Missão Pratica – Mundo 3 – Nível 5

Andrey Haertel Aires - Matricula: 2021.07.22851-2

Polo Centro - Palhoça – SC

Nível 5: Por que não paralelizar – T 9001 – 3º Semestre Letivo

Endereço Github: https://github.com/AndreyHaires/MissaoPraticaMundo3_N5

Objetivo da Prática

- Criar servidores Java com base em Sockets.
- Criar clientes síncronos para servidores com base em Sockets.
- Criar clientes assíncronos para servidores com base em Sockets.
- Utilizar Threads para implementação de processos paralelos.
- 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.

Estácio

1º Procedimento | Criando o Servidor e Cliente de Teste

Cadastro Server

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class CadastroServer {

    public static void main(String[] args) {
        try {
            ServerSocket serverSocket = new ServerSocket(4321);
            System.out.println("Servidor aguardando conexões...");

            while (true) {
                Socket clientSocket = serverSocket.accept();
                System.out.println("Cliente conectado!");

                ObjectInputStream in = new ObjectInputStream(clientSocket.getInputStream());
                ObjectOutputStream out = new ObjectOutputStream(clientSocket.getOutputStream());

                // Recebe login e senha do cliente
                String login = (String) in.readObject();
                String senha = (String) in.readObject();

                // Valida as credenciais (coloque sua lógica de validação aqui)
                if (validarCredenciais(login, senha)) {
                    out.writeObject("ok");
                    System.out.println("Credenciais válidas. Conexão estabelecida.");
                    // Continue aqui com a lógica do seu aplicativo
                } else {
                    out.writeObject("credenciais-invalidas");
                    System.out.println("Credenciais inválidas. Conexão encerrada.");
                    clientSocket.close();
                }
            }
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    private static boolean validarCredenciais(String login, String senha) {
        // Implemente sua lógica de validação aqui
        // Por exemplo, pode verificar em um banco de dados
        return "usuario".equals(login) && "senha123".equals(senha);
    }
}
```

Cadastro Client

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.InetAddress;
import java.net.Socket;

public class CadastroClient {

    public static void main(String[] args) {
        try {
            Socket clientSocket = new Socket(InetAddress.getByName("localhost"), 4321);
            ObjectOutputStream out = new ObjectOutputStream(clientSocket.getOutputStream());
            ObjectInputStream in = new ObjectInputStream(clientSocket.getInputStream());

            // Envia login e senha para o servidor
            out.writeObject("usuario");
            out.writeObject("senha123");

            // Recebe a resposta do servidor
            String resposta = (String) in.readObject();

            if ("ok".equals(resposta)) {
                System.out.println("Credenciais válidas. Conexão estabelecida.");
                // Continue aqui com a lógica do seu aplicativo
            } else {
                System.out.println("Credenciais inválidas. Conexão encerrada.");
            }

            // Fechar recursos
            out.close();
            in.close();
            clientSocket.close();
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Como funcionam as classes Socket e ServerSocket?

O ServerSocket é usado no lado do servidor para esperar por conexões, enquanto o Socket é usado no lado do cliente e servidor para estabelecer a conexão. O ServerSocket aguarda conexões em uma porta específica e cria um novo Socket para cada conexão aceita. Ambos os lados podem obter fluxos de entrada/saída para trocar dados. Essas classes são essenciais para a comunicação TCP em redes.

Qual a importância das portas para a conexão com servidores?

As portas são essenciais para a conexão com servidores, pois identificam serviços específicos em um servidor, facilitam o roteamento de tráfego, permitem a execução de múltiplos serviços no mesmo servidor e são fundamentais para a segurança, incluindo a configuração de firewalls. Elas seguem padrões de alocação estabelecidos, facilitando a configuração e interoperabilidade entre sistemas.

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 usados para serializar/desserializar objetos em Java, convertendo-os em bytes para transmissão eficiente por redes ou armazenamento em arquivos. A implementação da interface `Serializable` é essencial para indicar que a classe pode ser serializada com segurança.

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 é garantido porque as operações de banco de dados são centralizadas no servidor. Mesmo usando classes de entidades JPA no cliente, o acesso direto ao banco de dados é evitado, e o servidor controla e valida todas as operações, proporcionando segurança e gerenciamento eficiente dos dados.

2º Procedimento | Servidor Completo e Cliente Assíncrono

Cadastro Client V2

```
package cadastroclient;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.InetAddress;
import java.net.Socket;
import java.util.List;
import model.Produto;

public class CadastroClienteV2 {

   public static void main(String[] args) throws IOException, ClassNotFoundException {
      Socket clientSocket = null;
      ObjectInputStream in = null;
      ObjectOutputStream out = null;
      BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

      try {
         clientSocket = new Socket(InetAddress.getByName("localhost"), 4321);
         out = new ObjectOutputStream(clientSocket.getOutputStream());
         in = new ObjectInputStream(clientSocket.getInputStream());
```

```
        // Realizar login
        System.out.println("Digite o Usuário: ");
        out.writeObject(reader.readLine());

        System.out.println("Digite a Senha: ");
        out.writeObject(reader.readLine());

        String result = (String) in.readObject();
        if (!"ok".equals(result)) {
            System.out.println("Erro de login");
            return;
        }
        System.out.println("Login com sucesso");

        String comando;
        do {
            // Solicitar comando ao usuário
            System.out.println("Digite o Comando (L – Listar, E – Entrada, S – Saída, X – Finalizar): ");
            comando = reader.readLine();
            out.writeObject(comando);

            // Executar comandos
            switch (comando.toLowerCase()) {
                case "l":
                    List<Produto> produtos = (List<Produto>) in.readObject();
                    for (Produto produto : produtos) {
                        System.out.println(produto.getNome());
                    }
                    break;
                case "e":
                case "s":
                    realizarEntradaOuSaida(reader, out);
                    break;
            }

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

    } finally {
        closeResources(out, in, clientSocket);
    }
}

private static void realizarEntradaOuSaida(BufferedReader reader, ObjectOutputStream out) throws IOException {
    System.out.println("Digite o id da Pessoa");
    out.writeObject(reader.readLine());

    System.out.println("Digite o id do Produto");
    out.writeObject(reader.readLine());

    System.out.println("Digite a quantidade do Produto");
    out.writeObject(reader.readLine());

    System.out.println("Digite o valor do Produto");
    out.writeObject(reader.readLine());
}

private static void closeResources(ObjectOutputStream out, ObjectInputStream in, Socket socket) {
    try {
        if (out != null) {
            out.close();
        }
```

```
            if (in != null) {
                in.close();
            }
            if (socket != null) {
                socket.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

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

Threads podem ser usadas para tratar respostas assíncronas do servidor, permitindo que o cliente execute operações simultâneas. A Thread de comunicação pode usar callbacks ou polling para verificar e processar respostas, evitando bloqueios durante a espera. Em Java, classes como `SwingWorker` podem facilitar essa implementação.

Para que serve o método invokeLater, da classe SwingUtilities?

`invokeLater` da classe `SwingUtilities` em Java é usado para executar tarefas na Event Dispatch Thread (EDT), garantindo a consistência da interface do usuário e evitando problemas de concorrência.

Como os objetos são enviados e recebidos pelo Socket Java?

Em Java, os objetos são enviados e recebidos por Sockets utilizando as classes ObjectOutputStream para enviar e ObjectInputStream para receber. A serialização converte objetos em sequências de bytes para transmissão eficiente pelos Sockets. As classes dos objetos devem implementar a interface Serializable.

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

<u>Comportamento Síncrono:</u>
Bloqueio sequencial, aguardando a conclusão de cada operação.
Controle direto do fluxo.

<u>Comportamento Assíncrono:</u>
Não bloqueia o processamento principal.
Utiliza callbacks ou polling para lidar com respostas assíncronas.
Mais complexo devido à gestão de operações concorrentes.

Síncrono é mais simples e adequado quando a ordem das operações é crítica.
Assíncrono melhora a responsividade e é útil para lidar com operações concorrentes.