JEE :
(Eclipse) Jakarta EE

(anciennement ***Java 2 Enterprise Edition***,ou **J2EE**, puis ***Java Enterprise Edition*** ou **Java EE**),

FSTG Marrakech

BENHADDI Meriem

Gestion des événements JSF

- Nous avons déjà vu, qu'il existait la notion d'action JSF. Mais cette possibilité est plus liée à la gestion de la navigation. Si vous souhaitez juste exécuter un traitement côté serveur sans forcément rediriger vers une autre page, alors la notion de « **listeners JSF** » sera certainement plus adaptée.
- JSF propose deux manières de définir des listeners : Implémentation explicite/implémentation implicite.

Implémentation explicite des Listeners JSF

- Un listener consiste en une interface décrivant la (ou les) méthode(s) associée(s) à l'événement considéré. Chaque méthode de l'interface accepte un unique paramètre : l'objet d'événement contenant les informations qualifiant l'événement constaté.
- Il existe deux principales interfaces de listeners dans JSF :
  - **javax.faces.event.ActionListerner :** permet de définir un traitement en cas de clic sur un bouton (par exemple).
  - **javax.faces.event.ValueChangedListener** : permet de détecter un changement de valeur sur un champ de saisie entre deux allers/retours sur le serveur.

Implémentation explicite de l'interface *javax.faces.event.ActionListerner*

- Pour notre premier exemple, nous allons coder une nouvelle page Web. Cette page nous permettra de parcourir les articles proposés par notre site Web de vente en ligne.
- Dans un premier temps, nous allons juste gérer une donnée de type numérique qui correspondra à l'indice de l'article à afficher. Deux boutons nous permettront de passer à l'article précédent ou suivant : ce sont ces boutons qui nous intéressent pour notre gestion d'événements.

Notre bean

```
1 2  package webstore.ihm;
3 4  import java.io.Serializable;
5 6  import javax.enterprise.context.SessionScoped;
7 8  import javax.inject.Named;
  9
 10  @Named
 11  @SessionScoped
 12
 13  public class CatalogBrowserBean implements Serializable {
 14      private static final long serialVersionUID = 27297584327561082 74L;
 15      private int index;
 16      public int getIndex() {
 17          return index;
 18      }
 19       public void setIndex(int index) {
 20          this.index = index;
 21      }
 22  }
```

Notre vue

```
1  <!DOCTYPE html>
2  <html xmlns:f="http://xmlns.jcp.org/jsf/core"
3  xmlns:h="http://xmlns.jcp.org/jsf/html">
4      <f:view>
5          <head>
6              <title>View Article</title>
7              <link rel="stylesheet" type="text/css" href="../styles.css" />
8          </head>
9
10         <body>
11             <h1>View Article</h1>
12             <h:form>
13                 Identifiant : #{catalogBrowserBean.index} <br/>
14                     <!-- TODO: à finir ultérieurement -->
15                 <br/>
16                 <h:commandButton value="Précédent" />  
17                 <h:commandButton value="Suivant" />
18             </h:form>
19         </body>
20     </f:view>
21 </html>
```

- Cette page affiche l'indice de l'article ainsi que 2 boutons.

Implémenter une classe de listener associée au bouton « Suivant »

```
1 2  Package webstore.ihm;
3 4  import java.io.Serializable;
5 6  import javax.enterprise.context.SessionScoped;
7 8  import javax.faces.event.AbortProcessingException;
   9 import javax.faces.event.ActionEvent;
  10 import javax.faces.event.ActionListener;
  11 import javax.inject.Inject;
  12 import javax.inject.Named;
  13
  14 @Named
  15 @SessionScoped
  16 public class NextListener implements Serializable, ActionListener {
  17    private static final long serialVersionUID = -7752358388239085979L;
  18
  19    @Inject
  20    private CatalogBrowserBean catalogBrowserBean;
  21
  22    @Override
  23    public void processAction( ActionEvent event ) throws AbortProcessingException {
  24        catalogBrowserBean.setIndex( catalogBrowserBean.getIndex() + 1 );
  25     }
```

Explications

- On réalise une injection de dépendance via l'annotation @**Inject** : c'est le framework CDI (Context And Dependency Injection) qui se chargera de retrouver l'instance de la classe CatalogBrowserBean dans votre session utilisateur.
- Pour que CDI puisse correctement réaliser l'injection de dépendance, il faut absolument qu'il connaisse l’instance de la classe NextListener. C'est pour cela qu'on l'associe à CDI via l'annotation @**Named**.

- Maintenant il nous faut associer cette classe de listener avec le bouton : cela se fait directement dans la facelet viewArticle.xhtml en ajoutant un sous tag <f:actionListener /> dans le tag correspondant au champ de saisie.

```
1  <h:commandButton value="Suivant">
2       <f:actionListener binding="#{nextListener}" />
3  </h:commandButton>
```

Implémentation explicite de l'interface *javax.faces.ValueChangedListener*

```
1  Package webstore.ihm;
2  import javax.faces.event.AbortProcessingException;
3  import javax.faces.event.ValueChangeEvent;
4  import javax.faces.event.ValueChangeListener;
5
6  public class TextListener implements ValueChangeListener {
7      @Override
8       public void processValueChange(ValueChangeEvent arg0) throws
9  AbortProcessingException {
1            System.out.println( "Value changed" );
0       }
1  }
```

Implémentation implicite de listeners

- Une autre solution, certainement plus simple, consiste à laisser JSF produire le listener. Celui-ci aura pour responsabilité de rappeler une méthode particulière sur votre bean. La méthode en question doit respecter une signature bien précise.
- Voici un exemple de définition de deux gestionnaires d'événements (pour les deux boutons « Précédent » et « Suivant ») en utilisant cette technique.
-

```
1  package webstore.ihm;
2  import java.io.Serializable;
3  import javax.enterprise.context.SessionScoped;
4  import javax.faces.event.ActionEvent;
5  import javax.inject.Named;
6
7  @Named
8  @SessionScoped
9  public class CatalogBrowserBean implements Serializable {
10     private static final long serialVersionUID = 27297584327561082 74L;
11     private int index;
12
13     public int getIndex() {
14         return index;
15      }
16      public void setIndex(int index) {
17         this.index = index;
18      }
19      public void processPreviousAction( ActionEvent event ) {
20         index--;
21      }
22      public void processNextAction( ActionEvent event ) {
23         index++;
24       }
25 }
```

- Pour lier ces méthodes à vos boutons, il faut, pour chaque bouton, ajouter un attribut actionListener.

```
1  <h:commandButton value="Précédent"
2                   actionListener="#{catalogBrowserBean.processPreviousAction}"
   />
```

Code complet de la vue(facelet)

```
1 2 <!DOCTYPE html>
3 4 <html xmlns:f="http://xmlns.jcp.org/jsf/core"
5 6 xmlns:h="http://xmlns.jcp.org/jsf/html">
7 8    <f:view>
 9      <head>
10        <title>View Article</title>
11        <link rel="stylesheet" type="text/css" href="../styles.css" />
12      </head>
13      <body>
14        <h1>View Article</h1>
15        <h:form>
16           Identifiant : #{catalogBrowserBean.index} <br/>
17             <!-- TODO: à finir -->
18           <br/>
19           <h:commandButton value="Précédent"
20 actionListener="#{catalogBrowserBean.processPreviousAction}" />  
21           <h:commandButton value="Suivant"
22 actionListener="#{catalogBrowserBean.processNextAction}" />
23        </h:form>
24      </body>
25    </f:view>
26 </html>
```

Quelle technique privilégier ?

- L’implémentation implicite est bien plus simple à utiliser. Il est recommandé de l'utiliser. Toutes fois, si vous avez un code complexe et relativement long pour votre gestionnaire d'événement, le fait d'utiliser l’implémentation explicite vous permettra d'isoler ce bloc de code dans une classe autonome.

Liaison aux données dans vos formulaires JSF

- Nous allons poursuivre le codage de la page Web proposée dans le chapitre précédent. Elle permettra de parcourir les articles d'un catalogue de vente en ligne et de stocker certains de ces articles dans un panier. Par la suite nous afficherons le contenu du panier. La capture d'écran ci-dessous vous montre à quoi va devoir ressembler la première page Web à développer.

Modèle de données

- Notre première classe se nomme Catalog et référence tous les articles proposés dans notre catalogue.

```
1  Package webstore.business;
2  import java.util.ArrayList;
3  import java.util.List;
4  import javax.enterprise.context.ApplicationScoped;
5  import javax.inject.Named;
6  @Named
7  @ApplicationScoped
8  public class Catalog {
9    private List<Article> articles = new ArrayList<>();
10   public Catalog() {
11     articles.add( new Article( 1, "Drone", "Perroquet", 400 ) );
12     articles.add( new Article( 2, "Télévision", "SuperBrand", 350 ) );
13     articles.add( new Article( 3, "Souris", "Mulot", 35 ) );
14     articles.add( new Article( 4, "Smartphone", "MegaMark", 750 ) );
15     articles.add( new Article( 5, "Vacances", "DeRêve", 15_000 ) );
16   }
17   public List<Article> getArticles() {
18     return articles;
19   }
20   public int getSize() {
21     return articles.size();
22  }
23 }
```

Catalogue unique et partagé par tous les utilisateurs

```
1  Package webstore.business;
2
3  public class Article {
4     private int idArticle;
5    private String description;
6    private String brand;
7    private double price;
8    public Article() {
9      this( 1, "unknown", "unknown", 0 );
10   }
11   public Article( int idArticle, String description, String brand, double price ) {
12     this.setIdArticle( idArticle );
13     this.setDescription( description );
14     this.setBrand( brand );
15     this.setPrice( price );
16   }
17   public int getIdArticle() {
18     return idArticle;
19   }
20   public void setIdArticle(int idArticle) {
21     this.idArticle = idArticle;
22   }
23   public String getDescription() {
24      return description;
25   }
26   public void setDescription(String description) {
27      this.description = description;
28   }
29    public String getBrand() {
30       return brand;
31   }
```

Suite

```
32 public void setBrand(String brand) {
33     this.brand = brand;
34   }
35   public double getPrice() {
36     return price;
37   }
38   public void setPrice(double price) {
39      this.price = price;
40   }
41   @Override public String toString() {
42      return "Article [idArticle=" + idArticle + ", description=" + description + ",
43 brand=" + brand + ", price=" + price + "]";
44   }
45 }
```

La classe Batch représente un lot d'articles

```
1 2  Package webstore.business;
3 4  import java.security.InvalidParameterException;
5 6
7 8  public class Batch {
  9    private Article article;
 10    private int quantity;
 11    public Batch( Article article, int quantity ) {
 12      if ( article == null )
 13          throw new NullPointerException( "article cannot be null" );
 14      if ( quantity < 1 )
 15           throw new InvalidParameterException( "quantity must be a positive number" );
 16      this.article = article;
 17      this.quantity = quantity;
 18    }
 19    public Article getArticle() {
 20      return article;
 21    }
 22    public int getQuantity() {
 23      return quantity;
 24    }
 25    public void addOne() {
 26      quantity++;
 27    }
 28 }
```

Notre Bean :

```
1 2  Package webstore.ihm;
3 4  import java.io.Serializable;
5 6  import java.util.ArrayList;
7 8  import java.util.List;
9 10 import javax.enterprise.context.SessionScoped;
11   import javax.faces.event.ActionEvent;
12   import javax.inject.Inject;
13   import javax.inject.Named;
14   import webstore.business.Article;
15   Import webstore.business.Batch;
16   Import webstore.business.Catalog;
17
18   @Named
19   @SessionScoped
20   public class CatalogBrowserBean implements Serializable {
21     private static final long serialVersionUID = 27297584327561082 74L;
22
23      @Inject
24     private Catalog catalog;
25     private List<Batch> basket = new ArrayList<>();
26     private int index;
27     public Article getCurrentArticle() {
28        return catalog.getArticles().get( index );
29     }
30     public List<Batch> getBasket() {
31       return basket;
32   }
```

L'instance de Catalog est injéctée via CDI dans le Bean.

La méthode getCurrentArticle utilise l'attribut index pour retrouver l'article à afficher dans le catalogue

La méthode getBasket nous servira pour afficher l'intégralité des articles stockés dans le panier.

Suite :

```
33      public int getBasketSize() {
34          int quantity = 0;
35          for( Batch batch : basket ) {
36              quantity += batch.getQuantity();
37          }
38          return quantity;
39  }
40      // --- Event handler methods ---
41  public void processPreviousAction( ActionEvent event ) {
42      if ( --index < 0 ) {
43          index = catalog.getSize()-1;
44      }
45  }
46  public void processNextAction( ActionEvent event ) {
47      if ( ++index >= catalog.getSize() ) {
48          index = 0;
49      }
50  }
51  public void processAddAction( ActionEvent event ) {
52      for( Batch batch : basket ) {
53          if ( batch.getArticle().getIdArticle() == getCurrentArticle().getIdArticle() ) {
54              batch.addOne();
55              return;
56          }
57      }
58      basket.add( new Batch( getCurrentArticle(), 1 ) );
59    }
```

la méthode getBasketSize renvoie le nombre d'articles, en tenant compte des quantités de chaque lot, stockés dans le panier.

3 gestionnaires d'événements

La liaison aux données dans la vue

```
<!DOCTYPE html>
<html xmlns:f="http://xmlns.jcp.org/jsf/core"
xmlns:h="http://xmlns.jcp.org/jsf/html">
   <f:view>
      <head>
        <title>View Article</title>
        <link rel="stylesheet" type="text/css" href="styles.css" />
      </head>
      <body>
        <h1>View Article</h1>
        <h:form>
          Identifiant : #{catalogBrowserBean.currentArticle.idArticle} <br/>
          Description : #{catalogBrowserBean.currentArticle.description} <br/>
          Marque : #{catalogBrowserBean.currentArticle.brand} <br/>
          Prix : #{catalogBrowserBean.currentArticle.price} <br/> <br/>
           <h:commandButton value="Précédent"
                actionListener="#{catalogBrowserBean.processPreviousAction}" /> #1
           <h:commandButton value="Ajouter au panier"
                actionListener="#{catalogBrowserBean.processAddAction}" />  
           <h:commandButton value="Suivant"
                actionListener="#{catalogBrowserBean.processNextAction}" />
        <br/>
          Vous avez #{catalogBrowserBean.basketSize} article(s) dans votre panier.
        <br/>
        <a href="summary.xhtml">Voir le contenu du panier</a>
        </h:form>
       </body></f:view></html>
```

Accès à un élément d'un tableau

- Il est possible d'accéder à un élément particulier d'une collection par son indice:

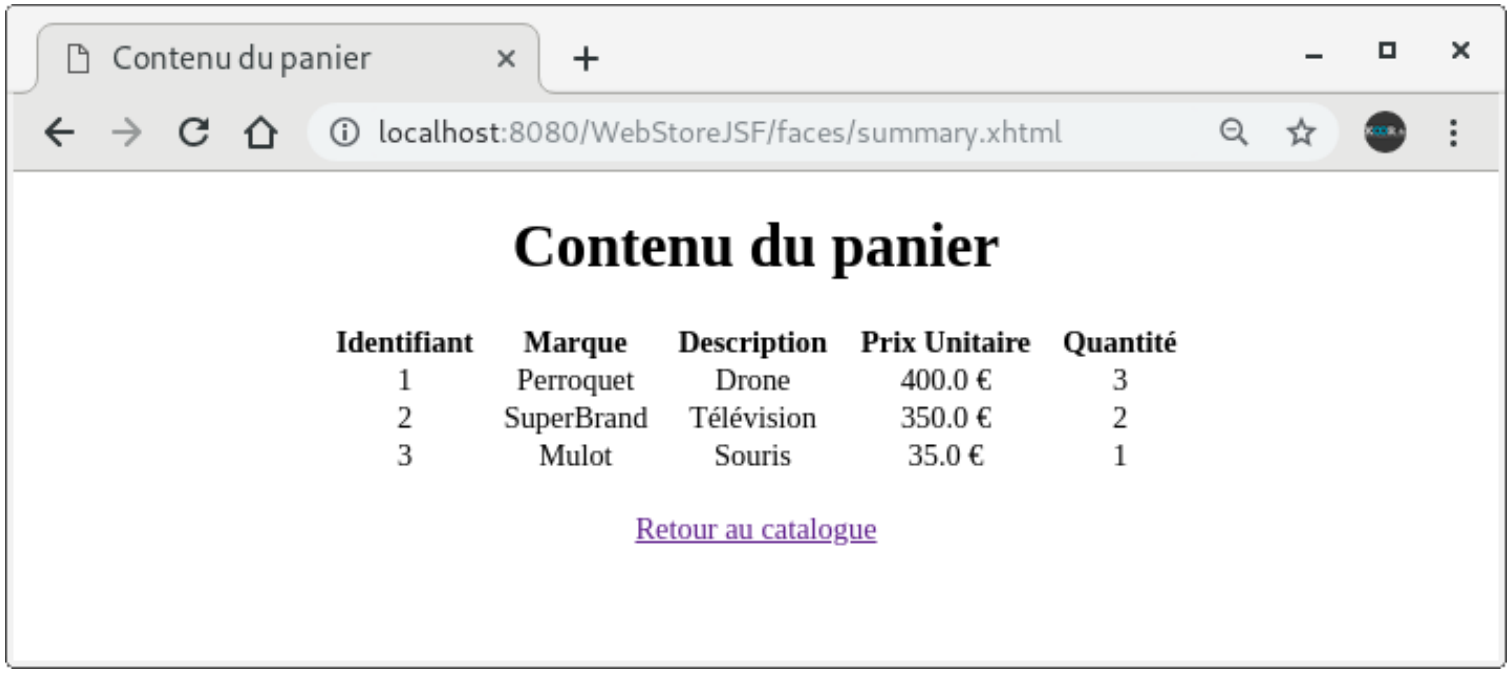
```
1  <div>
2     Prix du premier article du panier :
3     #{catalogBrowserBean.basket[0].article.price} euros
4  </div>
```

Utilisation du composant <h:dataTable />

- Nous allons maintenant mettre en oeuvre une nouvelle page JSF dont l'objectif est de présenter l'ensemble des articles sélectionnés dans le panier. Le panier correspond à l'attribut basket du «bean » nommé catalogBrowserBean et il est de type List<Batch>.
- Pour lier une collection à votre page web vous pouvez utiliser le composant <h:dataTable />.
- Le composant <h:dataTable /> va produire un tableau avec une ligne de titre et autant de ligne de données que d'éléments dans la collection considérée. Dans notre cas, la collection correspondra à notre panier : chaque élément de la collection sera une instance de la classe Batch. Un lot d'articles (batch en anglais) étant associé à un article, lui-même constitué de quatre attributs, et à une quantité, notre tableau aura donc 5 colonnes : identifiant de l'article, description, marque, prix et quantité.

```
<!DOCTYPE html>
<html xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <f:view>
    <head>
      <title>Contenu du panier</title>
    </head>
    <body>
      <h1 align="center">Contenu du panier</h1>
      <h:dataTable value="#{catalogBrowserBean.basket}" var="batch" style="width:
60%; margin: auto;">
        <h:column>
           <f:facet name="header">Identifiant</f:facet>
           #{batch.article.idArticle}
        </h:column>
        <h:column>
            <f:facet name="header">Marque</f:facet> #{batch.article.brand}
        </h:column>
        <h:column>
            <f:facet name="header">Description</f:facet>
            #{batch.article.description}
        </h:column>
        <h:column>
            <f:facet name="header">Prix Unitaire</f:facet> #{batch.article.price}
&#8364;
        </h:column>
        <h:column>
            <f:facet name="header">Quantité</f:facet> #{batch.quantity}
        </h:column>
      </h:dataTable> <br/>
      <a href="viewArticle.xhtml">Retour au catalogue</a>
</body> </f:view> </html>
```

Résultat produit par cette page:



Internationalisation des pages JSF

- L'internationalisation d'un site web et de ses pages consiste en l'ajout de code permettant son affichage dans de multiples langues et en tenant compte de la localisation géographique de son utilisateur.
- Le Java SE propose déjà un certain nombre de possibilité en termes d'internationalisation : ces possibilités sont principalement localisées dans les packages java.util et java.text. JSF complète ces mécanismes avec des solutions adaptées à la mise en oeuvre de vos facelets.

1- **Activation de l'internationalisation dans votre application JSF**

- La première étape consiste à activer l'internationalisation au sein du fichier de configuration JSF : le fichier WEB-INF/faces-config.xml.

- On doit notamment y lister l'ensemble de langues (voir des localisations) supportées par l'application ainsi que la configuration par défaut.

Fichier WEB-INF/faces-config.xml :

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <faces-config xmlns="http://xmlns.jcp.org/xml/ns/javaee"
3                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4                xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
5  http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_3.xsd" version="2.3">
6     <navigation-rule>
7       <from-view-id>/login.xhtml</from-view-id>
8       <navigation-case>
9          <from-outcome>success</from-outcome>
10         <to-view-id>/viewArticle.xhtml</to-view-id>
11      </navigation-case>
12    </navigation-rule>
13
14    <application>
15      <locale-config>
16         <default-locale>en</default-locale>
17         <supported-locale>fr</supported-locale>
18         <supported-locale>en</supported-locale>
19      </locale-config>
20    </application>
21
22 </faces-config>
```

2- Définir des traductions par localisation

- Cette seconde étape consiste à définir des fichiers contenant les traductions des chaînes de caractères sur site. Il y aura au moins un fichier par langue. Ces fichiers sont au format **.properties** : chaque ligne est une association type clé = valeur. Vous pouvez ajouter des commentaires dans ces fichiers et ils devront commencer par un caractère #.
- Ces fichiers sont à placer directement dans le dossier src du projet.

- A titre d'exemple, nous cherchons à internationaliser la page de connexion à notre site de vente en ligne (facelet login.xhtml). Voici un exemple d'un fichier pour la langue française :
  login_fr.properties

Ce fichier contient les chaînes de caractères de l'écran login en français.

title=Ecran de connexion
loginLabel=Identifiant :
loginValidatorMessage=C'est votre email

passwordLabel=Mot de passe:
passwordValidatorMessage=Votre mot de passe doit contenir entre 3 à 12 caractères

connectLabel=Se connecter
cancelLabel=Annuler

Celui en anglais : login_en.properties

This file contains strings for the english login screen

title=Login screen
loginLabel=Login:
loginValidatorMessage=It's your mail

passwordLabel=Password:
passwordValidatorMessage=3 to 12 characters are required

connectLabel=Connect
cancelLabel=Cancel

Injection des chaînes internationalisées dans la facelet

- On doit :
  - charger le fichier de ressources pour la langue adaptée au navigateur, en utilisant un tag **<f:loadBundle />**
  - injecter les chaînes de caractères contenues dans le fichier de ressources dans votre page Web

```
1  <!DOCTYPE html>
2  <html xmlns:f="http://xmlns.jcp.org/jsf/core"
3  xmlns:h="http://xmlns.jcp.org/jsf/html">
4    <f:view>
5      <head>
6        <title>#{bundle.title}</title>
7        <link rel='stylesheet' type='text/css' href='../styles.css' />
8        <f:loadBundle basename="login" var="bundle" />
9      </head>
10     <body>
11       <h1>#{bundle.title}</h1>
12       <h:form id="form">
13          #{bundle.loginLabel}
14          <h:inputText id="login" value="#{loginBean.login}"
15 validatorMessage="#{bundle.loginValidatorMessage}" />  
16          <h:message for="login" styleClass="errorBlock" />
17          <br/>
18
19          #{bundle.passwordLabel}
20          <h:inputSecret id="password" value="#{loginBean.password}"
21 validatorMessage="#{bundle.passwordValidatorMessage}" />  
22          <h:message for="password" styleClass="errorBlock" /> <br/><br/>
23          <h:commandButton action="#{loginBean.returnAction}"
24 value="#{bundle.connectLabel}" />
25          <h:commandButton value="#{bundle.cancelLabel}" immediate="true" />
26       </h:form>
27     </body>
28   </f:view>
29 </html>
```

Charger le bundle avec loadbundle

Le nom de chaque chaîne correspond à celui spécifié dans le fichier de ressources

Internationalisation des pages JSF

- **Testez en changant les préférences de votre navigateur en termes de localisation.**

Définition d'un template pour vos facelets

- Il est fréquent que plusieurs pages de votre application Web utilisent une même mise en page (même entête, même bloc de menu, même pied de page...). Il serait dommage, que sur chacune de ces pages, vous recopiez les mêmes blocs de tags. Si un jour vous souhaitez changer la mise en page pour l'ensemble de ces pages, vous devrez alors modifier chaque fichier HTML.
- Pour répondre à ce besoin, JSF vous permet de définir un (ou plusieurs) template(s). Un template correspond à un modèle de page Web. Vous pourrez alors composer chacune de vos pages (vos facelets) à partir de ses différentes parties qui seront injectées dans le template associé.

1- **Définition d'un template de facelet**

- Plusieurs zones à l'intérieur du template devront pouvoir changer (le titre de la page, le contenu de la page...). Il faut donc définir, dans le template, des zones « dynamiques » : celles qui seront spécifiques à chaque page Web.
- Cela se fait grâce au tag **<ui:insert name="areaName">** : chacun de ces tags sera associé à un nom qui permettra de réaliser le lien entre le contenu d'une page et la zone dans laquelle il devra être inséré.
- Dans notre exemple, nous allons utiliser quatre tags <ui:insert> pour rendre dynamique le titre de la page (dans le head et dans le body), la partie gauche de la zone centrale de la page et sa partie droite.
- Ils ont respectivement les noms title, title, aSide et content.

```
<!DOCTYPE html >
<html xmlns:ui="http://java.sun.com/jsf/facelets"
xmlns:h="http://java.sun.com/jsf/html">
   <h:head>
     <meta charset="UTF-8" />
     <link rel="stylesheet" type="text/css"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" />
     <title><ui:insert name="title">The page title</ui:insert></title>
   </h:head>
   <h:body>
      <header class="jumbotron text-center">
        <div class="container">
          <div class="row">
            <a href="faces/index.xhtml" class="col-md-2">
              <img src="images/logo.png" alt="Site logo" title="Retour à la page
principale" />
            </a>
            <h1 class="col-md-10 text-center">
              <ui:insert name="title">The page title</ui:insert>
            </h1>
          </div>
         </div>
      </header>
      <div class="container">
        <div class="row">
           <nav class="navbar col-md-12" style="margin-bottom: 30px;">
             <a href="index.xhtml">Accueil</a>
             <a href="product1.xhtml">Produit 1</a>
             <a href="product2.xhtml">Produit 2</a>
             <a href="product3.xhtml">Produit 3</a>
             <a href="contact.xhtml">Contact</a>
</nav> </div>
```

```
33      <div class="row">
34       <div class="col-md-4">
35         <ui:insert name="aSide">Side content</ui:insert>
36       </div>
37       <div class="col-md-8">
38         <ui:insert name="content">Main Content</ui:insert>
39       </div>
40      </div>
41     </div>
42     <br/><br/>
43     <footer class="jumbotron text-center">
44      <h2 style="margin-bottom: 20px;"> Ici, nous avons un pied de page<br/>avec
45des liens quelconques.
46      </h2>
47      <div class="container">
48       <div class="row">
49        <div class="col-md-4">
50         <a href="#">Link 1</a><br/>
51         <a href="#">Link 2</a><br/>
52         <a href="#">Link 3</a>
53        </div>
54        <div class="col-md-4">
55         <a href="#">Link 4</a><br/>
56         <a href="#">Link 5</a><br/>
57         <a href="#">Link 6</a>
58        </div>
59        <div class="col-md-4">
60         <a href="#">Link 7</a><br/>
61         <a href="#">Link 8</a><br/>
62         <a href="#">Link 9</a>
63        </div> </div> </div> </footer> </h:body> </html>
```

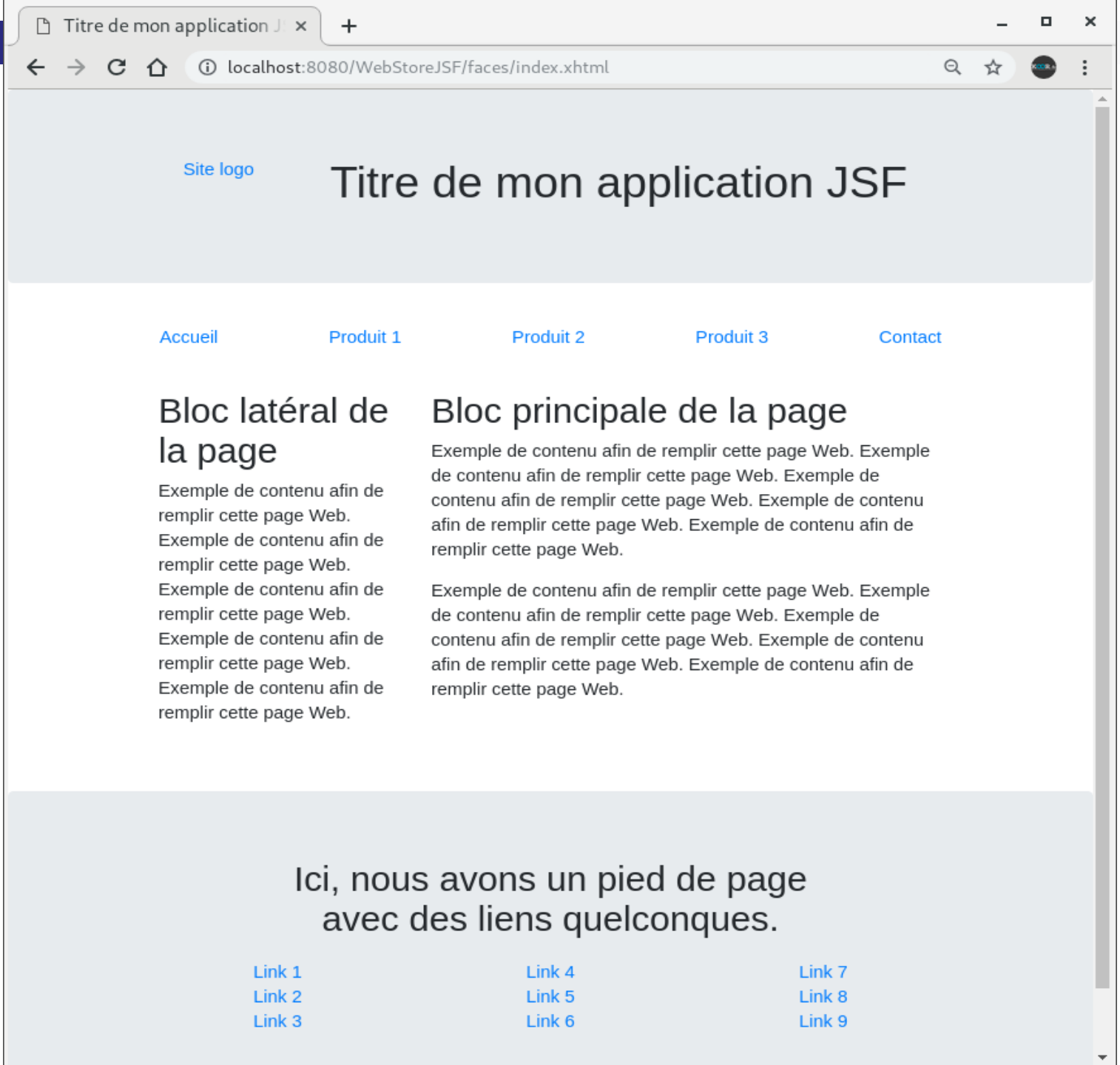
2- Coder une facelet JSF basée sur notre template

- Pour indiquer qu'une facelet JSF utilise un template, nous allons utiliser le tag **<ui:composition>** : il devra indiquer le nom du template à utiliser.

- Il nous faut aussi définir les contenus à injecter dans chaque zone du template. Cela se fait grâce au tag JSF **<ui:define name="areaName' />.** Les noms de blocs doivent correspondre aux noms des zones du template.

Code de la facelet :

```
1  <ui:composition template="./template.xhtml"
2                  xmlns:ui="http://java.sun.com/jsf/facelets"
3                  xmlns:h="http://java.sun.com/jsf/html">
4     <ui:define name="title"> Titre de mon application JSF </ui:define>
5     <ui:define name="aSide">
6       <h2>Bloc latéral de la page</h2>
7       <p>
8          Exemple de contenu afin de remplir cette page Web.
9          Exemple de contenu afin de remplir cette page Web.
10         Exemple de contenu afin de remplir cette page Web.
11 </p>
12    </ui:define>
13    <ui:define name="content">
14      <!-- Vous pouvez bien entendu utiliser des tags JSF -->
15      <h2><h:outputText value="Bloc principale de la page"/></h2>
16      <p>
17         Exemple de contenu afin de remplir cette page Web.
18         Exemple de contenu afin de remplir cette page Web.
19         Exemple de contenu afin de remplir cette page Web.
20 </p>
21      <p>
22         Exemple de contenu afin de remplir cette page Web.
23         Exemple de contenu afin de remplir cette page Web.
24         Exemple de contenu afin de remplir cette page Web.
25 </p>
26    </ui:define>
27 </ui:composition>
```


Titre de mon application J
localhost:8080/WebStoreJSF/faces/index.xhtml
Site logo
Titre de mon application JSF
Accueil
Produit 1
Produit 2
Produit 3
Contact
Bloc latéral de la page
Exemple de contenu afin de remplir cette page Web. Exemple de contenu afin de remplir cette page Web. Exemple de contenu afin de remplir cette page Web. Exemple de contenu afin de remplir cette page Web. Exemple de contenu afin de remplir cette page Web. Exemple de contenu afin de remplir cette page Web.
Bloc principale de la page
Exemple de contenu afin de remplir cette page Web. Exemple de contenu afin de remplir cette page Web. Exemple de contenu afin de remplir cette page Web. Exemple de contenu afin de remplir cette page Web. Exemple de contenu afin de remplir cette page Web.
Exemple de contenu afin de remplir cette page Web. Exemple de contenu afin de remplir cette page Web. Exemple de contenu afin de remplir cette page Web. Exemple de contenu afin de remplir cette page Web. Exemple de contenu afin de remplir cette page Web.
Ici, nous avons un pied de page avec des liens quelconques.
Link 1
Link 2
Link 3
Link 4
Link 5
Link 6
Link 7
Link 8
Link 9