

Création d'une application de type CRUD avec JSF et JPA

par Jawher Moussa ([Accueil](#))

Date de publication : 03/03/2008

Dernière mise à jour : 24/09/2009


Cet article a pour objectif de vous présenter la tâche de création d'une application complète de type *CRUD* (*Create, Read, Update, Delete*) permettant d'effectuer les opérations de création, consultation, suppression et modification d'une entité et ce en utilisant le *framework* web **JSF** et le *framework* de persistance **JPA** .

I - Introduction.....	3
II - Squelette de l'application.....	4
II-A - Configuration de JSF.....	4
II-B - Configuration de JPA.....	5
II-C - Couche métier.....	5
II-C-1 - Entité.....	6
II-C-2 - DAO.....	7
II-D - Couche contrôle.....	8
III - Implémenter l'opération Read.....	9
III-A - Dans la couche DAO.....	9
III-B - Dans la couche Control.....	9
III-C - Dans la couche View.....	9
III-D - Test.....	11
IV - Implémenter l'opération Create.....	12
IV-A - Dans la couche DAO.....	12
IV-B - Dans la couche Control.....	12
IV-C - Dans la couche View.....	13
IV-D - Test.....	14
V - Implémenter l'opération Delete.....	17
V-A - Dans la couche DAO.....	17
V-B - Dans la couche Control.....	17
V-C - Dans la couche View.....	18
V-D - Test.....	18
VI - Implémenter l'opération Update.....	20
VI-A - Dans la couche DAO.....	20
VI-B - Dans la couche Control.....	20
VI-C - Dans la couche View.....	21
VI-D - Test.....	22
VII - Télécharger.....	25
VIII - Conclusion.....	26
IX - Remerciements.....	27

I - Introduction



Une application *CRUD* permet d'effectuer les opérations de listing, ajout, modification et suppression sur une entité donnée. Ce cas d'utilisation est si fréquent dans le développement logiciel qu'il est rare de trouver une application qui ne fasse pas du *CRUD*.

La mise en place d'une telle application nécessite de pouvoir effectuer les opérations *CRUD* sur une source de données (Base de données, fichier plat, etc.) et de fournir une interface graphique (client lourd ou web, etc.) pour réaliser ces opérations.

Le but de cet article est donc de présenter et expliquer la création d'une application web Java permettant de faire du *CRUD* sur une seule entité en utilisant  **JSF** comme framework de présentation et **JPA** comme framework de persistance.

II - Squelette de l'application





Il s'agit d'une application web Java. Si vous utilisez un  **EDI** tel qu' *Eclipse* ou *NetBeans* , il est possible de générer automatiquement la structure d'une application web JSF grâce aux assistants de ces *EDIs* .

  **Eclipse** prend en charge le développement d'applications JSF via le pack *Web Tools Project 2* .

Les étapes de configuration d'Eclipse et de la création d'un projet **JSF** sont présentées sous forme de *démo Flash* dans [cet article](#) .

II-A - Configuration de JSF

Ceci revient à:

- Ajouter une implémentation **JSF** au  **classpath** . Dans l'exemple fourni avec cet article, J'utilise la *Sun Reference Implementation 1.2.6*  [téléchargeable ici](#) . Cette version supporte **JSF 1.2**.
- Déclarer la *Faces Servlet* dans *web.xml* et l'associer à ***.jsf** . Cette  **servlet** joue le rôle de la *Front Servlet* du  **MVC/Model 2** .
- Ajouter le fichier **faces-config.xml** qui permet de configurer l'application **JSF** (déclaration des *managed-beans / controllers* , déclaration des règles de navigation, etc.)

Après avoir déclaré la *Faces Servlet* ainsi que son affectation aux urls de type ***.jsf** , voici ce que devient *web.xml*:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <display-name>jsf-crud</display-name>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
  </servlet-mapping>
</web-app>
```

Et voici le fichier *faces-config.xml* (vide pour l'instant, mais il sera mis à jour au fur et à mesure de l'ajout de nouvelles fonctionnalités à l'application):


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

<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">

</faces-config>
```

II-B - Configuration de JPA

La configuration de **JPA** consiste à:

- Ajouter une implémentation **JPA** au classpath. Dans le cadre de cet article, j'utilise *Toplink 2.41* comme implémentation qui est  [téléchargeable ici](#) .
- Créer le descripteur de persistance qui sert à spécifier les paramètres de connexion à la base de données (url de connexion, login, mot de passe, etc.) ainsi qu'à la déclaration des classes persistantes.


J'utilise Toplink comme implémentation **JPA** et **HSQldb** comme base de données. Il faut donc mettre **toplink.jar** et **hsqldb.jar** dans le *classpath* de l'application.

Il faut ensuite créer un descripteur de persistance **JPA** (*persistence.xml*) dans un dossier *META-INF* à la racine du dossier source.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">

  <persistence-unit name="jsf-crud">
    <properties>
      <property name="toplink.logging.level" value="INFO" />
      <property name="toplink.target-database" value="HSQL" />
      <property name="toplink.jdbc.driver"
        value="org.hsqldb.jdbcDriver" />
      <property name="toplink.jdbc.url"
        value="jdbc:hsqldb:file:test" />
      <property name="toplink.jdbc.user" value="sa" />
      <property name="toplink.ddl-generation"
        value="create-tables" />
    </properties>
  </persistence-unit>
</persistence>
```

Pour l'instant, ce fichier ne contient que le nom de l'unité de persistance (**jsf-crud**) ainsi que les paramètres de connexion à la base de données qui sont:

- Pilote  **JDBC** : "org.hsqldb.jdbcDriver"
- **URL** de connexion: " **jdbc:hsqldb:file:test** " qui indique à **HSQldb** de stocker la base de données dans un fichier nommé " **test** ". Il est aussi possible de stocker la base dans la mémoire vive (**RAM**) en utilisant un chemin du type " **jdbc:hsqldb:mem:test** ". Mais les données seraient alors perdues à l'arrêt de l'application.
- Login: "sa" (l'équivalent du root dans **HSQldb**)
- Pilote **JDBC** : " **org.hsqldb.jdbcDriver** "
- **toplink.ddl-generation** : " **create-tables** " indique à *Toplink* de créer les tables si elles n'existent pas.

II-C - Couche métier

L'application qu'on désire développer permet de faire les opérations de création, modification, suppression et listing sur une seule entité. Normalement, la partie Model de l'application devrait être séparée en deux sous parties: **DAO** (*Data Access Object*) pour l'accès aux données et Service pour faire la logique métier. Mais dans le cadre de cet article, on se limitera à la couche **DAO** sans passer par la couche service vu que l'on n'a pas de logique métier à proprement parler.

II-C-1 - Entité

L'entité sur laquelle on va travailler représente une personne. On se contentera de deux attributs, nom et prénom. Voici son code:

```
package model.dto;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

/**
 * Cette classe représente une personne. C'est une entité persistente vu qu'on
 * l'a annoté avec l'annotation Entity.
 *
 * @author <a href="mailto:djo.mos.contact@gmail.com">djo.mos</a>
 *
 */
@Entity
public class Person {
    private Long id;
    private String firstName;
    private String lastName;

    /**
     * C'est l'accessor de l'identifiant.<br />
     * On indique qu'un champ est un identifiant en l'annotant avec Id. <br />
     * De plus, si on ajoute GeneratedValue, alors c'est la base de données ou
     * l'implémentation JPA qui se charge d'affecter un identifiant unique.
     *
     * @return l'identifiant.
     */
    @Id
    @GeneratedValue
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }


    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Comme vous le remarquez, cette classe est annotée avec **JPA** pour pouvoir être persistée ou retrouvé dans la base de données:

- L'annotation **@Entity** sur la classe **Person** pour la déclarer comme classe persistante. Cette annotation est obligatoire pour une classe persistante.
- L'annotation **@Id** sur le getter du champ id pour le déclarer comme l'identifiant. Cette annotation est obligatoire pour une classe persistante.

- L'annotation **@GeneratedValue** sur le getter du champ id pour déclarer qu'elle est auto-générée.

 *Notez que les champs `firstName` et `lastName` ne sont pas annotés. Ils seront pourtant persistés car dans un souci de simplification, **JPA** considère que tout champ d'une classe persistante est implicitement persistant, à moins qu'il soit annoté avec **@Transient**.*

Il faut ensuite déclarer cette classe dans le descripteur de persistance `persistence.xml` pour qu'elle soit prise en charge:

```
<class>model.dto.Person</class>
```

II-C-2 - DAO

On va maintenant encapsuler les opérations de persistance (création, modification, suppression et lecture) sur l'entité **Person** dans un **DAO**.

```
package model.dao;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.Persistence;

import model.dto.Person;


/**
 * C'est le DAO qui permet d'effectuer des opérations portant sur une personne
 * dans la base de données.
 *
 * @author <a href="mailto:djo.mos.contact@gmail.com">djo.mos</a>
 */
public class PersonDao {
    private static final String JPA_UNIT_NAME = "jsf-crud";
    private EntityManager entityManager;

    protected EntityManager getEntityManager() {
        if (entityManager == null) {
            entityManager = Persistence.createEntityManagerFactory(
                JPA_UNIT_NAME).createEntityManager();
        }
        return entityManager;
    }
}
```

Le **DAO** qu'on va développer va utiliser l'**API** de **JPA** pour réaliser les opérations de persistance. Pour pouvoir l'utiliser, il faut d'abord créer une instance de la classe **EntityManager**. Pour le faire, on passe par une fabrique (**Factory**) qu'on récupère via la méthode statique `Persistence.createEntityManagerFactory()` (Je sais, encore une autre fabrique :)) en lui passant comme paramètre le nom de l'unité de persistance (le même déclaré dans `persistence.xml`):

```
<persistence-unit name="jsf-crud">
```

On verra plus tard comment utiliser l'**EntityManager** créée pour effectuer les opérations de persistance sur une entité.

 *Dans un souci de simplicité, la gestion de l'**EntityManager** dans cet article est faite à la main. Mais dans la pratique, cette approche a de nombreuses limites, et on lègue la gestion de l'**EntityManager** à un conteneur tel que le serveur d'application ou encore un conteneur*

leger comme Spring . Il suffit alors d'indiquer au conteneur d'injecter l' *EntityManager* créée dans nos DAOs .

II-D - Couche contrôle

Il s'agit ici de créer un *managed-bean JSF* (*Controller*) qui fera le lien entre les pages et la couche métier.

```
package control;

public class PersonCtrl {

}
```

Il faut ensuite déclarer cette classe dans *faces-config.xml* pour l'exposer aux pages *JSF* :

```
<managed-bean>
  <managed-bean-name>personCtrl</managed-bean-name>
  <managed-bean-class>control.PersonCtrl</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

La déclaration d'un *managed-bean JSF* prend ces paramètres:

- ***managed-bean-name*** : Le nom sous lequel le bean sera accessible depuis les pages. En général, c'est le nom de la classe avec la première lettre en minuscule (C'est ce qui est fait dans ce cas: personCtrl). Mais rien ne vous empêche de le nommer comme bon vous semble.
- ***managed-bean-class*** : Le nom complet de la classe du *managed-bean* .
- ***managed-bean-scope*** : La durée de vie du *managed-bean* (*request* , *session* , *application* , *none*). Dans ce cas, je l'ai mis à session car on aura des opérations sur plusieurs requêtes.


Une fois le *managed-bean* déclaré dans *faces-config.xml* , il devient alors accessible depuis les pages *JSF* via l' ***EL*** (*Expression Language*) " ***#{nomDuBean}*** " .

III - Implémenter l'opération Read

III-A - Dans la couche DAO

Dans `PersonDao`, on ajoute une méthode `selectAll` qui retourne la liste de toutes les personnes depuis la base de données:

```
/**
 * L'opération Read
 * @return toutes les personnes dans la base de données.
 */
public List<Person> selectAll() {
    List<Person> persons = getEntityManager().createQuery(
        "select p from Person p").getResultList();
    return persons;
}
```

 Notez que l'on accède à l' `EntityManager` via son `getter` pour s'assurer qu'il soit créé.

Comme cité plus haut, on utilise l' `EntityManager` pour exécuter une requête sur la base de données. Notez que cette requête est exprimée en **JPA-QL** (*Java Persistence API Query Language*) et non pas en `SQL` . `JPA-SQL` est similaire à `SQL` mais est orienté Objet.

III-B - Dans la couche Control

On doit ensuite modifier le contrôleur pour offrir aux pages **JSF** la possibilité de lister les entités personnes. Dans la classe `PersonCtrl` , on ajoute une liste de personnes (pour stocker les personnes récupérées de la base de données) ainsi qu'une instance du `DAO` qu'on a créé (pour pouvoir exécuter les opérations de persistance sur la base de données):

```
private PersonDao pDao = new PersonDao();
private List<Person> persons;
```

Pour initialiser cette liste, mieux vaut éviter de le faire dans le constructeur du *managed-bean* vu que l'on n'est pas sûr de l'ordre d'initialisation des différents modules, et qu'il se peut que ce constructeur soit appelé alors que **JPA** ne soit pas encore initialisé. On va donc différer l'initialisation de cette liste dans son `getter`, de cette façon:

```
public List<Person> getPersons() {
    if(persons==null){
        persons = pDao.findAll();
    }
    return persons;
}
```



JSF accède toujours aux champs d'un *managed-bean* via les `accesseurs` et les `mutateurs`.

On est donc sûr de passer par `getPersons` en la référençant depuis **JSF** via "`#{personCtrl.persons}`".

III-C - Dans la couche View

On va maintenant afficher la liste des personnes dans une page **JSF** dans un format tabulaire. On crée une page `list.jsp` avec le contenu suivant:

```

<f:view>
<h:dataTable border="0" rules="all" value="#{personCtrl.persons}"
var="p">
<h:column>
<f:facet name="header">
<h:outputText value="Prénom" />
</f:facet>
<h:outputText value="#{p.firstName}" />
</h:column>
<h:column>
<f:facet name="header">
<h:outputText value="Nom" />
</f:facet>
<h:outputText value="#{p.lastName}" />
</h:column>
</h:dataTable>
</f:view>
    
```

On utilise le composant standard **dataTable** pour afficher la liste des personnes. Ce composant prend entre autre les paramètres suivants:

- **value** : une *EL* désignant la liste à afficher. Dans ce cas, on désire afficher la liste des personnes définies dans le managed-bean PersonCtrl.
- **var** : dataTable va itérer sur la liste déclarée dans l'attribut value. A chaque itération, l'élément courant est mis dans une variable temporaire dont le nom est contrôlé par la valeur de l'attribut **var** .
- **border** et **rules** sont utilisées pour configurer l'affichage (pas de bordure, afficher des lignes autour de chaque cellule)

A l'inverse des autres taglibs (*Struts* , *JSTL* , etc.), le modèle de composants de **JSF** introduit une abstraction lors de l'itération sur une liste de données en définissant les colonnes au lieu des lignes. Ici, on définit 2 colonnes:

- Une colonne pour afficher les prénoms des personnes: Ceci est fait via le composant **column** . Ce dernier répète son contenu à chaque itération sur la liste définie dans le **dataTable** parent. Dans ce cas, le contenu de column qui est le composant **outputText** (qui permet d'afficher du texte) sera répété pour chaque personne de la liste des personnes dans une ligne de la table. **outputText** prend un attribut **value** qui indique le texte à afficher. Ici, on a mis l' *EL* **#{p.firstName}** où p est la variable d'itération (déclarée dans l'attribut var du **dataTable** parent) qui est une instance de **Person** . Le sous composant **facet** permet de définir une sous partie du composant parent (**column**) et ne sera pas répété pour chaque itération. Ici, le **facet** définit le contenu de l'entête de la colonne qui est le texte " *Prénom* " .
- Une autre colonne pour afficher les noms des personnes. elle est définie de la même façon que la première colonne.

Il faut aussi créer une page **index.jsp** dont voici le contenu:

```

<body>
<jsp:forward page="list.jsf" />
</body>
    
```

index.jsp est la page d'accueil vu qu'on l'a déclaré dans web.xml.

Pour pouvoir accéder à une page **JSF** , il faut utiliser le suffixe **.jsf** .

C'est pour cette raison qu'on passe par l'élément forward qui permet de référence la page **JSF** avec le suffixe **.jsf** .

III-D - Test

En exécutant l'application, rien d'utile dans l'affichage vu qu'on n'a pas encore de données, il faut juste s'assurer qu'il n'y ait pas eu d'exceptions au démarrage.

Mais en suivant le *log* de l'application (dans la console d' *Eclipse* par exemple), on voit :

```
[TopLink Fine]: 2007.12.02 12:10:28.968
--ServerSession(3273383)
--Connection(15513215)--Thread(Thread[http-8080-2,5,main])
--CREATE TABLE PERSON (ID NUMERIC(19) NOT NULL, LASTNAME VARCHAR(255), FIRSTNAME VARCHAR(255), PRIMARY
KEY (ID))
:
:
[TopLink Fine]: 2007.12.02 12:10:29.318--ServerSession(3273383)
--Connection(15513215)
--Thread(Thread[http-8080-2,5,main])
--SELECT ID, LASTNAME, FIRSTNAME FROM PERSON
```

Ce qui montre que *Toplink* a créé la table *PERSON* dans la base de données et qu'il a ensuite effectué un *SELECT* lors de l'affichage de la table.

Vous pouvez voir la page de listing en action sur cette adresse : [list.jsf](#) (démon hébergé sur Developpez.com)


IV - Implémenter l'opération Create

IV-A - Dans la couche DAO

Dans **PersonDao** , on ajoute la méthode suivante:

```
/**
 * L'opération Create
 * @param u La personne à insérer dans la base de données.
 * @return La personne insérée
 */
public Person insert(Person u) {
    getEntityManager().getTransaction().begin();
    getEntityManager().persist(u);
    getEntityManager().getTransaction().commit();
    return u;
}
```

Cette méthode fait appel à l'API de **JPA** pour persister une personne dans la base de données.

 *Comme vous le remarquez, j'ai entouré l'opération persist par une ouverture d'une transaction et par son commit. C'est une très mauvaise idée de s'y prendre de cette façon, i.e. gérer les transactions au niveau du DAO . Dans une application réelle, on a en général plusieurs opérations qui doivent se réaliser d'une façon atomique. Par exemple: on ajoute une entité dans la base de données et on ajoute un lien vers cette entité dans une autre table. Ces deux opérations doivent se réaliser d'une façon atomique. C'est justement la raison d'être des transactions. Or puisque les DAO offrent des fonctionnalités unitaires (create , update , etc.), il faut plutôt implémenter les transactions dans la couche Service et optimalement d'une façon déclarative en utilisant **Spring** par exemple. Cette remarque vaut aussi pour les opérations update et delete .*

IV-B - Dans la couche Control

Dans **PersonCtrl** , on ajoute un champ de type **Person** qui servira à recueillir les informations de l'utilisateur.

```
private Person newPerson = new Person();
```

On ajoute aussi l'action createPerson dans le contrôleur **PersonCtrl** qui utilise le **DAO** et l'instance **newPerson** pour ajouter une personne dans la base de données.

Cette action doit être associée à un submit dans la page **JSF** d'ajout d'utilisateur.

```
public String createPerson() {
    pDao.insert(newPerson);
    newPerson = new Person();
    persons = pDao.selectAll();
    return "list";
}
```

Cette méthode ne fait qu'appeler la méthode insert du *DAO* .
 Ensuite, elle crée à nouveau *newPerson* pour la prochaine insertion.
 Enfin, elle met à jour la liste des personnes pour refléter l'ajout de la nouvelle personne.

Pour retourner à la page de listing suite à la création d'une personne, l'action doit retourner un littéral (" *list* " par exemple dans ce cas). On doit ensuite ajouter une règle de navigation qui part de la page d'ajout d'une personne *add.jsp* et mène vers *list.jsp* suite à un résultat " *list* " dans *faces-config.xml* :

```
<navigation-rule>
  <display-name>add</display-name>
  <from-view-id>/add.jsp</from-view-id>
  <navigation-case>
    <from-outcome>list</from-outcome>
    <to-view-id>/list.jsp</to-view-id>
    <redirect />
  </navigation-case>
</navigation-rule>
```

Une règle de navigation **JSF** se configure ainsi:

- l'élément **from-view-id** : le nom de la page source.
- **navigation-case** : cas de navigation:
- **from-outcome** : le littéral qui active ce cas de navigation
- **to-view-id** : où aller si ce cas de navigation est activé.
- **redirect** : un élément optionnel qui s'il est présent, le cas de navigation effectue un *redirect* au lieu d'un *forward* .



Il est possible de déclarer plusieurs cas de navigation dans la même règle de navigation. On peut voir ça comme ceci: les cas de navigation sont groupés par la page de départ dans une règle de navigation.

IV-C - Dans la couche View

On crée une page *add.jsp* , qui permet de saisir les données d'une nouvelle personne et d'invoquer l'action *createPerson* :

```
<f:view>
  <h:form>
    <h:panelGrid border="0" columns="3" cellpadding="5">
      <h:outputText value="Prénom" />
      <h:inputText id="firstName" value="#{personCtrl.newPerson.firstName}"
        required="true" requiredMessage="Prénom obligatoire" />
      <h:message for="firstName" />

      <h:outputText value="Nom" />
      <h:inputText id="lastName" value="#{personCtrl.newPerson.lastName}"
        required="true" requiredMessage="Nom obligatoire" />
      <h:message for="firstName" />
      <h:outputText />
      <h:commandButton value="Ajouter" action="#{personCtrl.createPerson}" />
    </h:panelGrid>
  </h:form>
</f:view>
```

Quelques notes:

- J'ai utilisé le composant **panelGrid** qui permet d'aligner ses fils en une grille pour la mise en page. Ici, j'ai défini 3 colonnes.
- J'ai utilisé le composant **inputText** qui permet de récupérer une valeur textuelle.

L'attribut *value* de ce composant permet d'associer sa valeur avec un champ d'un *managed-bean* via une **EL**. Ici, je lie le premier champ textuel avec le champ *firstName* de **personCtrl**. *newPerson* et le second avec le champ *lastName*.

- J'ai ajouté des contraintes *required=true* pour les deux champs texte (nom et prénom) pour indiquer à **JSF** que leurs valeurs ne peuvent pas être vides ainsi que des éléments **message** pour afficher les messages d'erreur en cas d'erreur de validation.
- Enfin, un **commandButton** pour invoquer l'action **PersonCtrl**. *createPerson* (spécifié via une **EL** dans l'attribut **action**).

Pour simplifier la navigation entre les deux pages **add.jsp** et **list.jsp**, on va ajouter un menu dans les deux pages dont voici le code:

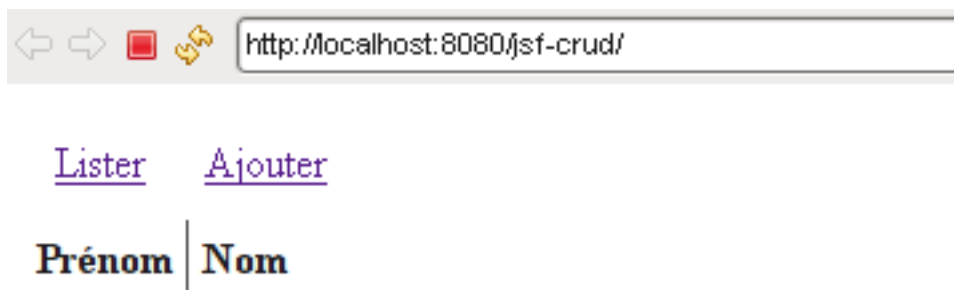
```
<h:panelGrid columns="2" cellpadding="10">
  <h:outputLink value="list.jsf">
    <h:outputText value="Lister" />
  </h:outputLink>

  <h:outputLink value="add.jsf">
    <h:outputText value="Ajouter" />
  </h:outputLink>
</h:panelGrid>
```

Ce ne sont que deux liens hypertexte dans une grille de 2 colonnes. Ce menu est à mettre dans les deux pages **add.jsp** et **list.jsp** au tout début de la page juste après le composant **<f:view>**.

IV-D - Test

On effectue un test en ajoutant quelques personnes:
Initialement la liste est vide:



Liste vide

En cliquant sur le lien "Ajouter", on passe à la page d'ajout:



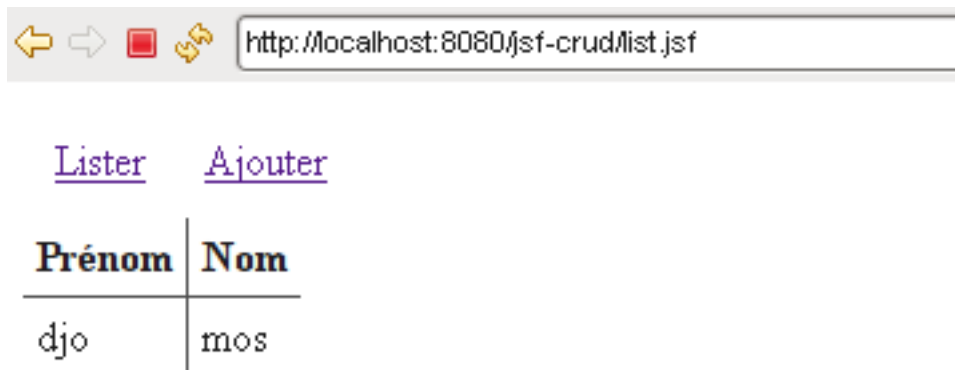
[Lister](#) [Ajouter](#)

Prénom

Nom

Page d'ajout d'une personne

Après avoir rempli et validé le formulaire, on revient à la liste des personnes:



[Lister](#) [Ajouter](#)

Prénom	Nom
djo	mos

Liste avec la personne ajoutée

Vous pouvez tester l'opération de création sur cette adresse : [add.jsf](#) (démon hébergé sur Developpez.com)


V - Implémenter l'opération Delete

V-A - Dans la couche DAO

On commence par ajouter une méthode delete à **PersonDao** :

```
/**
 * L'opération Delete
 * @param u La personne à supprimer de la base de donnés.
 */
public void delete(Person u) {
    getEntityManager().getTransaction().begin();
    u = getEntityManager().merge(u); //<-Important
    getEntityManager().remove(u);
    getEntityManager().getTransaction().commit();
}
```

 La remarque [mentionnée ici](#) s'applique aussi ici.

 L'instruction " `u = em.merge(u)` " est très importante, sans elle, la méthode `deletePerson` risque de déclencher une exception car l'instance qu'on lui aura passé est détachée de la session de persistance.

La méthode `merge` de la classe `EntityManager` s'occupe de rattacher une entité à la session de persistance en cours.

Cette étape est inutile dans un environnement managé (où l' `EntityManager` est géré par le conteneur plutôt que par le développeur).

V-B - Dans la couche Control

Pour implémenter la fonction supprimer, on ajoute généralement un lien ou un bouton supprimer dans chaque ligne de la liste d'affichage.

Suite à un clic sur le bouton supprimer, on invoque d'action delete qui doit récupérer la ligne sélectionnée et la supprimer.

Pour implémenter facilement un fonctionnement pareil dans **JSF** , on utilise un **DataModel** comme conteneur de la liste des valeurs et non plus une liste simple (`java.util.List`).

En effet, dans le code d'une action donnée, `DataModel` permet de récupérer à tout moment la ligne ayant déclenchée l'action.

On remplace donc dans **PersonCtrl** le champ `persons` de type `List` et son accesseur par:

```
private DataModel persons;

public DataModel getPersons() {
    if (persons == null) {
        persons = new ListDataModel();
        persons.setWrappedData(pDao.selectAll());
    }
    return persons;
}
```

`DataModel` est une interface tandis que `ListDataModel` est une implémentation (tout comme pour `List` et `ArrayList`).

Pour spécifier les éléments du *DataModel*, on passe une liste ordinaire (*java.util.List*, *java.util.Set*) comme paramètre à la méthode *setWrappedData*.

Il faut aussi mettre à jour la méthode *createPerson* pour refléter l'utilisation de *DataModel* au lieu de *List* :

```
public String createPerson() {
    pDao.insert(newPerson);
    newPerson = new Person();
    persons.setWrappedData(pDao.selectAll());
    return "list";
}
```

Voici ensuite la méthode *deletePerson* du contrôleur:

```
/**
 * L'opération de suppression à invoquer suite à la sélection d'une
 * personne.
 *
 * @return outcome pour la navigation entre les pages. Dans ce cas, c'est null pour
 *         rester dans la page de listing.
 */
public String deletePerson() {
    Person p = (Person) persons.getRowData();
    pDao.delete(p);
    persons.setWrappedData(pDao.selectAll());
    return null;
}
```

La première ligne récupère la personne correspondant à la ligne sélectionnée dans la table.

Reste plus qu'à invoquer la méthode *delete* du *DAO* et à mettre à jour la liste des personnes pour refléter la suppression.

V-C - Dans la couche View

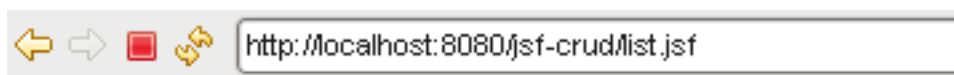
Coté présentation, on ajoute une nouvelle colonne à la table des personnes dans *list.jsp* qui contient un bouton delete sur chaque ligne:

```
<h:column>
  <f:facet name="header">
    <h:outputText value="Opérations" />
  </f:facet>
  <h:commandButton value="Supprimer"
    action="#{personCtrl.deletePerson}" />
</h:column>
```



Il ne faut pas oublier d'englober le *dataTable* dans un *<h:form>* vu que l'on a des *commandButton*.

V-D - Test



[Lister](#) [Ajouter](#)

Prénom	Nom	Opérations
djo	mos	<input type="button" value="Supprimer"/>

Liste des personnes avec les boutons supprimer

En cliquant sur supprimer dans une ligne de la table, la personne correspondante est supprimée de la base de données et la liste est mise à jour

Vous pouvez tester l'opération de suppression sur cette adresse : [list.jsf](#) (démon hébergé sur Developpez.com) en cliquant sur le bouton supprimer d'une ligne

VI - Implémenter l'opération Update


VI-A - Dans la couche DAO

Dans `PersonDao`, on ajoute la méthode `update` que voici:

```

/**
 * L'opération Update
 * @param u La personne à mettre à jour dans la base de données.
 * @return La personne mise à jour
 */
public Person update(Person u) {
    getEntityManager().getTransaction().begin();
    u = getEntityManager().merge(u);
    getEntityManager().getTransaction().commit();
    return u;
}

```

 Les remarques mentionnées [ici](#) et [ici](#) s'appliquent aussi [ici](#).


Normalement, une mise à jour est réalisée de la même façon qu'une insertion, c'est à dire avec la méthode `persist` de la classe `EntityManager`. **JPA** s'occupe ensuite de décider s'il s'agit d'une nouvelle entité et qu'il faut l'ajouter ou d'une entité déjà ajoutée et qu'il faut plutôt la mettre à jour.

Mais pour les mêmes raisons que pour la méthode `delete`, c'est à dire pour éviter le cas où l'entité passée est détachée, on utilise la méthode **merge** de la classe `EntityManager` qui rattache une entité à la session de persistance en cours tout en intégrant les modifications qu'elle a pu subir.

VI-B - Dans la couche Control

Dans le contrôleur `PersonCtrl`, on ajoute un champ `editPerson` de type `Person` qui servira à recueillir les données saisies par l'utilisateur ainsi que son accesseur:

```
private Person editPerson;
```

 Vous remarquerez qu'à l'inverse de `newPerson`, on n'a pas initialisé `editPerson` et qu'on l'a laissé à `null`.

En effet, `editPerson` sera utilisée dans la page `edit.jsp`, mais avant, on lui affectera la valeur de la personne sélectionnée pour édition dans la table des personnes.

Dans le contrôleur, on ajoute aussi la méthode `editPerson` qui sera appelée quand on clique sur le bouton modifier d'une ligne. Cette méthode affecte à `editPerson` la personne sélectionné et redirige l'utilisateur vers la page `edit.jsp`:

```

/**
 * Opération intermédiaire pour récupérer la personne à modifier.
 * @return outcome pour la navigation entre les pages. Dans ce cas, c'est "edit" pour
 * aller à la page de modification.
 */
public String editPerson() {
    editPerson = (Person) persons.getRowData();
    return "edit";
}

```

ainsi que la méthode qui effectue la mise à jour:

```


```

```

/**
 * L'opération Update pour faire la mise à jour.
 * @return outcome pour la navigation entre les pages. Dans ce cas, c'est "list" pour
 *         retourner à la page de listing.
 */
public String updatePerson() {
    pDao.update(editPerson);
    persons.setWrappedData(pDao.selectAll());
    return "list";
}
    
```

On ajoute alors la règle de navigation suivante:

```

<navigation-rule>
  <from-view-id>/list.jsp</from-view-id>
  <navigation-case>
    <from-outcome>edit</from-outcome>
    <to-view-id>/edit.jsp</to-view-id>
    <redirect />
  </navigation-case>
</navigation-rule>
    
```

Cette règle permet à partir de la page **list.jsp** d'aller à la page **edit.jsp** si on a un résultat " **edit** "

On ajoute aussi la règle de navigation suivante:

```

<navigation-rule>
  <from-view-id>/edit.jsp</from-view-id>
  <navigation-case>
    <from-outcome>list</from-outcome>
    <to-view-id>/list.jsp</to-view-id>
    <redirect />
  </navigation-case>
</navigation-rule>
    
```

Cette règle permet à partir de la page **edit.jsp** d'aller à la page **list.jsp** si on a un résultat " **list** "

VI-C - Dans la couche View

On ajoute aussi le bouton modifier dans la page **list.jsp** . Ce bouton sera ajouté dans la même colonne (opérations) que le bouton supprimer et il invoque la méthode `PersonCtrl.editPerson` :

```

<h:column>
  <f:facet name="header">
    <h:outputText value="Opérations" />
  </f:facet>
  <h:commandButton value="Modifier"
    action="#{personCtrl.editPerson}" />
  <h:commandButton value="Supprimer"
    action="#{personCtrl.deletePerson}" />
</h:column>
    
```

On crée la page **edit.jsp** qui est exactement similaire à **add.jsp** avec la seule différence qu'elle pointe vers `editPerson` au lieu de `newPerson` et qu'elle invoque `updatePerson` au lieu de `createPerson` :

```

<f:view>
  <h:panelGrid columns="2" cellpadding="10">
    <h:outputLink value="list.jsf">
      <h:outputText value="Lister" />
    </h:outputLink>
    
```

```

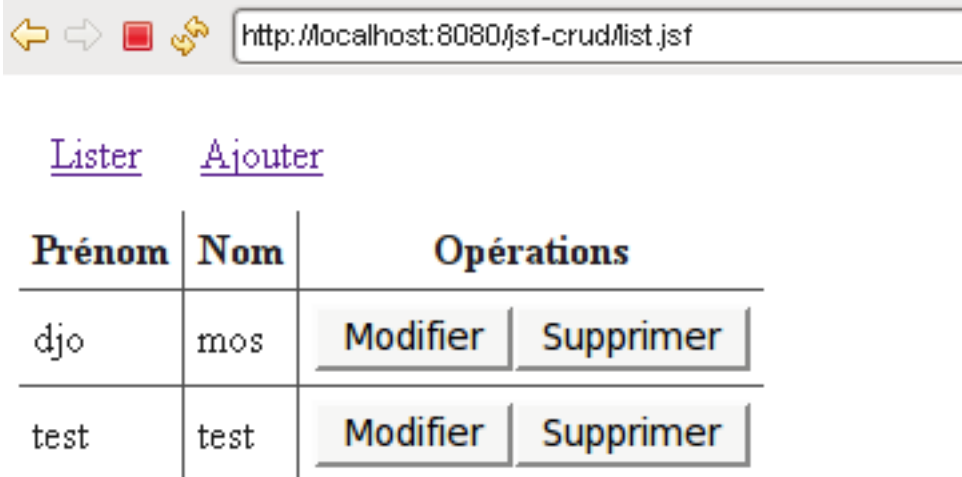
<h:outputLink value="add.jsf">
  <h:outputText value="Ajouter" />
</h:outputLink>
</h:panelGrid>
<h:form>
  <h:panelGrid border="0" columns="3" cellpadding="5">
    <h:outputText value="Prénom" />
    <h:inputText id="firstName"
      value="#{personCtrl.editPerson.firstName}" required="true"
      requiredMessage="Prénom obligatoire" />
    <h:message for="firstName" />

    <h:outputText value="Nom" />
    <h:inputText id="lastName" value="#{personCtrl.editPerson.lastName}"
      required="true" requiredMessage="Nom obligatoire" />
    <h:message for="firstName" />
    <h:outputText />
    <h:commandButton value="Mettre à jour"
      action="#{personCtrl.updatePerson}" />
  </h:panelGrid>
</h:form>
</f:view>

```

VI-D - Test

On commence par la page *list.jsp* :



Listes des personnes avec l'ajout du bouton modifier.

Comme vous le remarquez, on voit un bouton modifier dans chaque ligne de la table. En cliquant sur un de ces boutons, on passe à la page *edit.jsp* qui permet d'éditer la personne sélectionnée:



← → 🛑 🔄

[Lister](#) [Ajouter](#)

Prénom

Nom

La page d'édition d'une personne

On entre de nouvelles valeurs pour le nom et le prénom de la personne, ce qui donne:



← → 🛑 🔄

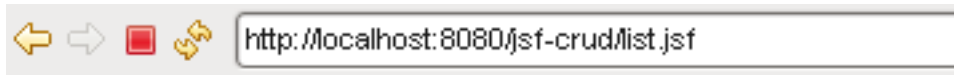
[Lister](#) [Ajouter](#)

Prénom

Nom

La page d'édition d'une personne

En validant, la personne est mise à jour dans la base de données et on revient à la page **list.jsp** où on voit la personne mise à jour:



[Lister](#) [Ajouter](#)

Prénom	Nom	Opérations	
jawher	moussa	Modifier	Supprimer
test	test	Modifier	Supprimer

Liste des personnes mise à jour

Vous pouvez tester l'opération de modification sur cette adresse : **list.jsf** (démon hébergé sur Developpez.com) en cliquant sur le bouton modifier d'une ligne.

VII - Télécharger

Le code source de cette application ainsi que toutes ses dépendances sont disponibles en téléchargement dans l'archive suivant.

[Source crud-jsf-jpa.zip](#)


- dossier **ant** : contient les jars d' **Apache Ant 1.7.0** . Ils ne sont pas utilisés par l'application mais servent uniquement au build.
- dossier **lib** : contient toutes les dépendances de l'application.
- dossier **src** : contient les sources *Java* de l'application ainsi que le descripteur de persistance.
- dossier **web** : contient les pages jsp de l'application ainsi que **web.xml** et **faces-config.xml** .
- fichier **build.xml** : fichier de *build Ant*
- fichier **build.bat** : permet de lancer le *build* sous *Windows* .
- fichier **build.sh** : permet de lancer le *build* sous *Linux* .

L'archive contient un script de *build Ant* .

Une fois extrait dans le disque local, il suffit de lancer la compilation en exécutant **build.bat** (Si vous êtes sous *Windows*) ou **build.sh** (Si vous êtes sous *Linux*).

Si tout se passe bien, le fichier *war* de l'application sera généré dans le dossier **dist/war** .

 Assurez vous avant qu'une version  **JDK 5** ou antérieur est installé et que la variable d'environnement **JAVA_HOME** soit bien définie et qu'elle pointe vers le **JDK** installé.

 Il se peut que vous ayez à ajouter le droit **x** (exécution) à **build.sh** sous *Linux* pour pouvoir l'exécuter.

VIII - Conclusion

Dans cet article, j'ai essayé de montrer d'une façon plus ou moins détaillée de montrer comment créer une application web de type *CRUD* qui utilise **JSF** comme *framework* web (présentation et contrôle) et **JPA** comme *framework* de persistance tout en expliquant au maximum chacune des étapes.

Il faut cependant noter que dans un souci de simplicité, j'ai dû m'y prendre d'une façon non-optimale et non-conforme aux design patterns en vigueur dans ce genre d'applications.

Les points à soigner sont les suivants:

- Absence de la couche service
- Gestion manuelle de l' *EntityManager*
- Gestion manuelle des transactions

Je compte montrer comment régler ces points en utilisant **Spring** dans un autre article.

IX - Remerciements

Mes remerciements les plus sincères vont à **azerr** , **heid** , **ChristopheJ Ricky81** pour leurs retours et **Baptiste Wicht** et **eric190** pour leur relecture.

Je tiens aussi à remercier **lunatix** pour la mise en place de **la démo en ligne de l'application présentée dans cet article**.