Ao realizar operações de leitura ou escrita em um Socket de forma síncrona, a thread atual fica bloqueada até que a operação seja concluída. Isso significa que, durante a comunicação com o servidor, a aplicação pode ficar inativa, aguardando a resposta.
- **Simplicidade:** A programação síncrona é geralmente mais simples de implementar e entender, pois segue um fluxo linear de execução.
- **Responsividade:** Em aplicações com interface gráfica ou que necessitam de alta responsividade, o bloqueio causado por operações síncronas pode degradar a experiência do usuário.

Comportamento Assíncrono:

- **Não Bloqueante:** Ao utilizar threads separadas para operações de E/S, a aplicação principal continua executando outras tarefas enquanto aguarda a conclusão da comunicação. Isso é especialmente útil em aplicações que requerem alta responsividade.



Estácio

CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

- **Complexidade:** A programação assíncrona introduz complexidade adicional, exigindo mecanismos de sincronização e gerenciamento de threads para evitar condições de corrida e outros problemas de concorrência.
- **Escalabilidade:** Aplicações assíncronas podem lidar melhor com múltiplas operações de E/S simultâneas, tornando-se mais escaláveis em cenários de alto desempenho.

CONCLUSÃO

Ao longo deste projeto, exploramos a construção de um servidor Java baseado em Sockets, destacando:

1. **Comunicação em Rede:** O uso de `ServerSocket` e `Socket` para estabelecer conexões, com portas atuando como identificadores únicos de serviços.
2. **Serialização de Objetos:** A importância de `ObjectInputStream` e `ObjectOutputStream` para transmitir dados complexos, exigindo que objetos implementem `Serializable`.
3. **Concorrência:** A aplicação de `Threads` tanto no servidor (para múltiplos clientes) quanto no cliente (para respostas assíncronas), evitando bloqueios na thread principal.
4. **Isolamento do Banco de Dados:** A separação clara entre a camada de persistência (via JPA no servidor) e o cliente, garantindo segurança e centralização das regras de negócio.
5. **Sincronia vs. Assincronia:** A comparação entre clientes síncronos (simples, porém bloqueantes) e assíncronos (complexos, mas responsivos), adaptáveis a diferentes necessidades.

Essa abordagem não apenas reforça conceitos fundamentais de programação em rede e concorrência, mas também ilustra boas práticas de arquitetura, como o desacoplamento entre clientes e banco de dados. O resultado é uma base sólida para



CAMPUS: Polo Bezerra de Menezes – Fortaleza/CE

CURSO: Desenvolvimento Full Stack

DISCIPLINA: Por Que Não Paralelizar

TURMA: 9001

SEMESTRE: 2024.4

ALUNO: Daniel Kilzer Brasil Dias

sistemas distribuídos escaláveis, preparados para lidar com demandas modernas de desempenho e segurança.