

# Tecnologias JPA e JEE

Prof. Denis Gonçalves Cople

Prof. Kleber Aguiar

## Descrição

Utilização de tecnologias JPA (Java Persistence API) e EJB (Enterprise JavaBeans), descrição das tecnologias e sua organização em uma arquitetura MVC (Model, View e Controller), construção de um exemplo completo observando o padrão Front Controller, com utilização das tecnologias estudadas.

## Propósito

Ao final dos estudos, você estará apto a construir sistemas na arquitetura MVC, com base nas tecnologias JPA e JEE, adotando interface Java para Web. O conhecimento adquirido será de grande utilidade para a inserção do aluno no mercado corporativo, tendo como base arquiteturas e componentes robustos. Também serão observados os métodos de automatização do NetBeans, visando a obter maior produtividade.

## Preparação

Antes de iniciar o conteúdo, é necessário configurar o ambiente, com a instalação do JDK e Apache NetBeans, definindo a plataforma de desenvolvimento. Também é necessário instalar o Web Server Tomcat e o Application Server GlassFish, definindo o ambiente de execução e testes.

Sugere-se configurar a porta do servidor Tomcat como 8084, de forma a evitar conflitos com o GlassFish na porta 8080.

# Objetivos

Módulo 1

## A tecnologia JPA

Descrever as características do JPA.

Módulo 2

## Componentes Enterprise Java Beans

Empregar componentes EJB na construção de regras de negócio.

Módulo 3

## Arquitetura MVC no Java

Descrever a utilização da arquitetura MVC na plataforma Java.

Módulo 4

## Padrão Front Controller em MVC/Java Web

Empregar padrão Front Controller em sistema MVC, com interface Java Web.



# Introdução

A tecnologia JPA (Java Persistence API), para o mapeamento objeto-relacional, e o uso de componentes do tipo EJB (Enterprise Java Bean), elemento central do JEE (Java Enterprise Edition), para implementar regras de negócio, são conhecimentos essenciais para o profissional de desenvolvimento Web.

Após compreender as duas tecnologias, analisaremos os elementos estruturais da arquitetura MVC (Model, View e Controller), e criaremos um sistema cadastral, com base na arquitetura, utilizando JPA, EJB e componentes Web, adotando, ainda, o padrão Front Controller na camada de visualização para Web.



## 1 - A tecnologia JPA

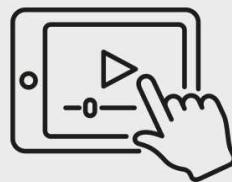
Ao final deste módulo, você será capaz de descrever as características do JPA.

# ORM: mapeamento objeto-relacional

## Mapeamento objeto-relacional

Conheça as principais características do mapeamento objeto-relacional.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Nos bancos de dados **relacionais**, temos uma estrutura baseada em tabelas, que comportam valores em registros, e que se relacionam a partir de campos identificadores ou chaves primárias. A manutenção dos relacionamentos, por sua vez, é implementada através de chaves estrangeiras e estruturas de índices.

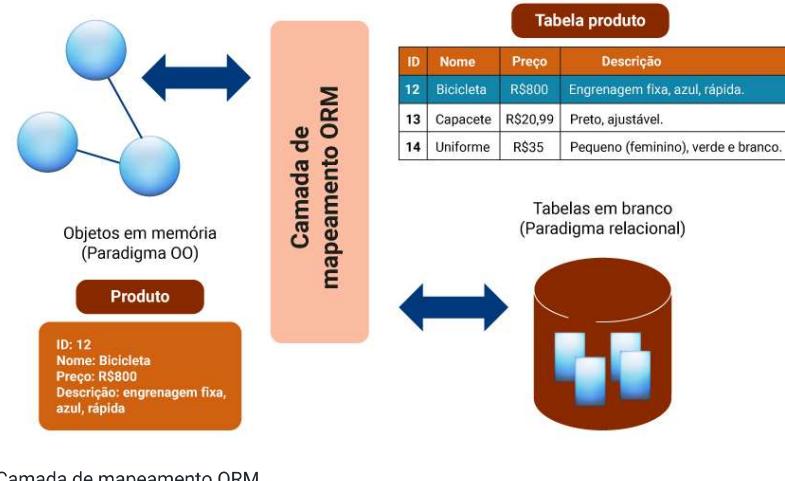
Olhando para o ambiente de programação orientada a **objetos**, temos classes, cujas instâncias, ou objetos, comportam valores, e que podem se relacionar com outras classes através de coleções ou atributos do tipo da classe relacionada. Não existe uma estrutura de indexação, mas uma relação bilateral, com o preenchimento de propriedades dos objetos envolvidos.

**Temos duas filosofias distintas, com peculiaridades inerentes a cada ambiente de execução, e como o ambiente de programação deve lidar com toda a lógica, ocorre um esforço natural para minimizar o uso de tabelas e registros, com a substituição por classes e objetos.**

A abordagem fica clara quando utilizamos o padrão de desenvolvimento DAO - Data Access Object -, no qual temos uma classe de entidade, e as consultas e operações sobre o banco de dados são concentradas em uma classe gestora, com a conversão para objetos e coleções, a partir da qual não utilizamos mais o enfoque relacional no sistema.

O uso de DAO deu origem à técnica de mapeamento **objeto-relacional**, ou **ORM**, na qual estamos nos referindo à forma como uma entidade

(objeto) é preenchida com os dados que vieram de um registro (relacional). Trata-se de um processo que inicialmente era efetuado de maneira programática, mas que foi modificado com o advento de **frameworks de persistência**, normalmente baseados no uso de XML - *Extended Markup Language* - ou **anotações**.



Com base nas configurações, indicando qual atributo da classe é refletido em determinado campo do registro, além da especificação de chaves e relacionamentos, o framework de persistência cria todos os comandos SQL - *Structured Query Language* necessários, de modo transparente, e transmite os comandos gerados para o banco de dados através de uma biblioteca de **middleware**.

## Comentário

No ambiente Java, os primeiros exemplos de tecnologias para ORM que obtiveram sucesso foram os **Entity Beans** e o **Hibernate**. As duas tecnologias adotaram o mapeamento baseado em XML, mas utilizaram padrões funcionais bastante distintos.

Os **Entity Beans** fazem parte do J2EE (Java 2 Enterprise Edition) e trabalham de acordo com o padrão **Active Record**, no qual cada leitura de propriedade equivale a um comando de seleção no banco de dados, a alocação de um objeto inicia um comando de inserção, e, quando alteramos o valor da propriedade, temos uma alteração no registro. Podemos chegar facilmente à conclusão de que o padrão é ineficiente, pois temos uma grande demanda de comandos SQL desnecessários, que poderiam ser executados em blocos, como pode ser visto a seguir:

Java

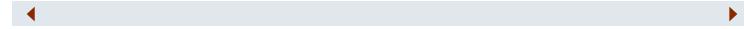


```

1 public abstract class ProdutoEntityBean implements E
2     public abstract int getCodigo();
3     public abstract void setCodigo(int codigo);

```

```
4     public abstract String getNome();  
5     public abstract void setNome(String nome);  
6     public abstract int getQuantidade();  
7     public abstract void setQuantidade(int quantidade)  
8     // O restante do código foi omitido  
9 }
```



No fragmento de código a seguir, temos o início da definição de um Entity Bean, onde o objeto concreto é gerado pelo servidor de aplicativos e as classes de entidade apresentam apenas as **propriedades**, definidas a partir de getters e setters **abstratos**, além de alguns métodos de gerência do ciclo de vida, aqui omitidos. O mapeamento do Entity Bean para a tabela deve ser feito com base na sintaxe XML, como no trecho apresentado a seguir, para o servidor **JBoss**. Observe:

XML



```
1 < enterprise-beans >  
2   < entity >  
3     < ejb-name >ProdutoEntityBean< /ejb-name >  
4     < table-name >PRODUTO< /table-name >  
5     < cmp-field >  
6       < field-name >codigo< /field-name >  
7         < column-name >COD_PRODUTO< /column-name >  
8       < /cmp-field >  
9     < cmp-field >  
10    < field-name >nome< /field-name >  
11      < column-name >NOME< /column-name >  
12    < /cmp-field >  
13    < cmp-field >  
14    < field-name >quantidade< /field-name >
```



Já no framework **Hibernate**, temos o padrão **DAO**, de forma implícita, com os comandos sendo gerados a partir dos métodos de um gestor de persistência, com base no conjunto de elementos de mapeamento, e os dados presentes nas entidades.

Java



```
1 public class Produto {
```

```
2     private int codigo;
3     private String nome;
4     private int quantidade;
5
6     public Produto(){}
7     // Os getters e setters das propriedades foram o
8 }
```

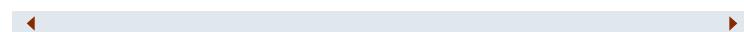


As entidades, para o **Hibernate**, são apenas classes comuns, sem métodos de negócios, com um conjunto de propriedades e um construtor padrão. Todo o mapeamento deve ser efetuado através da sintaxe **XML**, como no trecho apresentado a seguir.

XML



```
1 < hibernate-mapping schema="loja" >
2   < class name="model.Produto" table="PRODUTO" >
3     < id name="codigo" type="int" >
4       < column name="COD_PRODUTO" / >
5     < /id >
6     < property name="nome" type="string" column="NOM"
7       < property name="quantidade" type="int" column="
8     < /class >
9 < /hibernate-mapping >
```



Através do **XML**, temos um modelo documental para o mapeamento, retirando do código as referências aos elementos do banco de dados, o que garante maior flexibilidade e menor acoplamento, além de possibilitar a troca da base de dados com a simples alteração dos arquivos XML.

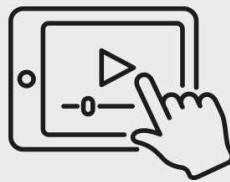
# JPA



## Java Persistence API (JPA) e Sintaxe JPQL

Entenda mais alguns aspectos do JPA associado à utilização da Sintaxe JPQL.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



### Java persistence API

Embora o uso de XML tenha se tornado um padrão muito aceito na configuração do mapeamento objeto-relacional, além de exigir a escrita de uma grande quantidade de texto, também traz a desvantagem de verificar eventuais erros apenas no momento da implantação. Uma nova abordagem passou a trabalhar com anotações de código, no lugar do XML, deixando a escrita mais concisa, e permitindo que o compilador efetue a análise sintática, com a detecção de muitos erros ainda na fase de codificação.

### Comentário

**Anotações** são metadados anexados ao código, utilizados ao nível da classe, atributos, métodos ou até parâmetros, e que permitem a leitura por ferramentas externas, com o objetivo de configurar funcionalidades adicionais.

Um dos maiores avanços do Java foi a criação do JPA, pois permitiu unificar os frameworks de persistência em uma arquitetura padrão, com apenas um arquivo de configuração, o **persistence.xml**. Não é apenas uma biblioteca, mas uma **API** que define a interface comum, configurável através de **anotações**, que deve ser seguida pelos

frameworks de persistência da plataforma Java, como Hibernate, Eclipse Link e Oracle Toplink.

Trabalhando no mesmo estilo do Hibernate, através do **JPA** temos um padrão **DAO** implícito, trazendo grande eficiência nas tarefas de persistência, o que ainda é otimizado com base em recursos de **cache** de entidades em **memória**. Não é por menos que, na plataforma JEE atual, temos a substituição dos Entity Beans, presentes no J2EE, pelo JPA.

Da mesma forma que no Hibernate clássico, para definir uma entidade JPA devemos criar um **POJO** (Plain Old Java Object), ou seja, uma classe sem métodos de negócios, mas com atributos definidos de forma privada e métodos de acesso públicos, além de um construtor padrão e alguns métodos utilitários, como **hash**. A entidade definida deve receber **anotações**, que serão responsáveis pelo **mapeamento** efetuado entre a classe e a tabela, ou seja, o mapeamento objeto-relacional, como pode ser visto a seguir:

Java



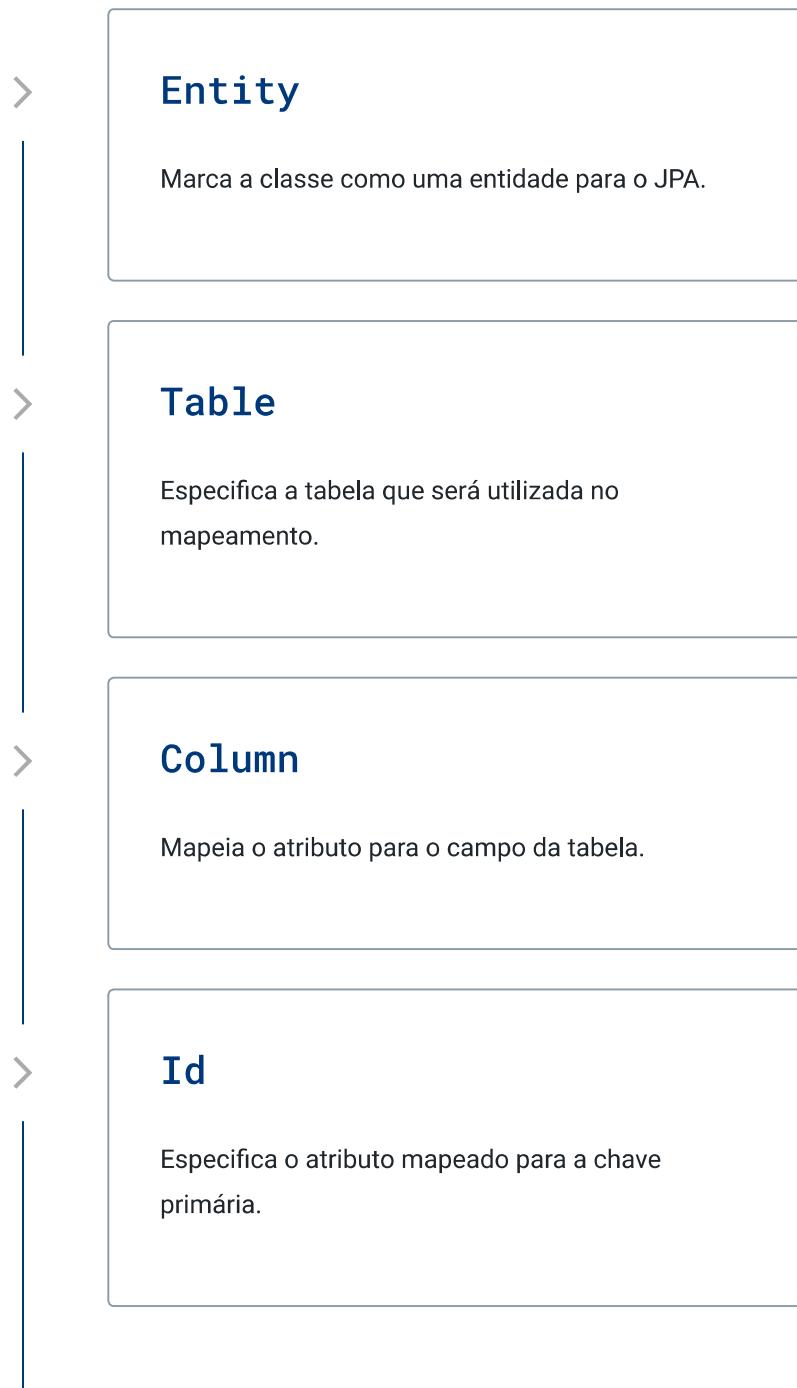
```
1 @Entity
2 @Table(name = "PRODUTO")
3 @NamedQueries({
4     @NamedQuery(name = "Produto.findAll",
5         query = "SELECT p FROM Produto p"))
6 public class Produto implements Serializable {
7     private static final long serialVersionUID = 1L
8     @Id
9     @Basic(optional = false)
10    @Column(name = "COD_PRODUTO")
11    private Integer codigo;
12    @Column(name = "QUANTIDADE")
13    private Integer quantidade;
14 }
```

A anotação **Entity** define a **classe** Produto como uma entidade para o JPA, enquanto **Table** especifica a tabela para a qual será mapeada no banco de dados, com base no parâmetro **name**. Temos ainda uma anotação **NamedQueries**, agrupando um conjunto de anotações **NamedQuery**, as quais são utilizadas para criar consultas através de uma sintaxe própria do JPA, denominada **JPQL** (Java Persistence Query Language).

## Dica

Após configurar a tabela que será representada pela entidade, precisamos completar as informações de mapeamento ao nível dos **atributos**, com o uso da anotação **Column** e o nome do campo definido no parâmetro **name**. Com o uso de **Id** definimos qual atributo representará a chave primária, e tornamos um atributo obrigatório através da anotação **Basic**, tendo o parâmetro **optional** configurado com valor **false**.

Vamos conferir as principais anotações para entrada do JPA?





## Basic

Define a obrigatoriedade do campo ou o modo utilizado para a carga de dados.



## OneToMany

Mapeia a relação 1XN do lado da entidade principal através de uma coleção.



## ManyToOne

Mapeia a relação 1XN do lado da entidade dependente, com base em uma classe de entidade.



## OneToOne

Mapeia o relacionamento 1X1 com atributos de entidade em ambos os lados.



## ManyToMany

Mapeia o relacionamento NXN com atributos de coleção em ambos os lados.



## OrderBy

Define a regra que será adotada para ordenar a coleção.



## JoinColumn

Especifica a regra de relacionamento da chave estrangeira ao nível das tabelas.

As entidades JPA devem conter dois **construtores**, sendo um vazio e outro baseado na chave primária, como podemos verificar no código de **Produto**, além dos métodos **equals** e **hashCode**.

Para a entidade **Produto**, os dois métodos utilitários são baseados no atributo **codigo**, o que é natural, já que ele recebe o valor da chave primária, tendo a capacidade de individualizar a instância da entidade em meio a uma coleção.

Além dos elementos descritos até aqui, utilizados na definição da estrutura de nossa entidade, precisamos do atributo **serialVersionUID**, referente à versão da classe, algo que será relevante para os processos de migração da base de dados. Temos ainda uma implementação de **toString**, que não é obrigatória, mas nos dá controle sobre a representação da entidade como texto em alguns componentes visuais, ou na impressão para a linha de comando.

Além das anotações nas entidades, precisamos de um arquivo de configuração, com o nome **persistence.xml**, onde serão definidos os aspectos gerais da conexão com o banco de dados. O arquivo deve ser criado na pasta **META-INF**, e os parâmetros podem incluir elementos como a classe de conexão **JDBC** (Java Database Connectivity) ou o pool de conexões do servidor, estando sempre presente a especificação do framework de persistência que será utilizado.

XML



```
1 < ?xml version="1.0" encoding="UTF-8"? >
2 < persistence version="2.1"
3   xmlns="http://xmlns.jcp.org/xml/ns/persistence"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-insta
```

```
5 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/p➥
6 http://xmlns.jcp.org/xml/ns/persistence/persisten➥
7 < persistence-unit name="ExemploSimplesJPAPU"➥
8     transaction-type="RESOURCE_LOCAL">
9     < provider >
10    org.eclipse.persistence.jpa.PersistenceProv➥
11    < /provider >
12    < class >model.Produto< /class >
13    < properties >
14
```

A primeira informação relevante é o nome da **unidade de persistência (ExemploSimplesJPAPU)**, com o tipo de **transação** que será utilizado. Transações são necessárias para garantir o nível de isolamento adequado entre tarefas, como no caso de múltiplos usuários acessando o mesmo banco de dados.

O controle transacional pode ocorrer a partir de um gestor próprio, para uso no ambiente **JSE** (Java Standard Edition) ou pelo **JEE** (Java Enterprise Edition) no modelo **não gerenciado**, mas também permite o modo gerenciado, através da integração com **JTA** (Java Transaction API). Observe, a seguir, a configuração das transações utilizadas pelo JPA:



## Resource local

Utiliza o gestor de transações do JPA, para execução no **JSE** ou no modelo **não gerenciado** do **JEE**.



## JTA

Ativa a integração com **JTA**, para utilizar o gerenciamento de transações pelo **JEE**.

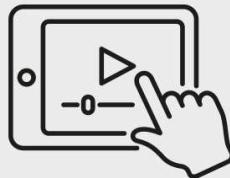
Em seguida, é definido o provedor de persistência no elemento **provider**, em nosso caso, a classe **PersistenceProvider** do **Eclipse Link**. As classes de entidade englobadas devem ser especificadas nos elementos **class**, para o modelo de acesso **local**, e as propriedades da conexão são definidas no grupo **properties**, o que inclui o driver **JDBC** utilizado, a **URL** de conexão, bem como o **usuário** e a **senha** do banco de dados.

# Manipulação de dados

## Consulta e manipulação de dados

Agora, é hora de se familiarizar com os principais pontos sobre consulta e manipulação de dados.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Com as entidades mapeadas e a conexão configurada, podemos efetuar as consultas e manipulações sobre os dados através de um objeto do tipo **EntityManager**. O gestor de entidades, ou EntityManager, concentra todos os métodos necessários para invocar os comandos **INSERT**, **UPDATE**, **DELETE** e **SELECT**, no banco de dados, montados pelo JPA a partir das anotações da entidade, de uma forma totalmente transparente. Observe a seguir:

Java



```
1 public class Principal {  
2     public static void main(String[] args) {  
3         EntityManagerFactory emf =  
4             Persistence.createEntityManagerFactory(  
5                 "ExemploSimplesJPAPU");  
6         EntityManager em = emf.createEntityManager();  
7         Query query = em.createNamedQuery("Produto.  
8             List< Produto > lista = query.getResultList();  
9             lista.forEach((e) -> {  
10                 System.out.println(e.getNome());  
11             });  
12         em.close();  
13     }  
14 }
```

Confira o passo a passo:

Passo 1



O primeiro passo em nosso código é a definição do **EntityManagerFactory**, utilizando o nome da **unidade de persistência** (ExemploSimplesJPAPU). Em seguida, obtemos uma instância de **EntityManager** a partir da fábrica de gestores, através de uma chamada para o método **createEntityManager**.

#### Passo 2

Com o gestor instanciado, obtemos um objeto do tipo **Query**, com a chamada para **createNamedQuery**, tendo como base uma anotação do tipo **NamedQuery** com **name** valendo **Produto.findAll**, a qual está anexada à classe **Produto**. As consultas nomeadas devem apresentar **nomes diferentes**, pois a busca é feita em todas as entidades, o que poderia gerar dualidades durante a execução.

#### Passo 3

O resultado da consulta ao banco de dados é obtido com a chamada para o método **getResultSet**, na forma de uma coleção de produtos do tipo **List**. Em termos práticos, a instrução JPQL é transformada em um comando SQL, o qual é transmitido para o banco de dados via JDBC, e o resultado da consulta é convertido em uma coleção de objetos, a partir do mapeamento efetuado com as anotações do JPA. Ao final, encerramos toda a comunicação com o banco de dados, utilizando o método **close** de nosso **EntityManager**. Embora o próprio coletor de lixo do Java possa causar a interrupção da conexão, a chamada para o método **close** é mais eficiente.

Fique atento:

Após receber a coleção de produtos, nós podemos percorrê-la através de um loop no estilo **foreach**, ou operador funcional equivalente, com a impressão do nome para cada produto encontrado.

Note que o **JPA** não elimina o uso de **JDBC**, pois o que temos é a geração dos comandos **SQL**, de forma automatizada, a partir das informações oferecidas pelas **anotações**.

Agora, podemos verificar como é feita a inclusão de um produto em nossa base de dados. Observe a seguir:

Java



```
1 public static void incluir(Produto p){  
2     EntityManagerFactory emf =  
3         Persistence.createEntityManagerFactory(  
4             "ExemploSimplesJPAPU");  
5     EntityManager em = emf.createEntityManager();  
6     try {  
7         em.getTransaction().begin();  
8         em.persist(p);  
9         em.getTransaction().commit();  
10    }catch(Exception e){  
11        em.getTransaction().rollback();  
12    }finally{  
13        em.close();  
14    }
```

Devido ao fato de uma inclusão poder gerar erros durante a execução, o ideal é que seja efetuada dentro de uma **transação**. Na verdade, qualquer manipulação de dados efetuada a partir do JPA exigirá uma transação, podendo ser local ou via JTA. Observe a sequência descrita a seguir:

Após obtermos uma instância de **EntityManager** na variável **em**, é definido um bloco de código protegido, onde a transação é iniciada com **begin**, seguida da inclusão do produto na base de dados através do método **persist**, e temos a confirmação da transação com o uso de **commit**.



Caso ocorra um erro, todas as alterações efetuadas são desfeitas com o uso de **rollback**, e ainda temos um trecho **finally**, onde fechamos a comunicação com o uso de **close**, independente da ocorrência de erros.

Para efetuar a alteração dos dados de um registro, temos um processo similar, trocando apenas o método **persist** por **merge**, observe:

Java



```
1 public static void alterar(Produto p){  
2     EntityManagerFactory emf =  
3         Persistence.createEntityManagerFactory(  
4             "ExemploSimplesJPAPU");  
5     EntityManager em = emf.createEntityManager();  
6     try {  
7         em.getTransaction().begin();  
8         em.merge(p);  
9         em.getTransaction().commit();  
10    }catch(Exception e){  
11        em.getTransaction().rollback();  
12    }finally{  
13        em.close();  
14    }
```

Na exclusão de um registro, inicialmente deve ser feita a busca, com base na classe da entidade e o valor da chave primária, através do método **find**. Será retornada uma entidade, como resultado da consulta, e nós a utilizaremos como parâmetro para a chamada ao método **remove**, efetuando a exclusão no banco de dados, como pode ser visto a seguir:

Java



```
1 public static void excluir(Integer codigo){  
2     EntityManagerFactory emf =  
3         Persistence.createEntityManagerFactory(  
4             "ExemploSimplesJPAPU");  
5     EntityManager em = emf.createEntityManager();  
6     try {  
7         em.getTransaction().begin();  
8         em.remove(em.find(Produto.class, codigo));  
9         em.getTransaction().commit();  
10    }catch(Exception e){  
11        em.getTransaction().rollback();  
12    }finally{  
13        em.close();  
14    }
```

Observando os trechos de código apresentados, podemos concluir facilmente que os métodos **find**, **persist**, **merge** e **remove** correspondem, respectivamente, à execução dos comandos **SELECT**, **INSERT**, **UPDATE** e **DELETE**, ao nível do banco de dados.

## Execução local de aplicativo com JPA

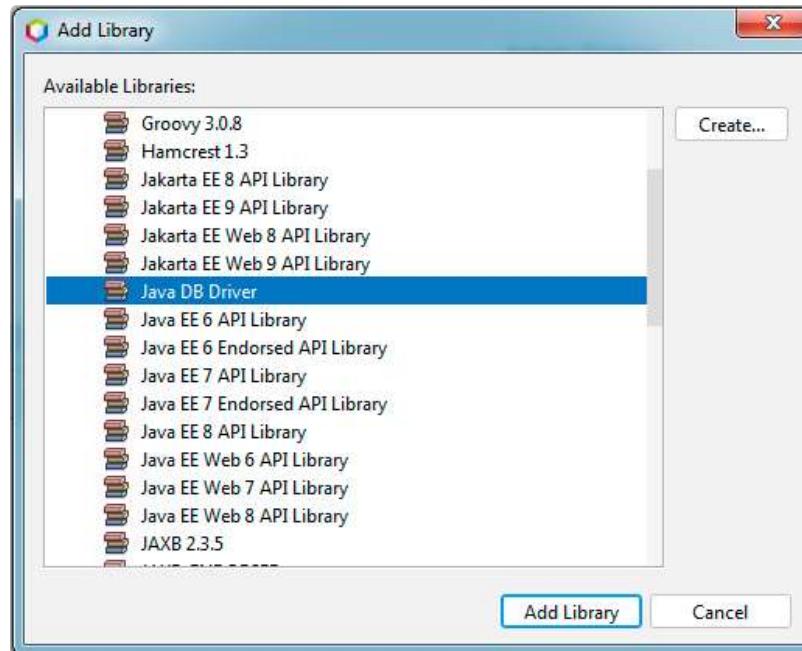
### Execução do aplicativo

Quando executamos um aplicativo com elementos JPA no servidor, temos todas as bibliotecas necessárias disponíveis, mas a execução local necessitará da inclusão de algumas bibliotecas ao projeto.

Trabalharemos com o banco de dados **Derby**, também chamado de **Java DB**, exigindo a inclusão da biblioteca **JDBC** correta.

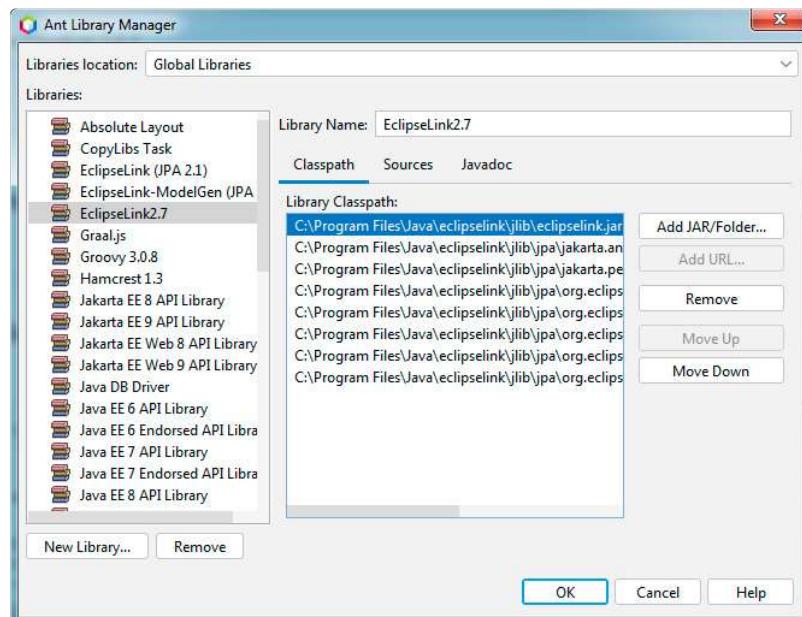
**É interessante observar que o Derby é um banco de dados implementado em Java, que faz parte da distribuição padrão da plataforma, e que pode ser executado de forma local ou no servidor.**

Para adicionar a biblioteca JDBC do Derby ao projeto, vamos clicar com o botão direito sobre a divisão **Libraries**, e escolher a opção **Add Library**. Na janela que se abrirá, selecionaremos apenas a opção **Java DB Driver** e clicaremos no botão **Add Library**. Veja a janela a seguir:

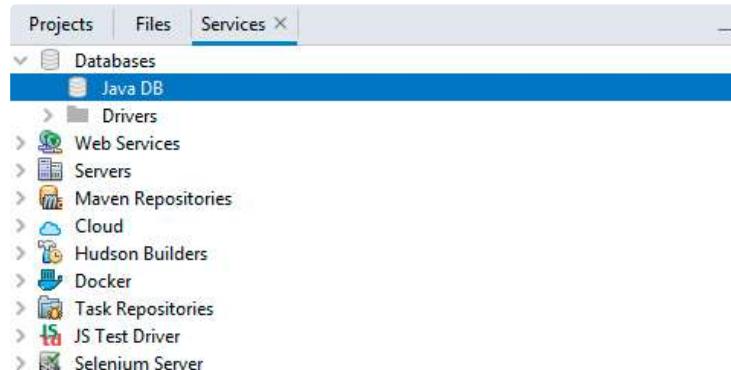


Precisamos acrescentar o framework JPA escolhido, no caso, o **Eclipse Link**, disponível para download na seção "Explore +".

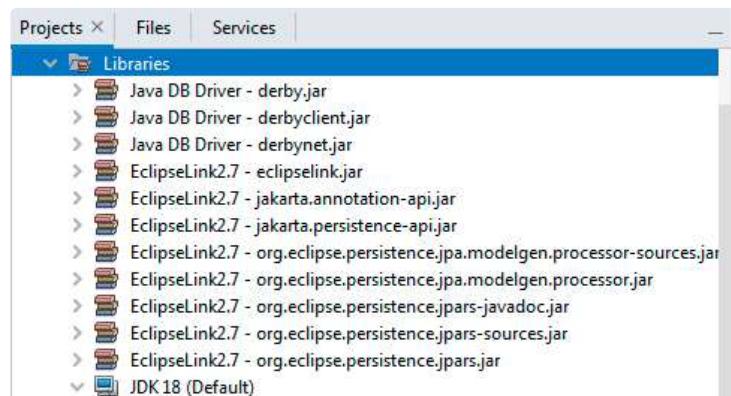
Após efetuar o download da versão mais recente do **Eclipse Link**, no formato **zip**, e extrair para algum diretório de fácil acesso, crie uma biblioteca através da opção de menu **Tools.Libraries**, seguido do clique sobre o botão **New Library**. Daremos a ela o nome **EclipseLink2.7**, e adicionaremos o arquivo **eclipselink.jar**, presente no diretório **jlib**, além de todos os arquivos no formato **jar** do subdiretório **jpa**. Veja a seguir:



Agora acompanhe os passos a seguir:

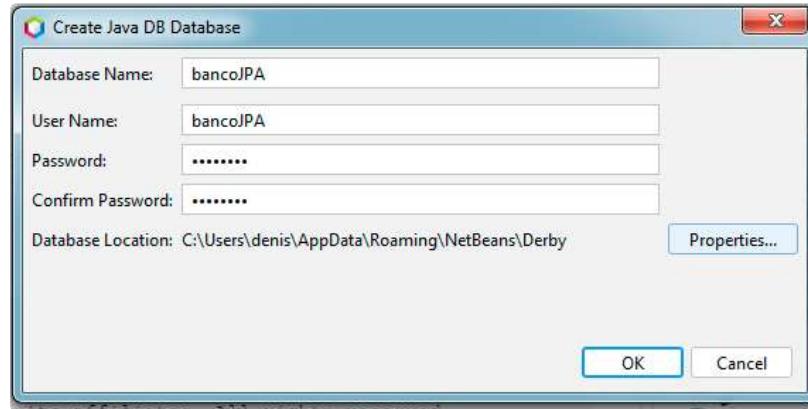


Após a definição da nova biblioteca, vamos adicionar ao projeto, da mesma forma que fizemos para o driver JDBC, clicando com o botão direito sobre a divisão **Libraries**, e escolhendo a opção **Add Library**. Ao final, teremos a configuração de bibliotecas para o projeto como a que é apresentada a seguir.



Agora só precisamos de um banco de dados Derby, que será criado de forma simples, através da aba **Services** do NetBeans, na divisão **Databases**.

Para criarmos um banco de dados, precisamos clicar com o botão direito sobre o driver **Java DB**, selecionável com a abertura da árvore de **Databases**, seguido da escolha da opção **Create Database**, no menu de contexto. Na janela que será aberta, efetuaremos o preenchimento do nome de nosso novo banco de dados com o valor "**bancoJPA**", bem como usuário e senha, onde podemos também utilizar o valor "**bancoJPA**" para ambos. Observe:



Ao clicar no botão de confirmação, o banco de dados será criado e ficará disponível para conexão, através do driver **JDBC**. A conexão é identificada por sua **Connection String**, tendo como base o endereço de rede (**localhost**), a porta padrão (**1527**) e a instância (**bancoJPA**).

A conexão é aberta com o duplo-clique sobre o identificador, ou o clique com o botão direito e escolha da opção **Connect**. Com o banco de dados aberto, vamos executar os comandos SQL necessários para a criação e alimentação da tabela, clicando com o botão direito sobre a conexão e escolhendo a opção **Execute Command**.

Veremos que a janela de edição de SQL será aberta, permitindo que seja digitado o script apresentado a seguir. Para executar nosso script, devemos pressionar **CTRL+SHIFT+E**, ou clicar sobre o botão de execução de SQL, disponibilizado na parte superior do editor. Observe:

SQL ✖

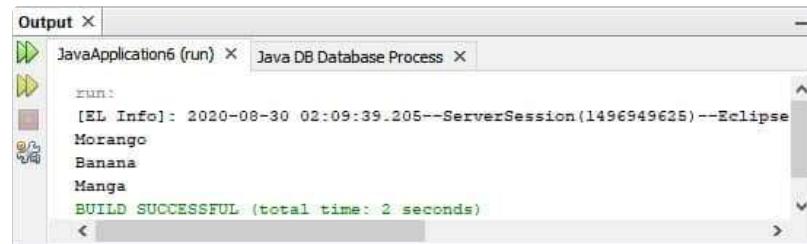
```

1 CREATE TABLE PRODUTO (
2     COD_PRODUTO INTEGER NOT NULL PRIMARY KEY,
3     NOME VARCHAR(50),
4     QUANTIDADE INTEGER);
5
6 INSERT INTO PRODUTO VALUES (1,'Morango',200);
7 INSERT INTO PRODUTO VALUES (2,'Banana',1000);
8 INSERT INTO PRODUTO VALUES (3,'Manga',600);
9
10 SELECT * FROM PRODUTO;

```

Ao final da execução, será apresentada a listagem da tabela, com os registros inseridos, na própria janela de edição, em uma divisão própria,

e agora podemos executar nosso projeto, gerando a saída apresentada a seguir.



```
[EL Info]: 2020-08-30 02:09:39.205--ServerSession(1496949625)--Eclipse Morango
Banana
Manga
BUILD SUCCESSFUL (total time: 2 seconds)
```

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

### Questão 1

O uso de mapeamento objeto-relacional se tornou uma necessidade básica para os sistemas cadastrais modernos, e diversos modelos foram adotados, normalmente com o uso de XML ou anotações. Entre as tecnologias utilizadas, temos os Entity Beans, do J2EE, que baseiam sua funcionalidade no padrão de desenvolvimento:

A Active Record

B DAO

C Facade

D Adapter

E Front Controller

Parabéns! A alternativa A está correta.

Ao utilizar os Entity Beans, qualquer operação efetuada sobre as entidades causa um efeito diretamente no banco de dados, ou seja, a criação gera um comando de INSERT, a remoção define o comando DELETE, e as alterações dos valores de atributos geram múltiplos comandos UPDATE, caracterizando o padrão Active Record.

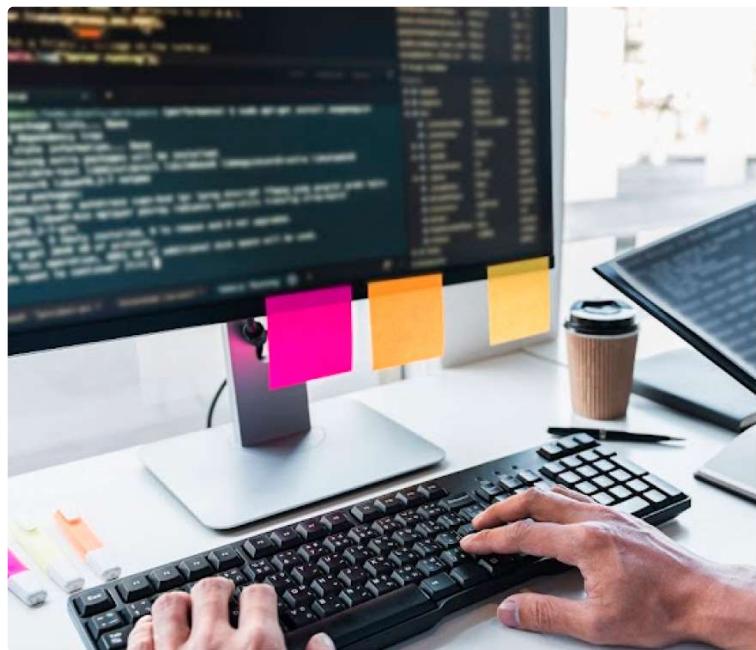
## Questão 2

A tecnologia JPA, assim como outras plataformas de mapeamento objeto-relacional, permite abstrair os comandos SQL e o uso de registros, trabalhando com entidades e coleções, que são retornadas diretamente a partir de comandos JPQL. O resultado de uma consulta pode ser obtido através do método getResultList, pertencente à classe:

- A Persistence
- B EntityManager
- C Query
- D Transaction
- E EntityManagerFactory

**Parabéns! A alternativa C está correta.**

A classe Query apresenta diversos métodos para consulta e manipulação de dados, com base na sintaxe JPQL, como getResultList, para a obtenção do resultado através de uma coleção, getSingleResult, para receber uma entidade simples, ou ainda, executeUpdate, para efetuar alterações e exclusões de entidades.



## 2 - Componentes Enterprise Java Beans

Ao final deste módulo, você será capaz de empregar componentes EJB na construção de regras de negócio.

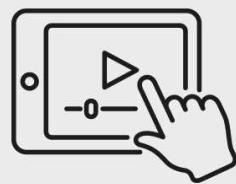
### Visão geral



### Arquitetura e tipos de Enterprise - Java Beans (EJB)

Conheça agora a tecnologia de objetos distribuídos dentro do Java.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



### Enterprise Java beans

De forma geral, uma arquitetura de objetos distribuídos é o elemento central para qualquer servidor de aplicativos. No ambiente Java, não poderia ser diferente, como podemos observar no uso de componentes **EJB** (Enterprise Java Bean) em servidores como **GlassFish**, **JBoss** e **WebSphere**. Veja a representação a seguir:



Conceito de troca de mensagens entre diferentes camadas de arquitetura.

Um **EJB** é um componente corporativo, utilizado de forma indireta, dentro de um ambiente de objetos distribuídos, suportando transações locais e distribuídas, recursos de autenticação e segurança, acesso a banco de dados via pool de conexões, e demais elementos da plataforma **JEE** (Java Enterprise Edition). Todo EJB executa dentro de um **pool** de objetos, em que o número de instâncias irá aumentar ou diminuir, de acordo com a demanda de solicitações efetuadas, segundo um intervalo de tempo estabelecido.

## Comentário

Um pool de objetos segue o padrão de desenvolvimento **Flyweight**, no qual o objetivo é responder a uma grande quantidade de requisições através de um pequeno conjunto de objetos.

O acesso aos serviços oferecidos pelo pool de EJBs deve ser solicitado a partir de uma interface local (**EJBLocalObject**) ou remota (**EJBObject**), onde as interfaces são geradas a partir de componentes de **fábrica**, criadas com a implementação de **EJBLocalHome**, para acesso local, ou **EJBHome**, para acesso remoto. Como é padrão na plataforma Java, as fábricas são registradas e localizadas via **JNDI** (Java Naming and Directory Interface).

O processo para acessar o pool de EJBs envolve três passos:

1

## Cliente acessa a fábrica de interfaces através de JNDI.

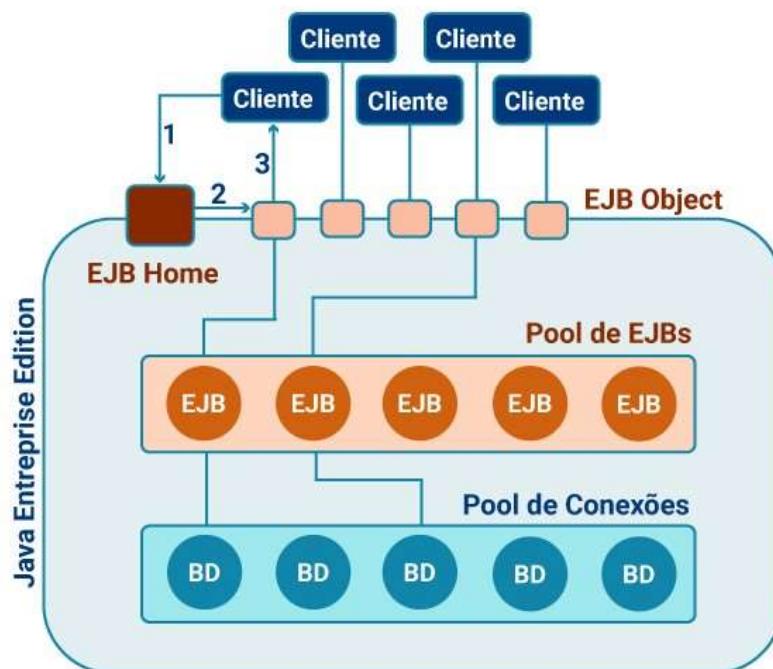
2

A interface de acesso é gerada pela fábrica.

3

Cliente recebe a interface, podendo iniciar o diálogo com o pool.

Observe a representação a seguir:



Representação do ambiente pool de EJBs.

Podemos observar muitos **padrões de desenvolvimento** no ambiente dos **EJBs**:

### Service locator

Utilizado no acesso aos componentes registrados via JNDI.

## Abstract factory

Utilizado na definição da fábrica de interfaces.

# Proxy

Utilizado na comunicação com clientes remotos.

O acesso ao banco de dados, no ambiente JEE, também ocorre de forma diferenciada, pois, em vez de uma conexão JDBC simples, obtida a partir do **DriverManager**, temos um **pool de conexões** JDBC, representado por um objeto do tipo **DataSource**, que, assim como os EJBs, é registrado via **JNDI**. Quando solicitamos uma conexão ao **DataSource**, não estamos abrindo uma nova conexão, mas reservando uma das disponíveis no pool, e quando invocamos o método **close**, não ocorre a desconexão, mas, sim, a liberação da conexão para a próxima requisição.

No fragmento de código seguinte, temos um exemplo de acesso e utilização de um pool de conexões, na linguagem Java, que poderia ser utilizado em um **Servlet**. Após obter o recurso via **JNDI**, através do método **lookup** de **InitialContext**, efetuamos a conversão para o tipo correto, no caso um **DataSource**, o qual fornece conexões através do método **getConnection**, e a partir daí, a programação é a mesma de um acesso local ao banco de dados, com a única diferença ocorrendo no comportamento do método **close**, que agora libera a conexão no pool para o próximo requisitante:

Java



```
1 protected void processRequest(HttpServletRequest  
2                                     throws ServletException  
3 {  
4     response.setContentType("text/html; charset=UTF-8");  
5     try (PrintWriter out = response.getWriter()) {  
6         out.println("< html >< body >");  
7         try {  
8             InitialContext ctx = new InitialContext();  
9             DataSource dts = (DataSource) ctx.lookup("jndi/DS");  
10            Connection c1 = dts.getConnection();  
11            Statement st = c1.createStatement();  
12            String sql = "SELECT * FROM users";  
13            ResultSet rs = st.executeQuery(sql);  
14            while (rs.next()) {  
15                String id = rs.getString("id");  
16                String name = rs.getString("name");  
17                String email = rs.getString("email");  
18                System.out.println(id + " " + name + " " + email);  
19            }  
20        } catch (NamingException e) {  
21            e.printStackTrace();  
22        }  
23    }  
24 }  
25 }
```

```
11     ResultSet rs = st.executeQuery("SELECT * FR▲  
12     while(rs.next())  
13         out.println(rs.getString("NOME")+"");  
14     ▶
```

Como utilizamos **JPA**, não é necessário efetuar toda essa codificação para localização e utilização do pool, ficando a cargo do framework de persistência. No fluxo normal de execução, o cliente faz uma solicitação para a interface de acesso, que é repassada para o pool de **EJBs**, sendo disponibilizado um deles para responder, e na programação do **EJB**, utilizamos o **JPA** para obter acesso ao banco de dados a partir do **DataSource**, com o controle transacional sendo efetuado através do **JTA** (Java Transaction API).

## Curiosidade

No **J2EE**, existia um EJB para persistência, denominado **Entity Bean**, que seguia o padrão **Active Record**, mas ele se mostrou inferior, em termos de eficiência, quando comparado a alguns frameworks de persistência, sendo substituído pelo **JPA** no **JEE5**.

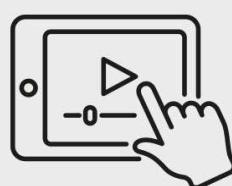
A programação, no modelo adotado a partir do **JEE5**, é bastante simples, e precisaremos apenas das **anotações** corretas para que os Application Servers, como o JBoss ou o GlassFish, se encarreguem de montar toda a estrutura necessária. Isso é bem diferente do processo de criação adotado pelo **J2EE**, com implementação baseada em contrato, envolvendo uma grande quantidade de interfaces, classes e arquivos **XML**, e tendo a verificação apenas no momento da implantação do sistema, já que não era um modelo de compilação formal.

# Sessão e mensagerias

## Session Beans

Conheça agora uma abordagem teórica sobre o Session Beans.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Quando trabalhamos com os componentes do tipo EJB, estamos preocupados com as **regras de negócio** de nosso aplicativo, ou seja, visamos à implementação da lógica do sistema, com base nas **entidades** e nos **requisitos** definidos, sem qualquer preocupação com a estrutura de persistência que será utilizada. Também devemos observar que as regras de negócio devem ser totalmente independentes das interfaces do sistema, não devendo ser direcionadas para nenhum ambiente específico.

O primeiro tipo de EJB que deve ser observado é o de **sessão**, responsável por efetuar processos de negócios de forma síncrona, e configurável de três formas distintas, como podemos observar a seguir:

---

## Stateless

Aqui, não é permitida a manutenção de estado, ou seja, não se guardam valores entre chamadas sucessivas.

---

## Stateful

Aqui, utiliza-se quando é necessário manter valores entre chamadas sucessivas, como no caso de somatórios.

## Singleton

Aqui, permite-se apenas uma instância por máquina virtual, garantindo o compartilhamento de dados entre todos os usuários.

Utilizamos **Stateless** quando não precisamos de informações dos processos anteriores ao corrente. Qualquer instância do pool de EJBs pode ser escolhida, e não é necessário efetuar a carga de dados

anteriores, definindo o padrão de comportamento mais ágil para um Session Bean.

O comportamento **Stateful** deve ser utilizado apenas quando precisamos de informações anteriores, como em uma cesta de compras virtual, ou processos com acumuladores em cálculos estatísticos, entre outras situações com necessidade de gerência de estados.

Antes de definir um Session Bean, devemos definir sua interface de acesso, com base na anotação **Local**, para acesso interno, ao nível do servidor, ou **Remote**, permitindo que o componente seja acessado remotamente. Em nossos estudos, o uso de acesso local será suficiente, já que teremos o acionamento dos EJBs a partir dos **Servlets**.

Java



```
1 @Local
2 public interface CalculadoraLocal {
3     int somar(int a, int b);
4 }
```

Ao criarmos o EJB, ele deverá **implementar** a interface de acesso, além de ser anotado como **Stateless** ou **Stateful**, dependendo da necessidade do negócio. Para uma calculadora simples, não precisaríamos de gerência de estados.

Java



```
1 @Stateless
2 public class Calculadora implements CalculadoraLocal {
3     @Override
4     public int somar(int a, int b) {
5         return a + b;
6     }
7 }
```

Quanto ao EJB do tipo **Singleton**, ele é utilizado quando queremos compartilhar dados entre todos os usuários conectados, mesmo na execução sobre múltiplas máquinas virtuais, em ambientes distribuídos. Não podemos esquecer que os EJBs são uma tecnologia corporativa, e que a execução de forma **clusterizada** não é uma exceção em sistemas de **missão crítica**.

## Comentário

Em um **cluster**, temos um conjunto de computadores atuando como se fossem apenas um, o que traz grande poder de processamento e menor possibilidade de interrupções, já que a falha de um computador causará a redistribuição das tarefas para os demais

Para utilizar um componente do tipo **Session Bean** a partir de um **Servlet**, o processo é trivial, como podemos observar a seguir:

Java



```
1  @WebServlet(name = "ServletSoma", urlPatterns = {▲
2  public class ServletSoma extends HttpServlet {
3
4      @EJB
5      CalculadoraLocal facade;
6
7      protected void doGet(HttpServletRequest request
8                      throws ServletException,
9                      response.setContentType("text/html;charset=UTF
10                 try (PrintWriter out = response.getWriter())
11                     out.println("< html >< body >");
12                     out.println("< h1 >Servlet ServletSoma: " +
13                         out.println("< /body >");
14                     out.println("< /html >")•
```

Tudo que precisamos fazer é anotar um atributo, do tipo da interface **local**, com **EJB**, e no código de resposta à chamada HTTP, invocamos os métodos da interface, como se fossem chamadas locais. No exemplo, temos a interface **CalculadorLocal** referenciada no

atributo **facade**, o que permite invocar o método **somar**, sendo executado pelo **pool**.

### Dica

O uso da interface **CalculadoraLocal** permite solicitar os serviços do **Session Bean**, em meio ao **pool** de objetos, como se fossem simples chamadas **locais**.

Quase todos os processos de negócio de um sistema corporativo, na plataforma JEE, são implementados através de Session Beans, mas alguns comportamentos não podem ser definidos de forma síncrona, exigindo um componente adequado para a comunicação com mensagerias, segundo um modelo assíncrono.

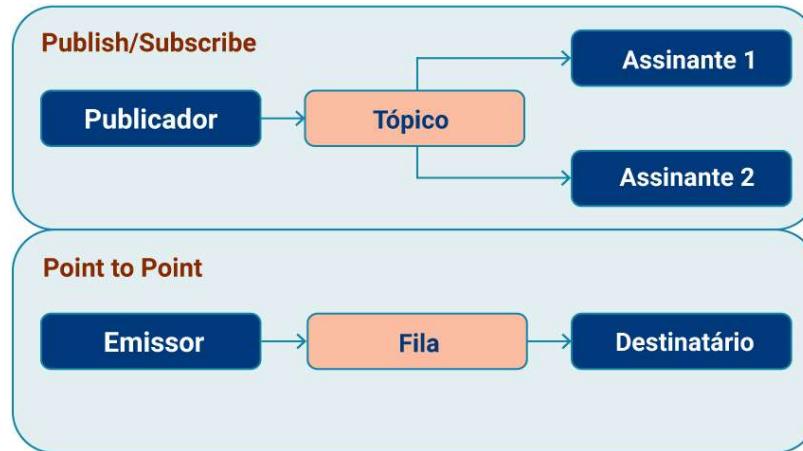
## Message Driven Beans

Conheça a tecnologia de mensagerias Message Driven Beans.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



A tecnologia de **mensagerias** é muito importante para satisfazer aos requisitos de um sistema corporativo. Elas atuam de forma **assíncrona** e permitem a adoção do domínio **point to point**, onde as mensagens são enfileiradas para um tratamento sequencial no destinatário, ou **publish/subscribe**, onde as mensagens são depositadas em tópicos, para que os assinantes as recuperem. Observe o esquema a seguir:



Tecnologia de **mensagerias**.

O uso de mensagerias permite a construção de sistemas B2B - *Business to Business* -, quase sem acoplamento, em que o único elemento de

ligação entre os sistemas das duas empresas é a mensagem transmitida entre elas. A rede bancária utiliza amplamente as mensagerias, ao efetuar transações eletrônicas entre instituições diferentes, o que possibilita que um cliente solicite uma transferência, sem precisar esperar que ela seja concluída, já que não ocorre bloqueio do emissor.

Após o cliente (**emissor** ou **publicador**) postar uma mensagem, a responsabilidade sobre a gerência dela passa a ser da mensageria, até o momento em que seja retirada pelos **receptores**. Já que as mensagerias funcionam como **repositórios**, gerenciando a persistência das mensagens, mesmo que o receptor esteja inativo, as mensagens não se perdem, sendo acumuladas até o momento em que o receptor seja ativado.

Para criar filas ou tópicos no **GlassFish**, é necessário utilizar o comando **asadmin**, como no exemplo seguinte, para a criação de uma fila denominada **jms/SimpleQueue**.

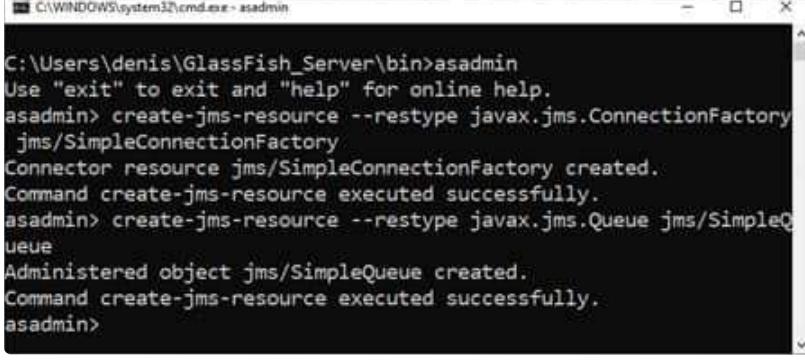
Terminal



```
1 asadmin create-jms-resource --restype javax.jms.ConnectionFactory  
2           jms/SimpleConnectionFactory  
3  
4 asadmin create-jms-resource --restype javax.jms.Queue  
5           jms/SimpleQueue
```



Também podemos abrir o console do **asadmin**, sem a passagem de parâmetros, e depois invocar os comandos internamente, como pode ser observado a seguir:



```
C:\Users\denis\GlassFish_Server\bin>asadmin
Use "exit" to exit and "help" for online help.
asadmin> create-jms-resource --restype javax.jms.ConnectionFactory
        jms/SimpleConnectionFactory
Connector resource jms/SimpleConnectionFactory created.
Command create-jms-resource executed successfully.
asadmin> create-jms-resource --restype javax.jms.Queue jms/SimpleQueue
Administered object jms/SimpleQueue created.
Command create-jms-resource executed successfully.
asadmin>
```

Console do asadmin.

Existe um tipo de EJB denominado MDB (Message Driven Bean), que tem como finalidade a comunicação com mensagerias via JMS (Java Message Service), possibilitando o processamento assíncrono no JEE. Através do MDB é possível trabalhar nos dois domínios de mensagerias, com o tratamento de mensagens sendo feito através do **pool** de EJBs, a partir de um único método, que tem o nome **onMessage** representando os eventos de recepção de mensagens. Observe o exemplo a seguir:

Java



```
1  @MessageDriven(activationConfig = {
2      @ActivationConfigProperty(propertyName = "desti
3          propertyName = "jms/SimpleQueue"),
4      @ActivationConfigProperty(propertyName = "desti
5          propertyName = "javax.jms.Queue")
6  })
7  public class Mensageiro001 implements MessageLi
8
9  public Mensageiro001() {
10 }
11
12 @Override
13 public void onMessage(Message message) {
14     +nv
```

No código de exemplo, temos uma anotação **MessageDriven**, para a definição do **MDB**, com a configuração para acesso a **jms/SimpleQueue**, do tipo **javax.jms.Queue**, através de anotações **ActivationConfigProperty**. Também é possível utilizar canais internos do projeto, mas o uso de canais do servidor viabiliza o comportamento B2B, com acesso a partir de qualquer plataforma que dê suporte ao uso de mensagerias.

Para o tratamento das mensagens, devemos implementar a interface **MessageListener**, que contém apenas o método **onMessage**.

A mensagem é recebida no parâmetro do tipo **Message**, que pode ser qualquer descendente da classe, segundo os princípios da orientação a objetos, o que torna necessário converter para o tipo correto, como no exemplo, onde temos a captura de um texto enviado via **TextMessage**, e a impressão da mensagem no console do GlassFish.

### Atenção!

O **MDB** foi projetado exclusivamente para receber mensagens, a partir de filas ou tópicos, o que faz com que não possa ser acionado diretamente, como os **Session Beans**. Para sua ativação, basta que um cliente poste uma mensagem.

Observe o código a seguir:

Java



```
1 @WebServlet(name = "ServletMessage", urlPatterns ^  
2 public class ServletMessage extends HttpServlet {  
3     @Resource(mappedName = "jms/SimpleConnectionFactory")  
4     private ConnectionFactory connectionFactory;  
5     @Resource(mappedName = "jms/SimpleQueue")  
6     private Queue queue;  
7  
8     public void putMessage() throws ServletException  
9     try {  
10         Connection connection = connectionFactory.c  
11         Session session = connection.createSession(  
12         MessageProducer messageProducer = session.c  
13         TextMessage message = session.createTextMes  
14         message.setText("Teste com MDB").  
▼
```

O processo é um pouco mais complexo que o adotado para os **Session Beans**, porém apresenta menor acoplamento. Observe as etapas a seguir:

1

#### Passo 1

Inicialmente, devemos mapear a fábrica de conexões da mensageria e a fila de destino do MDB, através de anotações **Resource**.

2

## Passo 2

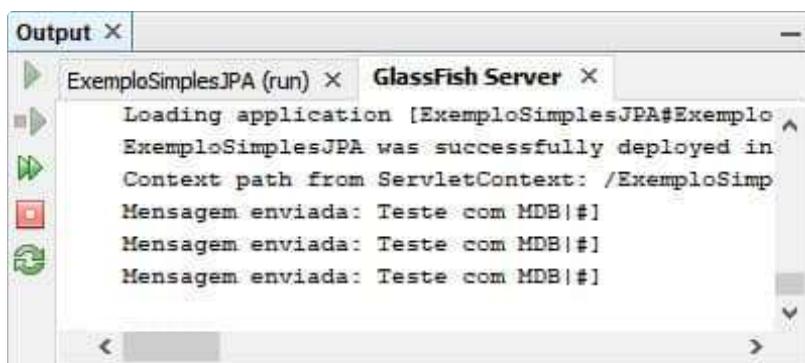
Com os recursos mapeados, definimos o método **putMessage**, para envio da mensagem, onde devemos criar uma **conexão** (Connection) a partir da fábrica, a **sessão** (Session) a partir da conexão, e o **produtor de mensagens** (MessageProducer) a partir da sessão. Na inicialização do MessageProducer, utilizamos o recurso de fila mapeado, indicando que ele deve apontar para a **fila** (Queue) destinada ao **MDB**.

3

## Passo 3

Após configurar a conexão, criamos a **mensagem de texto** (TextMessage) através da **sessão**, definimos o texto que será enviado com **setText**, e finalmente enviamos a mensagem, a partir do **produtor**, com o uso de **send**, finalizando a definição do método **putMessage**. Com o método completo, podemos utilizá-lo na resposta do Servlet ao protocolo HTTP, e verificar a mensagem na saída do GlassFish.

Confira o resultado na imagem a seguir:



The screenshot shows the GlassFish Server Output window. It displays the deployment log for the application 'ExemploSimplesJPA'. The log entries include:

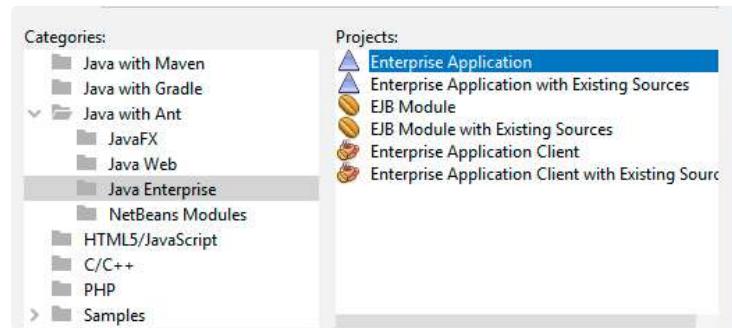
- 'Loading application [ExemploSimplesJPA#Exemplo]
- 'ExemploSimplesJPA was successfully deployed in Context path from ServletContext: /ExemploSimplesJPA'
- 'Mensagem enviada: Teste com MDB#[...]' (repeated three times)

Mensagem de saída do GlassFish.

# EJBs no NetBeans

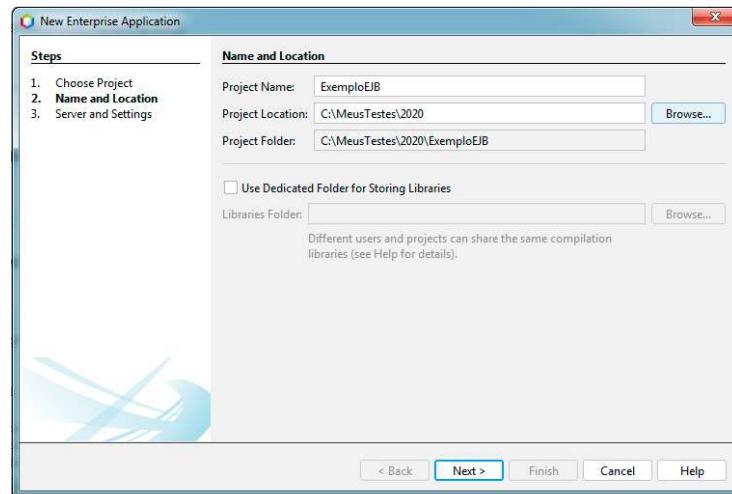
## Aplicativo corporativo

Para que possamos trabalhar com **EJBs**, através do ambiente do **NetBeans**, devemos definir um projeto corporativo. A sequência de passos para criar o Aplicativo Corporativo pode ser observada a seguir:



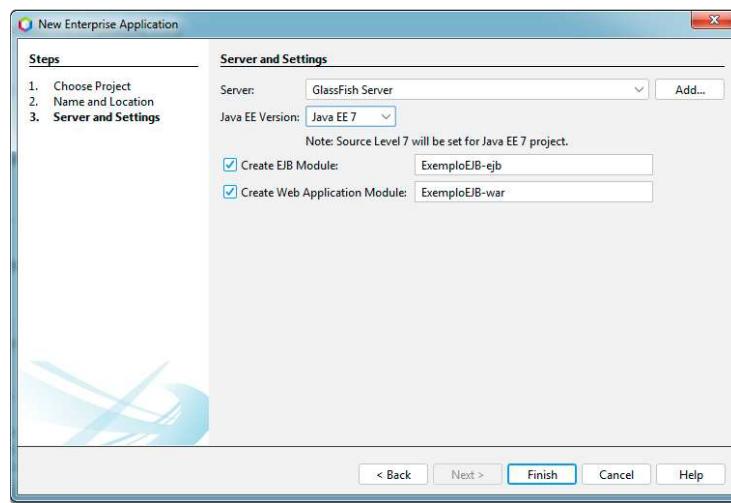
### Passo 1

Criar um projeto do tipo Enterprise Application, na categoria Java Enterprise.



### Passo 2

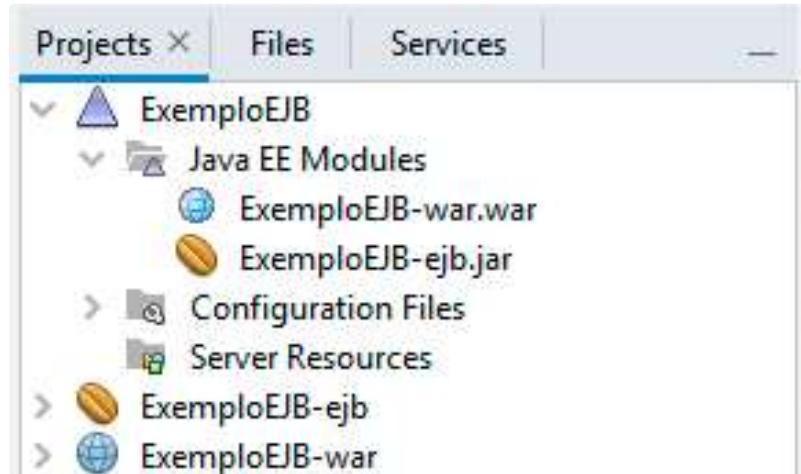
Preencher o nome (ExemploEJB) e local do projeto.



### Passo 3

Escolher o servidor (GlassFish) e versão do JEE (Java EE7), além de marcar as opções de criação para os módulos EJB e Web.

Seguindo os passos anteriores corretamente, serão gerados três projetos, e podemos visualizá-los na interface do NetBeans, como mostrado na imagem a seguir:



Projetos gerados na interface NetBeans.

Conheça a seguir as características de cada um dos projetos gerados para o Aplicativo Corporativo.



## ExemploEJB-ejb

Este projeto é utilizado na definição das entidades **JPA** e dos componentes **EJB**, sendo compilado com a extensão "jar".



## ExemploEJB-war

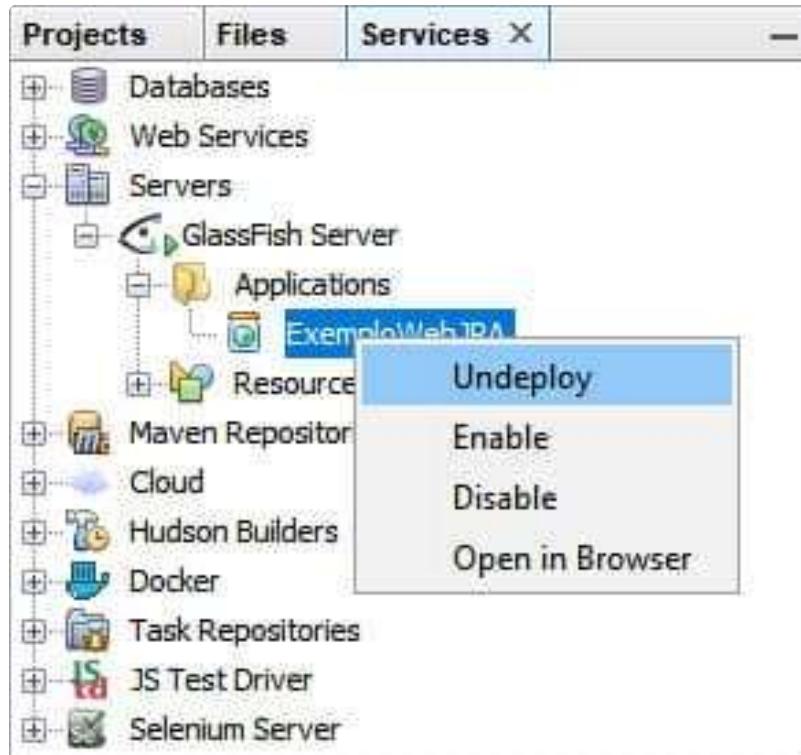
Este projeto contém os elementos para **Web**, como Servlets, Facelets e páginas XHTML, compilados para um arquivo "war".



## ExemploEJB

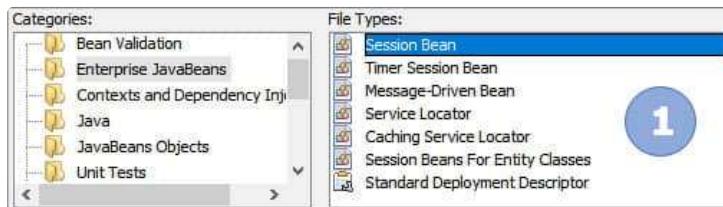
Este projeto agrupa os dois projetos anteriores, compactados em apenas um arquivo, com adoção da extensão "**ear**", para implantação.

Quando trabalhamos com um projeto corporativo, devemos implantar o projeto principal, com extensão **ear** (Enterprise Archived), cujo ícone é um **triângulo**, pois qualquer tentativa de implantar os dois projetos secundários irá impedir a execução correta do conjunto, exigindo que seja feita a remoção manual dos projetos anteriores pela aba de Serviços, como pode ser visto a seguir:



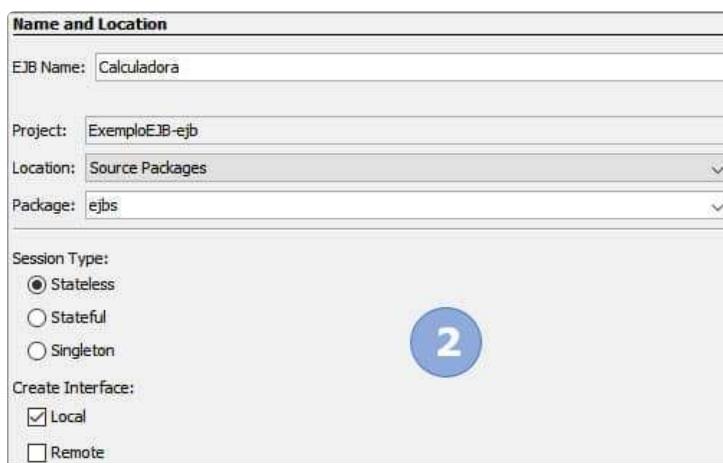
Remoção manual dos projetos anteriores pela aba de Serviços.

Agora, vamos criar nosso primeiro **Session Bean**, configurado como **Stateless**, no projeto secundário **ExemploEJB-ejb**, adicionando um novo arquivo e seguindo os seguintes passos:



## Passo 1

Selecionar o tipo Session Bean na categoria Enterprise Java Beans.



## Passo 2

Definir o nome (Calculadora) e pacote (ejbs) do novo Session Bean, escolher o tipo como Stateless e marcar apenas a interface Local.

Em diversas situações, a IDE mostra um **erro de compilação**, decorrente da importação dos componentes da biblioteca **javax.ejb**. Caso o problema ocorra, a solução é simples, com a inclusão da biblioteca **Java EE 7 API** no projeto, como pode ser observado na imagem a seguir:



Biblioteca Java EE 7 API inserida.

Após incluir as bibliotecas necessárias, podemos completar os códigos de **Calculadora** e **CalculadoraLocal**, de acordo com os exemplos apresentados anteriormente, e iremos testar o EJB, ainda seguindo os exemplos, através de um **Servlet**. Já que os componentes Web são criados ao nível de **ExemploEJB-war**, devemos acessar o projeto e adicionar novo arquivo do tipo **Servlet**, na categoria **Web**, com o nome **ServletSoma** e pacote **servlets**, sem adicionar informações ao arquivo **web.xml**.

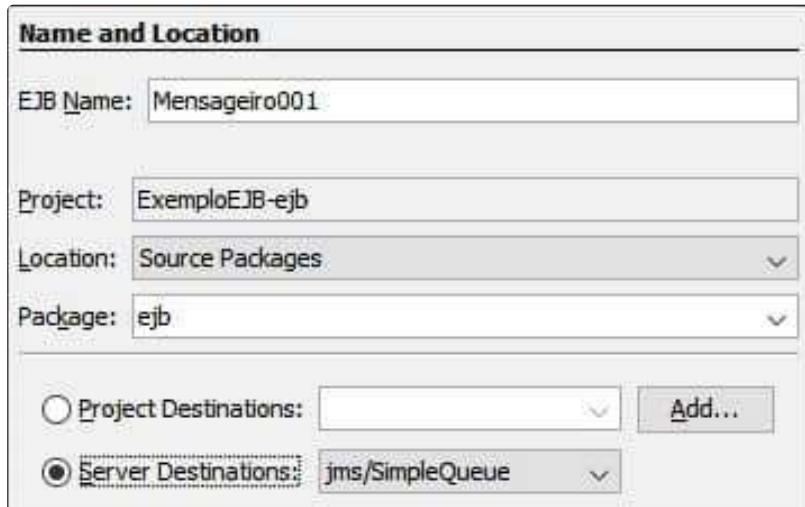
**Com o componente ServletSoma criado, utilizamos o código de exemplo definido antes, com a chamada para o EJB, executamos o projeto principal (**ExemploEJB**), e efetuamos a chamada appropriada: <http://localhost:8080/ExemploSimplesJPA-war/ServletSoma>**

Estando tudo correto, teremos a saída apresentada na imagem a seguir:



Quanto ao EJB do tipo **MDB**, para criá-lo, devemos adicionar, no projeto **ExemploEJB-ejb**, um novo arquivo do tipo **Message Driven Bean**, na categoria **Enterprise Java Beans**.

Precisamos definir o nome (**Mensageiro001**) e o pacote do componente, além de escolher a fila para as mensagens no servidor (**jms/SimpleQueue**), como demonstrado na imagem seguinte.



Escolha de fila para as mensagens no servidor.

Todos os passos para a codificação de **Mensageiro001** foram descritos anteriormente, bem como a do Servlet para postagem da mensagem, com o nome **ServletMessage**. Mas lembre-se de que o Servlet deve ser criado no projeto **ExemploEJB-war**.

**Com todos os componentes implementados, basta implantar o sistema, a partir do projeto principal, e efetuar a chamada correta no navegador:**  
**<http://localhost:8080/ExemploSimplesJPA-war/ServletMessage>**

Após efetuar a chamada, será apresentada a página com a informação de que ocorreu o envio da mensagem, o que poderá ser verificado no console de saída do GlassFish, conforme descrito anteriormente.

**Falta pouco para atingir seus objetivos .**

## Vamos praticar alguns conceitos?

### Questão 1

Componentes do tipo Session Bean são voltados para o processamento de regras de negócio de forma síncrona e podem ser configurados para alguns modelos de gerência de estados. Quando precisamos manter dados de uma conexão específica, entre várias chamadas sucessivas, devemos trabalhar no modelo:

- A Stateless
- B Stateful
- C Singleton
- D Asynchronous
- E Synchronous

**Parabéns! A alternativa B está correta.**

Os três modelos para gerência de estados, nos Session Beans, podem ser descritos de forma resumida como: Sem estado (Stateless), com estado para a conexão (Stateful), e estado global compartilhado (Singleton).

### Questão 2

Um componente do tipo MDB (Message Driven Bean) é voltado para o processamento de regras de negócio de forma assíncrona, com base na recepção de mensagens a partir de ferramentas como JBoss MQ e MQ Series. A conexão com as mensagerias é feita com base na biblioteca:

A JTA

B JDBC

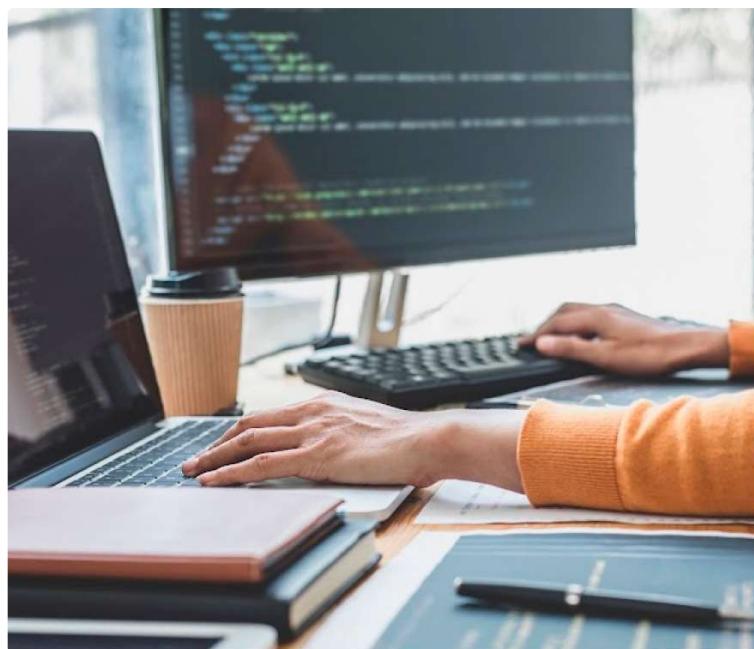
C JMS

D JAAS

E JTS

Parabéns! A alternativa C está correta.

A conexão entre o aplicativo e a mensageria deve ser feita através de um middleware adequado, classificado como MOM (Message Oriented Middleware), e na plataforma Java, temos o JMS (Java Message Service) como biblioteca para fornecimento de MOM.



### 3 - Arquitetura MVC no Java

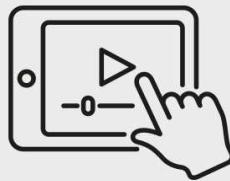
Ao final deste módulo, você será capaz de descrever a utilização da arquitetura MVC na plataforma Java.

# Padrões de desenvolvimento e arquitetURAIS

## Padrões de desenvolvimento

Conheça agora os principais aspectos dos padrões de desenvolvimento.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



A orientação a objetos representou um grande avanço na implementação de sistemas, pois aproximou a modelagem da codificação, mas o uso incorreto da metodologia levou à perda de diversas vantagens. Os **padrões de desenvolvimento**, que definem a formalização de soluções reutilizáveis, com nome, descrição da finalidade, modelagem UML e modo de utilização, permitiram recuperar as vantagens, baseando-se na adoção de soluções eficazes para a construção de sistemas orientados a objetos.

### Atenção!

Com os padrões, temos uma terminologia comum, pois eles permitem a utilização de um vocabulário básico e um ponto de vista semelhante sobre os problemas e possíveis soluções. Embora existam muitos padrões de desenvolvimento, alguns deles se destacam em um sistema cadastral corporativo, como: Facade, Proxy, Flyweight, Front Controller e DAO.

Quando abordamos a tecnologia **JPA**, ficou claro que ocorre uma divisão entre os dados da entidade e o mapeamento objeto-relacional. Isso porque, ao contrário dos atributos, as anotações não são serializáveis, impedindo a sua repercussão para outras camadas, o que é semelhante ao padrão DAO, no qual temos os comandos para acesso ao banco de dados agrupados em uma classe específica, separados da classe de entidade e de todo o restante do sistema.

Também vimos a utilização de diversos padrões de desenvolvimento na arquitetura de componentes do tipo **EJB**, como **Flyweight**, para o pool de

objetos, **Abstract Factory**, no fornecimento de interfaces de acesso locais ou remotas, **Service Locator**, necessário para a localização de recursos via **JNDI**, e **Proxy**, utilizado na comunicação remota.

## Comentário

Outros componentes do sistema também deverão seguir padrões de desenvolvimento, visando organizar as funcionalidades e facilitar a leitura do código-fonte, como o uso de: Facade, ao nível dos Session Beans, ou Front Controller, aplicado aos Servlets.

Conheça a descrição formal dos padrões de desenvolvimento citados, além de alguns outros, comuns em sistemas criados para o ambiente **JEE**. Vamos lá?

## Abstract Factory

Trata-se da definição de uma arquitetura abstrata para a geração de objetos, muito comum em frameworks.

## Command

Trata-se de encapsular o processamento da resposta para algum tipo de requisição, muito utilizado para o tratamento de solicitações feitas no protocolo HTTP.

## Data Access Object

Trata-se da utilização de classes específicas para concentrar as chamadas para o banco de dados.

## Facade

Trata-se de encapsular as chamadas para um sistema complexo, muito utilizado em ambientes corporativos.

## Flyweight

Trata-se da criação de grupos de objetos que respondem a uma grande quantidade de chamadas.

## Front Controller

Trata-se de concentrar as chamadas para o sistema, efetuando os direcionamentos corretos para cada chamada.

## Iterator

Trata-se do acesso sequencial aos objetos de uma coleção, o que é implementado nativamente no Java.

## Proxy

Trata-se da definição de um objeto para substituir a referência de outro, utilizado nos objetos remotos para deixar a conexão transparente para o programador.

## Service Locator

Trata-se de gerenciar a localização de recursos compartilhados, com base em serviços de nomes e diretórios.

## Singleton

Trata-se de garantir a existência de apenas uma instância para a classe, como em controles de acesso.

## Strategy

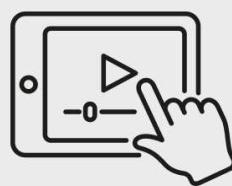
Trata-se da seleção de algoritmos em tempo de execução, com base em algum parâmetro fornecido.

A simbologia utilizada para representar os padrões de desenvolvimento é um **puzzle**, algo muito justo ao considerarmos que os sistemas atuais são implementados através da combinação de diversos padrões. Um exemplo seria a criação de um pool de processadores de resposta para solicitações de usuários remotos, o que poderia ser caracterizado por um **Flyweight** de **Strategies** para a escolha de **Commands**, além da utilização de **Proxy** na comunicação com os clientes.

## Padrões arquiteturais

Entenda um pouco mais sobre os padrões estruturais.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



A arquitetura de um sistema define a estrutura de alto nível do software, ou seja, formaliza a organização em termos de componentes e interações entre eles. Um dos objetivos é aproximar a visão de projeto da implementação do sistema, impedindo que ocorra a previsão de funcionalidades inviáveis para a fase de codificação.

**As especificações da arquitetura também definem as interfaces de entrada ou saída de informações, forma de comunicação com outros sistemas, e regras para o agrupamento dos componentes com base em suas áreas de atuação, entre outras características.**

Note que existem diversos perfis de sistemas, como aplicativos de linha de comando, ambientes baseados em janelas, ou até serviços sem interações diretas com o usuário, além da possibilidade de execução remota ou local, e cada perfil de execução traz algumas exigências estruturais.

Padrões arquiteturais definem as regras do sistema em termos estruturais, ou seja, uma visão macroscópica do programa e suas relações com o ambiente. Observe a comparação a seguir:

Modelo arquitetural	Padrão arquitetural
Define uma arquitetura de forma abstrata, com foco apenas no objetivo ou característica principal.	Define o perfil dos componentes estruturais, modelo de comunicação e os padrões de desenvolvimento mais adequados na implementação.

Por exemplo, o modelo de **Objetos Distribuídos** define apenas atributos essenciais para delimitar um ambiente com coleções de objetos respondendo a requisições remotas, enquanto o padrão arquitetural **Broker** define as regras de implementação, como o uso de **Proxy** na comunicação, ou a definição de pools de objetos no padrão **Flyweight**.

Existem diferentes definições de modelos para os padrões arquiteturais, e alguns deles satisfazem a mais de um modelo. Vejamos!

## Padrão Arquitetural

Modelo(s).

---

### Broker

Sistemas distribuídos.

---

### Camadas

Mud to Structure, Chamada e  
Retorno.

---

### Orientado a Objetos

Chamada e Retorno.

---

### Programa Principal e Sub- rotina

Chamada e Retorno.

---

### Pipes/Filters

Mud to Structure, Fluxo de  
Dados.

---

### Blackboard

Mud to Structure, Centrada em  
Dados.

---

## Lote

Fluxo de Dados.

---

## Repositório

Centrada em Dados.

---

## Processos Comunicantes

Componentes Independentes.

---

## Event-Driven

Componentes Independentes.

---

## Interpretador

Máquina Virtual.

---

## Baseado em Regras

Máquina Virtual.

---

## MVC

## PAC

Sistemas Interativos.

## Microkernel

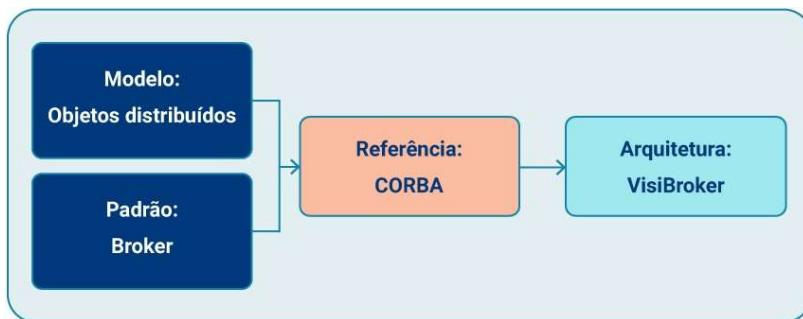
Sistemas Adaptáveis.

## Reflexiva

Sistemas Adaptáveis.

A **arquitetura de referência** serve para definir o mapeamento de um padrão arquitetural para componentes de software que sejam capazes de implementar as funcionalidades requeridas de forma cooperativa. Definida a referência, finalmente pode ser construída uma arquitetura, com todos os componentes codificados adequadamente.

Um exemplo de arquitetura é o **VisiBroker**, da fabricante Inprise, que utiliza como referência o **CORBA** (Common Object Request Broker Architecture), baseado no modelo de **objetos distribuídos**, segundo o padrão arquitetural **Broker**. Observe o esquema a seguir.



Modelo de arquitetura de referência.

Para utilizar uma arquitetura, é necessário compreender sua finalidade, de acordo com o padrão arquitetural adotado. Sistemas de baixa complexidade podem basear a arquitetura nos paradigmas adotados para a codificação, como a orientação a objetos, enquanto arquiteturas com maior complexidade seguem uma padronização mais robusta para

os elementos estruturais, como a forma de comunicação em rede, a gerência de pools de objetos, ou a utilização de processos assíncronos.

### Comentário

Enquanto para os mainframes era comum a arquitetura de processamento em lote, sistemas de linha de comando, no UNIX, utilizam amplamente o padrão de **pipes/filters**, no qual a saída obtida na execução de um programa serve como entrada para o próximo programa da sequência. Em ambos os casos, podemos observar arquiteturas baseadas no fluxo de dados.

Ambientes de execução remota, como RPC - Remote Procedure Call - e Web Services, são baseados em arquiteturas no padrão de **processos comunicantes**, onde servidores e clientes podem ser criados utilizando plataformas de desenvolvimento distintas, e o único fator de acoplamento é o protocolo de comunicação adotado.

É comum o uso de mensagerias nos sistemas corporativos, onde utilizamos o padrão arquitetural event-driven, baseado na ativação de processos de forma indireta, a partir de mensagens. O papel das mensagerias é tão importante que o componente adotado na comunicação é chamado de MOM (Message-Oriented Middleware), e as vantagens no uso desse padrão são o acoplamento quase nulo e o processamento assíncrono. Um sistema simples pode responder a um único padrão arquitetural, mas os sistemas corporativos são complexos e heterogêneos, sendo muito comum a adoção de múltiplos padrões combinados. De forma geral, sempre existe um padrão principal, que no caso dos sistemas cadastrais é o MVC.

## MVC



### Padrões de arquitetura e model-view-controller (MVC)

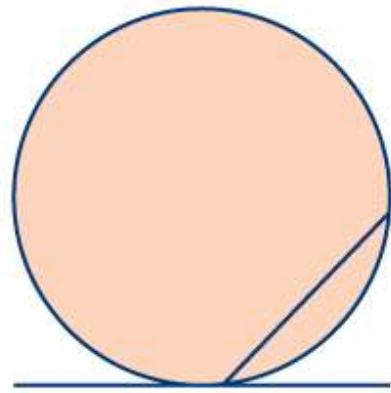
Saiba mais sobre a arquitetura MVC e as ferramentas do Java que se adequam a ela.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



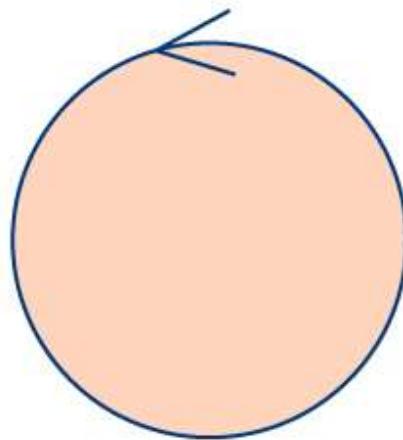
## Arquitetura MVC

A arquitetura MVC (Model-View-Controller) divide o sistema em três camadas, com responsabilidades específicas. Na camada **Model**, a mais baixa, temos as entidades e as classes para acesso ao banco de dados, na **Controller**, intermediária, concentramos os objetos de negócio, e na **View**, mais alta, são definidas as interfaces do sistema com o usuário ou com outros sistemas. Observe suas características a seguir:



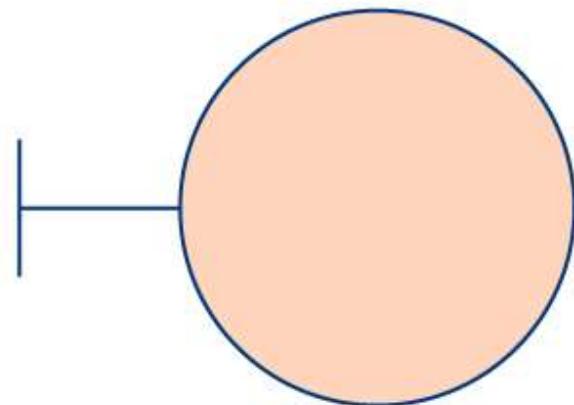
## Model (Modelo)

- Controla toda a persistência do sistema.
- Concentra as chamadas ao banco de dados.
- Encapsula o estado do sistema.
- Pode utilizar mapeamento objeto-relacional.
- Padrão DAO é aplicável.



## Controller (Controle)

- Implementa as regras de negócio do sistema.
- Solicita os dados à camada Model.
- Não pode ser direcionada para uma interface.
- Pode utilizar objetos distribuídos.
- Padrão Facade facilita a utilização da camada.



## View (Visualização)

- Define a interface do sistema.
- Faz requisições para a camada Controller.
- Contém apenas regras de formatação.
- Podem ser definidas múltiplas interfaces.
- Não pode acessar a camada Model.

Uma regra fundamental para a arquitetura MVC é a de que os elementos da camada View não podem acessar a camada Model. Somente os objetos de negócio da camada Controller podem acessar os componentes da Model, e os elementos da View devem fazer suas requisições exclusivamente para os objetos de negócio.

## Atenção!

A arquitetura MVC é baseada em camadas. Cada camada enxerga apenas a camada imediatamente abaixo.

Em uma arquitetura MVC, as entidades são as unidades de informação para trânsito entre as camadas, e todos os comandos SQL ficam concentrados nas classes do padrão **DAO**. Como apenas a camada Controller pode acessar a Model, e nela estão as classes DAO, nós garantimos que as interfaces não acessem o banco de dados diretamente.

O uso do padrão DAO e a popularização da arquitetura MVC demonstraram, de forma clara, a necessidade do mapeamento objeto-relacional em sistemas cadastrais criados com tecnologia orientada a objetos. Como as instruções SQL são bastante padronizadas, foi possível criar ferramentas para a geração dos comandos e preenchimento das entidades, de forma automática, bastando expressar a relação entre atributos da entidade e campos do registro.

A camada **Controller** precisa ser definida sem que seja voltada para algum ambiente específico, como interfaces SWING ou protocolo HTTP. A única dependência aceitável para os objetos de negócio deve ser com relação à camada **Model**, e como a gerência do uso dos componentes DAO ocorre a partir deles, uma das características observadas é a diminuição da complexidade nas atividades cadastrais que foram iniciadas na **View**, o que justifica dizer que temos a aplicação do padrão **Facade**.

## Comentário

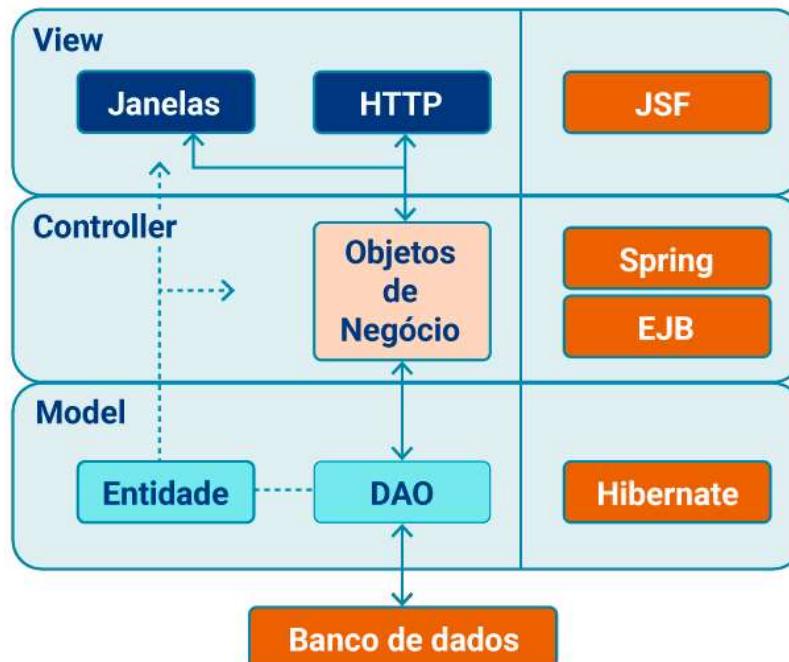
Uma observação importante é a de que, como as regras de negócio ficam concentradas na camada Controller, podemos gerenciar as transações eficientemente a partir dos componentes dela. Inclusive, com a adoção de objetos distribuídos, a funcionalidade das transações foi ampliada para um modelo com a participação de múltiplos servidores.

Da mesma forma, a camada Controller é o melhor local para definir as regras de autorização para o uso de funcionalidades do sistema, tendo

como base o perfil de um usuário autenticado. Com relação à autenticação, ela pode ser iniciada por uma tela de login na camada View, com a efetivação na camada Controller, e nos modelos atuais é comum a geração de um token, mantendo a independência entre as camadas.

## Componentes Java para MVC

Uma grande vantagem do MVC é o direcionamento do desenvolvedor e das ferramentas para as necessidades de cada camada. Com a divisão funcional, diversos frameworks foram criados, como o **JSF** (Java Server Faces), que define interfaces Web na camada View, **Spring** ou **EJB**, para implementar as regras de negócio da camada Controller, e **Hibernate**, para a persistência ao nível da Model. Além disso, o uso de camadas especializadas permite a divisão da equipe entre profissionais cujo perfil seja voltado para criação visual, negócios ou banco de dados.



Camadas MVC.

Os frameworks facilitam a manutenção e evolução do sistema, pois tendem a acompanhar as tecnologias que surgem ao longo do tempo, mas apenas empresas de grande porte e comunidades de código aberto são capazes de garantir as atualizações, não sendo raros os casos em que uma ferramenta menos conhecida é descontinuada.

### Dica

Utilizar as ferramentas oferecidas pelo fabricante da linguagem pode ser uma boa opção quando desejamos garantir a continuidade da evolução do sistema.

Em nosso contexto, a camada Model utiliza JPA, e como deve ser utilizada apenas pela camada Controller, é definida no mesmo projeto em que estão os componentes do tipo EJB. Note que a camada Controller oferece apenas as interfaces para os EJBs, com os dados sendo transitados na forma de entidades, sem acesso ao banco de dados, já que anotações não são serializáveis.

Confira as observações a seguir:

1

Com a abordagem adotada, definimos o núcleo funcional e lógico de nosso sistema, sem a preocupação de satisfazer a qualquer tipo de tecnologia para construção de interfaces de usuário.

2

A independência do núcleo garante que ele possa ser utilizado por diversas interfaces simultâneas, como SWING, HTTP ou Web Services, sem que ocorra qualquer modificação nos componentes do tipo JPA ou EJB.

3

Um erro comum nos sistemas Java para Web é definir os controladores no formato de Servlets, pois as regras de negócio se confundem com as rotinas de conversão utilizadas entre o protocolo HTTP e as estruturas da linguagem Java.

4

A abordagem errônea faz com que qualquer nova interface, como SWING, Web Services, ou até linha de comando, seja obrigada a solicitar os serviços através do protocolo HTTP, algo que não é uma exigência das regras de negócio dos sistemas, de forma geral.

Considere que a entidade **Produto**, definida anteriormente, com uso de tecnologia JPA, seja criada no projeto **ExemploEJB-ejb**, onde codificamos nosso Session Bean de teste, com o nome **Calculadora**. Com a presença da entidade no projeto, podemos adicionar outro

Session Bean do tipo **Stateless**, com o nome **ProdutoGestor** e uso de interface **Local**, para as operações cadastrais. Observe o exemplo a seguir.

Java



```
1  @Local
2  public interface ProdutoGestorLocal {
3      List< Produto > obterTodos();
4      void incluir(Produto p);
5  }
6
7  @Stateless
8  public class ProdutoGestor implements ProdutoGest
9  @Override
10 public List obterTodos() {
11     EntityManagerFactory emf = Persistence.
12         createEntityManagerFactory("ExemploSimplesJ
13     EntityManager em = emf.createEntityManager();
14     Query query = em.createNamedQuery("Produto find
```

Como podemos observar, nossos códigos de exemplos anteriores foram aproveitados aqui, com leves adaptações. Não há nada de novo no que se refere ao uso de JPA, ocorrendo apenas uma reorganização em termos das classes e anotações, para que as operações sejam disponibilizadas através do Session Bean, conforme descrito a seguir.

Precisamos adicionar o Com nossas arquivo **persistence.xml**, camadas **Model** e **Controller** completamente definido em nosso exemplo de JPA, ao diretório **conf** do projeto **ExemploEJB-ejb**, sem modificações, o que levará à utilização de controle transacional de forma local.

codificadas, podemos definir um **Servlet**, no projeto **ExemploEJB-war**, com o nome **ServletListaProduto**, o qual será parte da camada **View** do sistema, no modelo **Web**.

O objetivo do novo componente será a exibição da listagem dos produtos presentes na base de dados. Observe o exemplo a seguir.

Java

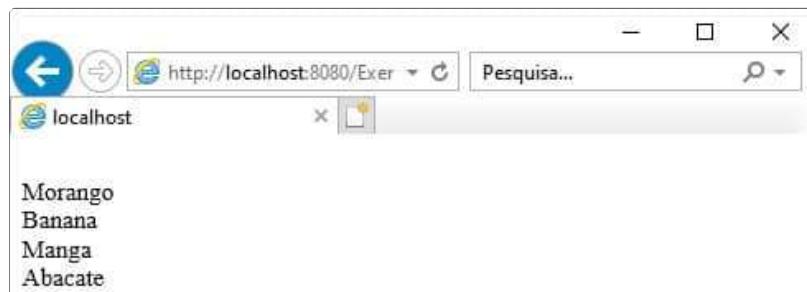


```
1  @WebServlet(name = "ServletListaProduto", urlPatt►
2  public class ServletListaProduto extends HttpServlet {
3      @EJB
4      ProdutoGestorLocal facade;
5
6      protected void doGet(HttpServletRequest request
7                          throws ServletException,
8                          response.setContentType("text/html; charset=UTF-8);
9      try (PrintWriter out = response.getWriter())
10         out.println("< html >< body >");
11         facade.obterTodos().forEach(p -> { out.prin
12         out.println("");
13     }
14 }
```

No código temos o atributo **facade**, do tipo **ProdutoGestorLocal**, utilizando a anotação **EJB** para injetar o acesso ao pool de Session Beans.

**Após configurar o acesso, invocamos o método `obterTodos`, na construção da resposta ao HTTP no modo GET, aceitando uma chamada como:**  
**`http://localhost:8080/ExemploEJB-war/ServletListaProduto`**

Estando tudo correto, teremos uma saída similar à apresentada na imagem seguinte, na tela do navegador.



Exibição da listagem dos produtos na base de dados.

**Falta pouco para atingir seus objetivos.**

## Vamos praticar alguns conceitos?

### Questão 1

Os programas que executam sobre sistemas operacionais baseados no UNIX utilizam amplamente um padrão arquitetural específico, no qual efetuamos o direcionamento do fluxo de saída do primeiro programa para o fluxo de entrada do segundo. O texto faz referência a qual padrão arquitetural?

A Broker

B MVC

C PAC

D Batch

E Pipes/Filters

Parabéns! A alternativa E está correta.

Um exemplo de comando comum, no UNIX, seria "ls -s | more", onde temos a listagem de um diretório, obtida no primeiro comando, sendo direcionada para um comando que efetua a paginação do resultado. Os comandos representam os Filters (filtros), que são concatenados pelo símbolo de Pipe.

### Questão 2

Em uma arquitetura MVC, utilizando componentes Java, é comum adotar ferramentas de mapeamento objeto-relacional, como Hibernate ou JPA, onde a persistência pode ocorrer em transições gerenciadas via JTA, ou de forma programática. Complete as

Iacunas do código-fonte, no modelo programático, com os termos corretos:

JPA



```
1 body {  
2     _____ emf = Persistence.createEntityManager  
3     _____ em = emf.createEntityManager();  
4     em.getTransaction().begin();  
5     c = new Curso(3, "EE");  
6     em._____ (c);  
7     em.getTransaction().commit();
```



A EntityManagerFactory; EntityManager; persist

B EntityFactory; EntityManager; save

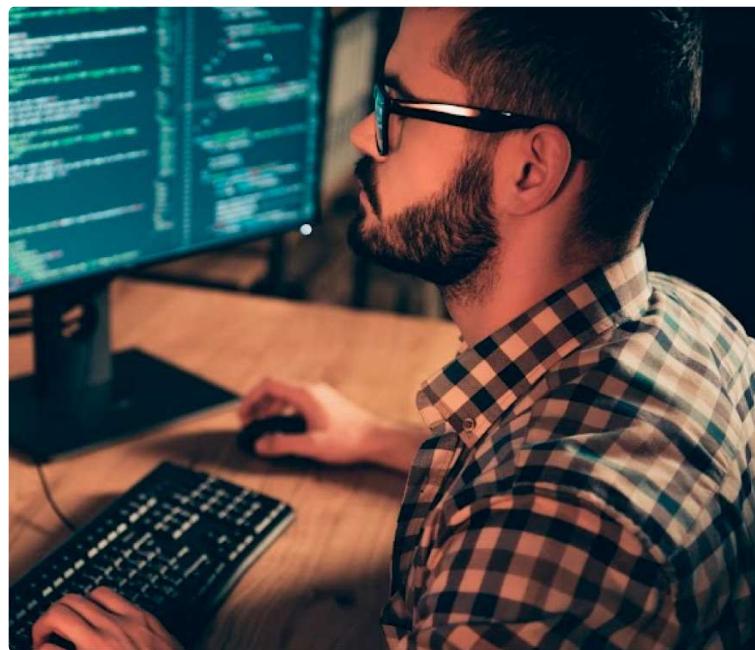
C Connection; Statement; execute

D Database; Entity; put

E Connection; EntityManager; execute

Parabéns! A alternativa A está correta.

Um elemento essencial no JPA é a classe EntityManager, responsável pelo manuseio das diversas entidades. Objetos desta classe são gerados a partir do EntityManagerFactory, e o método persist viabiliza a gravação de uma entidade no banco de dados.



## 4 - Padrão Front Controller em MVC/Java Web

Ao final deste módulo, você será capaz de empregar padrão Front Controller em sistema MVC, com interface Java Web.

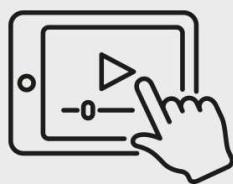
### Visão geral



## Uso de tecnologias Java com padrão MVC para Web

Saiba mais sobre arquitetura MVC com Front Controller no ambiente Web e o uso da plataforma Java.

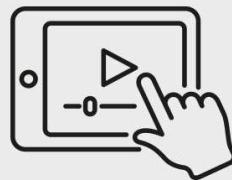
Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



# Padrões Front Controller

Conheça agora alguns importantes aspectos do padrão Front Controller.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



O objetivo do padrão **Front Controller** é a concentração das chamadas efetuadas em um único ponto de acesso, centralizando a orquestração dos serviços oferecidos a partir da camada **Controller**, e direcionando os resultados para a interface correta. A implementação do padrão **Front Controller** deve ocorrer ao nível da camada **View**, pois lida apenas com a conversão de formatos, o fluxo de chamadas e os redirecionamentos, sem interferir com regras de negócio.

Criaremos um projeto do tipo **Enterprise Application**, na categoria **Java Enterprise**, com o nome **CadastroEJB**, configurando para utilização do servidor **GlassFish** e do **Java EE 7**. Antes de iniciar a codificação, precisamos entender o modelo funcional do sistema, e criar as tabelas necessárias, conforme o SQL seguinte.

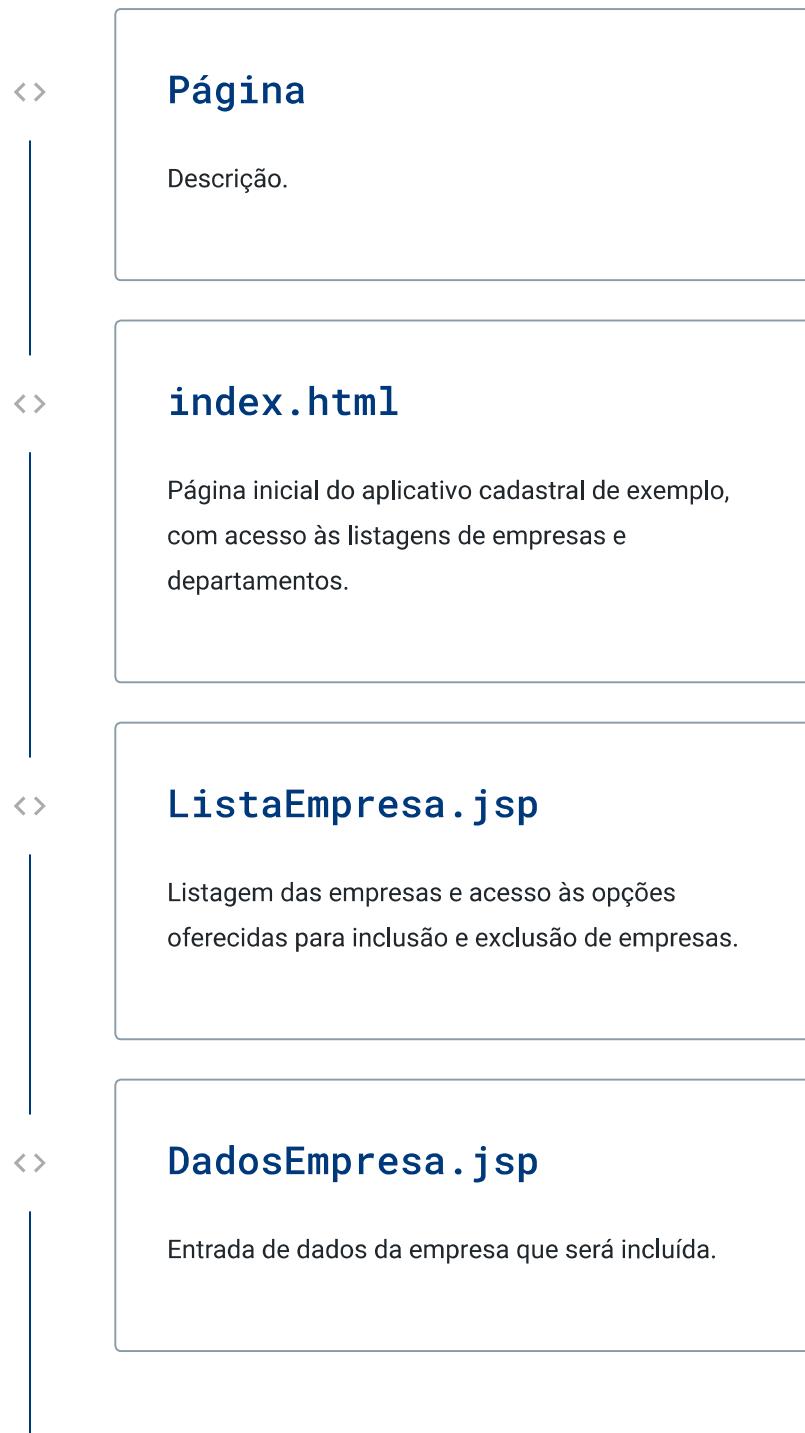
SQL

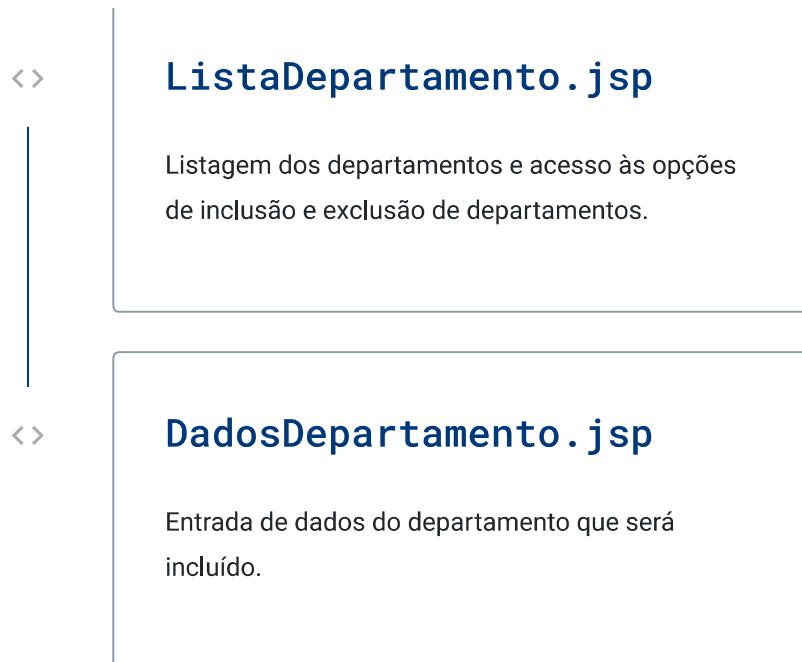


```
1 CREATE TABLE EMPRESA (
2     CODIGO INT NOT NULL PRIMARY KEY,
3     RAZAO_SOCIAL VARCHAR(50));
4
5 CREATE TABLE DEPARTAMENTO (
6     CODIGO INT NOT NULL PRIMARY KEY,
7     NOME VARCHAR(50),
8     COD_EMPRESA INT NOT NULL);
9
10 ALTER TABLE DEPARTAMENTO ADD FOREIGN KEY(COD_EMPR
11             REFERENCES EMPRESA(CODIGO);
12
13 CREATE TABLE SERIAIS (
14     NOME_TABELA VARCHAR(50) NOT NULL PRIMARY KEY,
```

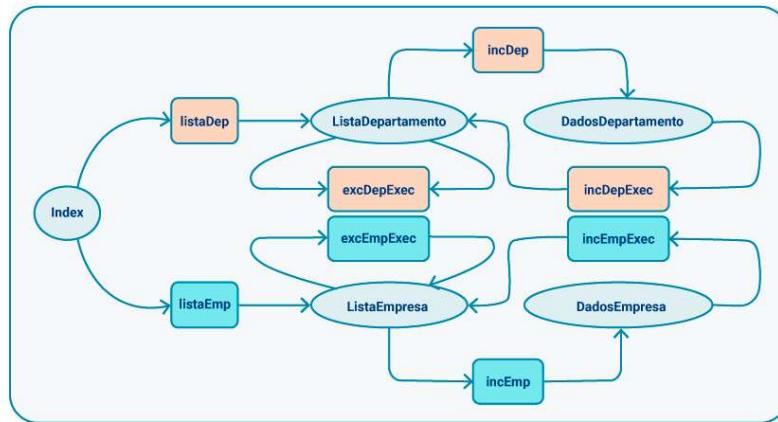
Aqui, temos as tabelas EMPRESA e DEPARTAMENTO, para a persistência de dados de um cadastro simples, com um relacionamento através do campo COD\_EMPRESA, da tabela DEPARTAMENTO. Também podemos observar uma terceira tabela, com o nome SERIAIS, que viabilizará o autoincremento, através de anotações do JPA.

Para a interface de nosso sistema, utilizaremos cinco páginas, as quais são descritas a seguir:





Agora, vamos definir os fluxos das chamadas, que ocorrerão a partir do HTTP, o que pode ser representado através de uma **Rede de Petri**, um ferramental que permite expressar os **estados** de um sistema, físico ou virtual, e definir as **transições** que ocorrem, efetuando a ligação entre estados e transições através de arcos. Ao modelar um sistema Web, os estados definem **páginas** e as transições são chamadas HTTP.



## Rede de Petri: fluxos das chamadas.

Em nossa Rede de Petri, temos as páginas dentro de figuras **elípticas**, que representam os **estados** do sistema, e as transições expressas através de figuras **retangulares**. A alternância entre páginas, ou estados, sempre ocorrerá com a passagem por uma transição, através de um **Front Controller**.

Como as chamadas HTTP utilizam parâmetros, com valores do tipo texto, vamos adotar um parâmetro para identificação da transição requerida, onde teremos o nome **acao** e o valor correspondente a um dos valores de nosso diagrama. Podemos observar, a seguir, os valores utilizados e operações que devem ser realizadas.

&gt;

### listaDep

- Obter a lista de departamentos.
- Direcionar o fluxo para **ListaDepartamento.jsp**.

&gt;

### listaEmp

- Obter a lista de empresas.
- Direcionar o fluxo para **ListaEmpresa.jsp**.

&gt;

### excDepExec

- Remover o departamento, de acordo com o código informado.
- Obter a lista de departamentos.
- Direcionar a informação para **ListaDepartamento.jsp**.

&gt;

### excEmpExec

- Remover a empresa, de acordo com o código informado.
- Obter a lista de empresas.
- Direcionar o fluxo para **ListaEmpresa.jsp**.



### incDep

- Direcionar o fluxo para **DadosDepartamento.jsp**.



### incDepExec

- Receber os dados para inclusão do departamento.
- Converter para o formato de entidade.
- Incluir o departamento na base de dados.
- Obter a lista de departamentos.
- Direcionar o fluxo para **ListaDepartamento.jsp**.



### incEmp

- Direcionar o fluxo para **DadosEmpresa.jsp**.



### incEmpExec

- Receber os dados para inclusão da empresa.
- Converter para o formato de entidade.
- Incluir a empresa na base de dados.
- Obter a lista de empresas.
- Direcionar o fluxo para **ListaEmpresa.jsp**.

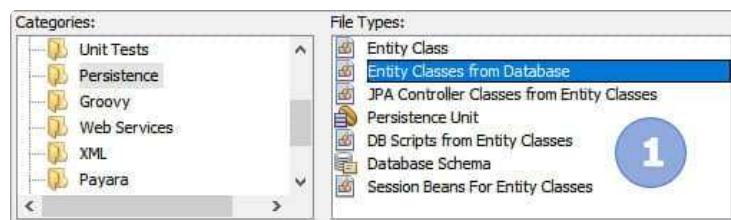
Nos aplicativos Java para Web, o padrão **Front Controller** pode ser implementado com base em um **Servlet**. O processo envolve a recepção de uma chamada HTTP, através dos métodos **doGet** ou **doPost**, execução de operações que envolvam chamadas aos **EJBs**, relacionadas às atividades de consulta ou persistência, e redirecionamento, ao final, para uma página, normalmente do tipo **JSP**, para a construção da resposta.

# Camadas Model, Controller e View

## Camadas Model e Controller

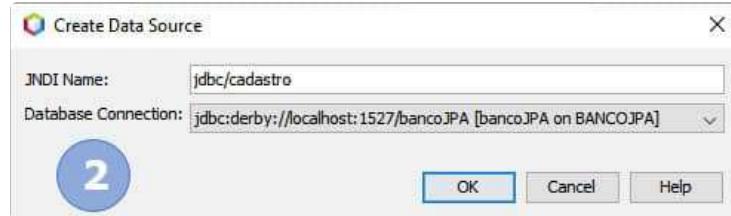
Nossas camadas **Model** e **Controller** serão criadas no projeto **CadastroEJB-ejb**, através das tecnologias JPA e EJB, iniciando com a codificação da camada **Model**, baseada na tecnologia **JPA**.

Utilizaremos as ferramentas de automação do NetBeans, e para gerar as entidades do sistema, executaremos os passos apresentados a seguir:



### Passo 1

Adicionar novo arquivo, escolhendo **Entity Classes from Database**, na categoria **Persistence**.



### Passo 2

Na configuração de **Data Source**, escolher **New Data Source**, com a definição do nome JNDI (**jdbc/cadastro**), e escolher a conexão para o banco de dados.



## Passo 3

Escolher as tabelas **DEPARTAMENTO** e **EMPRESA**, deixando marcada a opção de inclusão das tabelas relacionadas.



## Passo 4

Na tela seguinte, definir o nome do pacote como **model**, deixando marcada apenas a opção de criação da unidade de persistência.



## Passo 5

Escolher, ao chegar na última tela, o tipo de coleção como **List**, além de desmarcar todas as opções.

Teremos a geração das entidades **Departamento** e **Empresa**, no pacote **model**, embora com possíveis erros de compilação, que serão resolvidos com o acréscimo da biblioteca **Java EE 7 API** ao projeto **CadastroEJB-ejb**. As entidades geradas irão precisar apenas de uma pequena modificação, para configurar o uso de **autoincremento**, veja a seguir.

Java

```

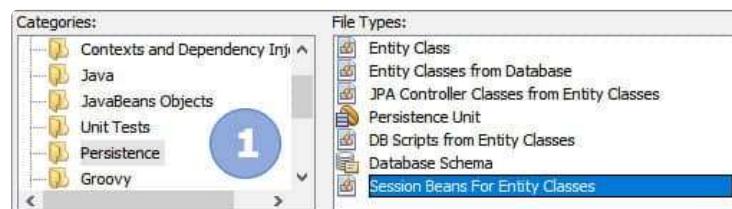
1  @Entity
2  @Table(name = "DEPARTAMENTO")
3  @NamedQueries({
4      @NamedQuery(name = "Departamento.findAll",
5          query = "SELECT d FROM Departamento d"))
6  public class Departamento implements Serializable
7  private static final long serialVersionUID = 1L
8  @Id
9  @TableGenerator(name = "DeptoTabGen", table = "
10         pkColumnName = "NOME_TABELA",
11         pkColumnValue = "DEPARTAMENTO",
12         valueColumnName = "VALOR_CHAVE")
13  @GeneratedValue(strategy = GenerationType.TABLE
14         generator = "DeptoTabGen")

```

A anotação **TableGenerator** define um gerador sequencial baseado em tabelas, sendo necessário definir a tabela que armazena os valores (**SERIAIS**), o campo da tabela que individualiza o gerador (**NOME\_TABELA**), e o valor utilizado para o campo, sendo aqui adotado o próprio nome da tabela para cada entidade, além do campo que irá guardar o valor corrente para o sequencial (**VALOR\_CHAVE**). Após definir o gerador, temos a aplicação do valor ao campo da entidade, com base na anotação **GeneratedValue**, tendo como estratégia o tipo **GenerationType.TABLE**, e a relação com o gerador a partir do **nome** escolhido.

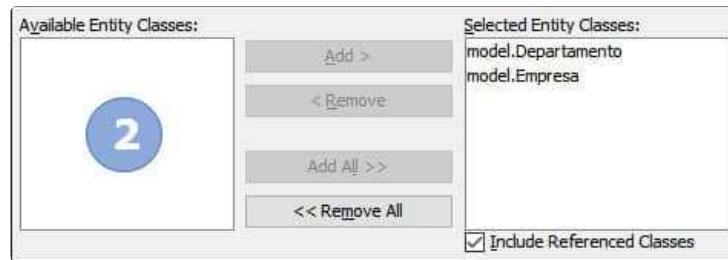
Durante a execução, quando uma entidade é **criada** e persistida no banco de dados, o **gerador** é acionado, incrementa o valor do campo **VALOR\_CHAVE** para a linha da tabela **SERIAIS** referente à entidade, e alimenta a **chave primária** com o novo valor. Existem outras estratégias de geração, como o uso de **SEQUENCE**, mas a adoção de uma tabela de identificadores deixa o sistema muito mais portável.

Com a camada **Model** pronta, vamos criar a camada **Controller**, usando componentes do tipo **EJB**, de acordo com os passos apresentados a seguir.



## Passo 1

Adicionar arquivo, escolhendo **Session Beans For Entity**  
**Classes**, na categoria **Persistence**.



## Passo 2

Selecionar todas as entidades do projeto.



## Passo 3

Definir o nome do pacote (**control**), além de adotar a interface **Local**.

Será gerada uma classe abstrata, com o nome **AbstractFacade**, que concentra todos os processos de acesso e manipulação de dados, com o uso de **elementos genéricos**, observe a seguir.

Java



```

1 public abstract class AbstractFacade {
2     private Class entityClass;
3     public AbstractFacade(Class entityClass) {
4         this.entityClass = entityClass;
5     }
6     protected abstract EntityManager getEntityManager();
7     public void create(T entity) {
8         getEntityManager().persist(entity);
9     }
10    public void edit(T entity) {
11        getEntityManager().merge(entity);
12    }

```

```
13     public void remove(T entity) {  
14 }
```

Observe que a classe é iniciada com um construtor, onde é recebida a classe da entidade gerenciada, e um método abstrato para retornar uma instância de **EntityManager**, ao nível dos descendentes, elementos utilizados pelos demais métodos da classe.

Os métodos **create**, **edit** e **remove**, voltados para as ações de inclusão, edição e exclusão da entidade, são implementados através da invocação dos métodos de **EntityManager**, semelhante aos nossos exemplos anteriores, com tecnologia JPA, bem como **find**, que retorna uma entidade a partir de sua chave primária.

**Quanto aos métodos findAll e findRange, eles utilizam o JPA no modo programado, com base em CriteriaQuery, que apresenta métodos para substituir o uso de comandos na sintaxe JPQL. A chamada ao método from, tendo como parâmetro a classe da entidade, combinado com o método select, permite efetuar uma consulta que retorna todas as entidades do tipo gerenciado a partir do banco de dados, mas em findRange temos ainda o recurso de paginação, sendo definidos o índice inicial (setFirstResult) e a quantidade de entidades (setMaxResults).**

No método **count**, que obtém a quantidade total de entidades, também temos a adoção de **CriteriaQuery**, agora de uma forma mais complexa, com a definição de um elemento **Root**, envolvendo o conjunto completo de entidades, e aplicação do operador **count** sobre o conjunto.

Com a definição do modelo funcional genérico, a construção dos **Session Beans** se torna muito simples, com base na herança e uso de anotações. Veja o exemplo a seguir:

Java

```
1  @Stateless  
2  public class DepartamentoFacade extends  
3      AbstractFacade< Departamento > implements  
4          DepartamentoFacadeLocal {  
5      @PersistenceContext(unitName = "CadastroEJB-ejb  
6      private EntityManager em;  
7  
8      @Override  
9      protected EntityManager getEntityManager() {  
10          return em;  
11      }
```

```
12     public DepartamentoFacade() {  
13         super(Departamento.class);  
14     }
```

Em todos os três EJBs temos o mesmo tipo de programação, onde ocorre a herança com base em **AbstractFacade**, passando o tipo da entidade. O método **getEntityManager** retorna o atributo **em**, e no construtor a superclasse é chamada, com a passagem da classe da entidade.

Os EJBs seguem o padrão **Facade**, e enquanto a anotação **Stateless** configura a classe para se tornar um **Stateless Session Bean**, o uso da anotação **PersistenceContext**, com a definição da unidade de persistência, instancia um **EntityManager** no atributo **em**.

Ainda são necessárias as interfaces de acesso ao pool de EJBs, o que deve ser feito **sem** o uso de elementos genéricos, como pode ser visto a seguir.

Java



```
1  @Local  
2  public interface DepartamentoFacadeLocal {  
3      void create(Departamento departamento);  
4      void edit(Departamento departamento);  
5      void remove(Departamento departamento);  
6      Departamento find(Object id);  
7      List findAll();  
8      List findRange(int[] range);  
9      int count();  
10 }  
11  
12 @Local  
13 public interface EmpresaFacadeLocal {  
14     void create(Empresa empresa);
```

Note que as interfaces apresentam métodos equivalentes aos que foram definidos em **AbstractFacade**, mas com a especificação da entidade, o que faz com que a herança ocorrida nos **Session Beans** implemente, de forma natural, as interfaces relacionadas.

## Comentário

Não foram utilizadas instruções para o controle transacional, o que decorre do fato de que o **contêiner EJB** será o responsável por gerenciar as transações, via **JTA**, dentro do modelo conhecido como **CMP** (Container Managed Persistence).

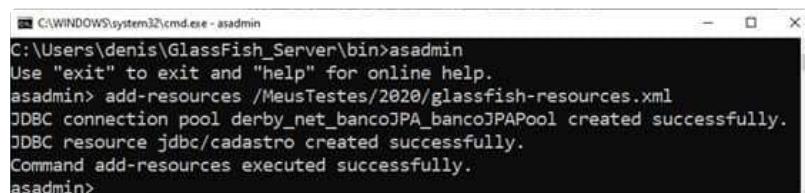
Para que o controle transacional ocorra da forma indicada, temos o atributo **transaction-type** com valor **JTA**, no arquivo **persistence.xml**, mas, há uma discrepância entre o **NetBeans** e o **GlassFish**, em termos da convenção de nomes, invalidando o identificador **JNDI**.

Vamos alterar o atributo **jndi-name**, no arquivo **glassfish-resources.xml**, bem como o elemento **jta-data-source**, no arquivo **persistence.xml**, adotando o valor **jdbc/cadastro** para ambos. A modificação é necessária pelo fato do servidor GlassFish não aceitar o uso do sinal de **dois pontos** no identificador **JNDI**. Observe o exemplo a seguir.

XML

```
1 < ?xml version="1.0" encoding="UTF-8"? >
2 < !DOCTYPE resources ... >
3 < resources >
4   < jdbc-connection-pool ... >
5     < property name="serverName" value="localhost"
6       < property name="portNumber" value="1527"/ >
7       < property name="databaseName" value="bancoJP
8         < property name="User" value="bancoJPA"/ >
9         < property name="Password" value="bancoJPA"/
10        < property name="URL"
11          value="jdbc:derby://localhost:1527/bancoJ
12          < property name="driverClass"
13            value="org.apache.derby.jdbc.ClientDriver
14        < /jdbc-connection-pool >
```

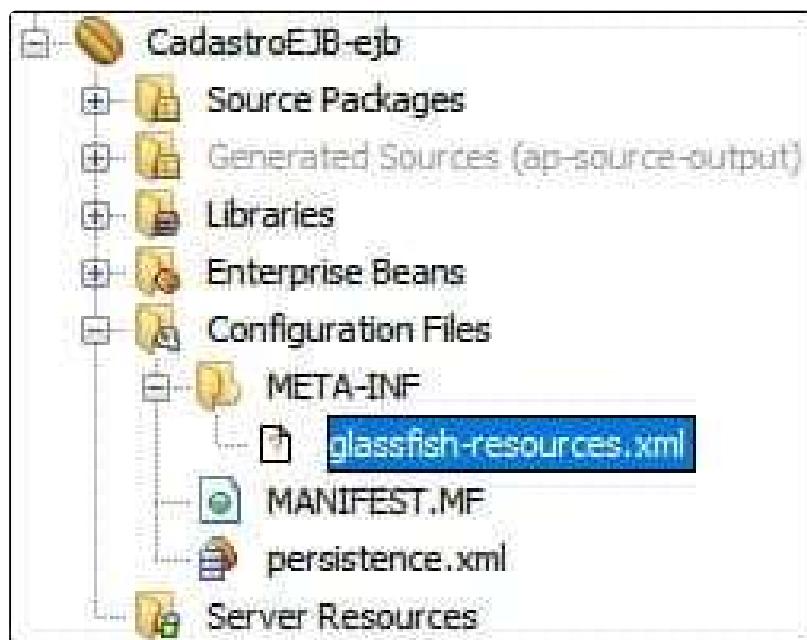
Caso ocorra erro na implantação, mesmo após alterar os arquivos, execute o programa **asadmin**, invocando, em seguida, o comando **add-resources**, com a passagem do nome completo do arquivo **glassfish-resources.xml**, conforme demonstrado na imagem a seguir.



```
C:\WINDOWS\system32\cmd.exe - asadmin
C:\Users\denis\GlassFish_Server\bin>asadmin
Use "exit" to exit and "help" for online help.
asadmin> add-resources /MeusTestes/2020/glassfish-resources.xml
JDBC connection pool derby_net_bancoJPA_bancoJPAPool created successfully.
JDBC resource jdbc/cadastro created successfully.
Command add-resources executed successfully.
asadmin>
```

Programa **asadmin**.

O arquivo **glassfish-resources.xml** fica disponível na divisão de configurações do projeto **CadastroEJB-ejb**, e foi gerado quando criamos as entidades a partir do banco de dados, como pode ser visto na próxima imagem.

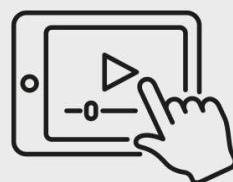


Divisão de configurações do projeto CadastroEJB-ejb.

## Camada View

Conheça importantes características da camada View.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



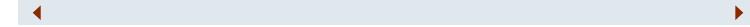
A construção da camada **View** ocorrerá no projeto **CadastroEJB-war**, e será iniciada com a geração das páginas **JSP** consideradas na Rede de Petri do sistema, começando pela página **DadosEmpresa.jsp**, que não apresenta conteúdo dinâmico, como você pode ver a seguir.

HTML



```
1 <%@page contentType="text/html" pageEncoding="UTF-8
2 <html><body>
3   <form action="CadastroFC" method="post">
4     <input type="hidden" name="acao" value="incEmpE
5     Razão Social: <input type="text" name="razao_so
6     <input type="submit" value="Cadastrar"/>
7   </form>
```

```
8    < /body >< /html >
```



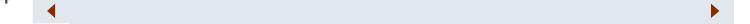
Observando o código-fonte, temos apenas um formulário comum, com um parâmetro do tipo **hidden** guardando o valor de **acao**, no caso **incEmpExec**, e um campo de texto para a razão social da empresa. Os dados serão enviados para **CadastroFC**, um Servlet no padrão Front Controller que iremos criar posteriormente.

O cadastro de um departamento será mais complexo, pois envolverá a escolha de uma das empresas do banco de dados. Vamos adicionar o arquivo **DadosDepartamento.jsp**, com o conteúdo apresentado a seguir.

JSP



```
1  < %@page import="model.Empresa"% >
2  < %@page import="java.util.List"% >
3  < %@page contentType="text/html" pageEncoding="UT
4  < html >
5    < body >
6      < form action="CadastroFC" method="post" >
7        < input type="hidden" name="acao" value="in
8          Empresa:
9            < select name="cod_empresa" >
10         <%
11           List< Empresa > lista = (List< Empresa >)
12             request.getAttribute("listaEmp");
13           for(Empresa e: lista){
14             %>
```



A coleção de empresas será recuperada a partir de um atributo de requisição, com o nome **listaEmp**, através do método **getAttribute**, e com base na coleção, preenchemos as opções de uma lista de seleção para o parâmetro **cod\_empresa**. Os outros campos são apenas um parâmetro do tipo **hidden**, definindo o valor de **acao** como **incDepExec**, e um campo de texto para o nome do departamento.

Agora, acompanhe a seguir, a criação do primeiro arquivo para listagem, com o nome **ListaEmpresa.jsp**.

JSP



```
1 < %@page import="model.Empresa"%>
2 < %@page contentType="text/html" pageEncoding="UT
3 < html>
4 < body>
5   < a href="CadastroFC?acao=incEmp">Nova Empres
6     < table border="1" width="100%">
7       < tr>< td>Código</td>< td>Razão Social</td>
8       <%
9         List< Empresa> lista = (List< Empresa>)
10            request.getAttribute("lista");
11            for(Empresa e: lista){
12              %
13            %
14          < tr>< td>%e.getCodigo()%</td>< /td>
```

A página é iniciada com a definição de um **link** para **CadastroFC**, com o parâmetro **acao** contendo o valor **incEmp**. Em seguida, é definida uma **tabela** para exibir os dados de cada empresa do banco de dados, contendo os títulos **Código**, **Razão Social** e **Opções**.

Recuperamos a coleção de empresas, através de um atributo de requisição com o nome **lista**, e preenchemos as células de cada linha da tabela a partir dos dados da entidade, além de um link para exclusão montado dinamicamente. A exclusão será efetuada com a chamada para **CadastroFC**, tendo o valor **excEmpExec** no parâmetro **acao**, e o código da empresa corrente no parâmetro **cod**.

Finalmente, temos a listagem de departamentos, no arquivo **ListaDepartamento.jsp**, veja a seguir.

JSP



```
1 < %@page import="model.Departamento"%>
2 < %@page import="java.util.List"%>
3 < %@page contentType="text/html" pageEncoding="UT
4 < html>
5   < body>
6     < a href="CadastroFC?acao=incDep">Novo Depart
7       < table border="1" width="100%">
8         < tr>< td>Código</td>< td>Nome</td>< td>Emp
9           < td>Opções</td>< /tr>
10          <%
11            List< Departamento > lista = (List< Departame
12               request.getAttribute("lista");
13               for(Departamento d: lista){
```

Em termos práticos, a listagem de departamentos é muito semelhante à de empresas, com a definição de um link de inclusão, agora com o valor **incDep** para **acao**, e a exibição dos dados através de uma tabela com os títulos **Código, Nome, Empresa e Opções**.

Agora temos uma coleção de departamentos para o preenchimento das células, e o link para exclusão faz a chamada para **CadastroFC**, tendo o valor **excDepExec** no parâmetro **acao**, e o código do departamento corrente no parâmetro **cod**. Observe como a razão social da empresa é recuperada facilmente, utilizando o atributo presente na entidade departamento, alimentado através de uma anotação **ManyToOne**.

Com a modificação de **index.html**, terminamos a construção das interfaces de usuário, observe a seguir.

HTML



```
1 < html>
2   < body>
3     < a href="CadastroFC?acao=listDep">
4       Listagem de Departamentos</a>
5     < a href="CadastroFC?acao=listEmp">
6       Listagem de Empresas</a>
7   </ body>
8 </ html>
```

Podemos observar que os links utilizados fazem referência a CadastroFC, com os valores de acao para obtenção de listagens para as entidades, que no caso são listaDep, para departamentos, e listaEmp, para empresas.

# Construção do Front Controller

## Implementação do Front Controller

Com as interfaces concluídas, devemos iniciar a construção do **Front Controller**, levando à conclusão da camada **View**. Utilizaremos também um padrão **Strategy**, com o objetivo de segmentar as chamadas aos EJBs, e todas as classes serão criadas no pacote **view**, do projeto **ExemploEJB-war**, como você pode ver a seguir.

Java



```
1 public abstract class Strategy {  
2     protected final K facade;  
3     public Strategy(K facade) {  
4         this.facade = facade;  
5     }  
6     public abstract String executar(String acao,  
7             HttpServletRequest request);  
8 }
```

Com base em uma classe **abstrata**, e uso de elemento genérico, definimos um padrão **Strategy**, onde a execução ocorrerá a partir do valor para **acao** e da **requisição HTTP**, além de um construtor recebendo a **interface** para o EJB, através do elemento genérico.

Definimos, em seguida, a estratégia para **empresas**, herdando da classe **Strategy**, com a adoção da interface **EmpresaFacadeLocal**, observe a seguir.

Java



```
1 public class EmpresaStrategy  
2     extends Strategy< EmpresaFacadeLocal > {  
3     public EmpresaStrategy(EmpresaFacadeLocal facade) {  
4         super(facade);  
5     }  
6     @Override  
7     public String executar(String acao,  
8             HttpServletRequest request) {
```

```

9     String paginaDestino = "ListaEmpresa.jsp";
10    switch(acao){
11        case "listaEmp":
12            request.setAttribute("lista", facade.find
13            break;

```

Durante a execução, temos a definição da página de destino como **ListaEmpresa.jsp**, a qual será diferente apenas quando o parâmetro **acao** tiver valor **incEmp**, onde temos, como destino, a página **DadosEmpresa.jsp**. O método **executar** irá retornar o nome da página para o **Front Controller**, e ele efetuará o redirecionamento.

**Além da definição da página correta, temos as operações efetuadas a partir do atributo **facade**, como a remoção da entidade, quando **acao** vale **excEmpExec**, ou a inclusão, para o valor **incEmpExec**. Para todas as ações que direcionam para a página de listagem, temos ainda a definição do atributo de requisição com o nome **lista**, contendo a coleção de empresas, obtidas a partir do método **findAll** do **facade**, o que permite que ocorra sua recuperação posterior, na página JSP.**

Agora podemos definir a classe **DepartamentoStrategy**, com **facade** baseado em uma interface do tipo **DepartamentoFacadeLocal**, e acessando um segundo EJB, através do atributo **facadeEmpresa**, do tipo **EmpresaFacadeLocal**. Note, no exemplo a seguir, que o construtor deve ser modificado para receber as duas interfaces utilizadas.

Java



```

1  public class DepartamentoStrategy
2      extends Strategy< DepartamentoFacadeLocal>{
3          private final EmpresaFacadeLocal empresaFacade;
4          public DepartamentoStrategy(DepartamentoFacadeL
5              EmpresaFacadeLocal empresaFacade) {
6                  super(facade);
7                  this.empresaFacade = empresaFacade;
8              }
9          @Override
10         public String executar(String acao,
11             HttpServletRequest request) {
12             String paginaDestino = "ListaDepartamento.jsp
13             switch(acao){
14                 case "listaDep".

```

A estratégia para departamentos é um pouco mais complexa, o que exigiu a utilização de duas interfaces para EJBs. Durante a execução, teremos a página de destino definida como **ListagemDepartamento.jsp**, a não ser para **acao** com valor **incDep**, onde a página de destino utilizada será **DadosDepartamento.jsp**.

Observe os passos a seguir:

---

## Passo 1

Da mesma forma que na estratégia de empresa, temos as operações efetuadas a partir do atributo **facade**, como a remoção da entidade, quando **acao** tem valor **excDepExec**, ou a inclusão, para o valor **incDepExec**, e a chamada para **findAll**, para o preenchimento do atributo **lista**, nas ações que direcionam para a página de listagem.

---

## Passo 2

Também temos a utilização de **facadeEmpresa**, para definir o atributo **listaEmp**, quando **acao** tem valor **incDep**, e para recuperar a empresa selecionada, quando **acao** vale **incDepExec**.

## Passo 3

Com as estratégias definidas, podemos executar o último passo na construção de nosso aplicativo, adicionando um **Servlet**, com o nome **CadastroFC**, que será

criado de acordo com o padrão **Front Controller**.

Agora, observe o exemplo a seguir.

Java



```
1  @WebServlet(name="CadastroFC", urlPatterns={"/Cadast..."}  
2  public class CadastroFC extends HttpServlet {  
3      @EJB  
4      EmpresaFacadeLocal empresaFacade;  
5      @EJB  
6      DepartamentoFacadeLocal departamentoFacade;  
7  
8      private final HashMap< String,Strategy> estrate...  
9          new HashMap<>();  
10     private final HashMap< String,String> acoes = n...  
11  
12     @Override  
13     public void init() throws ServletException {  
14         super.init();  
15     }  
16 }
```

Como toda a complexidade foi distribuída entre os descendentes de **Strategy**, o Servlet irá funcionar como um simples redirecionador de fluxo. Inicialmente, temos os atributos anotados para acesso aos **EJBs**, e a definição de dois HashMaps, um para armazenar as estratégias de empresa e departamento, com o nome **estrategia**, e outro para relacionar as ações do sistema com os identificadores dos gestores, com o nome **acoes**.

No método **init** temos a inicialização de ambos os HashMaps, e consequente criação das instâncias para os elementos **Strategy**. Note que as ações são facilmente relacionadas com o identificador de estratégia, apoiadas no uso de vetores de texto.

## Comentário

Quanto à resposta ao HTTP, no método **processRequest**, o primeiro passo é a verificação do parâmetro **acao**, obrigatório na chamada ao Servlet. Se o parâmetro estiver presente, localizamos a estratégia correta, a partir dos HashMaps, invocamos a execução, e com o nome da página obtido, efetuamos o redirecionamento via **RequestDispatcher**.

Agora, com o aplicativo completo, podemos executá-lo, testando a inclusão e a listagem, tanto para empresas quanto para departamentos.

Algumas das telas de nosso sistema podem ser observadas a seguir.



## Tela de saída com aplicativo completo



## Tela de cadastro



## Tela de exclusão

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

### Questão 1

No padrão Front Controller, temos a centralização das chamadas, com diversos fluxos de processamento, identificados a partir de algum parâmetro, e o redirecionamento para a visualização, ao final.

Se a visualização representa um estado do sistema, e as transições ocorrem nas chamadas, que diagrama seria adequado para a modelagem?

- A Diagrama de Classes
- B Rede de Petri
- C Diagrama de Entidade e Relacionamento
- D Fluxograma
- E Diagrama de Componentes

**Parabéns! A alternativa B está correta.**

Através de uma Rede de Petri é possível modelar o sistema de forma macroscópica, com a identificação das telas necessárias e fluxo de navegação, representando as transições, sempre passando pelo Front Controller. É um modelo bastante genérico, que define o funcionamento geral, sem descrever detalhes da implementação.

## Questão 2

Quando utilizamos o padrão Front Controller em sistemas Java Web, normalmente através de um Servlet, efetuamos o redirecionamento para a visualização correta, ao final das tarefas de conversão de dados e processamento de regras de negócio. Qual é o tipo de componente utilizado para efetuar o redirecionamento?

- A HttpSession
- B Enterprise Java Bean

C Java Server Pages

D RequestDispatcher

E HttpServletRequest

Parabéns! A alternativa D está correta.

Após receber a solicitação HTTP, e efetuar todos os processamentos necessários, ocorre a obtenção de um RequestDispatcher, apontando para a visualização correta, através do método getRequestDispatcher da requisição. Após a obtenção do objeto, o método forward é invocado, redirecionando o fluxo ao nível do servidor.

## Considerações finais

Analisamos duas tecnologias fundamentais na plataforma Java para o ambiente corporativo: JPA (Java Persistence API) e EJB (Enterprise Java Beans). Observamos que ambas as tecnologias utilizam anotações, e compreendemos o papel do JPA nas tarefas de persistência, com mapeamento objeto-relacional, e do EJB para implementação das regras de negócio de nossos sistemas, de forma síncrona ou assíncrona.

Vimos também alguns padrões arquiteturais e padrões de desenvolvimento, em especial a arquitetura MVC (Model, View e Controller) e o padrão Front Controller, além de avaliar a utilização de algumas tecnologias Java nas camadas do MVC.

Finalmente, construímos um sistema cadastral, no modelo Java para Web, com a adoção de JPA na camada Model e EJB na camada Controller, além de utilizar um Servlet Front Controller e algumas páginas JSP (Java Server Pages) na camada View. Na criação do

sistema, adotamos a automação do NetBeans, visando a um ganho de produtividade.

## 🎧 Podcast

Ouça mais detalhes sobre as tecnologias JPA e JEE.

Para ouvir o *áudio*, acesse a versão online deste conteúdo.



## Explore +

Confira as indicações que separamos especialmente para você!

- Busque o artigo da Oracle comparando Entity Beans CMP e JPA, Oracle.
- Consulte o tutorial de Java Persistence API, apresentado, Vogella.
- Examine a documentação da Oracle sobre componentes do tipo EJB, Oracle.
- Pesquise a documentação interativa acerca de padrões de desenvolvimento, Refactoring Guru.
- Acesse a página da Eclipse para baixar o Eclipse Link.

## Referências

CASSATI, J. P. **Programação Servidor em Sistemas Web**. Rio de Janeiro: Estácio, 2016.

CORNELL, G.; HORSTMANN, C. **Core JAVA**. 8. ed. São Paulo: Pearson, 2010.

DEITEL, P.; DEITEL, H. **AJAX**. Rich Internet Applications e Desenvolvimento Web para Programadores. São Paulo: Pearson Education, 2009.

DEITEL, P.; DEITEL, H. **JAVA**. Como Programar. 8. ed. São Paulo: Pearson, 2010.

FONSECA, E. **Desenvolvimento de Software**. Rio de Janeiro: Estácio, 2015.

MONSON-HAEFEL, R.; BURKE, B. **Enterprise Java Beans 3.0**. 5. ed. USA: O'Reilly, 2006.

SANTOS, F. **Programação I**. Rio de Janeiro: Estácio, 2017.

## Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.

[Download material](#)

O que você achou do conteúdo?



[! Relatar problema](#)