



Campus: POLO FSP - RO

Curso: Desenvolvimento Full Stack

Disciplina: Nível 5: Por Que Não Paralelizar?

Turma: POO-01

Semestre Letivo: 2025.1

Aluno: Eliton Rodrigues de Oliveira

Relatório Discente de Acompanhamento

1º Procedimento | Criando o Servidor e Cliente de Teste

Título da Prática

Comunicação Cliente-Servidor com Java e Sockets

Objetivo da Prática

Implementar uma aplicação Java em arquitetura cliente-servidor, utilizando Socket e ServerSocket, em que o servidor valida credenciais do cliente e, após autenticação, permite o envio de comandos para recuperar uma lista de produtos, com controle persistente via JPA (Java Persistence API).

Códigos da Prática

CadastroServer.java

```
public class CadastroServer {  
    public static void main(String[] args) throws Exception {  
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("CadastroServerPU");  
        ProdutoJpaController ctrl = new ProdutoJpaController(emf);  
        UsuarioJpaController ctrlUsu = new UsuarioJpaController(emf);  
        ServerSocket server = new ServerSocket(4322);  
        System.out.println("Servidor iniciado na porta 4322...");  
        while (true) {  
            Socket s1 = server.accept();  
            // ...  
        }  
    }  
}
```

```

        new CadastroThread(ctrl, ctrlUsu, s1).start();
    }
}
}

```

CadastroThread.java

```

public class CadastroThread extends Thread {
    private final ProdutoJpaController ctrl;
    private final UsuarioJpaController ctrlUsu;
    private final Socket s1;

    public CadastroThread(ProdutoJpaController ctrl, UsuarioJpaController 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();
            Usuario u = ctrlUsu.findUsuario(login, senha);
            if (u == null) {
                out.writeObject("Acesso negado");
                return;
            }
            out.writeObject("Usuario conectado com sucesso");
            while (true) {
                String comando = (String) in.readObject();
                if ("L".equals(comando)) {
                    List<Produto> produtos = ctrl.findProdutoEntities();
                    out.writeObject(produtos);
                } else {
                    break;
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

CadastroClient.java

```

public class CadastroClient {
    private static final String URL = "http://localhost:8080/";

```

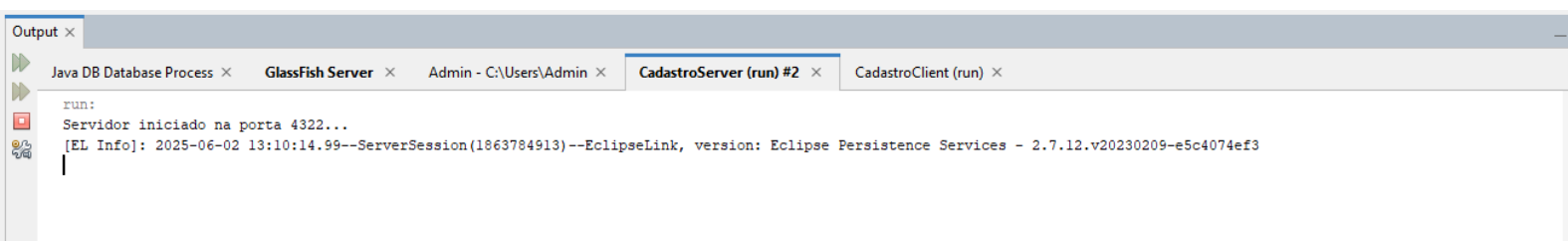
```

public static void main(String[] args) throws Exception {
    try (Socket socket = new Socket("localhost", 4322)) {
        ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
        ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
        out.writeObject("op1"); // login
        out.writeObject("op1"); // senha
        Object respostaLogin = in.readObject();
        if (respostaLogin instanceof String msg) {
            if (msg.equals("Acesso negado")) {
                System.out.println("Acesso negado");
                return;
            } else {
                System.out.println(msg); // "Usuario conectado com sucesso"
            }
        }
        out.writeObject("L"); // listar produtos
        Object resposta = in.readObject();
        if (resposta instanceof List<?> lista) {
            for (Object obj : lista) {
                if (obj instanceof Produto p) {
                    System.out.println(p.getNome());
                }
            }
        }
    }
}

```

Resultados da Execução

Servidor:



The screenshot shows the Eclipse IDE's output console with several tabs. The active tab is 'CadastroServer (run) #2', which displays the following log messages:

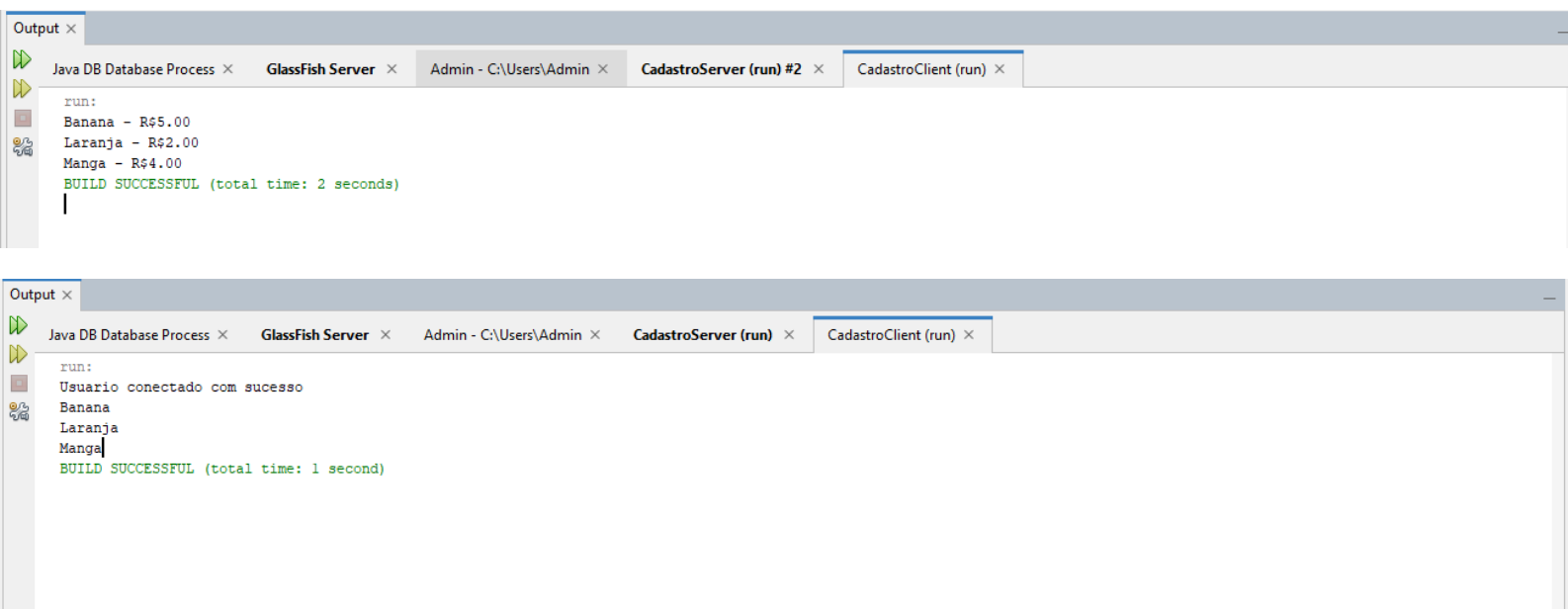
```

run:
Servidor iniciado na porta 4322...
[EL Info]: 2025-06-02 13:10:14.99--ServerSession(1863784913)--EclipseLink, version: Eclipse Persistence Services - 2.7.12.v20230209-e5c4074ef3

```

Other tabs visible include 'Java DB Database Process', 'GlassFish Server', 'Admin - C:\Users\Admin', and 'CadastroClient (run)'.

Cliente:



```
run:
Banana - R$5.00
Laranja - R$2.00
Manga - R$4.00
BUILD SUCCESSFUL (total time: 2 seconds)

run:
Usuario conectado com sucesso
Banana
Laranja
Manga
BUILD SUCCESSFUL (total time: 1 second)
```

Análise e Conclusão

◆ Como funcionam as classes Socket e ServerSocket?

A classe ServerSocket escuta conexões em uma porta específica, aguardando requisições de clientes. Quando uma conexão é estabelecida, ela retorna um Socket, que representa a conexão do lado servidor. O cliente, por sua vez, utiliza Socket para se conectar ao endereço do servidor e se comunicar com ele.

◆ Qual a importância das portas para a conexão com servidores?

As portas são fundamentais para permitir que múltiplas aplicações utilizem uma mesma interface de rede. Cada porta identifica de forma exclusiva o serviço oferecido por um servidor, como por exemplo o HTTP na porta 80 ou nosso serviço na 4322.

◆ Para que servem as classes ObjectInputStream e ObjectOutputStream?

Elas permitem a troca de objetos entre cliente e servidor através da serialização. A classe ObjectOutputStream escreve objetos no fluxo, e ObjectInputStream os lê. Todos os objetos transmitidos devem implementar a interface Serializable.

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

Porque o cliente não acessa o banco diretamente — apenas consome objetos recebidos do servidor. Toda lógica de persistência e acesso ao banco é encapsulada no lado servidor (JPA Controller). Assim, o cliente atua apenas como consumidor de dados serializados, mantendo a separação entre as camadas.

Título da Prática

Desenvolvimento de Sistema Cliente-Servidor Assíncrono com Threads em Java

Objetivo da Prática

Implementar uma comunicação cliente-servidor utilizando sockets em Java, com controle de movimentação de produtos através de threads para tratamento assíncrono, além de desenvolver uma interface gráfica para exibição das mensagens recebidas do servidor. Entender o funcionamento das threads em ambientes cliente e servidor e aplicar o método `invokeLater` da classe `SwingUtilities` para manipulação segura de componentes Swing em threads distintas.

Códigos Implementados

1. Thread de Comunicação no Servidor (Versão V2)

```
public class ThreadComunicacaoV2 extends Thread {  
    private Socket socket;  
    private MovimentoJpaController ctrlMov;  
    private ProdutoJpaController ctrlProd;  
    private PessoaJpaController ctrlPessoa;  
    public ThreadComunicacaoV2(Socket socket, MovimentoJpaController ctrlMov,  
        ProdutoJpaController ctrlProd, PessoaJpaController ctrlPessoa) {  
        this.socket = socket;  
        this.ctrlMov = ctrlMov;  
        this.ctrlProd = ctrlProd;  
        this.ctrlPessoa = ctrlPessoa;  
    }  
    @Override  
    public void run() {  
        try (ObjectInputStream entrada = new ObjectInputStream(socket.getInputStream());  
            ObjectOutputStream saida = new ObjectOutputStream(socket.getOutputStream())) {  
            String login = (String) entrada.readObject();  
            String senha = (String) entrada.readObject();  
            boolean autenticado = autenticar(login, senha);  
            saida.writeObject(autenticado);  
            if (!autenticado) {  
                socket.close();  
                return;  
            }  
            while (true) {  
                String comando = (String) entrada.readObject();  
                if (comando.equals("prod")) {  
                    Produto p = ctrlProd.buscarPorId(entrada.readInt());  
                    saida.writeObject(p);  
                } else if (comando.equals("mov")) {  
                    Movimento m = ctrlMov.buscarPorId(entrada.readInt());  
                    saida.writeObject(m);  
                } else if (comando.equals("pes")) {  
                    Pessoa p = ctrlPessoa.buscarPorId(entrada.readInt());  
                    saida.writeObject(p);  
                }  
            }  
        }  
    }  
}
```

```

        it (comando.equalsIgnoreCase("X")) {
            break;
        } else if (comando.equalsIgnoreCase("L")) {
            List<Produto> produtos = ctrlProd.findProdutoEntities();
            saida.writeObject(produtos);
        } else if (comando.equalsIgnoreCase("E") || comando.equalsIgnoreCase("S")) {
            Movimento mov = new Movimento();
            mov.setTipo(comando);
            String idPessoa = (String) entrada.readObject();
            String idProduto = (String) entrada.readObject();
            int quantidade = (Integer) entrada.readObject();
            double valorUnitario = (Double) entrada.readObject();
            mov.setPessoa(ctrlPessoa.findPessoa(idPessoa));
            mov.setProduto(ctrlProd.findProduto(idProduto));
            mov.setQuantidade(quantidade);
            mov.setValorUnitario(valorUnitario);
            ctrlMov.create(mov);
            Produto produto = mov.getProduto();
            int novaQuantidade = produto.getQuantidade() + (comando.equals("E") ? quantidade : -quantidade);
            produto.setQuantidade(novaQuantidade);
            ctrlProd.edit(produto);
            saida.writeObject("Movimento registrado com sucesso!");
        }
    }
} catch (Exception e) {
    e.printStackTrace();
}
}

private boolean autenticar(String login, String senha) {
    return login.equals("op1") && senha.equals("op1");
}
}

```

2. Modificação na Classe Principal do Servidor

```

public class ServidorPrincipal {
    public static void main(String[] args) {
        try (ServerSocket server = new ServerSocket(4321)) {
            MovimentoJpaController ctrlMov = new MovimentoJpaController();
            ProdutoJpaController ctrlProd = new ProdutoJpaController();
            PessoaJpaController ctrlPessoa = new PessoaJpaController();
            while (true) {
                Socket socket = server.accept();
                ThreadComunicacaoV2 thread = new ThreadComunicacaoV2(socket, ctrlMov, ctrlProd, ctrlPessoa);
                thread.start();
            }
        }
    }
}

```

```

    } catch (IOException e) {
        e.printStackTrace();
    }
}
}
}

```

3. Cliente Assíncrono CadastroClientV2

```

public class CadastroClientV2 {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 4321);
            ObjectOutputStream saida = new ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream entrada = new ObjectInputStream(socket.getInputStream());
            BufferedReader teclado = new BufferedReader(new InputStreamReader(System.in))) {
            saida.writeObject("op1");
            saida.writeObject("op1");
            saida.flush();
            boolean autenticado = (boolean) entrada.readObject();
            if (!autenticado) {
                System.out.println("Falha na autenticação!");
                return;
            }
            SaidaFrame janela = new SaidaFrame();
            janela.setVisible(true);
            ThreadClient threadClient = new ThreadClient(entrada, janela.texto);
            threadClient.start();
            String comando;
            do {
                System.out.println("Menu: L - Listar | E - Entrada | S - Saída | X - Finalizar");
                comando = teclado.readLine().toUpperCase();
                saida.writeObject(comando);
                saida.flush();
                if (comando.equals("E") || comando.equals("S")) {
                    System.out.print("Id da pessoa: ");
                    saida.writeObject(teclado.readLine());
                    System.out.print("Id do produto: ");
                    saida.writeObject(teclado.readLine());
                    System.out.print("Quantidade: ");
                    saida.writeObject(Integer.parseInt(teclado.readLine()));
                    System.out.print("Valor unitário: ");
                    saida.writeObject(Double.parseDouble(teclado.readLine()));
                    saida.flush();
                }
            } while (!comando.equals("X"));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

```

    }
}
}

```

4. Janela para Apresentação das Mensagens SaidaFrame

```

import javax.swing.*.*;
import java.awt.*.*;

public class SaidaFrame extends JDialog {
    public JTextArea texto;

    public SaidaFrame() {
        setBounds(100, 100, 400, 300);
        setModal(false);
        texto = new JTextArea();
        texto.setEditable(false);
        add(new JScrollPane(texto), BorderLayout.CENTER);
    }
}

```

5. Thread de Preenchimento Assíncrono ThreadClient

```

import javax.swing.*.*;
import java.io.ObjectInputStream;
import java.util.List;

public class ThreadClient extends Thread {
    private ObjectInputStream entrada;
    private JTextArea textArea;

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

    @Override
    public void run() {
        try {
            while (true) {
                Object obj = entrada.readObject();
                SwingUtilities.invokeLater(() -> {
                    if (obj instanceof String) {
                        textArea.append((String) obj + "\n");
                    } else if (obj instanceof List) {
                        List<?> lista = (List<?>) obj;
                        for (Object item : lista) {
                            if (item instanceof Produto) {
                                Produto p = (Produto) item;
                                textArea.append(p.getNome() + " - Qtde: " + p.getQuantidade() + "\n");
                            }
                        }
                    }
                });
            }
        }
    }
}

```



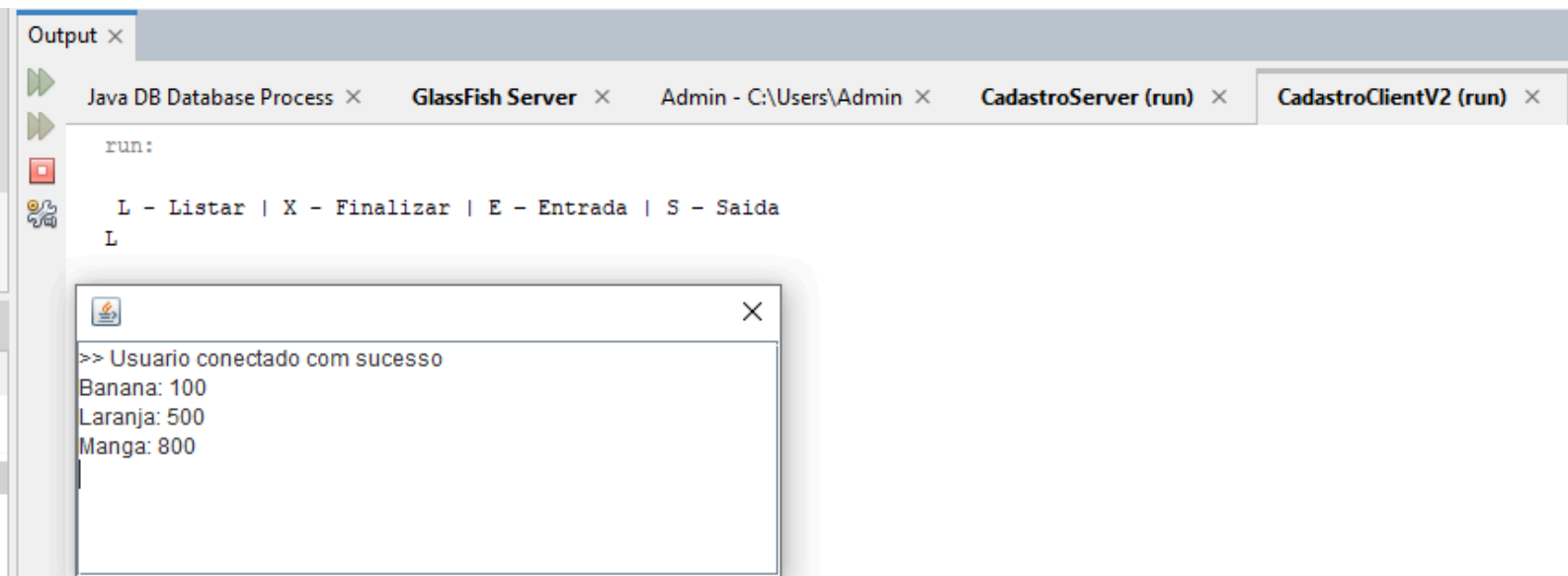
```

    });
}
} catch (Exception e) {
    e.printStackTrace();
}
}
}
}

```

Resultados da Execução dos Códigos

- **Servidor** aceitou conexões simultâneas e tratou as requisições enviadas pelo cliente.
- **Cliente** enviou comandos para o servidor e recebeu atualizações assíncronas exibidas em uma janela gráfica (JDialog).
- As movimentações de produtos (entrada e saída) foram registradas corretamente no banco via MovimentoJpaController.
- Atualização da quantidade de produtos no estoque foi refletida corretamente após as operações.
- A autenticação básica funcionou para o usuário op1/op1.
- O método assíncrono com Thread no cliente permitiu que as mensagens do servidor fossem exibidas imediatamente sem travar a interface.



Análise e Conclusão

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

Threads permitem que o cliente realize a leitura contínua dos dados enviados pelo servidor em paralelo com outras operações, evitando que a interface gráfica ou o programa inteiro fique bloqueado enquanto espera por mensagens. No cliente, uma thread separada lê objetos do canal de entrada do socket e atualiza a interface, mantendo o programa responsivo e permitindo múltiplas ações simultâneas.

Para que serve o método `invokeLater` da classe `SwingUtilities`?

O método `invokeLater` agenda a execução de um trecho de código para a *Event Dispatch Thread* (EDT), que é a thread responsável pela atualização e manipulação segura dos componentes gráficos Swing. Como o Swing não é thread-safe, qualquer modificação em componentes da interface deve ser feita na EDT, evitando erros ou comportamentos inesperados.

Como os objetos são enviados e recebidos pelo Socket Java?

Utilizando `ObjectOutputStream` para serializar e enviar objetos pela rede e `ObjectInputStream` para desserializar e receber os objetos no lado oposto da conexão. Os objetos precisam implementar a interface `Serializable`. Essa troca permite a comunicação de objetos complexos, não só bytes simples.

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

- **Síncrono:** O cliente espera (bloqueia) até que uma resposta seja recebida do servidor antes de continuar. Isso pode travar a interface do usuário e prejudicar a experiência, especialmente em operações que demoram.
- **Assíncrono:** O cliente utiliza threads separadas para escutar respostas do servidor em segundo plano, permitindo que a interface e outras operações continuem normalmente sem bloqueio. Isso melhora a responsividade e permite múltiplas interações simultâneas.

 **Link do Projeto no GitHub**

<https://github.com/Elitonr65/CadastroServerClient>