

Disciplina: Desenvolvimento de Software

Aula 8: Arquitetura MVC

Apresentação

Quando começamos a criar um sistema, devemos pensar em sua arquitetura, ou seja, na forma como os componentes serão organizados e poderão se comunicar na resolução de problemas.

Os padrões arquiteturais, como Broker e MVC, formalizam todo o fluxo de informações internas, bem como a comunicação do sistema com o meio externo, e, entre os diversos padrões existentes, o MVC acabou se destacando no mercado de softwares cadastrais, particularmente na Web.

Essa arquitetura viabilizou a adoção de alguns padrões de desenvolvimento, como Front Control e DAO, o que possibilitou definir diversas ferramentas para aumentar a produtividade nesses ambientes, a exemplo de tecnologias como JSF, Spring e Hibernate.

Objetivos

- Explicar os principais padrões arquiteturais;
- Identificar as características e benefícios do MVC;
- Usar MVC com Front Control em sistema Java Web.

Padrão arquitetural

Os **padrões de desenvolvimento** objetivam o reúso do conhecimento, trazendo modelos padronizados de soluções para problemas já conhecidos no mercado.

Utilizando esses padrões, temos uma visão mais estratégica para o projeto do sistema, facilitando a manutenção do código e evitando problemas já conhecidos devido a experiências anteriores de desenvolvimento.

Exemplo

Um exemplo simples de padrão com larga aceitação é o DAO, referente à concentração das operações de acesso a banco de dados, o que evita a multiplicação de comandos SQL ao longo de todo o código.

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

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

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

A padronização das arquiteturas de software permite que diversos produtos sejam criados para generalizar a parte comum dessas soluções, exigindo apenas a especialização de cada ferramental para a aplicação de nossas próprias regras de negócio, como no caso de frameworks como JSF, Spring e Hibernate.


O padrão arquitetural **Event-Driven** também é muito importante no mundo corporativo. Baseado em evento, o Event-Driven permite que sejam sequenciados diversos pedidos para atendimento de forma assíncrona, e o MOM é um típico exemplo desse tipo de arquitetura. As solicitações são enviadas para filas de mensagens com a finalidade de ser processadas posteriormente, sem bloquear o cliente.

Podemos observar uma classificação de arquiteturas na tabela seguinte.

Arquitetura MVC

Entre os diversos padrões arquiteturais existentes, o Model-View-Control, ou **MVC**, acabou dominando o mercado de desenvolvimento no que se refere às aplicações cadastrais.

O MVC promove a divisão do sistema em três camadas:

 Clique nos botões para ver as informações.

Persistência de dados (Model)

Na **primeira versão** do MVC, a camada **Model** era constituída de entidades, as quais gravavam seus próprios estados no banco, segundo o padrão de desenvolvimento **Active Record**. Esse modelo, inclusive, foi adotado pelos **Entity Beans** do J2EE, que acabaram se mostrando muito ineficientes, razão pela qual foram substituídos pelo JPA. Como a entidade continha os métodos de acesso ao banco, os dados eram transitados no padrão **VO**, que trazia os mesmos dados das entidades, mas sem métodos de persistência, impedindo a manipulação do banco pela **View**.

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

Processos de negócios (Control)

Acima da camada Model encontraremos a camada **Control**, onde serão implementadas as diversas regras de negócio de nosso sistema de forma **completamente independente** de qualquer interface visual que venha a ser criada. A camada **Control** funcionará como intermediário entre as camadas **View** e **Model**, e deverá ser criada de forma que possa ser utilizada por qualquer tecnologia para a criação de interfaces sem a necessidade de mudanças no código. Significa dizer que a interface se adequa ao controle, nunca o contrário.

Interação com o usuário (View)

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

Podemos observar, na figura seguinte, uma concepção simples do MVC.

Na figura, temos a **camada Model** com as classes de entidade e DAO, além do uso de JDBC para acesso à base de dados, seja de forma direta, seja com a adoção de um framework de persistência, como Hibernate.

Logo acima temos a **camada de controle** com a implementação das regras de negócio. Os componentes dessa camada são os únicos que poderão instanciar elementos do tipo DAO.

Por fim, na última camada, vemos algumas tecnologias que podem ser utilizadas para a criação de interfaces dentro do universo Java, como Servlet, JSP, JSF e GUI do tipo awt ou swing.

As classes mais relevantes do padrão MVC podem ser representadas através de artefatos específicos, com uma simbologia já popularizada no mercado, como podemos observar na tabela seguinte.

Padrão Front Control

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

O padrão **Front Control** foi criado com o objetivo de concentrar as chamadas efetuadas pela interface e direcioná-las para os controladores corretos, além de direcionar a saída para a visualização correta ao término do processo.

Embora o nome possa nos enganar, esse padrão não pertence à camada Control, mas sim à camada **View**, pois não há relação com as regras de negócios, tendo como função primordial o simples controle de navegação do sistema.

Uma implementação desse padrão pode ser observada na figura seguinte.

Dentro da arquitetura Java Web, e seguindo o padrão MVC, a implementação de um **Front Control** é feita através de um **Servlet**, o qual deverá receber as chamadas HTTP e, de acordo com os parâmetros fornecidos, efetuar as chamadas corretas aos elementos da camada Control.

Tais elementos, normalmente, são classes de controle Java comuns ou componentes EJB recebendo em seguida as respostas desses processamentos e direcionando o resultado obtido para algum elemento, como um JSP, na camada View, para que construa a resposta HTML ou XML.

O framework JSF implementa nativamente o padrão Front Control, pois toda a interpretação dos Facelets e o controle do ciclo de vida das páginas são feitos através de um único Servlet, o qual é denominado **FacesServlet**.

Também indica uma arquitetura **MVC**, pois temos uma clara divisão entre elementos de acesso ao **Control**, no caso Managed Beans, e elementos da camada View, que são as páginas JSF.

Modelagem para Web

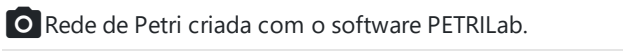
Uma metodologia de modelagem muito interessante para os sistemas Web é a adoção de **Redes de Petri**.

Uma metodologia de modelagem muito interessante para os sistemas Web é a adoção de Redes de Petri. Essa é uma técnica que surgiu na área de engenharia para a modelagem de processos paralelos em máquinas físicas, sendo adotada posteriormente na área de programação.

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

Como estamos criando a interface voltada para o ambiente Web, podemos considerar as páginas como os possíveis estados que nosso sistema pode assumir. As transições sempre serão efetuadas através de chamadas HTTP, ocorrendo a mudança da página visualizada, ou seja, do estado do sistema, através da resposta HTTP.

Na figura seguinte, podemos observar uma Rede de Petri para um sistema cadastral simples, focando a tecnologia Java Web.



Saiba mais

Antes de continuar seus estudos, conheça melhor o software [PETRILab](https://sourceforge.net/projects/petrilab/). <<https://sourceforge.net/projects/petrilab/>>

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

Toda e qualquer transição deverá passar por esse Servlet, de forma a recuperar os parâmetros fornecidos na requisição, efetuar as chamadas corretas aos componentes de controle, receber os dados processados, e encaminhá-los para a página correta. Toda essa orquestração é efetuada pelo FacesServlet no ambiente do JSF.

Ainda analisando o diagrama, podemos observar que os estados representam as páginas do sistema, como “index” e “ListaProd”.

O index começa com valor 1 e os demais estados com valor 0. Essa numeração indica que o index é o estado que começa ativo, e a cada transição o estado de origem assume valor 0 e o novo estado passa a valer 1, de forma similar à representação binária para “ligado” e “desligado”.

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



Saiba mais

Antes de continuar, conheça a versão teste de [Enterprise Architect](https://sparxsystems.com/products/ea/trial/request.html). <<https://sparxsystems.com/products/ea/trial/request.html>>

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

JSF e MVC

O framework **JSF** visa a fornecer um ambiente baseado em eventos, similar ao desktop, para o desenvolvimento de interfaces Java Web, mas internamente existe todo um comportamento voltado para a arquitetura **MVC**.

Claro que a melhor implementação de uma camada de controle seria aquela que garantisse a plena independência da interface visual, o que inclui as páginas JSF.

Contudo, boa parte dos desenvolvedores implementam o controle como Managed Beans, o que deixa a arquitetura como um todo menos expansível, devido à presença de anotações específicas para o contexto da interface visual.

Mas como é o ciclo de vida das páginas JSF?

Podemos observar, na figura seguinte, o resumo visual desse ciclo de vida.

Ao receber a requisição, tem início a fase **Restore View**, onde é montada toda a árvore de componentes JSF ao nível do servidor, indo em seguida para a fase **Apply Requests**, onde os valores digitados pelo usuário são aplicados aos componentes da árvore que foi montada.

Para manter a representação de uma **View** em memória, o JSF utiliza uma árvore de objetos, que se inicia em uma raiz do tipo **UIViewRoot**, e a partir da qual são vinculados os demais elementos, todos descendentes de **UIComponent**.

Ao final dessa segunda fase, todos os componentes estão com valores atualizados, e as mensagens e eventos estão enfileirados passando para a fase **Process Validations**. Nessa fase os validadores associados aos componentes são testados e, se ocorre um erro de validação, ocorre também o desvio para a fase **Render Response** com as mensagens de erro corretas.

Exemplo

Um exemplo simples de validador seria o campo aceitar apenas valores de 1 a 10.

Não ocorrendo erros, a fase **Update Model Values** é iniciada, efetuando-se a atualização dos valores dos **Managed Beans** a partir dos valores da árvore de componentes. Ocorrendo algum erro de conversão, ocorrerá o desvio para **Render Response**.

Caso o processo de atualização transcorra sem problemas, a próxima fase, denominada **Invoke Application**, é iniciada, sendo tratados de forma ordenada quaisquer eventos ou comandos solicitados.

Finalmente, na fase **Render Response**, a página de resposta é gerada a partir da árvore de componentes com valores atualizados, ou mensagens de erro decorrentes de fases anteriores.

Aqui é fácil observarmos o comportamento de um **Front Control**, pois ocorre a recepção e atualização de valores, chamada de **métodos de negócio**, e desvio para a visualização correta após o processamento efetuado pelo controle, o qual pode ser um **Managed Bean** ou alguma classe acionada a partir dele. Essa segunda estratégia é a mais adequada aos requisitos básicos de independência da arquitetura MVC.

Esse Front Control é o **Faces Servlet**, o qual é mapeado no arquivo **web.xml** ao adicionarmos o framework **JSF**, durante a criação do projeto, como podemos observar a seguir.

Exemplo

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1"
  xmlns="//xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="//www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="//xmlns.jcp.org/xml/ns/javaee
    //xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">
  <context-param>
    <param-name>javax.faces.PROJECT_STAGE</param-name>
    <param-value>Development</param-value>
  </context-param>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet
    </servlet-class>
    <load-on-startup </load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
  </servlet-mapping>
  <session-config>
    <ession-timeout>
      30
    </session-timeout>
  </session-config>
  <welcome-file-list>
    <welcome-file>faces/index.xhtml</welcome-file>
  </welcome-file-list>
</web-app>
```

Comentário

Pelo mapeamento efetuado, qualquer chamada precedida de /faces/ será direcionada automaticamente para Faces Servlet, o que justifica dizer que o JSF trabalha com o padrão Front Control de forma nativa.


```
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Além de trabalhar com um Front Control, toda a modelagem funcional do JSF é voltada para a integração da camada de View ao ambiente completo da arquitetura MVC.

Basta que observemos como ocorre a utilização de **XHTML** na construção da interface visual e de **Managed Beans** no trânsito dos dados e resposta aos eventos.

Nas versões mais antigas do JSF era utilizada a sintaxe JSP na construção das páginas, mas as versões atuais priorizam os Facelets criados com uso de XHTML.

Diversos frameworks foram criados com base no JSF, como Rich Faces, Prime Faces e Ice Faces. Todos eles seguem o mesmo modelo de eventos original e, da mesma forma que o JSF, facilitam a implementação de uma arquitetura MVC com Front Control.

Rich Faces

É um produto da comunidade JBoss que apresenta, além de vários avanços visuais para os componentes, um grande suporte nativo ao uso de tecnologia AJAX, viabilizando construção assíncrona de forma simples.

Prime Faces

Segue a mesma linha, mas acrescenta diversos elementos de integração com JQuery, tratando de uma variação do JSF com grande riqueza visual.

Ice Faces

Apresenta algumas vantagens em termos de responsividade, com extensões para ambientes móveis muito práticas, inclusive dando suporte a flip e outras operações exclusivas de dispositivos móveis.

Exemplo

Poderíamos criar um aplicativo Java Web na arquitetura MVC apenas com Servlets e JSPs, conforme foi apresentado nos diagramas anteriores, mas como estamos trabalhando com JSF, utilizaremos uma abordagem um pouco diferente. As camadas iniciais, Model e Control, devem ser criadas completamente independentes do JSF para que sejam plenamente reutilizáveis, e, por isso, as anotações de Managed Beans não serão utilizadas inicialmente.

Devemos criar um projeto Java Web, com uso de JSF, do mesmo modo que foi feito em aulas anteriores, adotando o nome ExemploMVCJSF.

Utilizando o banco criado também nessas aulas, teremos a classe Produto representando a tabela equivalente, bem como a classe ProdutoDAO, que deverá concentrar as chamadas ao banco de dados. Criaremos essas duas classes no pacote model.

Podemos observar o código das classes: Produto e ProdutoDAO a seguir.

Produto

```
package model;

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

    public Produto(){

    }

    public Produto(int codigo, String nome, int quantidade) {
        &nbsthis.codigo = codigo;
        &nbsthis.nome = nome;
        &nbsthis.quantidade = quantidade;
    }

    public int getCodigo() { return codigo; }
    public void setCodigo(int codigo) { this.codigo = codigo; }
    public String getNome() { return nome; }
    public void setNome(String nome) { this.nome = nome; }
    public int getQuantidade() { return quantidade; }
    public void setQuantidade(int quantidade)
{ this.quantidade = quantidade; }
}
```

ProdutoDAO

```
<span class='mdc-text-yellow'><p>
package model;

import java.sql.*;
import java.util.*;

public class ProdutoDAO {

    private Connection getConnection() throws Exception{
        Class.forName("org.apache.derby.jdbc.ClientDriver");
        return DriverManager.getConnection(
            "jdbc:derby://localhost:1527/LojaEAD",
            "LojaEAD", "LojaEAD");
    }
    private Statement getStatement() throws Exception{
        return getConnection().createStatement();
    }
    private void closeStatement(Statement st) throws Exception{
        st.getConnection().close();
    }
    public List<Produto> obterTodos(){
        ArrayList<Produto> lista = new ArrayList<>();
        try {
            ResultSet r1 = getStatement().executeQuery(
                "SELECT * FROM PRODUTO");
            while(r1.next())
                lista.add(new Produto(r1.getInt("codigo"),
                    r1.getString("nome"),
                    r1.getInt("quantidade")));
            closeStatement(r1.getStatement());
        }catch(Exception e){
        }
        return lista;
    }
    public void incluir(Produto p){
        try {
            PreparedStatement ps = getConnection().
                prepareStatement(
                    "INSERT INTO PRODUTO VALUES(?,?,?)");
            ps.setInt(1, p.getCodigo());
            ps.setString(2, p.getNome());
            ps.setInt(3, p.getQuantidade());
            ps.executeUpdate();
            closeStatement(ps);
        }catch(Exception e){
        }
    }
    public void excluir(int codigo){
        try {
            Statement st = getStatement();
            st.executeUpdate(
                "DELETE FROM PRODUTO WHERE CODIGO = "+codigo);
            closeStatement(st);
        }catch(Exception e){
        }
    }
    preview_lead_matricula.png
}
```

A classe Produto representa a tabela de mesmo nome, e por isso traz apenas os atributos equivalentes aos campos da tabela, métodos getters e setters, além de dois construtores (default e completo).

A classe ProdutoDAO contém três métodos utilitários internos:

Atenção

Para trabalharmos com múltiplas classes DAO, poderíamos criar uma classe BaseDAO com os métodos utilitários, mudar o nível de acesso para protected e herdar todas as classes DAO de BaseDAO, reaproveitando esses métodos e facilitando a manutenção do código.

Com o uso desses métodos utilitários, fica bem mais simples trabalhar com comandos internos, como podemos observar no método excluir, que apenas gera um executor padrão, invoca o SQL para exclusão e fecha o executor.

```
Statement st = getStatement();
st.executeUpdate("DELETE FROM PRODUTO WHERE CODIGO = "+codigo);
closeStatement(st);
```

A inclusão passa por um processo um pouco mais complexo, não sendo aconselhável utilizar o executor padrão, pois existem muitos parâmetros de diversos tipos, situação em que é melhor utilizar o **PreparedStatement**.

O processo envolve a obtenção da conexão, geração do PreparedStatement, preenchimento dos parâmetros do SQL, chamada da execução para o SQL e fechamento do executor parametrizado.

```
PreparedStatement ps = getConnection().prepareStatement(
    "INSERT INTO PRODUTO VALUES(?,?,?)");
ps.setInt(1, p.getCodigo());
ps.setString(2, p.getNome());
ps.setInt(3, p.getQuantidade());
ps.executeUpdate();
closeStatement(ps);
```

As **interrogações** funcionam como parâmetros indexados a partir de 1, e, como a ordem física dos campos da tabela é dada por código, nome e quantidade, tais parâmetros fornecem os valores para os três campos respectivamente.

```
PreparedStatement ps = getConnection().prepareStatement(
    "INSERT INTO PRODUTO VALUES(?,?,?)");
```

Para o preenchimento dos valores, são utilizados os atributos equivalentes aos campos, fornecidos pelo objeto **p**, da classe **Produto**, a partir de seus métodos **getters**, sempre considerando o tipo ao setar o valor do parâmetro.

```
ps.setInt(1, p.getCodigo());
ps.setString(2, p.getNome());
ps.setInt(3, p.getQuantidade());
```

Quanto à obtenção dos produtos, ocorre a seleção de todos os produtos em um **ResultSet**, e o mesmo é percorrido de forma a adicionar em uma **lista** os objetos da classe Produto equivalentes a cada registro encontrado.

```
ResultSet r1 = getStatement().executeQuery(
    "SELECT * FROM PRODUTO");
while(r1.next())
    lista.add(new Produto(r1.getInt("codigo"),
        r1.getString("nome"),
        r1.getInt("quantidade")));
closeStatement(r1.getStatement());
```

Em seguida, devemos criar a classe ControleProduto, no pacote control, e, segundo as regras da arquitetura MVC, será o único nível que poderá utilizar o DAO.

Podemos observar o código da classe ControleProduto a seguir.

Exemplo

```
package control;

import java.util.List;
import model.Produto;
import model.ProdutoDAO;

public class ControleProduto {

    private final ProdutoDAO dao = new ProdutoDAO();

    public List<Produto> obterProdutos(){
        return dao.obterTodos();
    }

    public void inserirProduto(Produto produto){
        dao.incluir(produto);
    }

    public void excluirProduto(int codigo){
        dao.excluir(codigo);
    }
}
```

É fácil observar como **ControleProduto** faz uso extensivo de **ProdutoDAO**, e que isso faz com que não haja SQL nessa camada, nem nas demais. Todos os comandos SQL ficarão encontrados na camada **Model**.

Agora só precisamos definir nossos **Managed Beans** que, segundo essa forma de interpretar a arquitetura, pertencem à camada de visualização apenas. Vamos aproveitar o mecanismo de herança para diminuir nosso esforço em termos de programação.

Inicialmente precisamos preparar o bean para a entidade.

Exemplo

```
package managed;

import javax.enterprise.context.RequestScoped;
import javax.inject.Named;
import model.Produto;

@Named(value = "produto")
@RequestScoped
public class ProdutoBean extends Produto{
}
```

Bastante simples, não?

Tudo que fizemos foi herdar a classe original de entidade e acrescentar as anotações para que possamos utilizar como um **Managed Bean** com escopo de **requisição**.

Esse bem, em particular, servirá apenas para recepção e preenchimento de valores em formulários, sem a presença de comandos a ser executados, ao contrário do bean de controle, conforme podemos observar a seguir.

Exemplo

```
package managed;

import control.ControleProduto;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Named;

@Named(value = "controleProduto")
@ApplicationScoped
public class ControleProdutoBean extends ControleProduto{
    public String inserirProduto(ProdutoBean produto){
        super.inserirProduto(produto);
        return "ListaProduto?faces-redirect=true";
    }
}
```

Inicialmente, efetuamos o mesmo processo utilizado no bean anterior, agora considerando o escopo de aplicativo.

Mas, por ser um controlador, devemos considerar a característica de navegabilidade entre páginas do JSF, o que era configurado antigamente via XML, e atualmente é feito de forma implícita.

Não modificamos os métodos **obterProdutos** e **excluirProduto** pois são ações que não causam mudança de página (continuam em **ListaProduto**), mas o método **inserirProduto** é chamado a partir de **InserirProduto**, devendo voltar para **ListaProduto**, o que exige a modificação do método para retornar uma **String** contendo o endereço correto do JSF de listagem.

```
public String inserirProduto(ProdutoBean produto){
    super.inserirProduto(produto);
    return "ListaProduto?faces-redirect=true";
}
```

Devemos observar que inicialmente é chamado de método **inserirProduto** de **ControleProduto** com o uso de **super**, e que ProdutoBean é descendente de Produto, podendo ser utilizado no lugar.

Comentário

Bastaria retornar “ListaProduto”, mas isso faria com que a URL do navegador não fosse atualizada, sendo essa atualização da URL efetuada com “ListaProduto?**faces-redirect=true**”.

Agora temos que pensar na criação da interface JSF, ou seja, das páginas **ListaProduto** e **InserirProduto**, ambas com sintaxe XHTML. Podemos observar o código de ListaProduto a seguir.

Exemplo

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Facelet Title</title>
    </h:head>
    <h:body>
        <h:form>
            <h:commandLink action="InserirProduto?faces-redirect=true"
                value="Novo Produto"/>
            <hr/>
            <h:dataTable value="#{controleProduto.obterProdutos()}"
                var="p" border="1">
                <h:column>#{p.codigo}</h:column>
                <h:column>#{p.nome}</h:column>
                <h:column>#{p.quantidade}</h:column>
                <h:column><h:commandButton value="Excluir"
                    action="#{controleProduto.excluirProduto(p.codigo)}/>
                </h:column>
            </h:dataTable>
        </h:form>
    </h:body>
</html>
```

Agora veremos o código de InserirProduto.

Exemplo

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
    <h:head>
        <title>Facelet Title</title>
    </h:head>
    <h:body>
        <h:form>
            Código:<br/><h:inputText value="#{produto.codigo}"/><br/>
            Nome: <br/><h:inputText value="#{produto.nome}"/> <br/>
            Quantidade:<br/>
            <h:inputText value="#{produto.quantidade}"/>
            <p><h:commandButton value="Incluir"
                action="#{controleProduto.inserirProduto(produto)}/></p>
        </h:form>
    </h:body>
</html>
```


Comentário

Por que trabalhar dessa forma?
Precisamos adotar essa metodologia pelo fato de que sistemas estão sempre sofrendo mudanças de interface, e se atrelamos os níveis **Control e Model** a uma determinada tecnologia, não podemos reutilizá-los.

Para deixar ainda mais organizado, poderíamos acrescentar os métodos **listar** e **inserir** no **ControleProdutoBean**, retornando, respectivamente, as páginas de **listagem** e **inclusão**, como podemos observar a seguir.

Exemplo

```
package managed;

import control.ControleProduto;
import javax.enterprise.context.ApplicationScoped;
import javax.inject.Named;

@Named(value = "controleProduto")
@ApplicationScoped
public class ControleProdutoBean extends ControleProduto{

    public String inserirProduto(ProdutoBean produto){
        super.inserirProduto(produto);
        return "ListaProduto?faces-redirect=true";
    }

    public String listar(){
        return "ListaProduto?faces-redirect=true";
    }
    public String inserir(){
        return "InserirProduto?faces-redirect=true";
    }
}
```

Essa última mudança tiraria os links com valor direto na página JSF, de forma a garantir a chamada ao bean no redirecionamento.

```
<h:commandLink action="#{controleProduto.inserir()}"
value="Novo Produto"/>
```

Agora só precisamos acrescentar a chamada no index e testar nosso sistema.

```
<h:form> <h:commandLink action="#{controleProduto.listar()}" value="Listar Produtos"/> </h:form>
```

Podemos observar a página de index e as duas telas do sistema nas figuras seguintes.

Esse exemplo simples demonstra a facilidade com que podemos criar um ambiente robusto, dentro da arquitetura MVC, com a vantagem da orientação a eventos do JSF.

Atividade

1- Existem diversos padrões arquiteturais com finalidades bastante específicas, como no caso de ambientes de objetos distribuídos (EJB, CORBA etc.). Para esse tipo de ambiente é utilizada a arquitetura:

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

Atividade

2. Na arquitetura MVC, em quais camadas podemos definir as CLASSES do tipo DAO?

- a)Em todas as camadas, pois é apenas uma representação dos registros.
- b) Apenas na camada Model, onde ficam os elementos de acesso ao banco.
- c) a camada Control, onde estão os diversos processos de negócio.
- d) Na camada de View, facilitando a manipulação de dados pela interface.
- e) Nas camadas Control e View, já que a Model tem apenas entidades.

Atividade

3. Considerando as classes **Produto** e **ProdutoDAO**, utilizadas como base desta aula, crie na classe **ControleProduto** um método chamado **limparProdutos** para remover do banco todos os produtos com quantidade zero, assim como a classe **ControleProdutoBean**, um Managed Bean referenciado como **controle**, e o método **limpar** deste bean efetuando a chamada ao método **limparProdutos** anterior e direcionando para a página **Sucesso**.

Notas

Título modal ¹

Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos. Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos. Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos.

Título modal ¹

Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos. Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos. Lorem Ipsum é simplesmente uma simulação de texto da indústria tipográfica e de impressos.

Referências

CASSATI, J. P. Ajax, rich internet applications e desenvolvimento web para programadores. São Paulo: Prentice Hall, 2005.

Java, como programar. 8. ed. São Paulo: Pearson, 2010.

Próxima aula

- Conceito de mapeamento objeto-relacional.
- Tecnologia JPA.
- Componentes do tipo EJB.

Explore mais

- texto com [Tutorial do JSF com NetBeans; <https://netbeans.org/kb/docs/web/jsf20-intro_pt_BR.html >](https://netbeans.org/kb/docs/web/jsf20-intro_pt_BR.html)
- texto com [Tutorial de DAO com Generics; <https://www.baeldung.com/java-dao-pattern >](https://www.baeldung.com/java-dao-pattern)
- texto com [Guia de Referência JSF – JavaScript Faces; <https://www.devmedia.com.br/guia/jsf-javascript-faces/38322 >](https://www.devmedia.com.br/guia/jsf-javascript-faces/38322)

- texto com [O que é MVC?;](https://tableless.com.br/mvc-afinal-e-o-que/) <<https://tableless.com.br/mvc-afinal-e-o-que/>>
- texto com [Abordando a arquitetura MVC e Design Patterns: observer, composite, strategy;](http://www.linhadecodigo.com.br/artigo/2367/abordando-a-arquitetura-mvc-e-design-patterns-observer-composite-strategy.aspx)
<[//www.linhadecodigo.com.br/artigo/2367/abordando-a-arquitetura-mvc-e-design-patterns-observer-composite-strategy.aspx](http://www.linhadecodigo.com.br/artigo/2367/abordando-a-arquitetura-mvc-e-design-patterns-observer-composite-strategy.aspx)>
- texto com [Padrões arquiteturais.](https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013) <<https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013>>