Tópicos Especiais em Análise e Desenvolvimento de Sistemas

Prof. Rafael Odon (rafael.alencar@prof.una.br)

ORIENTAÇÕES PARA ENTREGA:

- O roteiro deverá ser feito continuando **com a mesma dupla** dos roteiros de JPA e ou **individual** para quem decidiu não fazer em dupla.
- Enviar até o início da aula do dia 02/05/2014 o código e as respostas de **todas as perguntas** desse roteiro para o e-mail do professor (rafael.alencar@prof.una.br) com o título:
  - `Roteiro JSF 02 – Nome do 1º Aluno / Nome do 2º Aluno`
- Lembre-se: **todas as informações** para responder as perguntas são dadas em **sala de aula**. Preste atenção na aula e faça notas pessoais!
- As **respostas** devem ser pessoais e **escritas com suas palavras**. Não serão aceitas respostas longas e/ou **copiadas** de forma literal do material de referência ou Internet.

Java Server Faces

Roteiro de Aula 02 – 23/04/2014

Esse Roteiro se baseia no uso dos softwares:

- Netbeans 7.4 para JAVA EE
- GlassFish 4.x
- Java SDK 1.7
- Java EE 6
- Java DB (Apache Derby)

Fase 1: Configurando a aplicação

1. Crie um novo projeto **Java Web** no Netbeans de nome **SysFrotaWeb**
2. Repita os passos do Roteiro 01 sobre Java Server Faces para utiliar JSF nesse projeto:
   I. Adicione a biblioteca **JSF 2** ao projeto
   II. Crie o arquivo **web.xml**
   III. Apague o arquivo **index.jsp**

Fase 2: Composição de telas por templates

Ao fazermos um sistema Web, é comum que todas as telas possuam elementos que se repetem como menus, banners, cabeçalhos e rodapés. O JSF permite que uma tela herde o layout de uma outra através do recurso de composição. Nessa fase iremos explorar esse conceito.

1. Crie um XHTML chamado template.xhtml com o seguinte código:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>SysFrota</title>
    </h:head>
    <h:body>
        <div class="cabecalho">
            <h1>SysFrota</h1>
            <div clas="menu">
```

```
                <h:form>
                    <h:commandLink action="index.xhtml" value="Inicio"/> |
                    <h:commandLink action="sobre.xhtml" value="Sobre"/>
                </h:form>
            </div>
        </div>

        <div class="corpo">
            <h:messages />
            <ui:insert name="corpo"></ui:insert>
        </div>

        <div class="rodape">
            <hr/>
            SysFrotaWeb - Novembro/2012
        </div>
    </h:body>
</html>
```

2. Observe que o XHTML acima define uma página como outra qualquer mas, no entanto, **nele foi definido através da tag <ui:insert /> um ponto de inserção.** Em outras palavras, estamos dizendo que o conteúdo que existirá naquele local será definido por páginas que utilizam o documento criado como template.

3. Crie o arquivo **index.xhtml** com o seguinte código:

```
<?xml version="1.0" encoding="UTF-8"?>
<ui:composition xmlns:ui="http://java.sun.com/jsf/facelets"
                template="template.xhtml">
    <ui:define name="corpo">
        <h2>Bem vindo!</h2>
    </ui:define>
</ui:composition>
```

4. Observe que o XHTML do **index.xhtml** não mais define um HTML completo, e **sua tag raíz é a tag <ui:composition />**.

5. Observe que o **atributo template da tag <ui:composition />** está apontando justamente para o XHTML criado anteriormente de nome template.xhtml.

6. Se no template.xhtml inserimos uma tag `<ui:insert name="corpo">`, na página que herda o template devemos **inserir uma tag correspondente** **`<ui:define name="corpo">`**

7. Observe que a ligação entre as duas tags é o nome definido e o nome referenciado. O trecho de XHTML que vier dentro do par de tag <ui:define /> será substituído no template utilizado.

8. Observe que uma vez que estamos trabalhando composição de layout, o restante HTML da página **index.html** foi definido no template utilizado (**template.xhtml**).

9. Limpe e construa seu projeto e em seguida execute-o. Acesse a página **index.jsf** e peça para ver seu código fonte. Verifique que o HTML gerado é realmente o HTML do **template.xhtml** + a definição do arquivo **index.xhtml**.

10. Perceba que no **template.xhtml** definimos 2 links, um para **index.xhtml** e outro para **sobre.xhtml**. Inspirado na solução de template vista para o **index.html**, construa a página **sobre.xhtml** utilizando o mesmo template já criado. Essa página deve conter seu nome, e-mail, data da criação do sistema e nome da disciplina.

11. Limpe e construa seu projeto e em seguida execute-o. Acesse a página **index.jsf.** Navegue para a página **sobre.jsf** através do menu definido no cabeçalho do template. Verifique se o layout da página **sobre.jsf** se manteve o mesmo da página **index.jsf**.

- É possível definir vários pontos de inserção em um template. Para que cada ponto de inserção seja utilizado, basta que as telas subjacentes que o utilizam definam seu conteúdo através de múltiplas tags **<ui:define />**, uma para cada **<ui:insert />** respectivo.

Fase 3: Integrando o projeto JSF atual com o antigo projeto JPA SysFrota

Vamos agora integrar nossa solução JSF com nossa solução JPA dos roteiros anteriores! Muita atenção. Repita alguns procedimentos dos Roteiros de JPA para que o projeto SysFrotaWeb possa utilizar JPA do ponto de onde paramos:

1. Adicione o Driver JDBC do banco de dados utilizado nas bibliotecas do projeto;
2. Adicione ao projeto a biblioteca **Hibernate JPA;**
3. Crie uma nova Unidade de Persistência no projeto, com qualquer configuração, afim de gerar o arquivo **persistence.xml** no projeto web. Em seguida edite o novo arquivo criado e copie nele o conteúdo do arquivo **META-INF/persistence.xml** do projeto antigo, afim de configurar a aplicação web para conectar no mesmo banco.
4. Remova do **persistence.xml** a linha que permite criar tabelas automaticamente, para que os dados não sejam perdidos a cada implantação da aplicação:
   `<property name="hibernate.hbm2ddl.auto" value="create-drop"/>`
5. Copie os pacotes **sysfrota.entidade**, **sysfrota.persistence** e **sysfrota.persiscente.dao e outros criados na 1ª etapa** para o projeto atual (Botão direito no pacote > Copiar, Botão direito na Pasta de códigos-fonte > Colar)

Fase 4: Adicione Controle Transacional por Requisição Web

Aplicações Web geralmente são configuradas para que a cada *request* uma transação seja aberta antes da execução do código da aplicação e fechada antes de entregar a resposta para o usuário. Dessa forma, todas as operações de alteração de banco que ocorrerem entre o clique na tela do navegador e a resposta exibida irão ficar debaixo de uma única transação. Para isso, vamos criar um filtro web que interceptará todas as requisições e fará esse tratamento.

- Adicione a classe **OpenSessionAndTransactionInView** ao pacote **sysfrota.web.filter** do seu projeto com o seguinte conteúdo:

```
package sysfrota.web.filter;

import java.io.IOException;
import javax.persistence.EntityManager;
import javax.persistence.EntityTransaction;
import javax.servlet.*;
import javax.servlet.annotation.WebFilter;
import sysfrota.persistence.JPAUtil;

@WebFilter(urlPatterns = "/*")
public class OpenSessionAndTransactionInView implements Filter {

    @Override
    public void init(FilterConfig filterConfig) throws ServletException { }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException,
ServletException {
        EntityManager em = JPAUtil.getEntityManager();
        EntityTransaction tx = em.getTransaction();
        try {
            tx.begin();
            chain.doFilter(request, response);
            tx.commit();
        } catch (Exception e) { // ou em caso de erro faz o rollback
            if (tx != null && tx.isActive()) {
                tx.rollback();
            }
            throw new ServletException(e);
        }
    }

    @Override
    public void destroy() {
```

```
        JPAUtil.close();
    }
}
```

- Observe que a classe criada é **um Filtro Web** que irá interceptar todas as requisições para o padrão de URL /*, ou seja, qualquer *request* da aplicação.
- Observe que **quando o filtro é executado**, ele abre a transação, executa o restante da aplicação, e ao fim faz o *commit*. Se uma exceção ocorrer ele irá fazer o *rollback*.
- Observe também que **quando a aplicação é finalizada** o filtro desaloca os recursos de persistência através do *JPAUtil.close()*;
- Caso exista, remova todo o **controle transacional** das suas classes DAO. Exemplo:

```
...
    public void remover(Fabricante fabricante) {
        em.getTransaction().begin();
        em.remove(fabricante);
        em.getTransaction().commit();
    }...
```

Ficará:

```
...
    public void remover(Fabricante fabricante) {
        em.remove(fabricante);
    }
...
```

- Limpe e construa sua aplicação e execute-a novamente. Adicione um *breakpoint* no filtro e verifique se a transação está sendo aberta e fechada a cada requisição.

Fase 5: Desenvolvendo um CRUD simples com JSF + JPA

Vamos criar nosso primeiro CRUD. A sigla CRUD vem do inglês: ***C**reate, **R**ead, **U**pdate, **D**elete.* Traduzindo, teríamos: Criar, Ler, Atualizar e Remover. Essas seriam quatro operações básicas que o cadastro de uma entidade precisa contemplar para permitir que o usuário manipule os dados do banco sem precisar utilizar um SGBD.

A grande maioria dos Sistemas de Informação possui algumas telas de CRUD, geralmente advindas de casos de uso de manutenção. No caso do sistema SysFrota em que estamos trabalhando, podemos também imaginar a existência dos seguintes casos de uso: **Manter Fabricantes, Manter Modelos, Manter Carros** e **Manter Características.**

Vamos iniciar com a construção do **Manter Fabricantes** por se tratar de uma entidade simples apenas com os atributos **id e nome.**

1. Crie a classe **FabricanteMB** no pacote **sysfrota.ui.managedbean** com o seguinte código:

```
package sysfrota.ui.managedbean;

import java.io.Serializable;
import java.util.List;
import javax.annotation.PostConstruct;
import javax.faces.application.FacesMessage;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;
import javax.faces.context.FacesContext;
import sysfrota.entidades.Fabricante;
import sysfrota.persistence.dao.FabricanteDAO;

@ManagedBean
@SessionScoped
public class FabricanteMB implements Serializable {

    private static final String PAGINA_EDICAO = "/fabricanteEdit.xhtml";
```

```
    private static final String PAGINA_LISTAGEM = "/fabricanteList.xhtml";

    private Fabricante fabricante;
    private List<Fabricante> lista;

    private FabricanteDAO fabricanteDAO = new FabricanteDAO();

    public FabricanteMB() {

    }

    @PostConstruct
    private void atualizar() {
        fabricante = new Fabricante();
        lista = fabricanteDAO.listarTodos();
    }

    public String editar(Fabricante fabricante) {
        this.fabricante = fabricanteDAO.carregarPeloId(fabricante.getId());
        return PAGINA_EDICAO;
    }

    public String criar() {
        this.fabricante = new Fabricante();
        return PAGINA_EDICAO;
    }

    public String salvar() {
        fabricanteDAO.salvar(fabricante);
        FacesContext.getCurrentInstance().addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_INFO, "O fabricante foi salvo com sucesso!",null));
        return irParaListagem();
    }

    public String remover(Fabricante f) {
        fabricanteDAO.remover(f);
        FacesContext.getCurrentInstance().addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_INFO, "O fabricante foi removido com sucesso!",null));
        return irParaListagem();
    }

    public String irParaListagem() {
        atualizar();
        return PAGINA_LISTAGEM;
    }

    public String cancelar() {
        return irParaListagem();
    }

    //getters & setters
    public List<Fabricante> getLista() {
        return lista;
    }

    public Fabricante getFabricante() {
        return fabricante;
    }

    public void setFabricante(Fabricante fabricante) {
        this.fabricante = fabricante;
    }
}
```

2. Observe que nosso ManagedBean basicamente representa um Modelo composto por **uma lista de Fabricante e uma instância de Fabricante.**

3. Observe o **método atualizar() anotado com @PostConstruct** e invocado em diversos pontos do código. Esse método prepara nosso MB utilização. Desa forma inicialmente nosso MB sempre começa com a lista mais atual de Fabricantes advinda do DAO, e com uma instancia nova de Fabricante.

4. Observe o **método editar()** que é uma ação que ao ser chamada, carrega do banco o fabricante escolhido e o coloca na instância de Fabricante do nosso MB, e em seguida redireciona para a tela de edição.

5. Observe o **método criar()** que é uma ação que ao ser chamada, define a instância de Fabricante do nosso MB como um fabricante novo e vazio, e em seguida redireciona para a tela de edição.

6. Observe o **método salvar()** que é um ação que ao ser chamada salva a instância de Fabricante do nosso MB. Perceba que, de acordo com o código que fizemos no DAO, se a instância é nova

será criado um Fabricante novo no banco, e se a instância não é nova, será feito uma atualização no banco.

7. Observe o **mecanismo de envio de mensagem do JSF** que avisa que a inserção foi feita com sucesso. Essa mensagem será apresentada pela tag <h:message /> do template.

8. Crie o arquivo **fabricanteList.xhtml** com o código abaixo:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
                xmlns:h="http://java.sun.com/jsf/html"
                xmlns:ui="http://java.sun.com/jsf/facelets"
                xmlns:f="http://java.sun.com/jsf/core"
                template="/template.xhtml">
    <ui:define name="corpo">
        <h:form>
            <h:commandButton action="#{fabricanteMB.criar()}" value="Novo fabricante" />
            <h2>Lista de fabricantes</h2>
            <h:dataTable value="#{fabricanteMB.lista}" var="f" border="1">
                <h:column>
                    <f:facet name="header">Id</f:facet>
                    #{f.id}
                </h:column>
                <h:column>
                    <f:facet name="header">Nome</f:facet>
                    #{f.nome}
                </h:column>
                <h:column>
                    <f:facet name="header">Opções</f:facet>
                    <h:commandButton action="#{fabricanteMB.editar(f)}" value="Editar" />
                    <h:commandButton id="confirm" value="Remover" action="#{fabricanteMB.remover(f)}" />
                </h:column>
            </h:dataTable>
        </h:form>
    </ui:define>
</ui:composition>
```

9. Observe que essa tela faz o **uso do template** definido na Fase 2.

10. Observe que existe **um botão que chama a ação de criar do MB**. Verifique no código do MB o que faz essa ação.

11. Observe o **uso da tag <h:dataTable />** para definir uma tabela de dados. Perceba que os dados da tabela vem da propriedade **lista** do MB, e cada dado será representado nas definições da tabela pela variável **f**.

12. Observe o **uso das tags <h:column />** para definir as colunas da tabela. O conteúdo de cada par de tags <h:column> define o conteúdo da célula, bem o cabeçalho da coluna através do facet **header.**

13. Observe que existe uma coluna para ações onde **um botão invoca a ação editar do MB** e **outro invoca a ação remover do MB**. Relembre o código do MB para entender o funcionamento!

14. Crie o arquivo **fabricanteEdit.xhtml** com o código abaixo:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<ui:composition xmlns="http://www.w3.org/1999/xhtml"
                xmlns:h="http://java.sun.com/jsf/html"
                xmlns:ui="http://java.sun.com/jsf/facelets"
                xmlns:f="http://java.sun.com/jsf/core"
                template="/template.xhtml">
    <ui:define name="corpo">
        <h2>Editar fabricante</h2>
        <h:form>
            <h:panelGrid columns="2">
                <h:outputLabel value="Nome" for="nome"/>
                <h:inputText id="nome" value="#{fabricanteMB.fabricante.nome}"
                             required="true"
                             label="Nome"/>
            </h:panelGrid>
            <h:commandButton action="#{fabricanteMB.salvar}" value="Salvar" />
            <h:commandButton action="#{fabricanteMB.cancelar}" value="Cancelar" immediate="true"/>
        </h:form>
    </ui:define>
```

```
</ui:composition>
```

14. Observe que essa tela também faz o **uso do template** definido na Fase 2.

15. Observe o sudo de um **<h:panelGrid /> com 2 colunas**. Esse componente faz com que seus filhos sejam dispostos na forma de um grid. Cada componente filho irá em uma coluna e, como definimos 2 colunas, a cada 2 componentes ele irá quebrar para a linha debaixo.

16. Observe que **o input que recebe o nome está mapeado para a propriedade nome da instancia de Fabricante** existente no nosso MB.

17. Observe que o input de nome foi marcado com **a propriedade required e o valor true**. Isso faz com que a fase de validação do JSF verifique se o campo foi preenchido ou não, gerando uma mensagem correspondente se necessário.

    - Lembre-se que as mensagens de validação estão sendo exibidas no <h:messages /> do template!

    - Perceba que o nome do campo exibido nas mensagens de validação é o valor do atributo label do campo.

18. Observe que o **os métodos Salvar e Cancelar foram mapeados nas ações dos botões** da tela de edição. Verifique o código do MB para entender o que esses métodos fazem.

19. Observe que a propriedade **immediate com valor true** foi definida para o Cancelar. Isso faz com que essa ação seja chamada imediatamente no ciclo de vida do JSF, evitando que o JSF tente validar e refletir no modelo os valores da tela. Sem essa propriedade, se o campo obrigatório não fosse preenchido, a mensagem de validação apareceria bloqueando a conclusão da ação de Cancelar.

20. Antes de testar, modifique o seu **template.xhtml** para contemplar também um **<h:commandLink />** direcionado para o método **#{fabricanteMB.irParaListagem}**, afim de dar acesso à listagem dos fabricantes através do menu da aplicação.

21. Limpe e construa seu projeto e em seguida execute-o. Acesse a listagem de fabricantes. Certifique-se de ter cadastrado no banco alguns Fabricantes para verificar o correto funcionamento da listagem. Para isso, execute a classse Bootstrap dos roteiros JPA incluindo fabricantes, sem que o banco seja apagado pela aplicação.

22. Tente cadastrar alguns fabricantes.

23. Tente cadastrar um fabricante sem informar o nome.

24. Tente remover um fabricante.

25. Tente editar um fabricante.

Fase 6: A sua vez de botar a mão na massa!

1. Adicione um campo texto “País de Origem” na sua entidade Fabricante. Evolua o CRUD simples para contemplar esse campo na listagem e na edição.

2. Baseando-se na solução de CRUD simples apresentada na fase 4, construa o CRUD da classe de entidade **Caracteristica** que também é simples e possui apenas **id** e **nome.**

RESPONDA

I. Que tag permite criar um ponto de inserção em um template JSF? Dê um exemplo.

II. Em que ponto de um XHTML podemos dizer que uma página herdará a definição de template de outra?

III. Que tag permite definir o conteúdo que irá substituir um ponto de inserção numa página JSF que utiliza um template? Dê um exemplo.

IV. Por que uma página que herda um template não possui a tag raiz <html />?

V. O que é “escopo” em um Managed Bean?

VI. Que escopo é definido pela anotação @javax.faces.bean.SessionScoped em um ManagedBean? Que outras anotações relativas a escopo existem nesse pacote do JSF? *(Obs: não confundir com anotações de escopo do pacote javax.enterprise.context que são do CDI).*

VII. O que a anotação @PostConstruct define para um método do ManagedBean?

VIII. Como enviar mensagens do ManagedBean para o componente <h:messages />? Exemplifique.

IX. Explique o funcionamento do componente <h:dataTable />. Onde nele podemos informar qual lista de dados será utilizada? Como definimos colunas? Como fazemos para utilizar cada dado da lista na definição das colunas?

X. O que são facets? Dê um exemplo.

XI. O que faz a propriedade immediate dos componentes de comando do JSF (commandButton e commandLink)?

XII. Como definimos um campo como requerido? Quem será responsável por validar se o campo egrequerido foi preenchido?