



# Tecnologias JPA e JEE

Você vai compreender a utilização de tecnologias JPA (Java Persistence API) e EJB (Enterprise JavaBeans), elementos centrais do JEE (Java Enterprise Edition), tanto na descrição quanto na organização de uma arquitetura MVC (Model, View e Controller). Além disso, vamos vai descobrir como construir um exemplo completo, observando o padrão Front Controller, com a utilização das tecnologias estudadas. Esses conhecimentos são fundamentais para o profissional de desenvolvimento web.

Prof. Marcos Castro Jr.

### Preparação

Antes de iniciar seus estudos, 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. Sugerimos que você configure a porta do servidor Tomcat como 8084, para evitar conflitos com o GlassFish na porta 8080.

### Objetivos

- Descrever as características do JPA.
- Empregar componentes EJB na construção de regras de negócio.
- Descrever a utilização da arquitetura MVC na plataforma Java.
- Empregar o padrão Front Controller em sistema MVC, com interface Java Web.

### Introdução

Neste conteúdo, abordaremos 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 a implementação de regras de negócio.

Após compreender ambas as tecnologias, analisaremos os elementos estruturais da arquitetura MVC (Model, View e Controller) e desenvolveremos um sistema cadastral com base nessa arquitetura, utilizando JPA, EJB e componentes web. Além disso, adotaremos o padrão Front Controller na camada de visualização para web.

Neste vídeo você verá os principais tópicos do nosso estudo sobre as tecnologias JPA e JEE.

### Introdução



#### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

## Mapeamento objeto-relacional

Nos bancos de dados **relacionais**, a estrutura é baseada em tabelas que armazenam valores em registros, que se relacionam a partir de campos identificadores ou chaves primárias. A manutenção desses relacionamentos é realizada por meio de chaves estrangeiras. Por outro lado, na programação orientada a **objetos** temos as classes, cujas instâncias comportam valores, e que podem se relacionar com outras classes por meio de coleções ou atributos. Não existe uma estrutura de indexação, mas uma relação bilateral, que ocorre por meio de propriedades dos objetos envolvidos.

Neste vídeo, você verá o que é o mapeamento objeto-relacional e como através dele se torna possível fazer com que uma aplicação orientada a objetos se comunique adequadamente com um banco de dados.



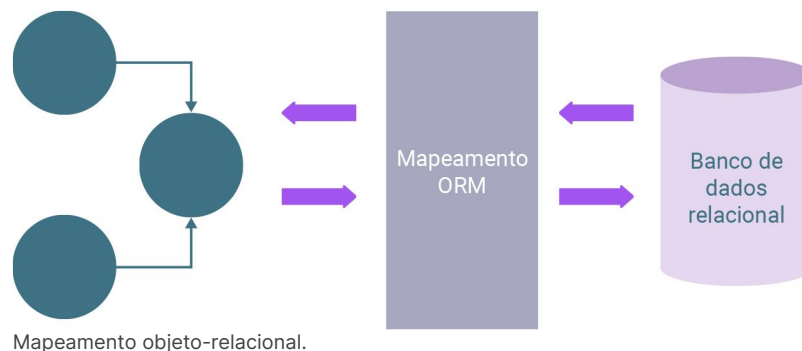
### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Temos, então, duas filosofias distintas. Como o ambiente de programação deve gerenciar toda a lógica, ocorre um esforço natural para minimizar o uso de tabelas e registros, substituindo-os por classes e objetos. Essa 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, abstraindo o enfoque relacional no SGBD.

## ORM (mapeamento objeto-relacional)

O uso de DAO deu origem à técnica de mapeamento **objeto-relacional**, ou **ORM**, na qual uma entidade (objeto) é preenchida com os dados de um registro (relacional). Esse processo, inicialmente realizado de maneira programática, foi modificado com o advento de **frameworks de persistência**, geralmente baseados no uso de XML (eXtended Markup Language) ou anotações.



Com base nas configurações, que indicam a relação entre atributos da classe e colunas do banco, bem como chaves e relacionamentos, o framework gera automaticamente todos os comandos SQL (Structured Query Language) necessários, transmitindo-os para o banco de dados.

## Entity Beans

São parte integrante do J2EE (Java 2 Enterprise Edition) e operam de acordo com o padrão **Active Record**, no qual cada operação com um objeto equivale a um comando executado no banco de dados. Assim, o padrão pode ser ineficiente, devido à grande quantidade de comandos SQL que poderiam ser executados em blocos.

java

```
public abstract class ProdutoEntityBean implements EntityBean {
    public abstract int getCodigo();
    public abstract void setCodigo(int codigo);
    public abstract String getNome();
    public abstract void setNome(String nome);
    public abstract int getQuantidade();
    public abstract void setQuantidade(int quantidade);
    // O restante do código foi omitido
}
```

No fragmento de código, temos o início da definição de um entity bean, em que o objeto é gerado pelo servidor de aplicativos e as classes de entidade apresentam apenas as **propriedades**, além de alguns métodos utilitários. O mapeamento do entity bean para a tabela deve ser feito com base na sintaxe XML, conforme o exemplo a seguir.

python

```
ProdutoEntityBean
PRODUTO

    codigo
    COD_PRODUTO

    nome
    NOME

    quantidade
    QUANTIDADE
```

## Hibernate

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

java

```
public class Produto {
    private int codigo;
    private String nome;
    private int quantidade;

    public Produto(){}
    // Os getters e setters das propriedades foram omitidos
}
```

Para o Hibernate, as entidades são apenas classes comuns, sem métodos de negócios, com um conjunto de propriedades e um construtor padrão. O mapeamento é realizado via **XML**, como no trecho apresentado a seguir.

python

Com o **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.

## Atividade 1

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

**QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO**

A

Active Record.

B

DAO.

C

Facade.

D

Adapter.

E

Front Controller.



A alternativa A está correta.

Ao utilizar os Entity Beans, que operam de acordo com o padrão Active Record, qualquer operação efetuada sobre as entidades causa um efeito diretamente no banco de dados.

## Java Persistence API

Além de ser verboso, o uso de XML faz com que alguns problemas surjam apenas no momento da execução. Um dos maiores avanços do Java foi a criação do JPA, que permitiu padronizar a arquitetura dos frameworks de persistência e concentrou as configurações no arquivo **persistence.xml**. Não é apenas uma biblioteca, mas uma **API** que define a interface comum, configurável por meio de anotações, que deve ser seguida pelos frameworks de persistência.

Neste vídeo, você vai conferir o Java Persistence API (JPA), um framework de mapeamento objeto-relacional (ORM) para Java. Também verá como o JPA simplifica a persistência de dados em aplicações Java, padroniza o desenvolvimento e torna o código mais legível.



#### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

A JPA tem o padrão DAO implícito, o que traz grande eficiência na persistência. Não é por menos que, na plataforma JEE atual, temos a substituição dos entity beans pelo JPA.

## Definindo uma entidade JPA

Para definir uma entidade JPA, devemos criar uma classe sem métodos de negócios, também conhecida como POJO (plain old java object). A entidade definida deve receber anotações para o mapeamento entre a classe e sua tabela, ou seja, o mapeamento objeto-relacional.

```

java

@Entity
@Table(name = "PRODUTO")
@NamedQueries({
    @NamedQuery(name = "Produto.findAll",
        query = "SELECT p FROM Produto p")})
public class Produto implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "COD_PRODUTO")
    private Integer codigo;
    @Column(name = "QUANTIDADE")
    private Integer quantidade;

    public Produto() {
    }
    public Produto(Integer codigo) {
        this.codigo = codigo;
    }
    // Os getters e setters das propriedades foram omitidos
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (codigo != null ? codigo.hashCode() : 0);
        return hash;
    }
    @Override
    public boolean equals(Object object) {
        if (object==null||!(object instanceof Produto)) {
            return false;
        }
        Produto other = (Produto) object;
        return this.codigo!=null &&
            this.codigo.equals(other.codigo);
    }
    @Override
    public String toString() {
        return "model.Produto[ codigo=" + codigo + " ]";
    }
}

```

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. Utilizamos ainda a anotação **NamedQueries** para criar consultas por meio de uma sintaxe denominada JPQL (Java Persistence Query Language).

Vejam, a seguir, as principais anotações do JPA.

- **Entity**: marca a classe como uma entidade para o JPA.
- **Table**: especifica a tabela que será utilizada no mapeamento.
- **Column**: mapeia o atributo para o campo da tabela.
- **Id**: especifica o atributo mapeado para a chave primária.
- **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 por meio 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 um construtor vazio e um outro baseado na chave primária, como podemos verificar no código de **Produto**, além dos métodos equals e hashCode. Ambos os métodos utilitários são baseados no atributo código, que identifica a instância.

Ainda precisamos do atributo **serialVersionUID**, referente à versão da classe e utilizado nos processos de migração da base de dados. Por fim, a implementação de **toString** nos dá controle sobre a representação da entidade como texto.

Além das anotações nas entidades, precisamos configurar o arquivo persistence.xml, definindo 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, sendo sempre presente a especificação do framework de persistência utilizado.

```
java
```

```
    org.eclipse.persistence.jpa.PersistenceProvider;
```

```
model.Produto
```

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, por meio da integração com JTA (Java Transaction API).

Em que temos:

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, definimos o provedor de persistência no elemento **provider**. O elemento **class** define as classes de entidade e as propriedades da conexão são definidas no grupo **properties**.

## Atividade 2

### Questão 1



HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

ID DA QUESTÃO FORA DE PADRÃO... VERIFICAR A QUESTÃO INTEIRA

## Consulta e manipulação de dados

Com as entidades mapeadas e a conexão configurada, podemos consultar e manipular os dados utilizando um gestor de entidades (**EntityManager**).

Neste vídeo, você verá como é possível manipular dados em aplicações Java utilizando as tecnologias JPA, entity manager e JPQL. Acompanhe!



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

### Entity Manager

Concentra os métodos que invocam os comandos SQL montados pelo JPA a partir das anotações da entidade, de uma forma totalmente transparente.

```
java
public class Principal {
    public static void main(String[] args) {
        EntityManagerFactory emf =
            Persistence.createEntityManagerFactory(
                "ExemploSimplesJPAPU");
        EntityManager em = emf.createEntityManager();
        Query query = em.createNamedQuery("Produto.findAll");
        List< Produto > lista = query.getResultList();
        lista.forEach((e) -> {
            System.out.println(e.getNome());
        });
        em.close();
    }
}
```

### Passo 1

O primeiro passo 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, utilizando o método createEntityManager.

### Passo 2

Com o gestor instanciado, obtemos um objeto do tipo Query, com a chamada para createNamedQuery, que utiliza uma NamedQuery da classe Produto. As consultas nomeadas devem apresentar nomes únicos pois, do contrário, poderiam gerar dualidade durante a execução.

### Passo 3

O método getResultList retorna o resultado da consulta ao SGBD em um objeto List. Em termos práticos, a instrução JPQL é transformada em um comando SQL, que 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 a comunicação com o banco de dados, utilizando o método close do EntityManager.



### Atenção

Note que o JPA não elimina o uso de JDBC, pois gera apenas os comandos SQL de forma automatizada, utilizando anotações.

## Inclusão de dados

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

```
java
public static void incluir(Produto p){
    EntityManagerFactory emf =
        Persistence.createEntityManagerFactory(
            "ExemploSimplesJPAPU");
    EntityManager em = emf.createEntityManager();
    try {
        em.getTransaction().begin();
        em.persist(p);
        em.getTransaction().commit();
    } catch (Exception e){
        em.getTransaction().rollback();
    } finally{
        em.close();
    }
}
```

Inclusões podem gerar erros, portanto, o ideal é utilizar transações para executá-las. Na verdade, qualquer manipulação de dados efetuada a partir do JPA exige uma transação.

Após obtermos uma instância de EntityManager na variável **em**, é definido um bloco de código protegido, no qual a transação é iniciada com `begin`, seguida da inclusão do produto na base de dados por meio 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`, em que fechamos a comunicação com o uso de `close`, independentemente da ocorrência de erros.

Para efetuar a alteração dos dados de um registro, temos um processo muito similar, trocando apenas o método `persist` por `merge`.

```

java

public static void alterar(Produto p){
    EntityManagerFactory emf =
        Persistence.createEntityManagerFactory(
            "ExemploSimplesJPAPU");
    EntityManager em = emf.createEntityManager();
    try {
        em.getTransaction().begin();
        em.merge(p);
        em.getTransaction().commit();
    }catch(Exception e){
        em.getTransaction().rollback();
    }finally{
        em.close();
    }
}

```

## Exclusão de dados

Para excluir um registro, devemos utilizar o método find para recuperá-lo. A exclusão em si será executada por meio do método remove, que receberá a instância em questão.

```

java

public static void excluir(Integer codigo){
    EntityManagerFactory emf =
        Persistence.createEntityManagerFactory(
            "ExemploSimplesJPAPU");
    EntityManager em = emf.createEntityManager();
    try {
        em.getTransaction().begin();
        em.remove(em.find(Produto.class, codigo));
        em.getTransaction().commit();
    }catch(Exception e){
        em.getTransaction().rollback();
    }finally{
        em.close();
    }
}

```

Podemos concluir que os métodos find, persist, merge e remove correspondem, respectivamente, aos comandos SELECT, INSERT, UPDATE e DELETE, ao nível do banco de dados.

## Atividade 3

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

**ID DA QUESTÃO FORA DE PADRÃO... VERIFICAR A QUESTÃO INTEIRA**

## Execução do aplicativo

Vamos evitar aqui algumas complexidades do ambiente corporativo, direcionando nosso foco apenas para a interação com bancos de dados. Para isso, vamos utilizar uma aplicação Java simples.

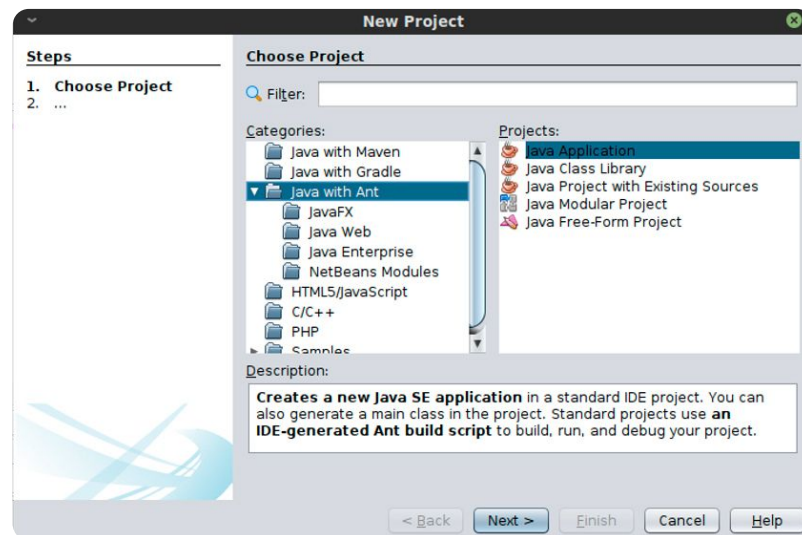
Neste vídeo, você aprenderá a criar e organizar um projeto utilizando os exemplos demonstrados ao longo do conteúdo estudado.



### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Veja, a seguir, como criamos a aplicação Java que iremos utilizar.

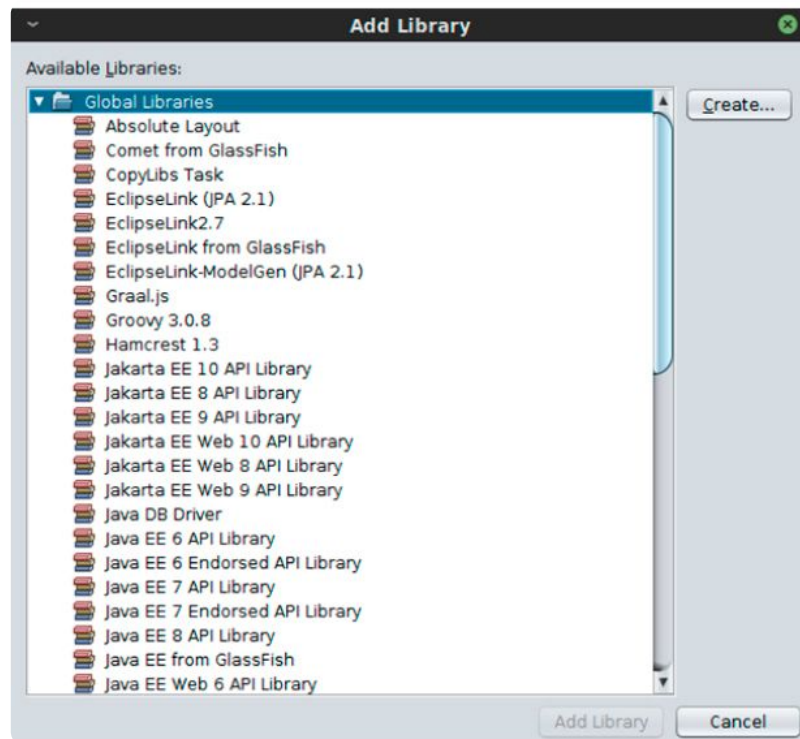


"Wizard" para criação de projetos.

Neste estudo, trabalharemos com o banco de dados **Derby**, também chamado de **Java DB**, exigindo a inclusão da biblioteca **jdbc** correspondente.

## Adicionando a biblioteca JDBC e o framework EclipseLink

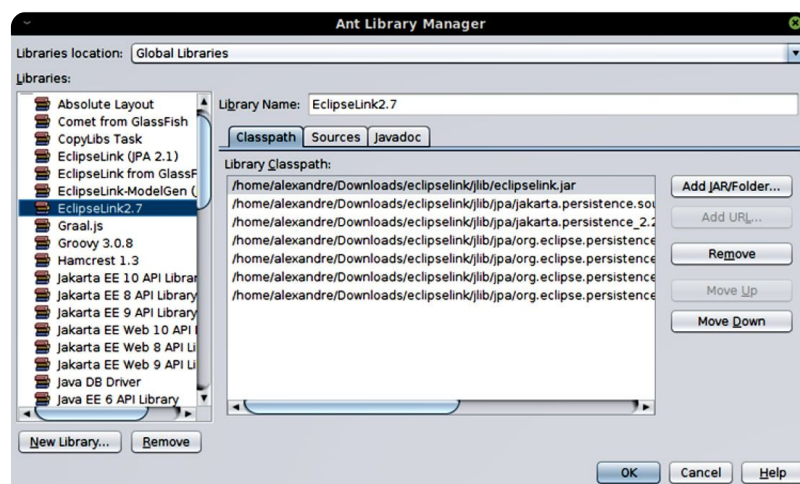
Para adicionar a biblioteca JDBC do Derby, vamos clicar com o botão direito sobre a divisão Libraries, e escolher a opção Add Library. Na janela seguinte, selecionaremos Java DB Driver e clicaremos no botão Add Library.



"Wizard" para inclusão de bibliotecas no projeto do NetBeans.

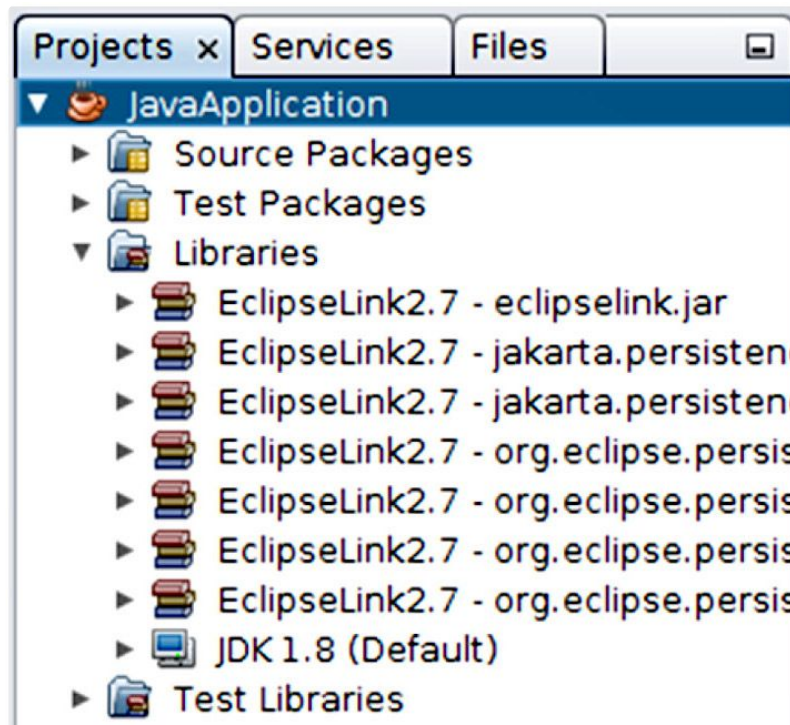
Precisamos acrescentar o framework JPA escolhido, no caso, o **EclipseLink**. Veja como fazer o download desse framework na seção Explore +.

Após efetuar o download da versão mais recente do EclipseLink no formato zip e extrair para algum diretório de fácil acesso, crie uma biblioteca utilizando a opção de menu Tools.Libraries. Em seguida, clique em New Library. A essa nova biblioteca daremos 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 na imagem a seguir.



"Wizard" para inclusão de bibliotecas no projeto do NetBeans.

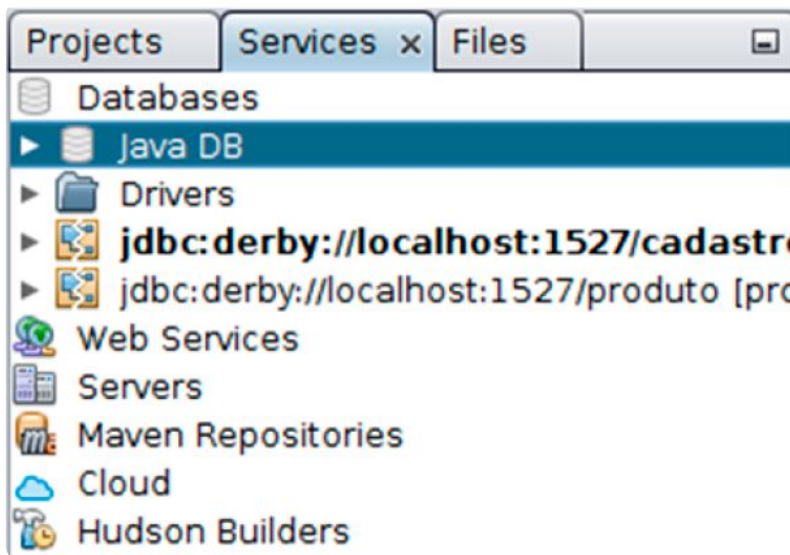
Após a definição da biblioteca, vamos adicionar ao projeto, da mesma forma que fizemos para o driver JDBC. Ao final, teremos a configuração de bibliotecas para o projeto conforme a imagem a seguir.



Navegador do projeto no NetBeans.

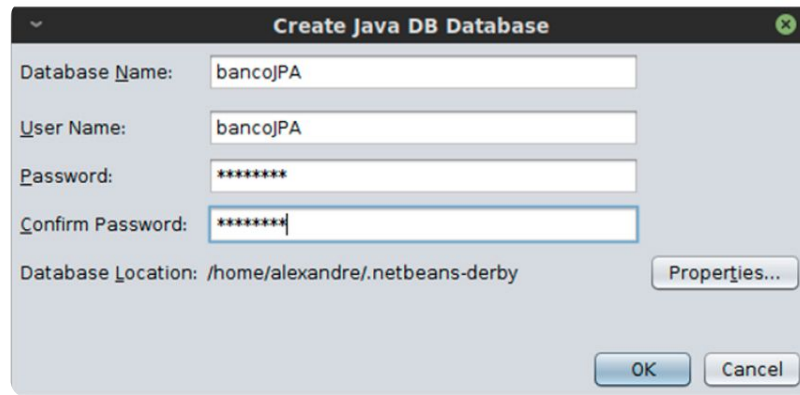
## Criando o banco de dados Derby

Agora, só precisamos de um banco de dados Derby, que será criado de forma simples, por meio da aba Services do NetBeans, na divisão Databases. Veja na imagem a seguir.



Navegador de serviços no NetBeans.

Para criarmos um banco de dados, clique com o botão direito sobre o driver Java DB da árvore de Databases e escolha da opção Create Database. Na janela seguinte, preencha o nome do banco de dados, o usuário e a senha com o valor bancoJPA. Veja na imagem a seguir.



"Wizard" para criação de banco de dados derby no NetBeans.

A conexão é aberta com um duplo-clique sobre o identificador do banco de dados. Após conectado, execute os comandos SQL de criação 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 no botão de execução de SQL na parte superior do editor.

```
sql
```

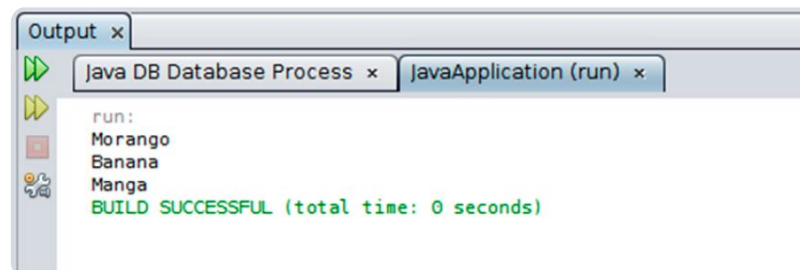
```
CREATE TABLE PRODUTO (  
    COD_PRODUTO INTEGER NOT NULL PRIMARY KEY,  
    NOME VARCHAR(50),  
    QUANTIDADE INTEGER);
```

```
INSERT INTO PRODUTO VALUES (1, 'Morango', 200);  
INSERT INTO PRODUTO VALUES (2, 'Banana', 1000);  
INSERT INTO PRODUTO VALUES (3, 'Manga', 600);
```

```
SELECT * FROM PRODUTO;
```

## Resultado da execução do aplicativo

Ao rodar o programa, a listagem da tabela com os registros inseridos será apresentada na própria janela de edição, em uma divisão própria. Agora, podemos executar nosso projeto, gerando a saída apresentada a seguir.



Saída do programa apresentada no console.

## Atividade 4

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

## Manipulando dados com NamedQueries

Como já vimos, é extremamente necessário estabelecer a comunicação entre a aplicação e o banco de dados. Normalmente, precisamos realizar alguma consulta parametrizada, o que inclui receber os parâmetros, como uma chave primária para identificação, e enviar a instrução para o banco de dados.

Neste vídeo, você verá na prática como utilizar parâmetros em NamedQueries para recuperar registros da tabela.



### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

## Roteiro de prática

Avançaremos um pouco mais, propondo a você o seguinte desafio: alterar o programa apresentado anteriormente, de forma que a classe `principal.Principal` utilize o código para selecionar um produto específico. Para isto, é importante realizar as etapas a seguir.

- Utilizar uma anotação para criar uma NamedQuery na classe `Produto`
- Alterar a NamedQuery utilizada em `Principal`
- Incluir os parâmetros no objeto query

Veja o resultado dessa prática.

### Resultado

---

Classe `"modelo.Produto"`:  
Classe `"principal.Principal"`:



```
java
```

```
@Entity
@Table(name = "PRODUTO")
@NamedQueries({
    @NamedQuery(name = "Produto.findAll", query = "SELECT p FROM Produto p"),
    @NamedQuery(name = "Produto.findByCodProduto", query = "SELECT p FROM Produto p WHERE
p.codProduto = :codProduto")
})
public class Produto implements Serializable {
    // código omitido
```

```
java
```

```
public class Principal {

    public static void main(String[] args) {
        EntityManagerFactory emf
            = Persistence.createEntityManagerFactory(
                "ExemploSimplesJPAPU");
        EntityManager em = emf.createEntityManager();
        Query query = em.createNamedQuery("Produto.findByCodProduto");
        query.setParameter("codProduto", 1);
        List< Produto> lista = query.getResultList();
        lista.forEach((e) -> {
            System.out.println(e.getCodProduto() + "-" + e.getNome());
        });
        em.close();
    }
}
```

```
java
```

```
@Entity
@Table(name = "PRODUTO")
@NamedQueries({
    @NamedQuery(name = "Produto.findAll", query = "SELECT p FROM Produto p"),
    @NamedQuery(name = "Produto.findByCodProduto", query = "SELECT p FROM Produto p WHERE
p.codProduto = :codProduto")
})
public class Produto implements Serializable {
    // código omitido
```

```
java
```

```
public class Principal {

    public static void main(String[] args) {
        EntityManagerFactory emf
            = Persistence.createEntityManagerFactory(
                "ExemploSimplesJPAPU");
        EntityManager em = emf.createEntityManager();
        Query query = em.createNamedQuery("Produto.findByCodProduto");
        query.setParameter("codProduto", 1);
        List< Produto> lista = query.getResultList();
        lista.forEach((e) -> {
            System.out.println(e.getCodProduto() + "-" + e.getNome());
        });
        em.close();
    }
}
```

## Faça você mesmo!

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

**ID DA QUESTÃO FORA DE PADRÃO... VERIFICAR A QUESTÃO INTEIRA**

# Enterprise Java Beans (EJB)

No ambiente Java, a arquitetura de objetos distribuídos geralmente é o elemento central dos servidores corporativos. Podemos observar a importância dessa arquitetura no uso de componentes EJB (Enterprise Java Bean) em servidores de aplicação, como o Glassfish.

Neste vídeo, você verá a importância dos EJBs, bem como suas aplicações e vantagens em um ambiente corporativo.



### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Os EJB são componentes corporativos utilizados de forma indireta dentro de um ambiente de objetos distribuídos, suportando elementos da plataforma JEE (Java Enterprise Edition). Todo EJB é executado em um pool de objetos, cujo número de instâncias varia de acordo com a demanda, segundo um intervalo de tempo estabelecido.

Podemos acessar os serviços oferecidos pelo pool de EJBs por meio de interface local (EJBLocalObject) ou remota (EJBObject), gerada a partir de componentes de fábrica, criados 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:

### Passo 1

Acesso à fábrica de interfaces, por meio de JNDI.

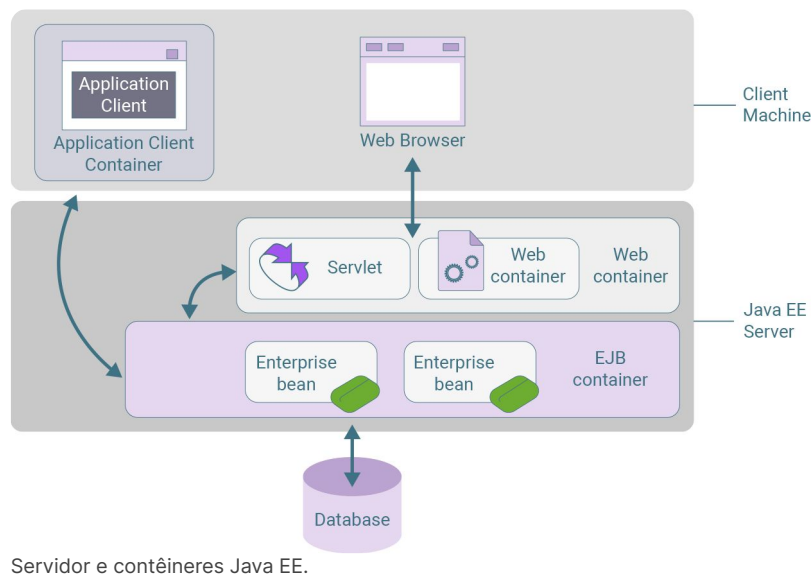
### Passo 2

Geração da interface de acesso pela fábrica.

### Passo 3

Entrega da interface ao cliente, permitindo iniciar o diálogo com o pool.

Veja a representação desses passos na imagem a seguir.



O acesso ao banco de dados é realizado por meio de um pool de conexões JDBC, representado por um objeto do tipo `DataSource`, que é registrado via JNDI. Quando solicitamos uma conexão ao `DataSource`, não estamos abrindo uma nova conexão, mas reservando uma das disponíveis no pool. Quando invocamos o método `close`, não ocorre a desconexão, mas sim a liberação da conexão para a próxima requisição.

No código a seguir, temos um exemplo de utilização de pool de conexões. Após obter o recurso via JNDI, por meio do método `lookup` de `InitialContext`, efetuamos a conversão para o tipo correto, no caso, um `DataSource` que fornece conexões através do método `getConnection`. O restante da programação é a mesma de um acesso local ao banco de dados.

```
java
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    try (PrintWriter out = response.getWriter()) {
        out.println("< html >< body >");
        try {
            InitialContext ctx = new InitialContext();
            DataSource dts = (DataSource) ctx.lookup("jdbc/loja");
            Connection c1 = dts.getConnection();
            Statement st = c1.createStatement();
            ResultSet rs = st.executeQuery("SELECT * FROM PRODUTO");
            while(rs.next())
                out.println(rs.getString("NOME")+"
                    ");
            c1.close();
        } catch (SQLException | NamingException ex) {
        }
        out.println("");
    }
}
```

O JPA abstrai a localização e utilização do pool, que fica 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. O pool disponibiliza um EJB para responder à solicitação. Já na programação do EJB, utilizamos o JPA para obter acesso ao banco de dados.

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, envolvendo uma

grande quantidade de interfaces, classes e arquivos XML, com a verificação sendo realizada apenas no momento da implantação do sistema.

## Atividade 1

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

**QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO**

A

Simplificar o desenvolvimento de interfaces gráficas em aplicações Java.

B

Fornecer uma API para criação de serviços web em Java.

C

Padronizar o desenvolvimento de aplicações Java que utilizam ORM.

D

Gerenciar a camada de apresentação em aplicações Java.

E

Implementar a lógica de negócio em aplicações Java.



A alternativa C está correta.

A JPA padroniza o desenvolvimento de aplicações Java que utilizam ORM, fornecendo uma solução consistente para mapear objetos Java para tabelas em bancos de dados. Trata-se de uma API que define a interface comum, e que deve ser seguida pelos frameworks de persistência.

## Session beans

Esses componentes fundamentais no desenvolvimento de aplicações empresariais em Java são responsáveis por encapsular a lógica de negócios e fornecer serviços específicos para os clientes. Os session beans são amplamente utilizados para implementar transações, acesso a bancos de dados e outras operações relacionadas à lógica de negócios.

Neste vídeo, você vai ver os session beans no Java Enterprise Edition (Java EE), que são componentes de negócios essenciais em aplicativos empresariais. Também verá os dois tipos de session beans, além de suas vantagens, como gerenciamento de transações, concorrência e escalabilidade.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Os EJBs são utilizados para implementar as regras de negócio do nosso aplicativo com base nas entidades e nos requisitos definidos. Esses componentes não podem acumular a responsabilidade sobre a estrutura de persistência utilizada. Também devemos observar que as regras de negócio devem ser totalmente independentes das interfaces do sistema.

O primeiro tipo de EJB que deve ser observado é o de sessão (session beans), responsável por efetuar processos de negócios de forma **síncrona**, e configurável, podendo apresentar três comportamentos distintos:

### Stateless

Não permite a manutenção de estado, ou seja, não guarda valores entre chamadas sucessivas.

### Stateful

É utilizado quando é necessário manter valores entre chamadas sucessivas, como no caso de somatórios.

### Singleton

Permite apenas uma instância por máquina virtual, garantindo o compartilhamento de dados entre todos os usuários.

## Stateless e stateful

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.

## Interface de acesso

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, será suficiente o uso de acesso local, já que teremos o acionamento dos EJBs a partir dos servlets.

```
java

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

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

@Stateless
public class Calculadora implements CalculadoraLocal {
    @Override
    public int somar(int a, int b) {
        return a + b;
    }
}

```

## Singleton

O session bean que implementa o Singleton é utilizado para compartilhar dados entre os usuários conectados, mesmo na execução em ambientes distribuídos. É importante observar que a tecnologia de EJBs é empregada em sistemas de missão crítica, que costumam trabalhar com clusters de computadores.

## Session bean com Servlet

O código a seguir representa o processo de utilização de um session bean a partir de um Servlet.

```

java

@WebServlet(name = "ServletSoma",
urlPatterns = {"/ServletSoma"})
public class ServletSoma extends HttpServlet {

    @EJB
    CalculadoraLocal facade;

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            out.println("< html >< body >");

            out.println("< h1 >Servlet ServletSoma: " +
                facade.somar(2, 3) + "< /h1 >");
            out.println("< /body >");
            out.println("< /html >");
        }
    }
}

```

Tudo que precisamos fazer é anotar um atributo, do tipo da interface local, com EJB. No exemplo, temos a interface `CalculadoraLocal` referenciada no atributo `facade`, o que permite invocar o método `somar`, sendo executado pelo pool.

Quase todos os processos de negócio de um sistema corporativo são implementados por meio de session beans. Contudo, alguns comportamentos não podem ser definidos de forma síncrona, exigindo comunicação com mensagens, seguindo um modelo assíncrono.

## Atividade 2

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

## Message-driven beans (MDB)

Protocolos de mensageria desempenham um papel fundamental em sistemas Java. Esses protocolos definem conjuntos de regras e formatos utilizados para facilitar a troca de mensagens entre diferentes sistemas. Eles determinam como as mensagens são enviadas, recebidas, processadas e entregues aos MDBs (message-driven Beans). Esses protocolos oferecem recursos como persistência de mensagens, filas de mensagens e garantia de entrega, possibilitando uma comunicação confiável e assíncrona para os MDBs.

Neste vídeo, você verá o que são os message-driven beans (MDB) e como eles são usados para processar mensagens assíncronas em aplicativos empresariais. Você também aprenderá sobre as vantagens de escalabilidade e tolerância a falhas que os MDBs oferecem.



### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

## Mensagerias

As mensagerias atuam de forma assíncrona e podem ser:

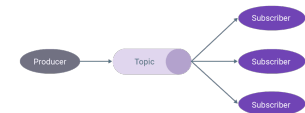
### Modelo publish/subscribe

As mensagens são depositadas em tópicos para que os assinantes as recuperem.



### Modelo point to point

As mensagens são enfileiradas para um tratamento sequencial no destinatário.



O uso de mensagerias permite a construção de sistemas B2B quase sem acoplamento, em que o único elemento de ligação entre os sistemas das duas empresas é a mensagem transmitida entre elas.

Após o emissor publicar uma mensagem, a gerência passa a ser da mensageria, até a retirada pelos receptores. 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**.

python

```
asadmin create-jms-resource --restype javax.jms.ConnectionFactory jms/  
SimpleConnectionFactory
```

```
asadmin create-jms-resource --restype javax.jms.Queue jms/SimpleQueue
```

Também podemos abrir o console do asadmin sem a passagem de parâmetros e invocar os comandos internamente. Vejamos.



```
C:\glassfish\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>
asadmin> create-jms-resource --restype javax.jms.Queue jms/SimpleQueue
Connector resource jms/SimpleQueue created.
Command create-jms-resource executed successfully
```

Comandos executados no shell do asadmin.

## MDB (message-driven bean)

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. Com o MDB, é possível trabalhar nos dois domínios de mensagerias, com o tratamento via pool de EJBs, a partir do método `onMessage`, representando os eventos de recepção de mensagens.

```
java
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
        propertyValue = "jms/SimpleQueue"),
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Queue")
})
public class Mensageiro001 implements MessageListener {

    public Mensageiro001() {
    }

    @Override
    public void onMessage(Message message) {
        try {
            System.out.println("Mensagem enviada: "+
                ((TextMessage) message).getText());
        } catch (JMSException ex) {
            System.out.println("Erro: "+ex.getMessage());
        }
    }
}
```

No código de exemplo, o MDB é definido utilizando a anotação `MessageDriven`, com a configuração para acesso a `jms/SimpleQueue`, do tipo **`javax.jms.Queue`**, por meio 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`, sendo necessário converter para o tipo correto, como no exemplo, em que temos a captura de um texto enviado via `TextMessage`, e a impressão da mensagem no console do GlassFish.



### Saiba mais

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.

O código é feito da seguinte forma:

```
java

@WebServlet(name = "ServletMessage",
urlPatterns = {"/ServletMessage"})
public class ServletMessage extends HttpServlet {
    @Resource(mappedName = "jms/SimpleConnectionFactory")
    private ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/SimpleQueue")
    private Queue queue;

    public void putMessage() throws ServletException {
        try {
            Connection connection =
                connectionFactory.createConnection();
            Session session =
                connection.createSession(false,
                    Session.AUTO_ACKNOWLEDGE);
            MessageProducer messageProducer =
                session.createProducer(queue);
            TextMessage message = session.createTextMessage();
            message.setText("Teste com MDB");
            messageProducer.send(message);
        } catch (JMSException ex) {
            throw new ServletException(ex);
        }
    }

    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            out.println("");
            out.println("");

            putMessage();

            out.println("< h1 >Mensagem Enviada< /h1 >");
            out.println("");
        }
    }
}
```

O processo é um pouco mais complexo que o adotado para os session beans, mas apresenta menor acoplamento. Veja o passo a passo:

### Passo 1

Mapear a fábrica de conexões da mensageria e a fila de destino do MDB, por meio de anotações Resource.

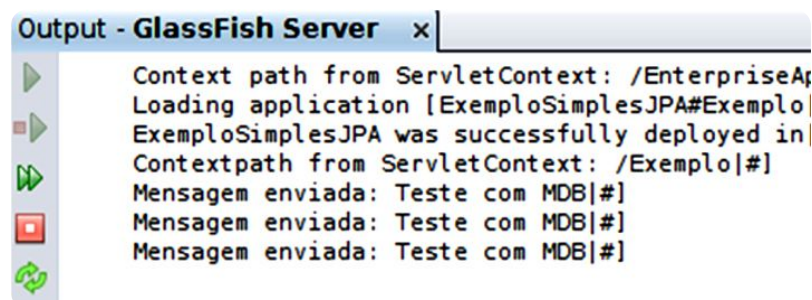
### Passo 2

Definir o método putMessage para envio da mensagem, criando uma conexão a partir da fábrica, uma sessão a partir da conexão, e o produtor de mensagens (MessageProducer) a partir da sessão.

### Passo 3

Criar a mensagem de texto (TextMessage) por meio da sessão, definir o texto que será enviado com setText, e finalmente enviar a mensagem, finalizando a definição do método putMessage.

Veja a representação na imagem a seguir:



```
Output - GlassFish Server x
Context path from ServletContext: /EnterpriseAp
Loading application [ExemploSimplesJPA#Exemplo
ExemploSimplesJPA was successfully deployed in
Contextpath from ServletContext: /Exemplo|#]
Mensagem enviada: Teste com MDB|#]
Mensagem enviada: Teste com MDB|#]
Mensagem enviada: Teste com MDB|#]
```

Console do Glassfish exibindo as mensagens de teste.

## Atividade 3

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

ID DA QUESTÃO FORA DE PADRÃO... VERIFICAR A QUESTÃO INTEIRA

## Aplicativo corporativo

Aplicativos Java corporativos são softwares desenvolvidos para atender às necessidades complexas e específicas das organizações. Eles são construídos utilizando tecnologias Java Enterprise Edition (Java EE), fornecendo recursos avançados, como gerenciamento de transações, segurança, escalabilidade e integração com sistemas legados. Esses aplicativos são projetados para lidar com grandes volumes de dados e suportar operações críticas para o funcionamento das empresas.

Neste vídeo você verá a estrutura de um aplicativo corporativo baseado no Java Enterprise Edition (Java EE) e como os Enterprise JavaBeans (EJB) e Message Driven Beans (MDB) desempenham um papel importante nessa estrutura.



## Conteúdo interativo

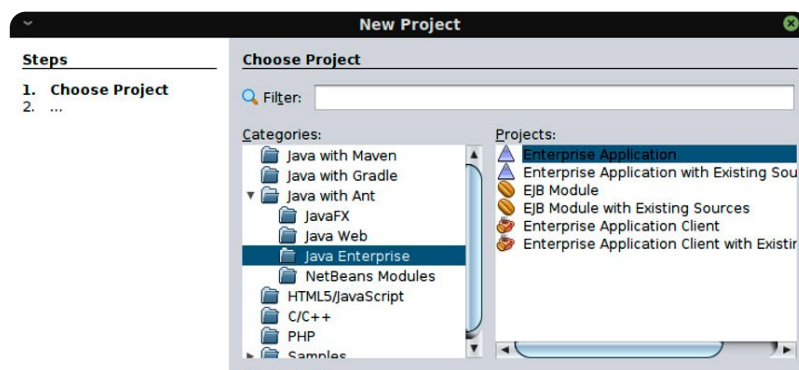
Acesse a versão digital para assistir ao vídeo.

## EJBs

Para que possamos trabalhar com **EJBs**, por meio 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

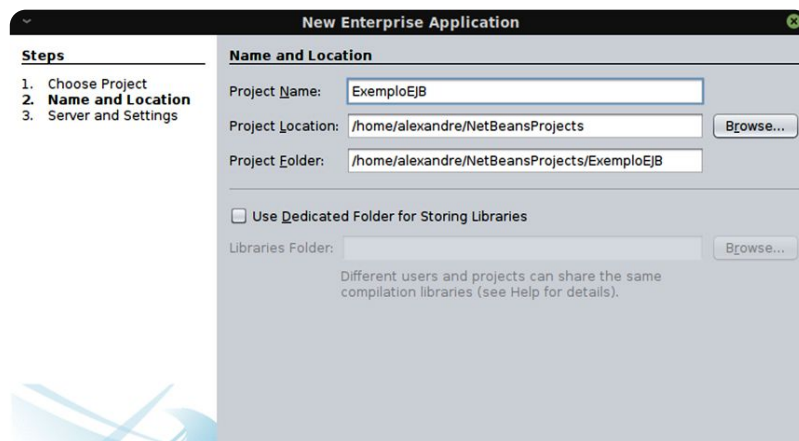
Criar um projeto do tipo Enterprise Application, na categoria Java Enterprise.



"Wizard" para criação de aplicação do NetBeans.

### Passo 2

Preencher o nome (ExemploEJB) e local do projeto.



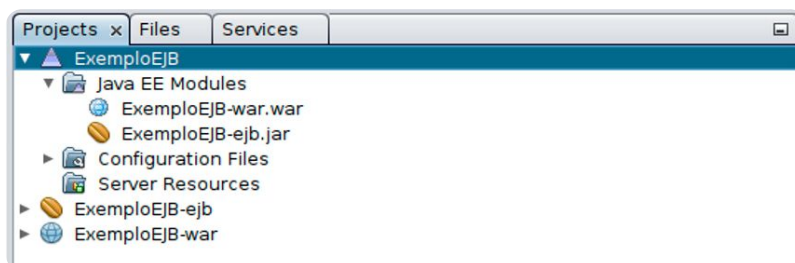
### Passo 3

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



"Wizard" para criação de aplicação do NetBeans.

Seguindo corretamente os passos apresentados, você verá que serão gerados três projetos. Poderemos visualizá-los na interface do NetBeans.



Navegador de projeto do NetBeans.

A seguir, apresentamos as características de cada um dos projetos.

#### ExemploEJB-ejb

Utilizado na definição das entidades JPA e dos componentes EJB, sendo compilado com a extensão "jar".

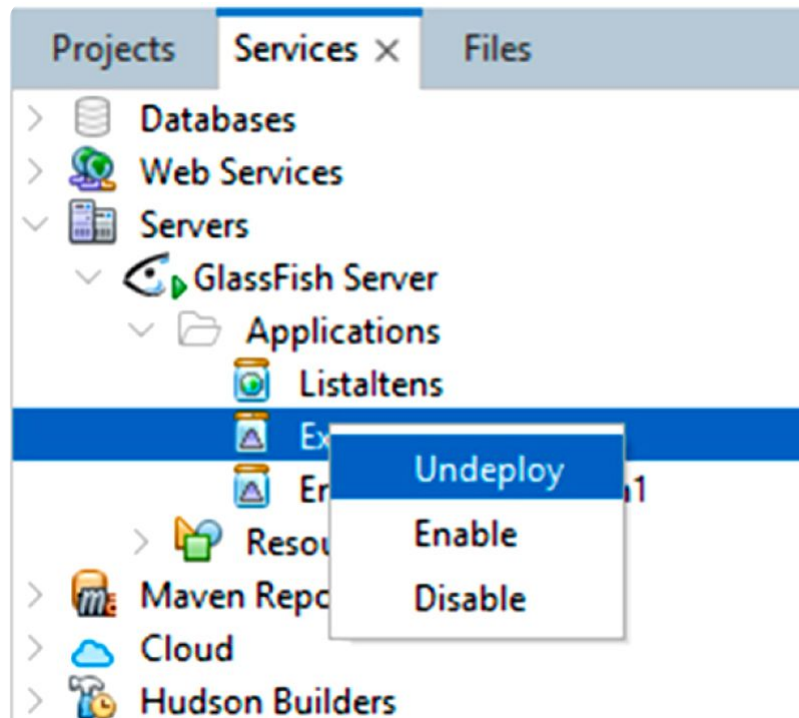
#### ExemploEJB-war

Contém os elementos para web, como servlets, facelets e páginas XHTML, compilados para um arquivo "war".

#### ExemploEJB

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.

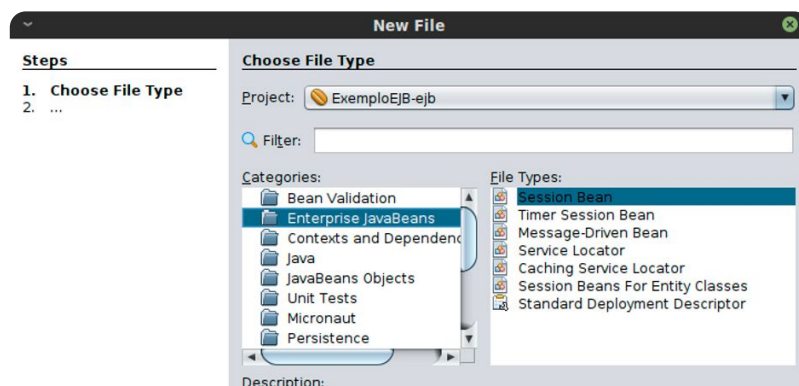


Remoção de aplicação web do servidor Glassfish.

Agora, vamos criar nosso primeiro session bean, configurado como stateless, no projeto secundário **ExemploEJB-ejb**, adicionando um novo arquivo e seguindo alguns passos:

## Passo 1

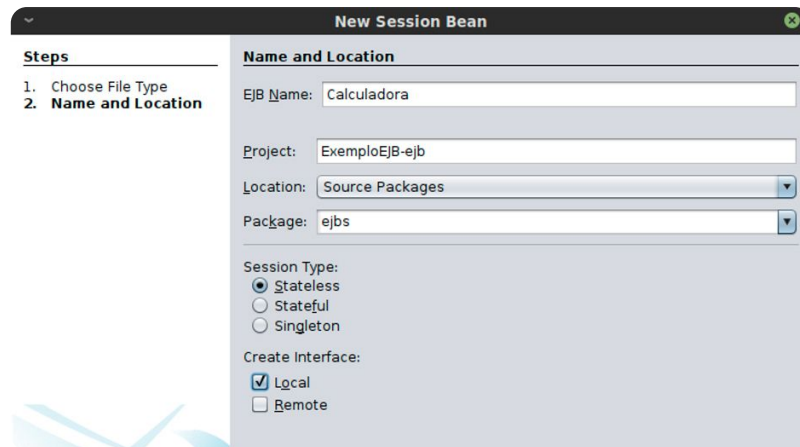
Selecionar o tipo de session bean na categoria Enterprise Java Beans.



"Wizard" de criação de arquivos do NetBeans.

## Passo 2

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



"Wizard" de criação de arquivos do NetBeans.

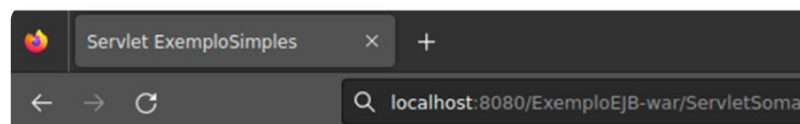
Em diversas situações, a IDE mostra um **erro de compilação** decorrente da importação dos componentes da biblioteca **javax.ejb**. Caso esse problema ocorra, a solução é simples com a inclusão da biblioteca **Java EE 7 API** no projeto.



Navegador de projeto do NetBeans.

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, por meio de um servlet. Como os componentes web são criados ao nível de *ExemploEJB-war*, devemos acessar o projeto e adicionar um 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 apropriada: `http://localhost:8080/ExemploSimplesJPA-war/ServletSoma`.



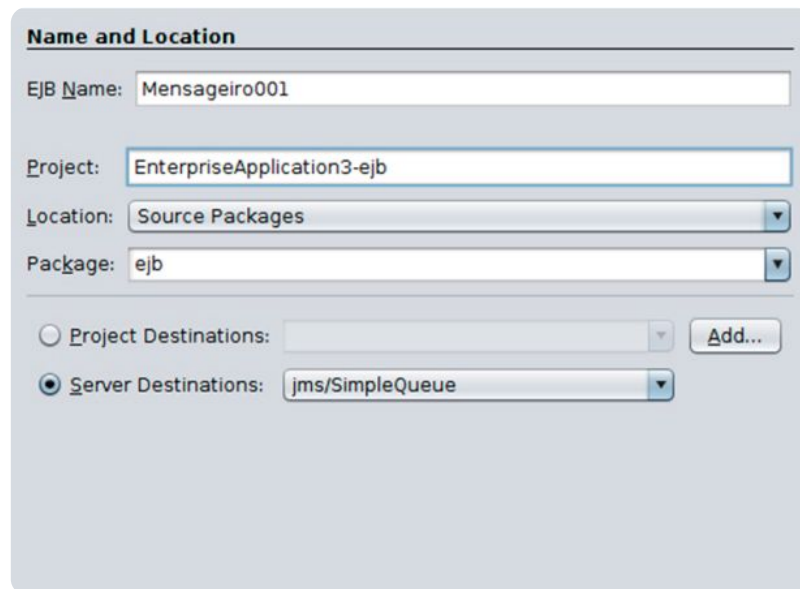
## Servlet ServletSoma: 5

Saída do servlet *ServletSoma*.

## MDBs

Para criar o EJB do tipo **MDB**, devemos adicionar, no projeto **ExemploEJB-ejb**, um novo arquivo do tipo **message-driven bean**, na categoria **Enterprise Java Beans**.

Precisamos definir o nome (**Mensagem001**) e o pacote do componente, além de escolher a fila para as mensagens no servidor (**jms/SimpleQueue**), como na imagem a seguir.



**Name and Location**

EJB Name: Mensageiro001

Project: EnterpriseApplication3-ejb

Location: Source Packages

Package: ejb

☐ Project Destinations:   
☒ Server Destinations: jms/SimpleQueue

Add...

"Wizard" para criação de um MDB.

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 pelo endereço: "http://localhost:8080/ExemploSimplesJPA-war/ServletMessage".

Após efetuar a chamada, você verá 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.

## Atividade 4

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

ID DA QUESTÃO FORA DE PADRÃO... VERIFICAR A QUESTÃO INTEIRA

## Aplicando os MDBs

No mundo web é bastante comum a utilização de recursos externos, sejam eles de outro projeto da sua organização ou mesmo de outra organização. Esses recursos podem ser desenvolvidos utilizando linguagens e respeitando diferentes padrões. Para facilitar a comunicação, uma excelente alternativa é a troca de mensagens.

Neste vídeo você verá como são utilizados Message-Driven Beans (MDBs) baseados em tópicos para processar mensagens assíncronas em aplicativos corporativos. Também a configuração do ambiente e a implementação do MDB.



Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

### Roteiro de prática



No mundo Java, utilizamos muito os MDB (message-driven beans). Portanto, vamos exercitar a criação desse tipo de componente desenvolvendo duas classes no mesmo projeto: uma para receber mensagens e outra para produzir. Para isso, vamos usar o padrão point-to-point, conforme as etapas a seguir:

- Crie a fábrica de conexões e a fila no glassfish, utilizando o asadmin.
- No projeto ejb, desenvolva a classe MeuReceptor, responsável por tratar as mensagens.
- No projeto war, desenvolva uma classe MeuProdutor, que contenha um servlet para enviar uma mensagem para o receptor.
- No projeto war, altere o index.html com um formulário de envio de mensagem.
- Verifique o resultado no console do glassfish.

Veja o resultado dessa prática.

### Resultado

---

"MeuReceptor.java"  
"MeuProdutor.java"  
"Index.html"

java

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue = "jms/
MinhaQueue"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Queue")
})
public class MeuReceptor implements MessageListener {

    public MeuReceptor() {
    }

    @Override
    public void onMessage(Message message) {
        try { System.out.println("Mensagem enviada: "
            + ((TextMessage) message).getText());
        } catch (JMSEException ex) {
            System.out.println("Erro: " + ex.getMessage());
        }
    }
}
```

java

```
@WebServlet("/MeuProdutor")
public class MeuProdutor extends HttpServlet {

    @Resource(mappedName = "jms/MinhaConnectionFactory")
    private ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/MinhaQueue")
    private Queue queue;

    public void putMessage(String mensagem) {
        try {
            Connection connection = connectionFactory.createConnection();
            Session session = connection.createSession(false,
                Session.AUTO_ACKNOWLEDGE);
            MessageProducer messageProducer
                = session.createProducer(queue);
            TextMessage message = session.createTextMessage();
            message.setText(mensagem);
            messageProducer.send(message);
        } catch (JMSException ex) {

        }
    }

    @Override
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws IOException{
        response.setContentType("text/html;charset=UTF-8");
        request.setCharacterEncoding("UTF-8");
        String mensagem = request.getParameter("mensagem");
        putMessage(mensagem);
        PrintWriter out = response.getWriter();
        out.println("

Sua Mensagem foi Enviada

");
        out.println("Mensagem: " + mensagem);
    }
}
```

python

java

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue = "jms/
MinhaQueue"),
    @ActivationConfigProperty(propertyName = "destinationType", propertyValue =
"javax.jms.Queue")
})
public class MeuReceptor implements MessageListener {

    public MeuReceptor() {
    }

    @Override
    public void onMessage(Message message) {
        try { System.out.println("Mensagem enviada: "
            + ((TextMessage) message).getText());
        } catch (JMSEException ex) {
            System.out.println("Erro: " + ex.getMessage());
        }
    }
}
```

java

```
@WebServlet("/MeuProdutor")
public class MeuProdutor extends HttpServlet {

    @Resource(mappedName = "jms/MinhaConnectionFactory")
    private ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/MinhaQueue")
    private Queue queue;

    public void putMessage(String mensagem) {
        try {
            Connection connection = connectionFactory.createConnection();
            Session session = connection.createSession(false,
                Session.AUTO_ACKNOWLEDGE);
            MessageProducer messageProducer
                = session.createProducer(queue);
            TextMessage message = session.createTextMessage();
            message.setText(mensagem);
            messageProducer.send(message);
        } catch (JMSException ex) {

        }
    }

    @Override
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response) throws IOException{
        response.setContentType("text/html;charset=UTF-8");
        request.setCharacterEncoding("UTF-8");
        String mensagem = request.getParameter("mensagem");
        putMessage(mensagem);
        PrintWriter out = response.getWriter();
        out.println("

Sua Mensagem foi Enviada

");
        out.println("Mensagem: " + mensagem);
    }
}
```

python

# Faça você mesmo!

Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

ID DA QUESTÃO FORA DE PADRÃO... VERIFICAR A QUESTÃO INTEIRA

## Padrões de desenvolvimento

A orientação a objetos representou um grande avanço na implementação de sistemas, pois aproximou a modelagem da codificação. Nesse contexto, os padrões de desenvolvimento fornecem ainda mais vantagens ao desenvolvimento, definindo soluções reutilizáveis, com nome, descrição da finalidade, modelagem UML e modo de utilização.

Neste vídeo, você entenderá sobre padrões de desenvolvimento, soluções reutilizáveis para problemas comuns na programação, bem como sua importância, e alguns exemplos, como Singleton e o Abstract Factory.



#### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

Entre os diversos padrões de projeto, alguns se destacam nos sistemas corporativos, como Facade, Proxy, Flyweight, Front Controller e DAO (data access object). Na tecnologia JPA observamos o padrão DAO, com comandos para acesso ao banco de dados agrupados em classes específicas, separadas do restante do sistema.

## Descrição dos padrões de desenvolvimento

Vejam os padrões de projeto citados, além de alguns outros, comuns em sistemas criados para o ambiente JEE.

- **Abstract Factory:** definição de uma arquitetura abstrata para a geração de objetos, muito comum em frameworks.
- **Command:** encapsula o processamento da resposta para algum tipo de requisição, muito utilizado para o tratamento de solicitações feitas no protocolo HTTP.
- **Data Access Object:** utiliza classes específicas para concentrar as chamadas para o banco de dados.
- **Facade:** encapsula as chamadas para um sistema complexo, muito utilizado em ambientes corporativos.
- **Flyweight:** cria grupos de objetos que respondem a uma grande quantidade de chamadas.
- **Front Controller:** concentra as chamadas para o sistema, efetuando os direcionamentos corretos para cada chamada.
- **Iterator:** permite acesso sequencial aos objetos de uma coleção, o que é implementado nativamente no Java.
- **Proxy:** Define um objeto para substituir a referência de outro, utilizado nos objetos remotos para deixar a conexão transparente para o programador.
- **Service Locator:** gerencia a localização de recursos compartilhados, com base em serviços de nomes e diretórios.
- **Singleton:** garante a existência de apenas uma instância para a classe, como em controles de acesso.
- **Strategy:** seleciona algoritmos em tempo de execução, com base em algum parâmetro fornecido.

Atualmente, os sistemas combinam diversos padrões de projeto em suas implementações.



### Exemplo

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.

## Atividade 1

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

**QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO**

A

Persistence.

B

EntityManager.

C

Query.

D

Transaction.

E

EntityManagerFactory.



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 meio de uma coleção.

## Padrões arquiteturais

A arquitetura de um sistema define sua estrutura de alto nível, formalizando a organização em termos de componentes e interações. 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.



Neste vídeo, você entenderá sobre padrões de projeto, bem como exemplos com 3 entre os diversos que existem: camadas, broker e Event-driven. Os padrões arquiteturais podem ser combinados de forma a entregar soluções robustas para problemas complexos.



### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

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. Entenda:

#### Modelo arquitetural

Define a arquitetura de forma abstrata, com foco apenas no objetivo ou característica principal.

#### Padrão arquitetural

Define o perfil dos componentes estruturais, o modelo de comunicação e até os padrões de desenvolvimento mais adequados.

Por exemplo, o modelo de **objetos distribuídos** define apenas atributos essenciais para delimitar um ambiente com coleções de objetos respondendo a requisições.

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

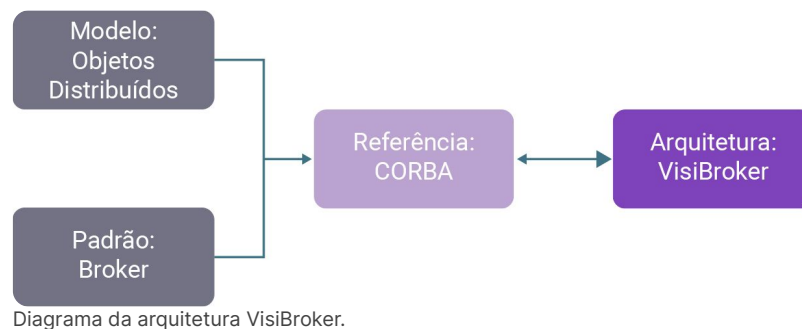
Padrão arquitetural	Modelo(s)
Broker	Sistemas Distribuídos
Camadas	Mud to structure, chamada e retorno
Orientado a objetos	Chamada e retorno
Programa principal e sub-rotina	Chamada e retorno
Pipes/Filters	Mud to structure, fluxo de dados
Blackboard	Mud to structure, centrada em dados
Lote	Fluxo de dados
Repositório	Centrada em dados
Processos comunicantes	Componentes independentes
Event-Driven	Componentes independentes
Interpretador	Máquina virtual
Baseado em regras	Máquina virtual
MVC	Sistemas interativos
PAC	Sistemas interativos
Microkernel	Sistemas adaptáveis

Padrão arquitetural	Modelo(s)
Reflexiva	Sistemas adaptáveis

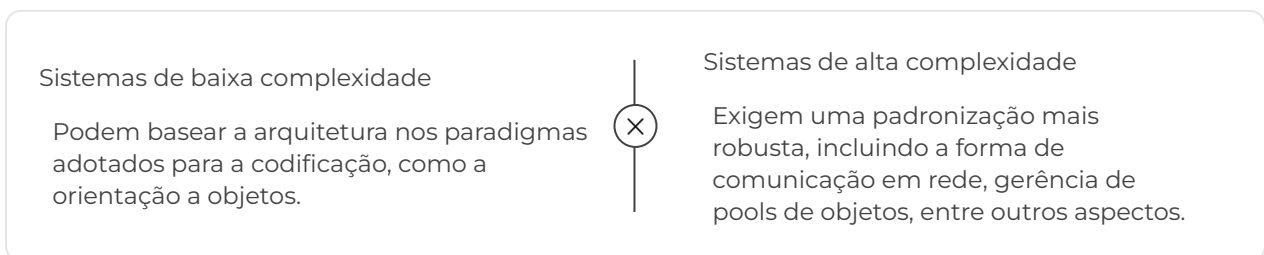
Tabela: Modelos dos padrões arquiteturais.  
Marcos Castro Jr.

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

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**. Veja essa representação na imagem a seguir.



Para utilizar uma arquitetura, é necessário compreender sua finalidade, de acordo com o padrão arquitetural adotado. Veja:



Ambientes de execução remota, como RPC (remote procedure call) e Web Services, são baseados em arquiteturas no padrão de **processos comunicantes**. Nesses ambientes, 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 mensagens nos sistemas corporativos, nas quais utilizamos o padrão arquitetural event-driven, baseado na ativação de processos de forma indireta, a partir de mensagens. O papel das mensagens é tão importante que o componente adotado na comunicação é chamado de MOM (message-oriented middleware). 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. Normalmente, é adotado um padrão principal, que, no caso dos sistemas cadastrais, é o MVC.

## Atividade 2

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

ID DA QUESTÃO FORA DE PADRÃO... VERIFICAR A QUESTÃO INTEIRA

# Arquitetura MVC

A arquitetura MVC (model-view-controller) divide o sistema em três camadas, com responsabilidades específicas.

- **Model:** nessa camada, temos as entidades e as classes para acesso ao banco de dados.
- **Controller:** aqui, concentramos os objetos de negócio.
- **View:** nessa camada, são definidas as interfaces do sistema com o usuário ou com outros sistemas.

Para saber um pouco mais sobre a arquitetura MVC, não deixe de assistir o vídeo a seguir.

Neste vídeo, você vai entender a arquitetura MVC no desenvolvimento web. Essa arquitetura consiste em separar a aplicação em três partes (modelo, visualização e controlador), permitindo o seu gerenciamento de forma independente.



## Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

## Camadas da MVC

A seguir, vamos ver as principais características de cada camada que compõe a arquitetura MVC.

## 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.
- 
- 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 (controlador)

- 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.
- 
- 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.
- 
- 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.

A arquitetura MVC é baseada em camadas, e cada camada enxerga apenas a camada imediatamente abaixo.

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

Como as instruções SQL são bastante padronizadas, é possível criar ferramentas para a geração dos comandos e preenchimento das entidades, bastando expressar a relação entre atributos da entidade e campos do registro.

A camada Controller precisa ser definida de maneira independente de ambiente específico, como interfaces SWING ou protocolo HTTP. A única dependência aceitável para os objetos de negócio deve ser com relação à camada Model. Como são os modelos que gerenciam os componentes DAO, isso diminui a complexidade nas atividades cadastrais iniciadas na View.

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. Nos modelos atuais, é comum a geração de um token, mantendo a independência entre as camadas.

## Atividade 3

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

ID DA QUESTÃO FORA DE PADRÃO... VERIFICAR A QUESTÃO INTEIRA

## 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, foram criados diversos frameworks, 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 camada Model.

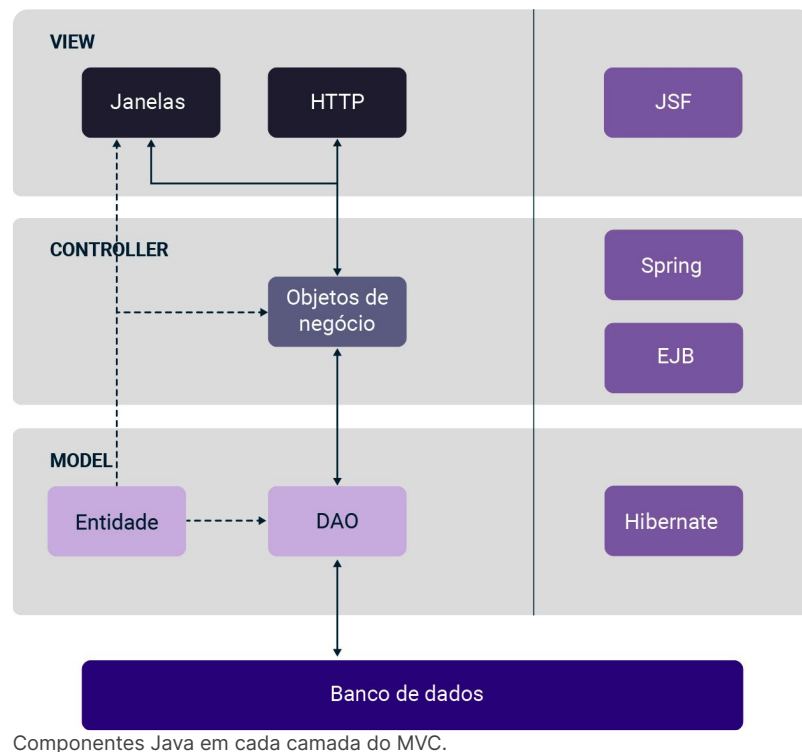
Neste vídeo, você verá o papel de quatro componentes Java para o padrão MVC: JSF, Spring, EJB e Hibernate.



### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

O uso de camadas especializadas permite a divisão da equipe entre profissionais cujo perfil seja voltado para criação visual, negócios ou banco de dados. Veja na imagem:



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.

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.

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. A independência do núcleo garante que ele possa ser utilizado por diversas interfaces simultâneas, como:

- SWING
- HTTP
- Web Services

Sem que ocorra qualquer modificação nos componentes do tipo JPA ou EJB.

Nos sistemas Java para web, um erro comum 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. Essa abordagem errônea faz com que qualquer nova interface, como SWING, Web Services, ou até uma linha de comando, seja obrigada a solicitar os serviços utilizando o protocolo HTTP, algo que, de forma geral, não é uma exigência das regras de negócio dos sistemas.

Considere que a entidade Produto, definida anteriormente, com uso de tecnologia JPA, seja criada no projeto ExemploEJB-ejb, no qual 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.

```

java

@Local
public interface ProdutoGestorLocal {
    List< Produto > obterTodos();
    void incluir(Produto p);
}

@Stateless
public class ProdutoGestor implements ProdutoGestorLocal {
    @Override
    public List obterTodos() {
        EntityManagerFactory emf = Persistence.
            createEntityManagerFactory("ExemploSimplesJPAPU");
        EntityManager em = emf.createEntityManager();
        Query query = em.createNamedQuery("Produto.findAll");
        List< Produto > lista = query.getResultList();
        em.close();
        return lista;
    }
    @Override
    public void incluir(Produto p) {
        EntityManagerFactory emf = Persistence.
            createEntityManagerFactory("ExemploSimplesJPAPU");
        EntityManager em = emf.createEntityManager();
        try {
            em.getTransaction().begin();
            em.persist(p);
            em.getTransaction().commit();
        } catch (Exception e) {
            em.getTransaction().rollback();
        } finally {
            em.close();
        }
    }
}

```

Repare que nossos códigos de exemplos anteriores foram aproveitados aqui, com leves adaptações, para que as operações sejam disponibilizadas por meio do session bean.

Precisamos adicionar o arquivo **persistence.xml**, 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.

Com nossas camadas Model e Controller completamente codificadas, podemos definir um Servlet, no projeto ExemploEJB-war, com o nome **ServletListaProduto**, que 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.

```

java

@WebServlet(name = "ServletListaProduto",
urlPatterns = {"/ServletListaProduto"})
public class ServletListaProduto extends HttpServlet {
    @EJB
    ProdutoGestorLocal facade;

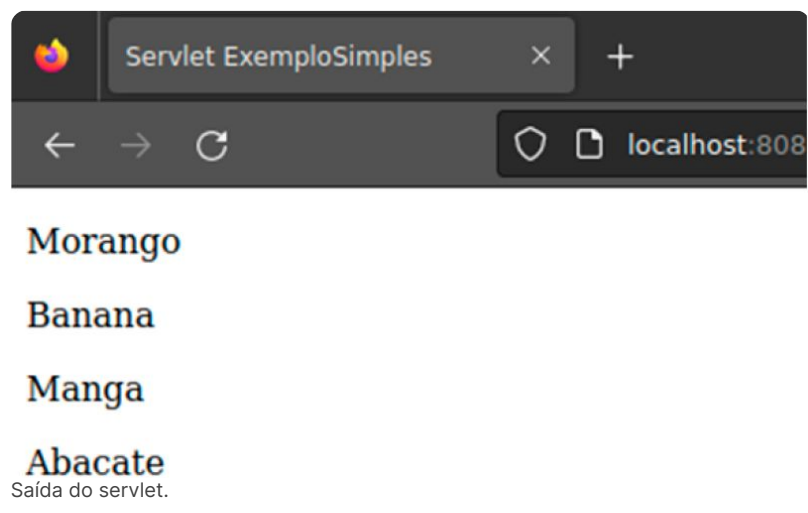
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            out.println("< html >< body >");
            facade.obterTodos().forEach(p -> {
                out.println("
                    " + p.getNome());
            });
            out.println("");
        }
    }
}

```

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 a apresentada a seguir.

<http://localhost:8080/ExemploEJB-war/ServletListaProduto>

Caso esteja tudo correto, teremos uma saída similar à que vemos a seguir, na tela do navegador.



## Atividade 4

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

ID DA QUESTÃO FORA DE PADRÃO... VERIFICAR A QUESTÃO INTEIRA

## Implementação de MVC no ambiente Java



A maioria dos sistemas possui diversas entidades de negócio que podem se relacionar. Esse relacionamento deve ser implementado nas camadas de banco e modelos. Contudo, esse tipo de situação costuma extrapolar as camadas mais baixas, reverberando até nas visualizações.

Neste vídeo, você verá a implementação de um sistema simples de gerenciamento de tarefas. Esta atividade envolve desde o registro nos bancos de dados até a elaboração de uma visualização correspondente, passando por criação de EJBs e modelos.



#### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

## Roteiro de prática

Utilize o exemplo de sistema de produtos e adicione as camadas correspondentes à entidade "empresa". Um produto deverá possuir apenas uma empresa. A empresa, por sua vez, pode possuir diversos produtos. Para executar os novos requisitos, siga as etapas:

- Criar as tabelas e inserir os dados de empresas e produtos.
- Crie a entidade "empresa" a partir do banco de dados.
- Desenvolva um EJB para intermediar a comunicação com o modelo Produto.
- Crie um Servlet responsável pela Listagem de produtos.

Veja o resultado dessa prática.

### Resultado

---

cria\_registros.sql:  
Produto.java  
Empresa.java  
ProdutoGestorLocal.java  
ProdutoGestor.java  
ListaProduto.java

sql

```
CREATE TABLE EMPRESA (  
  CODIGO INT NOT NULL PRIMARY KEY,  
  RAZAO_SOCIAL VARCHAR(50));  
  
CREATE TABLE PRODUTO (  
  CODIGO INT NOT NULL PRIMARY KEY,  
  NOME VARCHAR(50),  
  QUANTIDADE INTEGER,  
  COD_EMPRESA INT NOT NULL);  
  
ALTER TABLE PRODUTO ADD FOREIGN KEY(COD_EMPRESA)  
REFERENCES EMPRESA(CODIGO);  
  
INSERT INTO EMPRESA VALUES (1,'SKY NET');  
INSERT INTO EMPRESA VALUES (2,'MATRIX');  
  
INSERT INTO PRODUTO VALUES (1,'Morango',200, 1);  
INSERT INTO PRODUTO VALUES (2,'Banana',1000, 1);  
INSERT INTO PRODUTO VALUES (3,'Manga',600, 2);
```

java

```
@Entity  
@Table(name = "PRODUTO")  
@NamedQueries({  
    @NamedQuery(name = "Produto.findAll", query = "SELECT p FROM Produto p")})  
public class Produto implements Serializable {  
  
    private static final long serialVersionUID = 1L;  
    @Id  
    @Basic(optional = false)  
    @NotNull  
    @Column(name = "CODIGO")  
    private Integer codigo;  
    @Size(max = 50)  
    @Column(name = "NOME")  
    private String nome;  
    @Column(name = "QUANTIDADE")  
    private Integer quantidade;  
    @JoinColumn(name = "COD_EMPRESA", referencedColumnName = "CODIGO")  
    @ManyToOne(optional = false)  
    private Empresa codEmpresa;  
  
    // Restante do código omitido
```

```

java

@Entity
@Table(name = "EMPRESA")
@NamedQueries({
    @NamedQuery(name = "Empresa.findAll", query = "SELECT e FROM Empresa e")})
public class Empresa implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Column(name = "CODIGO")
    private Integer codigo;
    @Size(max = 50)
    @Column(name = "RAZAO_SOCIAL")
    private String razaoSocial;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "codEmpresa")
    private Collection produtoCollection;
    //Restante do código omitido

```

```

java

@Local
public interface ProdutoGestorLocal {
    List listaTodos();
}

```

```

java

@Stateless
public class ProdutoGestor implements ProdutoGestorLocal {

    @Override
    public List listaTodos() {
        EntityManagerFactory emf
            = Persistence.createEntityManagerFactory("EnterpriseApplication1-ejbPU2");
        EntityManager em = emf.createEntityManager();
        Query query = em.createNamedQuery("Produto.findAll");
        List lista = query.getResultList();
        em.close();
        return lista;
    }
}

```

java

```
@WebServlet(urlPatterns = {"/ListaProduto"})
public class ListaProduto extends HttpServlet {
    @EJB
    beans.ProdutoGestorLocal facade;
    protected void processRequest(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use following sample code. */
            out.println("");
            out.println("");
            out.println("");
            out.println("");
            out.println("");
            out.println("");
            out.println("");
            List lista = facade.listaTodos();
            for (Produto produto : lista) {
                out.println("

" + produto.getNome() + " - Empresa: " +
produto.getCodEmpresa().getRazaoSocial() + "
");
            }
            out.println("");
            out.println("");
        }
    }
    // Códito omitido.
}
```

sql

```
CREATE TABLE EMPRESA (
CODIGO INT NOT NULL PRIMARY KEY,
RAZAO_SOCIAL VARCHAR(50));

CREATE TABLE PRODUTO (
CODIGO INT NOT NULL PRIMARY KEY,
NOME VARCHAR(50),
QUANTIDADE INTEGER,
COD_EMPRESA INT NOT NULL);

ALTER TABLE PRODUTO ADD FOREIGN KEY(COD_EMPRESA)
REFERENCES EMPRESA(CODIGO);

INSERT INTO EMPRESA VALUES (1,'SKY NET');
INSERT INTO EMPRESA VALUES (2,'MATRIX');

INSERT INTO PRODUTO VALUES (1,'Morango',200, 1);
INSERT INTO PRODUTO VALUES (2,'Banana',1000, 1);
INSERT INTO PRODUTO VALUES (3,'Manga',600, 2);
```

java

```
@Entity
@Table(name = "PRODUTO")
@NamedQueries({
    @NamedQuery(name = "Produto.findAll", query = "SELECT p FROM Produto p")})
public class Produto implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Column(name = "CODIGO")
    private Integer codigo;
    @Size(max = 50)
    @Column(name = "NOME")
    private String nome;
    @Column(name = "QUANTIDADE")
    private Integer quantidade;
    @JoinColumn(name = "COD_EMPRESA", referencedColumnName = "CODIGO")
    @ManyToOne(optional = false)
    private Empresa codEmpresa;

    // Restante do código omitido
```

java

```
@Entity
@Table(name = "EMPRESA")
@NamedQueries({
    @NamedQuery(name = "Empresa.findAll", query = "SELECT e FROM Empresa e")})
public class Empresa implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Column(name = "CODIGO")
    private Integer codigo;
    @Size(max = 50)
    @Column(name = "RAZAO_SOCIAL")
    private String razaoSocial;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "codEmpresa")
    private Collection produtoCollection;

    //Restante do código omitido
```

java

```
@Local
public interface ProdutoGestorLocal {
    List listaTodos();
}
```

```

java

@Stateless
public class ProdutoGestor implements ProdutoGestorLocal {

    @Override
    public List listaTodos() {
        EntityManagerFactory emf
            = Persistence.createEntityManagerFactory("EnterpriseApplication1-ejbPU2");
        EntityManager em = emf.createEntityManager();
        Query query = em.createNamedQuery("Produto.findAll");
        List lista = query.getResultList();
        em.close();
        return lista;
    }
}

```

```

java

@WebServlet(urlPatterns = {"/ListaProduto"})
public class ListaProduto extends HttpServlet {
    @EJB
    beans.ProdutoGestorLocal facade;
    protected void processRequest(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            /* TODO output your page here. You may use following sample code. */
            out.println("");
            out.println("");
            out.println("");
            out.println("");
            out.println("");
            out.println("");
            out.println("");
            List lista = facade.listaTodos();
            for (Produto produto : lista) {
                out.println("

" + produto.getNome() + " - Empresa: " +
produto.getCodEmpresa().getRazaoSocial() + "
");
            }
            out.println("");
            out.println("");
        }
    }
    // Códito omitido.
}

```

**Faça você mesmo!**

Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

ID DA QUESTÃO FORA DE PADRÃO... VERIFICAR A QUESTÃO INTEIRA

## Padrão Front Controller

O objetivo do padrão Front Controller é a centralização das chamadas em um único ponto de acesso, direcionando os resultados para a interface correta. A implementação do padrão Front Controller deve ocorrer na camada View, pois lida apenas com conversão de formatos, fluxo de chamadas e redirecionamentos, isolado das regras de negócio.

Neste vídeo, você verá o padrão Front Controller. Além disso, será abordada a centralização de ações e chamadas por meio de servlets, de forma a utilizar apenas uma classe para gerenciar as chamadas e controlar as ações subsequentes no sistema.



### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

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

```
sql

CREATE TABLE EMPRESA (
    CODIGO INT NOT NULL PRIMARY KEY,
    RAZAO_SOCIAL VARCHAR(50));

CREATE TABLE DEPARTAMENTO (
    CODIGO INT NOT NULL PRIMARY KEY,
    NOME VARCHAR(50),
    COD_EMPRESA INT NOT NULL);

ALTER TABLE DEPARTAMENTO ADD FOREIGN KEY(COD_EMPRESA)
REFERENCES EMPRESA(CODIGO);

CREATE TABLE SERIAIS (
    NOME_TABELA VARCHAR(50) NOT NULL PRIMARY KEY,
    VALOR_CHAVE INT);

INSERT INTO SERIAIS VALUES ('EMPRESA',0);
INSERT INTO SERIAIS VALUES ('DEPARTAMENTO',0);
```

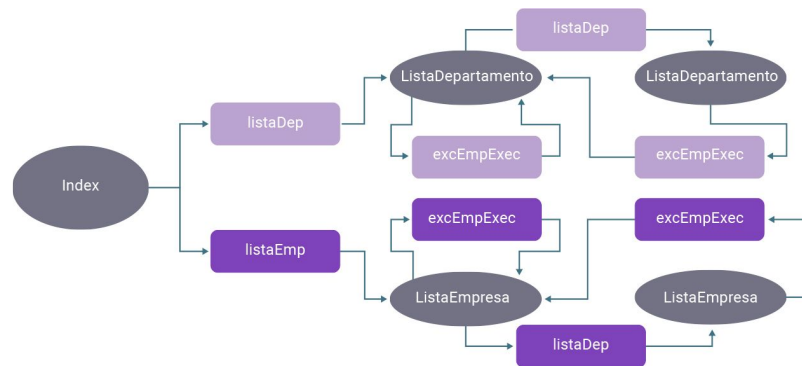
Nesse projeto, temos as tabelas EMPRESA e DEPARTAMENTO, para a persistência de dados de um cadastro simples, com um relacionamento por meio do campo COD\_EMPRESA, da tabela DEPARTAMENTO. Também podemos observar uma terceira tabela, com o nome SERIAIS, que viabilizará o autoincremento, através de anotações do JPA. Para a interface de nosso sistema, utilizaremos cinco páginas, descritas a seguir.

- **index.html:** página inicial do aplicativo cadastral de exemplo, com acesso às listagens de empresas e departamentos.
- **ListaEmpresa.jsp:** listagem das empresas e acesso às opções oferecidas para inclusão e exclusão de empresas.
- **DadosEmpresa.jsp:** entrada de dados da empresa que será incluída.
- **ListaDepartamento.jsp:** listagem dos departamentos e acesso às opções de inclusão e exclusão de departamentos.



- **DadosDepartamento.jsp**: entrada de dados do departamento que será incluído.

Agora, vamos definir os fluxos das chamadas, que ocorrerão a partir do HTTP, o que pode ser representado por meio de uma **Rede de Petri**, uma ferramenta que permite expressar os **estados** de um sistema. Ao modelar um sistema web, os estados definem **páginas** e as transições são chamadas HTTP.



Rede de Petri com as transições da aplicação.

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

Como as chamadas HTTP utilizam parâmetros, com valores do tipo texto, vamos adotar um parâmetro para identificação da transição requerida, no qual 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.

#### listaDep

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

#### listaEmp

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

## excDepExec

---

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

## excEmpExec

---

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

- 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
- 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

- 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, por meio dos métodos **doGet** ou **doPost**, execução de operações que envolvam chamadas aos **EJBs**, relacionadas às atividades de consulta ou persistência, e redirecionamento para uma página, normalmente JSP.

## Atividade 1

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

**QUESTÃO FORA DE PADRÃO... NÃO FOI POSSÍVEL MIGRAR O ENUNCIADO**

A

O NetBeans possui todas as bibliotecas necessárias ao projeto.

B

Caso o projeto necessite de bibliotecas externas, deve ser executado apenas no servidor.

C

Não há necessidade de preparar o servidor, pois o ambiente de produção possui as bibliotecas necessárias.

D

É necessário incluir as bibliotecas no projeto para garantir seu funcionamento.

E

Sem a inclusão das bibliotecas, o sistema funciona apenas no servidor.



A alternativa D está correta.

Para que o programa funcione de maneira correta, precisamos incluir as bibliotecas externas. Portanto o ambiente local e o servidor de produção devem possuir as dependências.

## Camadas Model e Controller

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

Neste vídeo, você vai conferir as camadas Model e Controller do padrão MVC, e verá como a camada Model gerencia dados e regras de negócio, enquanto o Controller coordena a interação entre usuário e Model. Por fim, acompanhe também a separação de responsabilidades e benefícios da arquitetura MVC.



Conteúdo interativo

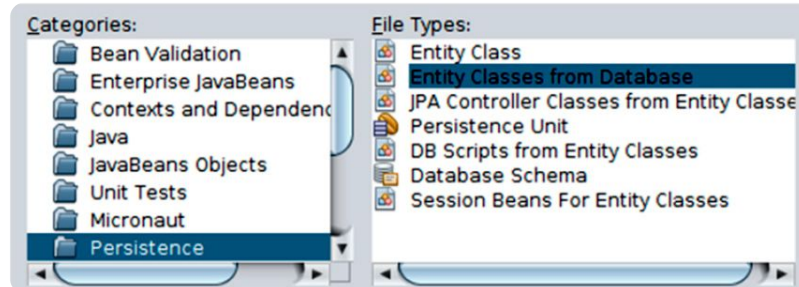
Acesse a versão digital para assistir ao vídeo.

## Criação da camada Model

Para a criação dessa camada, utilizaremos as ferramentas de automação do NetBeans e, para gerar as entidades do sistema, iremos executar os passos apresentadas a seguir:

### Passo 1

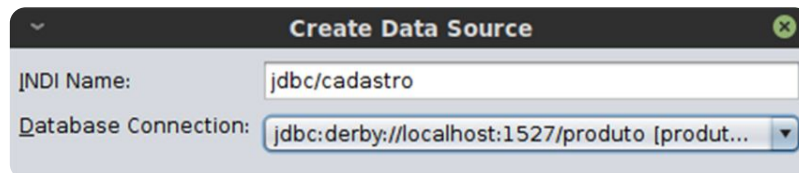
Adicionar novo arquivo, escolhendo Entity Classes from Database, na categoria Persistence.



"Wizard" de criação de arquivo do NetBeans.

### Passo 2

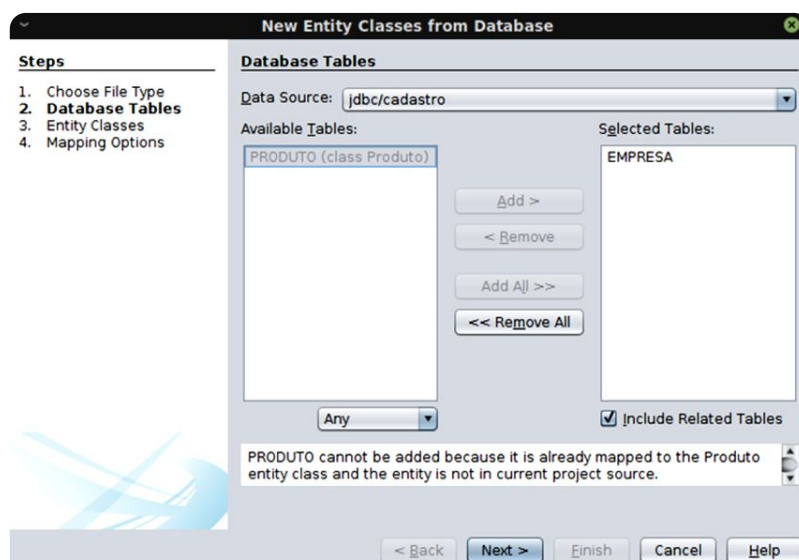
Na configuração de Data Source, escolha New Data Source, com a definição do nome JNDI (jdbc/cadastro), e escolha da conexão para o banco de dados.



"Wizard" de criação de entidades do NetBeans.

### Passo 3

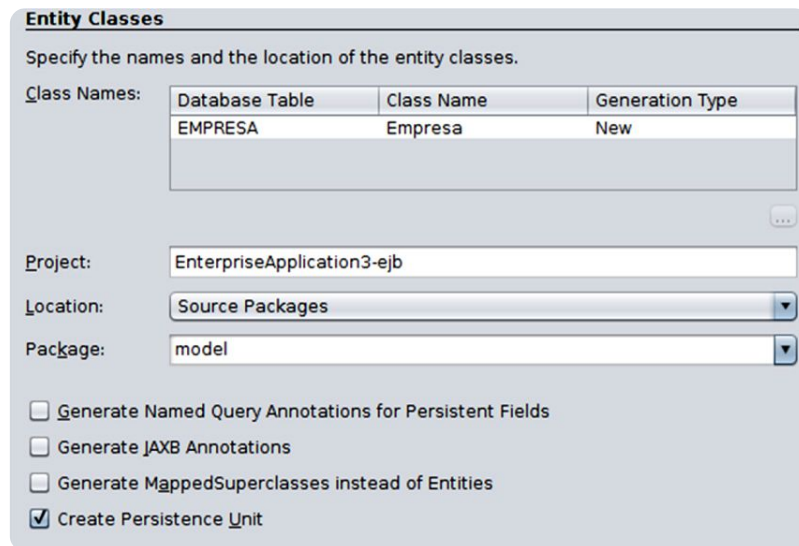
Escolha as tabelas DEPARTAMENTO e EMPRESA, deixando marcada a opção de inclusão das tabelas relacionadas.



"Wizard" de criação de fonte de dados do NetBeans.

### Passo 4

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



The "Entity Classes" wizard screen in NetBeans. It has a title bar "Entity Classes" and a subtitle "Specify the names and the location of the entity classes." Below this is a table for "Class Names" with columns "Database Table", "Class Name", and "Generation Type". The first row contains "EMPRESA", "Empresa", and "New". Below the table is a "Project:" field with "EnterpriseApplication3-ejb", a "Location:" dropdown menu with "Source Packages", and a "Package:" dropdown menu with "model". At the bottom are four checkboxes: "Generate Named Query Annotations for Persistent Fields", "Generate JAXB Annotations", "Generate MappedSuperclasses instead of Entities", and "Create Persistence Unit" (which is checked).

Database Table	Class Name	Generation Type
EMPRESA	Empresa	New

Project: EnterpriseApplication3-ejb  
Location: Source Packages  
Package: model

☐ Generate Named Query Annotations for Persistent Fields  
☐ Generate JAXB Annotations  
☐ Generate MappedSuperclasses instead of Entities  
☒ Create Persistence Unit

"Wizard" de criação de entidades do NetBeans.

## Passo 5

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



The final screen of the "Entity" wizard. It has two dropdown menus: "Association Fetch:" set to "default" and "Collection Type:" set to "java.util.List". Below these are five checkboxes, all of which are unchecked: "Fully Qualified Database Table Names", "Attributes for Regenerating Tables", "Use Column Names in Relationships", "Use Defaults if Possible", and "Generate Fields for Unresolved Relationships". A large blue circle with the number "5" is overlaid on the right side of the screen.

Association Fetch: default  
Collection Type: java.util.List

☐ Fully Qualified Database Table Names  
☐ Attributes for Regenerating Tables  
☐ Use Column Names in Relationships  
☐ Use Defaults if Possible  
☐ Generate Fields for Unresolved Relationships

"Wizard" de criação de entidades do NetBeans.

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.

```

java

@Entity
@Table(name = "DEPARTAMENTO")
@NamedQueries({
    @NamedQuery(name = "Departamento.findAll",
        query = "SELECT d FROM Departamento d")})
public class Departamento implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @TableGenerator(name = "DeptoTabGen", table = "SERIAIS",
        pkColumnName = "NOME_TABELA",
        pkColumnValue = "DEPARTAMENTO",
        valueColumnName = "VALOR_CHAVE")
    @GeneratedValue(strategy = GenerationType.TABLE,
        generator = "DeptoTabGen")
    @Basic(optional = false)
    @NotNull
    @Column(name = "CODIGO")
    private Integer codigo;
    // O restante do código foi omitido por não ser relevante
    // para a modificação efetuada
}

@Entity
@Table( name = "EMPRESA" )
@NamedQueries({
    @NamedQuery(name = "Empresa.findAll",
        query = "SELECT e FROM Empresa e")})
public class Empresa implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @TableGenerator(name = "EmpTabGen", table = "SERIAIS",
        pkColumnName = "NOME_TABELA",
        pkColumnValue = "EMPRESA",
        valueColumnName = "VALOR_CHAVE")
    @GeneratedValue(strategy = GenerationType.TABLE,
        generator = "EmpTabGen")
    @Basic(optional = false)
    @NotNull
    @Column(name = "CODIGO")
    private Integer codigo;

    // O restante do código foi omitido por não ser relevante
    // para a modificação efetuada
}

```

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.

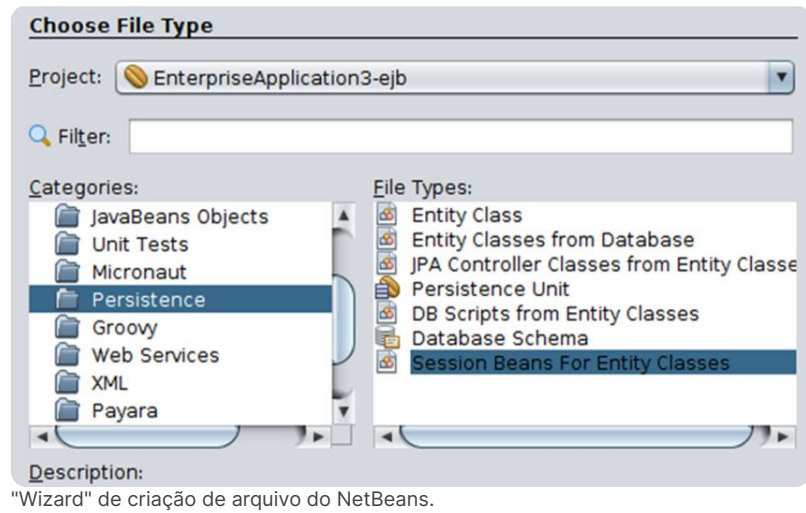
Durante a execução, quando uma entidade é criada e persistida no banco de dados, o gerador é acionado, incrementa o valor do campo `VALOR_CHAVE` para a linha da tabela `SERIAIS` referente à entidade, e alimenta a chave primária com o novo valor. Existem outras estratégias de geração, como o uso de `SEQUENCE`,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

## Criação da camada Controller

Com a camada Model pronta, vamos criar a camada Controller, usando componentes do tipo EJB, de acordo com os seguintes passos:

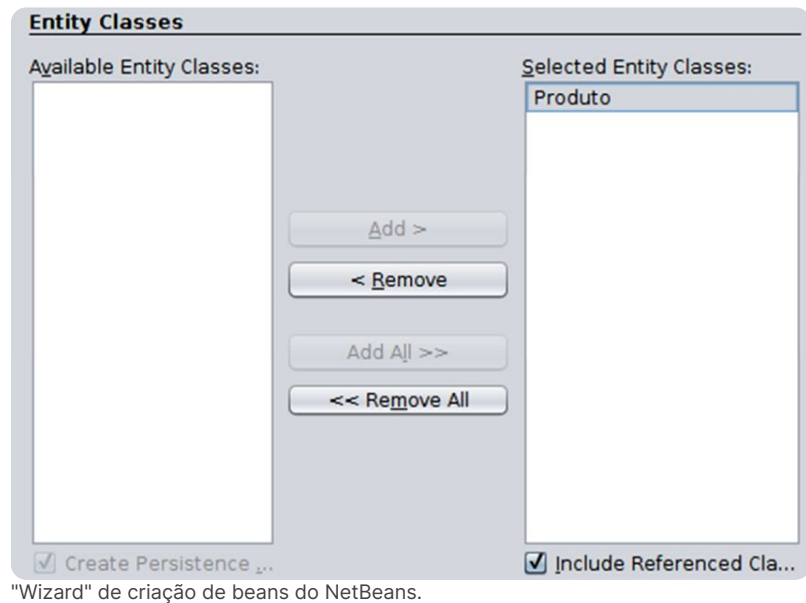
## Passo 1

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



## Passo 2

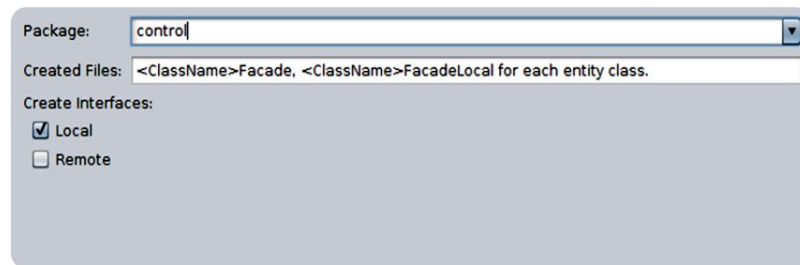
Selecionar todas as entidades do projeto.



## Passo 3

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





"Wizard" de criação de beans do NetBeans.

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.

```
java

public abstract class AbstractFacade {
    private Class entityClass;
    public AbstractFacade(Class entityClass) {
        this.entityClass = entityClass;
    }
    protected abstract EntityManager getEntityManager();
    public void create(T entity) {
        getEntityManager().persist(entity);
    }
    public void edit(T entity) {
        getEntityManager().merge(entity);
    }
    public void remove(T entity) {
        getEntityManager().remove(getEntityManager().merge(entity));
    }
    public T find(Object id) {
        return getEntityManager().find(entityClass, id);
    }
    public List findAll() {
        javax.persistence.criteria.CriteriaQuery cq =
            getEntityManager().getCriteriaBuilder().createQuery();
        return getEntityManager().createQuery(cq).getResultList();
    }
    public List< T > findRange(int[] range) {
        javax.persistence.criteria.CriteriaQuery cq =
            getEntityManager().getCriteriaBuilder().createQuery();
        cq.select(cq.from(entityClass));
        javax.persistence.Query q =
            getEntityManager().createQuery(cq);
        q.setMaxResults(range[1] - range[0] + 1);
        q.setFirstResult(range[0]);
        return q.getResultList();
    }
    public int count() {
        javax.persistence.criteria.CriteriaQuery cq =
            getEntityManager().getCriteriaBuilder().createQuery();
        javax.persistence.criteria.Root< T > rt =
            cq.from(entityClass);
        cq.select(
            getEntityManager().getCriteriaBuilder().count(rt));
        javax.persistence.Query q =
            getEntityManager().createQuery(cq);
        return ((Long) q.getSingleResult()).intValue();
    }
}
```

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.

### Métodos create, edit e remove

---

Esses métodos são voltados para as ações de inclusão, edição e exclusão da entidade, são implementados por meio da invocação dos métodos de EntityManager, semelhante aos nossos exemplos anteriores, utilizando tecnologia JPA, bem como find, que retorna uma entidade a partir de sua chave primária.

### Métodos findAll e findRange

---

Esses métodos utilizam o JPA no modo programado, com base em CriteriaQuery, que apresenta métodos para substituir o uso de comandos na sintaxe JPQL.

### Método from

---

Esse método, que tem como parâmetro a classe da entidade, combinada com o método select, permite efetuar uma consulta que retorna todas as entidades do tipo gerenciado a partir do banco de dados. Entretanto, em findRange, temos ainda o recurso de paginação, com a definição do índice inicial (setFirstResult) e a quantidade de entidades (setMaxResults).

### Método count

---

Esse método 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.

## Construção dos session beans

Com a definição do modelo funcional genérico, a construção dos session beans se torna muito simples, com base na herança e uso de anotações.

```

java

@Stateless
public class DepartamentoFacade extends
AbstractFacade< Departamento > implements
DepartamentoFacadeLocal {
    @PersistenceContext(unitName = "CadastroEJB-ejbPU")
    private EntityManager em;

    @Override
    protected EntityManager getEntityManager() {
        return em;
    }
    public DepartamentoFacade() {
        super(Departamento.class);
    }
}

@Stateless
public class EmpresaFacade extends
AbstractFacade< Empresa > implements
EmpresaFacadeLocal {
    @PersistenceContext(unitName = "CadastroEJB-ejbPU")
    private EntityManager em;

    @Override
    protected EntityManager getEntityManager() {
        return em;
    }

    public EmpresaFacade() {
        super(Empresa.class);
    }
}

```

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

```

java
@Local
public interface DepartamentoFacadeLocal {
    void create(Departamento departamento);
    void edit(Departamento departamento);
    void remove(Departamento departamento);
    Departamento find(Object id);
    List findAll();
    List findRange(int[] range);
    int count();
}

@Local
public interface EmpresaFacadeLocal {
    void create(Empresa empresa);
    void edit(Empresa empresa);
    void remove(Empresa empresa);
    Empresa find(Object id);
    List< Empresa > findAll();
    List< Empresa > findRange(int[] range);
    int count();
}

```

Observe 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 as interfaces relacionadas naturalmente.

Para que o controle transacional ocorra da forma indicada, temos o atributo `transaction-type` com valor `JTA`, no arquivo `persistence.xml`. Entretanto, 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.

```
python
```

```
jdbc/cadastro  
false
```

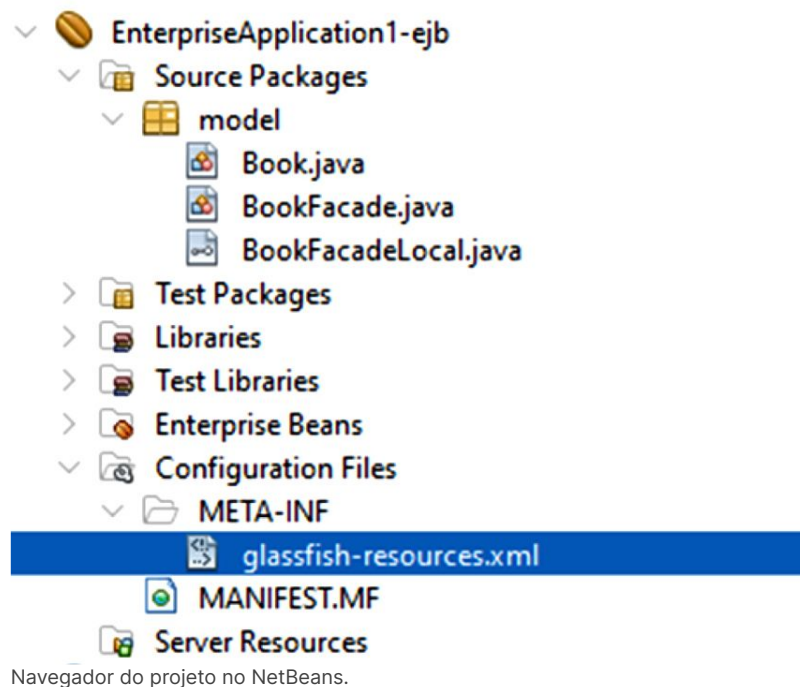
Caso ocorra um erro na implantação mesmo após a alteração dos arquivos, execute o programa asadmin.

Em seguida, invoque o comando **add-resources**, com a passagem do nome completo do arquivo **glassfish-resources.xml**.

```
C:\glassfish\bin>asadmin  
Use "exit" to exit and "help" for online help.  
asadmin> add-resources C:/Users/alexandre/testes/glassfish-resources.xml  
JDBC connection pool derby_net_bancoJPA_bancoJPAPool created successfully.  
JDBC resource jdbc/cadastro created successfully.  
Command add-resources executed successfully.  
asadmin>
```

Execução de comandos no shell do 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.



## Atividade 2

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

ID DA QUESTÃO FORA DE PADRÃO... VERIFICAR A QUESTÃO INTEIRA

## Camada View

A camada View no modelo MVC é responsável por apresentar as informações aos usuários e fornecer a interface com a qual eles interagem. É na camada View que são definidas as páginas, formulários e elementos visuais que compõem a interface do usuário. Ela desacopla a lógica de apresentação da lógica de negócios, permitindo a reutilização e a manutenção separada das duas partes do sistema.

Neste vídeo, você descobrirá como a Camada View atua no processo de interface dos usuários e interação, exibindo dados de forma compreensível e permitindo a interação do usuário. Entenda sua importância na arquitetura MVC.



### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

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.

python

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

Razão Social:

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.

python

```
<%@page import="model.Empresa"%>
<%@page import="java.util.List"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
```

Empresa:

```
<%
    List lista = (List)
    request.getAttribute("listaEmp");
    for(Empresa e: lista){
%>

    <% } %>
```

Nome:

A coleção de empresas será recuperada a partir de um atributo de requisição, com o nome listaEmp, por meio 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 vamos criar o primeiro arquivo para listagem, com o nome ListaEmpresa.jsp.

python

```
<%@page import="model.Empresa"%>
<%@page import="java.util.List"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
```

[Nova Empresa](#)

```
<%
    List lista = (List)
    request.getAttribute("lista");
    for(Empresa e: lista){
    %>

    <% } %>
```

Código	Razão Social	Opções
<%=e.getCodigo()%>	<%=e.getRazaoSocial()%>	<a href="#">Excluir</a>

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 com 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. Essa exclusão será efetuada com a chamada para CadastroFC, com 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.



python

```
<%@page import="model.Departamento"%>
<%@page import="java.util.List"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
```

[Novo Departamento](#)

```
<%
    List lista = (List)
    request.getAttribute("lista");
    for(Departamento d: lista){
    %>

    % } %>
```

Código	Nome	Empresa	Opções
<%=d.getCodigo() %>	<%=d.getNome() %>	<%=d.getEmpresa().getRazaoSocial() %>	<a href="#">Excluir</a>

Em termos práticos, a listagem de departamentos é muito semelhante à de empresas, com a definição de um link de inclusão, contando com o valor incDep para acao e a exibição dos dados por meio 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, com o valor excDepExec no parâmetro acao e o código do departamento corrente no parâmetro cod. Observe como a razão social da empresa é facilmente recuperada utilizando o atributo presente na entidade departamento, alimentado por meio de uma anotação **ManyToOne**. Com a modificação de index.html, terminamos a construção das interfaces de usuário.

python

[Listagem de Departamentos](#)

[Listagem de Empresas](#)

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.

## Atividade 3

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

ID DA QUESTÃO FORA DE PADRÃO... VERIFICAR A QUESTÃO INTEIRA

## Implementação do Front Controller

Em Java, implementar um Front Controller permite a centralização do processamento de todas as requisições recebidas pelo sistema. Esse componente atua como um ponto de entrada, gerenciando as requisições para os respectivos controladores. Assim obtemos a modularidade, a reutilização e a facilidade de manutenção, além de possibilitar o tratamento de tarefas comuns a todas as requisições, como autenticação e controle de acesso.

Neste vídeo, você verá como implementar o padrão Front Controller em Java usando o filtro. Descubra como centralizar o controle do fluxo de requisições e executar ações comuns antes de processar as solicitações específicas. Aprenda a otimizar o desenvolvimento de aplicações web.



### Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

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.

```
java

public abstract class Strategy {
    protected final K facade;
    public Strategy(K facade) {
        this.facade = facade;
    }
    public abstract String executar(String acao,
        HttpServletRequest request);
}
```

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

Em seguida, definimos a estratégia para empresas, herdando da classe Strategy, com a adoção da interface EmpresaFacadeLocal.

```

java

public class EmpresaStrategy
    extends Strategy< EmpresaFacadeLocal> {
    public EmpresaStrategy(EmpresaFacadeLocal facade) {
        super(facade);
    }
    @Override
    public String executar(String acao,
        HttpServletRequest request) {
        String paginaDestino = "ListaEmpresa.jsp";
        switch(acao){
            case "listaEmp":
                request.setAttribute("lista", facade.findAll());
                break;
            case "excEmpExec":
                Integer codigo =
                    new Integer(request.getParameter("cod"));
                facade.remove(facade.find(codigo));
                request.setAttribute("lista", facade.findAll());
                break;
            case "incEmpExec":
                String razaoSocial =
                    request.getParameter("razao_social");
                Empresa empresa = new Empresa();
                empresa.setRazaoSocial(razaoSocial);
                facade.create(empresa);
                request.setAttribute("lista", facade.findAll());
                break;
            case "incEmp":
                paginaDestino = "DadosEmpresa.jsp";
                break;
        }
        return paginaDestino;
    }
}

```

Durante a execução, temos a definição da página de destino como ListaEmpresa.jsp, que será diferente apenas quando o parâmetro acao tiver valor incEmp, que temos a página DadosEmpresa.jsp como destino. 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, obtida a partir do método findAll do facade, permitindo 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 por meio do atributo facadeEmpresa, do tipo EmpresaFacadeLocal. Note que o construtor deve ser modificado para receber as duas interfaces utilizadas.

java

```
public class DepartamentoStrategy
    extends Strategy< DepartamentoFacadeLocal>{
    private final EmpresaFacadeLocal empresaFacade;
    public DepartamentoStrategy(DepartamentoFacadeLocal facade,
        EmpresaFacadeLocal empresaFacade) {
        super(facade);
        this.empresaFacade = empresaFacade;
    }
    @Override
    public String executar(String acao,
        HttpServletRequest request) {
        String paginaDestino = "ListaDepartamento.jsp";
        switch(acao){
            case "listaDep":
                request.setAttribute("lista", facade.findAll());
                break;
            case "excDepExec":
                Integer codigo = new
                    Integer(request.getParameter("cod"));
                facade.remove(facade.find(codigo));
                request.setAttribute("lista", facade.findAll());
                break;
            case "incDepExec":
                String nome = request.getParameter("nome");
                Integer codEmpresa = new Integer(
                    request.getParameter("cod_empresa"));
                Empresa empresa = empresaFacade.find(codEmpresa);
                Departamento departamento = new Departamento();
                departamento.setNome(nome);
                departamento.setEmpresa(empresa);
                facade.create(departamento);
                request.setAttribute("lista", facade.findAll());
                break;
            case "incDep":
                request.setAttribute("listaEmp",
                    empresaFacade.findAll());
                paginaDestino = "DadosDepartamento.jsp";
                break;
        }
        return paginaDestino;
    }
}
```

A estratégia para departamentos é um pouco mais complexa, o que exige 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, em que a página de destino utilizada será DadosDepartamento.jsp. Veja o passo a passo:

## 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, além da 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, 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.

Faremos da seguinte forma:

```
java

@WebServlet(name="CadastroFC", urlPatterns={"/CadastroFC"})
public class CadastroFC extends HttpServlet {
    @EJB
    EmpresaFacadeLocal empresaFacade;
    @EJB
    DepartamentoFacadeLocal departamentoFacade;

    private final HashMap< String,Strategy> estrategia =
        new HashMap<>();
    private final HashMap< String,String> acoes = new HashMap<>();

    @Override
    public void init() throws ServletException {
        super.init();
        estrategia.put("empresa", new EmpresaStrategy(
            empresaFacade));
        estrategia.put("departamento", new DepartamentoStrategy(
            departamentoFacade,empresaFacade));
        String[] acoesEmpresa =
            {"listaEmp","incEmp","incEmpExec","excEmpExec"};
        for(String acao: acoesEmpresa)
            acoes.put(acao, "empresa");
        String[] acoesDeppto =
            {"listaDep","incDep","incDepExec","excDepExec"};
        for(String acao: acoesDeppto)
            acoes.put(acao, "departamento");
    }

    protected void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        String acao = request.getParameter("acao");
        if(acao==null)
            throw new ServletException("Parâmetro acao requerido");
        Strategy st = estrategia.get(acoes.get(acao));
        request.getRequestDispatcher(st.executar(acao, request)).
            forward(request, response);
    }

    @Override
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }

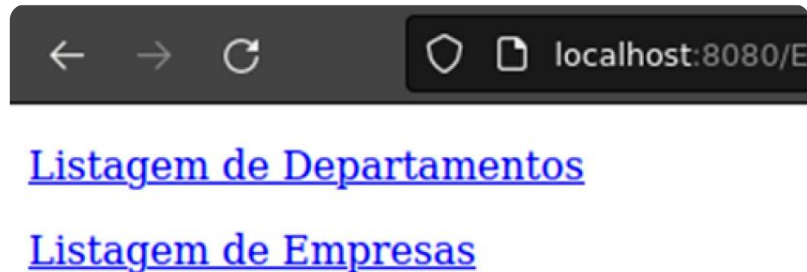
    @Override
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        processRequest(request, response);
    }
}
```

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 estratégia, 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.

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 inicial da aplicação.

Temos a tela de cadastro:



Tela de cadastro de empresas.

Temos a tela de listagem:



Tela listagem de empresas.

## Atividade 4

### Questão 1

HOUE UM ERRO AO MIGRAR ESTA ATIVIDADE:

ID DA QUESTÃO FORA DE PADRÃO... VERIFICAR A QUESTÃO INTEIRA

# Aplicação MVC com padrão Front Controller

O exercício que veremos no roteiro de prática a seguir permitirá que você aplique os conceitos do padrão Front Controller e do padrão MVC na implementação de uma aplicação web em Java. Lembre-se de seguir as boas práticas de programação e a separação de responsabilidades entre as camadas Model, View e Controller.

Neste vídeo, você verá como implementar o padrão Front Controller utilizando filtro no Java, centralizar o controle das requisições, direcioná-las para os controladores apropriados e retornar as respostas corretas. Uma abordagem prática e eficiente para o desenvolvimento de aplicações web.



## Conteúdo interativo

Acesse a versão digital para assistir ao vídeo.

## Roteiro de prática

Vamos simular a criação de uma aplicação web simples para uma livraria on-line. Seu objetivo é implementar o padrão Front Controller e o padrão MVC para essa aplicação. Para isso, é necessário seguir algumas etapas:

1. Implemente a camada de modelo utilizando ejbs. Para simplificar, utilize uma coleção de livros estática no método "findAll".
2. Crie as seguintes classes:
  - FrontController: classe responsável por receber todas as requisições e direcioná-las para o controlador apropriado.
  - Controller: interface que define os métodos que os controladores devem implementar.
  - HomeController: controlador responsável por lidar com as requisições relacionadas à página inicial.
  - BookController: controlador responsável por lidar com as requisições relacionadas aos livros.
4. Defina as possíveis ações da aplicação no Front Controller, mapeando as ações para os controladores adequados. Por exemplo: "acao=welcome": direciona para o HomeController. Rota " acao=books": direciona para o BookController.
5. Implemente as classes Controller (HomeController e BookController) de acordo com a interface Controller, definindo os métodos necessários para lidar com as requisições.
6. Teste sua implementação chamando os seguintes endereços: <http://localhost:8080/EnterpriseApplication1-war/FrontController?acao=welcome> <http://localhost:8080/EnterpriseApplication1-war/FrontController?acao=books>

1. Implemente a camada de modelo utilizando ejbs. Para simplificar, utilize uma coleção de livros estática no método "findAll".
2. Crie as seguintes classes:
  - FrontController: classe responsável por receber todas as requisições e direcioná-las para o controlador apropriado.
  - Controller: interface que define os métodos que os controladores devem implementar.
  - HomeController: controlador responsável por lidar com as requisições relacionadas à página inicial.
  - BookController: controlador responsável por lidar com as requisições relacionadas aos livros.
5. Defina as possíveis ações da aplicação no Front Controller, mapeando as ações para os controladores adequados. Por exemplo: "acao=welcome": direciona para o HomeController. Rota " acao=books": direciona para o BookController.
6. Implemente as classes Controller (HomeController e BookController) de acordo com a interface Controller, definindo os métodos necessários para lidar com as requisições.

7. Teste sua implementação chamando os seguintes endereços: <http://localhost:8080/EnterpriseApplication1-war/FrontController?acao=welcome> <http://localhost:8080/EnterpriseApplication1-war/FrontController?acao=books>

Veja o resultado dessa prática.

## Resultado

---

Classe FrontController:  
Interface Controller:  
Classe HomeController:  
Classe BookController:  
Classe Book

java

```
@WebServlet(name = "FrontController", urlPatterns = {"/FrontController"})
public class FrontController extends HttpServlet {

    static final Map controllers = new HashMap();

    static {
        controllers.put("welcome", new HomeController());
        controllers.put("books", new BookController());
    }

    protected void processRequest(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {

        String path = request.getParameter("acao");
        System.out.println(path);
        Controller controller = controllers.get(path);
        if (controller != null) {
            controller.handleRequest(request, response);
        } else {
            response.sendError(HttpServletResponse.SC_NOT_FOUND);
        }
    }
}
```

java

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public interface Controller {
    void handleRequest(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException;
}
```



java

```
public class HomeController implements Controller {
    @Override
    public void handleRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.sendRedirect("welcome.jsp");
    }
}
```

java

```
public class BookController implements Controller {

    @EJB
    private model.Book facade;
    @Override
    public void handleRequest(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        List books = facade.findAll();
        request.setAttribute("books", books);
        request.getRequestDispatcher("books.jsp").forward(request, response);
    }
}
```

java

```
public class Book {
    private String title;
    private String author;
    private int year;

    public Book(String title, String author, int year) {
        this.title = title;
        this.author = author;
        this.year = year;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    public int getYear() {
        return year;
    }

    public static List findAll() {
        List books = new ArrayList<>();
        books.add(new Book("Book 1", "Author 1", 2020));
        books.add(new Book("Book 2", "Author 2", 2018));
        books.add(new Book("Book 3", "Author 3", 2022));
        return books;
    }
}
```

java

```
@WebServlet(name = "FrontController", urlPatterns = {"/FrontController"})
public class FrontController extends HttpServlet {

    static final Map controllers = new HashMap();

    static {
        controllers.put("welcome", new HomeController());
        controllers.put("books", new BookController());
    }

    protected void processRequest(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {

        String path = request.getParameter("acao");
        System.out.println(path);
        Controller controller = controllers.get(path);
        if (controller != null) {
            controller.handleRequest(request, response);
        } else {
            response.sendError(HttpServletResponse.SC_NOT_FOUND);
        }
    }
}
```

java

```
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public interface Controller {
    void handleRequest(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException;
}
```

java

```
public class HomeController implements Controller {
    @Override
    public void handleRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
        response.sendRedirect("welcome.jsp");
    }
}
```

java

```
public class BookController implements Controller {

    @EJB
    private model.Book facade;
    @Override
    public void handleRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
        List books = facade.findAll();
        request.setAttribute("books", books);
        request.getRequestDispatcher("books.jsp").forward(request, response);
    }
}
```

java

```
public class Book {
    private String title;
    private String author;
    private int year;

    public Book(String title, String author, int year) {
        this.title = title;
        this.author = author;
        this.year = year;
    }

    public String getTitle() {
        return title;
    }

    public String getAuthor() {
        return author;
    }

    public int getYear() {
        return year;
    }

    public static List findAll() {
        List books = new ArrayList<>();
        books.add(new Book("Book 1", "Author 1", 2020));
        books.add(new Book("Book 2", "Author 2", 2018));
        books.add(new Book("Book 3", "Author 3", 2022));
        return books;
    }
}
```

## Faça você mesmo!

### Questão 1

HOUVE UM ERRO AO MIGRAR ESTA ATIVIDADE:

ID DA QUESTÃO FORA DE PADRÃO... VERIFICAR A QUESTÃO INTEIRA

## Considerações finais

- Utilização da Java Persistence API
- Ecossistema Java EE
- Estrutura de um aplicativo corporativo
- Utilização de Enterprise Java Beans
- Arquitetura MVC (Model-View-Controller)
- Arquitetura MVC (Model-View-Controller)

## Explore +

Leia o artigo da Oracle comparando Entity Beans CMP e JPA. Disponível na página da Oracle.

Leia o tutorial de Java Persistence API, disponível na página Vogella

Pesquise a documentação da Oracle sobre componentes do tipo EJB.

Pesquise a documentação interativa acerca de padrões de desenvolvimento, disponível na página Refactoring Guru.

O EclipseLink está disponível para download na página da Eclipse Foundation.

## Referências

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

CORNELL, G.; HORSTMANN, C. **Core JAVA**. 8ª 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. 8ª 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**. 5ª ed. USA: O'Reilly, 2006.

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