**JPA Entity Listeners**

**EntityListeners** permite você invocar algum código antes ou depois de certas operações de persistência.

**Detalhes**

**@PrePersist:** Chamado imediatamente antes que o objeto seja salvo na base de dados.

**@PostPersist**: Chamado imediatamente depois que o objeto é salvo na base de dados.

**@PreRemove**: Chamado imediatamente antes que o objeto seja removido da base de dados.

**@PostRemove**: Chamado imediatamente depois que o objeto é removido da base de dados.

**@PreUpdate**: Chamado imediatamente antes que o objeto seja atualizado.

**@PostUpdate**: Chamado imediatamente depois que o objeto é atualizado.

**Obs.:** As anotações para JPA Entity Listeners fazem parte do pacote **javax.persistence.**

**Código de Exemplo**

Para habilitar o uso de EntityListeners em uma entidade, apenas use a anotação **@EntityListners** na referida classe, através dela você listará as classes que ficarão observando as operações de persistência sobre a entidade.

@Entity

@EntityListeners({ProdutoListener.class, OutroListenerProduto.class})

public class Produto {

// ...

}

**Obs.:** Podemos listar várias classes separando-as por “,” (vírgula).

**Exemplo de um EntityListener**

public class ProdutoListener {

@PrePersist

public void prePersist(Produto produto) {

Date now = new Date();

produto.setDataCriacao(now);

produto.setDataAtualizacao(now);

}

@PreUpdate

public void preUpdate(Produto produto) {

produto.setDataAtualizacao(new Date());

}

}

No exemplo anterior, o **EntityListener** fica observando as operações de **persist** e **update** nos objetos da classe produto, e atribui a data da criação e atualização aos objetos antes que eles sejam salvos ou atualizados no banco.

**I18N (Internacionalization)**

Recurso utilizado para padronização de textos (rótulo, nomes, descrições) dentro da aplicação, bem como para tradução dos mesmos em outras linguagens.

A maioria dos frameworks web tem sua maneira particular de prover esse mecanismo, mas o que muita gente desconhece é que existe uma forma padrão de fazer isso, definida na especificação do **Java EE**, através da **JSP Standard TagLibs**.

**Definindo arquivos de i18n**

Esse arquivo de mensagens deve estar no *classpath* da sua aplicação web (por padrão é colocado no diretório raiz do projeto **/src**) e deve possuir a extensão **.properties** e compostos apenas de chaves e valores.

**Ex.:**

**Arquivo: messages.properties**

**Conteúdo:**

campo.preco = Informe o preço do produto

campo.nome = Informe o nome do produto

campo.quantidade = Informe a quantidade de produtos

**Configuração i18n com JSF**

Crie um arquivo chamado **faces-config.xml** dentro da pasta WEB-INF do projeto, com o seguinte conteúdo:

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<faces-config xmlns=*"http://xmlns.jcp.org/xml/ns/javaee"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig\_2\_2.xsd"*

version=*"2.2"*>

<application>

<locale-config>

<default-locale>pt\_BR</default-locale>

</locale-config>

<resource-bundle>

<base-name>messages</base-name>

<var>msg</var>

</resource-bundle>

</application>

</faces-config>

Onde:

* **resource-bundle:** identifica recursos que serão usados para internacionalização do sistema.
* **base-name:** indica onome do arquivo **.properties**.
* **var:** Identificador que será usado como referência nos arquivos **.xhtml** (view)

**Obs.:** É possível ter várias *tags* **resource-bundle**, uma para cada arquivo **.properties** do projeto.

**Utilizando i18n na *view***

<h:outputLabel value="#{msg['campo.nome']}" for="name" />

<p:inputText id="name" disabled="true" />

**Configuração i18n em *backend***

Podemos utilizar também internacionalização no *backend* do projeto, no caso, nas classes **java**. Para isso podemos criar uma classe responsável por gerenciar a leitura de mensagens no arquivo **.properties.**

public class MessageManager {

public static String readMessage(String key) {

public static String readMessage(String key) {

ResourceBundle bundle;

bundle = ResourceBundle.getBundle("mensagens", new Locale("pt\_br"));

return bundle.getString(key);

}

}

}

Perceba que o método **readMessage** é estático, portanto podemos invocá-lo sem a necessidade de instanciar um objeto do tipo **MessageManager.**

Supondo que desejamos lançar uma exceção qualquer no nosso código java, utilizando a classe **RuntimeException** (do pacote **java.lang**), que recebe uma *String* com o conteúdo da exceção, basta fazermos:

throw new RuntimeException (MessageManager.readMessage ( "error.ErroInterno");

Sendo que **error.ErroInterno** é a chave que deve estar contida no arquivo **.properties.**

**Referências**

**Livro de apoio**: <http://alga.works/livro-javaee/>

**JSF I18n:** <http://www.java2s.com/Tutorials/Java/JSF/3040__JSF_Internationalization.htm>

**JPA Entity Listeners:**

<https://docs.jboss.org/hibernate/entitymanager/3.6/reference/en/html/listeners.html>