TP JEE (1)
Développement Web avec JSF

Table des matières

1 Hello World

1.1 Introduction

JSF (JavaServer Faces) est un framework web écrit en Java dont l'objectif est de simplifier le développement de sites webs basés sur JEE. JSF est basé sur un modèle de développement par composants afin de réduire le couplage des parties "affichage" et codes métiers.

Les vues JSF appelées "facelets" définissent l'interface utilisateur. Elles sont écrites en XML et sont transformées par le serveur Java (ici JBoss) en fichiers HTML exploitables par le navigateur. Ces vues peuvent faire appel à des objets Java les "Backing Beans". Par exemple, le code source 1 montre comment appeler la méthode *sayHello* du "Backing Bean" *HelloBean* présenté sur le source 2.

Source 1 – Facelet

```
<?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">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">

    <h:head>
                <title>JSF 2.0 Hello World</title>
    </h:head>
    <h:body>
                <p>#{helloBean.sayHello()}</p>
    </h:body>
</html>
```

Les “Backing Beans” sont des classes java qui sont “exposées” au “Facelets” par un mécanisme appelé *injection de dépendance*. Pour cela on utilise l'annotation java @Model qui implémente les annotations @Named et @RequestScoped. L'annotation @Named permet de dire au moteur injection de dépendance que l'on pourra accéder à une classe Java par un nom ici le nom de la classe avec une minuscule au début. L'annotation @RequestScoped permet de spécifié que la durée de vie de l'objet est limitée à la requête (du client).

Source 2 – Backing Bean

```
@Model
public class HelloBean {

        public String sayHello() {
                return ``Hello World !'';
        }

}
```

1.2 Exercices :

Récupérer le projet squelette

1. Récupérez le projet Maven *ContactWeb*.

   ```
   hg clone https://bitbucket.org/jee_dutbx1/tp1-web
   ```

2. Copiez les dossier *skull-tp-war/ContactWeb* vers le dossier *tp1-web* de votre propre dépôt Mercurial.
3. Dans ContactWeb, générez un projet Eclipse et ouvrez-le.
4. Parcourez le projet et prenez vos marques.
5. Déployer un WAR du projet sur JBOSS et vérifiez que la facelet index.html affiche la chaine de caractère “Welcome !”.

Transformation de ContactController

1. Transformez la classe ContactController en Backing Bean pour l'exposer à la facelet.
2. Dans ContactController, créez une nouvelle méthode de type “HelloWorld” pour afficher le titre de la page. Appelez cette méthode depuis la vue JSF. Déployez et testez.

Tests d'intégration

Les tests d'intégration ont pour objectif de tester l'intégration de différents composants logiciels sur des architectures distribuées. Ici, nous voulons tester l'intégration de notre nouveau backing bean. Or, comme l'instance du backing bean est donnée par le serveur Java il est nécessaire d'utiliser l'injection de dépendance pour que l'instance du BackingBean soit injectée dans la classe de test. Pour cela il suffit d'utiliser l'annotation @Inject sur un attribut (ici un attribut de type ContactController).

1. Ajoutez un nouveau test d'intégration afin de tester votre nouvelle méthode.
2. Lancer les tests d'intégration avec Maven.

2 Exposer un attribut à une vue

2.1 Introduction

Notre classe ContactController va maintenant exposer à la vue un attribut de type Contact. Pour cela nous utilisons l'annotation Java @Produces sur, soit un attribut, soit une méthode (*getFormContact* par exemple). Bien entendu, il faudra aussi utiliser l'annotation @Named.

Il peut être parfois nécessaire de spécifier certaines choses lors de la construction d'un BackingBean (par exemple, initialiser des attributs, etc). Pour cela il est possible d'utiliser l'annotation @PostConstruct sur une méthode pour qu'elle soit automatiquement appelée après l'initialisation du bean.

2.2 Exercices

1. Dans ContactController créez un attribut de type Contact qui représentera le contact d'un *formulaire* de notre vue.
2. Ecrire les tests d'intégration correspondant. Assurez vous que le contact est initialisé.
3. Affichez les informations du contact *formContact* dans la facelet index.xhtml.
4. Déployez, testez.

3 Affichez une liste

3.1 Introduction

La balise h :dataTable permet de créer un tableau (table HTML) à partir d'une liste exposée à la facelet (voir extrait de code 3.1).

```
<h:dataTable value="#{contacts}" var="c" id="table" >
   <h:column>
   <!-- column header -->
         <f:facet name="header">Firstname</f:facet>
   <!-- row record -->
          #{c.firstname}
   </h:column>
</h:dataTable>
```

3.2 Exercices

1. Dans ContactController utilisez ContactService pour ajouter un ou deux contacts d'exemple.

2. Créez les tests d'intégration correspondants.
3. Affichez la liste de contacts dans la facelet.
4. Déployez, testez.

4 Un formulaire

4.1 Introduction

Comme le montre le code 4.1, la balise h :form permet de créer un formulaire qui sera traité lors du click sur le bouton d'identifiant "add" et exécutant la méthode *add* d'un BackingBean de type PersonController.

```
<h:form>
 <h:panelGrid columns="2">
   <h:outputLabel for="nom" value="Nom :" />
   <h:inputText id="nom" value="#{formPerson.nom}" />
  </h:panelGrid>

  <p>
    <h:commandButton id="add"
       action="#{personController.add}"
       value="Add" />
  </p>
</h:form>
```

4.2 Exercices

1. Dans ContactController créez une méthode permettant d'ajouter le contact *formContact* à la liste de contacts.
2. Créez les tests d'intégration correspondants.
3. Dans la facelet index.xhtml, afficher un formulaire permettant d'ajouter un contact.
4. Déployez, testez.
5. Réactualisez la page, pourquoi est ce que la liste affiche toujours un seul contact ?

1. Ajoutez à la classe ContactService l'annotation @ApplicationScoped afin que l'instance du bean soit valide pendant toute la durée de vie de l'application.
2. Modifiez la façon dont est instanciée la classe (dans ContactController et dans les tests) afin que le serveur d'application s'occupe du cycle de vie de l'objet).
3. Déployer et testez.

5 Validation des données

5.1 Introduction

Pour ajouter une contrainte sur un attribut il est possible d'ajouter des annotations qui seront validées lors de la soumission d'un formulaire. C'est annotations sont par exemple @Size pour valider par exemple la taille d'une chaine de caractère (voir code 5.1) ou encore @Email pour vérifier qu'une chaine de caractère est de la forme d'un email. Il est aussi nécessaire de préciser où seront afficher les messages d'erreurs dans la facelet en utilisant la balise h :message comme le montre les extraits de code suivant :

```
public class Person {
```

```
        @Size(min=8, max=300)
        private String nom;
        ....
}
```

```
<h:panelGrid columns="3" >
 <h:outputLabel for="nom" value="Nom :" />
 <h:inputText id="nom" value="#{formPerson.nom}" />
 <h:message for="nom" />
</h:panelGrid>
```

5.2 Exercices

1. Ajoutez les "bonnes" contraintes à la classe contact (par exemple : un numéro de téléphone valide en France, un nom plus grand que 10 lettres, etc).
2. Déployez, testez.
3. En vous inspirant de l'extrait de code suivant, qui permet de valider l'instance d'une classe Personne, créez une nouvelle classe de test d'intégration afin de tester chaque règle de validation.

```
@Inject
Validator validator;
void uneMethode() {
        Personne p = new Personne();
        Set<ConstraintViolation<Contact>> contraints = validator.
            validate(p);
        ....
}
```

4. Testez.
5. Modifiez la méthode ContactService.add(Contact c) afin de prendre en compte les règles de validation. Mettez les tests unitaires à jour.
6. Testez.

6 Navigation

6.1 Introduction

Il existe plusieurs façon de naviguer entre différente pages JSF. La première consiste à utiliser la balise h :commandButton ou h :link comme le montre les exemples suivant :

```
<h:form>
    <h:commandButton action="page2" value="Move to page2.xhtml" />
</h:form>

<h:link outcome="page2">Move to page2.xhtml</h:link>
```

Il peut être aussi utile de passer des paramètres à une vue JSF. Cela peut être réaliser de plusieurs façon, mais celle recommandé par la spécification utilise les balises f :param (du coté de l'appel : page1.xhtml) et f :viewParam du coté de la nouvelle vue à afficher. Voici un exemple :

```
// Dans page1.xhtml
<h:link outcome="page2" value="Move to page2">
        <f:param name="unParam" value="#{valeurDuParam}"/>
```

```
</h:link>
// Dans page2.xhtml
<f:metadata>
    <f:viewParam name="unParam" value="#{monBean.maPropriete}" />
</f:metadata>
```

Dans l'extrait de code précédant, la page2 "récupère" le paramètre appelé "unParam" qui lui est transférer et affecte l'attribut de la classe MonBean.maPropriete avec la valeur du paramètre. Il est donc nécessaire d'exposer la classe suivante à la vue JSF :

```
@Model
public class MonBean {

        @Produces
        @Named
        String maPropriete;


        @PostConstruct
        public void initContact() {
                maPropriete = new String();
        }

        public String getMaPropriete() {
                return maPropriete;
        }


        public void setMaPropriete(String p) {
                maPropriete = p;
        }
}
```

Pour finir, il peut aussi être utile d'initialiser certaines valeurs de la classe MonBean avant l'affichage de page2, mais après l'obtention du paramètre. Pour cela on utilisera la balise f :event pour déclencher une méthode du bean MonBean, lors de l'événement "preRenderView" comme le montre l'extrait de code suivant :

```
<f:metadata>
     <f:viewParam name="unParam" value="#{monBean.maPropriete}" />
     <f:event type="preRenderView" listener="#{monBean.init}" />
</f:metadata>
<ui:define name="content">
<p>Valeur de autrePropriete : #{monBean.autrePropriete}</p>
</ui:define>
```

Bien entendu la classe MonBean change :

```
@Model
public class MonBean {

        @Produces
        @Named
        String maPropriete;

        @Produces
        @Named
        String autrePropriete;
```

```
        @PostConstruct
        public void initContact() {
                maPropriete = new String();
                autrePropriete = new String();
        }

        // Appel lors du declenchement de l'evenement preRenderView
        public void init() {
                autrePropriete = "autre_"+maPropriete;
        }

        public String getMaPropriete() {
                return maPropriete;
        }
        public void setMaPropriete(String p) {
                maPropriete = p;
        }
        public String getAutrePropriete() {
                return autrePropriete;
        }
        public void setAutrePropriete(String p) {
                autrePropriete = p;
        }
}
```

6.2 Exercices

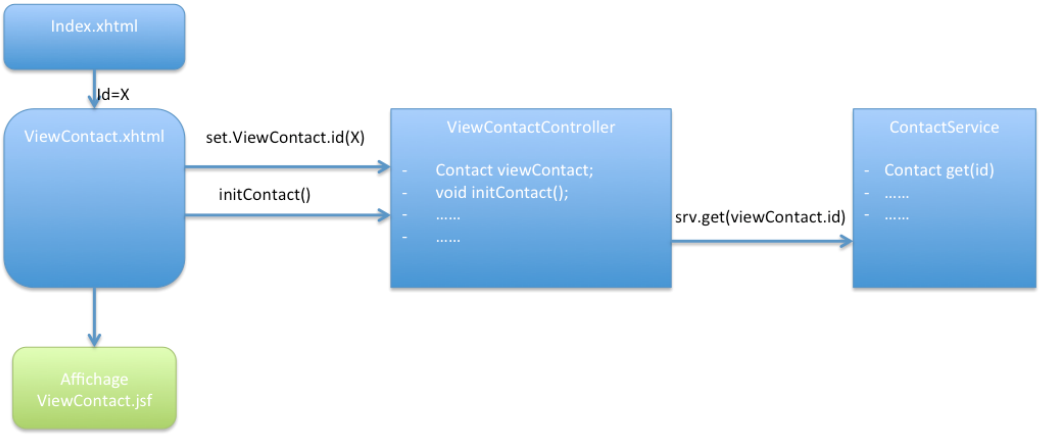

FIGURE 1 – Transfert du paramètre "id" pour afficher le détail d'un contact dans une nouvelle JSF.

1. Ajouter une nouvelle vue JSF permettant d'afficher le détail d'un contact sélectionné dans la liste des contacts. L'identifiant du contact à afficher sera transférer entre différents composants comme le montre la figure 1. Plusieurs étapes sont donc nécessaires :
   (a) Créez une page ViewContact.xhtml et une classe ViewContactController qui sera le BackingBean.
   (b) Pour chaque contact dans la liste des contacts (pour chaque ligne dans le tableau), ajoutez un lien vers la page ViewContact.xhtml.
   (c) Transferez un paramètre correspondant à l'id du contact à afficher à la page ViewContact.xhtml.

(d) Dans ViewContact.xhtml, récupérez l'id du contact passé en paramètre et affectez cette valeur à une propriété de ViewContactController.
(e) Dans ViewContact.xhtml faite en sorte de récupérer le contact à affichez (celui avec le bon id) avant que la vue JSF ne soit générée.