



# Estácio

**FACULDADE ESTÁCIO**

**CÂMPUS VOLTA REDONDA – RJ**

**DESENVOLVIMENTO FULL STACK**

**DISCIPLINA – POR QUE NÃO PARALELIZAR?**

**TURMA – 2023.2**

**SEMESTRE – 3**

**VOLTA REDONDA, SETEMBRO 2024.**

**DESENVOLVIMENTO FULL STACK**

**DISCIPLINA – POR QUE NÃO PARALELIZAR?**

**TURMA – 2023.2**

**SEMESTRE – 3**

**ALUNO – BRUNO SAMPAIO BASTOS**

**TUTOR – GUILHERME DUTRA**

**GITHUB - <https://github.com/BrunoTI-Code?tab=repositories>**

**VOLTA REDONDA, SETEMBRO 2024.**

## **1 1º PROCEDIMENTO | CRIANDO O SERVIDOR E CLIENTE DE TESTE**

### **1.1 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.

## 2 CRIAR O PROJETO DO SERVIDOR, UTILIZANDO O NOME CADASTROSERVER, DO TIPO CONSOLE, NO MODELO ANT PADRÃO, PARA IMPLEMENTAR O PROTOCOLO APRESENTADO A SEGUIR:

- Cliente conecta e envia login e senha.
- Servidor valida credenciais e, se forem inválidas, desconecta.
- Com credenciais válidas, fica no ciclo de resposta.
- Cliente envia letra L.
- Servidor responde com o conjunto de produtos.

```
package cadastroserver;

import cadastroserver.controller.MovimentoJpaController;
import cadastroserver.controller.PessoaJpaController;
import cadastroserver.controller.ProdutoJpaController;
import cadastroserver.controller.UsuarioJpaController;
import cadastroserver.model.Produto;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.List;
import java.util.logging.Logger;
import java.util.logging.Level;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class CadastroServer {

    private final int PORT = 4321;

    public CadastroServer() {

    }

    private void run() {
        try {
            ServerSocket serverSocket = new ServerSocket(PORT) {
                System.out.println("===== SERVIDOR CONECTADO - PORTA " + PORT + " =====");
                // Inicializa controladores
                EntityManagerFactory emf = Persistence.createEntityManagerFactory("CadastroServerPU");
                ProdutoJpaController produtoController = new ProdutoJpaController(emf);
                MovimentoJpaController movimentoController = new MovimentoJpaController(emf);
                PessoaJpaController pessoaController = new PessoaJpaController(emf);
                UsuarioJpaController usuarioController = new UsuarioJpaController(emf);
                while (true) {
                    System.out.println("Aguardando conexao de cliente...");
                    Socket socket = serverSocket.accept();
                    System.out.println("Cliente conectado.");
                    ClientHandler clientHandler = new ClientHandler(socket, produtoController,
                        movimentoController, pessoaController, usuarioController);
                    Thread thread = new Thread(clientHandler);
                    thread.start();
                }
            } catch (IOException e) {
                throw new RuntimeException("Erro ao criar o servidor", e);
            }
        }
    }
}
```

```

    } catch (IOException e) {
        Logger.getLogger(CadastroServer.class.getName()).log(Level.SEVERE, null, e);
    }
}

public static void main(String[] args) {
    new CadastroServer().run();
}

private class ClientHandler implements Runnable {
    private final Socket socket;
    private final ProdutoJpaController produtoController;
    private final UsuarioJpaController usuarioController;

    public ClientHandler(Socket socket, ProdutoJpaController produtoController,
        MovimentoJpaController movimentoController, PessoaJpaController pessoaController,
        UsuarioJpaController usuarioController) {
        this.socket = socket;
        this.produtoController = produtoController;
        this.usuarioController = usuarioController;
    }

    @Override
    public void run() {
        try {
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true)
        } {
            // Autenticacao
            String username = in.readLine();
            String password = in.readLine();
            if (validateCredentials(username, password)) {
                out.println("Autenticacao bem-sucedida. Aguardando comandos...");
                boolean outerSign = false;
                while (!outerSign) {
                    String command = in.readLine();
                    if (command != null) {
                        switch (command) {
                            case "L": sendProductList(out); break; // Enviar conjunto de produtos do banco de dados
                            case "S": outerSign = true; break; // Comando para sair
                            default: break;
                        }
                    }
                }
            } else {
                try (socket) {
                    out.println("Credenciais invalidas. Conexao encerrada.");
                }
            }
        }
        out.println("Credenciais invalidas. Conexao encerrada.");
    }

    } catch (IOException e) {
        Logger.getLogger(ClientHandler.class.getName()).log(Level.SEVERE, null, e);
    } catch (Exception e) {
        Logger.getLogger(ClientHandler.class.getName()).log(Level.SEVERE, null, e);
    }
}

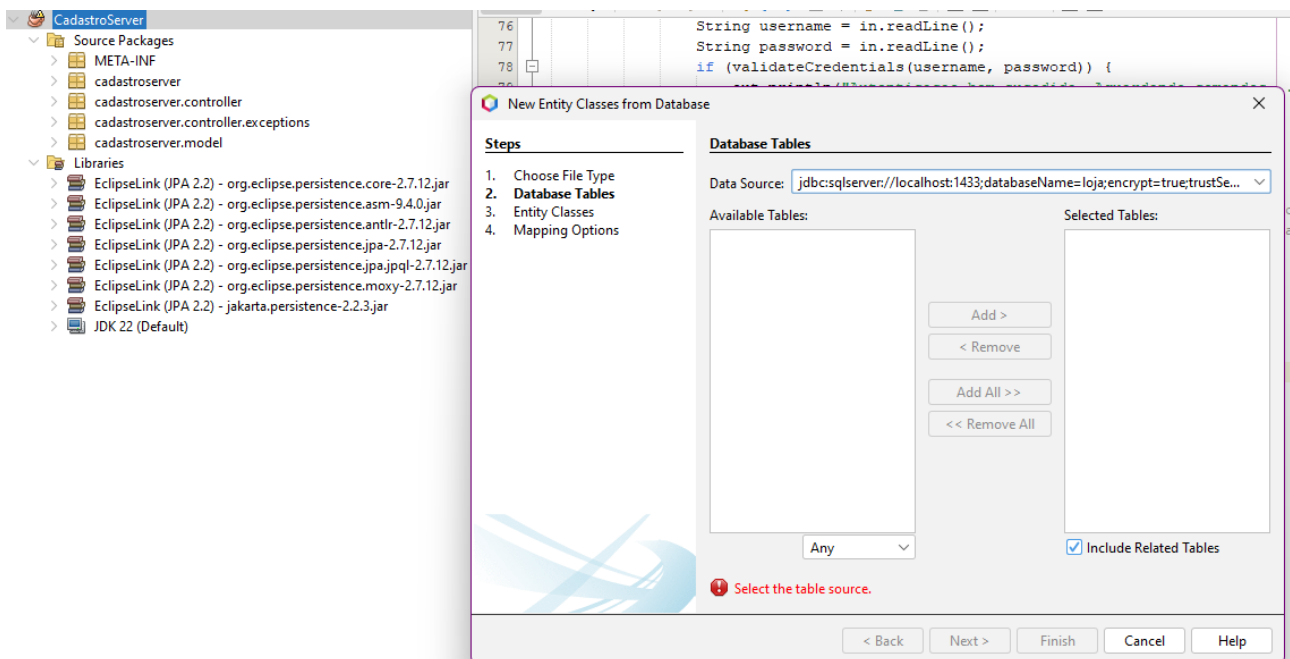
private boolean validateCredentials(String username, String password) {
    return usuarioController.validarUsuario(username, password) != null;
}

private void sendProductList(PrintWriter out) {
    List<Produto> productList = produtoController.findProdutoEntities();
    out.println("Conjunto de produtos disponiveis:");
    for (Produto product : productList) {
        out.println(product.getNome());
    }
    out.println();
}
}
}

```

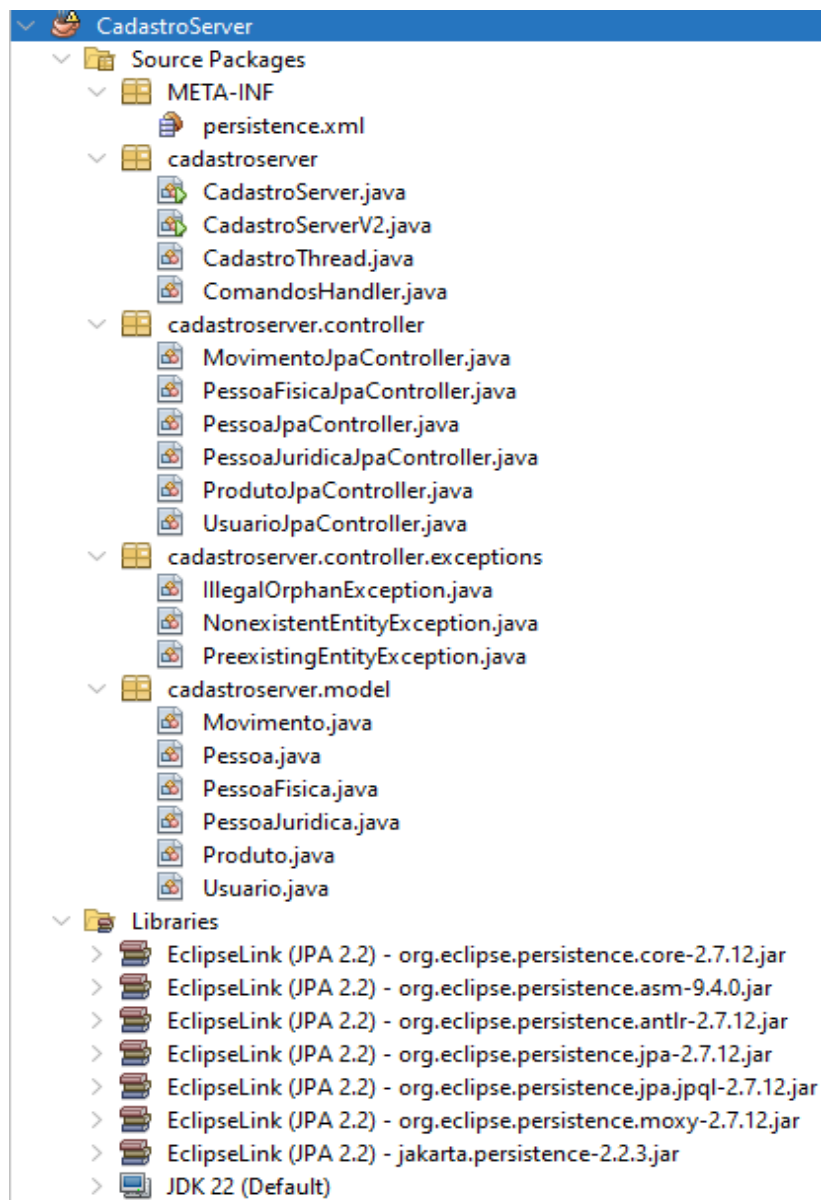
### 3 CRIAR A CAMADA DE PERSISTÊNCIA EM CADASTROSERVER.

- Criar o pacote model para implementação das entidades.
- Utilizar a opção New..Entity Classes from Database.
- Selecionar a conexão com o SQL Server, previamente configurada na aba Services, divisão Databases, do NetBeans e adicionar todas as tabelas.
- Acrescentar ao projeto a biblioteca Eclipse Link (JPA 2.1).
- Acrescentar o arquivo jar do conector JDBC para o SQL Server.
- **Observação! Por não executar o servidor sob o Tomcat, não será necessário ajustar os pacotes para a distribuição do Jakarta.**



#### 4 CRIAR A CAMADA DE CONTROLE EM CADASTROSERVER:

- Criar o pacote controller para implementação dos controladores.
- Utilizar a opção New...JPA Controller Classes from Entity Classes.
- Na classe UsuarioJpaController, adicionar o método findUsuario, tendo como parâmetros o login e a senha, retornando o usuário a partir de uma consulta JPA, ou nulo, caso não haja um usuário com as credenciais.
- Ao final o projeto ficará como o que é apresentado a seguir.



## **5 NO PACOTE PRINCIPAL, CADASTROSERVER, ADICIONAR A THREAD DE COMUNICAÇÃO, COM O NOME CADASTROTHREAD.**

- Acrescentar os atributos ctrl e ctrlUsu, dos tipos ProdutoJpaController e UsuarioJpaController, respectivamente.
- Acrescentar o atributo s1 para receber o Socket.
- Definir um construtor recebendo os controladores e o Socket, com a passagem dos valores para os atributos internos.
- Implementar o método run para a Thread, cujo funcionamento será o descrito a seguir.
- Encapsular os canais de entrada e saída do Socket em objetos dos tipos ObjectOutputStream (saída) e ObjectInputStream (entrada).
- Obter o login e a senha a partir da entrada.
- Utilizar ctrlUsu para verificar o login, terminando a conexão caso o retorno seja nulo.
- Com o usuário válido, iniciar o loop de resposta, que deve obter o comando a partir da entrada.
- Caso o comando seja L, utilizar ctrl para retornar o conjunto de produtos através da saída.



```

package cadastrserver;

import cadastrserver.controller.UsuarioJpaController;
import cadastrserver.model.Usuario;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.logging.Level;
import java.util.logging.Logger;

public class CadastroThread extends Thread {

    private final UsuarioJpaController ctrlUsu;

    private final Socket s1;

    public CadastroThread(UsuarioJpaController ctrlUsu, Socket s1) {
        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();

            Date dataAtual = new Date();
            SimpleDateFormat formato = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
            String dataFormatada = formato.format(dataAtual);
            System.out.println("==== Nova Comunicacao >> " + dataFormatada);

            boolean usuarioValido = (validar(login, senha) != null);

            if (usuarioValido) {
                out.writeObject(usuarioValido);
                out.writeObject(validar(login, senha).getIdUsuario());

                System.out.println("==== Usuario Logado ====");

                ComandosHandler comandos = new ComandosHandler(out, in);
                comandos.executarComandos();

                comandos.executarComandos();

            } else {
                out.writeObject(usuarioValido);

                out.writeObject(null);

            }

            out.flush();

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

    private Usuario validar(String login, String senha) {
        return ctrlUsu.validarUsuario(login, senha);
    }

}

```

## 6 IMPLEMENTAR A CLASSE DE EXECUÇÃO (MAIN), UTILIZANDO AS CARACTERÍSTICAS QUE SÃO APRESENTADAS A SEGUIR.

- Instanciar um objeto do tipo EntityManagerFactory a partir da unidade de persistência.
- Instanciar o objeto ctrl, do tipo ProdutoJpaController.
- Instanciar o objeto ctrlUsu do tipo UsuarioJpaController.
- Instanciar um objeto do tipo ServerSocket, escutando a porta 4321.
- Dentro de um loop infinito, obter a requisição de conexão do cliente, instanciar uma Thread, com a passagem de ctrl, ctrlUsu e do Socket da conexão, iniciando-a em seguida.
- Com a Thread respondendo ao novo cliente, o servidor ficará livre para escutar a próxima solicitação de conexão.

```
private void run() {
    try (ServerSocket serverSocket = new ServerSocket(PORT)) {
        System.out.println("===== SERVIDOR CONECTADO - PORTA " + PORT + " =====");
        // Inicializa controladores
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("CadastroServerPU");
        ProdutoJpaController produtoController = new ProdutoJpaController(emf);
        MovimentoJpaController movimentoController = new MovimentoJpaController(emf);
        PessoaJpaController pessoaController = new PessoaJpaController(emf);
        UsuarioJpaController usuarioController = new UsuarioJpaController(emf);
        while (true) {
            System.out.println("Aguardando conexao de cliente...");
            Socket socket = serverSocket.accept();
            System.out.println("Cliente conectado.");
            ClientHandler clientHandler = new ClientHandler(socket, produtoController,
                movimentoController, pessoaController, usuarioController);
            Thread thread = new Thread(clientHandler);
            thread.start();
        }
    } catch (IOException e) {
        Logger.getLogger(CadastroServer.class.getName()).log(Level.SEVERE, null, e);
    }
}
```

**7 CRIAR O CLIENTE DE TESTE, UTILIZANDO O NOME CADASTROCLIENT, DO TIPO CONSOLE, NO MODELO ANT PADRÃO, PARA IMPLEMENTAR A FUNCIONALIDADE APRESENTADA A SEGUIR:**

- Instanciar um Socket apontando para localhost, na porta 4321.
- Encapsular os canais de entrada e saída do Socket em objetos dos tipos ObjectOutputStream (saída) e ObjectInputStream (entrada).
- Escrever o login e a senha na saída, utilizando os dados de algum dos registros da tabela de usuários (op1/op1).
- Enviar o comando L no canal de saída.
- Receber a coleção de entidades no canal de entrada.
- Apresentar o nome de cada entidade recebida.
- Fechar a conexão.

```

package cadastrocliente;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.logging.Logger;
import java.util.logging.Level;

public class CadastroCliente {

    private final String ADDRESS = "localhost";
    private final int PORT = 4321;

    public CadastroCliente() {

    }

    private void run() {
        try {
            Socket socket = new Socket(ADDRESS, PORT);
            BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader consoleIn = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Conectado ao servidor CadastroServer.");
            System.out.print("Digite seu nome de usuario: ");
            String username = consoleIn.readLine();
            System.out.print("Digite sua senha: ");
            String password = consoleIn.readLine();
            out.println(username);
            out.println(password);
            String response = in.readLine();
            System.out.println(response);
            if (response.equals("Autenticacao bem-sucedida. Aguardando comandos...")) {
                boolean exitChoice = false;
                while (!exitChoice) {
                    System.out.print("Digite 'L' para listar produtos ou 'S' para sair: ");
                    String command = consoleIn.readLine().toUpperCase();
                    out.println(command);
                    switch (command) {
                        case "S" -> exitChoice = true;
                        case "L" -> receiveAndDisplayProductList(in);
                        default -> System.out.println("Opcao invalida!");
                    }
                }
            }
        } catch (IOException e) {
            } catch (IOException e) {
                Logger.getLogger(CadastroCliente.class.getName()).log(Level.SEVERE, null, e);
            }
        }

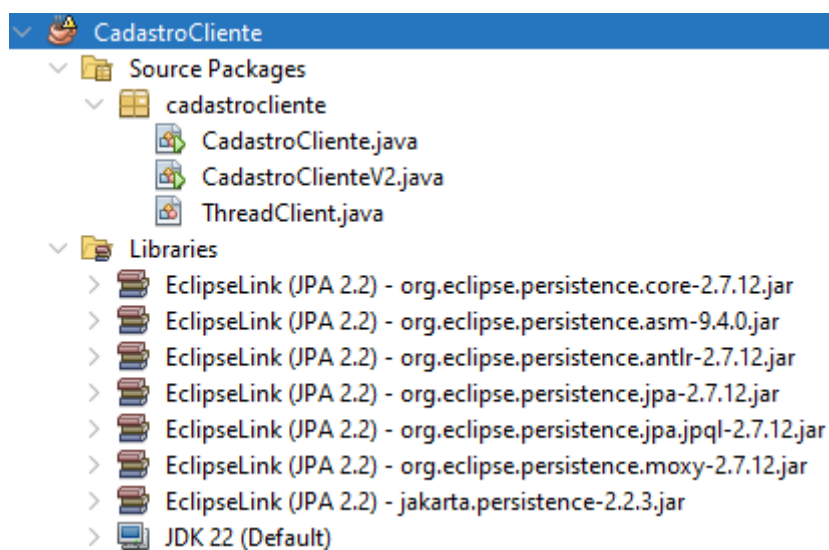
    private void receiveAndDisplayProductList(BufferedReader in) throws IOException {
        String line;
        while ((line = in.readLine()) != null && !line.isEmpty()) {
            System.out.println(line);
        }
    }

    public static void main(String[] args) {
        new CadastroCliente().run();
    }
}

```

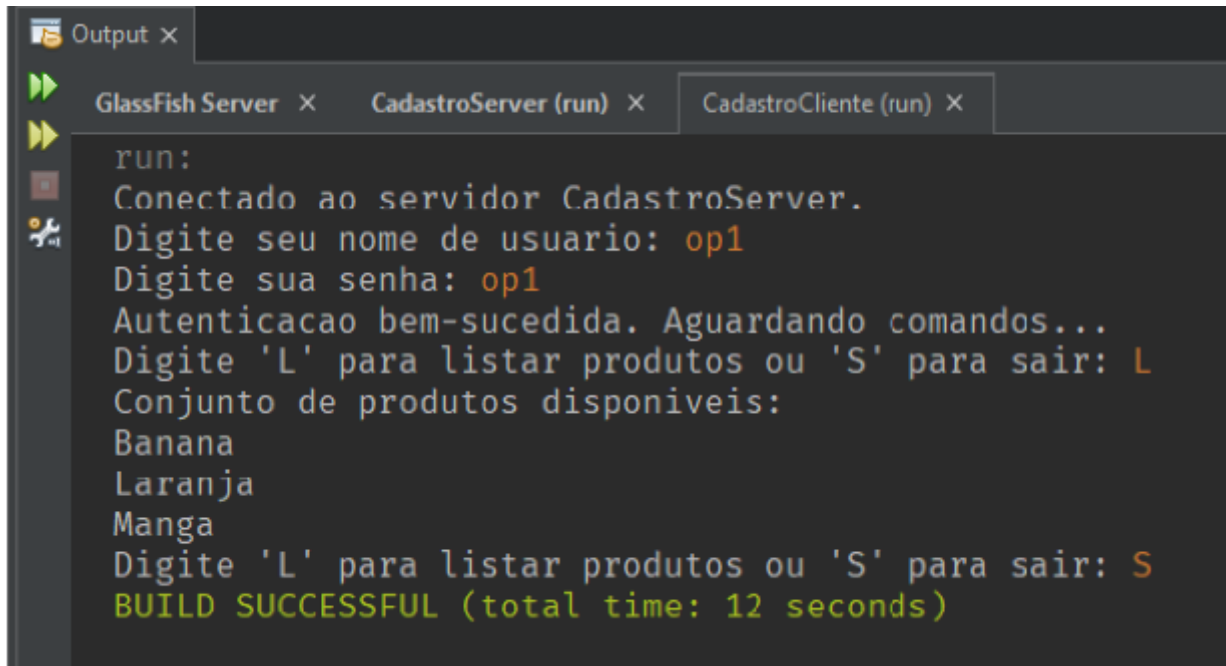
## 8 CONFIGURAR O PROJETO DO CLIENTE PARA USO DAS ENTIDADES:

- Copiar o pacote model do projeto servidor para o cliente.
- Adicionar a biblioteca Eclipse Link (JPA 2.1).
- A configuração final pode ser observada a seguir.



## 9 TESTAR O SISTEMA CRIADO, COM A EXECUÇÃO DOS DOIS PROJETOS:

- Executar o projeto servidor.
- Executar, em seguida, o cliente.
- A saída do cliente deverá ser como a que é apresentada a seguir.



```
run:
Conectado ao servidor CadastroServer.
Digite seu nome de usuario: op1
Digite sua senha: op1
Autenticacao bem-sucedida. Aguardando comandos...
Digite 'L' para listar produtos ou 'S' para sair: L
Conjunto de produtos disponiveis:
Banana
Laranja
Manga
Digite 'L' para listar produtos ou 'S' para sair: S
BUILD SUCCESSFUL (total time: 12 seconds)
```

## 10 ANÁLISE E CONCLUSÃO:

### 10.1 COMO FUNCIONAM AS CLASSES SOCKET E SERVERSOCKET?

As classes Socket e ServerSocket são fundamentais na criação de aplicações cliente-servidor em Java. Elas são usadas para realizar comunicação via rede através do protocolo TCP/IP, permitindo a troca de dados entre máquinas de maneira confiável.

#### **Classe Socket (Cliente)**


A classe Socket representa o cliente em uma conexão de rede. Ela é usada para conectar a um servidor remoto e trocar dados.

#### **Funcionamento:**

##### **Criando a Conexão:**

Quando um cliente quer se conectar a um servidor, ele cria um objeto Socket e tenta se conectar ao servidor, especificando o endereço IP (ou nome do host) e o número da porta do servidor.

java

 Copiar código


```
Socket socket = new Socket("endereco-do-servidor", 12345);
```

Esse código tenta abrir uma conexão TCP na porta 12345 do servidor cujo endereço é "endereco-do-servidor".

### ***Entrada e Saída de Dados:***

Após estabelecer a conexão, o cliente pode enviar e receber dados por meio dos fluxos de entrada e saída associados ao Socket.

java

 Copiar código

```
InputStream input = socket.getInputStream();  
OutputStream output = socket.getOutputStream();
```


Isso permite que o cliente envie dados ao servidor (usando o fluxo de saída) e receba respostas (usando o fluxo de entrada).

### ***Fechamento da Conexão:***

Quando a comunicação termina, o cliente deve fechar o Socket para liberar recursos e encerrar a conexão.

java



 Copiar código

```
socket.close();
```



## **Classe `ServerSocket` (Servidor)**

A classe `ServerSocket` é usada para representar o servidor. Ela aguarda conexões de clientes e cria um `Socket` para cada cliente que se conecta.

### **Funcionamento:**

#### **Criando o Servidor:**

Um servidor é criado abrindo um `ServerSocket` em uma porta específica, onde ele ficará ouvindo por conexões de clientes.

```
java Copiar código  
  
ServerSocket serverSocket = new ServerSocket(12345);
```

Esse servidor ficará ouvindo conexões na porta 12345.

#### **Aceitando Conexões:**

O servidor chama o método `accept()` para esperar e aceitar conexões de clientes. Esse método é bloqueante, ou seja, ele só continua após um cliente se conectar.


```
java Copiar código  
  
Socket clienteSocket = serverSocket.accept();
```

Quando um cliente se conecta, o método retorna um `Socket` que representa essa conexão.

### ***Entrada e Saída de Dados:***

O servidor pode então usar o Socket retornado pelo método `accept()` para se comunicar com o cliente. Ele acessa os fluxos de entrada e saída para enviar e receber dados.

java


 Copiar código

```
InputStream input = clienteSocket.getInputStream();  
OutputStream output = clienteSocket.getOutputStream();
```

### ***Fechamento do Servidor:***

Quando o servidor não precisa mais aceitar conexões, o `ServerSocket` pode ser fechado.

java

 Copiar código

```
serverSocket.close();
```

### ***Principais Pontos:***

- **Socket:** Utilizado pelo cliente para conectar-se ao servidor.
- **ServerSocket:** Usado pelo servidor para esperar conexões de clientes.
- **Fluxos de entrada e saída:** São usados para enviar e receber dados.
- **accept():** No servidor, aguarda a conexão de um cliente.
- **Conexão TCP:** Protocolo utilizado para garantir entrega confiável dos dados.

## 10.2 QUAL A IMPORTÂNCIA DAS PORTAS PARA A CONEXÃO COM SERVIDORES?

### ***importância das Portas na Conexão com Servidores***

#### ***Identificação de Serviços:***

Cada servidor pode executar vários serviços ou aplicações simultaneamente, como um servidor web, de banco de dados, ou de email. As portas são números que ajudam a identificar qual serviço o cliente está tentando acessar.

#### **Por exemplo:**

- Porta 80: HTTP (servidor web).
- Porta 443: HTTPS (servidor web seguro).
- Porta 3306: MySQL (banco de dados).
- Porta 25: SMTP (servidor de email).

#### ***Endereçamento Específico:***

Quando um cliente quer se conectar a um servidor, ele precisa não apenas do endereço IP (que identifica o servidor), mas também do número da porta (que identifica o serviço específico).

#### **Exemplo:**

<http://192.168.0.1:80/>: O navegador se conecta ao endereço IP 192.168.0.1 na porta 80, que é o serviço HTTP.

### ***Multiplexação de Conexões:***

Um servidor pode atender a vários clientes simultaneamente. As portas permitem multiplexar essas conexões, de forma que o servidor possa manter e diferenciar várias comunicações ao mesmo tempo.

Isso é possível porque cada conexão cliente-servidor usa um par de endereço IP + porta no servidor e no cliente, criando conexões exclusivas.

### ***Segurança e Controle de Acesso:***

As portas também são usadas para controle de acesso e segurança. Um firewall, por exemplo, pode bloquear ou permitir o tráfego em determinadas portas, limitando quais serviços podem ser acessados a partir de fora da rede.

Serviços sensíveis podem ser configurados para escutar em portas não padrão (ou portas altas) para reduzir a exposição a ataques automáticos que visam portas comumente usadas (como 22 para SSH).

### ***Organização e Padronização:***

Muitos serviços de rede usam portas padrão estabelecidas pela IANA (Internet Assigned Numbers Authority), o que facilita a organização e padronização do uso das portas.

Isso significa que um cliente que queira acessar um serviço de email saberá que pode se conectar à porta 25 (SMTP) por padrão, enquanto para um banco de dados MySQL, ele usará a porta 3306.

### ***Separação de Tráfego:***

O uso de diferentes portas permite que múltiplos serviços no mesmo servidor possam operar sem interferir uns nos outros. Por exemplo, um servidor pode rodar um site na porta 80 e um serviço de banco de dados na porta 5432, e os dois podem ser acessados simultaneamente sem conflito.

### 10.3 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` em Java são utilizadas para leitura e gravação de objetos em fluxos de entrada e saída, como arquivos ou redes. Elas permitem que objetos Java sejam transmitidos de uma forma que preserve seus estados.

#### ***Finalidade das Classes:***

##### ***ObjectOutputStream:***

É usada para gravar objetos em um fluxo de saída. Ao escrever um objeto, seus dados são convertidos para um formato binário que pode ser transmitido ou armazenado.

##### ***ObjectInputStream:***

É usada para ler objetos de um fluxo de entrada. Ela reconstrói o objeto a partir do formato binário lido do fluxo.

### ***Por que os Objetos Devem Ser Serializáveis?***

Para que um objeto possa ser transmitido ou armazenado através dessas classes, ele deve implementar a interface Serializable. Isso é necessário porque:

#### ***Serialização:***

A serialização é o processo de converter um objeto em um fluxo de bytes. Isso permite que o estado do objeto seja salvo ou enviado por uma rede.

#### ***Recriação do Objeto:***

Ao implementar a interface Serializable, o objeto pode ser recriado (desserializado) em outro ponto no tempo ou em outro ambiente, preservando seus atributos e estado.

#### 10.4 POR QUE, MESMO UTILIZANDO AS CLASSES DE ENTIDADES JPA NO CLIENTE, FOI POSSÍVEL GARANTIR O ISOLAMENTO DO ACESSO AO BANCO DE DADOS?

Mesmo utilizando as classes de entidades JPA no cliente, é possível garantir o isolamento do acesso ao banco de dados devido à arquitetura e aos mecanismos de controle do JPA e do próprio servidor de aplicação.

##### ***Motivos que garantem o isolamento:***

###### ***Transações gerenciadas pelo servidor:***

Em um ambiente típico de JPA, as transações são gerenciadas pelo servidor de aplicação ou por contêineres como o EJB. Isso significa que, mesmo que as entidades JPA sejam manipuladas no cliente, o controle de transações — que garante o isolamento — ocorre no servidor.

Isolamento de transação garante que as operações realizadas por uma transação não interfiram em outras transações até que sejam concluídas (commit).

###### ***Objetos de entidade desconectados:***

Quando uma entidade JPA é utilizada no cliente, ela geralmente é tratada como um objeto desconectado do contexto de persistência. Isso significa que, enquanto o cliente manipula a entidade, nenhuma operação é diretamente realizada no banco de dados.

As operações de inserção, atualização ou remoção no banco de dados ocorrem somente quando o objeto é reanexado a um contexto de persistência no servidor e uma transação é aberta.



### ***Controle de transações no servidor:***

O acesso real ao banco de dados ocorre somente quando a transação é iniciada no servidor. Isso permite que o controle de concorrência e isolamento do banco de dados seja aplicado de maneira centralizada, garantindo que o cliente não tenha acesso direto à manipulação dos dados.

### ***Lazy loading e proxies:***

Algumas associações entre entidades são carregadas de forma preguiçosa (*lazy loading*), o que significa que os dados relacionados não são acessados diretamente pelo cliente. O carregamento efetivo ocorre quando o objeto é reanexado ao contexto de persistência no servidor, mantendo o isolamento da camada de banco.

## 11 2º PROCEDIMENTO | SERVIDOR COMPLETO E CLIENTE ASSÍNCRONO

### 12 CRIAR UMA SEGUNDA VERSÃO DA THREAD DE COMUNICAÇÃO, NO PROJETO DO SERVIDOR, COM O ACRÉSCIMO DA FUNCIONALIDADE APRESENTADA A SEGUIR:

- Servidor recebe comando E, para entrada de produtos, ou S, para saída.
- Gerar um objeto Movimento, configurado com o usuário logado e o tipo, que pode ser E ou S.
  - Receber o Id da pessoa e configurar no objeto Movimento.
  - Receber o Id do produto e configurar no objeto Movimento.
  - Receber a quantidade e configurar no objeto Movimento.
  - Receber o valor unitário e configurar no objeto Movimento.
  - Persistir o movimento através de um MovimentoJpaController com o nome ctrlMov.
- Atualizar a quantidade de produtos, de acordo com o tipo de movimento, através de ctrlProd.
- **Observação! Devem ser acrescentados os atributos ctrlMov e ctrlPessoa, dos tipos MovimentoJpaController e PessoaJpaController, alimentados por meio de parâmetros no construtor**

```

package cadastroserver;

import cadastroserver.controller.MovimentoJpaController;
import cadastroserver.controller.PessoaJpaController;
import cadastroserver.controller.ProdutoJpaController;
import cadastroserver.controller.UsuarioJpaController;
import cadastroserver.controller.exceptions.NonexistentEntityException;
import cadastroserver.model.Movimento;
import cadastroserver.model.Pessoa;
import cadastroserver.model.Produto;
import cadastroserver.model.Usuario;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class ComandosHandler {

    private final ObjectOutputStream out;
    private final ObjectInputStream in;
    private final EntityManagerFactory emf = Persistence.createEntityManagerFactory("CadastroServerPU");

    private final MovimentoJpaController ctrlMov;
    private final PessoaJpaController ctrlPessoa;
    private final ProdutoJpaController ctrlProduto;
    private final UsuarioJpaController ctrlUsur;

    public ComandosHandler(ObjectOutputStream out, ObjectInputStream in) {
        this.out = out;
        this.in = in;

        this.ctrlMov = new MovimentoJpaController(emf);
        this.ctrlPessoa = new PessoaJpaController(emf);
        this.ctrlProduto = new ProdutoJpaController(emf);
        this.ctrlUsur = new UsuarioJpaController(emf);
    }

    public void executarComandos() throws IOException, ClassNotFoundException {
        try (out) {
            while (true) {
                String comando = (String) in.readObject();
                comando = comando.toLowerCase();
            }
        }
    }
}

```

```

comando = comando.toUpperCase();
Integer idPessoa;
Integer idUsuario;
Integer idProduto;
Integer quantidade;
Float valor_unitario;

Pessoa pessoa;
Produto produto;
Usuario usuario;

Movimento movimento;

switch (comando) {
    case "E" -> {
        idPessoa = Integer.valueOf((String) in.readObject());
        idProduto = Integer.valueOf((String) in.readObject());
        idUsuario = (Integer) in.readObject();
        quantidade = Integer.valueOf((String) in.readObject());
        valor_unitario = Float.valueOf((String) in.readObject());

        pessoa = ctrlPessoa.findpessoa(idPessoa);
        produto = ctrlProduto.findProduto(idProduto);
        usuario = ctrlUsur.findUsuario(idUsuario);

        if (produto == null) {
            System.out.println("Produto nao cadastrado! no banco de dados.");
            continue;
        }

        movimento = new Movimento();
        movimento.setIdPessoa(pessoa);
        movimento.setIdProduto(produto);
        movimento.setQuantidade(quantidade);
        movimento.setIdUsuario(usuario);
        movimento.setTipo("E");
        movimento.setvalor_unitario(valor_unitario);

        produto.setQuantidade(produto.getQuantidade() + quantidade);
        ctrlProduto.edit(produto);

        ctrlMov.create(movimento);
    }

    case "S" -> {
        idPessoa = Integer.valueOf((String) in.readObject());

```

```

        idPessoa = Integer.valueOf((String) in.readObject());
        idProduto = Integer.valueOf((String) in.readObject());
        idUsuario = (Integer) in.readObject();
        quantidade = Integer.valueOf((String) in.readObject());
        valor_unitario = Float.valueOf((String) in.readObject());

        pessoa = ctrlPessoa.findpessoa(idPessoa);
        produto = ctrlProduto.findProduto(idProduto);
        usuario = ctrlUsur.findUsuario(idUsuario);

        if (produto == null) {
            System.out.println("Produto não cadastrado!");
            continue;
        }

        movimento = new Movimento();
        movimento.setIdPessoa(pessoa);
        movimento.setIdProduto(produto);
        movimento.setQuantidade(quantidade);
        movimento.setIdUsuario(usuario);
        movimento.setTipo("S");
        movimento.setvalor_unitario(valor_unitario);

        produto.setQuantidade(produto.getQuantidade() - quantidade);
        ctrlProduto.edit(produto);

        ctrlMov.create(movimento);
    }
    case "L" -> {
        List<Produto> produtoList = ctrlProduto.findProdutoEntities();

        ArrayList<String> produtoName = new ArrayList<>();
        ArrayList<Integer> produtoQuantidade = new ArrayList<>();

        for (Produto item : produtoList) {
            produtoName.add(item.getNome());
            produtoQuantidade.add(item.getQuantidade());
        }
        out.writeObject(produtoName);
        out.writeObject(produtoQuantidade);
    }
}

} catch (IOException e) {
}

} catch (IOException e) {
    e.printStackTrace();
} catch (NonexistentEntityException ex) {
    Logger.getLogger(ComandosHandler.class.getName()).log(Level.SEVERE, null, ex);
} catch (Exception ex) {
    Logger.getLogger(ComandosHandler.class.getName()).log(Level.SEVERE, null, ex);
} finally {
    in.close();
}
}
}

```

**13 ACRESCENTAR OS CONTROLADORES NECESSÁRIOS NA CLASSE PRINCIPAL, MÉTODO MAIN, E TROCAR A INSTÂNCIA DA THREAD ANTERIOR PELA NOVA THREAD NO LOOP DE CONEXÃO.**

- Criar o cliente assíncrono, utilizando o nome CadastroClientV2, do tipo console, no modelo Ant padrão, para implementar a funcionalidade apresentada a seguir:
  - Instanciar um Socket apontando para localhost, na porta 4321.
  - Encapsular os canais de entrada e saída do Socket em objetos dos tipos ObjectOutputStream (saída) e ObjectInputStream (entrada).
  - Escrever o login e a senha na saída, utilizando os dados de algum dos registros da tabela de usuários (op1/op1) .
  - Encapsular a leitura do teclado em um BufferedReader.
  - Instanciar a janela para apresentação de mensagens (Passo 4) e a Thread para preenchimento assíncrono (Passo 5), com a passagem do canal de entrada do Socket.
- Apresentar um menu com as opções: L – Listar, X – Finalizar, E – Entrada, S – Saída.
  - Receber o comando a partir do teclado.
  - Para o comando L, apenas enviá-lo para o servidor.
  - Para os comandos E ou S, enviar para o servidor e executar os seguintes passos:
    - Obter o Id da pessoa via teclado e enviar para o servidor.
    - Obter o Id do produto via teclado e enviar para o servidor.
    - Obter a quantidade via teclado e enviar para o servidor.
    - Obter o valor unitário via teclado e enviar para o servidor.
  - j. Voltar ao passo f até que o comando X seja escolhido

```

package cadastrocliente;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;

public class CadastroClienteV2 {

    private final String ADDRESS = "localhost";
    private final int PORT = 4321;

    public CadastroClienteV2() {

    }

    private void run() {
        try {
            Socket socket = new Socket(ADDRESS, PORT);
            ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
            ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
            BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("=====");
            System.out.print("Login: ");
            String login = reader.readLine();
            System.out.print("Senha: ");
            String senha = reader.readLine();
            System.out.println("=====");
            out.writeObject(login);
            out.writeObject(senha);
            out.flush();
            ThreadClient threadClient = new ThreadClient(in,out);
            threadClient.start();
        } catch (IOException ex) {
            Logger.getLogger(CadastroClienteV2.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    public static void main(String[] args) {
        new CadastroClienteV2().run();
    }
}

```

## 14 CRIAR A JANELA PARA APRESENTAÇÃO DAS MENSAGENS:

- Definir a classe SaidaFrame como descendente de JDialog
- Acrescentar um atributo público do tipo JTextArea, com o nome texto
- Ao nível do construtor, efetuar os passos apresentados a seguir:
- Definir as dimensões da janela via setBounds
- Definir o status modal como false
- Acrescentar o componente JTextArea na janela

```
ArrayList<String> produtoList = (ArrayList<String>) in.readObject();
ArrayList<Integer> produtoQuantidade = (ArrayList<Integer>) in.readObject();
if (frame == null || !frame.isVisible()) {
    frame = new JFrame("Retorno do Servidor");
    frame.setSize(400, 600);
    textArea = new JTextArea(20, 50);
    textArea.setEditable(false);
    frame.add(new JScrollPane(textArea));
    frame.pack();
    frame.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
    frame.setVisible(true);
    frame.setVisible(true);
    SwingUtilities.invokeLater(() -> {
        output.add("Lista de Produtos:\n");
        for (int i = 0; i < produtoList.size(); i++) {
            output.add(produtoList.get(i) + " " + produtoQuantidade.get(i) + "\n");
        }
        for (String line : output) {
            textArea.append(line);
        }
        textArea.setCaretPosition(textArea.getDocument().getLength());
    });
} else {
    frame.setVisible(false);
}
```



**15 DEFINIR A THREAD DE PREENCHIMENTO ASSÍNCRONO, COM O NOME THREADCLIENT, DE ACORDO COM AS CARACTERÍSTICAS APRESENTADAS A SEGUIR :**

- Adicionar o atributo entrada, do tipo ObjectInputStream, e textArea, do tipo JTextArea, que devem ser preenchidos via construtor da Thread.
- Alterar o método run, implementando um loop de leitura contínua.
- Receber os dados enviados pelo servidor via método readObject.
- Para objetos do tipo String, apenas adicionar ao JTextArea.
- Para objetos do tipo List, acrescentar o nome e a quantidade de cada produto ao JTextArea.
- Observação! É necessário utilizar invokeLater nos acessos aos componentes do tipo Swing.

```
package cadastrocliente;
```

```
import java.awt.HeadlessException;  
import java.io.BufferedReader;  
import java.io.IOException;  
import java.io.InputStreamReader;  
import java.io.ObjectInputStream;  
import java.io.ObjectOutputStream;  
import java.util.ArrayList;  
import javax.swing.JFrame;  
import javax.swing.JScrollPane;  
import javax.swing.JTextArea;  
import javax.swing.SwingUtilities;
```

```
public class ThreadClient extends Thread {
```

```
    private ObjectOutputStream out = null;  
    private ObjectInputStream in = null;
```

```
    private JTextArea textArea;  
    private JFrame frame;  
    private ArrayList<String> output;
```

```
    public ThreadClient() {  
    }  
}
```

```
    public ThreadClient(ObjectInputStream in, ObjectOutputStream out) {  
        this.in = in;  
        this.out = out;  
    }  
}
```

```
@Override
```

```
    public void run() {  
        output = new ArrayList<>();  
        try {  
            Boolean validate = (Boolean) in.readObject();  
            Integer idUsuario = (Integer) in.readObject();  
  
            if (validate) {  
                BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));  
  
                String comando;  
                String idPessoa;  
                String idProduto;  
                String quantidade;  
                String valor unitario;
```

```

String idProduto;
String quantidade;
String valor_unitario;

do {
    System.out.print("""
        ===== Comandos =====

        L - Listar
        F - Finalizar
        E - Entrada
        S - Saida

        Digite o comando: """);
    comando = reader.readLine().toUpperCase();

    switch (comando) {
        case "E" -> {
            out.writeObject("E");

            System.out.println("===== Entrada =====");

            System.out.print("ID Pessoa: ");
            idPessoa = reader.readLine();
            out.writeObject(idPessoa);

            System.out.print("ID Produto: ");
            idProduto = reader.readLine();
            out.writeObject(idProduto);

            System.out.print("ID Usuario: " + idUsuario);
            out.writeObject(idUsuario);
            System.out.println("");

            System.out.print("Quantidade: ");
            quantidade = reader.readLine();
            out.writeObject(quantidade);

            System.out.print("Valor Unitario: ");
            valor_unitario = reader.readLine();
            out.writeObject(valor_unitario);

            output.add("Entrada realizada com sucesso.\n");
        }

        case "S" -> {
            out.writeObject("S");

```

```

case "S" -> {
    out.writeObject("S");

    System.out.println("===== Saida =====");

    System.out.print("ID Pessoa: ");
    idPessoa = reader.readLine();
    out.writeObject(idPessoa);

    System.out.print("ID Produto: ");
    idProduto = reader.readLine();
    out.writeObject(idProduto);

    System.out.print("ID Usuario: " + idUsuario);
    out.writeObject(idUsuario);
    System.out.println("");

    System.out.print("Quantidade: ");
    quantidade = reader.readLine();
    out.writeObject(quantidade);

    System.out.print("Valor Unitario: ");
    valor_unitario = reader.readLine();
    out.writeObject(valor_unitario);

    output.add("Saida realizada com sucesso.\n");
}

case "L" -> {
    out.writeObject("L");
    try {
        ArrayList<String> produtoList = (ArrayList<String>) in.readObject();
        ArrayList<Integer> produtoQuantidade = (ArrayList<Integer>) in.readObject();
        if (frame == null || !frame.isVisible()) {
            frame = new JFrame("Retorno do Servidor");
            frame.setSize(400, 600);
            textArea = new JTextArea(20, 50);
            textArea.setEditable(false);
            frame.add(new JScrollPane(textArea));
            frame.pack();
            frame.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE);
            frame.setVisible(true);
            frame.setVisible(true);
            SwingUtilities.invokeLater(() -> {
                output.add("Lista de Produtos:\n");
                for (int i = 0; i < produtoList.size(); i++) {
                    output.add(produtoList.get(i) + " " + produtoQuantidade.get(i) + "\n");
                }
            });
        }
    }
}

```

```

        frame.setVisible(true);
        SwingUtilities.invokeLater(() -> {
            output.add("Lista de Produtos:\n");
            for (int i = 0; i < produtoList.size(); i++) {
                output.add(produtoList.get(i) + " " + produtoQuantidade.get(i) + "\n");
            }
            for (String line : output) {
                textArea.append(line);
            }
            textArea.setCaretPosition(textArea.getDocument().getLength());
        });
    } else {
        frame.setVisible(false);
    }

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

case "F" -> {
    out.writeObject("F");
    System.out.println("===== Programa finalizado =====");
}

default -> System.out.println("Opcao invalida. Escolha novamente.");
}

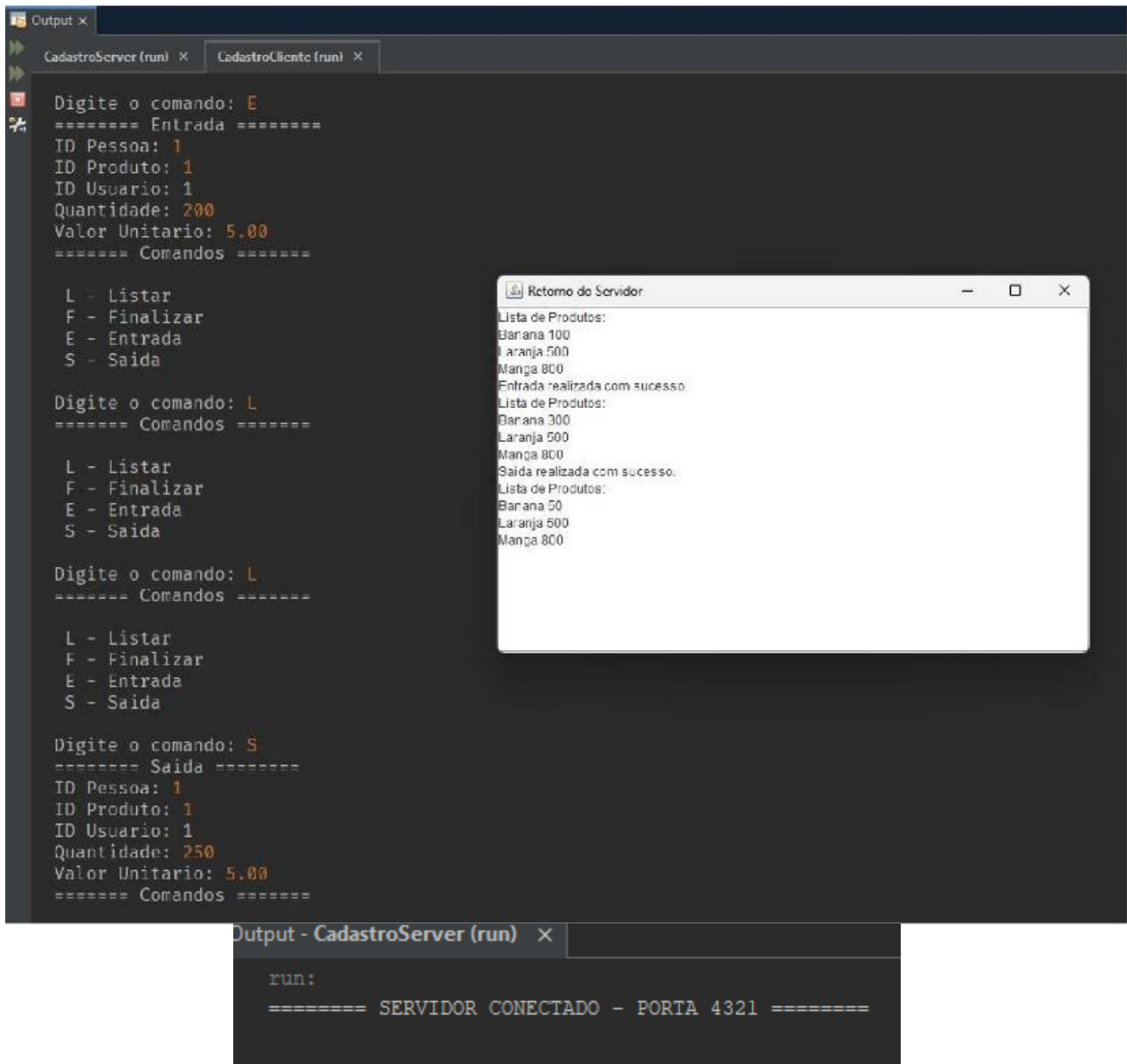
} while (!"f".equalsIgnoreCase(comando));

} else {
    System.out.println("Usuario ou senha nao conferem!");
}

} catch (HeadlessException | IOException | ClassNotFoundException e) {
    if (!(e instanceof java.io.EOFException)) {
        System.out.println("===== Thread Finalizada =====");
    }
}
}
}
}

```

**16 COM O PROJETO CADASTROSERVER EM EXECUÇÃO, INICIAR O SISTEMA DO CLIENTE, E TESTAR TODAS AS FUNCIONALIDADES OFERECIDAS.**



```
Output x
CadastroServer (run) x CadastroCliente (run) x

Digite o comando: E
===== Entrada =====
ID Pessoa: 1
ID Produto: 1
ID Usuario: 1
Quantidade: 200
Valor Unitario: 5.00
===== Comandos =====

L - Listar
F - Finalizar
E - Entrada
S - Saida

Digite o comando: L
===== Comandos =====

L - Listar
F - Finalizar
E - Entrada
S - Saida

Digite o comando: L
===== Comandos =====

L - Listar
F - Finalizar
E - Entrada
S - Saida

Digite o comando: S
===== Saida =====
ID Pessoa: 1
ID Produto: 1
ID Usuario: 1
Quantidade: 250
Valor Unitario: 5.00
===== Comandos =====

Retorno do Servidor:
Lista de Produtos:
Banana 100
Laranja 500
Manga 800
Entrada realizada com sucesso.
Lista de Produtos:
Banana 300
Laranja 500
Manga 800
Saida realizada com sucesso.
Lista de Produtos:
Banana 50
Laranja 500
Manga 800

Output - CadastroServer (run) x
run:
===== SERVIDOR CONECTADO - PORTA 4321 =====
```

## 17 ANÁLISE E CONCLUSÃO:

### 17.1 COMO AS THREADS PODEM SER UTILIZADAS PARA O TRATAMENTO ASSÍNCRONO DAS RESPOSTAS ENVIADAS PELO SERVIDOR?

Threads podem ser usadas para tratar respostas do servidor de forma assíncrona, permitindo que o cliente continue executando outras tarefas enquanto aguarda a resposta. A solicitação é enviada e uma thread separada monitora e processa a resposta assim que chega, sem bloquear a execução principal. Isso mantém a interface responsiva e permite tratamento de erros, além de usar estruturas como callbacks ou Futures para notificar a conclusão da resposta.

#### ***Threads para processamento assíncrono:***

Permitem a execução de outras tarefas enquanto a resposta do servidor é processada.

#### ***Thread separada:***

Monitora e trata a resposta assim que chega, sem bloquear a execução principal.

#### ***Interface responsiva:***

A aplicação continua funcionando sem travar.

#### ***Tratamento de erros:***

A resposta é manipulada separadamente, facilitando a detecção e gestão de erros.

#### ***Callbacks ou Futures:***

Usados para notificar quando a resposta foi processada.

## 17.2 PARA QUE SERVE O MÉTODO INVOKELATER, DA CLASSE SWINGUTILITIES?

O método `invokeLater`, da classe `SwingUtilities`, é utilizado para garantir que um código que atualiza a interface gráfica (GUI) seja executado na thread de despacho de eventos (Event Dispatch Thread, ou EDT) do Swing.

Essa abordagem é necessária porque as operações que modificam componentes da interface gráfica no Swing devem ser executadas na EDT, que é responsável por manipular eventos e atualizações visuais. Ao usar `invokeLater`, você agenda o código para ser executado nessa thread, evitando problemas de concorrência e garantindo uma atualização segura e correta da interface.

### ***Principais usos do `invokeLater`:***

#### ***Atualização de interface:***

Garante que modificações na GUI sejam feitas de forma segura.

#### ***Evita travamentos:***

Impede que a interface gráfica congele ao separar processamento pesado do fluxo de eventos.

#### ***Compatibilidade com múltiplas threads:***

Ajuda a manter a interface responsiva em aplicações multithread.



### 17.3 COMO OS OBJETOS SÃO ENVIADOS E RECEBIDOS PELO SOCKET JAVA?

No Java, objetos podem ser enviados e recebidos através de sockets utilizando as classes `ObjectOutputStream` e `ObjectInputStream`, que são especializadas para manipular objetos em streams de dados. Esses objetos precisam implementar a interface `Serializable` para permitir a serialização, ou seja, a conversão do objeto em uma sequência de bytes que pode ser transmitida pela rede.


#### ***Passos para enviar e receber objetos via Socket:***

##### ***Enviar objeto pelo Socket:***

No lado do cliente ou servidor, cria-se um `ObjectOutputStream` associado ao output stream do socket.

O método `writeObject()` é utilizado para enviar o objeto serializado pela rede.

java

 Copiar código

```
ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());  
out.writeObject(meuObjeto); // Envia o objeto serializado  
out.flush();
```

### ***Receber objeto pelo Socket:***

No outro lado da conexão, cria-se um `ObjectInputStream` associado ao input stream do socket.

O método `readObject()` é utilizado para ler e desserializar o objeto recebido

```
java Copiar código  
  
ObjectInputStream in = new ObjectInputStream(socket.getInputStream());  
MeuObjeto objRecebido = (MeuObjeto) in.readObject(); // Recebe o objeto desserializad
```

### ***Requisitos:***

#### ***Serialização:***

Os objetos enviados devem implementar a interface `Serializable`.

#### ***Tratamento de exceções:***

Devem ser tratadas exceções como `IOException` e `ClassNotFoundException` durante a transmissão e recepção.

Esse mecanismo permite a troca de dados complexos entre cliente e servidor em uma aplicação de rede Java, com a capacidade de transmitir objetos inteiros, como se fossem dados binários comuns.

## 17.4 COMPARE A UTILIZAÇÃO DE COMPORTAMENTO ASSÍNCRONO OU SÍNCRONO NOS CLIENTES COM SOCKET JAVA, RESSALTANDO AS CARACTERÍSTICAS RELACIONADAS AO BLOQUEIO DO PROCESSAMENTO.

A utilização de comportamento assíncrono ou síncrono nos clientes com socket Java traz diferentes características no tratamento de dados, especialmente relacionadas ao bloqueio de processamento.

### ***Comportamento Síncrono:***

#### ***Bloqueio do processamento:***

No modo síncrono, o cliente envia uma solicitação e aguarda uma resposta antes de continuar o processamento. Isso significa que o fluxo de execução fica bloqueado até que a resposta seja recebida, o que pode causar atrasos se o servidor demorar para responder.

#### ***Uso de recursos:***

O sistema pode ter maior latência, já que o processamento é sequencial e depende das respostas do servidor antes de avançar.

**Exemplo:** Utilização de `read()` e `write()` nos sockets que aguardam o término das operações de leitura/escrita.

#### ***Aplicações:***

Adequado para casos onde é aceitável ou necessário esperar pela resposta do servidor, como em transações onde a ordem e o tempo de execução são cruciais.

## **Comportamento Assíncrono:**

### **Não bloqueante:**

No modo assíncrono, o cliente envia uma solicitação e não aguarda pela resposta imediatamente. O processamento pode continuar, e a resposta é tratada posteriormente, quando estiver disponível.

Comumente implementado em Java com threads ou futuras (ex.: `CompletableFuture`), para que uma thread possa continuar trabalhando enquanto outra lida com a resposta do servidor.

### **Maior desempenho:**

As aplicações assíncronas tendem a ser mais responsivas, já que não aguardam bloqueadas por uma operação de I/O.

Ideal para operações que não dependem de tempo real ou respostas imediatas, permitindo que a aplicação continue executando outras tarefas.

### **Aplicações:**

Utilizado em sistemas que exigem alta performance e escalabilidade, como servidores web, sistemas distribuídos e serviços que lidam com múltiplos clientes simultaneamente.

## **Comparação:**

Característica	Síncrono	Assíncrono
Bloqueio	Bloqueia até receber a resposta	Não bloqueia, o processamento continua
Complexidade	Simples de implementar	Requer mais código, como uso de threads
Latência	Maior, pois espera a resposta	Menor, pois a execução continua
Escalabilidade	Limitada, pode sobrecarregar o servidor	Melhor, permite lidar com múltiplas conexões
Uso de Threads	Pode não utilizar threads adicionais	Utiliza múltiplas threads
Exemplo	<code>socket.read()</code> , <code>socket.write()</code>	Uso de <code>ExecutorService</code> , <code>invokeLater()</code>

**Conclusão:*****Síncrono:***

Mais simples e adequado para operações que precisam de respostas imediatas, mas pode gerar bloqueios e lentidão.

***Assíncrono:***

Mais eficiente para sistemas que precisam lidar com múltiplos clientes ou tarefas simultâneas, mas exige maior complexidade na implementação devido à necessidade de controle de threads.