

Disciplina: Programação Cliente-Servidor

Aula 9: Arquitetura MVC

Apresentação

Quando começamos a criar um sistema, devemos pensar em sua arquitetura, ou seja, na forma como os componentes serão organizados e como poderão se comunicar na resolução de problemas. Existem diversos padrões arquiteturais para a resolução de problemas recorrentes, bem como para implementar funcionalidades necessárias de sistemas operacionais e ambientes corporativos, como Broker e Pipes/Filters.

Entre esses diversos padrões, a arquitetura MVC acabou se destacando no mercado de softwares cadastrais, particularmente na Web; com a adoção de alguns padrões de desenvolvimento, como Front Control e DAO, foi possível definir diversas ferramentas para aumentar a produtividade destes ambientes. Em termos do ambiente Java para Web, uma arquitetura MVC será facilmente organizada com o uso de JPA, EJB e componentes de interface Web.

Objetivos

- Explicar o conceito de Padrão Arquitetural;
- Descrever a arquitetura MVC com Front Control;
- Aplicar JPA e EJB na criação de sistemas MVC para Web.

Padrão Arquitetural

Enquanto a programação orientada a objetos visa ao reúso de código, os **padrões de desenvolvimento** objetivam o reúso do conhecimento, trazendo modelos padronizados de soluções para problemas já conhecidos no mercado.

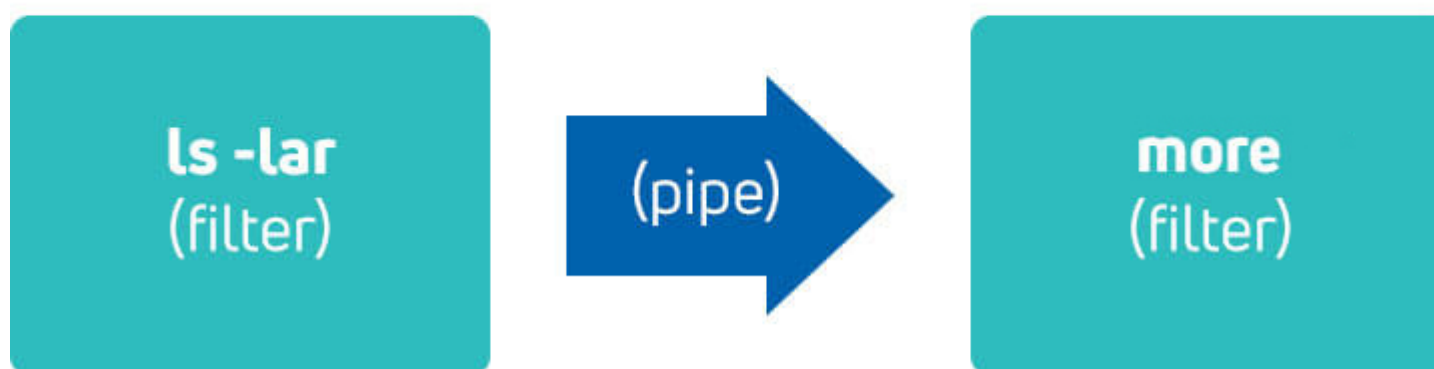
Utilizando soluções exaustivamente testadas no mercado, temos uma visão mais estratégica do modelo do sistema, facilitando a manutenção do código e evitando problemas já conhecidos devido a experiências anteriores de desenvolvimento.

Um exemplo simples é o padrão **DAO**, referente à concentração das operações de acesso a banco de dados, que evita a multiplicação de comandos SQL ao longo de todo o código; ou o padrão **Abstract Facade**, que define um modelo abstrato de fábrica e componente, permitindo a especialização deste modelo para a aplicação de regras de negócio específicas, como ocorre na criação de EJBs.

Embora os padrões já solucionem boa parte dos problemas internos do desenvolvimento de um sistema, devemos observar também que um software pode ter uma estratégia arquitetural que satisfaça a determinados domínios, e daí surge a ideia por trás dos padrões arquiteturais.

Tomando como exemplo os sistemas com processamento em lote de informações, podemos utilizar uma arquitetura do tipo **Pipes/Filters**, na qual um programa ou componente fornece uma saída que servirá de entrada para outro comando ou componente em uma sequência de processamento, como ocorre em diversos comandos do UNIX.

ls -lar | more



Saiba mais

Uma arquitetura muito comum no mercado corporativo é a de **Broker**, utilizada para objetos distribuídos, como CORBA e EJB, e que define a presença de stubs e skeletons, descritores de serviço, protocolo de comunicação, entre outros elementos, para viabilizar a distribuição do processamento.

O uso de um padrão arquitetural sugere a utilização de diversos padrões de desenvolvimento associados a ele, como o **Proxy** e **Fly Weight**, utilizados respectivamente na comunicação e gestão de pool de objetos dentro de uma arquitetura Broker.

Toda esta padronização traz enormes ganhos, pois surgem no mercado diversos produtos que generalizam a parte comum destas soluções e nos permitem especializar para aplicação de nossas próprias regras de negócio, como no caso dos frameworks de persistência.

Esta não é uma ideia nova, já vem sendo utilizada há muito tempo na engenharia com o surgimento do conceito de **COTS** (*Commercial Off-The-Shelf*), tratando de componentes comerciais competitivos e fáceis de integrar, mesmo pertencendo a diferentes fornecedores, o que aumenta a produtividade e diminui a possibilidade de ocorrência de falhas em um sistema qualquer.

Embora os primeiros COTS tratassem de elementos físicos, ou seja, hardware, hoje em dia temos muitos exemplos na área de software, como Hibernate, Spring, Prime Faces e Rich Faces, apenas considerando parte do universo Java.

O padrão arquitetural **Event-Driven**, muito importante no mundo corporativo, é baseado em eventos, o que quer dizer que são sequenciados diversos pedidos para serem atendidos de forma assíncrona, e o MOM é um típico exemplo deste tipo de arquitetura, pois as solicitações são enviadas para filas de mensagens com a finalidade de serem processadas posteriormente, sem bloquear o cliente.

Existem diversos outros padrões arquiteturais classificados conforme a tabela seguinte. Esta classificação foi definida por Frank Buschmann, e não é a única existente, mas talvez seja a mais abrangente:

Modelo	Estilo Arquitetural
Sistemas Distribuídos	Broker
Mud to Structure	Camadas
	Pipes/Filters
	Blackboard
Sistemas Interativos	MVC
	PAC
Sistemas Adaptáveis	Microkernel
	Reflexiva

Arquitetura MVC

Entre os diversos padrões arquiteturais existentes, o Model-View-Control, ou **MVC**, acabou dominando o mercado de desenvolvimento no que se refere às aplicações cadastrais. Ele promove a divisão do sistema em três camadas: persistência de dados (**Model**), interação com o usuário (**View**) e processos de negócios (**Control**).

Na primeira versão do MVC, a camada **Model** era constituída de entidades, as quais gravavam seus próprios estados no banco, segundo o padrão de desenvolvimento **Active Record**. Este modelo, inclusive, foi adotado pelos **Entity Beans** do J2EE, e acabaram se mostrando muito ineficientes, razão pela qual foram substituídos pelo JPA.

Ainda neste primeiro modelo, os dados eram transitados segundo o padrão de desenvolvimento **Value Object** (VO), que consistia basicamente dos dados das entidades, mas sem os métodos de persistência, de forma a impedir a manipulação de dados do banco pela View.

Já na segunda versão do MVC, a camada **Model** sofreu mudanças, e as entidades passaram a deter apenas os atributos referentes aos campos da tabela, como era feito anteriormente com o uso de VO, enquanto a persistência foi delegada para classes que seguem o padrão DAO.

Comentário

Com isso, as entidades passaram a ser transitadas entre as camadas, e agora apenas o DAO fica impedido de ser acessado pela camada View, o que impossibilita a manipulação direta dos dados a partir da interface visual.

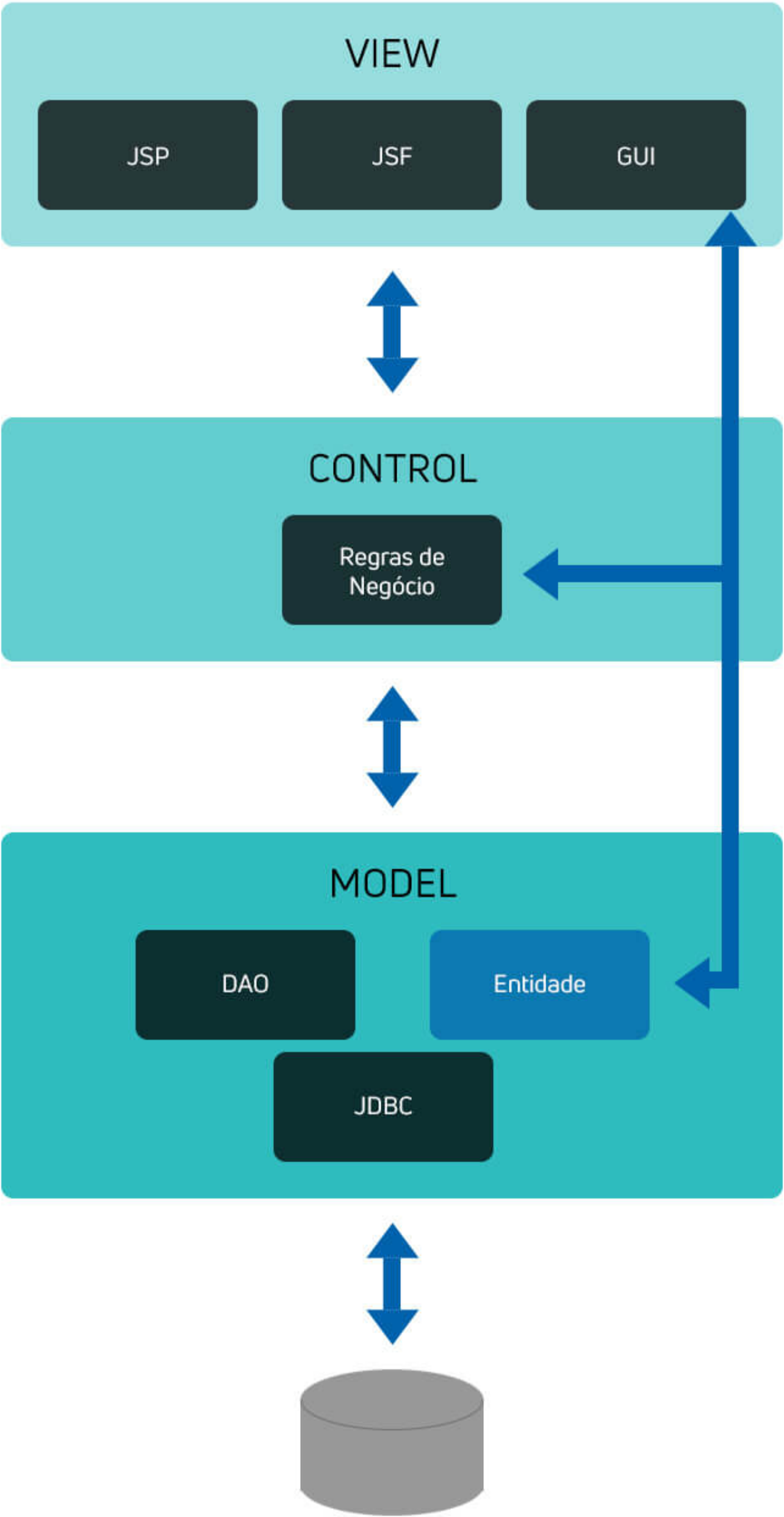
Como as anotações não são serializáveis, ao enviarmos uma entidade JPA de uma camada para outra, apenas os dados serão transitados, ficando as anotações com a funcionalidade do DAO, o que permite classificar o padrão arquitetural atual como da segunda versão.

Acima da camada **Model** encontraremos a camada **Control**, onde serão implementadas as diversas regras de negócio de nosso sistema de forma **completamente independente** de qualquer interface visual que venha a ser criada.

A camada **Control** funcionará como intermediária entre as camadas **View** e **Model** e deverá ser criada de forma que possa ser utilizada por qualquer tecnologia para a criação de interfaces sem a necessidade de mudanças no código, o que significa dizer que a interface se adequa ao controle, mas nunca o contrário.

Finalmente, temos a camada **View**, onde é criada a interface com o usuário, ou com outros sistemas, o que costuma ocorrer em integrações de ambientes corporativos.

Nesta última camada podemos utilizar as mais diversas tecnologias para a criação de interfaces, desde o uso direto de Servlets e JSPs até a adoção de frameworks como o Prime Faces. Podemos observar, na figura seguinte, uma concepção simples do MVC:

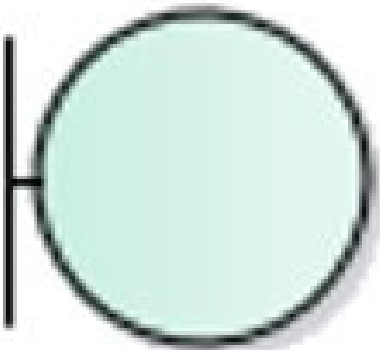
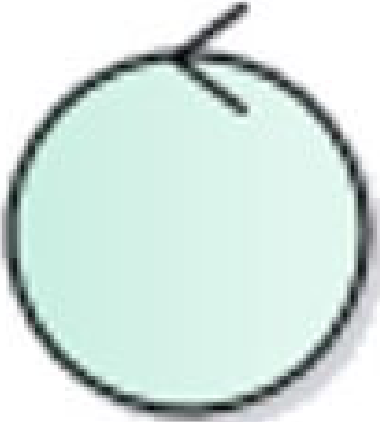
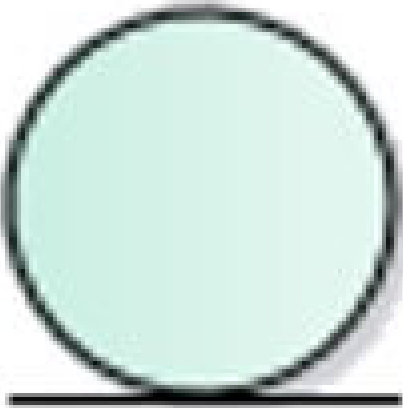


Podemos perceber que:

1. 1
- Na figura temos a camada Model com as classes de entidade e DAO, além do uso de JDBC para acesso à base de dados, seja de forma direta, ou via JPA;
2. 2
- Logo acima temos a camada de controle com as regras de negócio. Nesta camada podemos utilizar os EJBs, segundo o padrão Session Facade, para implementar as regras de negócio e controlar o acesso à camada Model;
3. 3
- Finalmente, na última camada, vemos algumas tecnologias que podem ser utilizadas para a criação de interfaces dentro do universo Java, como Servlet, JSP, JSF e GUI do tipo awt ou swing.

As classes mais relevantes do padrão arquitetural MVC são melhor representadas através de artefatos específicos, com uma simbologia já popularizada no mercado para elementos de interface, controle e modelo.

Podemos observar a simbologia utilizada na tabela seguinte:

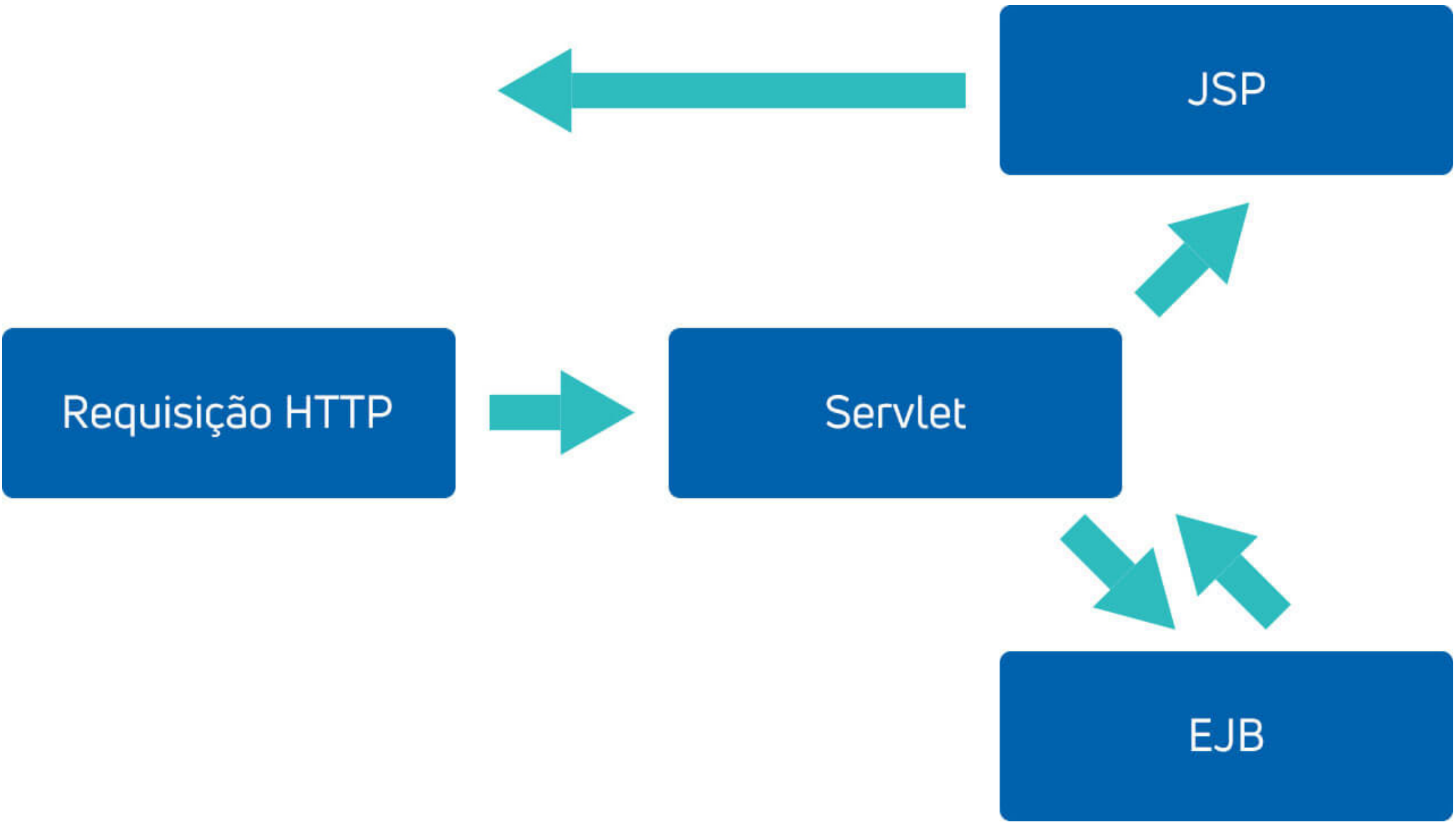
Símbolo/Camada	Características
<div></div> <div>View</div>	<ul style="list-style-type: none">- Indica uma classe limítrofe (boundary);- Exibe as informações do modelo fornecidas a partir do controle;- Envia as solicitações ao controle.
<div></div> <div>Control</div>	<ul style="list-style-type: none">- Utilizado para indicar uma classe de controle;- Define o comportamento do sistema;- Mapeia as opções disponíveis para consultas e alterações.
<div></div> <div>Model</div>	<ul style="list-style-type: none">- Utilizado para caracterizar uma entidade;- Encapsula o estado do sistema;- Associado a um DAO ou anotações de mapeamento.

Padrão Front Control

O principal problema no controle de interfaces de usuário é a disseminação de código com o objetivo de controlar entradas e saídas ao longo do sistema.

O padrão **Front Control** foi criado com o objetivo de concentrar as chamadas efetuadas pela interface e direcioná-las para os controladores corretos, além de direcionar a saída para a visualização correta ao término do processo. Embora o nome possa nos enganar, este padrão não pertence à camada Control, mas sim à camada **View**, pois não tem nenhuma relação com as regras de negócios, tendo como função primordial o simples controle de navegação do sistema.

Um exemplo de implementação deste padrão pode ser observado na figura seguinte:



Saiba mais

Dentro da arquitetura Java Web, e seguindo o padrão MVC, a implementação de um Front Control é feita através de um **Servlet**, o qual deverá receber as chamadas HTTP e, de acordo com os parâmetros fornecidos, efetuar as chamadas corretas aos elementos da camada Control, normalmente componentes EJB, recebendo em seguida as respostas destes processamentos e direcionando o resultado obtido para algum elemento, como um JSP, na camada View, para que o mesmo construa a resposta HTML ou XML.

Modelagem para Web

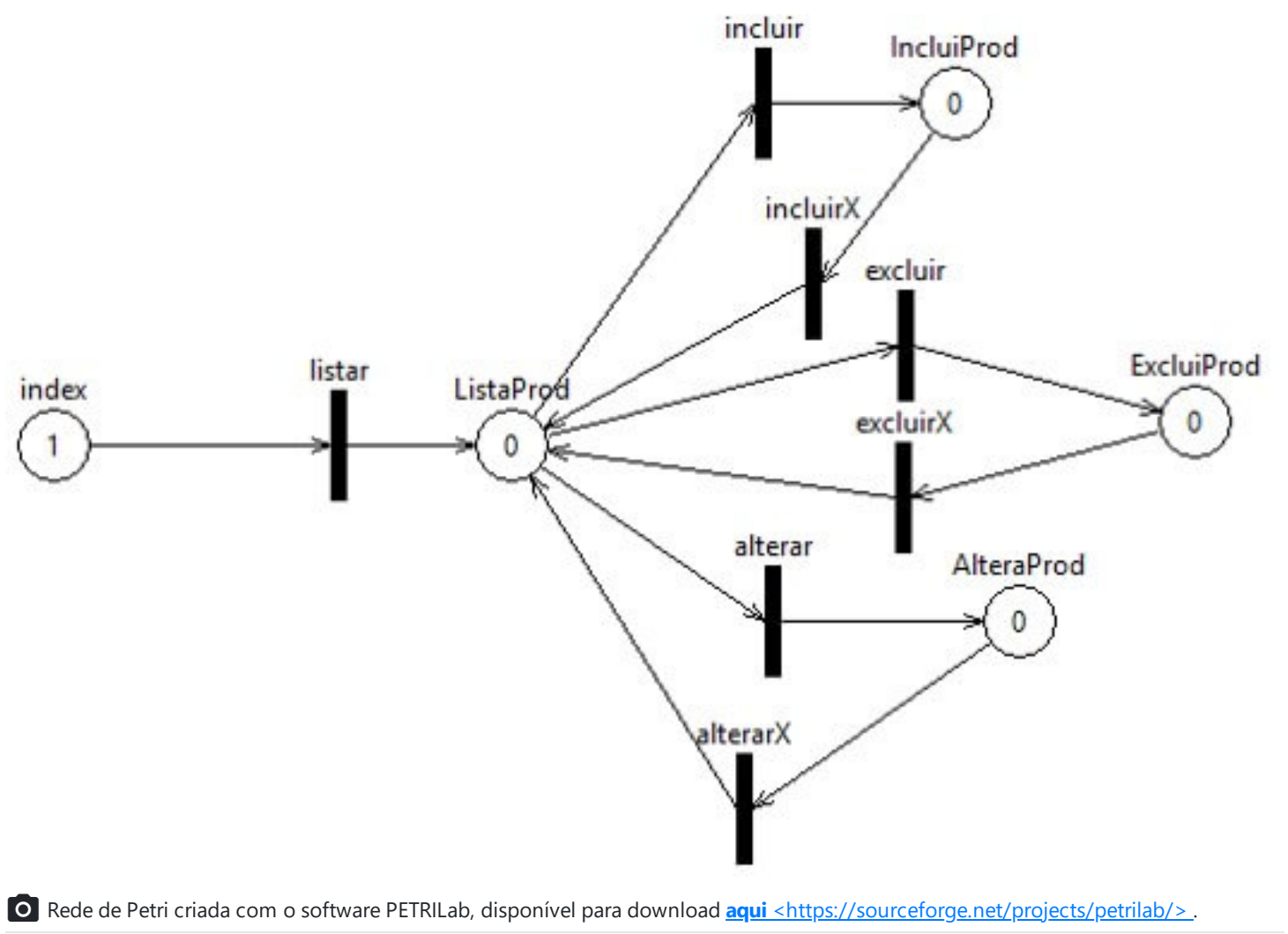
Uma metodologia de modelagem muito interessante para os sistemas Web é a adoção de [Redes de Petri¹](#). Esta é uma técnica que surgiu na área de engenharia para a modelagem de processos paralelos em máquinas físicas, sendo adotada posteriormente na área de programação.

Comentário

Podemos considerar este tipo de estrutura como um grafo, onde existem dois tipos de nós: os estados (ou lugares) e as transições (ou eventos). A mudança de estado sempre ocorrerá devido à ocorrência de uma transição.

Como estamos criando a interface com o usuário através de páginas JSP, podemos considerar estas páginas como os possíveis estados que nosso sistema pode assumir.

As transições sempre serão efetuadas através de chamadas HTTP, ocorrendo a mudança da página JSP visualizada, ou seja, do estado do sistema, através da resposta HTTP. Podemos observar um exemplo de Rede de Petri para um sistema cadastral simples, focando a tecnologia Java Web, na figura seguinte:



Rede de Petri criada com o software PETRILab, disponível para download [aqui <https://sourceforge.net/projects/petrlab/>](https://sourceforge.net/projects/petrlab/).

Neste exemplo podemos observar, em cada transição, um parâmetro obrigatório que recebe o nome de **ação**, assumindo valores como “listar”, “alterar” e “incluir”; e este parâmetro definirá qual o processamento a ser efetuado quando a chamada HTTP for recebida por um Servlet no padrão Front Control.

Toda e qualquer transição deverá passar por este Servlet, de forma a recuperar os parâmetros fornecidos na requisição, efetuar as chamadas corretas aos componentes EJBs, receber os dados processados e encaminhá-los para a página JSP correta.

Ainda analisando o diagrama, podemos observar que os estados representam as páginas do sistema, como “index.html” e “ListaProd.jsp”, sendo que o index começa com valor 1 e os demais estados com valor 0. Essa numeração indica que o index é o estado que começa ativo, e a cada transição o estado de origem assume valor 0 e o novo estado passa a valer 1, de forma similar à representação binária para “ligado” e “desligado”.

Todas as transições podem ser modeladas em termos de diagramas de sequência, como podemos observar para a ação “listar”:

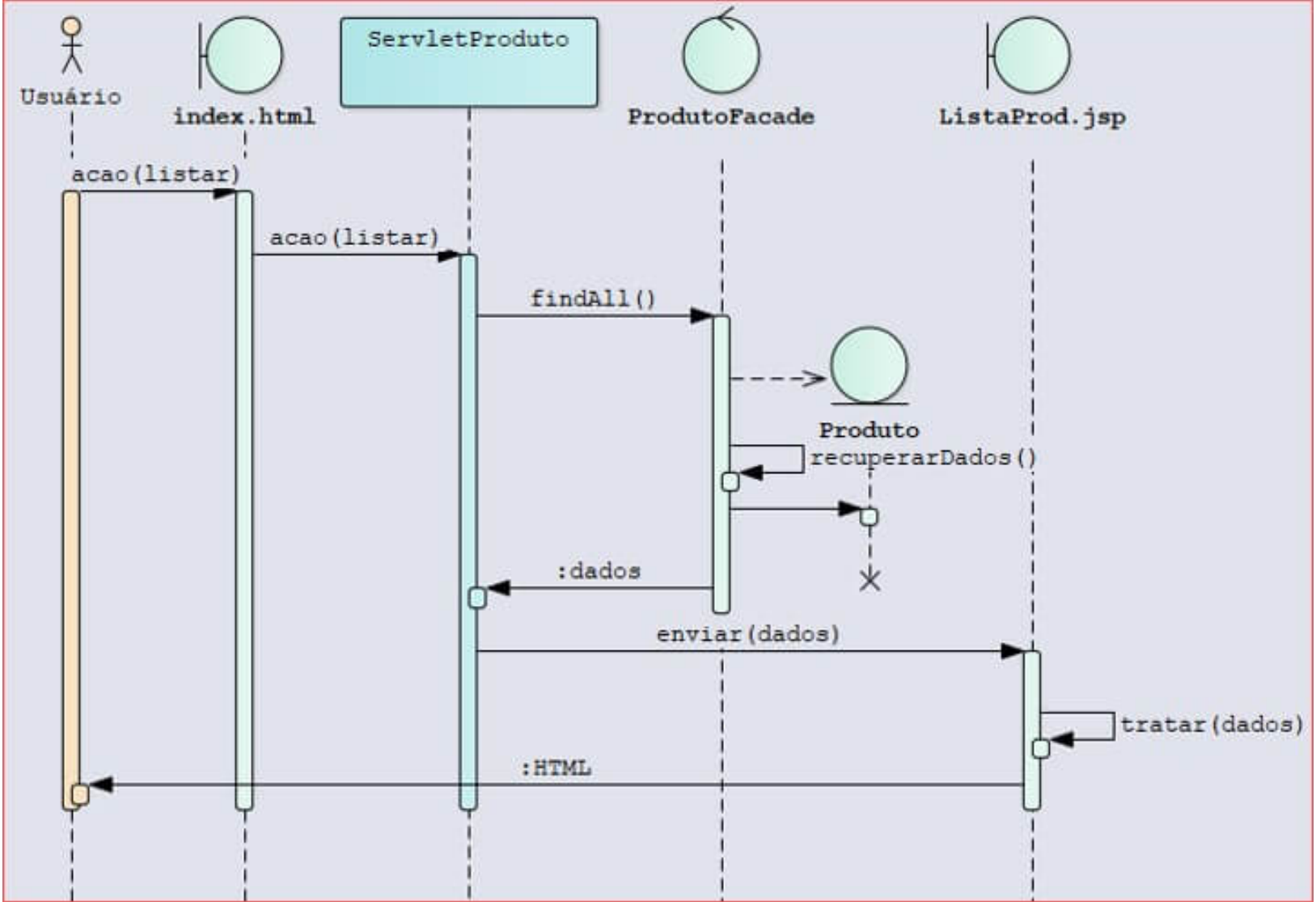


Diagrama criado com Enterprise Architect. A versão de teste pode ser obtida [aqui <https://sparxsystems.com/products/ea/trial-help.html>](https://sparxsystems.com/products/ea/trial-help.html).

Comentário

A grande vantagem do uso das Redes de Petri para modelagem Java Web é esta visão global simplificada do sistema, que traz uma relação direta com os processos, viabilizando a modelagem funcional de uma forma bastante intuitiva.

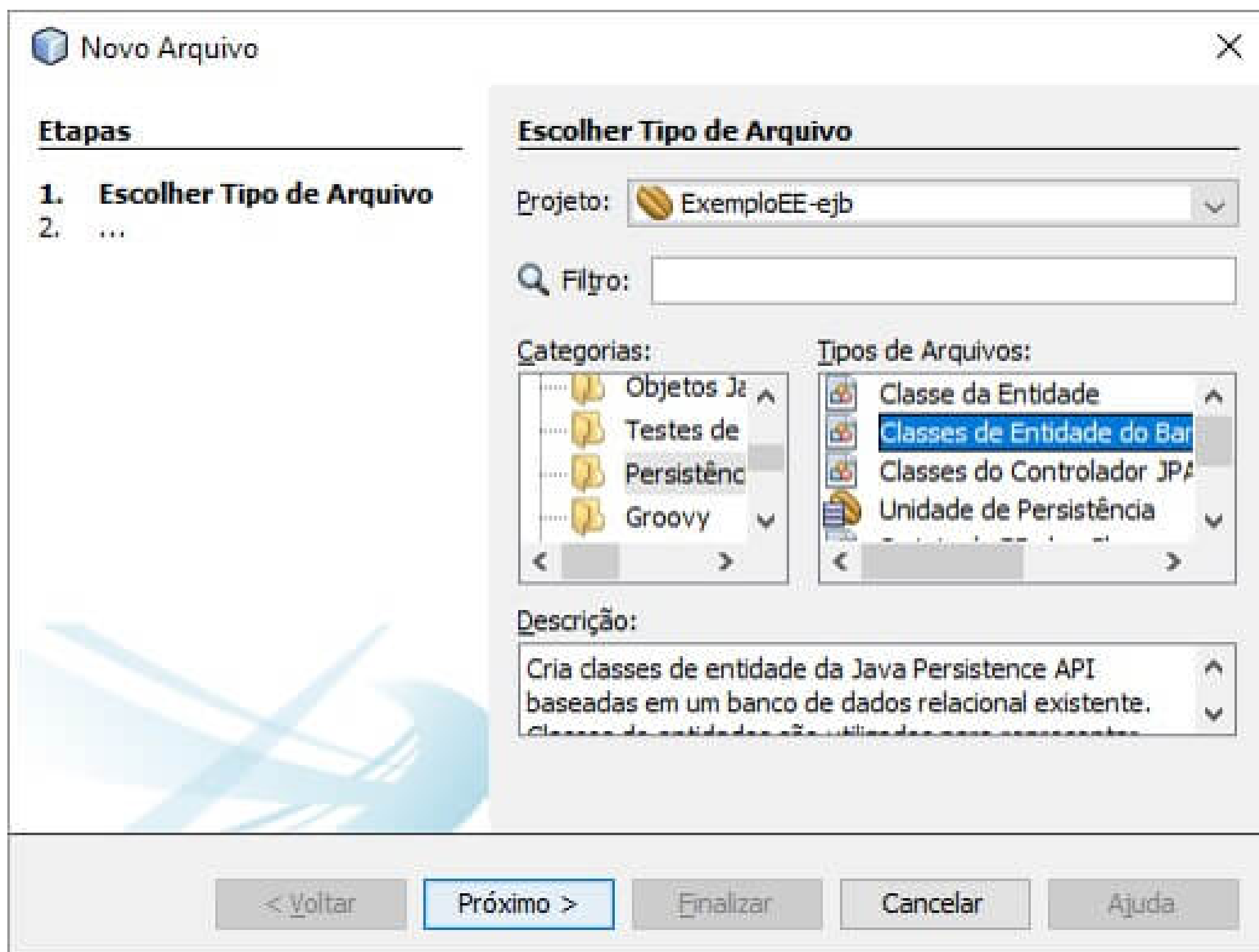
Exemplo Prático

Com os recursos do NetBeans, será muito fácil criarmos um sistema Java Web no padrão arquitetural MVC, com uso de Front Control, pois boa parte das tarefas será automatizada pelo ambiente.

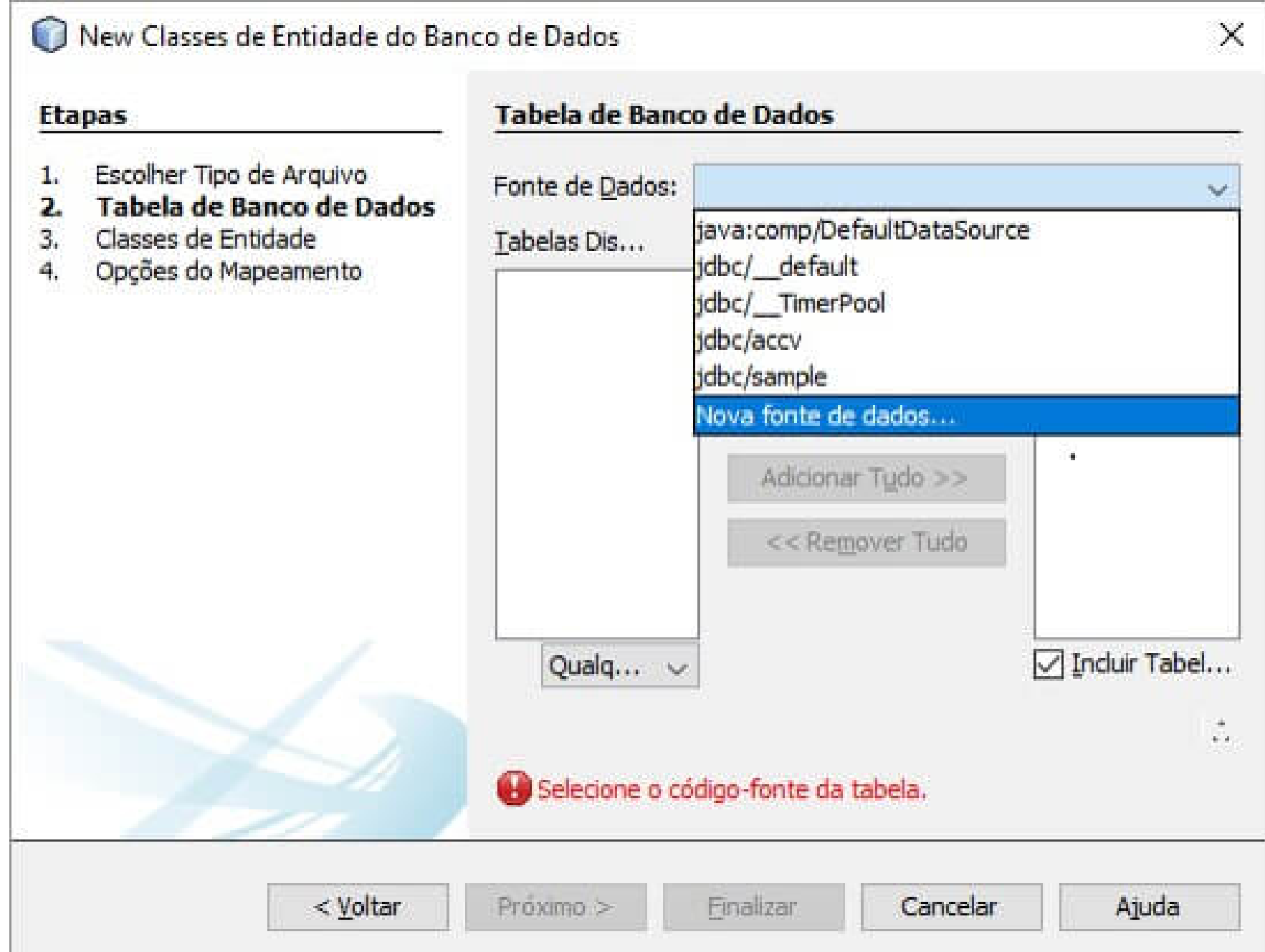
Inicialmente vamos criar um aplicativo corporativo, o qual chamaremos de **ExemploEE**, lembrando que devemos escolher o tipo de projeto como **Java EE..Aplicação Enterprise**.

Após a criação do aplicativo corporativo, vamos iniciar a implementação da camada **Model**, selecionando o projeto secundário **ExemploEE-ejb** e adicionando a ele as entidades de forma automatizada.

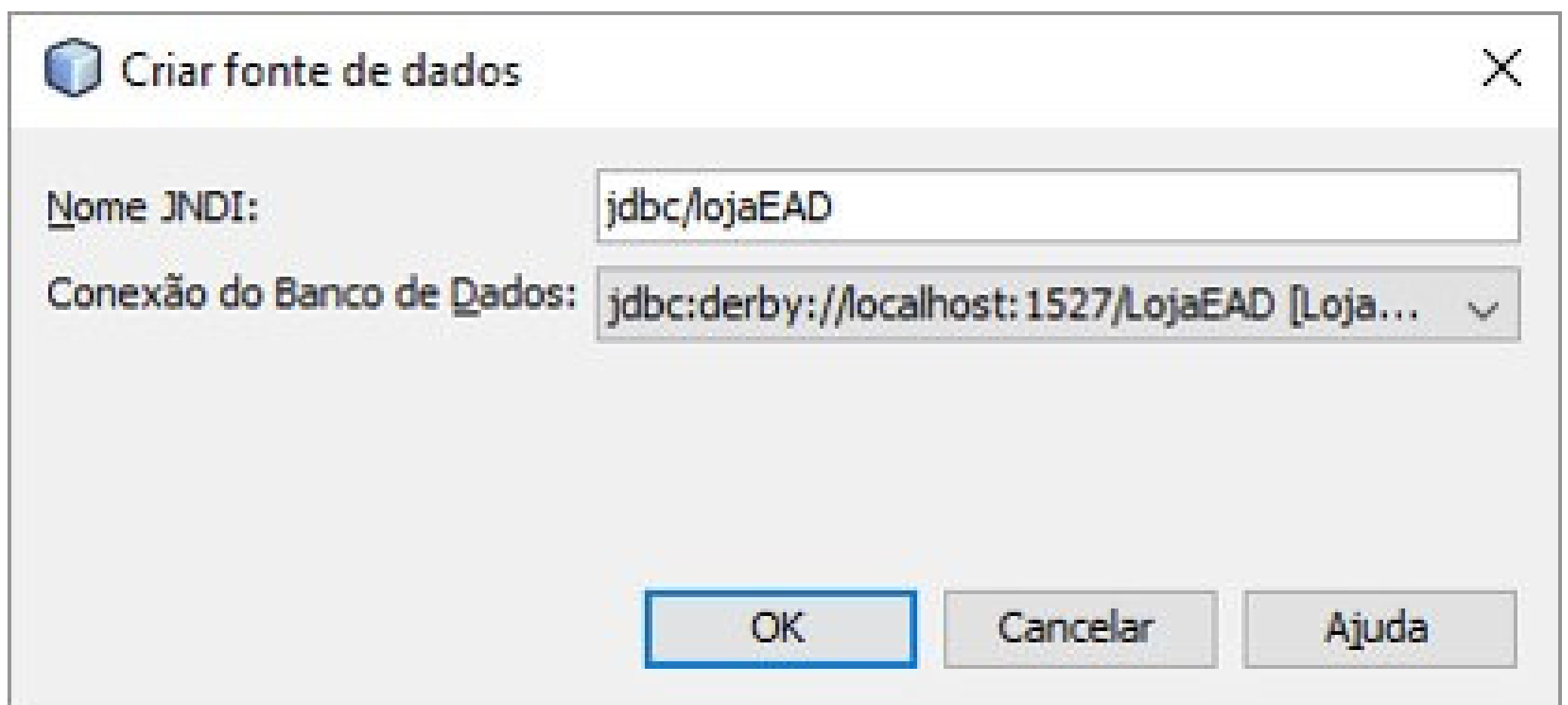
Para adicionar as entidades, devemos selecionar o menu **Arquivo..Novo Arquivo** e executar os seguintes passos:





1. Selecionar o tipo de componente **Persistência..Classes de Entidade do Banco de Dados**, e clicar em **Próximo**.



2. Adicionar um **Nova Fonte de Dados**.



3. Configurar a nova fonte com o nome JNDI **jdbc/lojaEAD**, apontando para a conexão com nosso banco de exemplo.

 Conectar

Nome de usuário:

LojaEAD

Senha:

●●●●●●●●

☒ Lembrar senha

OK

Cancelar

Ajuda

4. Digitar a senha do banco e escolher **Lembrar Senha**.

Etapas

1. Escolher Tipo de Arquivo
2. **Tabela de Banco de Dados**
3. Classes de Entidade
4. Opções do Mapeamento

Tabela de Banco de Dados

Fonte de Dados: java:module/jdbc/lojaEAD

Tabelas Dis...

PRODUTO

Adicionar >

< Remover

Adicionar Tudo >>

<< Remover Tudo

Qualq... v

Tabelas Seleccionad

☒ Incluir Tabel...

 Selecione ao menos uma tabela.

< Voltar

Próximo >

Finalizar

Cancelar

Ajuda

5. Verificar se as tabelas do banco foram recuperadas e clicar em **Adicionar Tudo**.

Etapas

1. Escolher Tipo de Arquivo
- 2. Tabela de Banco de Dados**
3. Classes de Entidade
4. Opções do Mapeamento

Tabela de Banco de Dados

Fonte de Dados: java:module/jdbc/lojaEAD

Tabelas Dis...

Qualq... ▾

Adicionar >

< Remover

Adicionar Tudo >>

<< Remover Tudo

Tabelas Seleccionad

PRODUTO

☒ Incluir Tabel...

< Voltar

Próximo >

Finalizar

Cancelar

Ajuda

6. Verificar se as tabelas foram adicionadas em **Tabelas Seleccionadas** e clicar em **Próximo**.

Classes de Entidade

Especifique os nomes e a localização das classes de entidade.

Nomes de Classes:

Tabela de Banco d...	Nome da Classe	Tipo de Geração
PRODUTO	Produto	Novo

...

Projeto:

ExemploEE-ejb

Localização:

Pacotes de Códigos-fonte



Pacote:

model



☐ Gerar Anotações de Consulta Nomeada para Campos Persistentes

☐ Gerar Anotações JAXB

☐ Gerar Superclasses Mapeadas em vez de Entidades

☒ Criar Unidade de Persistência

7. Definir o nome do pacote com **model**, deixar marcada apenas a opção de criação da **Unidade de Persistência** e clicar em **Finalizar**.

Etapas

1. Escolher Tipo de Arquivo
2. ...

Escolher Tipo de Arquivo

Projeto:  ExemploEE-ejb

Filtro:

Categorias:

- Objetos J2
- Testes de
- Persistência
- Groovy

Tipos de Arquivos:

- Classe da Entidade
- Classes de Entidade do Banco de Dados**
- Classes do Controlador JPA
- Unidade de Persistência

Descrição:

Cria classes de entidade da Java Persistence API baseadas em um banco de dados relacional existente. Classes de entidades são utilizadas para representar

< Voltar

Próximo >

Finalizar


Cancelar

Ajuda

Etapas

1. Escolher Tipo de Arquivo
2. **Tabela de Banco de Dados**
3. Classes de Entidade
4. Opções do Mapeamento

Tabela de Banco de Dados


Fonte de Dados: 

Tabelas Dis...

java:comp/DefaultDataSource
jdbc/__default
jdbc/__TimerPool
jdbc/accv
jdbc/sample
Nova fonte de dados...

Adicionar Tudo >>

<< Remover Tudo

Qualq... 

☒ Incluir Tabel...

 Selecione o código-fonte da tabela.

< Voltar

Próximo >

Finalizar

Cancelar

Ajuda

Nome JNDI:

jdbc/lojaEAD

Conexão do Banco de Dados:

jdbc:derby://localhost:1527/LojaEAD [Loja... ▾]

OK

Cancelar

Ajuda

Nome de usuário:

LojaEAD

Senha:

●●●●●●●●

☒ Lembrar senha

OK

Cancelar

Ajuda

Etapas

1. Escolher Tipo de Arquivo
- 2. Tabela de Banco de Dados**
3. Classes de Entidade
4. Opções do Mapeamento

Tabela de Banco de Dados

Fonte de Dados: java:module/jdbc/lojaEAD

Tabelas Dis...

PRODUTO

Adicionar >

< Remover

Adicionar Tudo >>

<< Remover Tudo

Qualq... ▾

Tabelas Seleccionad

☒ Incluir Tabel... **Selecione ao menos uma tabela.**

< Voltar

Próximo >

Finalizar

Cancelar

Ajuda

Etapas

1. Escolher Tipo de Arquivo
- 2. Tabela de Banco de Dados**
3. Classes de Entidade
4. Opções do Mapeamento

Tabela de Banco de Dados

Fonte de Dados: java:module/jdbc/lojaEAD

Tabelas Dis...

Tabelas Seleccionad

PRODUTO

Adicionar >

< Remover

Adicionar Tudo >>

<< Remover Tudo

Qualq... ▾

☒ Incluir Tabel...

< Voltar

Próximo >

Finalizar

Cancelar

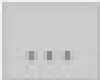
Ajuda

Classes de Entidade

Especifique os nomes e a localização das classes de entidade.

Nomes de Classes:

Tabela de Banco d...	Nome da Classe	Tipo de Geração
PRODUTO	Produto	Novo



Projeto:

ExemploEE-ejb

Localização:

Pacotes de Códigos-fonte

Pacote:

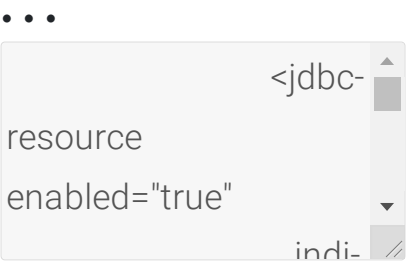
model

- ☐ Gerar Anotações de Consulta Nomeada para Campos Persistentes
- ☐ Gerar Anotações JAXB
- ☐ Gerar Superclasses Mapeadas em vez de Entidades
- ☒ Criar Unidade de Persistência

A entidade **Produto** será gerada no pacote **model** com o mesmo código que foi anteriormente considerado quando estudamos o JPA.

Nos passos dois até quatro, o que estamos fazendo é definir um novo pool de conexões registrado via JNDI no GlassFish, e a unidade de persistência utilizará este pool no acesso a dados, sendo esta a única diferença do método utilizado por nós na criação de entidades para ambiente desktop.

Isso pode ser observado nos **Arquivos de Configuração** do projeto, a começar pela definição do pool através de **glassfish-resources.xml**, presente no diretório **META-INF**. Esse arquivo será responsável pela definição do pool, cujo nome adotado será **derby_net_LojaEAD_LojaEADPool** e estará relacionado ao nome JNDI definido anteriormente na geração de entidades:



Como agora estamos em um ambiente corporativo, o uso do pool de conexões é um grande diferencial, e o arquivo **persistence.xml** irá configurar esta utilização para o JPA a partir do nome JNDI utilizado, conforme podemos observar na listagem seguinte:

• • •

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
xmlns="//xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="//www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="//xmlns.jcp.org/xml/ns/persistence //xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

<persistence-unit name="ExemploEE-ejbPU"

transaction-type="JTA">

<jta-data-source>java:module/jdbc/lojaEAD

</jta-data-source>

<exclude-unlisted-classes>false</exclude-unlisted-classes>

<properties/>

</persistence-unit>

</persistence>
```

Precisamos observar que as transações agora serão realizadas via JTA (Java Transaction API), ou seja, controladas pelo container EJB:

• • •

```
<persistence-unit name="ExemploEE-ejbPU" transaction-type="JTA">
```

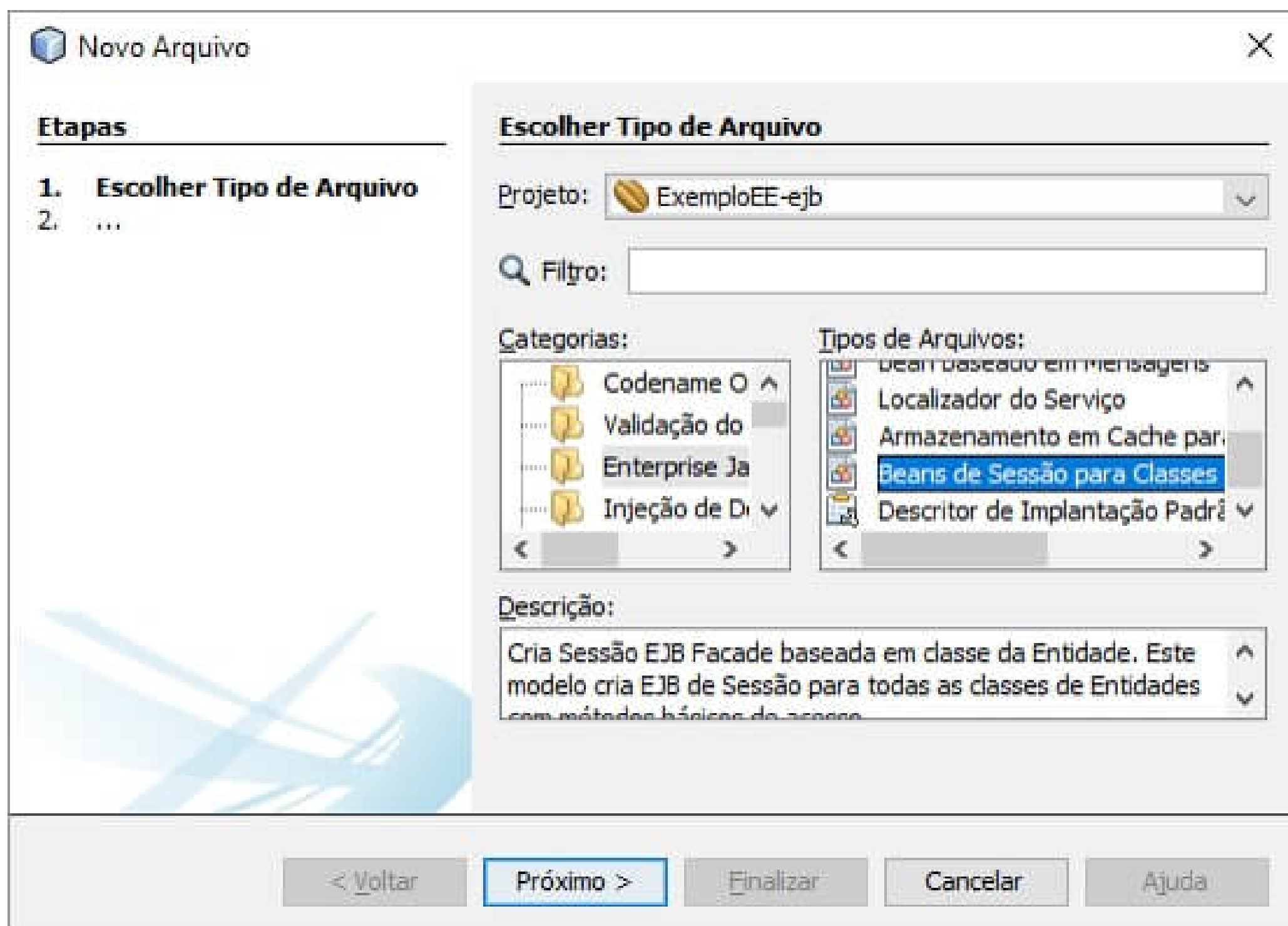
A conexão com o pool é configurada em seguida, sendo também determinada a informação que todas as classes de entidade serão utilizadas, mesmo que não estejam listadas no arquivo persistence.xml:

• • •

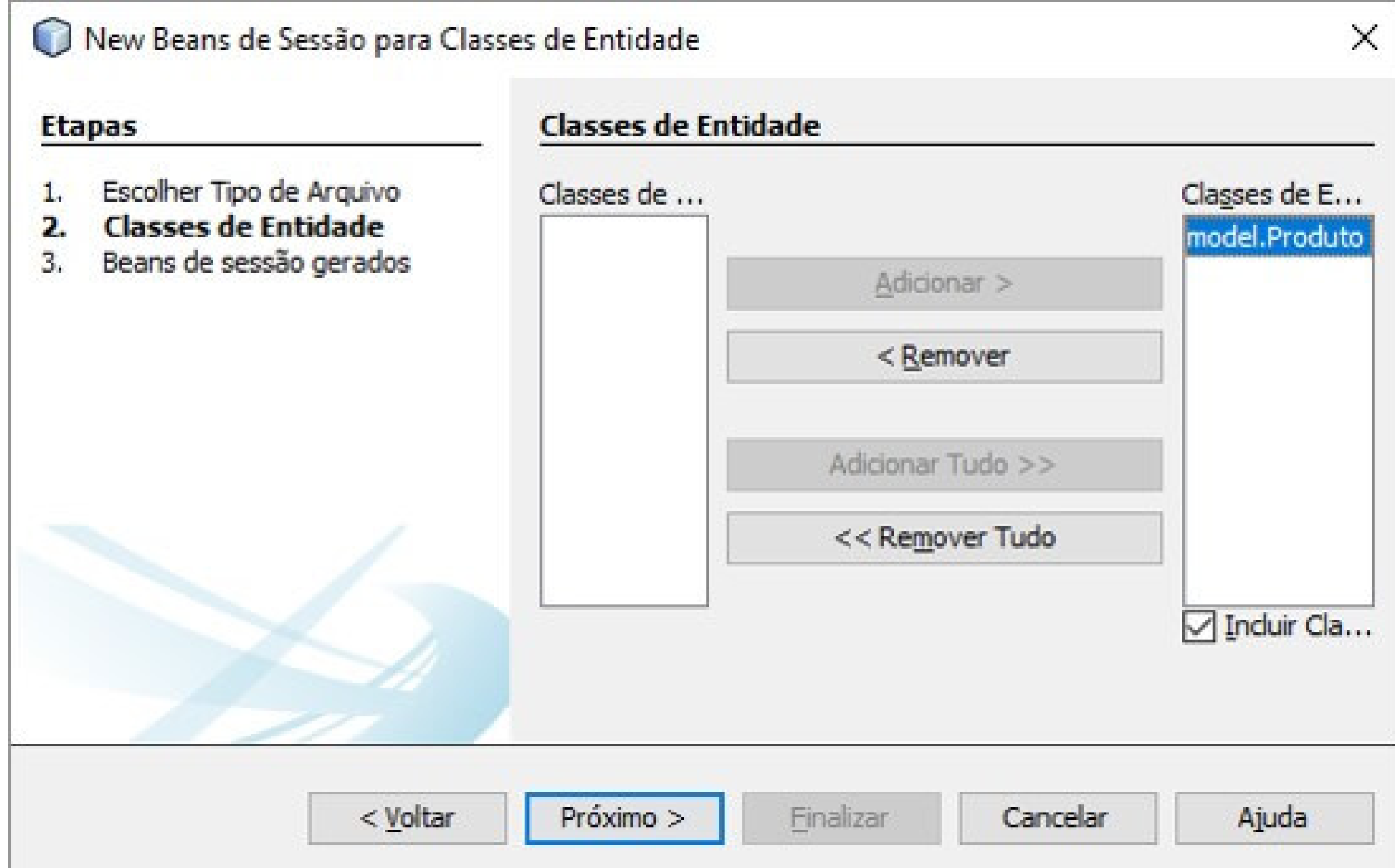
```
<jta-data-source>java:module/jdbc/lojaEAD</jta-data-source>
<exclude-unlisted-classes>false</exclude-unlisted-classes>
```

Com isso resolvemos a camada Model, e temos que implementar a camada **Control** do MVC, a qual também será definida no projeto **ExemploEE-ejb**.

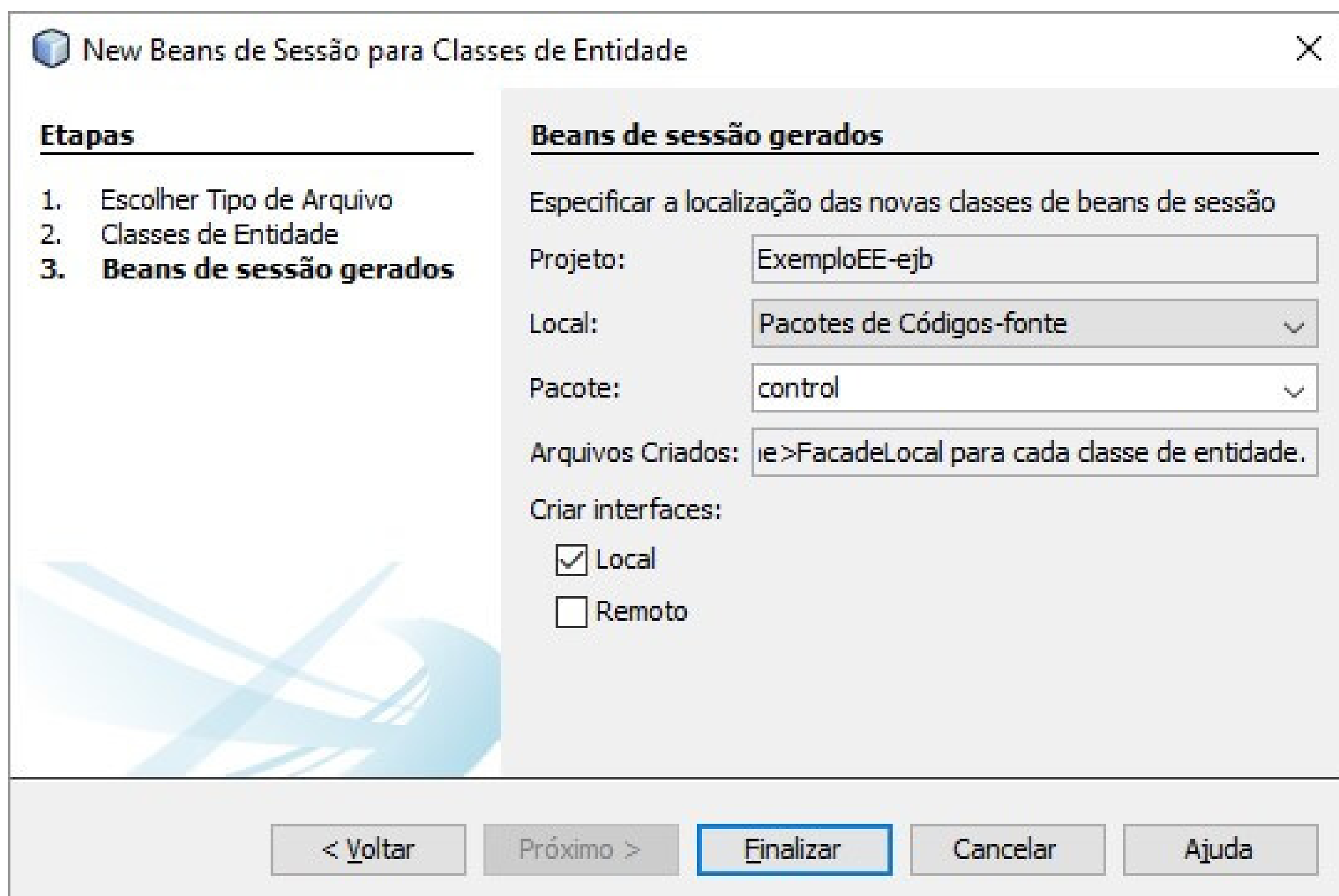
Uma forma muito simples de gerar a camada de controle neste caso será a geração dos Session Beans que funcionarão como Facade de forma automatizada, o que é feito adicionando um **Novo Arquivo** ao projeto e seguindo os seguintes passos:



1. Selecionar o tipo de componente **Enterprise JavaBeans..Beans de Sessão para Classes de Entidade** e clicar em **Próximo**.



2. Selecionar as classes de entidade que serão utilizadas e clicar em **Próximo**.



3. Definir o nome do pacote com **control**, deixando marcada apenas a opção de interface **Local** e clicar em **Finalizar**.

Etapas

1. Escolher Tipo de Arquivo
- 2. Classes de Entidade**
3. Beans de sessão gerados

Classes de Entidade

Classes de ...

Adicionar >

< Remover

Adicionar Tudo >>

<< Remover Tudo

Classes de E...

model.Produto

☒ Incluir Cla...

< Voltar

Próximo >

Finalizar

Cancelar

Ajuda

Etapas

1. Escolher Tipo de Arquivo
2. Classes de Entidade
- 3. Beans de sessão gerados**

Beans de sessão gerados

Especificar a localização das novas classes de beans de sessão

Projeto:

ExemploEE-ejb

Local:

Pacotes de Códigos-fonte

Pacote:

control

Arquivos Criados:

ie > FacadeLocal para cada classe de entidade.

Criar interfaces:

☒ Local☐ Remoto

< Voltar

Próximo >

Finalizar

Cancelar

Ajuda

<div> <div> <div>←</div> <div>→</div> </div> <div> <div> <div>http://localhost:8080/Exe</div> <div>↻</div> </div> <div> <div>Pesquisa...</div> <div>🔍</div> </div> </div> <div>localhost</div> <div> <div>×</div> <div>🔖</div> </div> </div>			
<div>Novo Produto</div>			
Código	Nome	Quantidade	Ação
1	Banana	1000	Excluir Produto
2	Morango	150	Excluir Produto
3	Laranja	400	Excluir Produto
4	Manga	350	Excluir Produto
5	Abacate	100	Excluir Produto

Ao final deste processo, serão gerados três arquivos: **ProdutoFacade**, **ProdutoFacadeLocal** e **AbstractFacade**.

Todos os processos de bancos de dados são bastante repetitivos e, por isso, o NetBeans gera uma classe chamada **AbstractFacade**, concentrando toda a parte comum da programação para banco com JPA. Podemos observar parte do código de AbstractFacade a seguir:

```
...
public abstract class AbstractFacade<T> {

    private Class<T> entityClass;

    public AbstractFacade(Class<T> entityClass) {

        this.entityClass = entityClass;

    }

    protected abstract EntityManager getEntityManager();

    public void create(T entity) {

        getEntityManager().persist(entity);

    }

    public void edit(T entity) {

        getEntityManager().merge(entity);

    }

}

...
public void edit(T entity) {

    getEntityManager().merge(entity);

}
```

Notou que não tem transação aqui? É porque a transação será gerenciada pelo container JEE com uso do JTA.

O método **getEntityManager** é abstrato, devendo ser implementado pelos seus descendentes como **ProdutoFacade**, o que podemos observar no código seguinte:

```
...
@Stateless

public class ProdutoFacade extends AbstractFacade<Produto>

    implements ProdutoFacadeLocal {

    @PersistenceContext(unitName = "ExemploEE-ejbPU")

    private EntityManager em;

    @Override

    protected EntityManager getEntityManager() {

        return em;

    }

}
```

```

}

public ProdutoFacade() {

super(Produto.class);

}

}

```

A classe **ProdutoFacade** é um componente do tipo **Stateless** Session Bean, herdando de **AbstractFacade** com uso da entidade Produto; ou seja, onde existe **T** em AbstractFacade, será utilizada a classe **Produto**. Este processo poderia ser feito para qualquer entidade, e o código comum não precisa ser replicado, facilitando muito a manutenção:

```

...
public void edit(Produto entity) {

entityManager().merge(entity);

}

```

Também temos a relação direta do **Session Bean** com o **EntityManager** através da anotação **@PersistenceContext**, necessitando apenas do nome da **unidade de persistência**:

```

...
@PersistenceContext(unitName = "ExemploEE-ejbPU")
private EntityManager em;

```

Finalmente, temos a interface local, com a assinatura dos métodos na classe de entidade alvo, ao invés do T existente nos métodos de AbstractFacade:

```

...
@Local

public interface ProdutoFacadeLocal {

void create(Produto produto);

void edit(Produto produto);

void remove(Produto produto);

Produto find(Object id);

List<Produto> findAll();

List<Produto> findRange(int[] range);

int count();

}

```

Agora já temos os métodos necessários para efetuar consultas, incluir, alterar e excluir os dados, com uso das camadas Control e Model. É interessante observarmos que não há nenhuma dependência de ambiente, pois sequer temos uma interface visual.

É neste ponto que devemos nos preocupar com a última camada do MVC, a camada **View**, que será implementada no projeto **ExemploEE-war**.

Inicialmente devemos adicionar um Servlet que terá o nome de **ServletProduto** e ficará no pacote **view**. Podemos observar o código principal do novo Servlet a seguir:

• • •

@EJB

ProdutoFacadeLocal facade;

protected void processRequest(HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

String acao = request.getParameter("acao");

if(acao==null)acao="listar";

String pagDestino = (acao.equals("incluir"))?

"incluirProduto.html":"ListaProduto.jsp";

if(acao.equals("incluirX")){

Produto p1 = new Produto(new Integer(

request.getParameter("codigo")));

p1.setNome(request.getParameter("nome"));

p1.setQuantidade(new Integer(

request.getParameter("quantidade")));

facade.create(p1);

}

if(acao.equals("excluirX")){

facade.remove(facade.find(new Integer(

request.getParameter("codigo"))));

}

request.setAttribute("listagem", facade.findAll());

request.getRequestDispatcher(pagDestino).

forward(request,response);

}

Pelo código do Servlet, temos um parâmetro **acao** que será utilizado para diferenciar o que deve ser feito a cada chamada. Por exemplo, se o valor for “**excluirX**”, será executado o método **remove** de ProdutoFacade.

• • •

if(acao.equals("excluirX")){

facade.remove(facade.find(new Integer(

request.getParameter("codigo"))));

}

Outras operações podem ser efetuadas, e ao final ocorrerá o redirecionamento para a página de visualização:

```
• • •
request.getRequestDispatcher(pagDestino).
```

```
forward(request,response);
```

Agora devemos adicionar o arquivo **ListaProduto.jsp** para iniciar nossos testes, conforme o código apresentado a seguir:

```
• • •
<%@page import="model.Produto"%>

<%@page import="java.util.List"%>

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<body>

<a href="ServletProduto?acao=incluir">Novo Produto</a>

<table border="1" width="100%">

<tr><td>Código</td><td>Nome</td>

<td>Quantidade</td><td>Ação</td>

</tr>

<%

List<Produto> listagem = (List<Produto>)

request.getAttribute("listagem");

for(Produto p: listagem){

%>

<tr><td><%=p.getCodigo()%></td><td><%=p.getNome()%></td>

<td><%=p.getQuantidade()%></td>

<td><a href=

"ServletProduto?acao=excluirX&codigo=<%=p.getCodigo()%>">

Excluir Produto</a></td></tr>

<% } %>

</table>

</body>

</html>
```

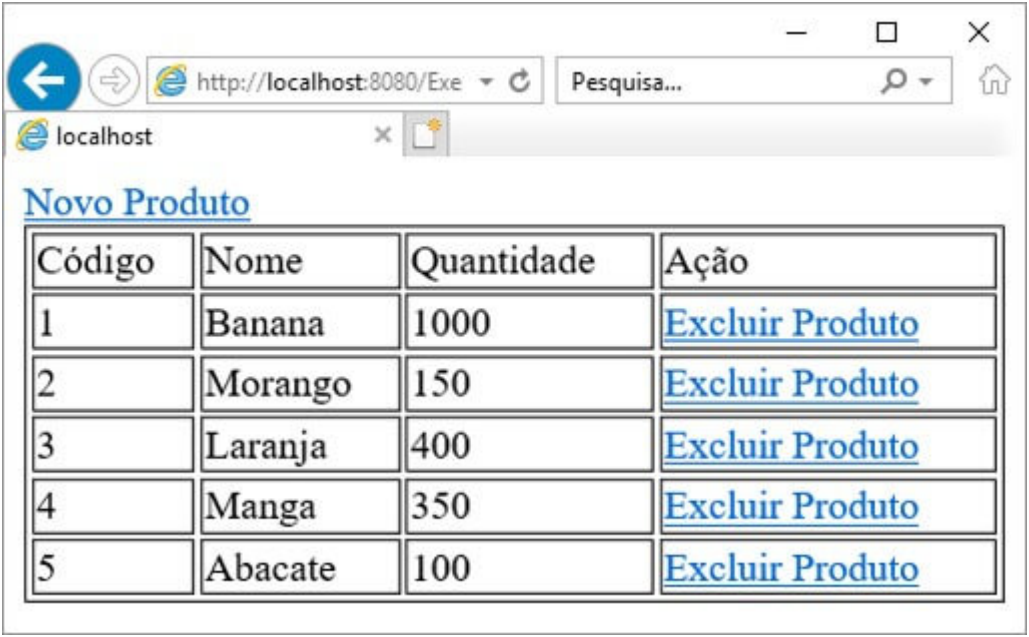
Podemos observar neste JSP a recepção dos dados enviados pelo Servlet no atributo “**listagem**” da requisição, o qual servirá de base para preenchimento da tela:

```
• • •
List<Produto> listagem = (List<Produto>)
```

```
request.getAttribute("listagem");
```

```
for(Produto p: listagem){
```

Agora nós podemos implantar o projeto principal, denominado **ExemploEE**, e exibir a tela cadastral apresentada a seguir, apenas digitando o endereço **//localhost:8080/ExemploEE-war/ServletProduto**.



Neste ponto a listagem e a exclusão de produtos estão funcionais, faltando apenas criar a página **incluirProduto.html**, como pode ser observada a seguir:

```
• • •
<html>

<body>

<form action="ServletProduto">

<input type="hidden" name="acao" value="incluirX"/>
```

```
Código:<br/>
```

```
<input type="text" name="codigo"/>
```

```
Nome:<br/>
```

```
<input type="text" name="nome"/>
```

```
Quantidade:<br/>
```

```
<input type="text" name="quantidade"/>
```

```
<input type="submit" value="Incluir"/>
```

```
</form>
```

```
</body>
```

```
</html>
```

Comentário

Agora podemos utilizar todas as funcionalidades de nosso pequeno projeto, o que contempla a listagem, exclusão e inclusão de produtos, implementado em uma arquitetura MVC, utilizando um Servlet como Front Control.

Sessões e Controle de Acesso

As sessões são de grande utilidade no ambiente Web, provendo uma forma de manutenção de estados na troca de páginas, pois ao contrário dos sistemas desktop, a cada nova página temos outro conjunto de variáveis na memória, desconsiderando-se todas aquelas existentes antes de a requisição ser efetuada.

Podemos controlar sessões de forma muito simples, com o uso da classe `HttpSession`; um exemplo típico de utilização é no [controle de login](#)².

Vamos, inicialmente, criar uma página JSP protegida, denominada “**Segura.jsp**”, como pode ser observado no código seguinte:

```
• • •
<%@page contentType="text/html" pageEncoding="UTF-8"%>

<%

if(session.getAttribute("usuario")==null)

response.sendRedirect("Login.jsp");

else {

%>

<!DOCTYPE html>

<html>

<body>

<h1>Esta é uma página protegida!</h1>

O usuário <%=session.getAttribute("usuario")%> está logado.

</body>

</html>

<% } %>
```

Enquanto os **parâmetros** da requisição assumem apenas valores do tipo texto, os **atributos** de uma sessão permitem guardar qualquer tipo de objeto, inclusive texto.


Na primeira parte do arquivo JSP temos o teste para a existência do atributo “usuario” na sessão, mas se o mesmo não existir, significa que não foi efetuado o processo de login, devendo ocorrer o redirecionamento para a página de login através de **sendRedirect**.

```
• • •
if(session.getAttribute("usuario")==null)

response.sendRedirect("Login.jsp");

else {
```

Existem dois tipos de redirecionamento no ambiente Java Web:

 Clique nos botões para ver as informações.

sendRedirect

▼

O modo **sendRedirect** envia um sinal de redirecionamento ao navegador para que efetue uma nova requisição ao servidor.

forward

▼

Através de **forward** nós efetuamos um redirecionamento interno no servidor, e as informações da requisição original são mantidas no envio de um componente para outro, sem ocorrer qualquer contato com o navegador no processo.

Ainda observando o código, podemos notar que a instrução **else** é aberta antes do início do código **HTML** e fechada apenas no final, logo após a tag de finalização **</html>**. Isto significa que todo este bloco será processado apenas se ocorre o atributo usuário na sessão, ou seja, se existe alguém logado no sistema.

Existindo um usuário logado, o bloco é executado e a página de resposta é montada, contando inclusive com a identificação do login atual:

• • •

O usuário <%=session.getAttribute("usuario")%> está logado.

Agora precisamos criar a página de login, denominada “**Login.jsp**”, e o Servlet responsável pela verificação dos dados para conexão, o qual será chamado de **ServletLogin**. Podemos observar o código da página de login a seguir:

• • •

<%@page contentType="text/html" pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<body>

<h1>Acesso ao Sistema</h1>

<form action="ServletLogin" method="post">

Login: <input type="text" name="login"/>

Senha: <input type="password" name="senha"/>

<input type="submit" value="login"/>

</form>

<%

if(request.getAttribute("erro")!=null) {

%>

<hr/>Ocorreu um erro: <%=request.getAttribute("erro")%>

<%

}

%>

</body>

</html>

Na primeira parte deste JSP temos um formulário HTML bastante simples, contendo as informações que deverão ser enviadas ao Servlet para verificação.

Na segunda parte temos a possibilidade de apresentar mensagens de erro enviadas pelo Servlet, como “Dados inválidos”. Note que este trecho será apresentado apenas se o atributo de “erro” estiver presente na chamada ao JSP.

Agora precisamos criar o **ServletLogin** e implementar o código apresentado a seguir:

• • •

```
@WebServlet(name = "ServletLogin",

urlPatterns = {"/ServletLogin"})

public class ServletLogin extends HttpServlet {

@Override

protected void doPost(HttpServletRequest request,

HttpServletResponse response)

throws ServletException, IOException {

HttpSession session = request.getSession();

if(request.getParameter("login").equals("admin")&&

request.getParameter("senha").equals("123")){

session.setAttribute("usuario", "Administrador");

response.sendRedirect("Segura.jsp");

} else {

request.setAttribute("erro","Dados inválidos.");

request.getRequestDispatcher("Login.jsp").

forward(request,response);

}

}

// Restante do código oculto.

}
```

Note que, por se tratar de um processo de login, apenas o método **doPost** poderá ser utilizado sendo feito um teste em que apenas o login “admin” e senha “123” em conjunto permitirão o acesso ao sistema, acrescentando à sessão o atributo “**usuario**”, com valor “Administrador” e redirecionando para a página segura.

Caso não sejam fornecidos os dados com os valores corretos, a página de login é acionada, passando o atributo de erro para a mesma através de um atributo de requisição.

Ao tentar acessar a página **Segura.jsp** será apresentada a tela de login e apenas com os dados corretos será exibida a página segura:



Comentário

Embora este seja apenas um exemplo simples de processo de autenticação de usuário, com valores pré-fixados, você pode alterá-lo facilmente para a utilização de uma base de dados com senhas criptografadas.

Atividade

1. Existem diversos padrões arquiteturais; os componentes EJB, como são objetos distribuídos, seguem a arquitetura:

- a) Pipes/Filters
- b) Microkernel
- c) Broker
- d) MVC
- e) PAC

2. Na arquitetura MVC para Web, considerando o ambiente Java, qual componente deve atuar como Front Control?

- a) Stateless Session Bean
- b) Servlet
- c) MDB
- d) EntityManager
- e) Stateful Session Bean

3. Considerando a forma de criação automatizada de JPA e EJB pelo NetBeans, complete o código do Servlet para remover da base todos os produtos com quantidade zero se o parâmetro “acao” for igual a “limparX”.

```
...
@WebServlet(name = "ServletProduto",

urlPatterns = {"/ServletProduto"})

public class ServletProduto extends HttpServlet {

    @EJB

    ProdutoFacadeLocal facade;

    protected void processRequest(HttpServletRequest request,

    HttpServletResponse response)

    throws ServletException, IOException {

        String acao = request.getParameter("acao");
```

Notas

Redes de Petri¹

Com o uso de uma Rede de Petri, podemos modelar os diversos estados de um sistema e as transições que levam de um estado para outro, podendo incluir parâmetros de chamada e valores de retorno.

Controle de Login²

Nas páginas JSP o controle de sessão é feito com o uso do objeto implícito **session**, da classe HttpSession.

Referências

CASSATI, J. P. **Programação servidor em sistemas web**. Rio de Janeiro: Estácio, 2016.

DEITEL, P.; DEITEL, H. **Java, como programar**. 8 ed. São Paulo: Pearson, 2010.

DEITEL, P, DEITEL, H. **Java, como programar**. 8.ed. São Paulo: Pearson, 2010.

MONSON-HAEFEL, R; BURKE, B. **Enterprise java beans 3.0**. 5.ed. São Paulo: Pearson, 2007.

SANTOS, F. **Tecnologias para internet II**. 1.ed. Rio de Janeiro: Estácio, 2017.

Próximos passos

- Serviços interoperáveis;
- Web Services do tipo SOAP;
- Web Services do tipo REST.

Explore mais

Leia:

- [10 padrões comuns de arquitetura de software em poucas palavras <https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013>](https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013)
- [Abordando a arquitetura MVC, e Design Patterns: Observer, Composite, Strategy <http://www.linhadecodigo.com.br/artigo/2367/abordando-a-arquitetura-mvc-e-design-patterns-observer-composite-strategy.aspx>](http://www.linhadecodigo.com.br/artigo/2367/abordando-a-arquitetura-mvc-e-design-patterns-observer-composite-strategy.aspx)

Assista:

- [Java Web MVC con JSP / Servlet / JPA / EJB con JavaEE 7 / GlassFish 4.1 | NetBeans 8.0 <https://www.youtube.com/watch?v=9JwXoL0FSBs>](https://www.youtube.com/watch?v=9JwXoL0FSBs)

Analise:

- [O que é o HttpSession? <https://www.studytonight.com/servlet/httpsession.php>](https://www.studytonight.com/servlet/httpsession.php)