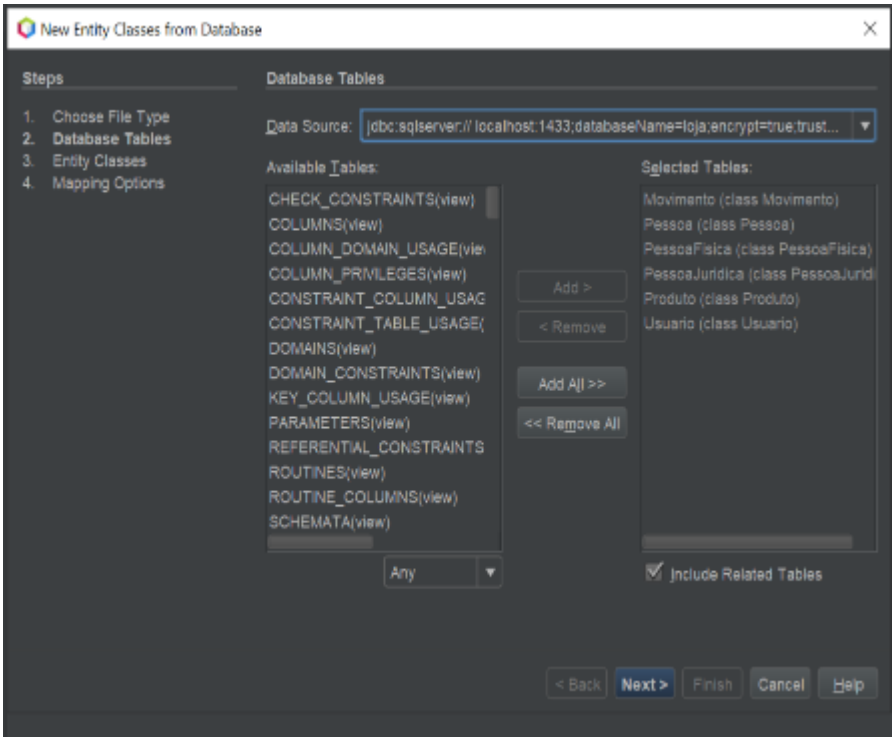
	**Campus**: Polo Cohama **Curso**: Desenvolvimento FullStack **Turma**: 9001 **Disciplina**: Por Que Não Paralelizar? **Nome**: João Victor Sá de Araújo

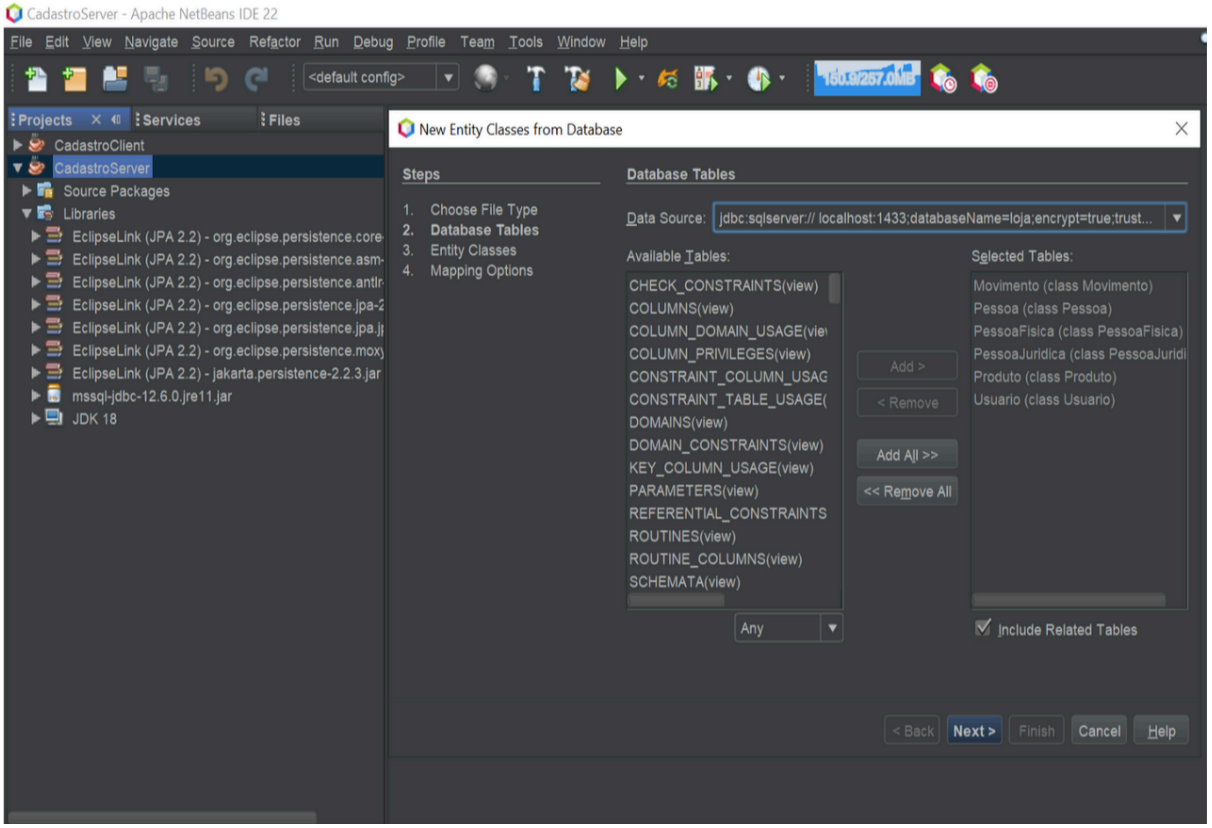
Relatório de Prática: Por Que Não Paralelizar?

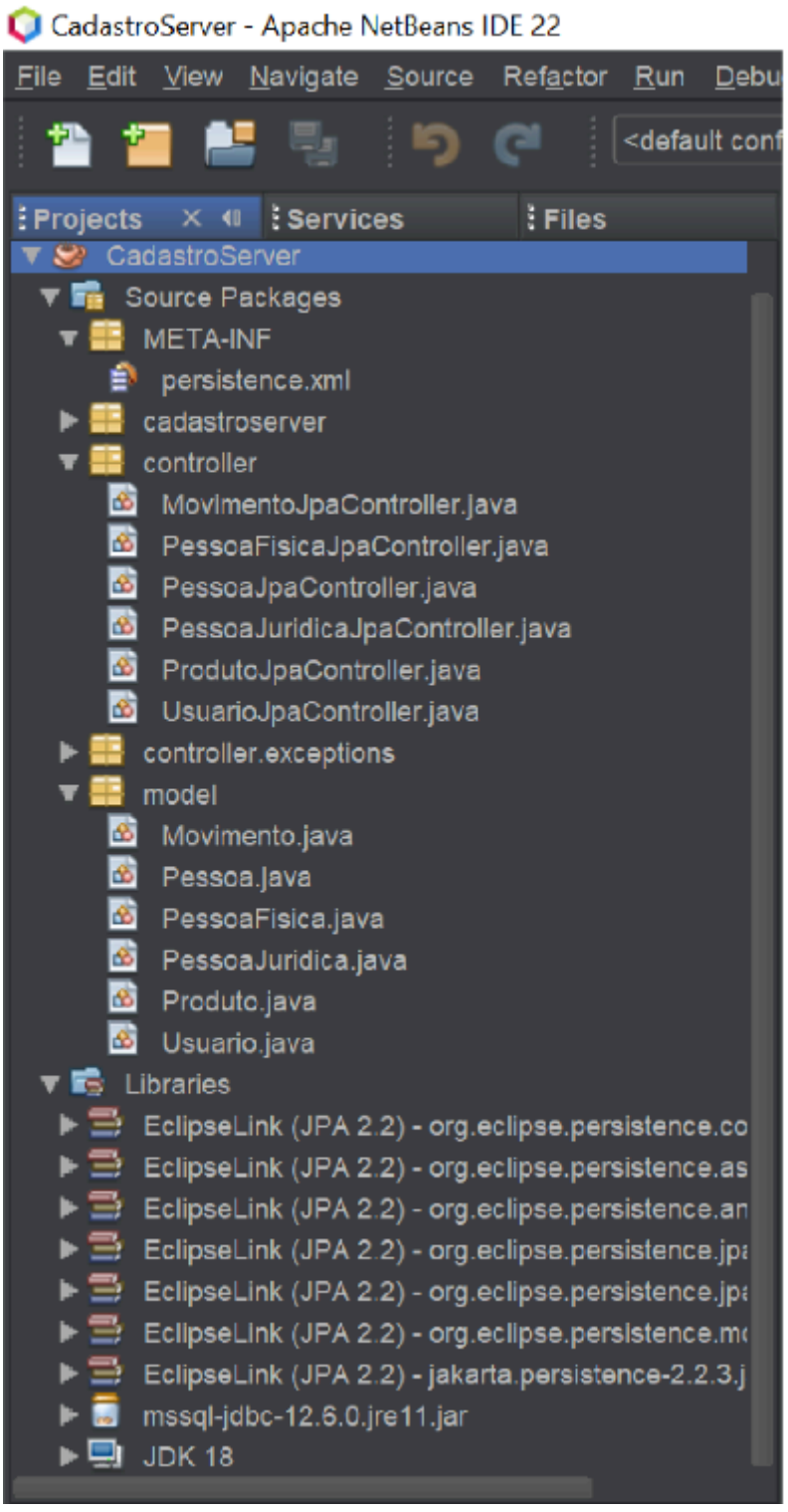
1. Objetivo da Prática

O objetivo da prática é desenvolver um sistema cliente-servidor em Java utilizando sockets, threads e persistência de dados com JPA e SQL Server. O servidor deverá autenticar usuários e responder a comandos enviados pelo cliente, permitindo a listagem e movimentação de produtos no banco de dados. A implementação incluirá o uso de `ObjectInputStream` e `ObjectOutputStream` para comunicação e a organização do código em camadas de modelo, controle e comunicação. Além disso, será desenvolvida uma segunda versão assíncrona do cliente para melhorar a interatividade, utilizando Swing para exibição de mensagens e `invokeLater` para manipulação segura da interface gráfica. A prática visa consolidar conhecimentos sobre redes, concorrência e acesso a banco de dados em aplicações distribuídas.

2. Resultados Obtidos




CadastroServer - Apache NetBeans IDE 22
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
Projects
Services
Files
CadastroClient
CadastroServer
Source Packages
Libraries
EclipseLink (JPA 2.2) - org.eclipse.persistence.core
EclipseLink (JPA 2.2) - org.eclipse.persistence.asm
EclipseLink (JPA 2.2) - org.eclipse.persistence.antlr
EclipseLink (JPA 2.2) - org.eclipse.persistence.jpa-2
EclipseLink (JPA 2.2) - org.eclipse.persistence.jpa.jp
EclipseLink (JPA 2.2) - org.eclipse.persistence.mox
EclipseLink (JPA 2.2) - jakarta.persistence-2.2.3.jar
mssql-jdbc-12.6.0.jre11.jar
JDK 18
New Entity Classes from Database
Steps
1. Choose File Type
2. Database Tables
3. Entity Classes
4. Mapping Options
Database Tables
Data Source: jdbc:sqlserver:// localhost:1433;databaseName=loja;encrypt=true;trust...
Available Tables:
CHECK_CONSTRAINTS(view)
COLUMNS(view)
COLUMN_DOMAIN_USAGE(vie
COLUMN_PRIVILEGES(view)
CONSTRAINT_COLUMN_USAG
CONSTRAINT_TABLE_USAGE(
DOMAINS(view)
DOMAIN_CONSTRAINTS(view)
KEY_COLUMN_USAGE(view)
PARAMETERS(view)
REFERENTIAL_CONSTRAINTS
ROUTINES(view)
ROUTINE_COLUMNS(view)
SCHEMATA(view)
Add >
< Remove
Add All >>
<< Remove All
Selected Tables:
Movimento (class Movimento)
Pessoa (class Pessoa)
PessoaFisica (class PessoaFisica)
PessoaJuridica (class PessoaJuridi
Produto (class Produto)
Usuario (class Usuario)
Any
Include Related Tables
< Back
Next >
Finish
Cancel
Help



CadastroThread.java:

```
package cadastroserver;
```

```
/**
 *
 * @author _joao
 */
```

```java
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.List;

import model.Produto;
import model.Usuario;
import controller.ProdutoJpaController;
import controller.UsuarioJpaController;

public class CadastroThread extends Thread {

    private ProdutoJpaController ctrlProduto;
    private UsuarioJpaController ctrlUsuario;
    private Socket s1;

    public CadastroThread(ProdutoJpaController ctrlProduto,
UsuarioJpaController ctrlUsuario, Socket s1) {
        this.ctrlProduto = ctrlProduto;
        this.ctrlUsuario = ctrlUsuario;
        this.s1 = s1;
    }

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

            String login = (String) in.readObject();
            String senha = (String) in.readObject();

            Usuario usuario = ctrlUsuario.findUsuario(login, senha);
            if (usuario == null) {
                System.out.println("Usuário inválido");
                s1.close();
                return;
            }
```

```
            LocalDateTime now = LocalDateTime.now();
            DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

            System.out.println("Usuário conectado com sucesso : " +
now.format(formatter));

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

                if (comando.equals("L")) {
                    List<Produto> produtos =
ctrlProduto.findProdutoEntities();
                    out.writeObject(produtos);
                } else {
                    System.out.println("Comando inválido");
                }
            }
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

CadastroServer.java:

```
package cadastroserver;

/**
 *
 * @author _joao
 */
import controller.MovimentoJpaController;
import controller.PessoaJpaController;
import controller.ProdutoJpaController;
import controller.UsuarioJpaController;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import javax.persistence.EntityManagerFactory;
```

```java
import javax.persistence.Persistence;

public class CadastroServer {

    public static void main(String[] args) throws IOException {

        System.out.println("Bem vindo! Por Favor Aguarde a Conexão");

        EntityManagerFactory emf =
Persistence.createEntityManagerFactory("CadastroServerPU");

        ProdutoJpaController ctrlProduto = new
ProdutoJpaController(emf);
        UsuarioJpaController ctrlUsuario = new
UsuarioJpaController(emf);
        MovimentoJpaController ctrlMov = new
MovimentoJpaController(emf);
        PessoaJpaController ctrlPessoa = new PessoaJpaController(emf);

        ServerSocket s1 = new ServerSocket(4321);

        while (true) {
            Socket s2 = s1.accept();

            CadastroThreadV2 t1 = new CadastroThreadV2(ctrlProduto,
ctrlUsuario, s2, ctrlMov, ctrlPessoa);
            t1.start();
        }
    }
}
```

3. Análise e Conclusão

Nesta prática, exploramos a implementação de um sistema cliente-servidor em Java, utilizando sockets para comunicação em rede, JPA para persistência de dados e threads para gerenciar múltiplas conexões simultaneamente. Através da classe `ServerSocket`, o servidor escutou conexões na porta definida, enquanto a classe `Socket` permitiu a comunicação do cliente com o servidor. O uso de `ObjectInputStream` e `ObjectOutputStream` garantiu a transmissão eficiente de objetos serializáveis, facilitando a troca de dados estruturados.

As portas são essenciais para direcionar as conexões entre clientes e servidores, garantindo que os serviços corretos sejam acessados. No cliente, mesmo utilizando as classes de entidades JPA do servidor, o isolamento do banco de dados foi mantido, pois as operações de persistência ficaram restritas ao servidor. A introdução da versão assíncrona do cliente permitiu uma interação mais fluida, pois a exibição das mensagens foi realizada em uma `JDialog` atualizada continuamente por uma thread separada. O método `invokeLater`, da classe `SwingUtilities`, foi necessário para garantir que atualizações da interface gráfica ocorressem de maneira segura na thread de eventos do Swing.

A prática demonstrou a importância das threads para o tratamento assíncrono das respostas do servidor, evitando bloqueios desnecessários no cliente e permitindo uma experiência mais responsiva. Comparando os dois modelos, o cliente síncrono exigia que cada operação fosse finalizada antes da próxima ser iniciada, enquanto o cliente assíncrono possibilitou a interação contínua sem congelamento da interface. Assim, concluímos que o uso de programação concorrente e comunicação em rede em Java é essencial para sistemas distribuídos escaláveis e eficientes.