Aula 1

Introdução ao JavaServer Faces

Este material de apoio tem como base ajudar o aluno a entender os principais conceitos de JSF, assim como conter exercícios de fixação dos conteúdos abordados.

O desenvolvimento Web e o protocolo HTTP

Antes de mergulharmos nos JSF, precisamos entender os conceitos básicos do protocolo HTTP e como ele é essencial no desenvolvimento Web.

Nessa abordagem há um servidor central, onde a aplicação é executada e processada, e todos os clientes podem acessá-la através do protocolo HTTP.

Basta que o usuáirio possua um navegador web, como Firefox ou Internet Explorer, que receberá o HTML, o CSS e o JavaScript, que são afinal tecnologias que o browser entende.

Entre cada requisição (*request*), trafega o HTML do lado servidor (*Server Side*) para o computador do cliente (*Client Side*). Em nenhum momento a aplicação está salva no cliente. Todas as regras da aplicação estão sendo processadas no lado do servidor. Por isso, essa abordagem também foi chamada de "cliente magro".

Essa arquitetura é de fácil manutenção e tem um gerenciamento centralizado, pois temos um lugar - central -, onde a aplicação é executada, mas ainda é preciso conhecer bastante de HTML, CSS e JavaScript para definir a interface com o usuário.

Também não há mais eventos, mas sim um modelo bem diferente orientado a requisição e resposta. Além disso, ainda é preciso conhecer o protocolo HTTP. Assim, toda essa responsabilidade fica a cargo do desenvolvedor.

Frameworks Web baseados em componentes

No mundo Java há algumas opções como JavaServer Faces (JSF), Apache Wicket, Vaadin, Tapestry ou GWT, do Google. Todos eles são *frameworks* web baseados em componentes.

Analisando um pouco melhor o JSF, percebemos que ele é na verdade um padrão ou especificação que faz parte do ***Java Enterprise Edition*** (Java EE). Por ser uma especificação, ou *Java Specification Request* (JSR), ele é mantido dentro do *Java Community Process* (JCP).

Vamos procurar então o JSF no site da JCP. Digitando 314 na busca das JSRs (buscar das especificações), é mostrado como resultado a última versão do JSF. Navegando no resultado, podemos ver todos os documentos disponíveis para os interessados na implementação dessa especificação JSF.
Baseado nesta especificação, há várias implementações. A mais famosa, e também implementação referencial (RI), é a **Oracle Mojarra**, aquela que mostra como o JSF deveria se comportar. Outra implementação famosa é da *Apache Software Foundation*, e se chama MyFaces.
Neste caso baixaremos a implementação Mojarra no link indicado para utilizá-lo depois em nosso projeto. Acessando o site javaserverfaces.java.net, na parte de *downloads*, podemos encontrar o *link* para baixar o JAR do Mojarra.

Introdução ao JSF com Mojarra e PrimeFaces

Como dito antes, nosso projeto utilizará a implementação Mojarra do JSF. Ela já define o modelo de desenvolvimento e oferece alguns componentes bem básicos. Nada além de inputs, botões e ComboBox simples. Não há componentes sofisticados dentro da especificação. Isto é proposital, pois o mundo web evolui rápido (principalmente na questão das interfaces gráficas).

Para atender a demanda dos desenvolvedores por componentes mais sofisticados, existem várias extensões do JSF que seguem o mesmo ciclo e modelo da especificação. São exemplos dessas bibliotecas: **PrimeFaces**, **RichFaces** ou **IceFaces**.

Todos eles oferecem *ShowCases* na web para mostrar seus componentes e suas funcionalidades. Daremos uma olhada no **PrimeFaces**, acessando o site primefaces.org. Podemos ver no ***demo online*** uma lista de componentes disponíveis. Para este treinamento usaremos **Mojarra** com o **PrimeFaces**.

Exercícios - Parte 1

1) O JSF é um framework que une o melhor do desenvolvimento Web e Desktop. Pensando assim, podemos dizer que o JSF:

   - Junta a facilidade de manutenção e implantação de aplicações web com as vantagens do desenvolvimento orientado ao componente utilizado na programação Desktop durante anos.

   - Junta a facilidade de manutenção de aplicações web e a facilidade de implantação de aplicações Desktop.

- *Junta a facilidade de manutenção e implantação de aplicações Desktop e o poder de aplicações web.*

2) São características comuns no desenvolvimento RAD (Rapid Application Development) e que também podem ser encontradas no desenvolvimento com JavaServer Faces:

- Componentes passivos, dificuldade de atualização e ordenação em árvores.

- **Componentes ricos, orientado ao evento e mantem o estado dos componentes (stateful).**

- Perda do estado dos componentes (stateless), uso de linguagem natural e organização em grafos.

3) Quais são as vantagens de seguir um especificação/padrão Java EE como JSF?

- **Um padrão é um consenso de boas praticas no mercado, formalmente descrito para cada um seguir. Sendo assim o JSF padroniza o desenvolvimento web que facilita a troca de experiencias e conhecimento. Além disso não há vendor-lockin (aprisionamento tecnológico).**

- Não há real vantagem.

- O JSF padroniza o desenvolvimento web que facilita a troca de experiencias e conhecimento mas infelizmente há vendor-lockin (aprisionamento tecnológico).

- Uma especificação deve ser evitada pois é mantido de curto prazo.

4) Mojarra e MyFaces são

- bibliotecas criadas por terceiros que não seguem a especificação JSF
- **implementações que seguem a especificação JSF.**
- especificações distintas, cada uma com suas vantagens e desvantagens.

5) A especificação define o modelo de desenvolvimento e oferece alguns componentes bem básicos, nada além de inputs, botões e combo boxes simples. Isso é intencional, porque:

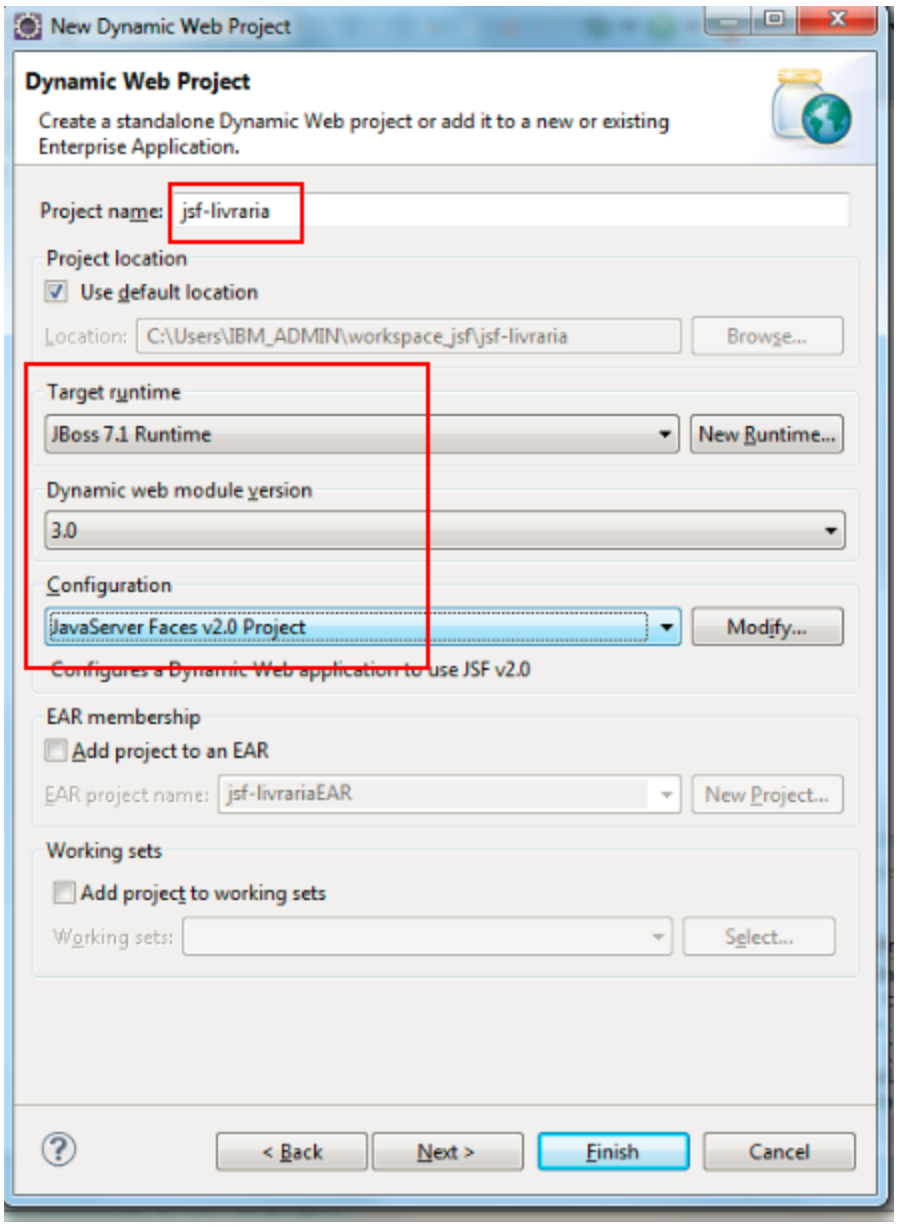
- Os criadores da especificação se preocuparam apenas em guardar o estado de aplicações web.
- Já são suficientes para resolverem todos os problemas conhecidos.
- **O mundo web evolui rápido, principalmente a interface gráfica.**

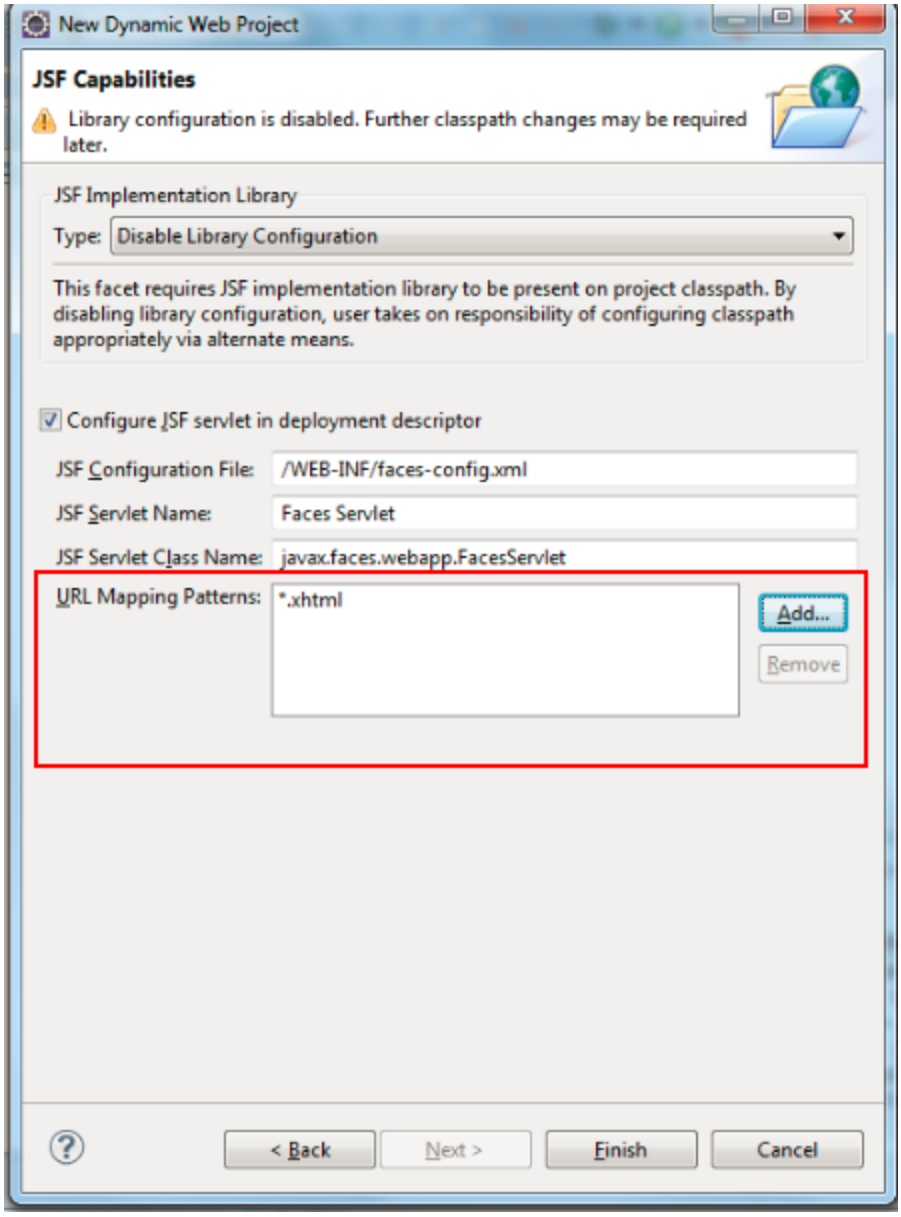
6) Existem as implementações do JSF da Oracle e da Apache e nada impede que outras implementações sejam criadas seguindo a especificação. Além delas, há extensões que adicionam componentes mais sofisticados. Exemplos delas são:

- Primefaces, myfaces e richfaces.
- **Primefaces, richfaces e icefaces.**
- Primefaces, mojarra e richfaces.

Preparando o ambiente

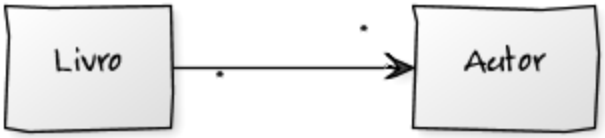
Criando um projeto




New Dynamic Web Project
JSF Capabilities
Library configuration is disabled. Further classpath changes may be required later.
JSF Implementation Library
Type: Disable Library Configuration
This facet requires JSF implementation library to be present on project classpath. By disabling library configuration, user takes on responsibility of configuring classpath appropriately via alternate means.
Configure JSF servlet in deployment descriptor
JSF Configuration File: /WEB-INF/faces-config.xml
JSF Servlet Name: Faces Servlet
JSF Servlet Class Name: javax.faces.webapp.FacesServlet
URL Mapping Patterns: *.xhtml
Add...
Remove
< Back
Next >
Finish
Cancel

Aplicação de exemplo

O nosso projeto web facilitará o trabalho em uma Livraria, onde o usuário poderá cadastrar livros e autores. Trabalharemos com objetos do tipo **Livro** associados a **autores**. Trata-se de uma relacionamento **muitos-para-muitos**. Antes de modelarmos as classes, vamos criar e configurar o projeto.



Vamos codar!!

Tendo nosso projeto JSF criado, vamos começar com o cadastro de livros. O objetivo é criar um formulário com os componentes da especificação.

Vamos selecionar a pasta WebContent, clicar com o botão direito, *New HTML File*. O arquivo se chama **livro.xhtml**. Cuidado com a extensão que deve ser XHTML. Ao apertar next, podemos escolher um template. Usaremos *xhtml 1.0 transitional*.

No arquivo apagaremos tudo que está dentro das tags HTML, pois utilizaremos os componentes JSF. Para declará-los é preciso adicionar um XML namespace na abertura da tag HTML.

Para tal, digitamos xmlns:h, onde o "h" é o apelido do namespace para a uri "http://java.sun.com/jsf/html". Ctrl + Espaço ajuda a auto-completar. Atenção para não confundir com o namespace JSTL.

```
<? xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

</html>
```

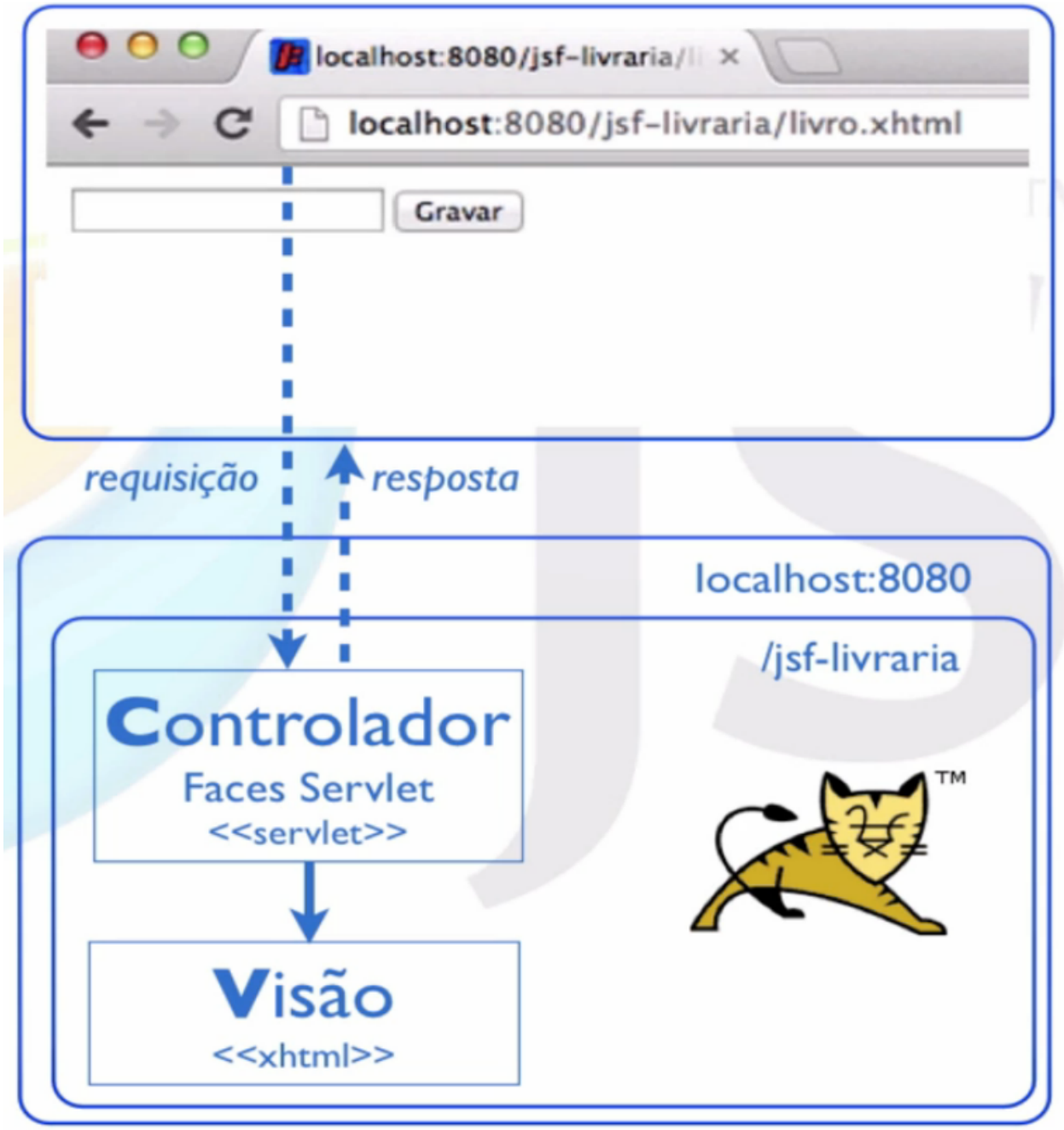
Como podemos observar, utilizaremos o apelido ***"h"*** para declarar os componentes JSF. O primeiro componente que usaremos é o ***h:body*** que define o corpo da página. Dentro do body vamos declarar o formulário através da tag h:form. Repare que aqui, diferente da tag form do HTML, o componente JSF não possui um atributo action.

Para a criação do formulário, vamos utilizar o componente que captura uma entrada do usuário, o ***h:inputText***. Vamos também utilizar um botão para executar uma ação. A especificação define comandos para isto. Nesse caso, um ***h:commandButton***. Com o atributo value definimos "Gravar", que aparecerá na tela.

```
<!-- cabeçalho omitido -->
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html">
    <h:body>
        <h:form>
            <h:inputText />
            <h:commandButton value="Gravar" />
        </h:form>
    </h:body>
</html>
```

já podemos testar a página pelo navegador, accessando http://localhost:8080/jsf-livraria/livro.xhtml.

Entendendo melhor os conceitos de View e Controller no JSF



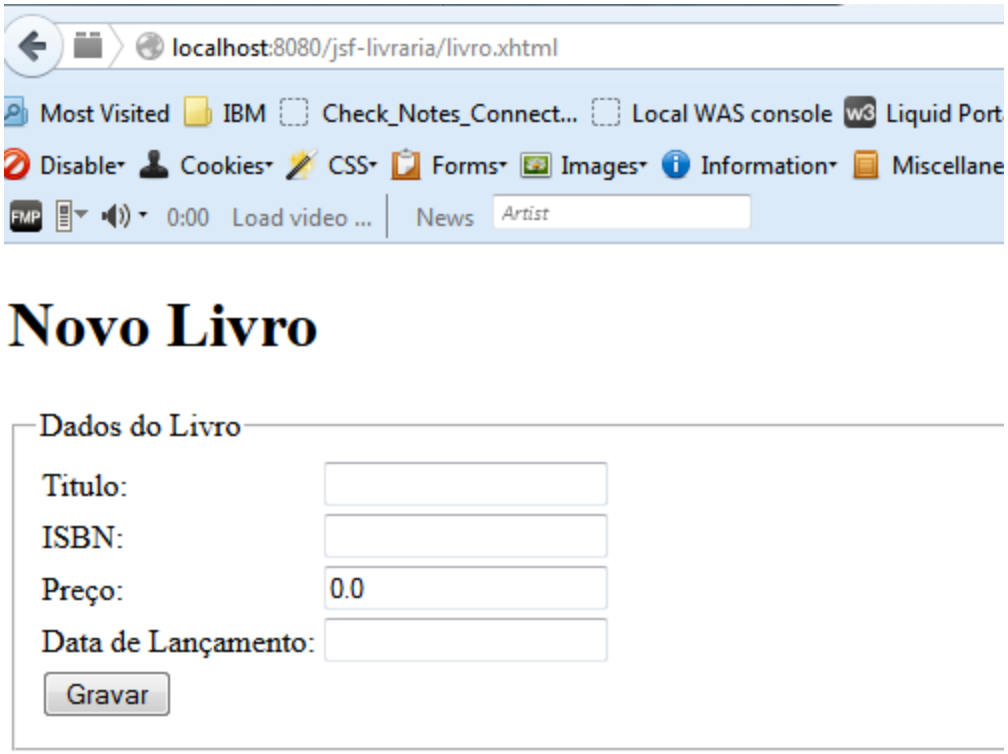
Vamos deixar nosso formuário de cadastro de livros mais interessante:

```
<!-- cabeçalho omitido -->
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html">
    <h:body>
        <h1>Novo Livro</h1>
        <h:form>
            <fieldset>
                <legend>Dados do Livro</legend>
                    <h:outputLabel value="Titulo:" for="titulo" />
                    <h:inputText id="titulo" />
                    <h:outputLabel value="ISBN:" for="isbn" />
                    <h:inputText id="isbn" />
                    <h:outputLabel value="Preço:" for="preco" />
                    <h:inputText id="preco" />
                    <h:outputLabel value="Data de Lançamento:" for="dataLancamento" />
                    <h:inputText id="dataLancamento" />
                    <h:commandButton value="Gravar" />
            </fieldset>
        </h:form>
    </h:body>
</html>
```

Agora vamos inserir o ***h:panelGrid*** para deixar nosso formuário mais organizado!

```
<!-- cabeçalho omitido -->
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:h="http://java.sun.com/jsf/html">
    <h:body>
        <h1>Novo Livro</h1>
        <h:form>
            <fieldset>
                <legend>Dados do Livro</legend>
                <h:panelGrid columns="2">
                    <!-- inputs omitido -->
                    <h:commandButton value="Gravar" />
                </h:panelGrid>
            </fieldset>
        </h:form>
    </h:body>
</html>
```

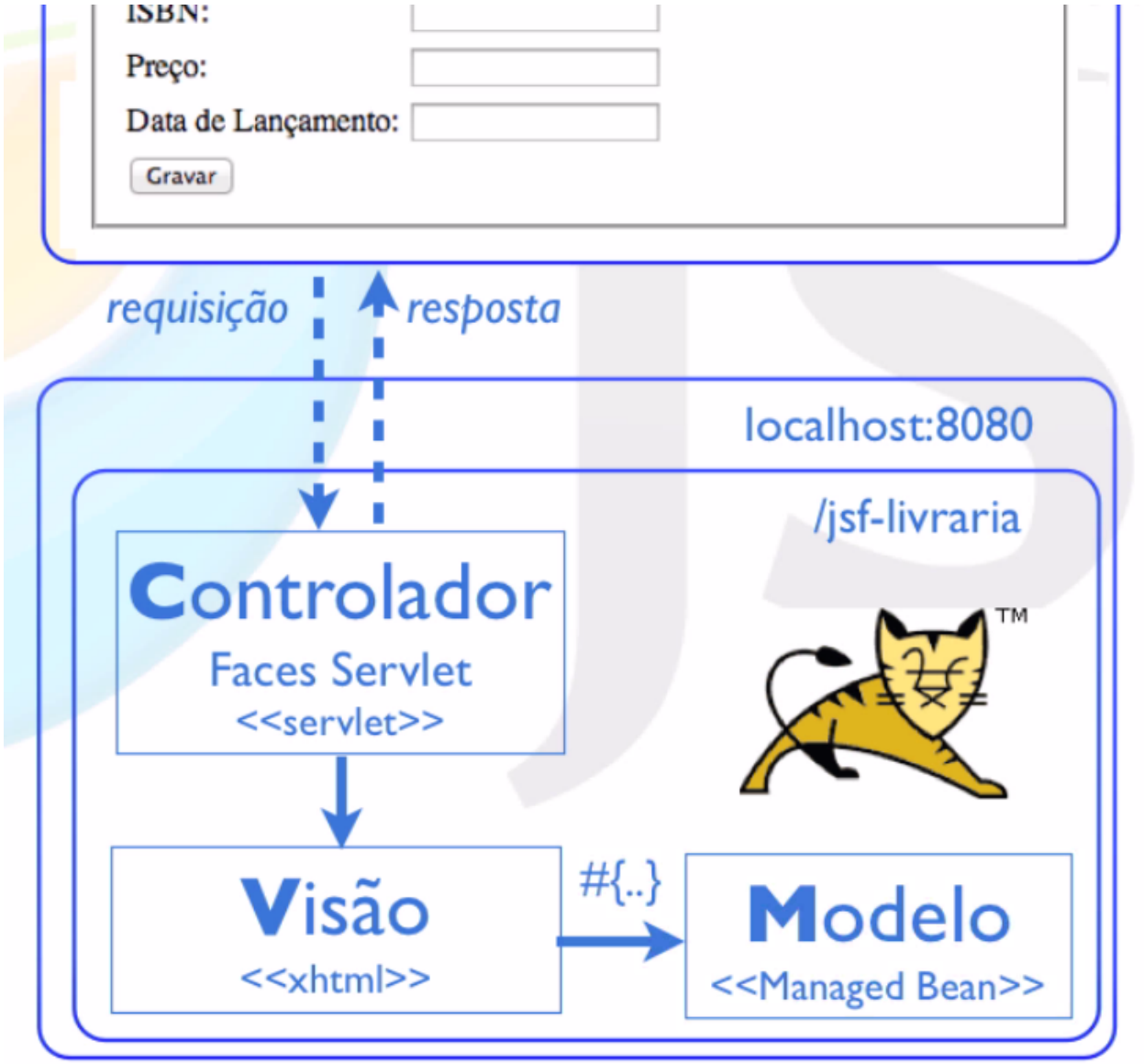
Ao final do redesign, teremos um form parecido com este:



Acoplando interface com classe Java - Managed Beans

Até agora temos apenas uma interface, mas precisamos executar regras de negócio da aplicação, como por exemplo: **gravar um livro**. Ou seja, quando o usuário apertar o botão "Gravar" no navegador, este componente disparará a execução de um método no lado do servidor. Para isso vamos criar uma **classe, associando-a com o componente**.

A figura abaixo explica a correlação das interfaces JSF com os componentes, os Managed Beans:

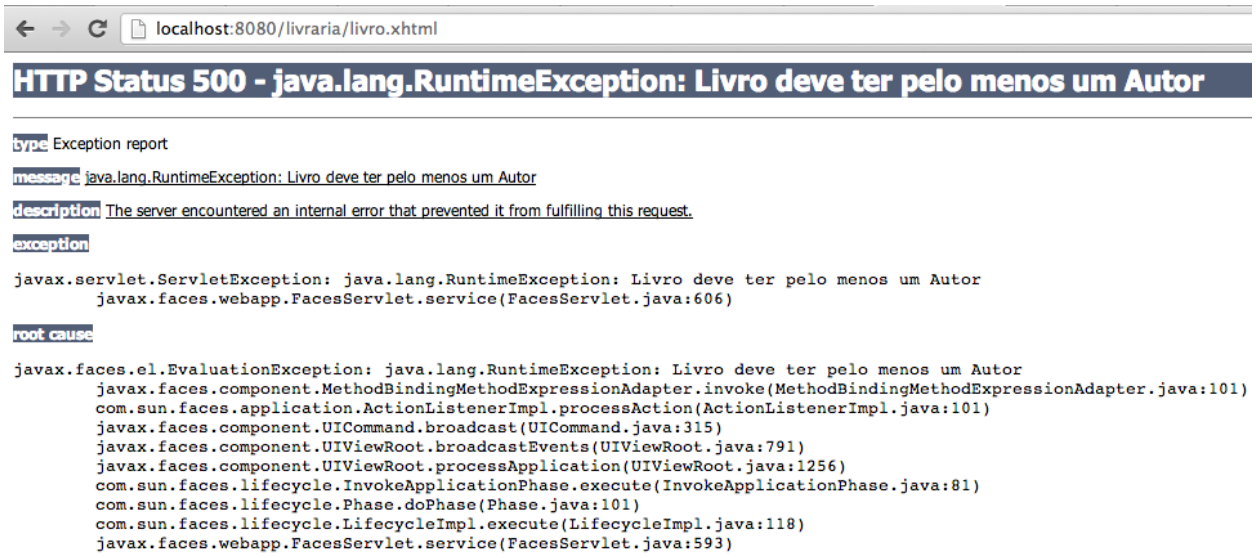


Acompanhe com o instrutor como implementar esta acoplação!

Aula 2

Relacionamento entre Livro e Autor

Vamos completare o formulário do Livro e criar o relacionamento entre Livro e Autor. A nossa aplicação já está rodando e vamos testar uma vez a inserção de um livro. Ao preencher o formulário e apertar o botão para gravar, recebemos uma exceção.



A mensagem da exceção indica que **não podemos gravar um livro sem autor**. Isso faz sentido, pois não deve existir um livro sem autor. O teste se o livro possui um Autor ou não já foi implementado nesse projeto. Ao abrir o LivroBean, notamos que dentro do método gravar() há um if que verifica se existe pelo menos um Autor, caso contrário, lança a exceção já vista.

```
public String gravar() {
    System.out.println("Gravando livro " + this.livro.getTitulo());

    if (livro.getAutores().isEmpty()) {
        throw new RuntimeException("Livro deve ter pelo menos um Autor.");
    }

    new DAO<Livro>(Livro.class).adiciona(this.livro);
    // limpando a tela
    this.livro = new Livro();
    return null;
}
```

Criação e definição do combobox

Vamos implementar o relacionamento no formulário **livro.xhtml**. Para selecionar o Autor do Livro, adicionaremos um *combobox*. Para isso criaremos um novo fieldset e, dentro dele, um legend indicando a seção do autor. Teremos um outputLabel com o valor *Selecione Autor*. O componente que representa um combobox se chama h:selectOneMenu. Vamos adicioná-lo depois do h:outputLabel. Além do combobox, também criamos um botão para confirmar a seleção do autor. O h:commandButton com o valor *Gravar Autor*.

```
<fieldset>
    <legend>Seleção do Autor</legend>
    <h:outputLabel value="Selecione um Autor:" for="autor" />
    <h:selectOneMenu value="#{livroBean.autorId}">
        <f:selectItems value="#{livroBean.autores}" var="autor"
            itemLabel="#{autor.nome}" itemValue="#{autor.id}" />
    </h:selectOneMenu>
    <h:commandButton value="Selecionar Autor"
        action="#{livroBean.gravarAutor}" />
    <br />
    <h:dataTable value="#{livroBean.autoresDoLivro}" var="autor">
        <h:column>
            <h:outputText value="#{autor.nome}" />
        </h:column>
    </h:dataTable>
</fieldset>
```

Preenchendo o h:selectOneMenu

O próximo passo é o preenchimento do selectOneMenu com os autores e o usuário fará isso, selecionando o nome do autor. Para tal, precisamos preencher o selectOneMenu com "*items*". Isto é feito por um outro componente que não faz parte do namespace "h", sendo necessário declarar uma nova biblioteca no início da página. Ela também já vem com a especificação e tem a URI *http://java.sun.com/jsf/core*. Utilizaremos o apelido padrão "f" para referenciá-la. Após a declaração, podemos utilizar os novos componentes, no nosso caso, f:selectItems. Para especificar os itens do combobox, utilizaremos o atributo value com a expression language #{livroBean.autores}. Nessa expressão, chamaremos um método da classe LivroBean, que devolverá uma lista de autores.

```
<fieldset>
    <legend>Seleção do Autor</legend>
    <h:outputLabel value="Selecione um Autor:" for="autor" />
    <h:selectOneMenu value="#{livroBean.autorId}">
        <f:selectItems value="#{livroBean.autores}" var="autor"
            itemLabel="#{autor.nome}" itemValue="#{autor.id}" />
    </h:selectOneMenu>
    <h:commandButton value="Selecionar Autor"
        action="#{livroBean.gravarAutor}" />
    <br />
    <h:dataTable value="#{livroBean.autoresDoLivro}" var="autor">
        <h:column>
            <h:outputText value="#{autor.nome}" />
        </h:column>
    </h:dataTable>
</fieldset>
```

Ao abrir a classe, adicionaremos o método getAutores(). Dentro do mesmo, aproveitaremos o DAO, que possui o método listaTodos().

```
public List<T> listaTodos() {
    EntityManager em = new JPAUtil().getEntityManager();
    CriteriaQuery<T> query = em.getCriteriaBuilder().createQuery(classe);
    query.select(query.from(classe));

    List<T> lista = em.createQuery(query).getResultList();

    em.close();
    return lista;
}
```

Voltando ao formulário, percebemos que falta definir o que queremos apresentar de cada autor. Vamos mostrar o nome do autor indicado pelo atributo itemLabel.

Para isso é preciso ter uma variável autor. Ou seja, o f:selectItems vai disponibilizar para cada item um autor. Baseado nesse autor, vamos usar a expression language #{autor.nome} para acessar o atributo nome. Igualmente será associado um valor ao item. Usaremos a ID do autor, ou seja, #{autor.id}.

```
<fieldset>
    <legend>Seleção do Autor</legend>
    <h:outputLabel value="Selecione um Autor:" for="autor" />
    <h:selectOneMenu value="#{livroBean.autorId}">
        <f:selectItems value="#{livroBean.autores}" var="autor"
            itemLabel="#{autor.nome}" itemValue="#{autor.id}" />
    </h:selectOneMenu>
    <h:commandButton value="Selecionar Autor"
        action="#{livroBean.gravarAutor}" />
    <br />
    <h:dataTable value="#{livroBean.autoresDoLivro}" var="autor">
        <h:column>
            <h:outputText value="#{autor.nome}" />
        </h:column>
    </h:dataTable>
</fieldset>
```

Recuperando dos valores dentro do ManagedBean

Agora vamos implementar o botão *Gravar Autor* e chamar um método ao clicar no botão. Mas antes disso é preciso saber qual autor o usuário selecionou no combobox. O componente h:selectOneMenu, como outros inputs, também possui um atributo value que usaremos para capturar a ID do autor selecionado.
<h:selectOneMenu value="#{livroBean.autorId}" id="autor"> <f:selectItems value="#{livroBean.autores}" var="autor" itemLabel="#{autor.nome}" itemValue="#{autor.id}"/> </h:selectOneMenu>

No f:selectItems definimos como itemValue a ID do autor, e justamente essa ID que recebemos na classe LivroBean. Vamos então criar no LivroBean um atributo autorId com um *getter* e *setter* para capturar a ID do autor.

@ManagedBean public class LivroBean { //outros trechos omitidos private Integer autorId; public void setAutorId(Integer autorId) { this.autorId = autorId; } public Integer getAutorId() { return autorId; } //outros trechos omitidos }

Para finalizar devemos criar o método para gravar o autor. Aqui não há novidade. Usaremos a expression language #{livroBean.gravarAutor} para chamar o método do LivroBean. Vamos então voltar à classe para implementar o método. O método na verdade não grava no banco, e sim associa apenas o autor com o livro usando o relacionamento já definido, mas para isso é preciso carregar o autor, pois temos apenas a ID dele. Usaremos o

DAO que possui um método buscaPorId(id) passando a ID do autor (autorId) como parâmetro. O retorno do método é o autor selecionado. Depois da busca do autor, vamos relacionar o livro com o autor pelo método auxiliar adicionaAutor(). Pronto! E quando gravarmos o livro, o JPA também criará o relacionamento no banco de dados.

@ManagedBean public class LivroBean { //outros trechos omitidos public void gravarAutor() { Autor autor = new DAO<Autor>(Autor.class).buscaPorId(this.autorId); this.livro.adicionaAutor(autor); } //outros trechos omitidos }

Resumo e teste dos componentes

Resumindo, o ***f:selectItems*** recebe uma lista de autores. No mesmo componente definimos o que queremos mostrar do autor e qual é o valor associado (e enviado no request). Para receber o autor selecionado, definimos o atributo autorId. Para associar o autor com o livro, criamos o método gravarAutor(), que é chamado pelo commandButton. Chegou a hora de testar a tela no navegador, mas antes vamos reiniciar o servidor. Ao atualizar a página aparece no navegador e o combobox preenchido com os autores. Ótimo! Vamos preencher os dados do livro. São eles: **titulo**, **isbn**, **preço** e **data**, e depois selecionaremos o autor. Não podemos esquecer de apertar o botão *Gravar Autor* para criar o relacionamento. No final gravamos o livro.

Entendo os escopos no desenvolvimento web

Para nossa surpresa recebemos novamente uma exceção, e pior, recebemos a mesma exceção acusando que o livro não possui um autor. Vamos tentar de novo, mas agora, para deixar mais claro, imprimimos o nome do autor no método gravarAutor() para verificar a existência do autor. Ao voltar no formulário, vamos gravar o autor e verificar o console. Aqui está tudo certo, o autor foi relacionado com livro. A implementação está certa. Vamos testar de novo e gravar o livro, mas infelizmente o exceção continua. Para entender o que aconteceu, vamos visualizar o fluxo da aplicação. Após ter preenchido o formulário, selecionamos o autor e apertamos o botão *Gravar Autor*. Isso disparou um request que o controlador passou para a árvore de componentes, a nossa tela em memória. Como associamos os componentes com o LivroBean, o controlador cria um objeto dessa classe e junto com o LivroBean, também é criado o Livro. No método gravarAutor(), carregamos o

Autor associando-o com o Livro. Até aqui é tudo como o esperado. Após isso, apertamos no formulário o botão *Gravar* para realmente gravar o Livro. Isso causou um novo request que novamente cai na nossa tela, porém agora é criado um **NOVO** livroBean e consequentemente um novo livro. Perdemos o relacionamento do livro com o autor pois o LivroBean antigo não existe mais. Em geral, cada request causa a criação de um novo LivroBean, pois a vida do **Managed Bean** dura apenas um request. Isso também se chama "*escopo de requisição*". Porém, na nossa aplicação, queremos uma vida mais longa pra ele. Queremos que o LivroBean exista enquanto a tela existir. Esse novo escopo se chama **ViewScoped** e sobrevive a vários requests.

Usando ViewScoped

A configuração do escopo no ManagedBean também é feita através de anotações. O padrão é o escopo de requisição que podemos deixar explícito com a anotação ***@RequestScoped***. Para configurar o ViewScoped usaremos a anotação ***@ViewScoped.***

Após reiniciar o servidor e atualizar a página, testamos novamente no navegador. Vamos inserir os dados e selecionar o Autor. Não podemos esquecer de apertar o botão *Gravar Autor*. Agora podemos gravar o livro.
Dessa vez não recebemos nenhuma exceção, e ao verificar o console no Eclipse, podemos ver o SQL gerado pelo JPA. Foi inserido o livro e o relacionamento no banco de dados. Coisa Linda!!

Exibindo autores do livro com o h:dataTable

Agora vamos melhorar mais um pouco a usabilidade da aplicação. Um livro pode ser escrito por vários autores e faz sentido mostrar todos os autores do livro antes de gravar o mesmo. Para que isso seja feito, vamos renderizar uma tabela abaixo do combobox. Mãos a obra, então. O componente o recebe uma lista pelo atributo value, novamente pela expression language #{livroBean.autoresDoLivro}.

```
<fieldset>
    <legend>Dados do Autor</legend>

    <!-- outros trechos omitidos -->

    <h:dataTable value="#{livroBean.autoresDoLivro}" >

    </h:dataTable>
</fieldset>
```

Já no LivroBean, criaremos um método que devolve uma lista de autores. Ao abrir a classe, vamos criar o método getAutoresDoLivro(), que devolve os autores do livro atual.

```
@ManagedBean
@ViewScoped
public class LivroBean {

    //outros trechos omitidos

    public List<Autor> getAutoresDoLivro() {
        return this.livro.getAutores();
    }
```

Agora só falta declarar a variável que representa um autor. Depois disso vamos definir o que queremos mostrar em cada coluna. Para isso usaremos o componente , que conterá o componente . O outputText usa a variável para declarar o nome do autor pela expression language. No nosso caso vamos usar apenas uma coluna.

```
<fieldset>
    <legend>Dados do Autor</legend>

    <!-- outros trechos omitidos -->

    <h:dataTable value="#{livroBean.autoresDoLivro}" var="autor">
        <h:column>
            <h:outputText value="#{autor.nome}"/>
        </h:column>
    </h:dataTable>
</fieldset>
```

Reinicie o servidor e teste novamente o formulário