



João Paulo Martins Duarte é analista de sistemas na Evoluti, atuando com Java há mais de um ano com foco no desenvolvimento de sistemas web.

Struts 2 + AJAX + JPA + Spring

O Struts é uma das frameworks para desenvolvimento web mais utilizadas, aqui neste artigo apresentaremos sua nova versão o Struts 2, utilizando a Java Persistence API para realizarmos a persistencia de nossos objetos e ainda integrando essas duas tecnologias com o Spring 2.

Para este artigo estaremos utilizando a nova versão do Web Tools Project do eclipse, o Eclipse Europa, o site para download esta no final do artigo assim como os outros links para as frameworks que iremos abordar neste artigo.

Esta nova versão do WTP vem mais recheada que o normal, ela já vem com algumas perspectivas a mais, que é o caso da JPA Development e a Database Development entre outras.

Também nesta versão temos suporte para a versão 6 do Tomcat entre outras diferenças, no final do artigo tem o link para quem quer saber mais a respeito do Eclipse Europa.

O processo de instalação dele é igual ao do antigo, basta descompactar o arquivo baixado do site www.eclipse.org para uma pasta eclipse.

Estamos utilizando aqui o MySQL 5 como banco de dados para a aplicação exemplo de nosso artigo.

Para nosso artigo criamos um projeto **Dynamic Web Project** que será chamado de **Cadastro**, nele teremos os pacotes **br.jm.actions** para nossas actions, **br.jm.entidades** para as entidades que serão mapeadas utilizando a JPA, **br.jm.servico** que terão nossas classes e interfaces que proveram o acesso do banco de dados para nossas actions e por fim o pacote **br.jm.persistencia** onde ficara nossos daos.

Estamos utilizando a versão 2.0.6 do Struts 2, já esta disponivel a versão 2.0.8, mas aconselho que se baixe essa versão para evitar problemas na execução da aplicação exemplo.

E para a JPA estamos usando a implementação do Hibernate 3

A lista de dependencias do projeto esta na figura 5, são as libs necessárias para o funcionamento das frameworks que utilizaremos, foi necessário colocar o jar do dwr, pois o Struts 2 utiliza o dwr para validações.

Quando fizer o download do spring, faça o download do spring com dependencias, assim vai ter disponivel as libs para integrar o Spring com outras tecnologias.

Por se tratar de uma aplicação web, é interessante que estas dependencias sejam copiadas para o diretório *nomeProjeto/WebContent/WEB-INF/lib*, assim você não terá problemas quando fazer o deploy da aplicação.

Nossa aplicação de exemplo terá uma tela de login que validará um usuário e uma senha, e ira drecionar caso o usuário seja válido para uma tela onde o usuário possa cadastrar um novo

contato e listar os contatos já cadastrados, também poderá remover qualquer contato ou toda sua lista, apenas para demonstrarmos a utilização do Struts 2, JPA e Spring.

Para que o login funcione, é necessário fazer um insert na tabela *CT_USUARIOWEB* com um login e uma senha para efetuarmos o login.

Parte I – O Struts 2

Houve muitas mudanças nesta versão do Struts, quem já era familiarizado com a versão 1.x dessa framework com certeza pode se sentir confuso no começo, ao contrário de quem já utilizou o WebWork, pois, muito desta framework foi agregado ao Struts 1x para a criação do Struts2.

No Struts 1.3.8, a versão antiga do Struts, é necessário utilizar funções javascript para que sua aplicação web tenha funcionalidades AJAX. Embora frameworks como o DWR auxiliem na implementação do AJAX, ainda sim temos um contato com o javascript que pode ser cansativo, mas agora no Struts 2, podemos criar páginas com conteúdo dinâmico sem esforço, e sem javascript como será visto neste artigo.

O Struts 2 continua uma framework MVC, mas diferente de seu antecessor, nos traz facilidades na integração de AJAX entre outras como para geração de relatórios com o Jasper Report, por exemplo.

E agora temos interceptors, que é algo novo no Struts, mas já usado no WebWork, interceptors são objetos que são executados antes e/ou depois da execução das actions. Este login por exemplo, poderia ser um interceptor que executaria antes da action de listar contatos e validaria se existe um usuario válido na session e se o mesmo tivesse permissão de acesso nessa lista e o direcionaria para um login caso não tenha ou seguiria o fluxo normal.

O assunto quanto a interceptors é longo e não abordaremos neste artigo apenas usaremos o interceptor padrão do Struts para que qualquer exception que possa ocorrer em nossa aplicação seja redirecionada para a página *erroServicoContato.jsp* no caso da action de login. Veja mais sobre interceptors no fim do artigo links a respeito.

Actions e o struts.xml

Nossa action que realiza o login do usuário, chamamos de *LoginAction*, como na listagem 1.

```
package br.jm.actions;

import br.jm.persistencia.LoginDAO;
import br.jm.servico.LoginImpl;

import com.opensymphony.xwork2.ActionSupport;

public class LoginAction extends ActionSupport{

    private LoginImpl servico;
    private String login;
    private String senha;

    public LoginAction() {

    }

    public LoginAction(LoginImpl servico) {
        this.servico = servico;
    }

    public String validaLogin() {
        if(servico.valida(login, senha)) {
```

```

        return "valido";
    }
    return "invalido";
}

//métodos getters e setters
}

```

Listagem 1 – LoginAction.java

Você pode reparar que nossa action estende a classe ActionSupport, estamos herdando desta classe para que possamos utilizar as funções de validações do struts, pois temos as propriedades *login* e *senha* que irão ter suas entradas validadas.

No Struts 2 você pode desenvolver actions que sejam simples POJOS(Plain Old Java Objects) sem herdar de nenhuma classe, veremos um exemplo mais adiante.

O atributo **servico** fornecerá nosso acesso ao banco de dados sempre que necessário e este será injetado pelo Spring no momento que nossa action for instanciada.

Vale a pena lembrar que para que não haja problemas na instanciação desta action pelo Spring deve-se declarar um construtor default, ou seja, sem parametros, para que as propriedades *login* e *senha* de nossa action sejam acessadas na página *index.jsp* é necessário criar os métodos getters e setters dessas propriedades, isso pode causar erros se não for feito.

Esta action possui apenas o método.

```

public String validaLogin() {
    if(servico.valida(login, senha)) {
        return "valido";
    }
    return "invalido";
}

```

E é ele que iremos mapear no struts.xml, para que um método possa ser mapeado é necessário que o mesmo tenha como tipo de retorno uma string, não passamos nada como parametro para nosso método, pois, estamos utilizando os atributos de nossa action para receber os dados de input do usuário em uma página jsp, por isso que se faz necessário ter os métodos getters e setters de um atributo, pois eles são utilizados para atribuir ou obter um valor de uma propriedade quando acessado pelo jsp.

Para utilizarmos nosso método *validaLogin()* estaremos utilizando a página *index.jsp* para criar um form que será explicado mais adiante para fazermos o input dos dados. Abaixo como ficará a declaração de nossa action no arquivo struts.xml.

```

<action name="login!*" method="{1}"
        class="br.jm.actions.LoginAction">
    <result name="input">/index.jsp</result>
    <result name="valido">/contato.jsp</result>
    <result name="invalido">/erroLogin.jsp</result>
</action>

```

Aqui estamos criando uma action chamada *login* e estamos configurando o primeiro método desta action para ser acessado utilizando o *parametro method* apontando que deve-se utilizar o primeiro método desta action, pode-se observar no nome de nossa action que colocamos um ponto de exclamação e * (login!*), o struts irá fazer um split na string e como tem um asterisco ele irá ler todos os métodos de nossa action, mais adiante temos um exemplo especificando o nome do método a ser mapeado.

Pode parecer estranho, sendo que na versão antiga do Struts tínhamos apenas o método

execute() que tinha dependências com *HttpServletRequest* e *HttpServletResponse*, aqui no Struts 2, não temos essa dependência diretamente, e podemos mapear um ou mais métodos de nossas actions, ou utilizar o método execute() da classe ActionSupport que não possui dependências como antigamente e retorna uma String.

No nosso caso, o retorno *valido* irá direcionar o usuário para a página *contato.jsp* após a execução do método *validaLogin()* e o retorno *invalido* direciona para a página *erroLogin.jsp*, finalmente o retorno *input* é utilizado para possíveis erros que possam acontecer na validação dos campos.

Nossa próxima action é a *InsererContatoAction*, conforme a listagem 2. Esta action é responsável por receber os dados do usuário e executar o método que irá persistir essas informações no nosso banco de dados, para isso temos a propriedade *contato* que será populada por um form na página *contato.jsp*.

```
package br.jm.actions;

import br.jm.entidade.Contato;
import br.jm.servico.ServicoContatoImpl;

import com.opensymphony.xwork2.Action;
import com.opensymphony.xwork2.ActionSupport;

public class InsererContatoAction extends ActionSupport{

    private ServicoContatoImpl servico;
    private Contato contato;

    public InsererContatoAction() {

    }

    public InsererContatoAction(ServicoContatoImpl servico) {
        this.servico = servico;
    }

    public String execute() throws Exception {
        if(hasActionErrors() || hasFieldErrors()) {
            return "input";
        }
        servico.salvarContato(contato);

        return Action.SUCCESS;
    }

    //métodos getters e setters
}
```

Listagem 2 – InsererContatoAction.java

Repare no retorno de nosso método execute(). O struts disponibiliza alguns retornos default como, SUCCESS, ERROR, etc pela interface com.opensymphony.xwork2.Action.

Nesta action sobreescrevemos o método execute() da classe **ActionSupport**, como pode ser observado no mapeamento da mesma no struts.xml

```
<action name="insereContato"
        class="br.jm.actions.InsererContatoAction">

    <result name="input">/contato.jsp</result>
    <result>/contato.jsp</result>
</action>
```

Não especificamos qual o método que será mapeado nesta action, isto porque quando não é informado qual método será acessado, o default para o struts é o método `execute()`, então quando esta action for executada o método `execute()` será executado por default.

Temos então apenas o nome `insereContato` que será utilizado para referenciar esta action na página `jsp`, o parametro `class` informamos onde esta a action com o caminho completo.

Esta action também tem uma propriedade `servico` que disponibilizara meios para acessar nosso banco para inserir este novo registro, também usaremos o Spring para disponibilizar este serviço para nossa action, mas adiante veremos como funciona este `ServicoContatoImpl`.

Quanto ao mapeamento desta action, pode ser visto abaixo que é semelhante ao de nossa action de login, so que este direcionara o usuário para a página `contato.jsp` após ter executado a action.

No tipo de retorno desta action, esta especificado o tipo `input` que como vimos é para possíveis erros de validação, e um outro result sem identificação que direciona para a página `contato.jsp`, como estamos utilizando o tipo de retorno fornecido pelo Struts 2, o `Action.SUCCESS`, e este retorna a string "success" que é definida como padrão pelo mesmo, então ao criarmos um result sem nome este será de acordo com a string "success".

Para listarmos todos os contatos do usuário utilizaremos a action `ListaContatosAction` como na listagem 3, para obtermos a lista do banco de dados. também será disponibilizado acesso ao banco pelo `ContatoImpl`.

```
package br.jm.actions;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import br.jm.entidade.Contato;
import br.jm.servico.ServicoContatoImpl;

import com.opensymphony.xwork2.Action;
import com.opensymphony.xwork2.ActionSupport;

public class ListaContatosAction extends ActionSupport{

    private ServicoContatoImpl servico;
    private List<Contato> contatos;

    public ListaContatosAction() {

    }

    public ListaContatosAction(ServicoContatoImpl servico) {
        servico = servico;
    }

    public String execute() throws Exception {
        contatos = servico.listarContatos();

        return Action.SUCCESS;
    }
}
```

Listagem 2 – ListaContatosAction.java

Temos uma propriedade `contatos` que é a nossa lista, essa é preenchida quando chamado o método `execute` de nossa action. Para utilizarmos nossa action devemos configurar no `struts.xml` conforme abaixo.

```

<action name="listaContatos"
        class="br.jm.actions.ListaContatosAction">
    <interceptor-ref name="exception" />
    <interceptor-ref name="basicStack" />
    <exception-mapping result="erroServicoContato"
        exception="java.lang.Exception" />

    <result name="erroServicoContato">
        /erroServicoContato.jsp
    </result>
    <result>/listarContatos.jsp</result>
</action>

```

Nossa action se chama *listaContatos*, e possui um intercetor para qualquer excetion que possa ocorrer e ira direcionar para a página *erroServicoContato.jsp*, e seu result default onde também não especificamos um nome para ele é a página *listarContato.jsp*.

Para utilizarmos este interceptor para exceptions apenas configuramos o inteceptor e mapeamos para um result apontando o tipo de exception que este devera filtrar, que neste caso é *java.lang.Exception* que consequentemente ira ser qualquer exception que ocorra já que todas herdam dela, e configuramos o result como a página *erroLogin.jsp*.

```

<interceptor-ref name="exception" />
<interceptor-ref name="basicStack" />
<exception-mapping result="erroServicoContato"
    exception="java.lang.Exception" />

<result name="erroServicoContato">
    /erroServicoContato.jsp
</result>

```

Aqui a única coisa que alteramos foi o parametro *result* onde criamos o nome *erroExcetion* e apontamos este nome para a página *erroServicoContato.jsp*.

Criamos uma página jsp conforme a listagem 4 sem nenhum cabeçalho, apenas escrevemos a navegação da propriedade *contatos*.

```

<%@taglib uri="/struts-tags" prefix="s"%>

<table border="1" bordercolor="blue" cellpadding="2" cellspacing="2">

    <tr>
        <td><b>Contatos Registrados</b></td>
    </tr>
    <tr>
        <td style="width: 160px; text-align: center;">Nome</td>
        <td style="width: 160px; text-align: center;">Sobrenome</td>
        <td style="width: 160px; text-align: center;">Data De
Nascimento</td>
        <td style="width: 160px; text-align: center;">DDD</td>
        <td style="width: 160px; text-align: center;">Telefone</td>
        <td style="width: 160px; text-align: center;">Remover</td>
    </tr>

    <s:iterator value="contatos">
        <tr>
            <td><s:property value="pessoa.nome"/></td>
            <td><s:property value="pessoa.sobreNome"/></td>
            <td><s:date name="pessoa.dataDeNascimento"
format="dd/MM/yyyy"/></td>

```

```

        <td><s:property value="telefone.ddd"/></td>
        <td><s:property value="telefone.numero"/></td>
        <td>

        <s:url id="link1" action="/removeContato!
removeUmContato.action">
            <s:param name="id"><s:property value="pessoa.id"/>
        </s:param>
        </s:url>

        <s:a id="linkRemove"
            href="%{link1}"
            theme="ajax"
            notifyTopics="listaContatoTopic"
            loadingText="Removendo contato.."
            errorText="Ocorreu um erro durante o
processamento.."
            showLoadingText="true"
            showErrorTransportText="true">Excluir</s:a></td>
    </tr>
</s:iterator>
</table>

```

Listagem 4 – listaContatos.jsp

Iremos ver no próximo tópico as páginas jsp que criamos para nossa aplicação exemplo.

Agora vamos ver a action que ira remover os registros do usuário, na action *RemoveContatoAction* temos dois métodos, um para remover apenas um registro e o outro para remover todos os registros do usuário, foi dividido em duas declarações no struts.xml para demonstrar outras formas de trabalhar com o Struts 2.

```

<action name="removeContato!*" method="removeLista"
    class="br.jm.actions.RemoveContatoAction">
    <interceptor-ref name="exception" />
    <interceptor-ref name="basicStack" />
    <exception-mapping result="erroServicoContato"
        exception="java.lang.Exception" />

    <result name="erroServicoContato">
        /erroServicoContato.jsp
    </result>
    <result>/contato.jsp</result>
</action>

```

Repare que temos o mesmo nome para nossa action, vamos apenas especificar qual método será acessado, no caso acima informamos que o método *removeLista* pelo *parametro* *method* terá um interceptor para qualquer exception e seu resultado deve ser direcionado para a página *contato.jsp*.

Para nossa outro mapeamento para a action *RemoveContatoAction* informamos que o método *removeUmContato()* irá ser acessado e tem os mesmos retornos e tipo de interceptor do método *removeLista()*.

```

<action name="removeContato!*" method="removeUmContato"
    class="br.jm.actions.RemoveContatoAction">
    <interceptor-ref name="exception" />
    <interceptor-ref name="basicStack" />
    <exception-mapping result="erroServicoContato"
        exception="java.lang.Exception" />

    <result name="erroServicoContato">
        /erroServicoContato.jsp
    </result>

```

```

        </result>
        <result>/contato.jsp</result>
    </action>

```

Nossa action para remover como pode ser observado na listagem 5 é um pojo, não herda de nenhuma classe e segue o padrão `JavaBean`, mas mesmo assim o utilizamos como uma action do `Struts 2`, assim fica mais prático de se testar as actions, visto que não há a necessidade de um container para executa-las.

```

package br.jm.actions;

import com.opensymphony.xwork2.Action;
import br.jm.servico.ServicoContatoImpl;

public class RemoverContatoAction {

    private Long id;
    private ServicoContatoImpl servico;

    public RemoverContatoAction() {

    }

    public RemoverContatoAction(ServicoContatoImpl servico) {
        this.servico = servico;
    }

    public String removeUmContato() {
        servico.removerContato(id);

        return Action.SUCCESS;
    }

    public String removeLista() {
        servico.removerContatos();

        return Action.SUCCESS;
    }
}

```

Listagem 5 – `RemoverContatoAction.java`

Vocês podem estar pensando que seria melhor ter desenvolvido uma action `ContatoAction` e nela ter os métodos para inserir, remover, listar, etc. Mas o objetivo em separar em várias classes nossas actions foi de ilustrar as diferentes maneiras que podem ser utilizadas para configurar sua action no `Struts 2`, veja os links no final do artigo para mais informações.

Validação de Dados no Struts 2

Vamos começar a ver como validamos dados de entrada do usuário usando o `Struts 2`.

Para validarmos a entrada de dados dos atributos de uma action é necessário criar um arquivo xml que segue o padrão `nomeDaAction-validation.xml`, observe o exemplo na listagem 6.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE validators PUBLIC
    "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
    "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">

<validators>

    <field name="contato.pessoa.nome">
        <field-validator type="requiredstring">
            <message>Digite o nome</message>
        </field-validator>
    </field>

```

```

        </field-validator>
    </field>

    <field name="contato.pessoa.dataDeNascimento">
        <field-validator type="date">
            <message>Selecione uma data</message>
        </field-validator>
    </field>

    <field name="contato.pessoa.sobreNome">
        <field-validator type="requiredstring">
            <message>Digite o sobrenome</message>
        </field-validator>
    </field>

    <field name="contato.email">
        <field-validator type="email">
            <message>Digite um email válido</message>
        </field-validator>
    </field>

    <field name="contato.telefone.ddd">
        <field-validator type="requiredstring">
            <message>Digite um código DDD</message>
        </field-validator>
    </field>

    <field name="contato.telefone.numero">
        <field-validator type="requiredstring">
            <message>Digite o número do telefone</message>
        </field-validator>
    </field>

</validators>

```

Listagem 6 – InserirContatoAction-validation.xml

Este arquivo, o *InserirContatoAction-validation.xml* deve assim como qualquer outro arquivo xml de validação ficar no mesmo pacote que a action, no nosso caso estará dentro do pacote **br.jm.actions**.

A validação é configurada entre o elemento raiz `<validators/>` dentro informamos os **<field/>** (campos), que deverão ser validados e como fazer essa validação.

A tag **<field/>** tem como parâmetro o nome do atributo que deve ser validado, no nosso caso como esta dentro de um objeto contato basta escrevermos os atributos com o nome exato da classe que pertence no parâmetro *name*, aqui também são utilizados os métodos getters e setters da propriedade, então se não tiverem não irá funcionar.

Também configuramos o tipo de validação desta propriedade utilizando o parâmetro **<field-validator/>** que recebe o tipo de validação que deve ser feita, no caso do *nome*, *sobreNome*, *ddd* e *numero* são utilizados o tipo **requiredstring** que significa que deve ser informado uma string, o usuário tem que digitar algum valor.

Como já temos alguns tipos comuns fornecidos pelo struts que é o caso do tipo **date**, **email**, vocês podem ver que nossa validação do atributo *email* é do tipo **email**, logo deverá ser informado uma string no formato de um endereço de email válido.

No atributo *dataDeNascimento* é o tipo **date** que já definido pelo Struts válida a entrada de uma data, existem outros tipos interessantes, por exemplo uma validação de um atributo do tipo inteiro (Integer).

```

<field name="numeroExemplo">

```

```

    <field-validator type="int">
      <param name="max">5</param>
      <message>Digite um número entre 2 e 5 válido</message>
      <param name="min">2</param>
    </field-validator>
  </field>

```

Define que o *numeroExemplo* deve ser um número entre 5 e 2, muito simples validação no Struts 2, não!?!

No caso de nossa tela de login também temos algumas validações como na listagem 7.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE validators PUBLIC
    "-//OpenSymphony Group//XWork Validator 1.0.2//EN"
    "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">

<validators>

  <field name="login">
    <field-validator type="requiredstring">
      <message>É necessário digitar o login</message>
    </field-validator>
  </field>

  <field name="senha">
    <field-validator type="requiredstring">
      <message>É necessário digitar a senha</message>
    </field-validator>
  </field>

</validators>

```

Listagem 7 – LoginAction-validation.java

Tanto o login quanto a senha devem ser informados, caso contrário irá ser exibido os erros na tela do usuário, conforme a figura 1.

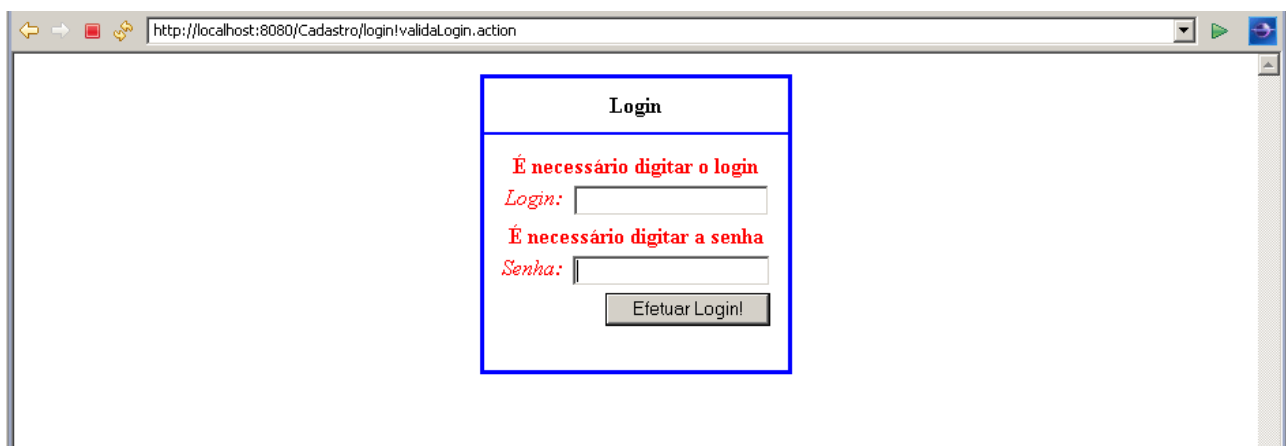


Figura 1 – Mensagem de erros de validação do login

Observem agora a tela de cadastro de um novo contato, que também esta validando os dados de entrada do formulário.

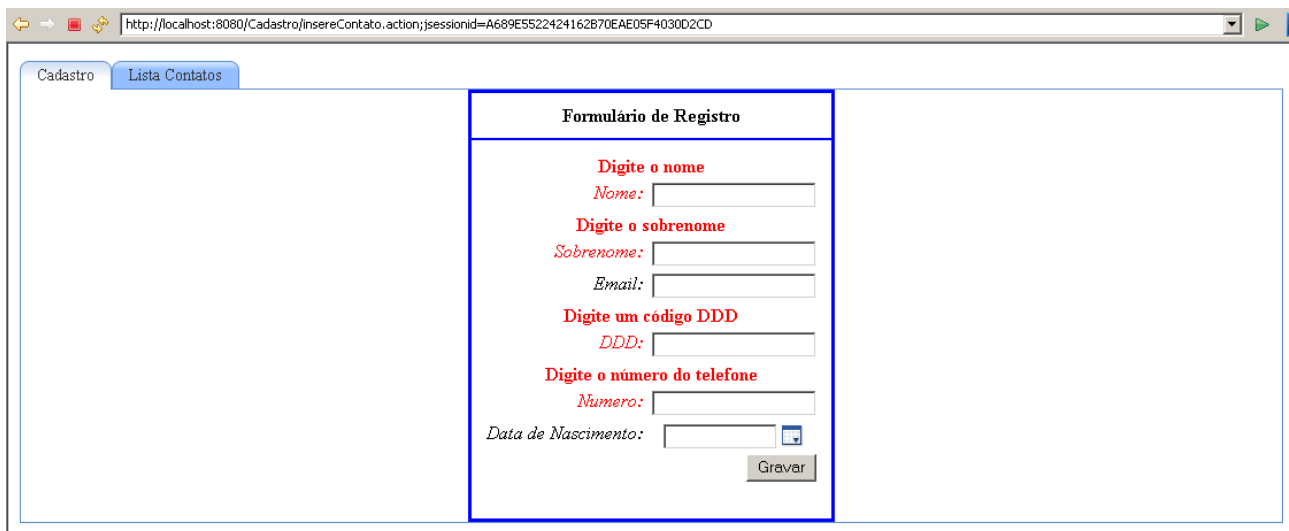


Figura 2 - Mensagem de erros de validação do formulário de cadastro

Sempre que houver algum erro na validação será exibido as mensagens como demonstradas acima, não é mais necessário informar aonde será exibido o erro como no Struts 1x que tínhamos que utilizar uma tag, mas isso também pode ser feito nessa nova versão, mas não utilizaremos neste artigo.

Note que nossa validação "pisca" a tela, para isso não acontecer basta configurarmos o tema de nosso form na página jsp como AJAX e configurar o servlet do dwr que é utilizada para validação pelo Struts 2 no web.xml como a listagem 7 mostra.

Struts 2 + AJAX

Para entender como funciona a integração do AJAX no Struts 2, primeiramente precisamos entender das tags do struts, na versão antiga as tags estavam divididas em três taglibs, no Struts 2, temos apenas que importar.

```
<%@taglib prefix="s" uri="/struts-tags"%>
```

Importando essa taglib temos acesso a todos os tipos de tags do Struts 2. Usamos o 's' como prefixo por convenção, é aconselhável mantê-lo.

Como visto na listagem 3, criamos uma tabela onde colocamos o cabeçalho da nossa listagem com alguns estilos, e na outra linha já utilizamos a tag **<s:iterator/>** do Struts 2 para navegarmos na nossa lista.

Passamos para o parametro **value** o nome da propriedade que é uma lista e apenas acessamos o valor de cada atributo utilizando a tag **<s:property/>** do struts passando o nome do atributo para o parametro **value** desta tag.

Assim já criamos nosso laço da lista de contatos. Uma tag interessante do Struts 2 é a tag **<s:url/>** onde criamos um link para uma action, passando para o parametro **action** o caminho para nossa action que nesse caso é [/removerContato!removeUmContato.action](#) e também passamos um parametro utilizando a tag **<s:param/>** que mapeamos o atributo **id** do objeto pessoa configurando o parametro **value** para **pessoa.id**.

Pronto, criamos nosso link para a action que ira remover este nosso registro passando como parametro o identificador do pessoa, simples não?

A tag **<s:a/>** também é outra tag importante do Struts 2, com ela criamos um link para nossa url utilizando ajax, configurando sua mensagem de erro, e mensagem de espera para nosso link como veremos agora explicando também a integração do AJAX com o Struts.

Antes de mais nada, se quiser utilizar ajax em suas páginas utilizando o Struts 2 deve-se utilizar a tag `<s:head theme="ajax"/>`, que informa ao struts para que a página possa utilizar o tema ajax, o Struts 2 utiliza o conceito de temas para renderizar uma página, existem outros tipos, mas não abordaremos neste artigo, sugiro os links no final do artigo para que possam obter mais informações.

É interessante não se esquecer de usar a tag `<s:head/>` configurando o tema para ajax, pois, pode ocorrer erros se isso não for feito.

No nosso exemplo iremos utilizar um `tabbedPanel` que é utilizado para criar abas na nossa página utilizando ajax.

Isto é apenas uma das várias funcionalidades para o desenvolvimento da camada de visão que o Struts 2 nos fornece a partir de suas tags. Procurem mais informações nos links que vocês puderam ver que este o Struts 2 realmente facilita a vida do desenvolvedor web.

Para criarmos nosso exemplo basta configurar a tag `<s:tabbedPanel/>` com o parametro `id` para informar que este panel pode ser identificado pelo nome `panell`, o `id` é utilizado pela ongl* para que um div remoto, por exemplo, consiga identificar este `tabbedPanel`, no caso da página `listaContatos.jsp` nossa url é identificada pelo atributo `href` da tag `<s:a/>` utilizando o id que configuramos.

Como pode ser visto no exemplo abaixo, com atributo `href` criamos referencias para nossas actions ou outras tags.

```
<s:url id="link1" action="/removerContato!removeUmContato.action">
  <s:param name="id"><s:property value="pessoa.id"/>
</s:param>
</s:url>

<s:a id="linkRemove"
  href="%{link1}"
  theme="ajax"
  notifyTopics="listaContatoTopic"
  loadingText="Removendo contato.."
  errorText="Ocorreu um erro durante o
              processamento.."
  showLoadingText="true"
  showErrorTransportText="true">Excluir</s:a>
```

Esta propriedade `id` é comum para todas as outras tags **assim href**.

A propriedade **theme** configurada para ajax informa que esta tag envia e recebe chamadas remotas, ou seja, estara utilizando ajax.

```
<s:tabbedPanel id="panell" theme="ajax" >
```

O atributo **theme** também é comum para todas as tags, logo nosso `tabbedPanel` esta utilizando ajax.

Para criarmos as abas, criamos divs remotos dentro do `tabbedPanel`, que terá o tema configurado para ajax, e na propriedade **label** informamos o nome que será exibido na tela, não confunda com o nome que é dado na propriedade `id`, na propriedade `id` configuramos um nome para que o mesmo possa ser acessado dentro da página por outros divs, urls, etc.

```
<s:div id="listaContato"
  href="listaContatos.action"
  listenTopics="listaContatoTopic"
  theme="ajax"
  showLoadingText="true"
  showErrorTransportText="true"
  errorText="Um erro ocorreu, tente novamente..."
```

```
loadingText="Obtendo lista contatos...">
</s:div>
```

Um `<s:div/>` do struts possui todas as funções `<div/>` do html, e agrega a funcionalidade ajax, que o torna um div remoto, isso sendo possível porque utilizamos de propriedades como **`href="/inserirContato.action"`**, que cria um link com essa action, assim o resultado dessa action ira para este div.

No div `listaContato` esse recebera o conteúdo da página `listaContato.jsp` que criamos anteriormente.

No atributo **listenTopics** informamos qual tópico este div estara ouvindo. Como vemos no exemplo abaixo usamos a tag **`<s:submit/>`** para gravar um novo contato, e este submit esta configurado para "notificar o tópico" `listaContatoTopic` que o nosso div remoto listaContato esta "ouvindo" e ainda utilizar ajax para realizar a tarefa.

Podem ser configurados vários tópicos para ouvir e notificar ao mesmo tempo.

```
<s:form action="inserirContato.action" validate="true">
  <s:textfield name="contato.pessoa.nome" label="Nome"></s:textfield>
  <s:textfield name="contato.pessoa.sobreNome"
label="Sobrenome"></s:textfield>
  <s:textfield name="contato.email" label="Email"></s:textfield>
  <s:textfield name="contato.telefone.ddd" label="DDD"></s:textfield>
  <s:textfield name="contato.telefone.numero"
label="Numero"></s:textfield>
  <s:datetimepicker name="contato.pessoa.dataDeNascimento"
label="Data de Nascimento" theme="ajax" displayFormat="dd/MM/yyyy" /
>
  <s:submit value="Gravar" notifyTopics="listaContatoTopic"/>
</s:form>
```

Agora a tag **`<s:form/>`**!

É com esta tag que criamos nosso formulário, e configuraremos para utilizar ajax e validação dos campos. Não a necessidade de informar onde será exibido os erros, somente atribuir **true** a propriedade **validate** para que o form seja validado, e adivinha!?! Basta utilizar tema como ajax para que seja utilizado ajax para fazer isso, mas o Struts 2 utiliza o DWR para validações e DOJO para as outras funcionalidades ajax que estavamos utilizando ate agora.

Então se quiser validação dos campos com ajax deve configurar o servlet do DWR no arquivo web.xml, demonstrado na listagem 8 e informar que o tema de nosso form é ajax.

Utilizamos a tag **`<s:textfield/>`** para exibir o nome que ira aparecer na página jsp com o atributo **label** e configuramos a propriedade que ira receber com o atributo **name**, com a tag **`<s:datetimepicker/>`** criamos um calendário utilizando o tema ajax, configuramos também um **label** e um **name** exatamente como a tag **`<s:textfield/>`** e usamos o atributo **displayFormat** para configurar como será exibido a data, usamos o formato `dd/MM/yyyy` em nosso exemplo.

Somente isso é necessário para criar um calendario usando ajax e o mais importante, nenhuma linha de javascript.

```
<servlet>
  <servlet-name>dwr</servlet-name>
  <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>>true</param-value>
  </init-param>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>dwr</servlet-name>
  <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

Listagem 8 – Declaração do servlet do DWR no arquivo web.xml

Com a tag **<s:submit/>** tendo o parametro **notifyTopics** configurado para *listaContatoTopic* toda vez que um submit for realizado sera notificado quem estiver ouvindo este tópico logo estas serão atualizadas.

Pronto!

Seu formulário esta sendo validado e utilizando ajax, na figura 3 pode se ver a página de login que foi criada.

Como nosso div *listaContato* esta ouvindo o topico *listaContatoTopic* pelo atributo **listenTopics** este div sera atualizado toda vez que for notificado.

Aqui criamos então nossa camada de visão utilizando os tabbedPanel do struts, como cada aba é um **<s:div/>** poderíamos criar quantos quiséssemos e apenas configura-los para usar o tema ajax e pronto!

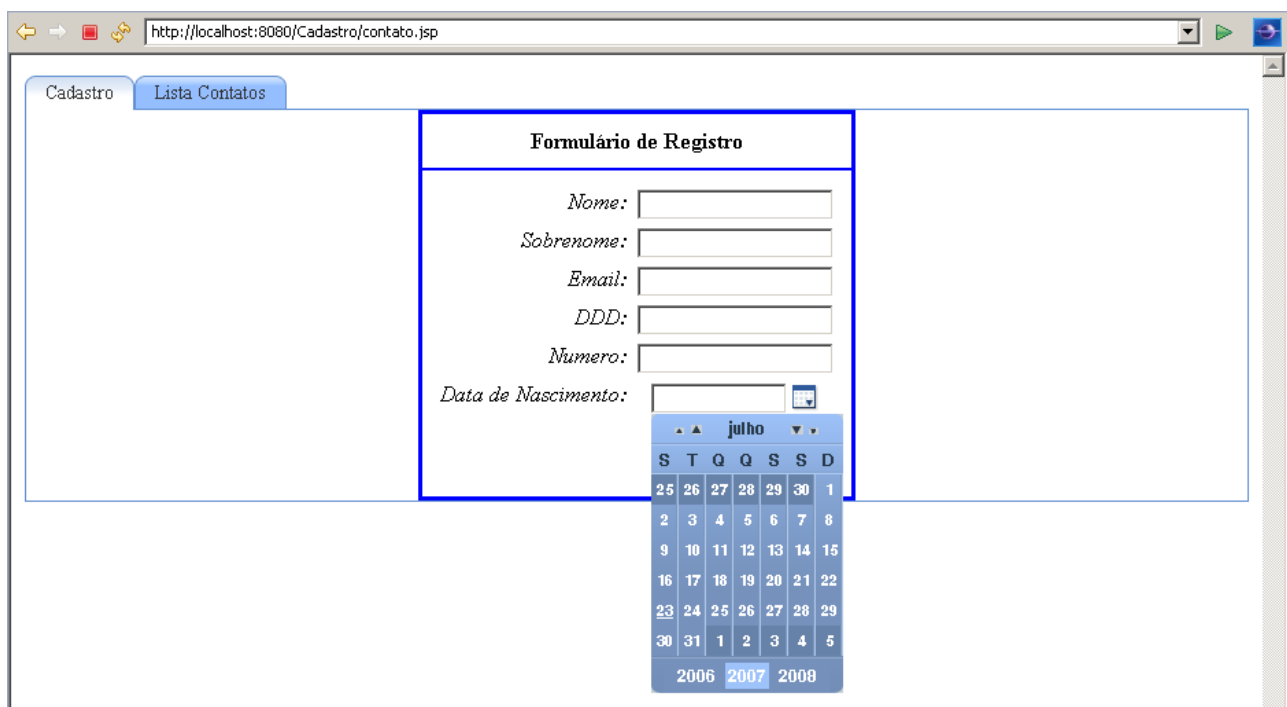


Figura 3 - Tela de Cadastro

| Contatos Registrados | | | | | |
|----------------------|-----------|--------------------|-----|----------|-------------------------|
| Nome | Sobrenome | Data De Nascimento | DDD | Telefone | Remover |
| Joao | Paulo | 16/01/1983 | 62 | 81132189 | Excluir |
| Teeste | Manoel | 14/07/2007 | 62 | 6222222 | Excluir |
| Teeste | Manoel | 14/07/2007 | 62 | 6222222 | Excluir |
| Marcos | Paulo | 13/07/1984 | 62 | 41755569 | Excluir |
| Marcos | Paulo | 13/07/1984 | 62 | 41755569 | Excluir |
| Marcos | Paulo | 13/07/1984 | 62 | 41755569 | Excluir |
| Paulo | Manoel | 25/07/1984 | 62 | 81145569 | Excluir |
| Carlos | Silva | 25/11/1984 | 62 | 98995656 | Excluir |

Figura 4 – Listagem de contatos

Vocês podem ver que nosso cadastro esta em uma aba de nosso tabbedPanel e a listagem que possui o link para remover o registro esta em outra aba, vejam a aba da listagem de contatos na figura 4.

```
<%@ 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>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>JavaMagazine</title>
</head>

<s:head theme="ajax" />

<body background="lightblue">

<s:tabbedPanel id="panel1" theme="ajax" >
    <s:div label="Cadastro" theme="ajax">
        <center>
            <div>
                <table border="2" cellspacing="0" cellpadding="10"
bordercolor="blue"
                    background="buttonhighlight" align="center">
                    <tr>
                        <td><b>Formulário de Registro</b></td>
                    </tr>
                    <tr>
                        <td>
                            <s:form action="insereContato.action" validate="true">
                                <s:textfield name="contato.pessoa.nome"
label="Nome"></s:textfield>
                                <s:textfield name="contato.pessoa.sobreNome"
label="Sobrenome"></s:textfield>
                                <s:textfield name="contato.email" label="Email"></
s:textfield>
                                <s:textfield name="contato.telefone.ddd"

```

```

label="DDD"></s:textfield>
        <s:textfield name="contato.telefone.numero"
label="Numero"></s:textfield>
        <s:datetimepicker
name="contato.pessoa.dataDeNascimento"
        label="Data de Nascimento" theme="ajax"
displayFormat="dd/MM/yyyy" />
        <s:submit value="Gravar"
notifyTopics="listaContatoTopic" ></s:submit>
    </s:form></td>
    </tr>
</table>
</div>
</center>
</s:div>

<s:div label="Lista Contatos" theme="ajax">
    <center>
        <s:div id="listaContato"
            href="listaContatos.action"
            listenTopics="listaContatoTopic"
            theme="ajax"
            showLoadingText="true"
            showErrorTransportText="true"
            errorText="Um erro ocorreu, tente novamente..."
            loadingText="Obtendo lista contatos..."
        </s:div>

        <s:submit href="removerContato!removeLista.action"
            value="Limpar Lista"
            notifyTopics="listaContatoTopic"
            theme="ajax"
            align="center"
            showLoadingText="true"
            showErrorTransportText="true"
            errorText="Um erro ocorreu, tente novamente..."
            loadingText="Limpando lista contatos..."
        />

    </center>
</s:div>

</s:tabbedPanel>

</body>
</html>

```

Listagem 9 – contato.jsp

A página *contato.jsp* completa esta na listagem 9.

Parte II - Persistencia com JPA

Estamos utilizando a implementação do Hibernate da JPA, que será nosso Persistence provider, que executará o acesso ao banco de dados, como estaremos utilizando a JPA, então podemos a qualquer hora mudar sua implementação para utilizar o TopLink da Oracle por exemplo, desde que utilizemos suas anotações definidas no *pacote javax.persistence*.

Não se preocupem em criar as tabelas, a partir da primeira execução da aplicação exemplo, a JPA irá criar todas as tabelas automaticamente.

Uma das vantagens quando usamos as anotações da JPA é que por se tratar de um padrão definido para frameworks de mapeamento objeto/relacional podemos mudar sua implementação sem alterar nossas entidades.

Aqui na JPA temos o conceito de *Entidades*, uma entidade é um simples POJO que possui um construtor default e os métodos getters e setters dos atributos que não podem ser *final* e representa uma tabela no banco de dados.

A JPA utilizamos um `EntityManagerFactory` para podermos criarmos nossa `EntityManager` que nos dara acesso a funções de `insert`,`select`,`update`,etc. É semelhante ao `SessionFactory` do hibernate.

A entidade representa a tabela e os atributos dessa entidade representam as colunas da tabela, para isso temos a anotação **@Entity** que informa a JPA que este pojo é uma entidade.

Quando utilizando frameworks de persistencia, nos temos os seguintes tipos de relacionamentos possíveis entre as entidades. *OneToOne*, *ManyToOne*, *OneToMany*, *ManyToMany*

Estes relacionamentos possuem dois parametros importantes o **cascade** e o **fetch** que serão explicados mais adiante.

Como temos que importar as libs do hibernate para que possamos utilizar a JPA devemos nos atentar para que todas as anotações que importemos para nossas classes sejam do pacote `javax.persistence`, assim garantimos que é uma anotação da especificação EJB 3.0.

Para utilizar a JPA crie dentro do seu diretório **src** a pasta **META-INF**, e dentro desta pasta crie o arquivo `persitence.xml`, podem ver um exemplo na listagem 13, mas o nosso `persistence.xml` não terá nada já que estaremos utilizando o Spring.

Vamos também utilizar o arquivo **log4j.properties** que deve ser copiado para o diretório **src** da aplicação, utilizamos este arquivo para podermos ativar a saída da console de informações ou debug do hibernate, esse arquivo tem mais utilidades mas para nos neste artigo utilizaremos apenas para poder exibir as informações do hibernate.

Este arquivo pode ser encontrado dentro do diretório **hibernate-3.2.3.ga\hibernate-3.2\etc**.

Criando nossas Entidades

Começaremos analisando a classe `Usuario`, que utilizamos na nossa action de login. Conforme a listagem 10.

```
package br.jm.entidade;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="CT_USUARIOWEB")
public class Usuario {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;
    @Column(length=10)
    private String login;
```

```

@Column(length=10)
private String senha;

public Usuario() {

}

public Usuario(String login, String senha) {
    this.login = login;
    this.senha = senha;
}

//métodos getters e setters
}

```

Listagem 10 – Usuario.java

Aqui também não podemos esquecer de um construtor default e os métodos getters and setters.

Antes de mais nada deve-se utilizar a anotação *@Entity*, somente isto já basta para persistirmos um objeto desta classe, a JPA por default associa o nome da classe a tabela e o nome dos atributos como nome das colunas, aqui estamos utilizando a anotação **@Table** passando para o parametro **name** o nome da tabela, que neste caso quer dizer que a classe Usuario representa a tabela *CT_USUARIOWEB*.

Todo objeto por regra tem que possuir um atributo que possa ser usado para identifica-lo no banco de dados, com a anotação **@Id**, também utilizamos a anotação **@GeneratedValue** para informar que deve ser gerado um valor para este atributo, e o parametro **strategy** define a estrategia com a qual será utilizada para gerar o valor deste campo identificador. Neste caso utilizamos a enumeração da JPA.

```
@GeneratedValue(strategy=GenerationType.IDENTITY)
```

Similar ao **identity** do banco de dados. Existem outros tipos de estrategias que não serão abordadas neste artigo, mas no final do artigo existem links onde você podera saber mais a respeito.

A propriedade **id** é um Long e sera utilizado como identificador deste objeto, nas propriedades **login** e **senha** apenas utilizamos a anotação **@Column** para definir o tamanho da coluna, o default é para campos string criar o tamanho máximo, também existe a possibilidade de configurar o nome que a aquela propriedade representa utilizando essa mesma anotação.

Na listagem 11 temos a classe Telefone que representa o telefone do contato que sera adicionado.

```

package br.jm.entidade;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="CT_TELEFONECONTATO")
public class Telefone {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;
}

```

```

private String ddd;
@Column(length=20)
private String numero;

public Telefone() {

}
//métodos getters e setters

```

Listagem 11 – Telefone.java

Também definimos as mesmas anotações para esta classe e relacionamos esta classe com a *tabela* **CT_TELEFONECONTATO**, e definimos o tamanho do campo dos atributos, temos um exemplo de relacionamento na listagem 12.

```

package br.jm.entidades;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinColumns;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name="CT_CONTATO")
public class Contato {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;
    @OneToOne(cascade=CascadeType.ALL, fetch=FetchType.LAZY)
    @JoinColumn(name="fk_pessoa")
    private Pessoa pessoa;
    @Column(length=100)
    private String email;
    @OneToOne(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
    @JoinColumn(name="fk_telefone")
    private Telefone telefone;

    public Contato() {

}
//métodos getters e setters

```

Listagem 12 – Contato.java

Esta classe, além de ter as anotações já conhecidas, possui um relacionamento do tipo um para um com a classe Pessoa e a Classe Telefone, na JPA este tipo de relacionamento deve ser configurado utilizando a anotação **@OneToOne**, são disponíveis os tipos de relacionamento **@OneToMany**, **@ManyToMany**, **@ManyToOne** e **@OneToOne** mas neste artigo so utilizaremos relacionamento um-para-um, mas os elementos principais e comuns a esses relacionamentos são explicados abaixo.

Na *anotação* **@OneToOne** igual as outras anotações de relacionamentos possui o *parametro* **cascade** onde definimos que tipo de cascade deve ser realizado caso haja alguma operação na entidade, utilizamos a enumeração **CascadeType** para escolher entre: **MERGE**, **PERSIST**, **REMOVE**, **REFRESH** e **ALL**.

Por exemplo, no nosso caso se for removido o contato, também será removido a pessoa, pois colocamos o tipo como **ALL**, logo qualquer operação que for realizada na entidade *contato* será realizada também nos seus relacionamentos.

Em sistemas de produção não é aconselhável a utilização do parâmetro **cascade** configurado para todos, como o nosso exemplo, pois é interessante ter controle sobre suas entidades.

No parâmetro **fetch** configuramos o modo que será obtido o objeto que faz o relacionamento do banco de dados.

```
Contato = entityManager.find(Contato.class, id);
```

Configurando o parâmetro **fetch** para **LAZY**, os atributos *email* da classe *contato* é recuperado do banco de dados após a execução deste método *find()*, mas não o atributo *pessoa*, será simplesmente criada uma referência e não será armazenado o conteúdo do banco de dados na memória dentro do objeto *pessoa*, somente quando for realizada a chamada para o seu método *get*.

```
Pessoa pessoa = contato.getPessoa();
```

Depois dessa chamada sim, será então populado o objeto *pessoa* com as informações do banco de dados. Isso é muito útil quando necessário ter objetos de tabelas com grande quantidade de dados e não for necessário utilizar esses dados a toda hora.

Já na propriedade *telefone*, configuramos para **EAGER** o parâmetro **fetch**, logo quando for realizado um *load/find* será obtido na hora o valor do banco de dados e populado na memória o objeto *telefone*.

Isso pode ser melhor visualizado em modo debug, coloque um breakpoint no método *acharContato* da classe *ContatoDao* que você poderá ver que não é carregado o objeto *pessoa*, mas que o objeto *telefone* tem todos seus atributos devidamente carregados em memória.

Agora a classe *Pessoa* que está na listagem 14.

```
package br.jm.entidade;

import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name="CT_PESSOA")
public class Pessoa {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    @Column(length=150)
    private String nome;
    @Column(length=150)
    private String sobreNome;
    @Temporal(TemporalType.DATE)
    private Date dataDeNascimento;
```

```

    public Pessoa() {

    }

    //métodos getters e setters

}

```

Listagem 14 – Pessoa.java

Nossa classe pessoa esta mapeada para a tabela **CT_PESSOA** e diferente de nossas outras entidades possui um atributo do tipo **date** que é a data de nascimento do contato.

Neste caso utilizamos a anotação **@Temporal** para indicar que este atributo vai ser uma coluna do tipo **DATE**, a enumeração **TemporalType** possui vários tipos como hora, data, etc. Aqui usaremos o tipo **DATE**, por se tratar de uma data de aniversário.

Pronto nossas entidades estão prontas para utilizarmos o banco de dados em nossa aplicação de exemplo.

Poderíamos criar um dao genérico que atendesse as nossas necessidades, mas neste artigo criaremos dois daos, um para o contato e outro para o login.

Abaixo na listagem 15 o LoginDao.

```

package br.jm.persistencia;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.NoResultException;
import javax.persistence.Persistence;
import javax.persistence.Query;

import br.jm.entidade.Usuario;

public class LoginDAO {

    private EntityManager entityManager;
    private EntityManagerFactory entityManagerFactory;

    public LoginDAO() {

    }

    public LoginDAO(EntityManagerFactory entityManagerFactory) {
        this.entityManagerFactory = entityManagerFactory;
        this.entityManager =
this.entityManagerFactory.createEntityManager();
    }

    public Boolean validaLogin(String login, String senha) {
        Query query = this.entityManager.createQuery("SELECT u FROM Usuario
u WHERE u.login = :login AND u.senha = :senha");
        query.setParameter("login", login);
        query.setParameter("senha", senha);

        Usuario user = null;
        try {
            user = (Usuario) query.getSingleResult();
        } catch (NoResultException e) {
            e.printStackTrace();
        }
    }
}

```

```

        return (user != null);
    }
    //métodos getters e setters

```

Listagem 15 – LoginDAO.java

Nossa EntityManagerFactory será injetada no nosso dao pelo Spring, mais adiante veremos como isso é feito.

Criamos nossa EntityManager a partir do método *createEntityManager()* da EntityManagerFactory, e somente com isso já podemos obter acesso ao banco.

No método *validaLogin* é recebido um login e uma senha, e aqui utilizamos um objeto do tipo **Query** para criarmos nossa JPAQL, que é uma linguagem semelhante a **HQL** do Hibernate onde temos como objetivo realizar consultas SQL pensando em objetos .

Neste exemplo realizamos a seguinte JPAQL.

```

Query query = this.entityManager.createQuery("SELECT u FROM Usuario u
WHERE u.login = :login AND u.senha = :senha");

```

Quer dizer que vamos realizar um select do objeto usuário onde o login e a senha forem como as passadas como parametro. É usado a sintaxe *:nomeParametro* para informar que este sera substituido por uma string utilizando o método *setParameter()* da classe Query.

```

query.setParameter("login", login);
query.setParameter("senha", senha);

```

Assim criamos nossa consulta para validar o usuário.

Caso não seja encontrado nenhum registro será lançada a exceção **NoResultException** que no nosso caso retornara false, lembrando que isso é apenas um exemplo um login para um sistema é mais complexo e possui direcionamentos variados.

Ate agora bem simples de se realizar uma consulta, mas existem maneiras mais simples de se fazer isso, vamos mostrar abaixo agora outras operações.

```

package br.jm.persistencia;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Query;

import br.jm.entidade.Contato;

public class ContatoDAO {

    private EntityManagerFactory entityManagerFactory;
    private EntityManager entityManager;
    private EntityTransaction transaction;

    public ContatoDAO() {
    }

    public ContatoDAO(EntityManagerFactory entityManagerFactory) {
        this.entityManagerFactory = entityManagerFactory;
        this.entityManager =

```

```

this.entityManagerFactory.createEntityManager();
    this.transaction = this.entityManager.getTransaction();
}

public void adiciona(Contato contato) {
    if(this.transaction.isActive()) {
        this.entityManager.persist(contato);
        this.transaction.commit();
    }else {
        this.transaction.begin();
        this.entityManager.persist(contato);
        this.transaction.commit();
    }
}

public List<Contato> obtenListaContato() {
    Query query = this.entityManager.createQuery("SELECT c FROM Contato
c");

    return query.getResultList();
}

public void removerContato(Long id) {
    Contato contato = acharContato(id);
    if(this.transaction.isActive()) {
        this.entityManager.remove(contato);
        this.transaction.commit();
    }else {
        this.transaction.begin();
        this.entityManager.remove(contato);
        this.transaction.commit();
    }
}

public Contato acharContato(Long id) {
    Contato contato = null;
    contato = this.entityManager.find(Contato.class, id);

    return contato;
}

public void removeContatos() {
    Query query = null;

    if(this.transaction.isActive()) {
        query = this.entityManager.createQuery("DELETE Contato");
        query.executeUpdate();
        this.transaction.commit();
    }else {
        this.transaction.begin();
        query = this.entityManager.createQuery("DELETE Contato");
        query.executeUpdate();
        this.transaction.commit();
    }
}
//métodos getters e setters
Listagem 15 - ContatoDAO.java

```

Neste nosso ContatoDAO que pode ser visto na listagem 15 o EntityManagerFactory também será injetado pelo Spring, veremos então como funciona suas funcionalidades utilizadas aqui.

```

public void adiciona(Contato contato) {

```

```

        if (this.transaction.isActive()) {
            entityManager.persist(contato);
            transaction.commit();
        } else {
            transaction.begin();
            entityManager.persist(contato);
            transaction.commit();
        }
    }
}

```

O método *adiciona()* recebe um objeto do tipo contato e com um bloco **if..else** ele valida se existe uma transação ativa, para algumas operações como insert é necessário ter uma transação ativa para realizar o insert utilizando o *método persist()* da JPA, caso você não inicie uma transação e não realize o commit da mesma, nenhum erro será exibido, mas nenhuma alteração no banco de dados será realizada.

Fazemos uma verificação com método *isActive()* da classe **EntityTransaction** e se estiver ativa então apenas chamamos o método *persist()* do nosso **EntityManager** passando o objeto que deve ser inserido no banco de dados. Para realizar um update utiliza o método *merge()* da JPA que realiza o update, passa-se um objeto também para este método. Após a execução do método *persist()* é necessário realizar o commit da transação para que seja realmente realizado o insert no banco de dados.

Se nossa transação não estiver aberta ela então chamará o método *begin()* do objeto **EntityTransaction** para poder iniciar a inserção do objeto contato.

```

    public List<Contato> obtemListaContato() {
        Query query = this.entityManager.createQuery("SELECT c FROM Contato c");

        return query.getResultList();
    }

```

No método *obtemListaContato()* utilizamos novamente a JPAQL para junto com o objeto do tipo **Query** nos fornecer a lista de todos os registros no banco de dados do objeto Contato. Repare que com a JPAQL realmente pensamos em consultas de objetos e existem várias maneiras de se realizarem consultas complexas, no final do artigo veja os links a respeito.

O tipo de retorno é a saída do método *getResultList()*, também possui o método *getSingleResult()* que obtem apenas um resultado o contrário do que estamos utilizando.

```

    public Contato acharContato(Long id) {
        Contato contato = null;
        contato = this.entityManager.find(Contato.class, id);

        return contato;
    }

```

Neste método recebemos um *id* que é o identificador do objeto. Apenas retornamos o objeto contato recebido através do método **find** similar ao **load** do Hibernate, mas como aqui é utilizado genéricos do Java 5 então não há a necessidade de fazermos um cast para um objeto, diferente do **load** do hibernate que utiliza um Object, e caso não for encontrado o objeto então o retorno será null.

Os parametros para o método *find()* são apenas o tipo da classe, ou seja, seu *.class* e seu identificador. Isso já basta para você obter um objeto do banco de dados, lembrando que nesse momento o relacionamento do objeto pessoa no objeto contato não será carregado na memória pois seu tipo de **fetch** esta configurado para **LAZY**, então somente após realizado um get na propriedade pessoa será realizado um load nesse objeto.

```

public void removerContato(Long id) {
    Contato contato = acharContato(id);
    if (this.transaction.isActive()) {
        this.entityManager.remove(contato);
        this.transaction.commit();
    } else {
        this.transaction.begin();
        this.entityManager.remove(contato);
        this.transaction.commit();
    }
}

```

Para remover o contato receberemos o *id* desse contato e obtemos o objeto a partir de uma chamada para o método *acharContato()*, lembrem-se que criamos uma url para nossa action remover contato que recebia como paramtreo um id?

Então é este método que a action ira executar chamando o servico.

Aqui também precisamos de uma transação para realizar essa operação, fazemos a mesma verificação do método *adicionar()* apenas chamamos o método *remove()* do nosso **EntityManager** e esse objeto será removido, então realizamos um commit na operação para que essa seja concretizada, nosso último método deste dao é o:

```

public void removeContatos() {
    Query query = null;

    if (this.transaction.isActive()) {
        query = this.entityManager.createQuery("DELETE Contato");
        query.executeUpdate();
        this.transaction.commit();
    } else {
        this.transaction.begin();
        query = this.entityManager.createQuery("DELETE Contato");
        query.executeUpdate();
        this.transaction.commit();
    }
}

```

Este método remove todos os contatos do banco de dados, utilizando a JPAQL para fazer o delete.

Para isso criamos um objeto do tipo **Query** e a partir do **EntitManager** criamos a query:

```
DELETE Contato
```

Também realizamos tanto a validação para verificar se nossa transação esta ativa e realizamos um commit nessa operação.

Estes daos poderiam estar organizados em um dao genérico que atendesse as necessidades de nossas actions, mas fiz desta maneira para não me prender muito a explicação de genéricos do Java 5, talvez para um próximo artigo.

Nosso *LoginImpl* e *ContatoImpl* são apenas objetos que tem suas operações definidas em interfaces que servem para fornecer o acesso ao banco pelas nossas actions, e também para vermos a utilização do spring quando injetamos nossos daos em nossos serviços e nosso serviços em nossas actions.

Parte III - Integrando o Spring com Struts 2 + JPA

Bem, como visto ate agora, nossas actions irão utilizar as classes *LoginImpl* e *ContatoImpl* para poderem acessar o banco de dados.

Injeção de Dependências

Agora vamos ver como a dependencia de nossos objetos serão injetadas pelo Spring.

Nossos serviços de dados terão seus atributos que são referencias a classes daos instanciados pelo spring, logo nossas classes daos teram sua dependencia que no caso delas é o EntityManagerFactory que é o objeto que tem carregado todas a informações e meios de acesso ao banco de dados também instanciado pelo spring.

O Spring age como um container que fabrica seus objetos a partir de um arquivo de configuração, embora esta nova versão do spring já possua suporte a annotations este artigo não engloba este tópico, deixaremos para a próxima.

Existem algumas maneiras para realizar a injeção de dependencias, neste artigo iremos abordar os dois estilos que são:

Constructor Injection : é quando a injeção da dependencia é realizada pelo construtor da classe, utilizando uma referência do objeto a ser passado como abaixo:

```
<bean id="contatoDao" class="br.jm.persistencia.ContatoDAO">
  <constructor-arg ref="entityManagerFactory"></constructor-arg>
</bean>
```

Aqui nosso construtor da classe ContatoDAO receberá uma referencia ao bean *entityManagerFactory*.

Setter Injection : é quando o container, no nosso caso o Spring, utiliza do método set da propriedade para realizar a atribuição por valor da propriedade, como no exemplo abaixo:

```
<bean id="removeContato" class="br.jm.actions.RemoverContatoAction"
scope="prototype">
  <property name="servico" value="servicoContato"></property>
</bean>
```

No bean *removeContato* é atribuido por valor utilizando o método set do atributo *servico* o valor do bean *servicoContato*

É com a tag **<bean/>** que iremos configurar o spring, esta tag é inserida dentro do elemento raiz **<beans/>** configurado no arquivo **applicationContext.xml** que deve ser criado dentro do seu diretório nomeProjeto/WebContent/WEB-INF/

Vamos ver primeiramente a classe *LoginImpl* que implementa a interface *LoginIF* que esta na listagem 16.

```
package br.jm.servico;

public interface LoginIF {

    public Boolean valida(String login, String senha);

}
```

Listagem 16 – LoginIF.java

Esta interface possui somente um método que retorna um boolean e recebe duas strings, o login e a senha do usuário, agora vamos a ver a sua implementação de acordo com a listagem 17.

```
package br.jm.servico;

import br.jm.persistencia.LoginDAO;

public class LoginImpl implements LoginIF {
```

```

    private LoginDAO loginDAO;

    public LoginImpl() {
    }

    public LoginImpl(LoginDAO loginDAO) {
        this.loginDAO = loginDAO;
    }

    public Boolean valida(String login, String senha) {

        return this.loginDAO.validaLogin(login, senha);
    }

    //métodos getters e setters
}

```

Listagem 17 - LoginImpl

Como a classe *LoginImpl* possui um atributo *loginDAO* e este será injetado pelo Spring então temos que declarar um construtor default e um construtor recebendo um LoginDAO caso iremos utilizar uma injeção do tipo constructor injection que é quando se recebe uma referência do objeto no construtor.

Para que não ocorra erros utilizando o spring é necessário que o atributo *loginDAO* tenha seus métodos getters e setters. Nos iremos configurar o spring para que este instancie e injete qualquer dependencias desses objetos.

Agora vamos ver a classe *ContatoImpl* que implementa a interface *ContatoIF* como pode ser visto nas listagens 19 e 20.

```

package br.jm.servico;

import java.util.List;

import br.jm.entidade.Contato;

public interface ContatoIF {

    public void salvarContato(Contato contato);

    public Contato acharContato(Long id);

    public List<Contato> listarContatos();

    public void removerContato(Long id);

    public void removerContatos();
}

```

Listagem 19 - ContatoIF.java

```

package br.jm.servico;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import br.jm.entidade.Contato;
import br.jm.persistencia.ContatoDAO;

public class ContatoImpl implements ContatoIF {

```

```

private ContatoDAO contatoDAO;

public ContatoImpl() {
}

public ContatoImpl(ContatoDAO contatoDAO) {
    this.contatoDAO = contatoDAO;
}

public Contato acharContato(Long id) {
    return contatoDAO.acharContato(id);
}

public List<Contato> listarContatos() {
    List<Contato> lista = new ArrayList<Contato>();
    List<Contato> lista2 = contatoDAO.obtemListaContato();

    if(lista2 != null) {
        lista.addAll(lista2);
    }

    return Collections.unmodifiableList(lista);
}

public void removerContato(Long id) {
    contatoDAO.removerContato(id);
}

public void salvarContato(Contato contato) {
    contatoDAO.adiciona(contato);
}

public void removerContatos() {
    contatoDAO.removeContatos();
}

```

Listagem 20 – ContatoImpl.java

Integrando JPA ao Spring

Agora vamos configurar para que o spring instancie nosso *EntityManagerFactory* para que possamos acessar o banco de dados, tendo nossos daos configurados como spring beans.

```

<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
<property name="entityManagerFactory"
ref="entityManagerFactory"></property>
<property name="dataSource" ref="dataSource"></property>
</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>

```

Criamos um bean *chamado transactionManager*, estamos aqui utilizando nomes comuns podem usar qualquer nome para o atributo **id** que serve para o spring identifica-los e você poder instancia-los em outros objetos.

No atributo **class** informamos o nome da classe e pacote que ela pertence, temos agora a tag `<property/>` que como parametro possui **name** que é o nome da propriedade, ou seja nome do atributo da classe, **ref** é que se esta atribuindo uma referencia ao objeto *entityManagerFactory* que é um bean no Spring.

Temos também a propriedade chamada **dataSource** que recebe uma referencia do bean *dataSource*, abaixo temos a tag `<tx:annotation-driven/>` que possui o parametro **transaction-manager** que passamos o bean *transactionManager*.

Mas, nos não criamos isso? Para que declaramos esses beans e esse annotation-driven?

Isso é utilizado pelo Spring para podermos instanciar nosso **EntityManagerFactory** configurando todos seus parametros e ainda dar suporte as annotations da especificação EJB 3.0, ou seja, estamos preparando o spring para se integrar a JPA.

Agora vamos criar o bean *dataSource*.

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName"
value="com.mysql.jdbc.Driver"></property>
  <property name="url"
value="jdbc:mysql://localhost:3306/javamagazine"></property>
  <property name="username" value="root"></property>
  <property name="password" value="root"></property>
</bean>
```

Neste bean iremos configurar nosso *dataSource*, vocês podem ver que estaremos delegando ao spring todas as funções de instanciação de nossas actions e daos, como também nossas classes de serviço. A utilização de spring pode ser útil ate na utilização de webservices, mas acredito que isso seja assunto para um outro artigo.

Temos as *propriedades* **driverClassName** onde informamos o nome do *driver*, **url** onde configuramos nossa url de conexão, e as propriedades **username** e **password** são como o próprio nome diz, usuário e senha do banco de dados.

Com nosso *dataSource* pronto so nos resta agora criamos nosso bean *entityManagerFactory* para que possamos instanciar nossos daos.

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="persistenceUnitName" value="testeStruts"></property>
  <property name="dataSource" ref="dataSource"></property>
  <property name="jpaDialect">
    <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"></bean>
  </property>
  <property name="jpaVendorAdapter">
    <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
      <property name="database" value="MYSQL"></property>
      <property name="showSql" value="true"></property>
      <property name="generateDdl" value="true"></property>
    </bean>
  </property>
</bean>
```

A primeira propriedade é **persistenceUnitName** que é atribuido o valor *testeStruts* que esta configurado no arquivo **persistence.xml**, que seria o lugar onde normalmente iriamos configurar nossa url de conexão, usuário, senha..etc, aqui criamos sim o arquivo *persistence.xml* e configuramos uma unidade de persistencia, mas apenas configuramos o nome para *testeStruts* nada mais, como podem ver na listagem 17.

Configuramos a propriedade **dataSource** do nosso bean *entityManagerFactory* com uma referencia ao bean *dataSource* que criamos.

Temos a propriedade **jpaDialect** que tem que ser criada apontando para a classe [org.springframework.orm.jpa.vendor.HibernateJpaDialect](#) que possui as propriedades que

nos permite configurarmos a implementação do hibernate da JPA.

Agora na propriedade **jpaVendorAdapter** criamos um bean para a classe org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter que possui as algumas propriedades que nos iremos configurar neste artigo para podermos criar nosso **EntityManagerFactory**, são a propriedade **dataBase** onde *informaremos* **MYSQL** que é o banco que estaremos utilizando, a propriedade **showSql** com o valor **true**, para que seja exibido a sql gerada pelo hibernate.

E por fim a *propriedade* **generateDdl** com o valor configurado para **true** para que ele crie nossas tabelas caso essas não existam, se existirem não serão recriadas, semelhante a atribuir o valor **update** na propriedade `hibernate.hbm2ddl.auto` que é a que também estamos usando aqui, so não estamos vendo diretamente.

Pronto agora o spring já esta configurado para nos fornecer um **EntityManagerFactory** que possamos utilizar para acessar o banco de dados.

Mas, ate agora deu um trabalho para configurar o spring com jpa, o que temos que fazer para instanciar nossos daos?

```
<bean id="loginDao" class="br.jm.persistencia.LoginDAO">
  <constructor-arg ref="entityManagerFactory"></constructor-arg>
</bean>

<bean id="contatoDao" class="br.jm.persistencia.ContatoDAO">
  <constructor-arg ref="entityManagerFactory"></constructor-arg>
</bean>
```

criamos um bean *loginDao* que é a classe `LoginDAO` e recebe no construtor a referencia para o bean *entityManagerFactory* e o bean *contatoDao* também recebe no construtor uma referencia ao bean *entityManagerFactory*.

Pronto nossos dois daos, já tiveram suas dependencias configuradas e sempre que precisarmos teremos nossos daos prontos para o uso.

Vamos ver nossos servicos que irão prover o acesso ao banco de dados para nossas actions.

```
<bean id="servicoContato" class="br.jm.servico.ContatoImpl"
scope="singleton">
  <property name="contatoDAO" ref="contatoDao"></property>
</bean>

<bean id="servicoLogin" class="br.jm.servico.LoginImpl" scope="singleton">
  <property name="loginDAO" ref="loginDao"></property>
</bean>
```

Aqui esta sendo utilizado a *Setter Injection*, pois ate agora estavamos utilizando a *Constructor Injection* a propriedade do bean *servicoContato* `contatoDAO` será usado o seu método `set` para injetar o bean `contatoDao`.

Assim também foi configurado o *servicoLogin*, so que já estamos utilizando outro atributo chamado **scope** que configuramos como **singleton** que para criar somente uma instancia desse objeto, vamos ver a outra opção vendo nossas actions.

Integrando o Struts 2 ao Spring

```
<bean id="insereContato" class="br.jm.actions.InsereContatoAction"
scope="prototype">
```

```

    <property name="servico" ref="servicoContato"></property>
  </bean>

  <bean id="listaContatos" class="br.jm.actions.ListaContatosAction"
scope="prototype">
    <property name="servico" ref="servicoContato"></property>
  </bean>

  <bean id="login" class="br.jm.actions.LoginAction" scope="prototype">
    <property name="servico" ref="servicoLogin"></property>
  </bean>

  <bean id="removeContato" class="br.jm.actions.RemoverContatoAction"
scope="prototype">
    <property name="servico" value="servicoContato"></property>
  </bean>

```

Criamos o bean *insereContato* que recebe em seu atributo *servico* por o bean *servicoContato* e esta com o **scope** configurado para **prototype** que informa que deve ser criada uma instancia deste objeto cada vez que for requisitado.

No bean *listaContatos* recebemos também uma referencia do bean *servicoContato*. Vemos que o bean *removeContato* recebe não uma referencia mas sim uma atribuição por valor do bean *servicoContato*.

E por fim o bean *login* que recebe uma referencia ao bean *servicoLogin* que prove acesso ao banco de dados.

Pronto, basta acessar <http://localhost:8080/Cadastro> que nossa aplicação exemplo estara funcionando tendo a integração do Struts 2 + AJAX + JPA + Spring.

Simple não!?!

Conclusão

Como pode ter sido observado durante este artigo, tecnologias como Struts 2, JPA, Spring auxiliam bastante o trabalho do desenvolvedor.

Embora seja recente o Struts 2, já esta disponivel a versão 2.0.8 do Struts 2, e podemos esperar ainda mais amadurecimento desta framework conforme o tempo, mas suas facilidades como a integração com AJAX, ou para o desenvolvimento da camada de visão realmente são pontos positivos que devem levar em conta para a utilização desta framework, sem contar que a mesma oferece a possibilidade de se integrar com a versão antiga do struts, o que facilita o processo de migração para quem já tem sistemas com o Struts 1x.

Quanto a JPA, quem já teve que persistir objetos enormes no banco de dados, ou realizar consultas complexas, sabe o quanto uma framework de mapeamento O/R ajuda no processo de desenvolvimento. Enquanto frameworks como o Hibernate possuem funcionalidades a mais que a JPA, um exemplo disso é o *Criteria* que para quem já desenvolve com hibernate pode sentir falta na JPA, mas já esta nos planos para a próxima versão da JPA a agregação dessa e outras funcionalidades do hibernate e outras frameworks O/R, o que definitivamente incentiva o uso desta em sistemas de produção.

Voces podem ter percebido que temos toda a instanciação dos objetos de nossa aplicação em único arquivo, um dos pontos altos de se trabalhar com injeção de dependencias é buscar o desacoplamento do codigo, a utilização de interfaces ajuda bastante, mas não resolve todos nosso problemas, por isso cada vez mais desenvolvedores aderem ao uso do Spring em suas aplicações.

Espero ter demonstrado com este artigo que estas tecnologias estão aí para facilitar a vida de nós, desenvolvedores Java que estamos sempre buscando agilidade no desenvolvimento/manutenção de nossas aplicações e principalmente a busca em se

desenvolver aplicações web dinamicas com facilidade e tornar nossa aplicação fácil de migrar para outros bancos de dados caso necessário.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">

  <persistence-unit name="testeStruts"/>
```

Listagem 13 - persistence.xml

| | |
|---|-----------|
| freemarker-2.3.8.jar ognl-2.6.11.jar struts2-core-2.0.6.jar struts2-spring-plugin-2.0.6.jar xwork-2.0.1.jar | Struts 2 |
| spring.jar spring-jpa.jar | Spring |
| antlr-2.7.6.jar asm.jar asm-attrs.jar cglib-2.1.3.jar commons-collections-2.1.1.jar commons-logging-1.0.4.jar dom4j-1.6.1.jar ehcache-1.2.3.jar ejb3-persistence.jar hibernate3.jar hibernate-annotations.jar hibernate-commons-annotations.jar hibernate-entitymanager.jar javassist.jar jboss-archive-browsing.jar jta.jar | Hibernate |
| mysql-connector-java-5.0.5-bin.jar | MySQL |
| Dwr.jar | DWR |

Figura 5 - Dependências do projeto

ONGL : **O**bject-**G**raph **N**avigation **L**anguage - linguagem para obter e atribuir propriedades de objetos java.

| | |
|-------------------------------|---|
| Downloads | |
| Struts 2.0.6 | http://struts.apache.org/2.0.6/index.html |
| Spring Framework 2.0.6 | http://www.springframework.org/download |
| Hibernate Core 3.2.4 | http://www.hibernate.org/6.html |
| Hibernate Annotations 3.3.0 | http://www.hibernate.org/6.html |
| Hibernate EntityManager 3.3.1 | http://www.hibernate.org/6.html |
| Eclipse Eudora | http://download.eclipse.org/webtools/downloads/drops/R2.0/R-2.0-200706260303/ |

| | |
|-----------------|--|
| Para Saber Mais | |
| Hibernate | http://www.hibernate.org/5.html |
| Spring 2 | http://www.springframework.org/documentation http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework |
| Struts 2 | http://struts.apache.org/2.0.6/docs/home.html |