Jonathan Domingos Carneiro da Silva
matricula: 2023.04.12744-1
3274 POLO CENTRO - ITAITINGA – CE

Nível 5: 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.

– DESENVOLVIMENTO FULL STACK 2023.1 – 3º Semestre – 2024.2

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.

1º Procedimento – Criando o Servidor e Cliente de Teste

CadastroServer:

```
package cadastroserver;

import controller.MovimentosJpaController;
import controller.PessoasJpaController;
import controller.ProdutosJpaController;
import controller.UsuariosJpaController;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class CadastroServer {

    /**
     * @param args the command line arguments
     * @throws java.io.IOException
     */
    public static void main(String[] args) throws IOException {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("CadastroServerPU");
        ProdutosJpaController ctrlProd = new ProdutosJpaController(emf);
        UsuariosJpaController ctrlUsu = new UsuariosJpaController(emf);
        MovimentosJpaController ctrlMov = new MovimentosJpaController(emf);
```

```
        PessoasJpaController ctrlPessoa = new PessoasJpaController(emf);

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

            while (true) {
                Socket socket = serverSocket.accept();
                CadastroThreadV2 thread = new CadastroThreadV2(ctrlUsu, ctrlMov, ctrlProd, ctrlPessoa, null, socket);
                thread.start(); // Inicia a thread
                System.out.println("thread iniciado!");
            }

        }
    }
}
```

CadastroThread:

```
package cadastroserver;

import controller.ProdutosJpaController;
import controller.UsuariosJpaController;
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.Produtos;
import model.Usuarios;

public class CadastroThread extends Thread {
    private final ProdutosJpaController ctrl;
    private final UsuariosJpaController ctrlUsu;
    private final Socket s1;

    public CadastroThread(ProdutosJpaController ctrl, UsuariosJpaController ctrlUsu, Socket s1) {
        this.ctrl = ctrl;
        this.ctrlUsu = ctrlUsu;
        this.s1 = s1;
    }

    @Override
public void run() {
    try (ObjectOutputStream out = new ObjectOutputStream(s1.getOutputStream());
        ObjectInputStream in = new ObjectInputStream(s1.getInputStream())){
```

```
    String login = (String) in.readObject();
    String senha = (String) in.readObject();
    List<Usuarios> usuariosList = ctrlUsu.findUsuariosEntities();
    Usuarios usuarioAutenticado = null;

    for (Usuarios usuario : usuariosList) {
        if (usuario.getLogin().equals(login) && usuario.getSenha().equals(senha)) {
            usuarioAutenticado = usuario;
            break;
        }
    }

    if (usuarioAutenticado == null) {
        System.out.println("Credenciais inválidas. Desconectando cliente.");
        return;
    }

    System.out.println("Usuario autenticado: " + usuarioAutenticado.getLogin());

    while (true) {
        String comando =(String) in.readObject();

        if (comando.equals("L")) {
            List<Produtos> produtos = ctrl.findProdutosEntities();
            out.writeObject(produtos);
            System.out.println("Enviando lista de produtos para o cliente.");
            break;
        }
    }

try {
    if (out != null) {
        out.close();
    }
    if (in != null) {
        in.close();
    }
    if (s1 != null && !s1.isClosed()) {
        s1.close();
    }
} catch (IOException ex) {
    System.err.println("Erro ao fechar os fluxos e o socket: " + ex.getMessage());
}

} catch (IOException ex) {
    System.err.println("Erro de comunicação: " + ex.getMessage());
```

```
    }   catch (ClassNotFoundException ex) {
            Logger.getLogger(CadastroThread.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

CadastroClient:

```
package cadastroclient;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.List;
import model.Produtos;

public class CadastroClient {

    /**
     * @param args the command line arguments
     * @throws java.io.IOException
     * @throws java.lang.ClassNotFoundException
     */
    public static void main(String[] args)throws IOException, ClassNotFoundException {
        try (Socket socket = new Socket("localhost", 4321);
             ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
             ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {

            out.writeObject("op1");
            out.writeObject("op1");
            out.writeObject("L");
            System.out.println("Usuario conectado com sucesso");

            List<Produtos> produtos = (List<Produtos>) in.readObject();
            for (Produtos produto : produtos) {
                System.out.println(produto.getNome());
            }
        }
    }
}
```

2º Procedimento – Alimentando a Base

Como as Threads podem ser utilizadas para o tratamento assíncrono das respostasenviadas pelo servidor?

Threads podem ser utilizadas para o tratamento assíncrono das respostas enviadas pelo servidor ao permitir que cada conexão de cliente seja gerenciada em uma thread separada. Isso possibilita que o servidor lide com múltiplas conexões simultaneamente sem bloquear a execução de outras operações. Ao utilizar threads, cada vez que o servidor aceita uma conexão de um cliente, uma nova thread é criada para lidar com essa conexão. Isso permite que o servidor continue a aceitar novas conexões enquanto processa as solicitações e respostas dos clientes de forma assíncrona.

Para que serve o método invokeLater, da classe SwingUtilities?

O método invokeLater da classe SwingU tilities é usado em aplicações Java que utilizam a biblioteca Swing para garantir que as atualizações na interface gráfica do usuário (GUI) sejam executadas na thread de despacho de eventos (Event Dispatch Thread - EDT). A EDT é uma thread especial em que todos os eventos de GUI, incluindo repinturas e atualizações de componentes, são processados. O método invokeLater aceita um objeto que implementa a interface Runnable e agenda sua execução na EDT. O código dentro do método run do Runnable é então executado na EDT assim que possível.

Como os objetos são enviados e recebidos pelo Socket Java?

Em Java, para enviar e receber objetos através de sockets, utilizam-se as classes ObjectOutputStream e ObjectInputStream. Essas classes permitem a serialização de objetos, ou seja, a conversão de um objeto em um fluxo de bytes que pode ser transmitido pela rede, e para que um objeto possa ser enviado através de um socket, ele deve implementar a interface Serializable. Esta interface é uma marcação que indica que o objeto pode ser serializado.

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

Comunicação Síncrona
A utilização de comportamento assíncrono ou síncrono nos clientes com Socket Java tem implicações significativas no comportamento da aplicação, especialmente em termos de bloqueio do processamento. Vamos explorar e comparar essas abordagens, e na comunicação síncrona, o cliente espera que cada operação de I/O (entrada/saída) seja concluída antes de prosseguir para a próxima operação. Isso significa que as chamadas de leitura e escrita bloqueiam até que os dados sejam completamente enviados ou recebidos.

Comunicação assíncrona

Na comunicação assíncrona, as operações de I/O não bloqueiam a thread que as executa. Em vez disso, o cliente pode continuar executando outras tarefas enquanto aguarda a conclusão das operações de I/O. Isso geralmente é implementado utilizando threads separadas ou mecanismos de callback. A implementação de comunicação assíncrona é mais complexa devido à necessidade de gerenciar múltiplas threads ou callbacks.

Conclusão Final

Ao desenvolver a aplicação cliente-servidor em Java com o uso de sockets e JPA permitiu uma exploração prática de conceitos fundamentais da comunicação distribuída. A utilização de threads e de objetos serializáveis contribuiu para uma implementação mais eficiente e interativa, evidenciando a importância do tratamento assíncrono para uma experiência de usuário fluida e responsiva.

A prática ressaltou a relevância das técnicas de paralelismo e concorrência no tratamento de múltiplas conexões simultâneas, demonstrando como a divisão de tarefas entre threads pode otimizar o desempenho e a escalabilidade do sistema. Além disso, a serialização de objetos facilitou a transmissão de dados complexos entre cliente e servidor, oferecendo uma forma robusta de comunicação.

Essa experiência consolidou meu entendimento sobre o funcionamento dessas tecnologias e sua aplicação em cenários do mundo real, destacando a importância de uma abordagem assíncrona para melhorar a responsividade das aplicações. Em resumo, o trabalho não apenas aprimorou minhas habilidades técnicas, mas também forneceu uma compreensão mais profunda sobre a integração de diferentes componentes em uma arquitetura distribuída eficiente e moderna.