MÓDULO VIII

CURSO TÉCNICO DE INFORMÁTICA

Carlos Eduardo Pantoja

Ministério da Educação

GOVERNO FEDERAL
BRASIL
PAÍS RICO É PAÍS SEM POBREZA

Carlos Eduardo Pantoja
Módulo:VIII

Tópicos Avançados IV

Disciplina doEixo de Disciplinas do Currículo do
Curso Técnico de Informática CEFET/RJUnED NI

Edição: CEFET/RJ UnED NI – COORDENAÇÃO DE INFORMÁTICA

Local: Estrada de Adrianópolis, 1317 – Santa Rita, Nova Iguaçu - RJ

Editora: CEFET/RJ

Ano de Publicação: 2015

CEFET/RJ

rede e-Tec Brasil

Ficha catalográfica elaborada pela Biblioteca Central do CEFET/RJ

Curso técnico de Informática, módulo VIII: Tópicos Avançados IV: disciplina do Eixo de Disciplinas Transversais ao Currículo do Curso Técnico de Informática CEFET-RJ / Carlos Eduardo Pantoja (org.). Rio de Janeiro: CEFET/RJ, – 2015.

Apresentação do e-Tec Brasil

Prezado estudante,

Bem vindo ao e-Tec Brasil!

Você faz parte de uma rede nacional pública de ensino, a Escola Técnica Aberta do Brasil, instituída pelo Decreto n° 6.301, de 12de dezembro de 2007, com o objetivo de democratizar o acesso ao ensino técnico público, na modalidade a distância. O programa é resultado de uma parceria entre o Ministério da Educação, por meio das Secretarias de Educação Profissional e Tecnológica (SETEC), as universidades e escolas técnicas estaduais e federais.

A educação a distância no nosso país, de dimensões continentais e grande diversidade regional e cultural, longe de distanciar, aproxima as pessoas ao garantir acesso à educação de qualidade, e promover o fortalecimento da formação de jovens moradores de regiões distantes, geograficamente ou economicamente, dos grandes centros.

O e-Tec Brasil leva os cursos técnicos a locais distantes das instituições de ensino e para a periferia das grandes cidades, incentivando os jovens a concluir o ensino médio. Os cursos são ofertados pelas instituições públicas de ensino e o atendimento ao estudante é realizado em escolas-polo integrantes das redes públicas municipais e estaduais.

O Ministério da Educação, as instituições públicas de ensino técnico, seus servidores técnicos e professores acreditam que uma educação profissional qualificada – integradora do ensino médio e educação técnica, - é capaz de promover o cidadão com capacidades para produzir, mas também com autonomia diante das diferentes dimensões da realidade cultural, social, familiar, esportiva, política e ética.

Nós acreditamos em você!

Desejamos sucesso na sua formação profissional!

Ministério da Educação

Janeiro de 2010

Nosso contato

etecbrasil@mes.gov.br

Indicação de ícones

Curiosidades: indica informações interessantes que enriquecem o assunto.

Interrogação: indica perguntas frequentes do aluno em relação ao tema e respostas às mesmas.

Você sabia? : oferece novas informações e notícias recentes relacionadas ao tema estudado.

Lembrete: enfatiza algum ponto importante sobre o assunto.

Tome nota 1: espaço dedicado às anotações do aluno.

Tome nota 2: espaço também dedicado às anotações do aluno.

Mãos a obra: apresenta atividades em diferentes níveis de aprendizagem para que o estudante possa realizá-las e conferir o seu domínio do tema estudado.

Bibliografia: apresenta a bibliografia da apostila.

SUMÁRIO

COM A PALAVRA, O PROFESSOR...

Caros (as) alunos (as):

Apresento a disciplina do Módulo VIIITópicos Avançados IV. Nesta parte do Curso será apresentado o desenvolvimento de sistemas para internet utilizando o Java Enterprise Edition; MVC; Hibernate e Struts. O desenvolvimento das páginas JSP se dará no ambiente de desenvolvimento Eclipse onde serão utilizadas outras tecnologias como o Tomcat e o MySQL.

Ao final deste curso, o aluno será capaz dedesenvolver um aplicativo para web usando páginas JSP com acesso a um banco de dados MySQL, utilizando um padrão de projeto que divide a implementação do sistema em camadas independentes que se comunicam, facilitando a manutenção e extensão do sistema.

É importante lembrar que esta disciplina, necessita de exercícios de cunho prático para que o aluno (a) adquira condições suficientes na elaboração de programas, mas para isso acontecer é necessário que o aluno (a) tenha o hábito de desenvolver os exercícios propostos na apostila. Bons estudos!

O organizador.

Apresentação da Disciplina

MóduloVIII–Tópicos AvançadosIV

Carga Horária: 30 Horas

Espera-se que o(a) cursista desenvolva as seguintes competências:

- Conhecer e desenvolver projetos *web* com o uso da plataforma Java *Enterprise Editione* o *framework* Eclipse, fazendo o uso de páginas JSP e acesso a banco de dados utilizando o MySQL.
- Utilizar o *Hibernate* para realizar o acesso ao banco de dados, deixando a camada de acesso a dados mais simplificada e genérica.
- Utiliza o *Struts*para minimizar a utilização de *servlets*na camada de Controladora do modelo MVC.

Projeto instrucional

Disciplina: Tópicos Avançados IV(30 horas)

Ementa: Aprimorar os conhecimentos de desenvolvimento de aplicações WEB com JSP e Servlets. Desenvolver protótipos de websites através dos frameworks MVC mais utilizados no mercado (Hibernate e Struts).

AULA	OBJETIVOS DE APRENDIZAGEM	MATERIAIS	CARGA HORÁRIA 30 (horas)
1- Introdução ao Java Enterprise Edition	Conhecer a tecnologia Java EE; o ambiente de desenvolvimento Eclipse; o container Tomcat; e páginas JSP.	impresso	4
2-Usando o Modelo MVC no Projeto Web	Conhecer o padrão de projeto MVC aplicado a um sistema de cadastro de filmes.	impresso	4
3–Realizando um CRUD em um Projeto Java EE	Realizar um CRUD utilizando o padrão de projeto MVC em um projeto Java EE.	impresso	5
4 – Hibernate .	Utilizar o Hibernate para realizar o acesso a banco de dados.	impresso	5
5 – Realizando um CRUD em um Projeto com Hibernate	Realizar um CRUD utilizando o padrão de projeto MVC em um projeto Java EE e Hibernate.	impresso	4
6 – Struts	Conhecer o framework Struts para desenvolvimento com MVC.	impresso	4
7 –Struts com Acesso a Banco de Dados	Utilizar o Struts em conjunto com banco de dados.	impresso	4

Olá, caro estudante!

Saudações Cefetianas!

Antes de iniciarmos nosso estudo sobre Tecnologia, reflita sobre essa ideia:

"Tudo aquilo que não enfrentamos em vida acaba se tornando o nosso destino."

Carl Jung

Bom estudo!

1. Introdução ao Java Enterprise Edition

Nesta aula serão apresentados os conceitos da arquitetura do Java *Enterprise Edition*(EE) para programação de páginas *web* dinâmicas; será apresentado o ambiente de desenvolvimento Eclipse, que será utilizado para a programação das páginas dinâmicas; e também serão apresentados como fazer a configuração do ambiente de desenvolvimento através da criação de uma página padrão inicial.

1.1. Java EE

O JavaEEé uma plataforma de desenvolvimento de aplicações que estende a utilização da linguagem Java (*Micro Edition* e *Standard Edition*) com objetivo de criar *softwares*executados no lado do servidor.Atualmente o Java EE é utilizado para a construção de páginas dinâmicas na Internet que representam sistemas de informações institucionais com manipulação de informações através da utilização de banco de dados. Popularmente o Java EE é conhecido como Java *Web*.

O Java EE utiliza *Java Server Pages* (JSP), que são páginas dinâmicas com conteúdo HTML e Java, onde o conteúdo em Java é processado no lado do servidor e é retornado estaticamente em conteúdo HTML. No lado do cliente, as informações são transformadas e exibidas por um navegador.Uma outra solução disponível é a*Servlet* que é uma classe Java que trata de requisições do protocolo HTTP para processar dinamicamente o conteúdo e roda diretamente em um servidor *web*. Para serem executadas, as*Servlets* precisam de um *container web*.

As páginas em JSP e as *Servlets* também podem acessar um banco de dados residente em um Sistema Gerenciador de Banco de Dados (SGBD) através de um conector *Java DataBaseConnector* (JDBC). A arquitetura básica da utilização do Java EE pode ser vista na Figura 1.

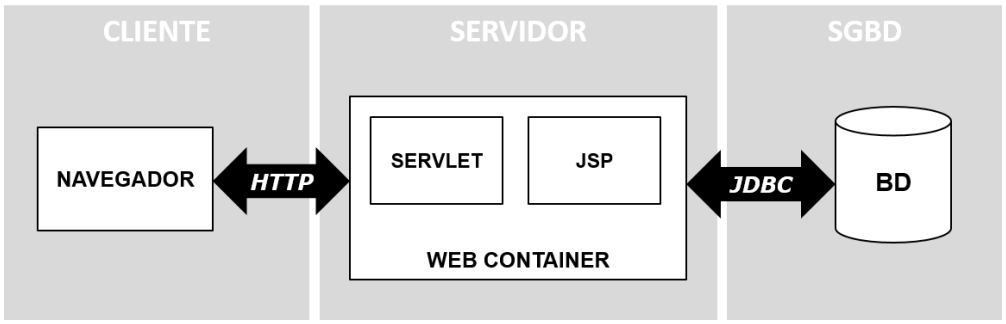


Figura 1. A arquitetura de desenvolvimento do Java EE.

Para se criar e manter as páginas JSP é necessário um servidor web com suporte a *servlets*(também chamado de *container web*). Existem diversas tecnologias dão este tipo de suporte às aplicações Java EE como o Apache Tomcat, Apache Geronimo, GlassFisheJetty.

1.2. Ambiente de Desenvolvimento Eclipse

O Eclipse é um ambiente de desenvolvimento integrado para diversas linguagens de programação em uma interface genéricae é uma das ferramentas mais adotadas no desenvolvimento de *softwares*na atualidade. Além disso, é uma ferramenta gratuita que permite diversas extensões através da instalação de *plug-ins*. Neste curso será utilizado o EclipseGalileo -Java EE IDE *For Web Developers*. Contudo existem diversas outras versões disponíveis para desenvolvedores de sistemas da internet em https://eclipse.org/.

A tela principal do ambiente do Eclipse pode ser visto na Figura 2. Do lado esquerdo da tela é possível encontrar o *Package Explorer*onde as pastas e os arquivos dos projetos estarão localizados; no centro da tela encontra-se o editor de arquivos onde serão realizadas as codificações; na parte inferior da tela encontra-se um conjunto de abas para auxiliar o programador, onde se destacam a aba *Console* responsável por exibir mensagens provenientes da execução de um programa; e a aba *Problems*, que exibem os problemas e erros encontrados em seu projeto em tempo de *Design*.

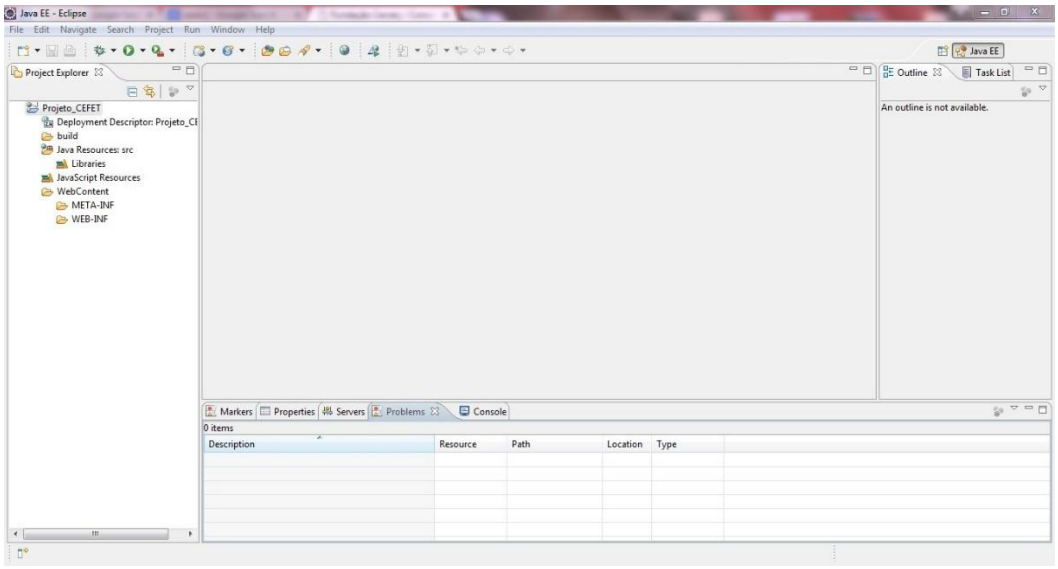


Figura 2. A tela inicial do Eclipse.

Na aba *Servers* do *Eclipse*, o *Server* configurado, como por exemplo o *TomCat*, fica disposto podendo ser inicializado e interrompido como mostra a Figura 3. Clicando no botão *start*é possível testar o funcionamento do servidor *web* configurado. A execução do *TomCat* pode ser visualizada na aba *console* do *Eclipse*. Para interromper a execução do *Server*, o botão *stop* deve ser pressionado.

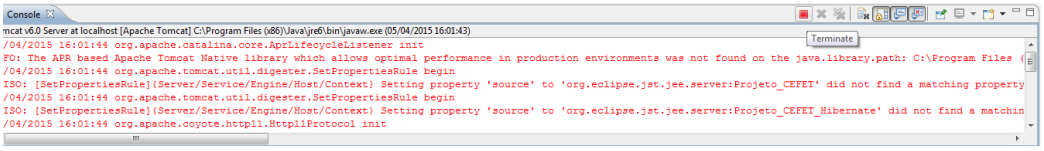


Figura 3. A aba com o servidor *Tomcat*inicializado.

Ao iniciar o Eclipse, um diretório que será usado para armazenar os seus projetos deve ser escolhido. Neste momento, um diretório já existente pode ser escolhido ou um diretório novo pode ser criado. Esse diretório é chamado de *Workspace* pelo *Eclipse*. Todos os projetos criados ficaram dispostos dentro do *Workspace* em suas respectivas pastas nomeadas com o mesmo nome do projeto.

1.3. Criando um Novo Projeto Java EE

Para se criar um novo projeto *web*é necessário clicar no menu *File>New>Dynamic Web Project*, que abrirá a tela de preenchimento de dados sobre o projeto. Preencha nessa tela o nome do seu projeto (*Project Name*) e deixe os demais itens sem alteração. Em seguida clique em *Finish* para criar seu novo projeto web. Você poderá a estrutura do seu projeto na aba *Package Explorer* do Eclipse. O processo pode ser visto na Figura 4.

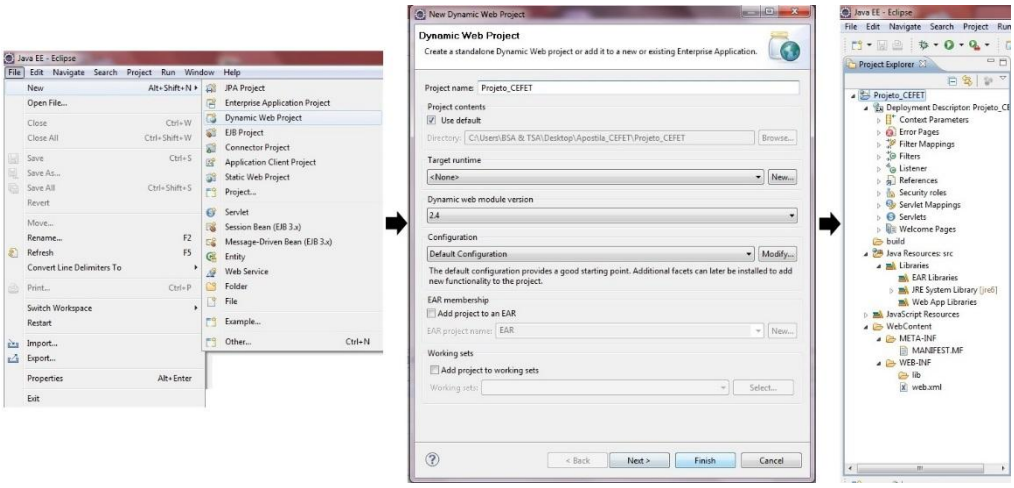


Figura 4. Criação de um novo projeto *web* e o diretório do projeto no *Package Explorer*.

Os diretórios criados por um p*rojetoJava EE* são organizados de forma a separar os arquivos ***.java*** dos arquivos *web.* No diretório *WebContent,* os arquivos *web (JSP,HTML,CSS)* ficam dispostos. *E e*m *Java Resources: src* ficam armazenados os arquivos***.java***bem como os seus pacotes.

No diretório *WEB-INF* fica o arquivo ***web.xml,*** que contém as configurações para uma aplicação *web, conforme a Figura 5.* Os códigos da figura são automaticamente gerados pelo *Eclipse* no momento de criação do projeto *web*. O elemento <*display-name*> define o nome da aplicação e no elemento <*welcome-file-list*>a lista de arquivos procurados por padrão quando o projeto for executado. *N*o diretório *build* ficam os arquivos **.class,** o **.java** compilado.

1.4. Instalação e Configuraçãodo Apache Tomcat

O Apache Tomcat é um software de servidor *web* gratuito para a implementação das tecnologias Java EE (JSP e Servlet). O Tomcat é disponibilizado no endereço eletrônico http://tomcat.apache.org/ e possui as seguintes versões disponíveis: Tomcat 6, Tomcat 7 e Tomcat8. Neste curso será utilizado o Apache Tomcat 6.0.43. O *download* pode ser feito no endereço http://tomcat.apache.org/download-60.cgi

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
 http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>Projeto_CEFET_Hibernate</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
```

Figura 5. O arquivo *web.xml*.

Após o download da versão 6 do Tomcat, executar o arquivo *apache-tomcat-6.0.43.exe.* Durante o processo de instalação, será requisitado o nome de usuário para administrador do Tomcat e sua respectiva senha. Também será pedido o caminho do diretório onde está instalado o *Java RuntimeEnvironment* (JRE). As telas podem ser vistas na Figura 6.

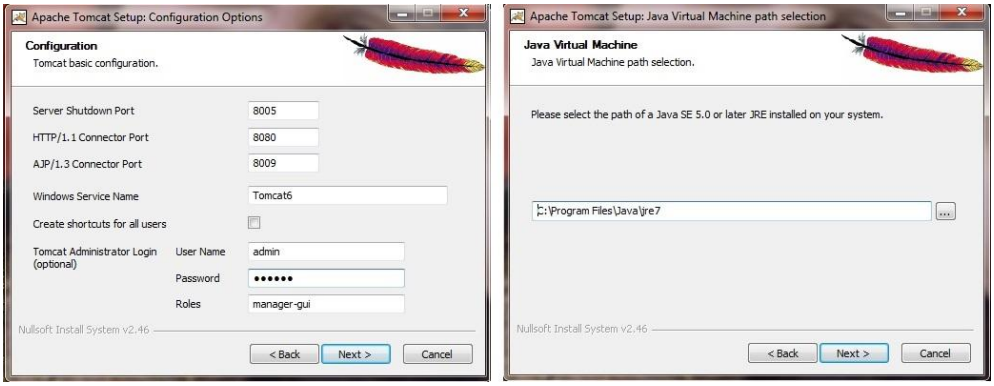


Figura 6. Configuração do Apache Tomcat.

Depois que a instalação do Tomcat for realizada, é necessário a criação de um novo servidor (*Server*) no ambiente Eclipse (este processo fará a configuração entre o Eclipse e o Tomcat instalado no

computador).Para isso é necessário clicar em *File>New>Other*e em seguida, localizar *Server*. Após clicar em *Next*, selecionar a pasta Apache e a versão do servidor que estiver utilizando (*Tomcat v6.0 Server*). Por fim, é necessário identificar o diretório onde foi instalado (na sua máquina) o Tomcat. O processo de criação do *Server* pode ser visto na Figura 7.

Além disso, é necessário adicionar a *Library*referente aoTomcat no projeto. Para isso, clique com o botão direito em cima do projeto e depois selecione as opções, consecutivamente, *Properties>Java Build Path>Libraries>add Library>Server Runtime>Apache Tomcat v6.0>Finish>ok.*

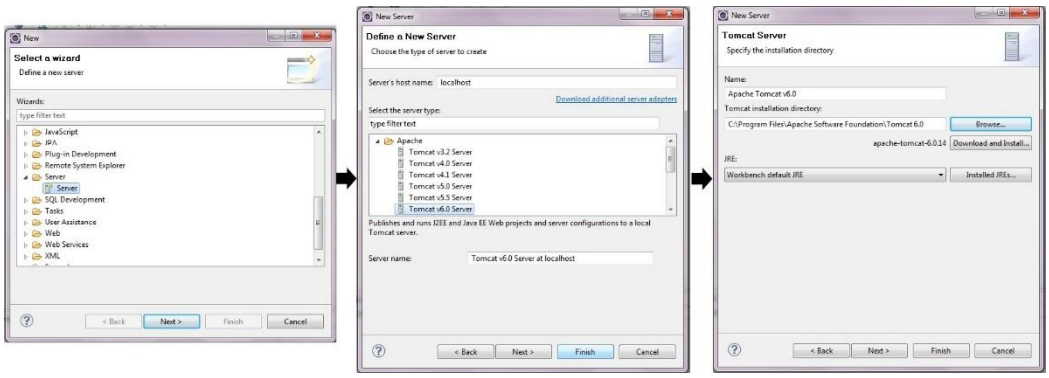


Figura 7. Criação de um novo *server* do Tomcat no Eclipse.

Após a criação do *Server* aparecerá como um novo projeto o diretório *Servers* no *Package Explorer* do Eclipse (Figura 8).

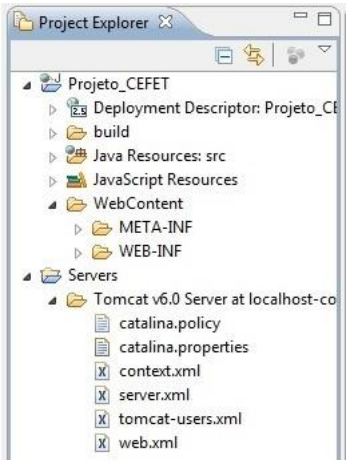


Figura 8. A pasta *Servers* do eclipse com o servidor Tomcat 6 configurado.

1.5. Criando uma Página JSP

Para se criar uma nova página JSP é necessário clicar em *File>New>JSP*, selecionar a pasta *Web Content* do projeto onde se deseja criar a nova página e digitar o nome do arquivo. Ao final, ao dar dois cliques na página JSP, a mesma será aberta no Eclipse para edição.

A sintaxe para uso no JSP é a mesma do Java, porém é necessário colocar o fragmento do código Java entre **<%** e **%>**para diferencia-las das *tags* próprias do HTML. Esse tipo de fragmento também é chamado de *scriptlets*.Note que para separar o código *Java* das *tags*em *HTML,* os *spriptlets*são inseridos, possibilitando o uso das classes em *Java*. Para que a data atual seja exibida uma *tag* de resultado é utilizada.As tags de *scriptlet*mais usadas são:

- Comentário: **<**%-- --%**>**
- Expressão de resultado: **<**%= %**>**
- Declaração de atributos ou métodos: **<**%! %**>**

As Diretivas fornecem informações especiais sobre a página JSP.Dentre elas,as principais são:

- *page***<%**@ *page***%>**: essa diretiva define as diretivas da página. No exemplo da Figura 9, a diretiva *page* é utilizada juntamente com atributo *import* e o language. Com o atributo *import*ela permite a importação das classes em *Java,* no exemplo, as classes *Date* e *SimpleDateFormat*foram importadas. Já com o atributo *language,* a diretiva *page* especifica a linguagem que está sendo utilizada, no exemplo, a linguagem *Java* é especificada.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<%@page import="java.util.Date"%>
<%@page import="java.text.SimpleDateFormat"%><html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Usando a diretiva page</title>
</head>
<body>

    <h2>#Começando a usar a diretiva page</h2>

    <%
        Date dataDeHoje = new Date();
        SimpleDateFormat formato = new  SimpleDateFormat("dd/MM/yy");
    %>

    <h2>Hoje, dia <b><%=formato.format(dataDeHoje)%></b> usei pela primeira vez a diretiva page.</h2>

</body>
</html>
```

Figura 9. Uma página JSP com diretivas.

- *taglib:* o uso dessa diretiva permite que uma biblioteca de *tags*seja importada e utilizada em uma página *JSP.*

- *include<%@ include %>:* essa diretiva permite que um arquivo seja inserido dentro de uma página *JSP*. Como por exemplo, a Figura 10, onde o cabeçalho de uma pagina *JSP* é inserida através de um *include*, informando o caminho do arquivo HTML que contém o cabeçalho através do atributo *file*.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>Locadora Sessão Das Dez</title>
    <link rel="stylesheet" type="text/css" href="Estilo/meuEstilo.css">
</head>
<body>

    <%@ include file="Estrutura/topo.html" %>
    <br>
```

Figura 10. A diretiva *include*.

Depois que a diretiva *include* é inserida, o cabeçalho com o logo do projeto ficará visível. A Figura 11 mostra o resultado do *include* para o projeto da Locadora Sessão das Dez.

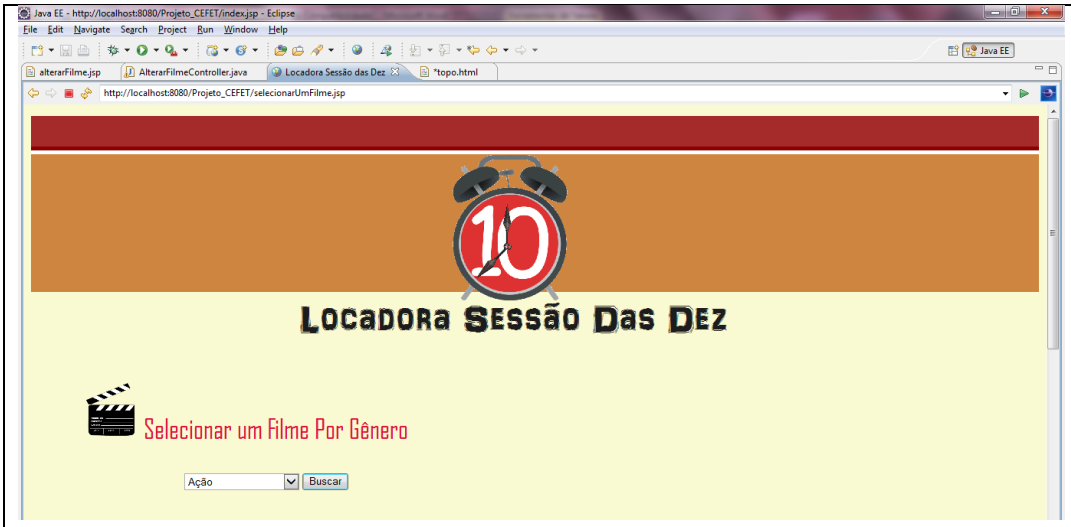


Figura 11. O resultado da diretiva *include*.

Além dos atributos explicados acima, cada diretiva possui um conjunto de atributos que podem ser utilizados para atribuir diversas características.

1.6. Compilando e Executando um Projeto Java EE

Para testar e executar um projeto Java EEno Eclipse é preciso selecionar a página JSP que se deseja executar e clicar no ícone *start* (seta verde na barra de ferramentas). Para executar o projeto direto da pasta do projeto no*eclipse*, clique em cima do projeto com o botão direito vá em *Run As>Runon Server*(Figura 12).

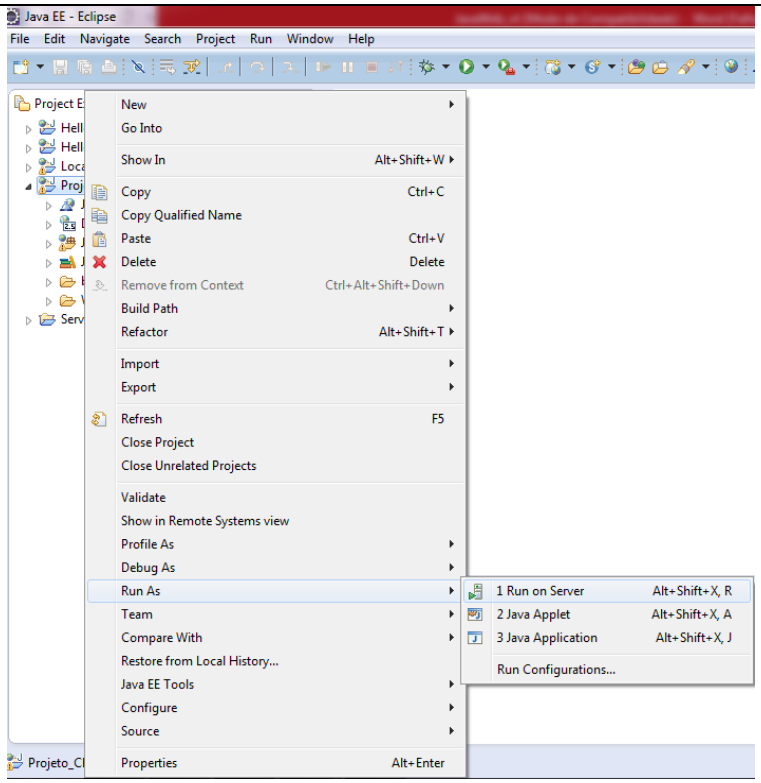


Figura12. ***Run As>Run on Server.***

2. Usando o Modelo MVC no Projeto Web

Nesta aula será apresentado o padrão de projeto Modelo-Visão-Controladora (MVC) para desenvolvimento de um projeto Java EE. Além disso, será desenvolvido um sistema de controle de filmes com um cadastro, mostrando passo-a-passo a utilização do JSP, do padrão MVC e do acesso a um banco de dados relacional.

2.1. MVC

O padrão de desenvolvimento de *software* MVC foi desenvolvido inicialmente para a linguagem *smalltalk*na década de 70 e apresentada oficialmente por Krasner e Pope (1988). O padrão consiste em dividir uma aplicação em três partes interligadas, separando as regras do negócio do sistema modelado da forma como este será exibido para o usuário. Em outras palavras, o padrão MVC permite que o usuário tenha acesso apenas as informações processadas pelo servidor sem ter acesso direto a lógica de negócio. O padrão é recomendado para sistemas complexos e *web* o que facilita a manutenção do sistema e a inclusão de novas funcionalidades. O padrão MVC não é recomendado para pequenas aplicações que rodam exclusivamente em máquinas isoladas.A Figura 13 mostra as camadas presentes em um sistema que utiliza o MVC.

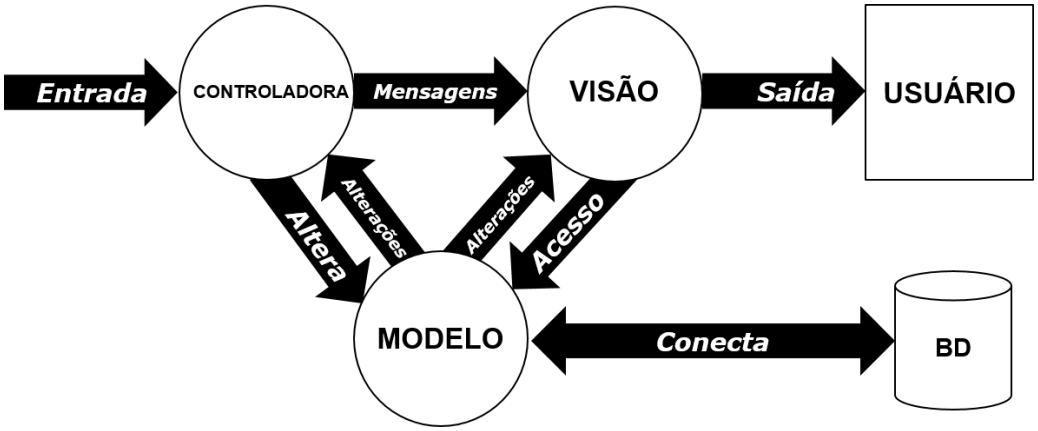


Figura 13. Arquitetura MVC [Krasner; Pope, 1988].

O modelo é composto de três camadas distintas: a Modelo, a Controladora e a Visão. A camada Controladora pode alterar a camada do Modelo e enviar mensagens para a Visão mudar os dados que são exibidos ao usuário. Por sua vez, a camada de Modelo envia para as outras camadas quando houver mudanças em seu estado, dessa forma é possível atualizar as informações exibidas pela Visão e os comandos disponíveis para a Controladora. Por fim, a camada de Visão requisita informações ao Modelo para atualizar e gerar uma representação de dados para o usuário.

Para exemplificar o modelo MVC será utilizado um exemplo de cadastro de filmes, onde o usuário irá cadastrar registros de filmes informando o título, gênero (ação, aventura, etc.), indicação se o filme concorreu ao Oscar, a sua duração e uma sinopse (descrição textual do filme). Dividindo a construção do *software* em camadas MVC, as Visões (*views*) serão as páginas JSP de cadastro do filme; as Controladoras (*controllers*) serão as *servlets*, enquanto as Modelos (*models*) serão as classes em Java representando as regras de negócio do sistema.

Na *servlet* (Controladora) estão os métodos *doPost*ou*doGet*, que são responsáveis por receber as informações que foram submetidas pelo formulário e por verificar se todos os dados foram preenchidos pelo usuário no cadastramento de um filme na página JSP. Na classe *Java* (Modelo) são definidos os atributos e os métodosreferentes a filmes, bem como os métodos de comunicação com a classe dao, a qual é responsável por realizar as operações no banco de dados.

Quando o usuário preencher as informações para cadastrar um filme através da página JSP no seu *browser* e clicar no botão *submit,*os dados preenchidos no formulário serão enviados para a controladora pelo método *post*, que verificará se todos os dados necessários para armazenar um filme foram preenchidos e enviará os dados para a classe do Modelo. Uma vez que a classe Modelo receber os dados enviados pela Controladora, ela irá chamar um método da classe *dao* responsável pela inserção no banco de dados.

2.2. Criando a Camada de Visão (*View*)

Inicialmente crie um novo projeto de página dinâmica Java EE conforme a aula anterior. Em seguida insira uma página JSP para cadastro de um filme contendo os seguintes campos: título, gênero, duração, oscar e descrição. Na Figura 14 é possível ver a página de cadastro de filme usada neste exemplo.

Figura 14. A página *cadastroFilme.jsp*.

Na página JSP para o cadastro de um filme estão presentes as representações gráficas da página usando HTML, as *tags* em código Java (*spriptlets*) usadas para exibir uma mensagem caso algum campo seja enviado sem preenchimento para a Controladora, duas diretivas *include*para inserir um arquivo de cabeçalho e outro de rodapée a diretiva*page*para definir a linguagem *Java*, conforme aFigura 15. Para a submissão dos dados do formulário para a Controladora, o atributo *method* da *tagform em HTML* será definido como*post* e o atributo *action*

receberá o nome da URL M*appings*definido no momento de criação de uma *servlet* específica para cadastrar um filme. Mais adiante será explicado passo a passo como a *servletCadastrarFilmeController* foi criada.

```
<form action="CadastrarFilmeController.do" method="post">
    <table class="tabela">
        <tr>
            <td><label for="titulo">Título:   </label>
            <input type="text" id="titulo" name="titulo" ></td>
            <td class="tamanho" ></td>
        </tr>
```

Figura 15. A *tagform* da página *cadastroFilme.jsp*.

2.3. Criando a Camada do Modelo (*Model*)

A camada de modelo representa a lógica do negócio modelado e consiste em classes comuns em *Java*. Para isso, crie uma nova classeFilme.java dentro do pacote (*package*) *model*na pasta *Java Resources>src*.Nesta classe crie os atributos título, descrição, gênero e oscar do tipo *String*; e os atributos id e duração do tipo *int*. Crie também todos os *getters* e *setters* destes atributos. Em seguida crie o método *cadastroFilme* com um parâmetro do tipo *Filme*. A classe criada pode ser vista na Figura 16.

```
public class Filme {

    private int id;
    private String titulo;
    private String descricao;
    private int duracao;
    private String genero;
    private String oscar;

    public String getOscar() {
    public void setOscar(String oscar) {
    public int getId() {
    public void setId(int id) {
    public String getTitulo() {
    public void setTitulo(String titulo) {
    public String getDescricao() {
    public void setDescricao(String descricao) {
    public int getDuracao() {
    public void setDuracao(int duracao) {
    public String getGenero() {
    public void setGenero(String genero) {

    public void cadastroFilme(Filme filme){

    }
}
```

Figura 16. A classe *Filme.java*.

O método *cadastroFilme* será implementado mais a frente para fazer o efetivo cadastro de um filme em um banco de dados, onde a camada de acesso de dados será programada para realizar uma conexão com o MySQL Server.

2.4. Criando a Camada de Controle (*Controller*)

A camada de controle consiste de *servlets*específicas para cada ação realizada (e.g. cadastro). Para isso crie uma nova *servlet CadastrarFilmeController.java*dentro do pacote (*package*) *controller*na pasta *Java Resources>src*. Os passos para a criação podem ser vistos na Figura 17. Note que o campo *Classname* receberá o nome de *CadastrarFilmeControlle*r por convenção a terminação ***.do*** será usada para definir o nome da *servlet* na URL *Mappings.* Para isso,o nome na URL *Mappings* deve ser editado para *CadastrarFilmeController.do.* A página JSP para cadastro de um filme utilizará o nome atribuído na URL *Mappings*para chamar esta*servlet*que éresponsável pelo cadastramento

de um filme. Sendo assim, o nome definido na URL *Mappings* deve ser atribuído a *tagaction*do formulário na página JSP.

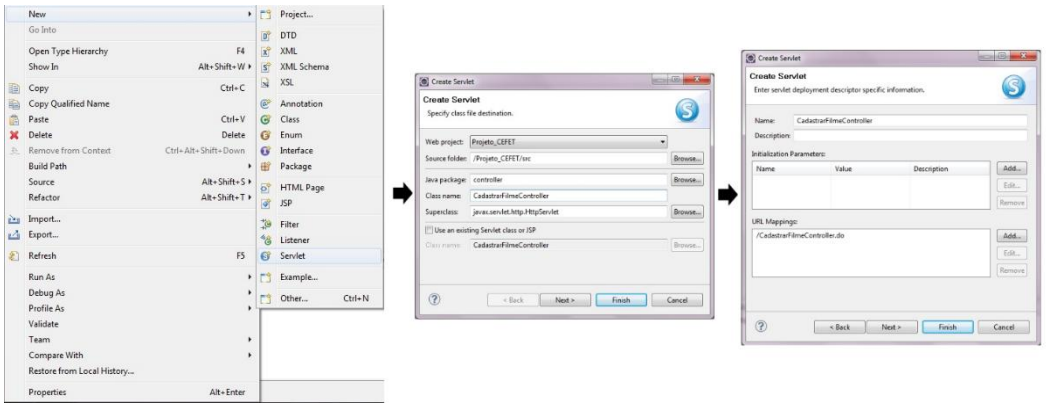


Figura 17. Criação da *servletCadastrarFilmeController*.

Quando uma *servlet*for criadae a nomeada, automaticamente, o arquivo *web.xml* pertencente ao diretório *WEB-INF*será modificado, recebendo as informações para realizar o mapeamento de cada *servlet.*Dentre os códigos gerados, automaticamente, no arquivo *web.xml*se encontra o elemento <*url-pattern*>que receberá o nome informado na URL *Mappings*da *servlet*no momento de sua criação. Esse elemento será usado para realizar o mapeamento e direcionamento para uma determinada *servlet* que foi chamada por uma página JSP.

AFigura 18apresenta o código do arquivo *web.xml* gerado após a criação da *servletCadastrarFilmeController.*Alguns elementos de descrição da *servlet*também foram acrescentados no arquivo *web.xml* como o<*display-name*>,<*servlet-name*> e o<*servlet-class*>.

```
<servlet>
  <description></description>
  <display-name>CadastrarFilmeController</display-name>
  <servlet-name>CadastrarFilmeController</servlet-name>
  <servlet-class>controller.CadastrarFilmeController</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>CadastrarFilmeController</servlet-name>
  <url-pattern>/CadastrarFilmeController.do</url-pattern>
</servlet-mapping>
<servlet>
```

Figura 18. O arquivo *web.xml*.

Para cada *servlet* criada, os mesmos códigos ilustrados serão

gerados no arquivo *web.xml*, alterando apenas os nomes atribuídos aos elementos (*<url-pattern>, <display-name>,<servlet-name>, <servlet-class>*) já que estes dependem da nomenclatura utilizada na *servlet* e do pacote a qual a mesma está associada.

O próximo passo é programar o método *doPost* ou *doGet,*para receber as informações vindas da camada de Visão. Neste exemplo foi escolhido utilizar o *doPost* pois as informações enviadas do formulário para controladora serão transferidas de forma encapsulada, ou seja, as informações não estarão visíveis na URL da página.

O método *doPost*, assim como o *doGet*, possui dois parâmetros: o *request* do tipo *HttpServletRequest*, que é responsável por obter os dados vindos da camada de visão pelo método *post* do HTML; e o *response* do tipo *HttpServletResponse*, que é responsável por enviar uma informação para a camada de visão (e.g. um objeto).Dentro do método *doPost*, o parâmetro *request* possui o método *getParameter*, que efetivamente obterá os campos enviadospelo formulário através do atributo *name*utilizado dentro da *taginput* do HTML.A Figura 19 mostra a implementação inicial do método *doPost*.

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

    String titulo = request.getParameter("titulo");
    String genero = request.getParameter("genero");
    String oscar = request.getParameter("oscar");
    String duracao = request.getParameter("duracao");
    String descricao = request.getParameter("descricao");
}
```

Figura 19. A programação inicial do método *doPost*.

Dentro do método *doPost,*a validação para campos nulos será realizada através de dois *arrays*: um para verificar se os campos são nulos e outro para guardar as mensagens com os campos que estiverem nulos. A Figura 20 exibe o código de validação para a controladora.

```
String mensagem=null;
String [] arrayParametros={titulo,genero,oscar,duracao,descricao};
String [] arrayCampos={"Título","Gênero","Indicação ao oscar","Duração","Descrição"};

try {
      for(int contador=0; contador < arrayParametros.length; contador++){
         if(arrayParametros[contador]==null || arrayParametros[contador].length()<=0){
            if(mensagem==null){
                mensagem=arrayCampos[contador];
            }else{
                mensagem=mensagem+" , "+arrayCampos[contador];
            }
        }

      }
```

Figura 20. A validação de dados na controladora.

Caso existam campos que estejam nulos, uma mensagem será enviada para a página *cadastroFilme.jsp*por meio de um parâmetro do método *setAttribute* do objeto *request*.Caso contrário, a classe *Filme* da camada de modelo será instanciadae todos os campos recebidos da camada de visão serão adicionados nos respectivos atributos de *Filme*. Por fim, o método para cadastrar o filme é chamado. O restante da programação da controladora pode ser visto na Figura 21.

```
if (mensagem != null) {
    mensagem = "Preencha corretamente o(s) campo(s): " + mensagem
            + ".";
    request.setAttribute("mensagem", mensagem);
    RequestDispatcher dispatcher = request
            .getRequestDispatcher("cadastroFilme.jsp");
    dispatcher.forward(request, response);
} else {
    filme = new Filme();
    filme.setTitulo(titulo);
    filme.setGenero(genero);
    filme.setOscar(oscar);
    filme.setDuracao(Integer.parseInt(duracao));
    filme.setDescricao(descricao);

    filme.cadastroFilme(filme);
    RequestDispatcher dispatcher = request
            .getRequestDispatcher("index.jsp");
    dispatcher.forward(request, response);
}
```

Figura 21. Atribuindo valores na camada de modelo.

Para que a mensagem enviada pela controladora seja exibida na página *cadastroFilme.jsp*, o método getAttribute do objeto *request*deve

ser utilizadorecebendo como parâmetro o nome *"mensagem"*definido no método*setAttribute.*

2.5. Criando o *Data Access Object* (DAO)

O *Data Access Object* (DAO) é um objeto responsável pelo acesso a um banco de dados de forma independente do modelo do negócio. A classe DAO separa a camada de Modelo do MVC da camada de dados, provendo um tratamento diferenciado dos métodos do negócio modelado e dos métodos que conectam, incluem, excluem, alteram e recuperam as informações em um banco de dados.

O DAO consiste de uma classe em *Java* comum com os métodos que fazem acesso a banco de dados (inserção, exclusão, alteração e seleção) e é instanciada nas classes da camada de Modelo. No exemplo desta aula será criado o DAO para a classe *Filme,* de forma que os dados inseridos pelo usuário possam ser persistidos. Para isso, crie uma classe com o nome de*FilmeDAO* dentro do pacote *dao*na pasta *Java Resources>src*.

Na classe *FilmeDAO* crieum método chamado de*cadastroFilme* sem tipo de retorno e que receba como parâmetro um objeto tipo *Filme.*A Figura 22 exibe o código do método*cadastroFilme*pertencente a classe *FilmeDAO* e que realiza o cadastro de filmes.

```
public void cadastroFilme(Filme filme) throws ExceptionDAO{

    String sql = "insert into filme (titulo,genero,oscar,duracao,descricao) values (?,?,?,?,?)";
    PreparedStatement stmt = null;
    Connection connection = null;
    try {
        connection= new ConexãoBancoDeDados().getConnection();
        stmt = connection.prepareStatement(sql);
        stmt.setString(1, filme.getTitulo());
        stmt.setString(2, filme.getGenero());
        stmt.setString(3, filme.getOscar());
        stmt.setInt(4, filme.getDuracao());
        stmt.setString(5, filme.getDescricao());
        stmt.execute();

    } catch (SQLException e) {
        e.printStackTrace();
        throw new ExceptionDAO("Erro ao cadastrar filme:"+ e);

    } finally {
        try {
            if (stmt != null) {
                stmt.close();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        try {
            if (connection != null) {
                connection.close();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Figura 22. O método*cadastroFilme*.

O método *cadastroFilme* da classe FilmeDAO possui dois objetos importantes para a comunicação com um banco de dados: a conexão com o banco de dados (*connection*) e o comando em *Structured Query Language* (SQL) (*PreparedStatement*). Para se inserir informações em um banco de dados é preciso realizar uma conexão entre o programa que está sendo desenvolvido e o servidor de banco de dados. Além disso, é necessário um comando, preparado com as informações do objeto da camada de Modelo.

Para se criar uma conexão, insira uma nova classe chamada de *ConexãoBancoDeDados*dentro dopacote *dao*. Neste exemplo utilizaremos o *MySQL Server*5.6 e *MySQL Workbench* 6.1 como Sistema Gerenciador de Banco de Dados (SGBD). A classe de conexão ao banco de dados no Java EE é idêntica a um programa em Java *Standard Edition* (SE), portanto é necessário definir um *Driver*de conexão e adicionar o *JAR*do *driverno projeto. Para isso,* clique com o botão direito na pasta do projeto e selecione as opções Properties>*Java Build*

Path>AddExternalJars.

Para criar um Driver de conexão, será preciso definir qual servidor e o banco de dados acessar. Neste exemplo, está sendo usado o *localhost* como endereço do servidor e base de dados foi denominada de*locadorasessaodasdez*.Para isso,uma tabela *Filme* com as colunas *id* como chave primária, *titulo*, *genero*, *oscar*, *duracao* e *descrição* foi criada na base de dados *locadorasessaodasdez*. O código referente a classe de conexão pode ser visto na Figura 23.

```
package dao;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConexãoBancoDeDados {

    public Connection getConnection(){

        Connection conn = null;

        try{
            Class.forName("com.mysql.jdbc.Driver");
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        try {

        conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/locadorasessaodasdez","root",null);

        }catch (SQLException e) {
        e.printStackTrace();
        System.out.println(e);
    }
        return conn;
    }
}
```

Figura 23. A classe de conexão ao banco de dados.

Para finalizar a implementação do método para cadastrar um filme usando o modelo MVC, o método cadastrar da classe *Filme* deve ser definido. Esse método irá instanciar um objeto do tipo da classe*FilmeDao*para chamar o método cadastrar filme referente a essa classe, conforme ilustrado na Figura 24.

```
public void cadastroFilme(Filme filme) throws ExceptionDAO {
    new FilmeDAO().cadastroFilme(filme);
}
```

Figura 24. A método de cadastrar filmes na classe *Filme*.

O próximo passo é rodar o projeto e inserir um filme na página

JSP, preenchendo todas as informações requisitadas. Após a execução, faça uma verificação no banco de dados para conferir se o registro referente ao cadastro foi inserido.

3. Realizando um CRUD em um Projeto Java EE

Um sistema deve ser capaz de realizar operações de inserção, exclusão, alteração e recuperação de informações em um banco de dados como forma de interação com usuário. Essas operações em um sistema são conhecidas como *Create-Read-Update-Delete* (CRUD). Na aula anterior foi realizado o cadastro de um filme representando um *Create*(criação ou inserção). Nessa aula serãorealizadas as outras operações presentes em um CRUD: recuperação, alteração e exclusão.

3.1. Recuperação de Dados (*Read*)

Para o exemplo de recuperação de dados, serão criadas duas páginas: *selecionarFilme.jsp*, que será responsável por listar os filmes por gênero e possibilitará a exclusão de um registro; e*alterarFilme.jsp* que listará todos os filmes por gênero e possibilitará a alteração de um deles.

Primeiramente é preciso modificar a classe *Filme* da camada de modelo para ser capaz de listar filmes e a classe *FilmeDAO* para fazer a comunicação com o banco de dados e retornar os registros esperados. Na classe *Filme* será necessário a inclusão de um método chamado *listarFilmesPorGenero*, que instanciará a classe *FilmeDAO*.O código referente à implementação do método pode ser visto na Figura 25.

```
public ArrayList<Filme> listarFilmesPorGenero() throws ExceptionDAO {
    return new FilmeDAO().listarFilmesPorGenero(this.getGenero());
}
```

Figura 25. A método de listar filmes na classe *Filme*.

A classe *FilmeDAO* também terá um método chamado *listarFilmesPorGenero.* O código relativo ao método *listarFilmesPorGenero*da classe *FilmeDAO*pode ser visto na Figura 26. O código de tratamento de exceção foi suprimido por questões de

apresentação.

```
public ArrayList<Filme> listarFilmesPorGenero(String genero) throws ExceptionDAO{
        ResultSet rs =null;
        Connection conn=null;
        PreparedStatement stmt = null;
        Filme filme =null;
        ArrayList<Filme> listaDeFilme = null;
    try {
            String sql="select id,titulo,duracao,descricao from filme where genero=?";
            conn=new ConexãoBancoDeDados().getConnection();
            stmt=conn.prepareStatement(sql);
            stmt.setString(1, genero);
            rs=stmt.executeQuery();
                if(rs != null){
                    listaDeFilme = new ArrayList<Filme>();
                    while(rs.next()){
                        filme= new Filme();
                        filme.setId(rs.getInt("id"));
                        filme.setTitulo(rs.getString("titulo"));
                        filme.setDuracao(rs.getInt("duracao"));
                        filme.setDescricao(rs.getString("descricao"));
                        listaDeFilme.add(filme);
                    }
                }
```

Figura 26. A método de listar filmes na classe *FilmeDAO*.

O próximo passoserá criar uma nova controladora chamada de *ListarFilmePorGeneroController.java*, que será responsável por gerenciar a lista de filmes a ser exibida na página *selecionarFilme.jsp* ou na página *alterarFilme.jsp*. Foi utilizada a mesma controladora para listar os filmes tanto para a página *selecionarFilme.jsp*(pelo método doGet) *quanto para a página alterarFilme.jsp (pelo método doPost).* A diferença entre os métodos *doGet*e do *doPost* da controladora ListarFilmePorGeneroController.javaestá no fato do resultado da consultaser despachadapara páginas diferentes.

Para que a página*selecionarFilme.jsp* requisiteuma listagem de filme à controladora *ListarFilmePorGeneroController.do* é necessário programar o método *doGet*. Este método será responsável por adicionar um atributo (um *array*de filmes) em um objeto do tipo *request*através do método setAttribute,que será enviado para a página *selecionarFilme.jsp*. O método *getRequestDispatcher*receberá como parâmetro o nome da página*selecionarFilme.jsp*para que oarray de filme sejadirecionado para esta página.

Note que o método*doGet*receberá da página JSP um parâmetro do tipo Stringdenominado de *gênero*. Este será utilizado na camada

daopara recuperar no banco de dados os registros dos filmes que possuam o valor do campo *gênero* idêntico ao valor do parâmetro.
O método *doGet*pode ser visto na Figura 27.

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    String genero= request.getParameter("genero");
    ArrayList<Filme> arrayDeFilmePorGenero=null;
    Filme filme=null;

    try {
        if(genero!=null && genero.length()>0){
            filme= new Filme();
            filme.setGenero(genero);
            arrayDeFilmePorGenero=filme.listarFilmesPorGenero();
            request.setAttribute("arrayDeFilmePorGenero", arrayDeFilmePorGenero);
            RequestDispatcher dispatcher = request.getRequestDispatcher("selecionarFilme.jsp");
            dispatcher.forward(request, response);
        }}catch (Exception e) {
            response.getWriter().write("Erro ao listar filme: "+ e);
        }
    }
```

Figura 27. O método *doGet* para listar filmes.

Em seguida é preciso preparar a página *selecionarFilme.jsp*para listar os filmes retornados pela controladora. Para isso é precisoresgatar o atributo *arrayDeFilmePorGenero*, que foi enviado pela controladora por meio do método *getAttribute*, caso a página verifique que este atributo não é nulo (esse fato indica que uma consulta foi realizada). Uma vez que o atributo foi resgatado é possível iterar todos seus itens e lista-los na página conforme mostra a Figura 28.

```
    <% if(request.getAttribute("arrayDeFilmePorGenero")!=null){
    ArrayList<?> arrayDeFilmePorGenero=null;
    Filme filme =null;
    arrayDeFilmePorGenero=(ArrayList<?>)request.getAttribute("arrayDeFilmePorGenero");
    Iterator <?>iteratorArrayDeFilmePorGenero=arrayDeFilmePorGenero.iterator();
    while(iteratorArrayDeFilmePorGenero.hasNext()){
        filme=(Filme)iteratorArrayDeFilmePorGenero.next(); %>

<form action="AlterarFilmeController.do" method="post">
    <table class="tabela">
        <tr>
            <th class="formatoTabelaBusca">Título</th>
            <th class="formatoTabelaBusca">Duração</th>
            <th class="formatoTabelaBusca">Descrição</th>
            <th class="formatoTabelaBusca">Alterar</th>
        </tr>
        <tr>
            <td class="formatoTabelaBusca"><input type="text" name="titulo" value="<%out.print(filme.getTitulo());%>" ></td>
            <td class="formatoTabelaBusca" >
                <input type="text" name="duracao" value="<%out.print(filme.getDuracao());%>" >
                <input type="hidden" name="id" value="<%out.print(filme.getId());%>" >
                <input type="hidden" name="genero" value="<%out.print(filme.getGenero());%>">
                <input type="hidden" name="oscar" value="<%out.print(filme.getOscar());%>">
            </td>
            <td class="formatoTabelaBusca"><input type="text" name="descricao" value="<%out.print(filme.getDescricao());%>" ></td>
            <td class="formatoTabelaBusca"><button type="submit">Alterar</button></td>
```

Figura 28. O código em java na página JSP.

Para que os objetos possam ser instanciados sem erros é

necessário importar as bibliotecas das classes no início do arquivo JSP através da diretiva *page*usando o atributo*import*. Na Figura 29 é possível ver as importações necessárias para a página *selecionarFilme.jsp*.

```
<%@page import="java.util.ArrayList"%>
<%@page import="model.Filme"%>
<%@page import="com.sun.org.apache.bcel.internal.generic.INSTANCEOF"%>
<%@page import="java.util.Iterator"%><html>
```

Figura 29. As importações na página JSP.

Agora é só rodar a página de listar filmes, escolher um gênero e clicar no botão *Buscar*. No atributo*action*desse formulário o nome ListarFilmePorGeneroController.do deve ser informado e o atributo *method* deve ser definido como *get*. Todos os filmes presentes no banco de dados serão listados conforme pode ser visto na Figura 30.

Figura 30. A lista de filmes exibidos pela página JSP.

3.2. Exclusão de Dados (*Delete*)

A página *selecionarFilme.jsp*também será utilizada para fazer a exclusão de registros no banco de dados, visto que para se excluir um registro é preciso primeiro listar e selecioná-lo. Ao listar um filme, do lado direito dos registros aparece uma coluna *Excluir*com um ícone de um “X’

que será usado para excluir um registro. Ao se clicar nesse ícone, a página *selecionarFilme.jsp*irá chamar uma controladora pelo método *get* passando o *id*do filme a ser excluído.

Inicialmente é necessário criar o método *excluirFilme*tanto na classe *Filme* quanto na classe *FilmeDAO*. Na classe *Filme*o método será responsável apenas por instanciar a classe *FilmeDAO* e chamar o método para efetivamente excluir o registro. A Figura 31 exibe a implementação do método na classe *Filme*.

```
public void excluirFilme() throws ExceptionDAO {
    new FilmeDAO().excluirFilme(this.getId());
}
```

Figura 31. A método de excluir filmes na classe *Filme*.

No método *excluirFilme* na classe *FilmeDAO*é preciso preparar o comando SQL que será responsável pela exclusão de um registro levando em consideração o *id* do filme que será passado por parâmetro. Em seguida o método executará o comando SQL que efetivamente excluirá o registro da base de dados. A implementação do método pode ser vista na Figura 32.

```
public void excluirFilme(int id) throws ExceptionDAO{

    String sql = "delete from filme where id=?";
    PreparedStatement stmt = null;
    Connection connection = null;
    try {
        connection= new ConexãoBancoDeDados().getConnection();
        stmt = connection.prepareStatement(sql);
        stmt.setInt(1, id);
        stmt.execute();

    } catch (SQLException e) {
        e.printStackTrace();
        throw new ExceptionDAO("Erro ao excluir filme: "+ e);
```

Figura 32. A método de excluir filmes na classe *FilmeDAO*.

Agora, é preciso criar a controladora *ExcluirFilmeController.java*e implementar o método *doPost*para que um registro seja excluído. Quando o usuário clicar no ícone de exclusão (*tag* âncora em HTML) de

um filme a controladora *ExcluirFilmeController.java*será chamada através do método *get*. O método *doGet*dessa controladora fará uma chamada ao método *doPost*também pertencente a essa controladora por meio do código *doPost(resquest, response)*. Por isso, a implementação do método excluir será feito no método *doPost* da controladora.

O método*doPost* fará a chamada para o método *excluirFilme* da classe *Filme*, que repassará o comando para a camada dao para realizar a exclusão do registro. O método *doPost* da controladora pode ser visto na Figura 33.

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    Filme filme=null;
    String id = request.getParameter("id");

     try{
         filme = new Filme();
         filme.setId(Integer.parseInt(id));
         filme.excluirFilme();
         RequestDispatcher dispatcher = request.getRequestDispatcher("index.jsp");
         dispatcher.forward(request, response);
     }   catch (Exception e) {
          response.getWriter().write("Erro ao excluir filme: "+ e);
        }

       }
    }
```

Figura 33. O método *doPost* da classe *ExcluirFilmeController*.

Na página *selecionarFilme.jsp,* a chamada a controladora *ExcluirFilmeController.java*será feita através de uma *tag* âncora do HTMLpassando o id referente ao filme a ser excluído.Esse código pode visualizado na Figura 28. Usando essa *tag*para fazer uma chamada a controladora o método *get* será utilizado.Agora é só rodar o exemplo e realizar uma exclusão de registro.

3.3. Alteração de Dados (*Update*)

Para realizar uma alteração de dados, será preciso primeiro listar os registros por um determinado gênero antes de efetuar alguma alteração nos dados. A controladora *ListarFilmePorGeneroController*através do método*doPost*será

responsável pela listagem dos filmes que serão habilitados para alteração.

Na controladora *ListarFilmePorGeneroController.java*, o método*doPost*será implementado de forma semelhante ao método *doGet*dessa controladora.Com apenas uma alteração no método*getRequestDispatcher*que agora receberá como parâmetro o nome da página*alterarFilme.jsp*para que o array de filme seja direcionado para esta página. No atributo*action*do formulário para listar um filme na*alterarFilme.jsp* o nomeListarFilmePorGeneroController.dodeve ser informado e o atributo*method*deve ser definido como*post.*

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    String genero= request.getParameter("genero");
    ArrayList<Filme> arrayDeFilmePorGenero=null;
    Filme filme=null;

    try {
        if(genero!=null && genero.length()>0){
            filme= new Filme();
            filme.setGenero(genero);
            arrayDeFilmePorGenero=filme.listarFilmesPorGenero();
            request.setAttribute("arrayDeFilmePorGenero", arrayDeFilmePorGenero);
            RequestDispatcher dispatcher = request.getRequestDispatcher("alterarFilme.jsp");
            dispatcher.forward(request, response);
        }}catch (Exception e) {
            response.getWriter().write("Erro ao listar filme: "+ e);
        }
    }
```

Figura 34. O método *doPost* da classe *ListarFilmePorGeneroController*.

Agora, é preciso adicionar o método *alterarFilme* nas classes *Filme* e *FilmeDAO* (como foi feito nos demais exemplos). O método *alterarFilme* na classe Filme será responsável por fazer a chamada a classe *FilmeDAO* e ao método para efetivamente realizar a alteração no banco de dados. A Figura 35 mostra o código do método *alterarFilme*.

```
public void alterarFilme(Filme filme) throws ExceptionDAO {
    new FilmeDAO().alterarFilme(filme);
}
```

Figura 35. O método *alterarFilme* da classe *Filme*.

No método *alterarFilme* da classe *FilmeDAO* ocorrerá a conexão

com o banco de dados e a preparação do comando SQL para efetiva alteração de dados de um determinado filme. O código referente ao método da classe *FilmeDAO* pode ser visto na Figura 36.

```
public void alterarFilme(Filme filme) throws ExceptionDAO{

    String sql = "update filme set titulo=?,duracao=?,descricao=? where id=?";
    PreparedStatement stmt = null;
    Connection connection = null;
    try {
        connection= new ConexãoBancoDeDados().getConnection();
        stmt = connection.prepareStatement(sql);
        stmt.setString(1, filme.getTitulo());
        stmt.setInt(2, filme.getDuracao());
        stmt.setString(3, filme.getDescricao());
        stmt.setInt(4, filme.getId());
        stmt.execute();

    } catch (SQLException e) {
        e.printStackTrace();
        throw new ExceptionDAO("Erro ao alterar filme: "+ e);
```

Figura 36. O método *alterarFilme* da classe *FilmeDAO*.

O próximo passo é criar a classe controladora *AlterarFilmeController.java*dentro do pacote *controller*. Esta classe terá implementação semelhante com a classe *CadastrarFilmeController* implementada na aula anterior. As diferenças consistem em: o tratamento de erros e o objeto do tipo *Filme* fará a chamada ao método *alterarFilme* na controladora.

O último passo é fazer com que a camada de visão (*alterarFilme.jsp*) exiba a lista de filmes por gênero e efetuar a alteração de dados digitados pelo usuário. O código para listar os filmes na página de alteração é similar ao código da página de listar filmes. No caso da alteração os valores listados aparecerão em uma caixa de texto com possibilidade de edição e ao lado de cada registro aparecerá um botão de alterar. Um trecho de código da página *alterarFilme.jsp* pode ser visto na Figura 37.

Para que a controladora*AlterarFilmeController*seja chamada pela página *alterarFilme.jsp,* o atributo*action*do formulário para alterar um filme deve receber o nome *AlterarFilmeController*.doe o atributo*method*deve ser definido como*post.*

```
<%
    if (request.getAttribute("arrayDeFilmePorGenero") != null) {
        ArrayList<?> arrayDeFilmePorGenero = null;
        Filme filme = null;
        arrayDeFilmePorGenero = (ArrayList<?>) request
                .getAttribute("arrayDeFilmePorGenero");
        Iterator<?> iteratorArrayDeFilmePorGenero = arrayDeFilmePorGenero
                .iterator();
        while (iteratorArrayDeFilmePorGenero.hasNext()) {
            filme = (Filme) iteratorArrayDeFilmePorGenero.next();
%>
<tr>
    <td class="formatoTabelaBusca"><input type="text" name="titulo"
        value="<%out.print(filme.getTitulo());%>"></td>
    <td class="formatoTabelaBusca"><input type="text"
        name="duracao" value="<%out.print(filme.getDuracao());%>">
        <input type="hidden" name="id"
        value="<%out.print(filme.getId());%>"></td>
    <td class="formatoTabelaBusca"><input type="text"
        name="descricao" value="<%out.print(filme.getDescricao());%>">
    </td>
    <td class="formatoTabelaBusca"><button type="submit">Alterar</button></td>
</tr>

<%
```

Figura 37. O método *alterarFilme* da classe *FilmeDAO*.

Agora é necessário rodar o projeto e listar os filmes por um determinado gênero (e.g. Ação). Uma listagem de filmes será exibida e ao se modificar os dados de um registro as informações serão persistidas no banco de dados. A Figura 38 exibe a página de alteração em execução.

Figura 38. A lista de filmes exibidos pela página *alterarFilme.jsp*.

4. *Hibernate*

Nesta aula, uma nova abordagem de acesso ao banco de dados será discutida. Algumas ferramentas foram desenvolvidas para facilitar o acesso ao banco de dados, de forma a melhorar a transição de dados utilizados por um objeto ao um modelo tabular manipulado pelo banco de dados ou vice-versa. Essas ferramentas realizam um trabalho de mapeamento com acesso ao banco de dados, funcionando como um intérprete entre o banco de dados e as classes da camada de modelo.Uma das ferramentas de mapeamento existentesé o*Hibernate*.

O *Hibernate* é um *framework* que se relaciona com o banco de dados por intermédio de um mapeamento objeto/relacional para Java. Nenhuma alteração na aplicação é necessária para utilizar o *Hibernate*, bastando apenas algumas configurações para que a persistência de dados possa ser gerenciada pelo *framework*. Dessa forma a responsabilidade de gerenciar o acesso ao banco de dados é entregue ao *Hibernate*.

O uso do *Hibernate* simplifica as classes pertencentes à camada DAO, já que a ferramenta gerencia a transição de informações das classes para o banco de dados ou do banco de dados para as classes. Em poucas linhas de código essa transição é realizada, permitindo que o programador se dedique mais tempo construindo a lógica do negócio. Em comparação com a classe DAO das aulas iniciais, será visível como o uso do *framework* reduzirá as linhas de código simplificando as operações realizadas no banco de dados.

Os exemplos das primeiras aulas serão modificados para que o gerenciamento do banco de dados seja realizado pelo *Hibernate*. Nesta aula, a operação de inserção no banco de dados será explicada passo a passo.

4.1. Instalando e Configurando o *Hibernate*

Primeiramente, os arquivos necessários para configurar o *Hibernate* devem ser baixadospara serem referenciados no projeto. Neste exemplo, será utilizada a versão 3.6.10 do *Hibernate* disponibilizada para *download* no site http://hibernate.org/.

Após o *download*, o arquivo do *Hibernate*deve ser descompactado para que os arquivos e pastas visualizados na Figura 39 estejam disponíveis para a configuração.

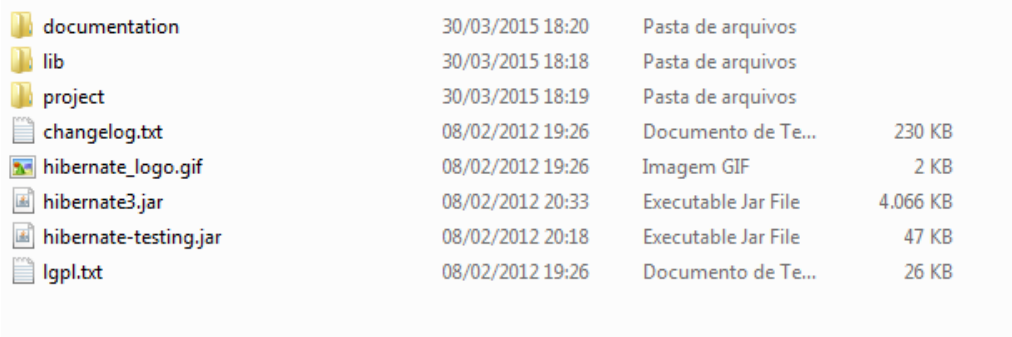


Figura 39. Os arquivos disponíveis do *Hibernate*.

Por seguinte, alguns arquivos JAR deverão ser selecionados para a configuração:

- *hibernate3.jar* - Esse arquivo se encontra no diretório raiz;
- todos os arquivos encontrados dentro da pasta *lib>required*;
- o arquivo JAR encontrado dentro pasta *jpa*.

A Figura 40 mostra todos os arquivos JAR que foram separados para serem referenciados ao projeto. Para referenciar esses arquivos ao projeto, clique com o botão direito do mouse em cima do projeto; selecione a opção *Properties>Java Build Path>Libraries*.Para reunir todos os arquivos JAR referentes ao *Hibernate*uma *Library* deve ser criada.Para isso, as opções*addLibrary>UserLibrary>UserLibraries>New*devemselecionadas, consecutivamente. Logo após,o nome *Hibernate* deve ser informado como nome da *Library*. Esse nome foi escolhido porque todas as JAR

que serão adicionadas são utilizadas para a configuração do *framework*.

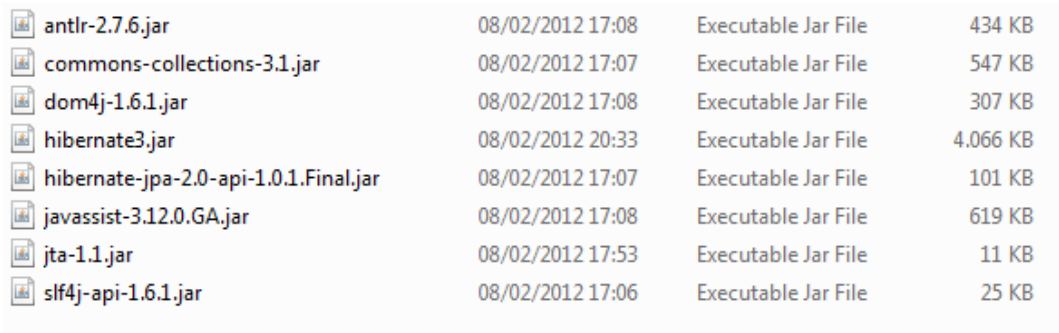


Figura 40. Os arquivos JAR do *Hibernate* para serem referenciados.

Após a criação da *Library Hibernate*, é preciso adicionar a ela os arquivos JAR referentes ao *Hibernate*. Para isso selecione a opção *addJAR*, adicione todos os arquivos JAR já separadose clique em ok.

AFigura 41 mostra o resultado dos arquivos referenciados no projeto. Todos os JAR adicionados ficaram associados à *Library Hibernate*que foi criada justamente para agrupar os arquivos JAR necessário à configuração do*Hibernate.* O arquivo JAR do conector MySQL continuará sendo necessário para estabelecer uma conexão com o banco de dados, por isso o mesmo também deve estar referenciado no *Java Build Path do projeto.*

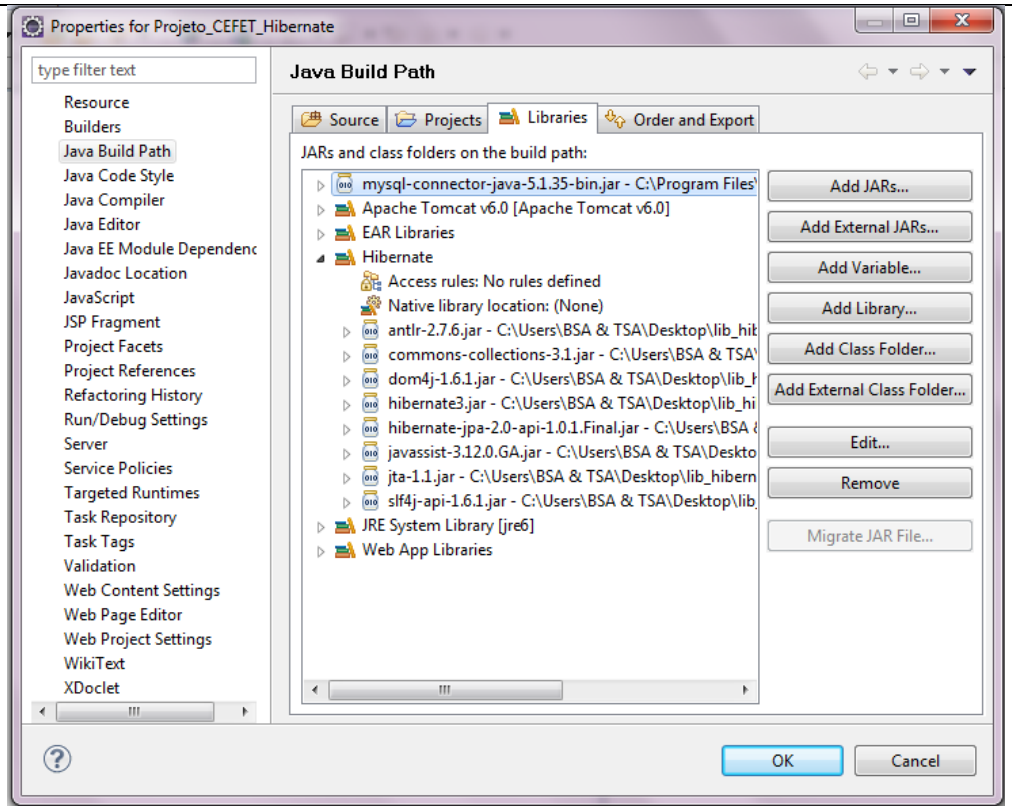


Figura 41. As bibliotecas referenciadas no projeto.

Todos arquivos JAR referenciados no *Java Build Path* do projeto devem ser copiados para a pasta *lib* da instalação doTomcat. Caso os arquivos JAR não sejam copiados um erro ocorrerá no momento de execução do seu projeto.

4.2. Criando o Arquivo XML de Configuração do Hibernate

Para iniciar as modificações no código do projeto, um arquivo XML de configuração deveser criado dentro do pacote *util* com o nome de *Hibernate.cgf.xml*. Para a criação de um arquivo XML clique em *File>New>XML*.

Nesse arquivo XML, estão presentes todas as linhas de código e os parâmetros necessários para estabelecer uma conexão com o banco de dados, conforme Figura 42. Esses códigos representam um modelo básico de configuração, onde as únicas variáveis que devem ser substituídas são os nomes do banco de dados, do usuário e da senha utilizados em seu projeto.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-
//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/locadorasessaodasdez</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">123</property>

        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">update</property>

    </session-factory>
</hibernate-configuration>
```

Figura 42. O arquivo *Hibernate.cgf.xml*.

Repare que a classe *ConexãoBancoDeDados.java* do pacote *dao* usada nas aulas iniciais será substituída por esse arquivo de configuração, pois todas as informações dessa classe são migradas para este arquivo XML usado pelo *Hibernate*.

4.3. Criando Uma Sessão Com o Banco de Dados

Dentro do pacote *útil* é necessário criar a classe *HibernateUtil*, que é responsável por estabelecer uma sessão de comunicação com o banco de dados através de uma conexão JDBC. Essa conexão será armazenada em um atributo estático *session* do tipo *SessionFactory*, que tem a função demanter os mapeamentos e as configurações do *Hibernate*em tempo de execução. A Figura 43 mostra os códigos pertencentes a classe *HibernateUtil.*

```
package util;

import org.hibernate.SessionFactory;

public class HibernateUtil {

    public static final SessionFactory session = buildSession();

    private static SessionFactory buildSession() {
    try {

        Configuration cfg = new Configuration();
        cfg.configure("util/hibernate.cfg.xml");
        return cfg.buildSessionFactory();

    } catch (Throwable e) {
        throw new ExceptionInInitializerError("Não foi possível estabelecer uma conexão com o banco de dados:" +e);
      }
    }


    public static SessionFactory getSessionFactory(){
        return session;

    }


}
```

Figura 43. O código da classe *HibernateUtil*.

Para estabelecer uma sessão, as configurações estabelecidas no arquivo *Hibernate.cgf.xml*, serão invocadas pelo método *configure*. Para isso o caminho do arquivo XML, *Hibernate.cgf.xml*, deve ser informado como parâmetro neste método da classe *HibernateUtil*. Caso ocorra algum erro no estabelecimento de uma sessão, uma exceção do tipo*ExceptionInInitializerError* será lançada juntamente com uma mensagem de erro. Aindanesta classe, o método*getSessionFactory é* responsável por retornar um objeto do tipo *SessionFactory*.

Após essas alterações no projeto, a visão geral dos diretórios do projeto é mostrada na Figura 44. No pacote *util*, a classe *HibernateUtil*e o arquivo XML, *Hibernate.cgf.xml,* foram criados. Além disso, a *Library Hibernate*foi criada com todos os arquivos JAR necessários. Já no pacote dao, a classe *ConexãoBancoDeDados*foi removida, pois esta foi substituída pelo arquivo de configuração XML do pacote *util*.

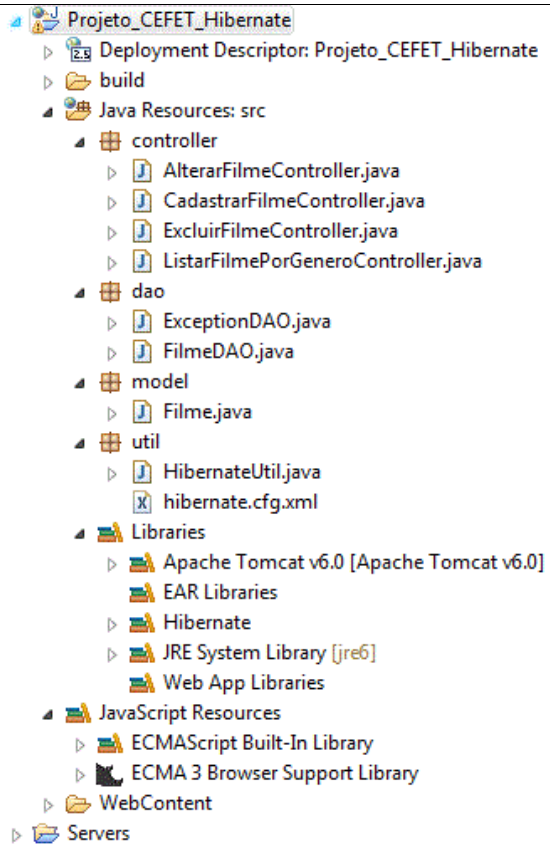


Figura 44. O diretório do projeto utilizando *Hibernate*.

4.4. Mapeando Uma Tabela

Para que o *Hibernate* identifique as tabelas que são utilizadas por uma aplicação, algumas anotações devem ser feitas nas classes pertencentes ao pacote da *model* e o elemento *hibernate-mapping* é acrescentado no arquivo de configuração *Hibernate.cgf.xml*. Neste exemplo, o mapeamento da tabela *Filme* será implementado.

É necessário modificar a classe *Filme* inserindo anotações que serão utilizadas para o mapeamento objeto/relacional. Essas anotações pertencem ao pacote *javax.persistence* e permitem que o *Hibernate* crie uma tabela com os respectivos atributos de uma classe segundo as descrições e as restrições anotadas.A Figura 45 mostra as anotações inseridas na classe *Filme*.

```
@Entity
@Table(name="filme")
public class Filme {


    @Id
    @GeneratedValue
    @Column(name="id_filme")
    private int id;

    @Column(length=50, nullable=false)
    private String titulo;

    @Column(length=500, nullable=false)
    private String descricao;

    @Column(nullable=false)
    private int duracao;

    @Column(length=20,nullable=false)
    private String genero;

    @Column(length=10,nullable=false)
    private String oscar;
```

Figura 45. As anotações na classe *Filme*.

As anotações disponíveis para utilização pelo *Hibernate* são:

- *@Entity* - Identifica uma classe como entidade. Por padrão, o nome é o próprio nome da classe;
- *@Table* - Mapeia e define um nome(*name*) para a tabela. O valor padrão do nome é o nome da classe;
- *@Id* - Define a chave primária de uma tabela;
- *@Column* - Mapeia e define um nome (*name*) para a coluna da tabela. É utilizado o atributo *length* para definir o tamanho do campo e o atributo *@nullable* para definir se o valor do campo pode ser nulo ou não. O valor padrão do nome é nome do atributo da classe.
- *@GeneratedValue* - Define a propriedade de auto incremento da chave primária.

Além dessas, outras anotações e atributos podem ser acrescentados no mapeamento da entidade. É possível aplicar restrições e especificar as características para que o processo de mapeamento objeto/relacional se a ajuste as necessidades do usuário referentes ao banco de dados.

Por fim, o elemento *hibernate-mapping* juntamente com o atributo *class* é acrescentado no arquivo *Hibernate.cgf.xml* para apontar para a classe *Filme*, que contém as anotações. A Figura 46 mostra o mapeamento. Todas as classes utilizadas em seu projeto devem seguir os mesmos passos explicados com a classe *Filme*: primeiro as anotações e depois o mapeamento no arquivo XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-
//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/locadorasessaodasdez</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">123</property>

        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
        <property name="hibernate.hbm2ddl.auto">update</property>


        <!-- Mapeamento das Classes no Model -->
        <mapping class="model.Filme"/>

    </session-factory>
</hibernate-configuration>
```

Figura 46. O mapeamento das classes da camada do modelo.

4.5. Modificando a classe DAO

Como agora o *Hibernate* será utilizado como intérprete entre a tabela e a classe *Filme*, a classe *FilmeDAO* deve ser reestruturada de forma mais simples. Para isso será modificado o método *cadastrar* da classe *FilmeDAO*.A Figura 47 possui todos os códigos usados para inserir um registro de um filme no banco de dados.

```
public class FilmeDAO {

    private Session session;

        public void cadastroFilme(Filme filme) throws ExceptionDAO{
            session = HibernateUtil.getSessionFactory().openSession();

            try{
                session.beginTransaction();
                session.save(filme);
                session.getTransaction().commit();
            }catch (Exception e) {
                session.getTransaction().rollback();
                throw new ExceptionDAO("Erro ao cadastrar filme: " +e);
            }finally{
                session.close();
            }
        }
```

Figura 47. A classe *FilmeDAO*utilizando o *Hibernate*.

Inicialmente, um atributo *session* do tipo *SessionFactory* na classe *FilmeDAO* é definido. Esse atributo é utilizado por todos os métodos da classe *FilmeDAO* para receber uma sessão iniciada. Para iniciar uma sessão, o método estático *getSessionFactory* da classe *HibernateUtil* é chamado, retornando um objeto do tipo *SessionFactory*. Através do método *openSession*, a sessão é aberta para que uma operação (inserção, alteração, exclusão, busca) seja executada no banco de dados.

Para executar uma operação no banco de dados, o método *beginTransaction* inicia um processo de transação. As operações são efetuadas pelas transações através do método *commit*. Em caso de erro na transação, este é capturado pelo bloco catch que, por sua vez, lança uma exceção do tipo *ExceptionDAO* e chama o método rollback, responsável por desfazer a transação. Independentemente, se a transação for concretizada com sucesso ou não, o bloco *finally* é executado, chamando o método *close* que é encarregado de fechar a sessão.Essa é a descrição genérica do funcionamento dos métodos da classe *FilmeDAO* para executar uma operação no banco de dados.

4.6. Inserindo um Novo Registro No Banco de Dados

O método *cadastroFilme* é responsável por armazenar novos

registros de um filme no banco de dados, recebendo como parâmetro da camada de modelo um objeto do tipo *Filme*. Esse método executa os passos genéricos de funcionamento explicados anteriormente e atravésdo método *save* do objeto *session a* operação de inserção é chamada recebendo como parâmetro um objeto do tipo filme.

Quando o projeto é executado, o *Hibernate* cria uma tabela filme no banco de dados locadora *sessaodasdez* com as colunas *id*, *título*, *descrição*, *duração* e *gênero* de acordo com as especificações e restrições anotadas na classe *Filme* e mapeadas pelo arquivo XML do *Hibernate*. Atribuindo o valor *true* a propriedade *hibernate.show_sql* do arquivo *Hibernate.cgf.xml*, no console do Eclipse é possível observar o comando SQL que foi executado pelo *Hibernate* ao realizar uma operação de inserção, conforme a Figura 48.

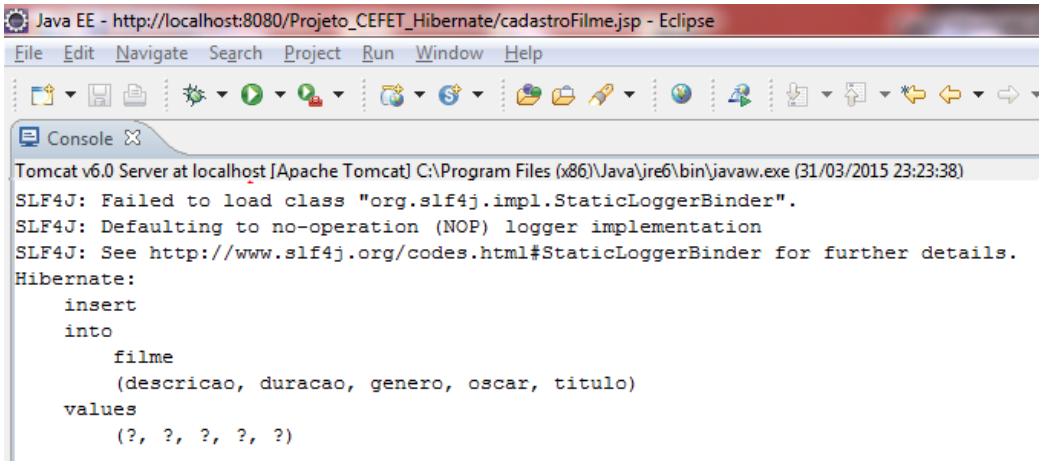


Figura 48. O comando SQL executado pelo *Hibernate*.

Na classe *FilmeDao*, abordada anteriormente, o valor de cada atributo da classe *Filme* é atribuído, um a um, a um objeto do tipo *PreparedStatement* para que o método *execute* possa ser chamado. Com o uso do *Hibernate* essas atribuições não são mais necessárias e a aplicação se ajusta de forma mais simples ao banco de dados. Isso ocorre, justamente, pois o *Hibernate* tem a incumbência de mapear o objeto do tipo filme para a tabela filme do banco de dados.

É possível observar o banco de dados *locadorasessaodasdez* sem nenhuma tabela associadaem um momento antes da execução do

projeto. Após a execução do projeto, verificamos que a tabela *Filme* foi criada com as respectivas colunas e que o registro foi inserido com sucesso. As colunas foram criadas segundo as anotações feitas na classe *Filme*.

5. Realizando um CRUD em um Projeto com *Hibernate*

Nesta aula, as operações de alteração, exclusão e busca da classe *FilmeDAO* serão modificadas para o uso do *Hibernate*, complementando a implementação de um CRUD.

5.1. Alteração de Dados (*Update*)

O método *alterarFilme* é responsável por atualizar as colunas *título*, *duração* e *descrição* de um registro no banco de dados associado ao valor de uma determinada chave primária, conforme os códigos da Figura 49.

```
public void alterarFilme(Filme filme) throws ExceptionDAO{
        session = HibernateUtil.getSessionFactory().openSession();

        try{
            session = HibernateUtil.getSessionFactory().openSession();
            session.beginTransaction();
            session.saveOrUpdate(filme);
            session.getTransaction().commit();
        }catch (Exception e) {
            session.getTransaction().rollback();
            throw new ExceptionDAO("Erro ao alterar filme: " +e);
        }finally{
            session.close();
        }
    }
```

Figura 49. O método *alterarFilme*usando *Hibernate*.

Esse método executa os passos genéricos de funcionamento explicados na aula anterior e através do método *saveOrUpdate* do objeto de tipo *Session* a operação de alteração é chamada recebendo como parâmetro um objeto do tipo filme.A Figura 50 mostra o SQL gerado após a execução do projeto.

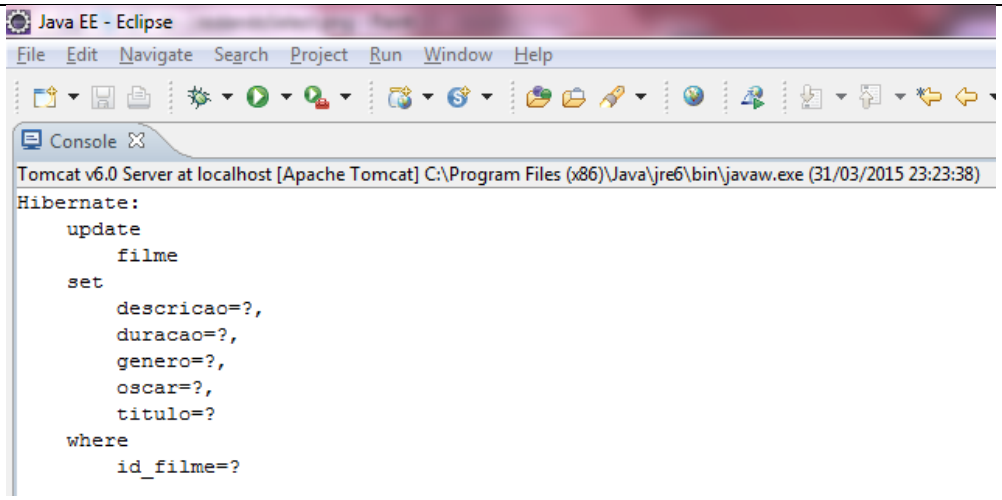


Figura 50. O comando SQL executado pelo *Hibernate*.

5.2. Exclusão de Dados (*Delete*)

O método *excluirFilme* exclui um registro do banco de dados associado ao valor de uma chave primária, conforme os códigos da Figura 51. O método *load* é usado para carregar todos os dados referentes ao objeto *filme* com o valor do *id* associado. Esse carregamento das informações é necessário, pois o método *excluirFilme* recebeu como parâmetro apenas o valor do id.

```
public void excluirFilme(int id) throws ExceptionDAO{
            session = HibernateUtil.getSessionFactory().openSession();

            try{
                session = HibernateUtil.getSessionFactory().openSession();
                session.beginTransaction();
                Filme filme = (Filme) session.load(Filme.class, id);
                session.delete(filme);
                session.getTransaction().commit();
            }catch (Exception e) {
                session.getTransaction().rollback();
                throw new ExceptionDAO("Erro ao exluir filme: " +e);
            }
            finally{
                session.close();

            }
        }
```

Figura 51. O método *excluirFilme*usando *Hibernate*.

O funcionamento dessa operação também segue os passos explicados na aula anterior e através do método *delete* do objeto de

tipo*Session* a operação de exclusão é chamada recebendo como parâmetro um objeto do tipo filme. A Figura 52 mostra o SQL gerado após a execução do projeto.

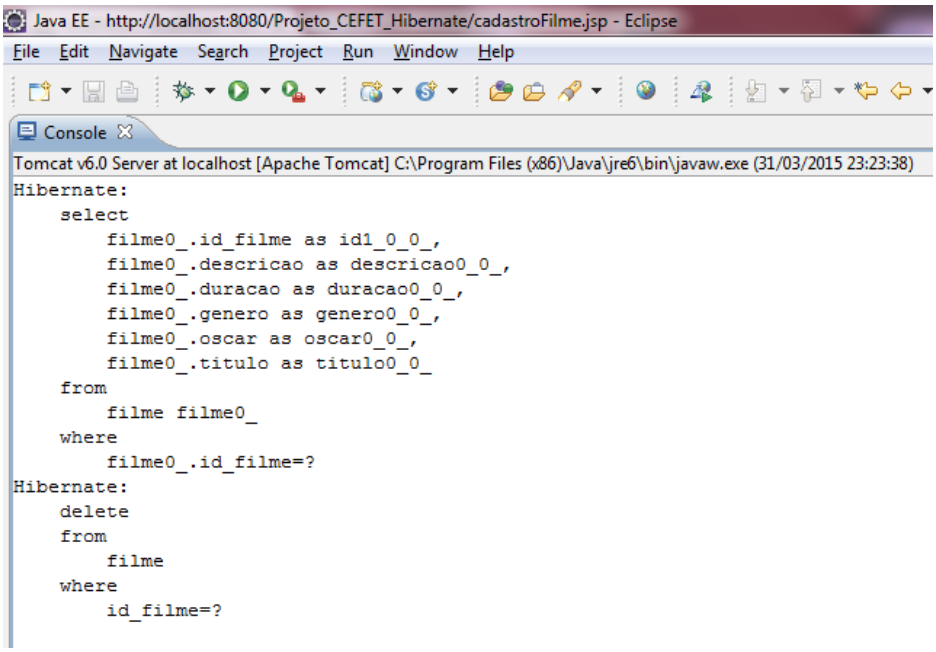


Figura 52. O comando SQL executado pelo *Hibernate*.

5.3. Recuperação de Dados (*Read*)

O método *listarFilmesPorGenero* traz todos os registros da tabela *Filme* em que o valor do campo *gênero* do banco de dados seja igual ao valor do parâmetro *gênero* passado para o método, conforme os códigos da Figura 53. Nesse método, a operação de busca usa uma API *Criteria*, importada por *org.hibernate.Criteria*, que permite construir consultas estruturadas.

```
    @SuppressWarnings("unchecked")
    public ArrayList<Filme> listarFilmesPorGenero(String genero) throws ExceptionDAO{
            session = HibernateUtil.getSessionFactory().openSession();

            try{
                Criteria criteria = session.createCriteria(Filme.class);
                criteria.add(Restrictions.eq("genero", genero));
                ArrayList<Filme> listaDeFilmePorGenero = (ArrayList<Filme>) criteria.list();
                return listaDeFilmePorGenero;
            }catch (Exception e) {
                session.getTransaction().rollback();
                throw new ExceptionDAO("Erro ao listar filme: " +e);
            }finally{
                session.close();

            }
        }
}
```

Figura 53. O método *listarFilmePorGenero*usando *Hibernate*.

O método *createCriteria* recebe como parâmetro a classe *Filme*. Quando o *Hibernate* executar uma consulta, um objeto do tipo *Criteria* será criado retornando instâncias da classe *Filme*. Para que apenas os dados dos filmes que possuam o gênero desejado sejam retornados, a API *Criteria* permite o uso de restrições de consulta. O método *add* é usado para adicionar uma restrição que juntamente com método *Restrictions.eq* definem uma restrição de igualdade, ou seja, a consulta realizada retorna apenas os dados da tabela *Filme* em que a coluna gênero seja igual a variável gênero. A Figura 54 mostra o SQL gerado após a execução do projeto.

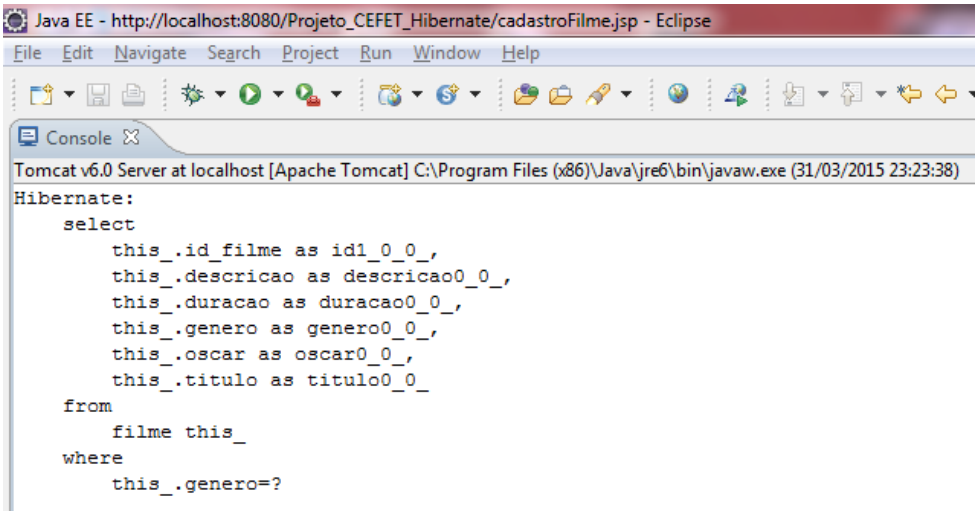


Figura 54. O comando SQL executado pelo *Hibernate*.

Para suprimir a mensagem de alerta de compilação emitida por

uma operação de *casting*, a anotação *@SuppressWarnings("unchecked")* é utilizada. Essa anotação informa que o programador se assegura que o tipo de objeto retornado na consulta pelo método *list* pode ser convertido para o tipo *ArrayList<Filme>.* Na Figura 55 repare que se a notação for retirada uma mensagem de alerta aparecerá em amarelo.

```
62
63          public ArrayList<Filme> listarFilmesPorGenero(String genero) throws ExceptionDAO{
64                  session = HibernateUtil.getSessionFactory().openSession();
65
66                  try{
67                      Criteria criteria = session.createCriteria(Filme.class);
68                      criteria.add(Restrictions.eq("genero", genero));
69  Type safety: Unchecked cast from List to ArrayList<Filme>taDeFilmePorGenero = (ArrayList<Filme>) criteria.list();
70                      return listaDeFilmePorGenero;
71                  }catch (Exception e) {
72                      throw new ExceptionDAO("Erro ao listar filme: " +e);
73                  }finally{
74                      session.close();
75
76                  }
77              }
78      }
```

Figura 55. A mensagem de alerta gerada.

6. *Struts*

O *Struts* é um framework gratuito que utiliza o padrão de projeto MVC para criação de páginas *web* utilizando a tecnologia Java EE. O *Struts*permite a separação da camada de HTML da camada de processamento, através de classes pré-definidas. Para representação da camada HTML é utilizada a classe *ActionForm* e para a representação da camada de processamento é utilizada a classe *Action*. O uso do *Struts* não é recomendado para pequenas aplicações.

O *Struts* oferece em sua biblioteca uma *servlet* controladora chamada de *ActionServlet*.Para instalar o *Struts* é necessário realizar o *download* em *https://struts.apache.org/download.cgi#struts2320*. Em seguida, descompacte o arquivo *struts-2.3.20-all.zip*em qualquer diretório.

6.1. Criando um Projeto Java EE com *Struts*2

O *Struts*é um *framework* para ser usado em um projeto de páginas *web* dinâmicas, para isso será necessário criar um *dynamic web project* no eclipse. Crie um projeto chamado *HelloCefetStruts*e avance até aparecer a tela *Web Module* onde é necessário marcar a caixa *generate web.xml deploymentdescriptor*(Figura 56).

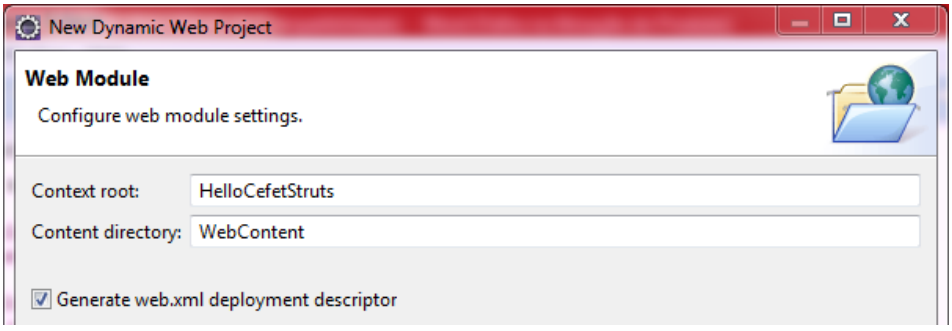


Figura 56. Marque a opção para gerar o web.xml *descriptor*.

Em seguida, copie dentro da pasta *WebContente/WEB-INF/lib* do

projeto criado as bibliotecas especificadas na Figura 57. Essas bibliotecas estão no diretório *struts2.3.20/lib*da instalação do Struts2. Para facilitar a criação de novos projetos, é recomendado a cópia dessas bibliotecas em uma pasta separada de fácil acesso.

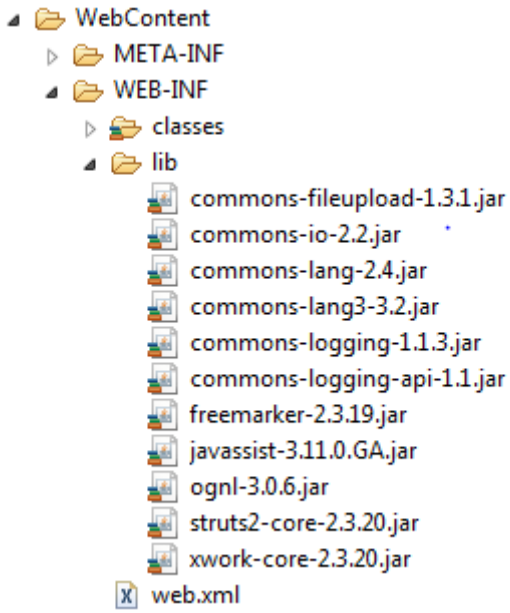


Figura 57. As bibliotecas do *struts* que precisam ser copiadas para qualquer projeto *web*.

O próximo passo é a criação de uma classe de ação no diretório *Java Resources/src* do projeto. Crie uma classe com o nome *HelloAction*no pacote *br.cefet.actions*(o pacote onde todas as ações do projeto ficarão). Na classe é necessário especificar os atributos que serão mapeados da camada de Visão (um arquivo JSP), assim como todos os *getters* e *setters*(métodos que serão usados pelo Struts para armazenar e retornar informações).

Além disso, a classe deverá ter uma função chamada *execute*, que será executada quando uma ação for executada no JSP. A função *execute*retornará um valor em *String*que será mapeado para que uma página de resposta seja exibida ao usuário. O código da classe deve estar de acordo com a Figura 58.

```
package br.cefet.actions;

public class HelloAction {

    private String name;

    public String execute() throws Exception {
        return "success";
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Figura 58. O código da classe *HelloAction*.

O próximo passo é criar a pasta de nome *classes* dentro do diretório *WebContent/WEB-INF*. Dentro dessa pasta também é necessário criar um arquivo xml chamado *struts.xml* que fará o mapeamento de ações do projeto. Agora, é necessário criar as duas páginas JSP do projeto: a primeira chamada de *index.jsp* onde o usuário digitará um campo texto e o enviará através de um clique em um botão; e o *response.jsp*, que é a página de resposta ao clique do usuário. O código da página *index*pode ser visto na Figura 59, enquanto o código da página *response* pode ser visto na Figura 60.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
   <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Hello CEFET</title>
</head>
<body>
   <h1>Hello CEFET com Struts 2</h1>
   <form action="hello">
      <label for="name">Digite o seu nome</label><br/>
      <input type="text" name="name"/>
      <input type="submit" value="Say Hello"/>
   </form>
</body>
</html>
```

Figura 59. O código da página *index.jsp*.

Na página *index*é importante definir a ação de formulário *action* com o valor *hello*. Esse valor será mapeado no *struts.xml* para quando o formulário for executado, chamar a classe*HelloAction*onde o valor para o nome digitado pelo usuário será armazenado. O *struts*ao perceber que uma ação foi executada, pega a resposta enviada pela ação (o retorno do método *execute*) e retorna uma página de resposta associada no arquivo de mapeamento. Essa página de resposta, ainda pode acessar os valores da classe de ação, para retornar valores (de um banco de dados por exemplo).

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="s" uri="/struts-tags" %>

<html>
<head>
<title>Hello CEFET</title>
</head>
<body>
   Hello CEFET, <s:property value="name"/>
</body>
</html>
```

Figura 60. O código da página *response.jsp*.

O mapeamento das ações e páginas a serem exibidas é realizado no arquivo *struts.xml*. Para se mapear uma ação é necessário utilizar a *tagaction*onde o nome da ação (*name*no xml) é o nome da ação executado no JSP; a *tagclass* é onde a classe da ação em java é especificada; a tag*method* indica qual a função que será usada na execução da ação; e, por fim, a *tagresult*que mapeia o retorno da função *execute* e retorna uma página JSP como resposta. Na Figura 61 é possível ver o mapeamento no arquivo *struts.xml*.

Agora, o último passo é fazer a configuração do arquivo *web.xml* gerado ao se criar o projeto. No arquivo de configuração é preciso informar que as *servlets* serão todas gerenciadas pelo *struts*. Também será definida como página inicial o *index.jsp*.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
   "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
<constant name="struts.devMode" value="true" />
   <package name="helloworld" extends="struts-default">

      <action name="hello"
            class="br.cefet.actions.HelloAction"
            method="execute">
            <result name="success">/response.jsp</result>
      </action>

   </package>
</struts>
```

Figura 61. O código do arquivo de mapeamento *struts.xml*.

No arquivo *web.xml*(Figura 62), a *tagwelcome-file* define a página inicial; e a *tagfilter* faz com que o *struts* seja o responsável pelo controle do fluxo de informação entre o modelo e a visão, não necessitando mais das *servlets*tradicionais como no modelo MVC tradicional.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
   http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
   id="WebApp_ID" version="3.0">

   <display-name>Struts 2</display-name>
   <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
   </welcome-file-list>
   <filter>
      <filter-name>struts2</filter-name>
      <filter-class>
         org.apache.struts2.dispatcher.FilterDispatcher
      </filter-class>
   </filter>

   <filter-mapping>
      <filter-name>struts2</filter-name>
      <url-pattern>/*</url-pattern>
   </filter-mapping>
</web-app>
```

Figura 62. O código do arquivo *web.xml*.

Para executar o projeto de páginas dinâmica desenvolvido é necessário exportar o arquivo *war*do projeto clicando com o botão direito

no nome do projeto e ir em *Export>WAR file*(Figura 63). Em seguida mova o arquivo gerado para a pasta *webapps* do diretório de instalação do tomcat.

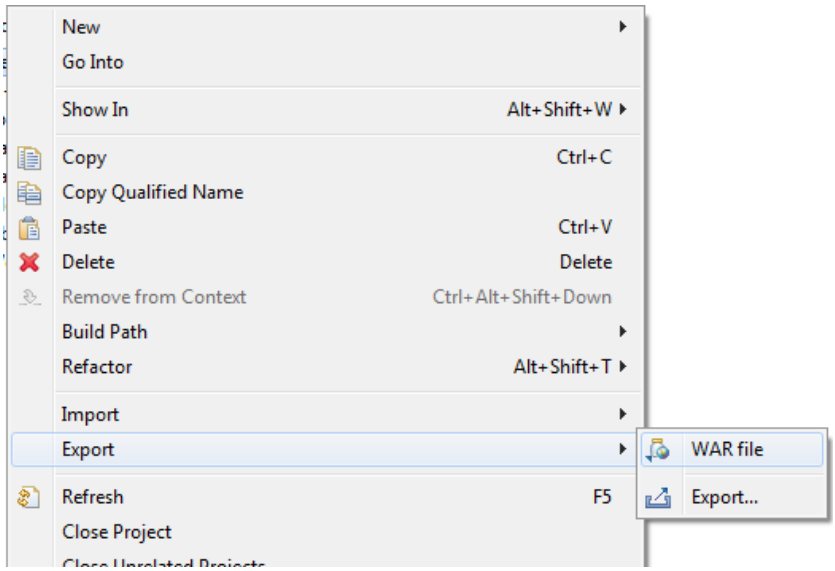


Figura 63. Projeto>*Export>WAR File*.

Agora basta digitar no navegador o endereço http://localhost:8080/HelloCefetStruts/index.jsp, que a página index será exibida conforme a Figura 64.

Figura 64. A página *index.jsp* em execução.

Ao digitar qualquer informação e clicar no botão *SayHello*, a ação será verificada no arquivo de mapeamento e o *struts* executará a classe de ação indicada no arquivo. Uma vez que a ação for executada, este retornará uma *String* que disparará a página de resposta. Para o

exemplo desenvolvido, a página de resposta em execução (com valor etec de entrada) pode ser vista na Figura 65.

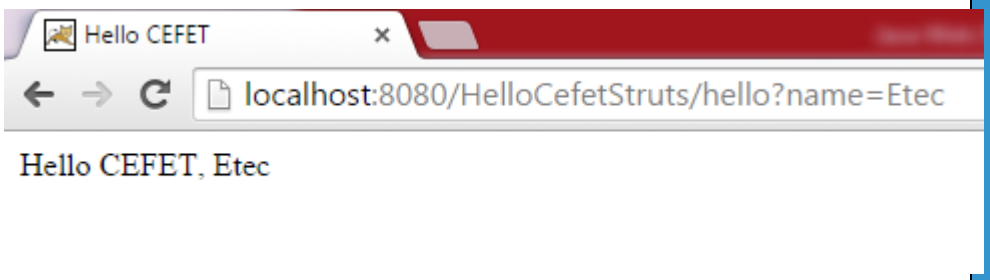


Figura 65. A página *response.jsp* em execução.

7. *Struts*com acesso a Banco de Dados

Para se fazer acesso a um banco de dados utilizando o*Struts*, basta programa a camada de acesso a dados a partir da camada de modelo como foi realizado na aula 2. Uma vez que o *Struts*provê uma *servlet* genérica de controle de fluxo de informação entre a camada de visão e a camada controladora, não existe diferenças significativas no acesso ao banco de dados.

7.1. Criando um Projeto Java EE com *Struts* 2 e acesso ao BD

Nesta aula será utilizado o sistema de locadora das aulas desenvolvido nas aulas anteriores onde o usuário cadastrará um filme através de uma página JSP e o mesmo será armazenado em um banco de dados.

Então, crie um novo projeto dinâmico conforme as aulas anteriores, adicione as bibliotecas do *Struts* e crie o *cadastroFilme.jsp*. Em seguida programe a página JSP para realizar o cadastro de um Filme em uma locadora e uma página de resposta informando que o filme foi cadastrado com sucesso.

Em seguida crie a classe *FilmeAction*, com os atributos *título*, *duração*, *gênero*, *oscar* e *descrição*. Defina também todos os métodos *gets* e *sets* para os atributos.Antes de criar o método *execute*, crie a classe da camada de Modelo chamada *Filme*e a classe da camada de acesso a dados *FilmeDAO* (conforme as aulas iniciais).

Agora é necessário programar o método *execute* de forma que,a cada ação executada, o *struts* possa pegar as informações oriundas da página JSP e inseri-las em um objeto do tipo *Filme*, que em seguida fará chamada ao método para inserção do registro no banco de dados.O código do método *execute* pode ser visto na Figura 66.

```
public String execute() throws Exception {
    Filme filme = new Filme();
    filme.setTitulo(this.titulo);
    filme.setDescricao(this.descricao);
    filme.setGenero(this.genero);
    filme.setOscar(this.oscar);
    filme.setDuracao(Integer.parseInt(this.duracao));
    filme.cadastroFilme(filme);
    return "success";
}
```

Figura 66. O método *execute* da classe *FilmeAction*.

Após a criação das classes, é preciso configurar o arquivo de mapeamento *struts.xml*conforme a aula 6. A *tagaction* foi configurada com o nome *cadastrarFilme*; a *tagclass* foi configurada com o endereço completo da classe *FilmeAction* (incluindo o pacote); e a página de *response.jsp* foi configurada como resposta.A Figura 67 mostra o código do *struts.xml* para este exemplo.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
   "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
<constant name="struts.devMode" value="true" />
   <package name="cadastro" extends="struts-default">

      <action name="cadastrarFilme"
            class="br.cefet.locadora.actions.FilmeAction"
            method="execute">
            <result name="success">/response.jsp</result>
      </action>

   </package>
</struts>
```

Figura 67. O arquivo *struts.xml*para o exemplo do cadastro de filme.

Ao executar o projeto Java EE com o *Struts*, a página de cadastro aparecerá para que seja digitado as informações necessárias para o cadastro de um filme. Ao enviar as informações, o filme será armazenado no banco de dados e a página de resposta (Figura 68) será exibida.

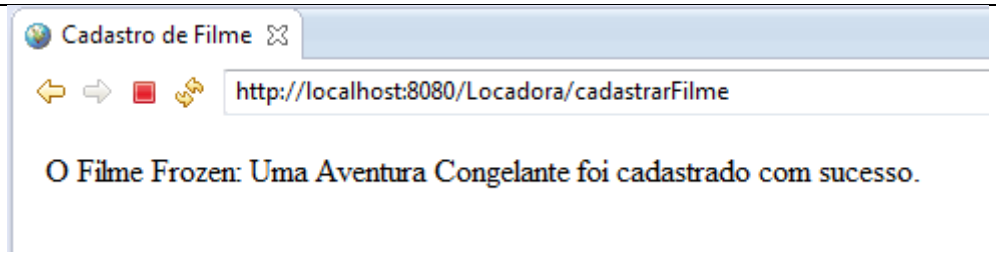


Figura 68. A página de resposta para o cadastro de filme.

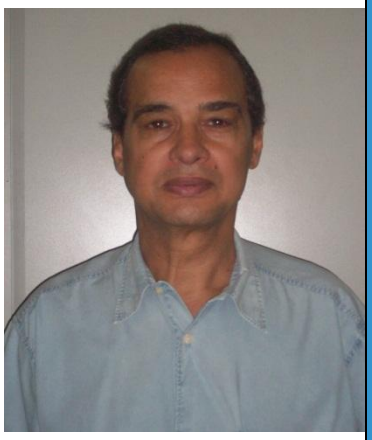
8. Referências Bibliográficas

Kurniawan, B. (2002). Java para Web com Servlets, JSP e EJB. Editora Ciência Moderna, Rio de Janeiro, 2002.

Fields, D. K., Kolb, M.A. (2000). Desenvolvimento na Web com JavaServerPages. Editora Ciência Moderna, Rio de Janeiro, 2000.

Metlapalli, P. (2010). Páginas JavaServer (JSP). Editora LTC, Rio de Janeiro, 2010.

COM A PALAVRA, O COORDENADOR GERAL ...
Professor D. Sc. Mauro Godinho Gonçalves



A implementação de curso a distância sugere como requisito de desenvolvimento o conhecimento da tecnologia digital e das ferramentas que possibilitam a interação entre seus interlocutores.

Nesta obra sobre educação tecnológica, são discutidas as diversas possibilidades que a educação a distância oferece e apresentada a importância do mundo digital para sua implementação. Discute-se, também, as tecnologias da informação e comunicação e as várias redes sociais muito utilizadas na comunicação entre as pessoas atualmente.

O autor deste trabalho é professor com reconhecida competência nesta área, lecionando no ensino técnico, superior e pós. Assim, espero que apreciem o seu trabalho.