



**Estácio**

**CAMPUS:** Polo Jacareí - SP – Centro

**CURSO:** Desenvolvimento FullStack

**DISCIPLINA:** Por que não paralelizar

**TURMA:** 2025.1

**SEMESTRE LETIVO:** Primeiro Semestre (2025)

**ALUNO:** Maurício Pereira Campos

**MATRÍCULA:** 202403843447

### **Título da prática:**

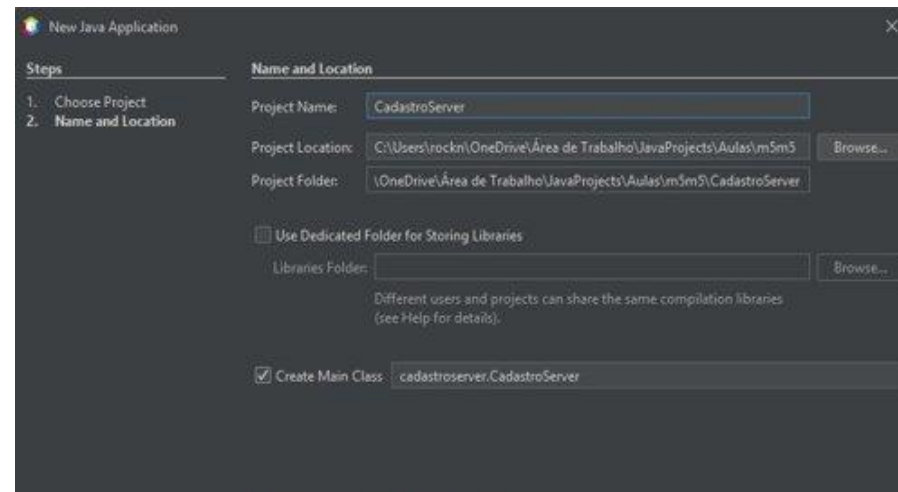
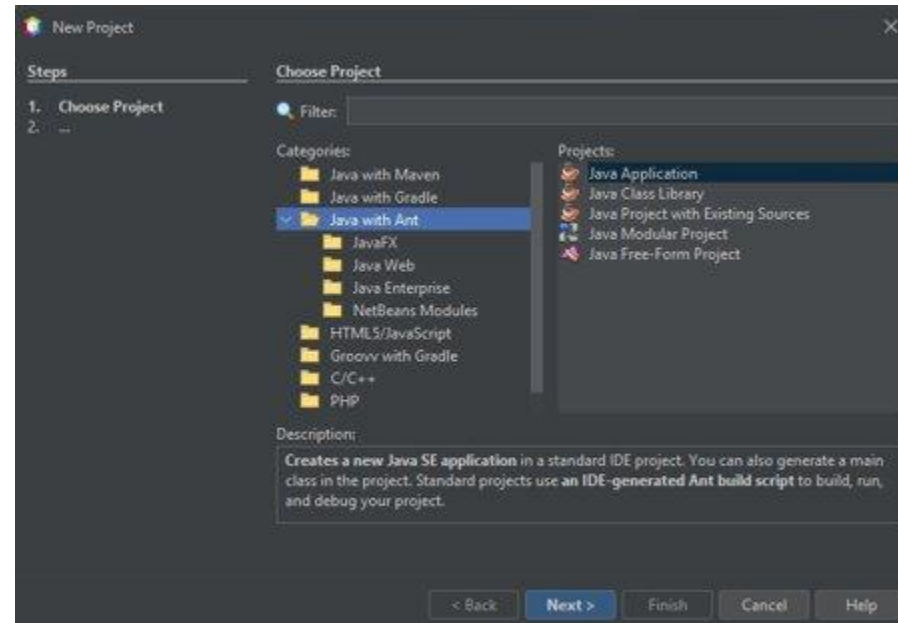
Servidores e clientes baseados em Socket, com uso de Threads tanto no lado cliente quanto no lado servidor, acessando o banco de dados via JPA.

### **Objetivos 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.

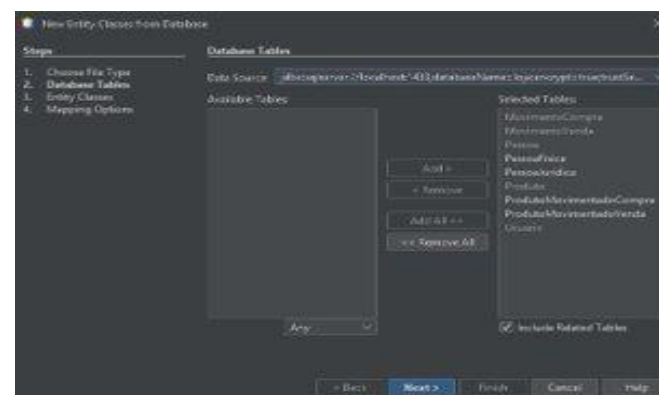
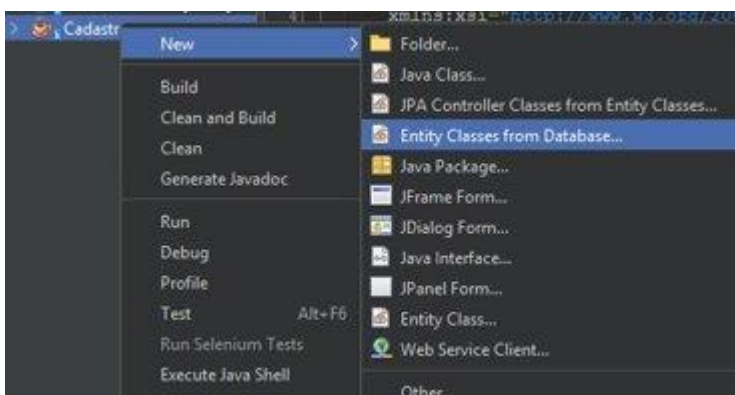
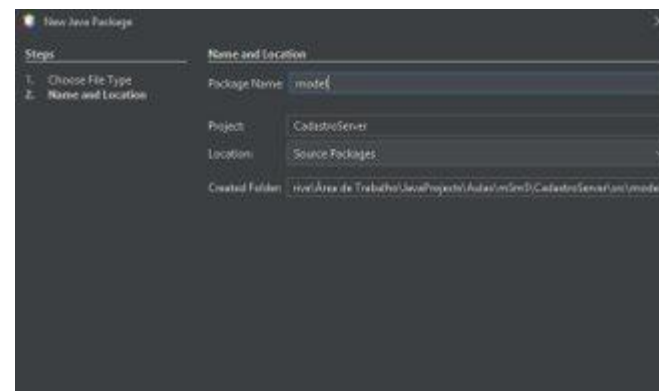
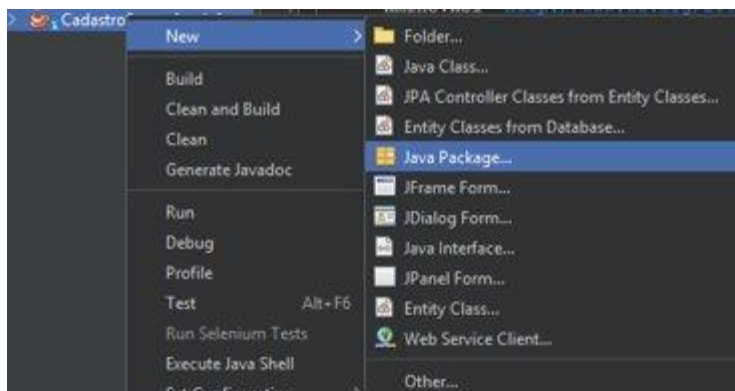
# 1º Procedimento | Criando o Servidor e Cliente de Teste

## Códigos usados neste roteiro



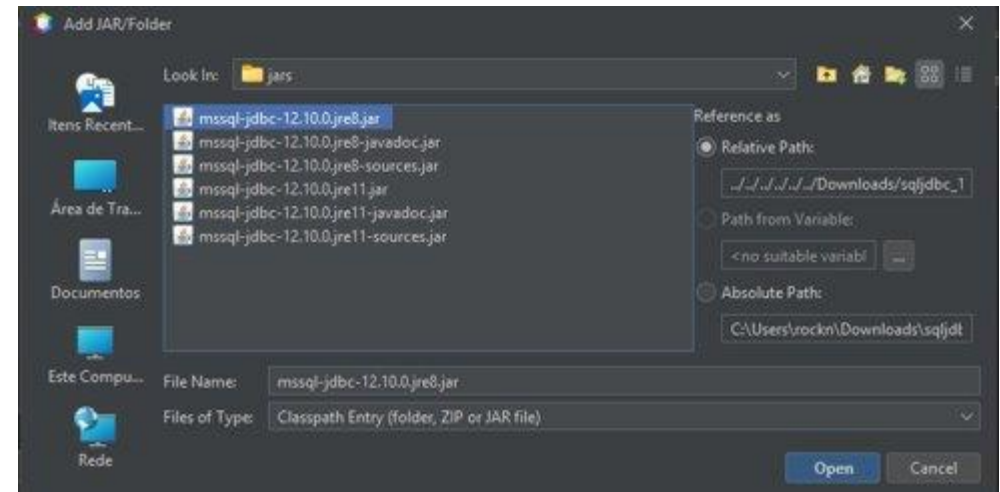
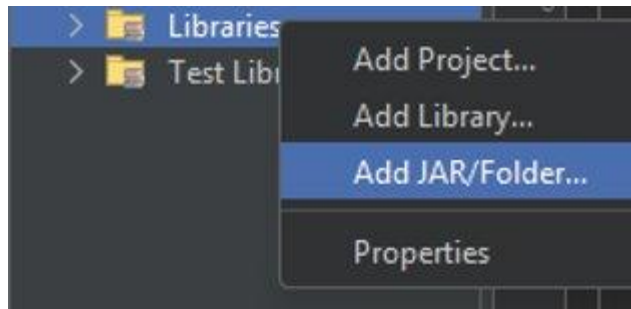
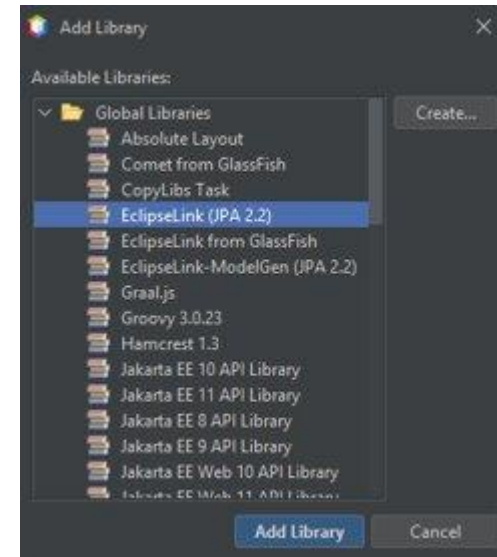
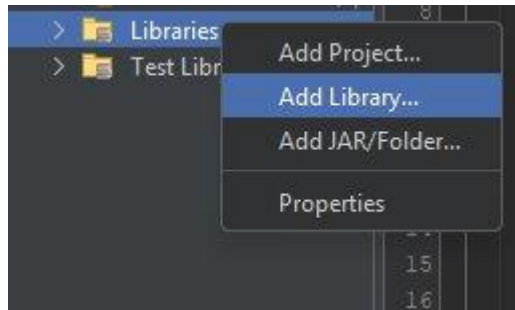
# 1º Procedimento | Criando o Servidor e Cliente de Teste

## Códigos usados neste roteiro



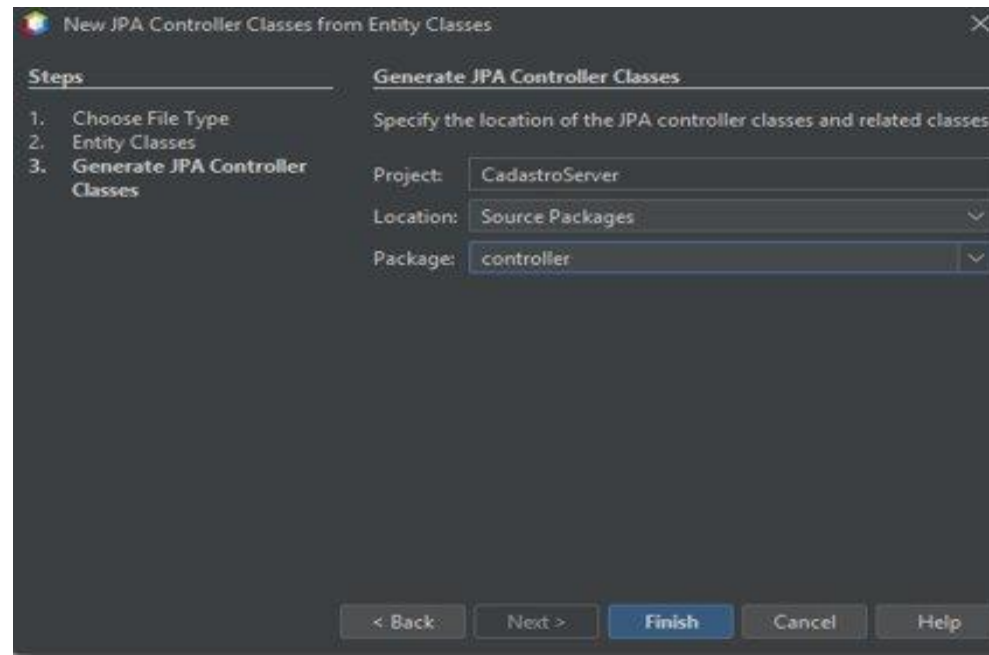
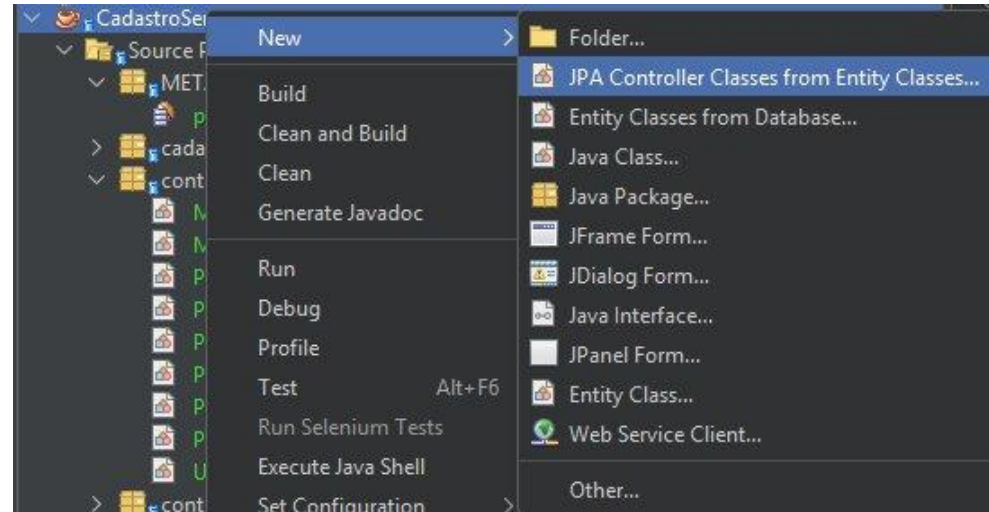
# 1º Procedimento | Criando o Servidor e Cliente de Teste

## Códigos usados neste roteiro



# 1º Procedimento | Criando o Servidor e Cliente de Teste

## Códigos usados neste roteiro



# 1º Procedimento | Criando o Servidor e Cliente de Teste

## Códigos usados neste roteiro

```
package controller;

import java.io.Serializable;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.TypedQuery;
import model.Usuario;

public class UsuarioJpaController implements Serializable {

    private EntityManagerFactory emf = null;

    public UsuarioJpaController(EntityManagerFactory emf) {
        this.emf = emf;
    }

    public EntityManager getEntityManager() {
        return emf.createEntityManager();
    }

    public Usuario findUsuario(String login, String senha) {
        EntityManager em = getEntityManager();
        try {
            TypedQuery<Usuario> query = em.createQuery(
                "SELECT u FROM Usuario u WHERE u.login = :login AND u.senha = :senha", Usuario.class);
            query.setParameter("login", login);
            query.setParameter("senha", senha);
            return query.getSingleResult();
        } catch (Exception e) {
            return null;
        } finally {
            em.close();
        }
    }
}
```



# 1º Procedimento | Criando o Servidor e Cliente de Teste

## Códigos usados neste roteiro

```
package cadastroserver;

import controller.ProdutoJpaController;
import controller.UsuarioJpaController;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.List;
import model.Produto;
import model.Usuario;

// Sua classe herda de Thread
public class CadastroThread extends Thread {

    // Atributos necessários para a comunicação e acesso ao banco
    private ProdutoJpaController ctrl;
    private UsuarioJpaController ctrlUsu;
    private Socket socket;

    // Construtor para receber os controladores e o Socket
    public CadastroThread(ProdutoJpaController ctrl, UsuarioJpaController ctrlUsu, Socket socket) {
        this.ctrl = ctrl;
        this.ctrlUsu = ctrlUsu;
        this.socket = socket;
    }
}
```

# 1º Procedimento | Criando o Servidor e Cliente de Teste

## Códigos usados neste roteiro

```
// O método run() contém o código que será executado em paralelo quando a thread iniciar
@Override
public void run() {
    try {
        // Cria os streams de entrada e saída para comunicação via Socket
        ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
        ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());

        // Exemplo de leitura de credenciais (login e senha) enviados pelo cliente
        String login = (String) ois.readObject();
        String senha = (String) ois.readObject();
        System.out.println("Login recebido: " + login);

        // Validação usando o controlador de usuário
        Usuario usuario = ctrlUsu.findUsuario(login, senha);
        if (usuario == null) {
            oos.writeObject("Credenciais inválidas.");
            System.out.println("Credenciais inválidas para: " + login);
            socket.close();
            return;
        } else {
            oos.writeObject("Usuário válido. Conectado.");
        }

        // Laço para processar comandos enviados pelo cliente
        boolean running = true;
        while (running) {
            // Lê o comando (por exemplo, "L" para listar produtos ou "fim" para encerrar)
            String comando = (String) ois.readObject();
            System.out.println("Comando recebido: " + comando);

            if (comando == null) {
                break;
            }

            if (comando.equalsIgnoreCase("L")) {
                // Se comando for "L", consulta a lista de produtos no banco e envia ao cliente
                List<Produto> produtos = ctrl.findProdutoEntities();
                oos.writeObject(produtos);
            } else if (comando.equalsIgnoreCase("fim")) {
                // Comando para encerrar a comunicação
                running = false;
            } else {
                oos.writeObject("Comando não reconhecido.");
            }
        }

        // Fecha a conexão ao fim do processamento
        socket.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

# 1º Procedimento | Criando o Servidor e Cliente de Teste

## Códigos usados neste roteiro

```
package cadastroserver;

import controller.ProdutoJpaController;
import controller.UsuarioJpaController;
import java.net.ServerSocket;
import java.net.Socket;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class CadastroServer {
    public static void main(String[] args) {
        try {
            // Cria o EntityManagerFactory com sua unidade de persistência
            EntityManagerFactory emf = Persistence.createEntityManagerFactory("CadastroServerPU");

            // Cria os controladores JPA correspondentes
            ProdutoJpaController ctrl = new ProdutoJpaController(emf);
            UsuarioJpaController ctrlUsu = new UsuarioJpaController(emf);

            // Cria o ServerSocket para escutar na porta 4321
            ServerSocket serverSocket = new ServerSocket(4321);
            System.out.println("Servidor iniciado na porta 4321...");

            // Loop infinito para aceitar clientes
            while (true) {
                Socket clientSocket = serverSocket.accept(); // Aguarda um cliente
                System.out.println("Cliente conectado: " + clientSocket.getInetAddress());

                // Cria uma nova thread para o cliente
                CadastroThread thread = new CadastroThread(ctrl, ctrlUsu, clientSocket);
                thread.start(); // Inicia a thread (a execução do método run() ocorre em paralelo)
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

# 1º Procedimento | Criando o Servidor e Cliente de Teste

## Códigos usados neste roteiro

```
package controller;

import java.io.Serializable;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import model.Produto;

public class ProdutoJpaController implements Serializable {

    private EntityManagerFactory emf = null;

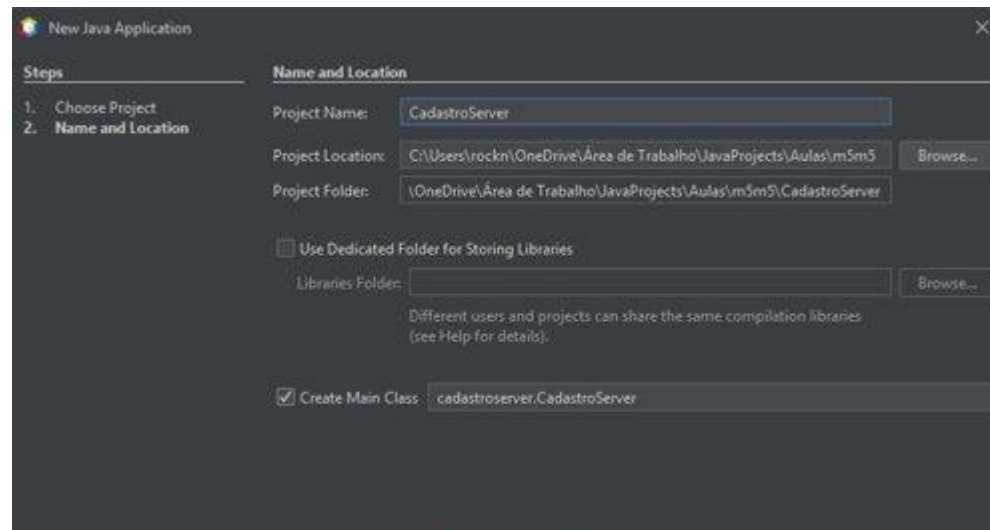
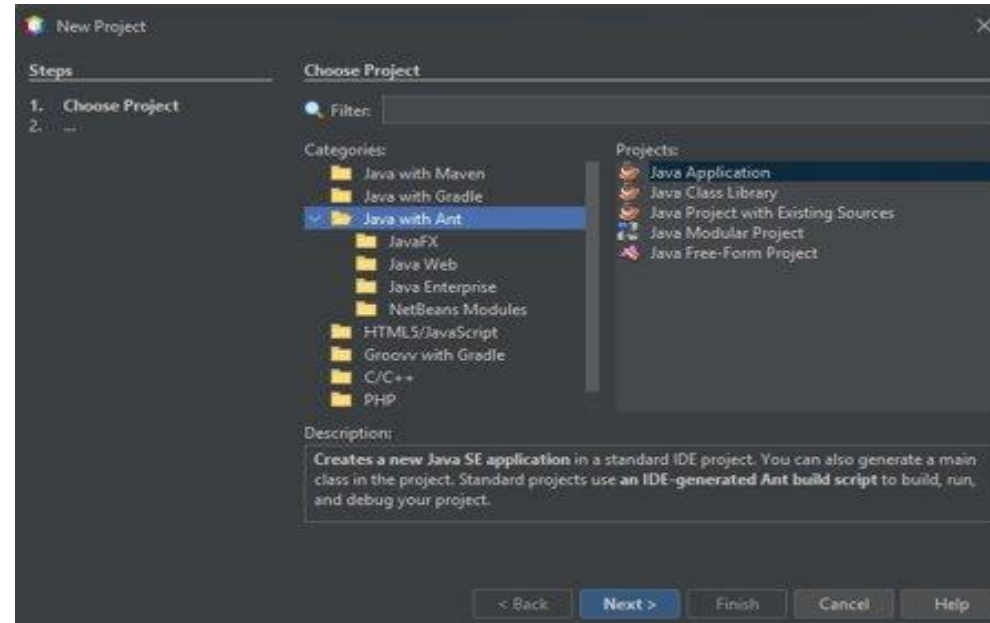
    public ProdutoJpaController(EntityManagerFactory emf) {
        this.emf = emf;
    }

    public EntityManager getEntityManager() {
        return emf.createEntityManager();
    }

    public List<Produto> findProdutoEntities() {
        EntityManager em = getEntityManager();
        try {
            return em.createQuery("SELECT p FROM Produto p", Produto.class).getResultList();
        } finally {
            em.close();
        }
    }
}
```

# 1º Procedimento | Criando o Servidor e Cliente de Teste

## Códigos usados neste roteiro



# 1º Procedimento | Criando o Servidor e Cliente de Teste

## Códigos usados neste roteiro

```
package cadastroclient;

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

public class CadastroClient {
    public static void main(String[] args) {
        try {
            // Conecta ao servidor na porta 4321 do localhost
            Socket socket = new Socket("localhost", 4321);
            ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());

            // Envia as credenciais (exemplo: login "opl" e senha "opl")
            oos.writeObject("opl");
            oos.writeObject("opl");

            // Lê a resposta do servidor para validação
            String resposta = (String) ois.readObject();
            System.out.println("Resposta do servidor: " + resposta);
            if (!resposta.startsWith("Usuário válido")) {
                socket.close();
                return;
            }

            // Envia o comando "L" para solicitar a listagem de produtos
            oos.writeObject("L");

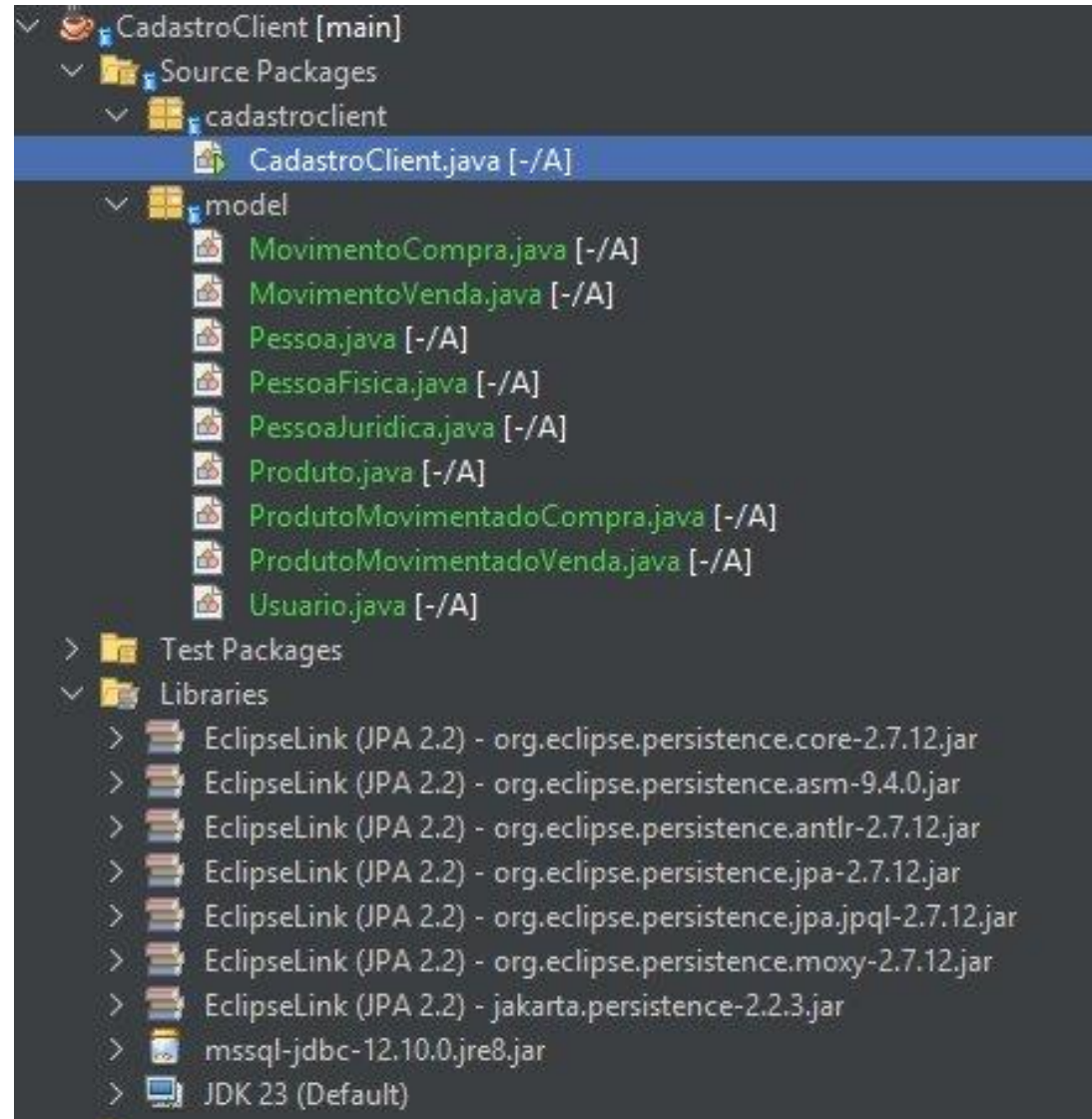
            // Recebe a lista de produtos
            List<Produto> produtos = (List<Produto>) ois.readObject();
            System.out.println("Lista de Produtos:");
            for (Produto p : produtos) {
                System.out.println("Produto: " + p.getNome());
            }

            // Envia o comando para encerrar a comunicação
            oos.writeObject("fim");
            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



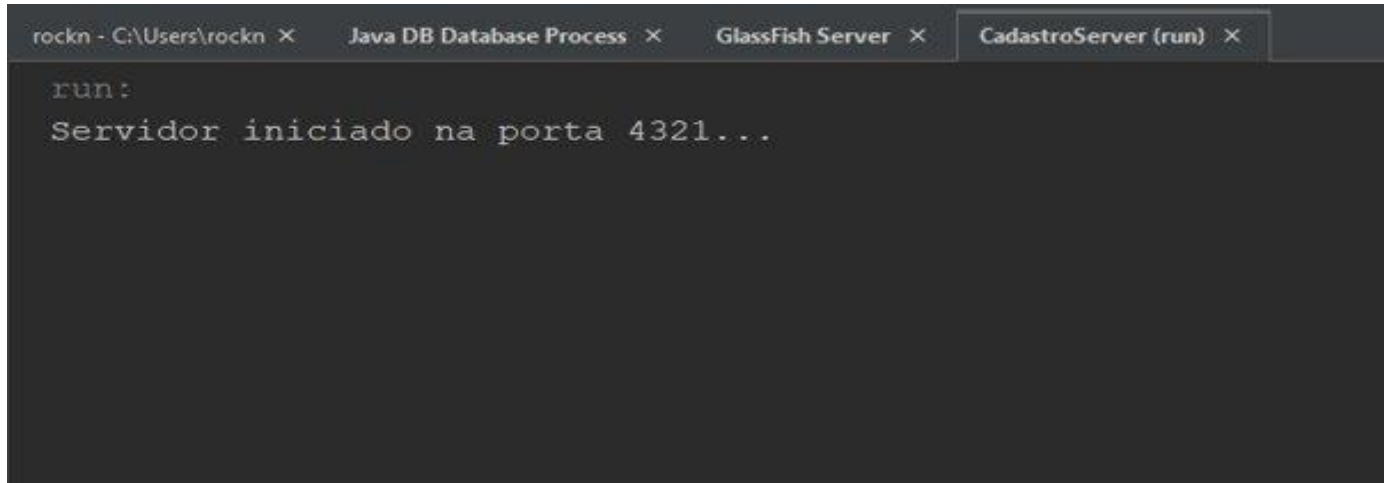
# 1º Procedimento | Criando o Servidor e Cliente de Teste

## Códigos usados neste roteiro



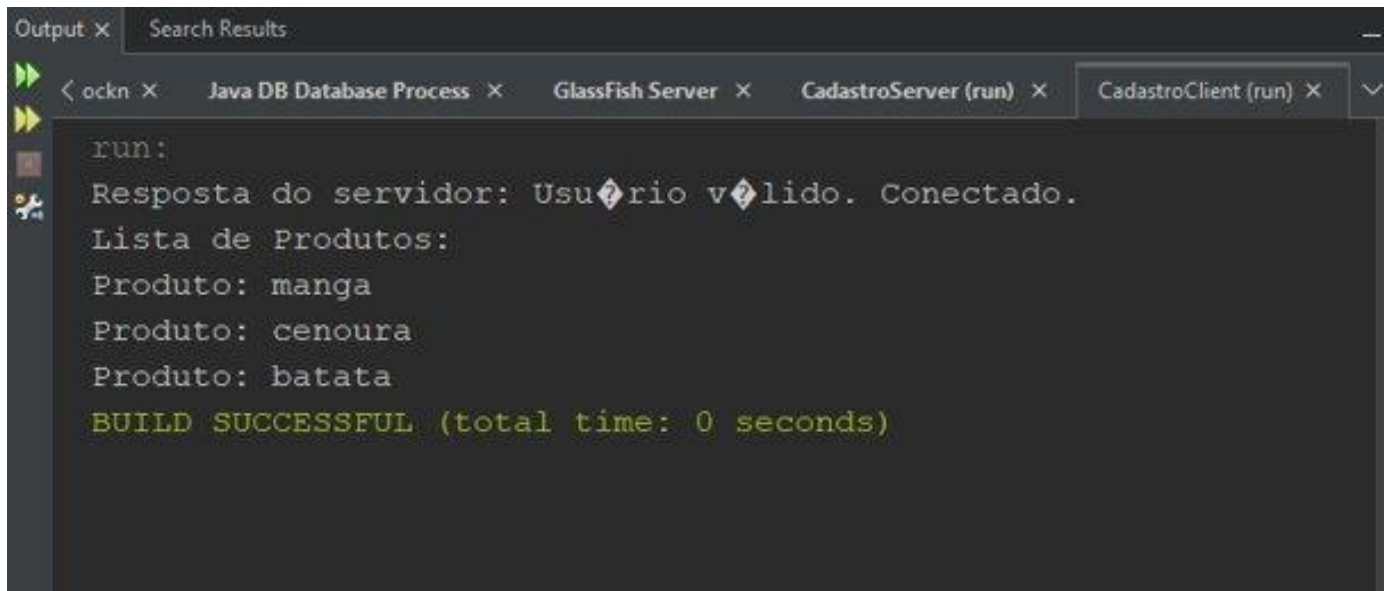
# 1º Procedimento | Criando o Servidor e Cliente de Teste

## Códigos usados neste roteiro



The screenshot shows an IDE with four tabs: 'rockn - C:\Users\rockn', 'Java DB Database Process', 'GlassFish Server', and 'CadastroServer (run)'. The 'CadastroServer (run)' tab is active, displaying the output of the server's execution.

```
run:
Servidor iniciado na porta 4321...
```



The screenshot shows the same IDE with five tabs: 'ockn', 'Java DB Database Process', 'GlassFish Server', 'CadastroServer (run)', and 'CadastroClient (run)'. The 'CadastroClient (run)' tab is active, displaying the output of the client's execution. The output shows a successful connection to the server and a list of products.

```
run:
Resposta do servidor: Usuário válido. Conectado.
Lista de Produtos:
Produto: manga
Produto: cenoura
Produto: batata
BUILD SUCCESSFUL (total time: 0 seconds)
```



## **Análise e conclusão:**

### **1. Como funcionam as classes Socket e ServerSocket?**

As classes Socket e ServerSocket fazem parte do pacote java.net e são utilizadas para a comunicação em rede via protocolo TCP em Java. Elas permitem a criação de aplicações cliente-servidor, em que um lado (servidor) fica escutando conexões e o outro (cliente) se conecta a ele para trocar dados.

### **2. Qual a importância das portas para a conexão com servidores?**

As portas são fundamentais para o funcionamento da comunicação em rede, especialmente em aplicações cliente-servidor. Elas atuam como pontos de acesso lógicos dentro de um computador que permitem que diferentes serviços de rede funcionem simultaneamente.

### **3. Para que servem as classes de entrada e saída ObjectInputStream e ObjectOutputStream, e por que os objetos transmitidos devem ser serializáveis?**

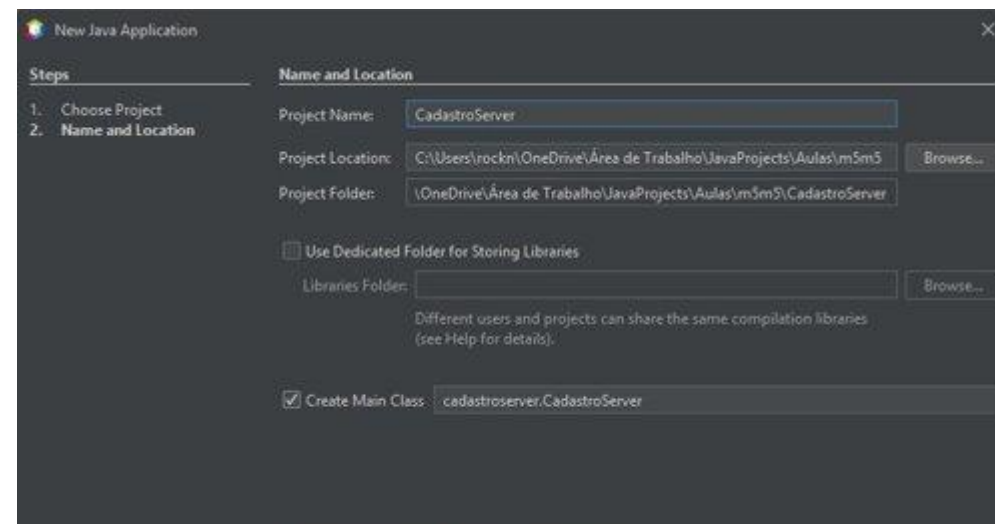
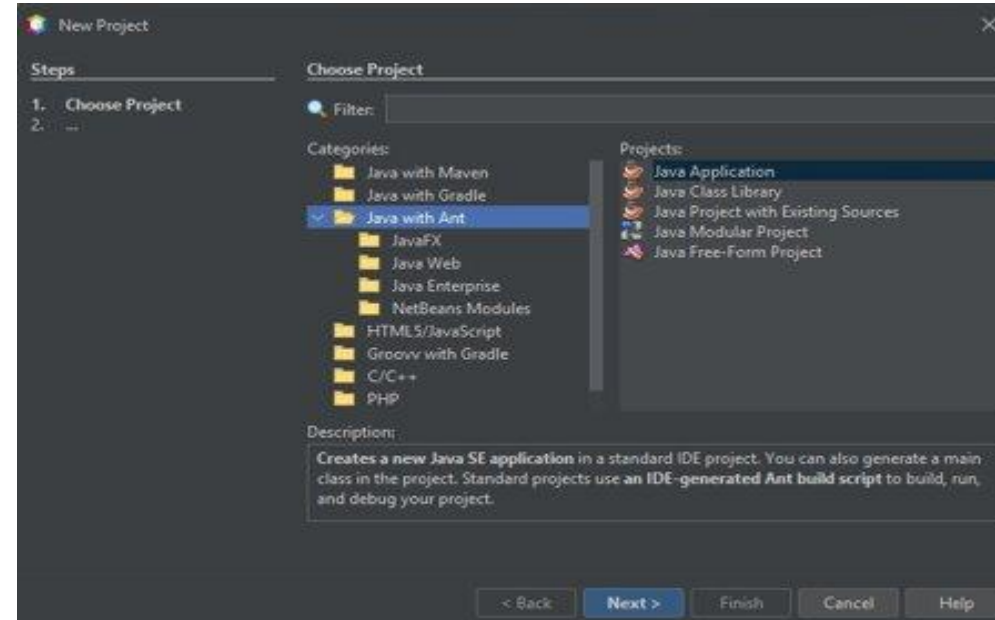
As classes ObjectInputStream e ObjectOutputStream são utilizadas em Java para serialização de objetos , ou seja, para permitir a leitura e escrita de objetos em fluxos de dados (streams) . Elas são muito úteis quando você precisa transmitir objetos entre programas ou armazená-los em arquivos ou na rede.

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

A pergunta está relacionada ao uso de classes de entidades JPA (Java Persistence API) no lado do cliente, e como isso não comprometeu o isolamento do acesso ao banco de dados

## 2º Procedimento | Servidor Completo e Cliente Assíncrono

### Códigos usados neste roteiro



## 2º Procedimento | Servidor Completo e Cliente Assíncrono

### Códigos usados neste roteiro

```
package cadastroserver;

import controller.*;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.List;
import model.*;

public class CadastroThreadV2 extends Thread {

    private ProdutoJpaController ctrlProd;
    private UsuarioJpaController ctrlUsu;
    private MovimentoCompraJpaController ctrlMovCompra;
    private MovimentoVendaJpaController ctrlMovVenda;
    private PessoaJpaController ctrlPessoa;
    private Socket socket;

    public CadastroThreadV2(ProdutoJpaController ctrlProd,
                           UsuarioJpaController ctrlUsu,
                           MovimentoCompraJpaController ctrlMovCompra,
                           MovimentoVendaJpaController ctrlMovVenda,
                           PessoaJpaController ctrlPessoa,
                           Socket socket) {

        this.ctrlProd = ctrlProd;
        this.ctrlUsu = ctrlUsu;
        this.ctrlMovCompra = ctrlMovCompra;
        this.ctrlMovVenda = ctrlMovVenda;
        this.ctrlPessoa = ctrlPessoa;
        this.socket = socket;
    }
}
```

## 2º Procedimento | Servidor Completo e Cliente Assíncrono

### Códigos usados neste roteiro

```
@Override
public void run() {
    try {
        ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
        ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());

        String login = ((String) ois.readObject()).trim();
        String senha = ((String) ois.readObject()).trim();
        Usuario usuario = ctrlUsu.findUsuario(login, senha);
        if (usuario == null) {
            oos.writeObject("Credenciais inválidas.");
            oos.flush();
            socket.close();
            return;
        } else {
            oos.writeObject("Usuário válido. Conectado.");
            oos.flush();
        }

        boolean running = true;
        while (running) {
            String comando = ((String) ois.readObject()).trim();

            if (comando.equalsIgnoreCase("L")) {
                List<Produto> produtos = ctrlProd.findProdutoEntities();
                oos.writeObject(produtos);
                oos.flush();
            } else if (comando.equalsIgnoreCase("E") || comando.equalsIgnoreCase("S")) {

                int idPessoa = Integer.parseInt(((String) ois.readObject()).trim());
                Pessoa pessoa = ctrlPessoa.findPessoa(idPessoa);

                int idProduto = Integer.parseInt(((String) ois.readObject()).trim());
                Produto produto = ctrlProd.findProduto(idProduto);

                int quantidade = Integer.parseInt(((String) ois.readObject()).trim());
                float valorUnitario = Float.parseFloat(((String) ois.readObject()).trim());
```

## 2º Procedimento | Servidor Completo e Cliente Assíncrono

### Códigos usados neste roteiro

```
        if (comando.equalsIgnoreCase("E")) {
            MovimentoCompra movCompra = new MovimentoCompra();
            movCompra.setUsuario(usuario);
            movCompra.setPessoa(pessoa);
            movCompra.setProduto(produto);
            movCompra.setQuantidade(quantidade);
            movCompra.setValorUnitario(valorUnitario);
            ctrlMovCompra.create(movCompra);

            produto.setQuantidadeEstoque(produto.getQuantidadeEstoque() + quantidade);
            ctrlProd.edit(produto);

            produto = ctrlProd.findProduto(produto.getIdProduto());

            oos.writeObject("Movimento de Entrada processado.");
            oos.flush();
        } else if (comando.equalsIgnoreCase("S")) {
            MovimentoVenda movVenda = new MovimentoVenda();
            movVenda.setUsuario(usuario);
            movVenda.setPessoa(pessoa);
            movVenda.setProduto(produto);
            movVenda.setQuantidade(quantidade);
            movVenda.setValorUnitario(valorUnitario);
            ctrlMovVenda.create(movVenda);

            produto.setQuantidadeEstoque(produto.getQuantidadeEstoque() - quantidade);
            ctrlProd.edit(produto);

            produto = ctrlProd.findProduto(produto.getIdProduto());

            oos.writeObject("Movimento de Saída processado.");
            oos.flush();
        }
    } else if (comando.equalsIgnoreCase("X")) {
        running = false;
    } else {
        oos.writeObject("Comando não reconhecido.");
        oos.flush();
    }
}
socket.close();
} catch (Exception e) {
    e.printStackTrace();
}
}
```



## 2º Procedimento | Servidor Completo e Cliente Assíncrono

### Códigos usados neste roteiro

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.2" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_2.xsd">
  <persistence-unit name="CadastroServerPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>model.Produto</class>
    <class>model.PessoaJuridica</class>
    <class>model.Usuario</class>
    <class>model.MovimentoVenda</class>
    <class>model.ProdutoMovimentadoCompra</class>
    <class>model.ProdutoMovimentadoVenda</class>
    <class>model.PessoaFisica</class>
    <class>model.MovimentoCompra</class>
    <class>model.Pessoa</class>
    <properties>
      <!-- Conexão com MySQL -->
      <!-- Configuração do EclipseLink -->
      <property name="eclipseLink.logging.level" value="FINE"/>
      <property name="eclipseLink.ddl-generation.output-mode" value="database"/>
      <property name="javax.persistence.jdbc.url" value="jdbc:sqlserver://localhost:1433;databaseName=loja;encrypt=true;trustServerCertificate=true; "/>
      <property name="javax.persistence.jdbc.user" value="loja"/>
      <property name="javax.persistence.jdbc.driver" value="com.microsoft.sqlserver.jdbc.SQLServerDriver"/>
      <property name="javax.persistence.jdbc.password" value="loja"/>
      <property name="javax.persistence.schema-generation.database.action" value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

## 2º Procedimento | Servidor Completo e Cliente Assíncrono

### Códigos usados neste roteiro

```
package cadastroclient;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import ui.SaidaFrame;
import ui.ThreadClient;

public class CadastroClientV2 {
    public static void main(String[] args) {
        try {
            Socket socket = new Socket("localhost", 4321);
            ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream ois = new ObjectInputStream(socket.getInputStream());

            oos.writeObject("op1");
            oos.flush();
            oos.writeObject("op1");
            oos.flush();

            String resposta = (String) ois.readObject();
            if (!resposta.startsWith("Usuário válido")) {
                socket.close();
                return;
            }

            SaidaFrame saida = new SaidaFrame();
            saida.setVisible(true);

            ThreadClient threadClient = new ThreadClient(ois, saida.texto);
            threadClient.start();

            BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
            while (true) {
                String comando = reader.readLine().trim();
                oos.writeObject(comando);
                oos.flush();
                if (comando.equalsIgnoreCase("X")) break;
            }

            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

## 2º Procedimento | Servidor Completo e Cliente Assíncrono

### Códigos usados neste roteiro

```
package ui;

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

public class SaidaFrame extends JDialog {
    public JTextArea texto;

    public SaidaFrame() {
        setTitle("Saída de Mensagens");
        setBounds(100, 100, 400, 300);
        setModal(false);
        texto = new JTextArea();
        add(texto);
    }
}
```



## 2º Procedimento | Servidor Completo e Cliente Assíncrono

### Códigos usados neste roteiro

```
package ui;

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

public class ThreadClient extends Thread {

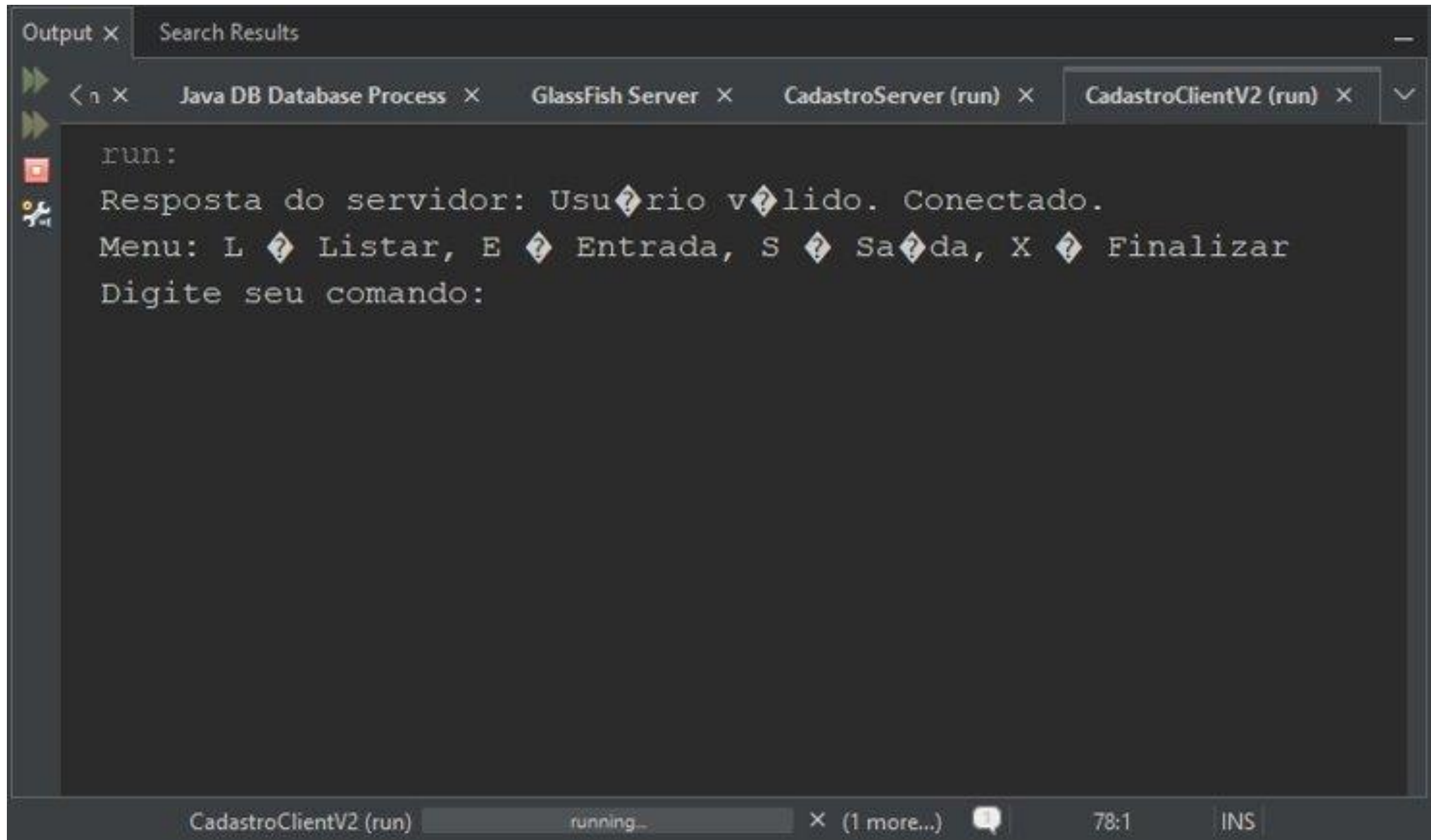
    private ObjectInputStream entrada;
    private JTextArea textArea;

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

    @Override
    public void run() {
        try {
            while (true) {
                Object obj = entrada.readObject();
                if (obj == null) {
                    break;
                }

                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("Produto: " + p.getNome() + " - Quantidade: " + p.getQuantidadeEstoque() + "\n");
                            }
                        }
                    }
                });
            }
        } catch (SocketException se) {
            SwingUtilities.invokeLater(() -> textArea.append("Conexão encerrada pelo servidor.\n"));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

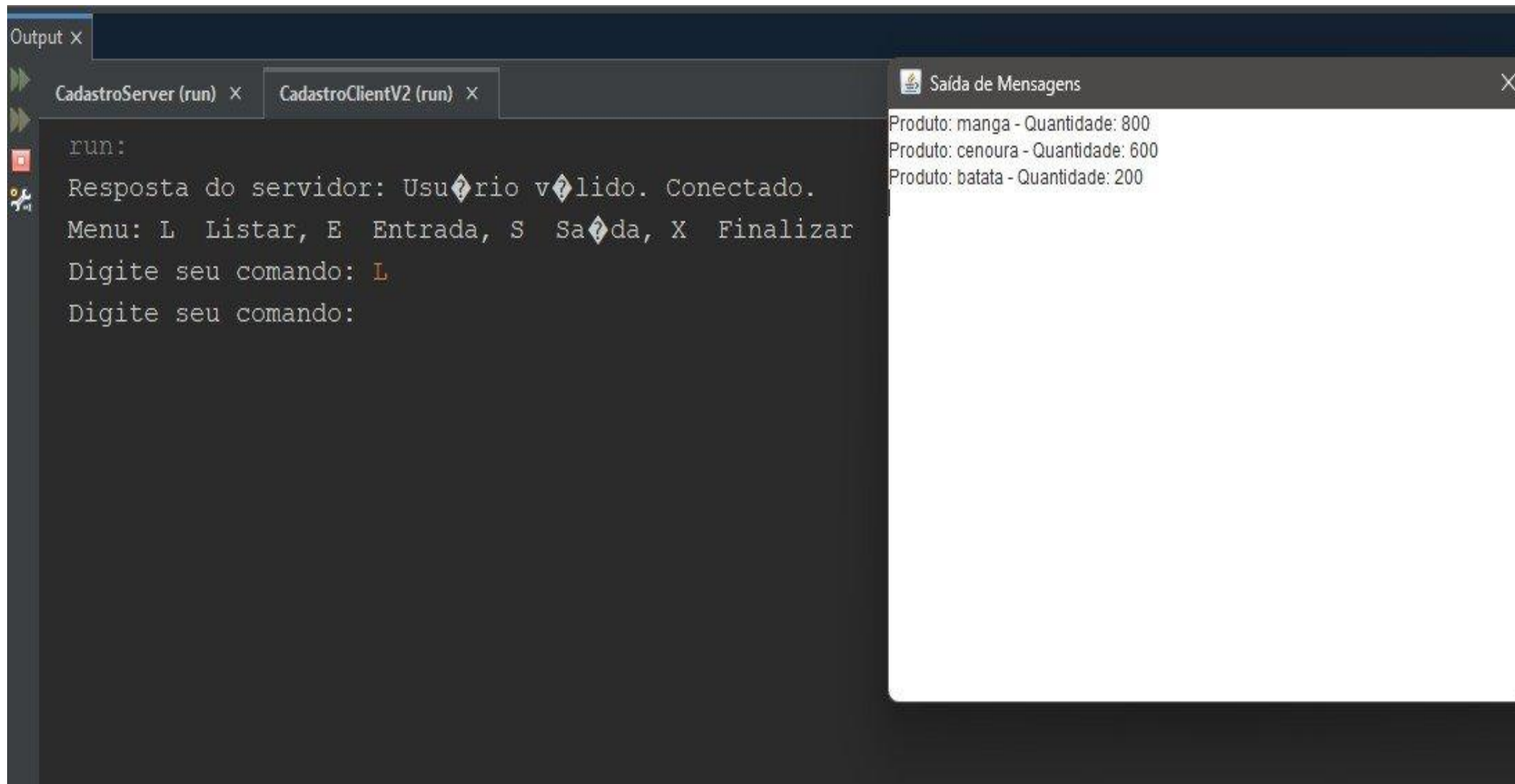
## 2º Procedimento | Servidor Completo e Cliente Assíncrono



```
run:
Resposta do servidor: Usuário válido. Conectado.
Menu: L Listar, E Entrada, S Saída, X Finalizar
Digite seu comando:
```

CadastroClientV2 (run) running... (1 more...) 78:1 INS

## 2º Procedimento | Servidor Completo e Cliente Assíncrono



```
Output X
CadastroServer (run) X CadastroClientV2 (run) X
run:
Resposta do servidor: Usuário válido. Conectado.
Menu: L Listar, E Entrada, S Saída, X Finalizar
Digite seu comando: L
Digite seu comando:
```

Saída de Mensagens

Produto: manga - Quantidade: 800  
Produto: cenoura - Quantidade: 600  
Produto: batata - Quantidade: 200

## **Análise e conclusão:**

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

O uso de Threads em Java permite implementar tratamento assíncrono de operações, como respostas enviadas pelo servidor. Isso é especialmente útil em aplicações servidoras (como servidores web ou APIs) que precisam lidar com múltiplas requisições simultaneamente , sem bloquear o fluxo principal da aplicação.

### **2. Para que serve o método invokeLater, da classe SwingUtilities?**

O método invokeLater, da classe SwingUtilities , é usado para agendar a execução de um código na Event Dispatch Thread (EDT) do Swing. Ele é fundamental no desenvolvimento de interfaces gráficas em Java para garantir que todas as atualizações da interface sejam feitas de forma segura e correta.

### **3. Como os objetos são enviados e recebidos pelo Socket Java?**

Enviar e receber objetos por meio de sockets em Java é possível graças as classes ObjectOutputStream e ObjectInputStream, que permitem serializar (Converter em bytes) e desserializar objetos para transmissão em rede.

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

Use síncrono quando a simplicidade for mais importante que a performance, como em aplicações de linha de comando ou protótipos.

Use assíncrono em aplicações GUI, servidores com múltiplos clientes ou qualquer sistema onde a responsividade e escalabilidade são essenciais.

Ambos os modelos podem ser implementados com Socket e `ObjectInputStream/ObjectOutputStream` , mas o uso de Threads ou frameworks assíncronos (como NIO ou Netty) ajuda a lidar melhor com comunicação assíncrona e não-bloqueante.

#### **LINK DO GITHUB:**

[https://github.com/mauriciocampos1234/Missao\\_Pratica\\_5\\_BackEnd\\_Estacio](https://github.com/mauriciocampos1234/Missao_Pratica_5_BackEnd_Estacio)