

# TP7: JSF

## 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

### II-A. Création du projet

- Créez un projet dans Eclipse
- Ajoutez les jars fournis
- Créez 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. Pour cela, créez un fichier `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="miageTP7">
    <properties>
      <property name="toplink.logging.level" value="FINE" />
      <property name="toplink.target-database" value="PostgreSQL" />
      <property name="toplink.jdbc.driver" value="org.postgresql.Driver" />
      <property name="toplink.jdbc.url" value="jdbc:postgresql://127.0.0.1:5432/postgres" />
      <property name="toplink.jdbc.user" value="postgres" />
      <property name="toplink.jdbc.password" value="postgres" />
      <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 (**miageTP7**) ainsi que les paramètres de connexion à la base de données qui sont:

- Pilote JDBC: "org.postgresql.jdbcDriver"
- URL de connexion: "**jdbc:postgresql://127.0.0.1:5432/postgres**" qui indique à **postgreSQL** de stocker les données dans une base nommée "**postgres**".
- Login: "postgres"
- Password: "postgres"
- `toplink.ddl-generation`: "**create-tables**" indique à *Toplink* de créer les tables si elles n'existent pas.

### II-B. 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 ce TP, 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-B-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.
 */
@Entity
public class Person {
    private Long id;
    private String firstName;
    private String lastName;

    /**
     * C'est l'accesseur 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 que sa valeur 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-B-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.
 */
public class PersonDao {
    private static final String JPA_UNIT_NAME = "miageTP7";
    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()` en lui passant comme paramètre le nom de l'unité de persistance (le même déclaré dans *persistence.xml*):

```
<persistence-unit name="miageTP7">
```

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

## II-C. 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>
```

### 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;
}
```

### 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. Ici, on définit 2 colonnes nom et prénom.

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

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

**index.jsp** est la page d'accueil déclarée dans `web.xml`.

### III-D. Test

Testez l'application (vérifiez les logs, consultez la base de donnée).

## 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, l'opération `persist` est entourée par une ouverture d'une transaction et par son `commit`. Cette façon de faire est utilisée ici mais n'est pas conseillée, 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>
```

#### 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}" />
      <h:message for="firstName" />

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

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

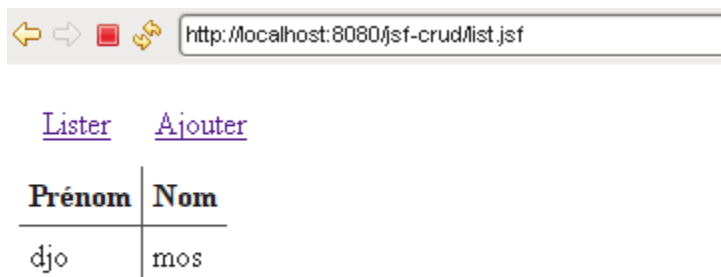
On effectue un test en ajoutant quelques personnes:  
Initialement la liste est vide:



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



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



#### 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 données.
 */
public void delete(Person u) {
    getEntityManager().getTransaction().begin();
    u = getEntityManager().merge(u); //<-Important
    getEntityManager().remove(u);
    getEntityManager().getTransaction().commit();
}
```



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

Côté 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	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

## 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;
}
```

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 sélectionnée 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>update</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 "***update***"

## 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*:

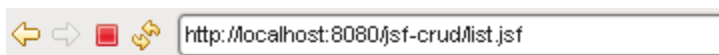
```
<f:view>
    <h:panelGrid columns="2" cellpadding="10">
        <h:outputLink value="list.jsf">
            <h:outputText value="Lister" />
        </h:outputLink>

        <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="lastName" />
            <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**:



[Lister](#)   [Ajouter](#)

Prénom	Nom	Opérations	
djo	mos	Modifier	Supprimer
test	test	Modifier	Supprimer

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

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

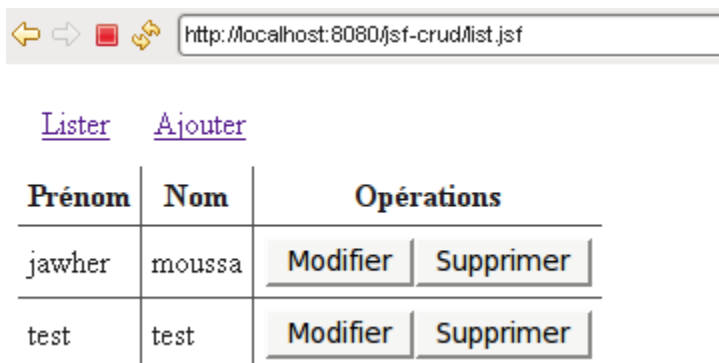


[Lister](#) [Ajouter](#)

Prénom

Nom

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	<input type="button" value="Modifier"/>	<input type="button" value="Supprimer"/>
test	test	<input type="button" value="Modifier"/>	<input type="button" value="Supprimer"/>