www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Spring MVC par l'exemple
- Partie 4 -

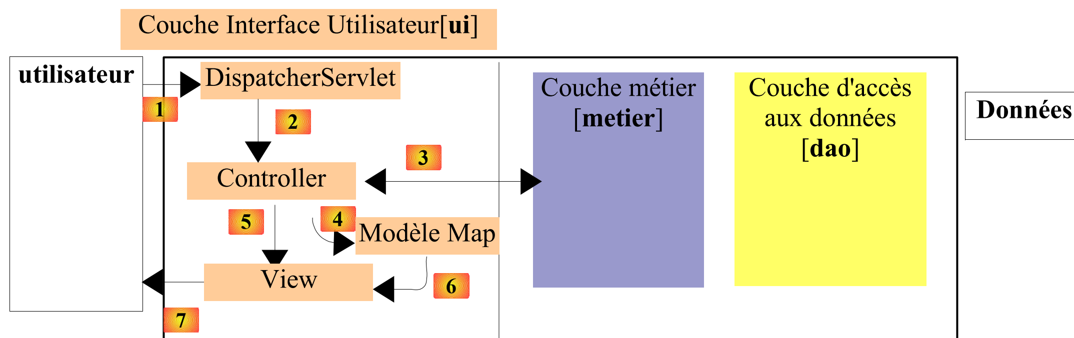
serge.tahe@istia.univ-angers.fr, avril 2006

1 Rappels

Nous poursuivons dans cet article le travail fait dans les précédents articles :

- Spring MVC par l'exemple – partie 1 : [http://tahe.developpez.com/java/springmvc-part1]
- Spring MVC par l'exemple – partie 2 : [http://tahe.developpez.com/java/springmvc-part2]
- Spring MVC par l'exemple – partie 3 : [http://tahe.developpez.com/java/springmvc-part3]

L'architecture d'une application Spring MVC est la suivante :



Dans ces trois articles, nous avons étudié les points suivants :

article 1 :

- les différentes stratégies de résolutions d'URL qui associent à une URL demandée par le client, un contrôleur implémentant l'interface [Controller] qui va traiter la demande du client (1,2)
- les différentes façons qu'avait un contrôleur [Controller] d'accéder au contexte de l'application, par exemple pour accéder aux instances des couches [métier] et [dao]
- les différentes stratégies de résolutions de noms de vue qui, à un nom de vue rendu par le contrôleur [Controller] associe une classe implémentant l'interface [View] chargée d'afficher le modèle [Map] construit par le contrôleur [Controller] (4,5)

article 2 :

- les classes [HandlerInterceptor] qui filtrent la requête avant de la passer au [Controller] qui doit la traiter (2)
- les différentes façons de gérer l'internationalisation (*localisation*) d'une application web
- les gestionnaires des exceptions qui peuvent de produire dans une application web
- la gestion des formulaires grâce à la classe [SimpleFormController]

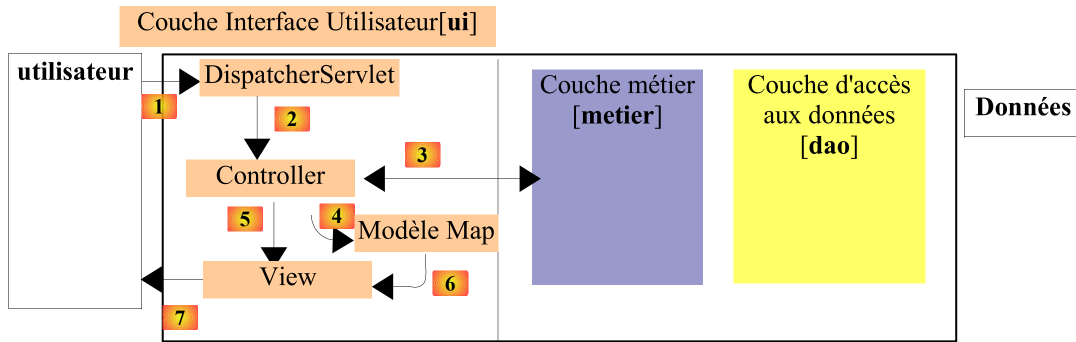
article 3 :

- des implémentations de l'interface [Controller] pas encore abordées dans les autres articles
- des implémentations de l'interface [View] permettant de générer des documents PDF et Excel
- des outils facilitant l'écriture de formulaires destinés à télécharger des documents du poste client vers le serveur

2 Spring MVC dans une architecture 3tier – Exemple 1

2.1 Présentation

Jusqu'à maintenant, nous nous sommes contentés d'exemples à visée pédagogique. Pour cela, ils se devaient d'être simples. Nous présentons maintenant, une application basique mais néanmoins plus riche que toutes celles présentées jusqu'à maintenant. Elle aura la particularité d'utiliser les trois couches d'une architecture 3tier :



L'application web correspondante va permettre de gérer un groupe de personnes avec quatre opérations :

- liste des personnes du groupe
- ajout d'une personne au groupe
- modification d'une personne du groupe
- suppression d'une personne du groupe

On reconnaîtra les quatre opérations de base sur une table de base de données. Nous écrirons deux versions de cette application :

- dans la version 1, la couche [dao] n'utilisera pas de base de données. Les personnes du groupe seront stockées dans un simple objet [ArrayList] géré en interne par la couche [dao]. Cela permettra au lecteur de tester l'application sans contrainte de base de données.
- dans la version 2, nous placerons le groupe de personnes dans une table de base de données. Nous montrerons que cela se fera sans impact sur la couche web de la version 1 qui restera identique.

Les copies d'écran qui suivent montrent les pages que l'application échange avec l'utilisateur.

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1145455950968	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1145455950968	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	1145455950968	Charles	Lemarchand	01/01/1986	false	0	Modifier	Supprimer

Ajout

Une liste initiale de personnes est tout d'abord présentée à l'utilisateur. Il peut ajouter une personne ->

Ajout/Modification d'une personne

Id	-1
Version	0
Prénom	Sophie
Nom	Maxima
Date de naissance (JJ/MM/AAAA)	13/03/1946
Marié	◉ Oui ○ Non
Nombre d'enfants	4

Valider Annuler

L'utilisateur a créé une nouvelle personne qu'il valide avec le bouton [Valider] ->

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1145455950968	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1145455950968	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	1145455950968	Charles	Lemarchand	01/01/1986	false	0	Modifier	Supprimer
4	1145456218562	Sophie	Maxima	13/03/1946	true	4	Modifier	Supprimer

Ajout

La nouvelle personne a été ajoutée. On la modifie maintenant ->

Ajout/Modification d'une personne

Id	4
Version	1145456218562
Prénom	Sophie
Nom	Maxima
Date de naissance (JJ/MM/AAAA)	13/03/1956
Marié	○ Oui ◉ Non
Nombre d'enfants	2

Valider Annuler

On modifie la date de naissance, l'état marital, le nombre d'enfants et on valide ->

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1145455950968	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1145455950968	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	1145455950968	Charles	Lemarchand	01/01/1986	false	0	Modifier	Supprimer
4	1145456441968	Sophie	Maxima	13/03/1956	false	2	Modifier	Supprimer

Ajout

On retouve la personne telle qu'elle a été modifiée. On la supprime maintenant ->

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1145455950968	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1145455950968	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	1145455950968	Charles	Lemarchand	01/01/1986	false	0	Modifier	Supprimer

Ajout

Elle n'est plus là.

Ajout/Modification d'une personne

Id	-1
Version	0
Prénom	
Nom	
Date de naissance (JJ/MM/AAAA)	
Marié	○ Oui ⦿ Non
Nombre d'enfants	xx

Valider Annuler

Les erreurs de saisie sont signalées ->

Ajout/Modification d'une personne

Id	-1	
Version	0	
Prénom		Le nom est obligatoire !
Nom		Le prénom est obligatoire !
Date de naissance (JJ/MM/AAAA)		Donnée incorrecte !
Marié	○ Oui ⦿ Non	
Nombre d'enfants	xx	Donnée incorrecte !

Valider Annuler

On notera que le formulaire a été renvoyé tel qu'il a été saisi (Nombre d'enfants). Le lien [Annuler] permet de revenir à la liste des personnes ->

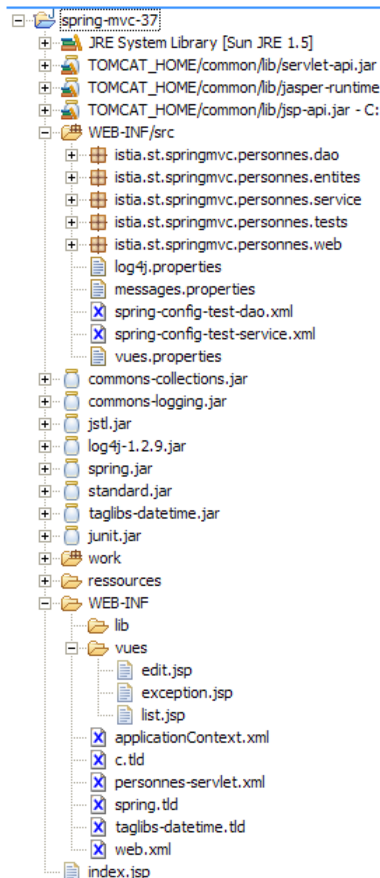
Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1145455950968	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1145455950968	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	1145455950968	Charles	Lemarchand	01/01/1986	false	0	Modifier	Supprimer

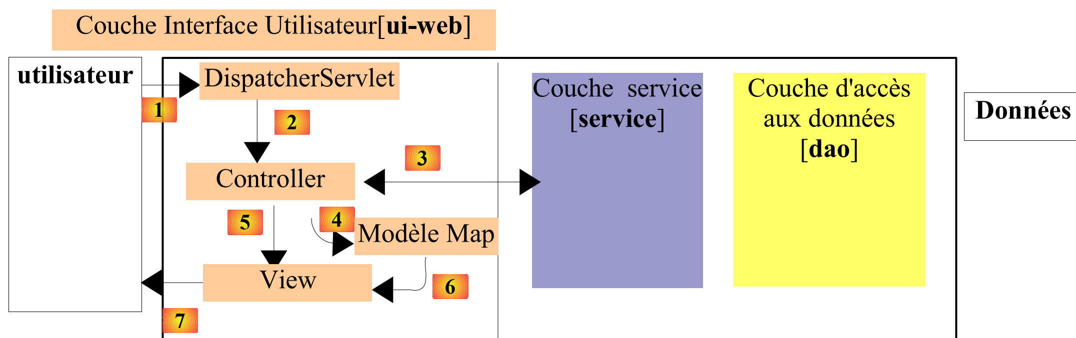
Ajout

2.2 Le projet Eclipse / Tomcat

Le projet de l'application s'appelle [spring-mvc-37] :



Ce projet recouvre les trois couches de l'architecture 3tier de l'application :

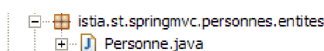


- la couche [dao] est contenue dans le paquetage [istia.st.springmvc.personnes.dao]
- la couche [metier] ou [service] est contenue dans le paquetage [istia.st.springmvc.personnes.service]
- la couche [web] ou [ui] est contenue dans le paquetage [istia.st.springmvc.personnes.web]
- le paquetage [istia.st.springmvc.personnes.entites] contient les objets partagés entre différentes couches
- le paquetage [istia.st.springmvc.personnes.tests] contient les tests Junit des couches [dao] et [service]

Nous allons explorer successivement les trois couches [dao], [service] et [web]. Parce que ce serait trop long à écrire et peut-être trop ennuyeux à lire, nous serons peut-être parfois un peu rapides sur les explications sauf lorsque ce qui est présenté est nouveau.

2.3 La représentation d'une personne

L'application gère un groupe de personnes. Les copies d'écran page 3 ont montré certaines des caractéristiques d'une personne. Formellement, celles-ci sont représentées par une classe [Personne] :



La classe [Personne] est la suivante :

```
1.  package istia.st.springmvc.personnes.entites;
2.  
3.  import java.text.SimpleDateFormat;
4.  import java.util.Date;
5.  
6.  public class Personne {
7.  
8.    // identifiant unique de la personne
9.    private int id;
10.   // la version actuelle
11.   private long version;
12.   // le nom
13.   private String nom;
14.   // le prénom
15.   private String prenom;
16.   // la date de naissance
17.   private Date dateNaissance;
18.   // l'état marital
19.   private boolean marie = false;
20.   // le nombre d'enfants
21.   private int nbEnfants;
22. 
23.   // getters - setters
24. ...
25. 
26.   // constructeur par défaut
27.   public Personne() {
28. 
29.   }
30. 
31.   // constructeur avec initialisation des champs de la personne
32.   public Personne(int id, String prenom, String nom, Date dateNaissance,
33.       boolean marie, int nbEnfants) {
34.     setId(id);
35.     setNom(nom);
36.     setPrenom(prenom);
37.     setDateNaissance(dateNaissance);
38.     setMarie(marie);
39.     setNbEnfants(nbEnfants);
40.   }
41. 
42.   // constructeur d'une personne par recopie d'une autre personne
43.   public Personne(Personne p) {
44.     setId(p.getId());
45.     setVersion(p.getVersion());
46.     setNom(p.getNom());
47.     setPrenom(p.getPrenom());
48.     setDateNaissance(p.getDateNaissance());
49.     setMarie(p.getMarie());
50.     setNbEnfants(p.getNbEnfants());
51.   }
52. 
53. 
54.   // toString
55.   public String toString() {
56.     return "[" + id + "," + version + "," + prenom + "," + nom + ","
57.         + new SimpleDateFormat("dd/MM/yyyy").format(dateNaissance)
58.         + "," + marie + "," + nbEnfants + "]";
59.   }
60. }
```

- une personne est identifiée par les informations suivantes :
  - **id** : un n° identifiant de façon unique une personne
  - **nom** : le nom de la personne
  - **prenom** : son prenom
  - **dateNaissance** : sa date de naissance
  - **marie** : son état marié ou non
  - **nbEnfants** : son nombre d’enfants
- l’attribut [**version**] est un attribut artificiellement ajouté pour les besoins de l’application. D’un point de vue objet, il aurait été sans doute préférable d’ajouter cet attribut dans une classe dérivée de [Personne]. Son besoin apparaît lorsqu’on fait des cas d’usage de l’application web. L’un d’entre-eux est le suivant :

  > Au temps T1, un utilisateur U1 entre en modification d’une personne P. A ce moment, le nombre d’enfants est 0. Il passe ce nombre à 1 mais avant qu’il ne valide sa modification, un utilisateur U2 entre en modification de la même personne P. Puisque U1 n’a pas encore validé sa modification, U2 voit le nombre d’enfants à 0. U2 passe le nom de la personne P en majuscules. Puis U1 et U2 valident leurs modifications dans cet ordre. C’est la modification de U2 qui va gagner : le nom va passer en majuscules et le nombre d’enfants va rester à zéro alors même que U1 croit l’avoir changé en 1.

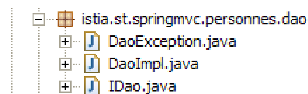
La notion de version de personne nous aide à résoudre ce problème. On reprend le même cas d'usage :

Au temps T1, un utilisateur U1 entre en modification d'une personne P. A ce moment, le nombre d'enfants est 0 et la version V1. Il passe le nombre d'enfants à 1 mais avant qu'il ne valide sa modification, un utilisateur U2 entre en modification de la même personne P. Puisque U1 n'a pas encore validé sa modification, U2 voit le nombre d'enfants à 0 et la version à V1. U2 passe le nom de la personne P en majuscules. Puis U1 et U2 valident leurs modifications dans cet ordre. Avant de valider une modification, on vérifie que celui qui modifie une personne P détient la même version que la personne P actuellement enregistrée. Ce sera le cas de l'utilisateur U1. Sa modification est donc acceptée et on change alors la version de la personne modifiée de V1 à V2 pour noter le fait que la personne a subi un changement. Lors de la validation de la modification de U2, on va s'apercevoir qu'il détient une version V1 de la personne P, alors qu'actuellement la version de celle-ci est V2. On va alors pouvoir dire à l'utilisateur U2 que quelqu'un est passé avant lui et qu'il doit repartir de la nouvelle version de la personne P. Il le fera, récupèrera une personne P de version V2 qui a maintenant un enfant, passera le nom en majuscules, validera. Sa modification sera acceptée si la personne P enregistrée a toujours la version V2. Au final, les modifications faites par U1 et U2 seront prises en compte alors que dans le cas d'usage sans version, l'une des modifications était perdue.

- lignes 32-40 : un constructeur capable d'initialiser les champs d'une personne. On omet le champ [version].
- lignes 43-51 : un constructeur qui crée une copie de la personne qu'on lui passe en paramètre. On a alors deux objets de contenu identique mais référencés par deux pointeurs différents.
- ligne 55 : la méthode [toString] est redéfinie pour rendre une chaîne de caractères représentant l'état de la personne

2.4 La couche [dao]

La couche [dao] est constituée des classes et interfaces suivantes :



- [IDao] est l'interface présentée par la couche [dao]
- [DaoImpl] est une implémentation de celle-ci où le groupe de personnes est encapsulé dans un objet [ArrayList]
- [DaoException] est un type d'exceptions non contrôlées (unchecked), lancées par la couche [dao]

L'interface [IDao] est la suivante :

```
1.  package istia.st.springmvc.personnes.dao;
2.
3.  import istia.st.springmvc.personnes.entites.Personne;
4.
5.  import java.util.Collection;
6.
7.  public interface IDao {
8.    // liste de toutes les personnes
9.    Collection getAll();
10.   // obtenir une personne particulière
11.   Personne getOne(int id);
12.   // ajouter/modifier une personne
13.   void saveOne(Personne personne);
14.   // supprimer une personne
15.   void deleteOne(int id);
16. }
```

- l'interface a quatre méthodes pour les quatre opérations que l'on souhaite faire sur le groupe de personnes :
  - **getAll** : pour obtenir une collection de personnes
  - **getOne** : pour obtenir une personne ayant un *id* précis
  - **saveOne** : pour ajouter une personne (*id*=-1) ou modifier une personne existante (*id* <> -1)
  - **deleteOne** : pour supprimer une personne ayant un *id* précis

La couche [dao] est susceptible de lancer des exceptions. Celles-ci seront de type [DaoException] :

```
1.  package istia.st.springmvc.personnes.dao;
2.
3.  public class DaoException extends RuntimeException {
4.
5.    // code erreur
```

```
6.   private int code;
7.
8.   public int getCode() {
9.     return code;
10.  }
11.
12. // constructeur
13.  public DaoException(String message,int code) {
14.    super(message);
15.    this.code=code;
16.  }
17. }
```

- ligne 3 : la classe [DaoException] dérivant de [RuntimeException] est un type d'exception non contrôlée : le compilateur ne nous oblige pas à :
  - gérer ce type d'exceptions avec un try / catch lorsqu'on appelle une méthode pouvant la lancer
  - mettre le marqueur " throws DaoException " dans la signature d'une méthode susceptible de lancer l'exception

  Cette technique nous évite d'avoir à signer les méthodes de l'interface [IDao] avec des exceptions d'un type particulier. Toute implémentation lançant des exceptions non contrôlées sera alors acceptable amenant ainsi de la souplesse dans l'architecture.

- ligne 6 : un code d'erreur. La couche [dao] lancera diverses exceptions qui seront identifiées par des codes d'erreur différents. Cela permettra à la couche qui décidera de gérer l'exception de connaître l'origine exacte de l'erreur et de prendre ainsi les mesures appropriées. Il y a d'autres façons d'arriver au même résultat. L'une d'elles est de créer un type d'exception pour chaque type d'erreur possible, par exemple NomManquantException, PrenomManquantException, AgeIncorrectException, ...

- lignes 13-16 : le constructeur qui permettra de créer une exception identifiée par un code d'erreur ainsi qu'un message d'erreur.

- lignes 8-10 : la méthode qui permettra au code de gestion d'une exception d'en récupérer le code d'erreur.

La classe [DaoImpl] implémente l'interface [IDao] :

```
1.  package istia.st.springmvc.personnes.dao;
2.
3.  import istia.st.springmvc.personnes.entites.Personne;
4.
5.  import java.text.ParseException;
6.  import java.text.SimpleDateFormat;
7.  import java.util.ArrayList;
8.  import java.util.Collection;
9.
10. public class DaoImpl implements IDao {
11.
12.   // une liste de personnes
13.   private ArrayList personnes = new ArrayList();
14.
15.   // n° de la prochaine personne
16.   private int id = 0;
17.
18.   // initialisations
19.   public void init() {
20.     try {
21.       Personne p1 = new Personne(-1, "Joachim", "Major",
22.           new SimpleDateFormat("dd/mm/yyyy").parse("13/11/1984"),
23.           true, 2);
24.       saveOne(p1);
25.       Personne p2 = new Personne(-1, "Mélanie", "Humbort",
26.           new SimpleDateFormat("dd/mm/yyyy").parse("12/02/1985"),
27.           false, 1);
28.       saveOne(p2);
29.       Personne p3 = new Personne(-1, "Charles", "Lemarchand",
30.           new SimpleDateFormat("dd/mm/yyyy").parse("01/03/1986"),
31.           false, 0);
32.       saveOne(p3);
33.     } catch (ParseException ex) {
34.       throw new DaoException(
35.           "Erreur d'initialisation de la couche [dao] : "
36.               + ex.toString(), 1);
37.     }
38.   }
39.
40.   // liste des personnes
41.   public Collection getAll() {
42.     return personnes;
43.   }
```

```
44.
45.   // obtenir une personne en particulier
46.   public Personne getOne(int id) {
47.     // on cherche la personne
48.     int i = getPosition(id);
49.     // a-t-on trouvé ?
50.     if (i != -1) {
51.       return new Personne(((Personne) personnes.get(i)));
52.     } else {
53.       throw new DaoException("Personne d'id [" + id + "] inconnue", 2);
54.     }
55.   }
56.
57.   // ajouter ou modifier une personne
58.   public void saveOne(Personne personne) {
59.     // le paramètre personne est-il valide ?
60.     check(personne);
61.     // ajout ou modification ?
62.     if (personne.getId() == -1) {
63.       // ajout
64.       personne.setId(getNextId());
65.       personne.setVersion(1);
66.       personnes.add(personne);
67.       return;
68.     }
69.     // modification - on cherche la personne
70.     int i = getPosition(personne.getId());
71.     // a-t-on trouvé ?
72.     if (i == -1) {
73.       throw new DaoException("La personne d'Id [" + personne.getId()
74.           + "] qu'on veut modifier n'existe pas", 2);
75.     }
76.     // a-t-on la bonne version de l'original ?
77.     Personne original = (Personne) personnes.get(i);
78.     if (original.getVersion() != personne.getVersion()) {
79.       throw new DaoException("L'original de la personne [" + personne
80.           + "] a changé depuis sa lecture initiale", 3);
81.     }
82.     // on attend 10 ms
83.     //wait(10);
84.     // c'est bon - on fait la modification
85.     original.setVersion(original.getVersion()+1);
86.     original.setNom(personne.getNom());
87.     original.setPrenom(personne.getPrenom());
88.     original.setDateNaissance((personne.getDateNaissance()));
89.     original.setMarie(personne.getMarie());
90.     original.setNbEnfants(personne.getNbEnfants());
91.   }
92.
93.   // suppression d'une personne
94.   public void deleteOne(int id) {
95.     // on cherche la personne
96.     int i = getPosition(id);
97.     // a-t-on trouvé ?
98.     if (i == -1) {
99.       throw new DaoException("Personne d'id [" + id + "] inconnue", 2);
100.    } else {
101.      // on supprime la personne
102.      personnes.remove(i);
103.    }
104.  }
105.
106.  // générateur d'id
107.  private int getNextId() {
108.    id++;
109.    return id;
110.  }
111.
112.  // rechercher une personne
113.  private int getPosition(int id) {
114.    int i = 0;
115.    boolean trouvé = false;
116.    // on parcourt la liste des personnes
117.    while (i < personnes.size() && !trouvé) {
118.      if (id == ((Personne) personnes.get(i)).getId()) {
119.        trouvé = true;
120.      } else {
121.        i++;
122.      }
123.    }
124.    // résultat ?
125.    return trouvé ? i : -1;
126.  }
127.
128.  // vérification d'une personne
```

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

```
129. private void check(Personne p) {
130.   // personne p
131.   if (p == null) {
132.     throw new DaoException("Personne null", 10);
133.   }
134.   // id
135.   if (p.getId() != -1 && p.getId() < 0) {
136.     throw new DaoException("Id [" + p.getId() + "] invalide", 11);
137.   }
138.   // date de naissance
139.   if (p.getDateNaissance() == null) {
140.     throw new DaoException("Date de naissance manquante", 12);
141.   }
142.   // nombre d'enfants
143.   if (p.getNbEnfants() < 0) {
144.     throw new DaoException("Nombre d'enfants [" + p.getNbEnfants()
145.         + "] invalide", 13);
146.   }
147.   // nom
148.   if (p.getNom() == null || p.getNom().trim().length() == 0) {
149.     throw new DaoException("Nom manquant", 14);
150.   }
151.   // prénom
152.   if (p.getPrenom() == null || p.getPrenom().trim().length() == 0) {
153.     throw new DaoException("Prénom manquant", 15);
154.   }
155. }
156.
157. // attente
158. private void wait(int N) {
159.   // on attend N ms
160.   try {
161.     Thread.sleep(N);
162.   } catch (InterruptedException e) {
163.     // on affiche la trace de l'exception
164.     e.printStackTrace();
165.     return;
166.   }
167. }
168.}
```

Nous n’allons donner que les grandes lignes de ce code. Nous passerons cependant un peu de temps sur les parties les plus délicates.

- ligne 13 : l’objet [ArrayList] qui va contenir le groupe de personnes
- ligne 16 : l’identifiant de la dernière personne ajoutée. A chaque nouvel ajout, cet identifiant va être incrémenté de 1.

La classe [DaoImpl] va être instanciée en un unique exemplaire. C’est ce qu’on appelle un singleton. Une application web sert ses utilisateurs de façon simultanée. Il y a à un moment donné plusieurs threads exécutés par le serveur web. Ceux-ci se partagent les singletons :

- celui de la couche [dao]
- celui de la couche [service]
- ceux des différents contrôleurs, validateurs de données, ... de la couche web

Si un singleton a des champs privés, il faut tout de suite se demander pourquoi il en a. Sont-ils justifiés ? En effet, ils vont être partagés entre différents threads. S’ils sont en lecture seule, cela ne pose pas de problème s’ils peuvent être initialisés à un moment où on est sûr qu’il n’y a qu’un thread actif. On sait en général trouver ce moment. C’est celui du démarrage de l’application web alors qu’elle n’a pas commencé à servir des clients. S’ils sont en lecture / écriture alors il faut mettre en place une synchronisation d’accès aux champs sinon on court à la catastrophe. Nous illustrerons ce problème lorsque nous testerons la couche [dao].

- la classe [DaoImpl] n’a pas de constructeur. C’est donc son constructeur par défaut qui sera utilisé.
- lignes 19-38 : la méthode [init] sera appelée au moment de l’instanciation du singleton de la couche [dao]. Elle crée une liste de trois personnes.
- lignes 41-43 : implémente la méthode [getAll] de l’interface [IDao]. Elle rend une référence sur la liste des personnes.
- lignes 46-55 : implémente la méthode [getOne] de l’interface [IDao]. Son paramètre est l’id de la personne cherchée.

  Pour la récupérer, on fait appel à une méthode privée [getPosition] des lignes 113-126. Cette méthode rend la position dans la liste, de la personne cherchée ou -1 si la personne n’a pas été trouvée.

  Si la personne a été trouvée, la méthode [getOne] rend une référence (ligne 51) sur une **copie** de cette personne et non sur la personne elle-même. En effet, lorsqu’un utilisateur va vouloir modifier une personne, les informations sur celle-ci vont-être demandées à la couche [dao] et remontées jusqu’à la couche [web] pour modification, sous la forme d’une référence sur un objet [Personne]. Cette référence va servir de conteneur de saisies dans le formulaire

de modification. Lorsque dans la couche web, l'utilisateur va poster ses modifications, le contenu du conteneur de saisies va être modifié. Si le conteneur est une référence sur la personne réelle du [ArrayList] de la couche [dao], alors celle-ci est modifiée alors même que les modifications n'ont pas été présentées aux couches [service] et [dao]. Cette dernière est la seule habilitée à gérer la liste des personnes. Aussi faut-il que la couche web travaille sur une copie de la personne à modifier. Ici la couche [dao] délivre cette copie.

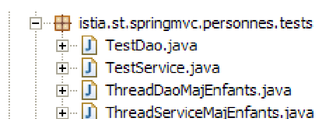
Si la personne cherchée n'est pas trouvée, une exception de type [DaoException] est lancée avec le code d'erreur 2 (ligne 53).

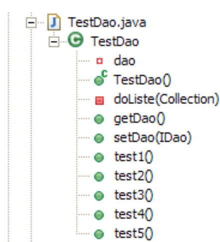
- lignes 94-104 : implémente la méthode [deleteOne] de l'interface [IDao]. Son paramètre est l'id de la personne à supprimer. Si la personne à supprimer n'existe pas, une exception de type [DaoException] est lancée avec le code d'erreur 2.
- lignes 58-91 : implémente la méthode [saveOne] de l'interface [IDao]. Son paramètre est un objet [Personne]. Si cet objet à un id=-1, alors il s'agit d'un ajout de personne. Sinon, il s'agit de modifier la personne de la liste ayant cet id avec les valeurs du paramètre.
    - ligne 60 : la validité du paramètre [Personne] est vérifiée par une méthode privée [check] définie aux lignes 129-155. Cette méthode fait des vérifications basiques sur la valeur des différents champs de [Personne]. A chaque fois qu'une anomalie est détectée, une [DaoException] avec un code d'erreur spécifique est lancé. Comme la méthode [saveOne] ne gère pas cette exception, elle remontera à la méthode appelante.
    - lignes 62 : si le paramètre [Personne] a son id égal à -1, alors il s'agit d'un ajout. L'objet [Personne] est ajouté à la liste interne des personnes (ligne 66), avec le 1er id disponible (ligne 64), et un n° de version égal à 1 (ligne 65).
    - si le paramètre [Personne] a un [id] différent de -1, il s'agit de modifier la personne de la liste interne ayant cet [id]. Tout d'abord, on vérifie (lignes 70-75) que la personne à modifier existe. Si ce n'est pas le cas, on lance une exception de type [DaoException] avec le code d'erreur 2.
    - si la personne est bien présente, on vérifie que sa version actuelle est la même que celle du paramètre [Personne] qui contient les modifications à apporter à l'original. Si ce n'est pas le cas, cela signifie que celui qui veut faire la modification de la personne n'en détient pas la dernière version. On le lui dit en lançant une exception de type [DaoException] avec le code d'erreur 3 (lignes 79-80).
    - si tout va bien, les modifications sont faites sur l'original de la personne (lignes 85-90)

  On sent bien que cette méthode doit être synchronisée. Par exemple, entre le moment où on vérifie que la personne à modifier est bien là et celui où la modification va être faite, la personne a pu être supprimée de la liste par quelqu'un d'autre. La méthode devrait être donc déclarée [synchronized] afin de s'assurer qu'un seul thread à la fois l'exécute. Il en est de même pour les autres méthodes de l'interface [IDao]. Nous ne le faisons pas, préférant déplacer cette synchronisation dans la couche [service]. Pour mettre en lumière les problèmes de synchronisation, lors des tests de la couche [dao] nous arrêterons l'exécution de [saveOne] pendant 10 ms (ligne 83) entre le moment où on sait qu'on peut faire la modification et le moment où on la fait réellement. Le thread qui exécute [saveOne] perdra alors le processeur au profit d'un autre. Nous augmentons ainsi nos chances de voir apparaître des conflits d'accès à la liste des personnes.

2.5 Tests de la couche [dao]

Un test JUnit est écrit pour la couche [dao] :



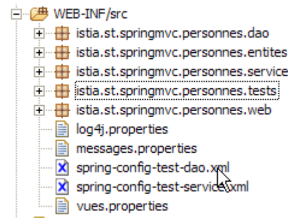


[TestDao] est le test JUnit. Pour mettre en évidence les problèmes d'accès concurrents à la liste des personnes, des threads de type [ThreadDaoMajEnfants] sont créés. Ils sont chargés d'augmenter de 1 le nombre d'enfants d'une personne donnée.

[TestDao] a cinq tests [test1] à [test5]. Nous ne présentons que deux d'entre-eux, le lecteur étant invité à découvrir les autres dans le code source associé à cet article.

```
1.  package istia.st.springmvc.personnes.tests;
2.
3.  import java.text.ParseException;
4.  import java.text.SimpleDateFormat;
5.  import java.util.Collection;
6.  import java.util.Iterator;
7.  import org.springframework.beans.factory.xml.XmlBeanFactory;
8.  import org.springframework.core.io.ClassPathResource;
9.
10. import istia.st.springmvc.personnes.dao.DaoException;
11. import istia.st.springmvc.personnes.dao.IDao;
12. import istia.st.springmvc.personnes.entites.Personne;
13. import junit.framework.TestCase;
14.
15. public class TestDao extends TestCase {
16.
17.   // couche [dao]
18.   private IDao dao;
19.
20.   public IDao getDao() {
21.     return dao;
22.   }
23.
24.   public void setDao(IDao dao) {
25.     this.dao = dao;
26.   }
27.
28.   // constructeur
29.   public TestDao() {
30.     dao = (IDao) (new XmlBeanFactory(new ClassPathResource(
31.         "spring-config-test-dao.xml"))).getBean("dao");
32.   }
33.
34.   // liste des personnes
35.   private void doListe(Collection personnes) {
36.     Iterator iter = personnes.iterator();
37.     while (iter.hasNext()) {
38.       System.out.println(iter.next());
39.     }
40.   }
41.
42.   // test1
43.   public void test1() throws ParseException {
44. ...
45.   }
46.
47.   // modification-suppression d'un élément inexistant
48.   public void test2() throws ParseException {
49. ...
50.   }
51.
52.   // gestion des versions de personne
53.   public void test3() throws ParseException, InterruptedException {
54. ...
55.   }
56.
57.   // optimistic locking - accès multi-threads
58.   public void test4() throws Exception {
59. ...
60.   }
61.
62.   // tests de validité de saveOne
63.   public void test5() throws ParseException {
64.   ...
65. }
```

- lignes 18-26 : la classe de test détient une référence de type [IDao] (ligne 18) sur la couche [dao]. C'est donc bien une interface qui est testée. La référence d'une implémentation concrète de celle-ci est obtenue dans le constructeur.
- lignes 29-32 : le constructeur du test JUnit. Il récupère la référence de l'implémentation de la couche [dao] à tester dans le fichier de configuration Spring [spring-config-test-dao.xml] :



Le fichier [spring-config-test-dao.xml] est dans [WEB-INF/src] parce que Spring va le chercher dans le *ClassPath* du projet Eclipse. Son contenu est le suivant :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- implémentation de la couche [dao] -->
5.    <bean id="dao" class="istia.st.springmvc.personnes.dao.DaoImpl" init-method="init"/>
6.  </beans>
```

On voit que le bean [dao] défini ligne 5 ci-dessus est celui utilisé par le constructeur de l'instance de test (ligne 31 du constructeur). La classe d'implémentation de l'interface [IDao] testée sera de type [DaoImpl]. Une fois cette instance construite, sa méthode [init] sera exécutée. On rappelle qu'elle met trois personnes arbitraires dans la liste des personnes.

La méthode [test1] teste les quatre méthodes de l'interface [IDao] de la façon suivante :

```
1.    public void test1() throws ParseException {
2.    // liste actuelle
3.    Collection personnes = dao.getAll();
4.    int nbPersonnes = personnes.size();
5.    // affichage
6.    doListe(personnes);
7.    // ajout d'une personne
8.    Personne p1 = new Personne(-1, "X", "X", new SimpleDateFormat(
9.        "dd/MM/yyyy").parse("01/02/2006"), true, 1);
10.   dao.saveOne(p1);
11.   int id1 = p1.getId();
12.   // vérification - on aura un plantage si la personne n'est pas trouvée
13.   p1 = dao.getOne(id1);
14.   assertEquals("X", p1.getNom());
15.   // modification
16.   p1.setNom("Y");
17.   dao.saveOne(p1);
18.   // vérification - on aura un plantage si la personne n'est pas trouvée
19.   p1 = dao.getOne(id1);
20.   assertEquals("Y", p1.getNom());
21.   // suppression
22.   dao.deleteOne(id1);
23.   // vérification
24.   int codeErreur = 0;
25.   boolean erreur = false;
26.   try {
27.     p1 = dao.getOne(id1);
28.   } catch (DaoException ex) {
29.     erreur = true;
30.     codeErreur = ex.getCode();
31.   }
32.   // on doit avoir une erreur de code 2
33.   assertTrue(erreur);
34.   assertEquals(2, codeErreur);
35.   // liste des personnes
36.   personnes = dao.getAll();
37.   assertEquals(nbPersonnes, personnes.size());
38. }
```

- ligne 3 : on demande la liste des personnes
- ligne 6 : on affiche celle-ci

```
[1,1,Joachim,Major,13/01/1984,true,2]
[2,1,Mélanie,Humbort,12/01/1985,false,1]
[3,1,Charles,Lemarchand,01/01/1986,false,0]
```

Le test ensuite ajoute une personne, la modifie et la supprime. Ainsi les quatre méthodes de l'interface [IDao] sont-elles utilisées.

- lignes 8-10 : on ajoute une nouvelle personne (id=-1).
- ligne 11 : on récupère l'id de la personne ajoutée car l'ajout lui en a donné un. Avant elle n'en avait pas.

- ligne 13-14 : on demande à la couche [dao] une copie de la personne qui vient d'être ajoutée. Il faut se rappeler que si la personne demandée n'est pas trouvée, la couche [dao] lance une exception. On aura alors un plantage ligne 13. On aurait pu gérer ce cas plus proprement. Ligne 14, on vérifie le nom de la personne retrouvée.
- lignes 16-17 : on modifie ce nom et on demande à la couche [dao] d'enregistrer les modifications.
- lignes 19-20 : on demande à la couche [dao] une copie de la personne qui vient d'être ajoutée et on vérifie son nouveau nom.
- ligne 22 : on supprime la personne ajoutée au début du test.
- lignes 23-34 : on demande à la couche [dao] une copie de la personne qui vient d'être supprimée. On doit obtenir une [DaoException] de code 2.
- lignes 36-37 : la liste des personnes est redemandée. On doit obtenir la même qu'au début du test.

La méthode [test4] cherche à mettre en lumière les problèmes d'accès concurrents aux méthodes de la couche [dao]. Rappelons que celles-ci n'ont pas été synchronisées. Le code du test est le suivant :

```
1.  public void test4() throws Exception {
2.    // ajout d'une personne
3.    Personne p1 = new Personne(-1, "X", "X", new SimpleDateFormat(
4.        "dd/MM/yyyy").parse("01/02/2006"), true, 0);
5.    dao.saveOne(p1);
6.    int id1 = p1.getId();
7.    // création de N threads de mise à jour du nombre d'enfants
8.    final int N = 10;
9.    Thread[] taches = new Thread[N];
10.   for (int i = 0; i < taches.length; i++) {
11.     taches[i] = new ThreadDaoMajEnfants("thread n° " + i, dao, id1);
12.     taches[i].start();
13.   }
14.   // on attend la fin des threads
15.   for (int i = 0; i < taches.length; i++) {
16.     taches[i].join();
17.   }
18.   // on récupère la personne
19.   p1 = dao.getOne(id1);
20.   // elle doit avoir N enfants
21.   assertEquals(N, p1.getNbEnfants());
22.   // suppression personne p1
23.   dao.deleteOne(p1.getId());
24.   // vérification
25.   boolean erreur = false;
26.   int codeErreur = 0;
27.   try {
28.     p1 = dao.getOne(p1.getId());
29.   } catch (DaoException ex) {
30.     erreur = true;
31.     codeErreur = ex.getCode();
32.   }
33.   // on doit avoir une erreur de code 2
34.   assertTrue(erreur);
35.   assertEquals(2, codeErreur);
36. }
```

- lignes 3-6 : on ajoute dans la liste une personne P avec aucun enfant. On note son [id] (ligne 6).
- lignes 7-13 : on lance N threads. Chacun d'eux va incrémenter le nombre d'enfants de la personne P de 1 unité. Au final, la personne P devra avoir N enfants.
- lignes 15-17 : la méthode [test4] qui a lancé les N threads attend qu'ils aient terminé leur travail avant de regarder le nouveau nombre d'enfants de la personne P.
- lignes 18-21 : on récupère la personne P et on vérifie que son nombre d'enfants est N.
- lignes 22-35 : la personne P est supprimée puis on vérifie qu'elle n'existe plus dans la liste.

Ligne 11, on voit que les threads sont de type [ThreadDaoMajEnfants]. Le constructeur de ce type a trois paramètres :

1. le nom donné au thread, ceci pour le suivre au moyen de logs
2. une référence sur la couche [dao] afin que le thread y ait accès
3. l'id de la personne sur laquelle le thread doit travailler

Le type [ThreadDaoMajEnfants] est le suivant :

```
1.  package istia.st.springmvc.personnes.tests;
2.
3.  import istia.st.springmvc.personnes.dao.DaoException;
4.  import istia.st.springmvc.personnes.dao.IDao;
5.  import istia.st.springmvc.personnes.entites.Personne;
6.
7.  public class ThreadDaoMajEnfants extends Thread {
8.    // nom du thread
9.    private String name;
10.   // référence sur la couche [dao]
```

```
11.  private IDao dao;
12.  // l'id de la personne sur qui on va travailler
13.  private int idPersonne;
14.
15.  // constructeur
16.  public ThreadDaoMajEnfants(String name, IDao dao, int idPersonne) {
17.    this.name = name;
18.    this.dao = dao;
19.    this.idPersonne = idPersonne;
20.  }
21.
22.  // coeur du thread
23.  public void run() {
24.    // suivi
25.    suivi("lancé");
26.    // on boucle tant qu'on n'a pas réussi à incrémenter de 1
27.    // le nbre d'enfants de la personne idPersonne
28.    boolean fini = false;
29.    int nbEnfants = 0;
30.    while (!fini) {
31.      // on récupère une copie de la personne d'idPersonne
32.      Personne personne = dao.getOne(idPersonne);
33.      nbEnfants = personne.getNbEnfants();
34.      // suivi
35.      suivi("" + nbEnfants + " -> " + (nbEnfants + 1) + " pour la version "+personne.getVersion());
36.      // attente de 10 ms pour abandonner le processeur
37.      try {
38.        // suivi
39.        suivi("début attente");
40.        // on s'interrompt pour laisser le processeur
41.        Thread.sleep(10);
42.        // suivi
43.        suivi("fin attente");
44.      } catch (Exception ex) {
45.        throw new RuntimeException(ex.toString());
46.      }
47.      // attente terminée - on essaie de valider la copie
48.      // entre-temps d'autres threads ont pu modifier l'original
49.      int codeErreur = 0;
50.      try {
51.        // incrémente de 1 le nbre d'enfants de cette copie
52.        personne.setNbEnfants(nbEnfants + 1);
53.        // on essaie de modifier l'original
54.        dao.saveOne(personne);
55.        // on est passé - l'original a été modifié
56.        fini = true;
57.      } catch (DaoException ex) {
58.        // on récupère le code erreur
59.        codeErreur = ex.getCode();
60.        // doit être une erreur de version 3 - sinon on relance
61.        // l'exception
62.        if (codeErreur != 3) {
63.          throw ex;
64.        } else {
65.          // suivi
66.          suivi(ex.getMessage());
67.        }
68.        // l'original a changé - on recommence tout
69.      }
70.    }
71.    // suivi
72.    suivi("a terminé et passé le nombre d'enfants à " + (nbEnfants + 1));
73.  }
74.
75.  // suivi
76.  private void suivi(String message) {
77.    System.out.println(name + " [" + new Date().getTime()+ "] : " + message);  }
78. }
```

- ligne 7 : [ThreadDaoMajEnfants] est bien un thread
- lignes 16-20 : le constructeur qui initialise le thread avec trois informations
  1. le nom [name] donné au thread
  2. une référence [dao] sur la couche [dao]. On notera qu'une nouvelle fois, nous travaillons avec le type de l'interface [IDao] et non celui de l'implémentation [DaoImpl].
  3. l'identifiant [id] de la personne sur laquelle le thread doit travailler

Lorsque [test4] lance un thread [ThreadDaoMajEnfants] (ligne 12 de test4), la méthode [run] (ligne 23) de celui-ci est exécutée :

- lignes 76-78 : la méthode privée [suivi] permet de faire des logs écran. La méthode [run] en use pour permettre le suivi du thread dans son exécution.

- le thread va chercher à incrémenter de 1 le nombre d'enfants de la personne P d'identifiant [id]. Cette mise à jour peut nécessiter plusieurs tentatives. Prenons deux threads [TH1] et [TH2]. [TH1] demande une copie de la personne P à la couche [dao]. Il l'obtient et constate qu'elle a la version V1. [TH1] est interrompu. [TH2] qui le suivait fait la même chose et obtient la même version V1 de la personne P. [TH2] est interrompu. [TH2] reprend la main, incrémente le nombre d'enfants de P et sauvegarde ses modifications. Nous savons qu'alors, celles-ci sont sauvegardées et que la version de P va passer à V2. [TH1] a fini son travail. [TH2] reprend la main et fait de même. Sa mise à jour de P sera refusée car il détient une copie de P de version V1 alors que l'original P a désormais la version V2. [TH2] doit alors reprendre tout le cycle [lecture -> mise à jour -> sauvegarde]. C'est pourquoi, nous trouvons la boucle des lignes 30-70. Dans celle-ci, le thread :
- demande une copie de la personne P à modifier (ligne 32)
- attend 10 ms (ligne 41). Ceci est artificiel et vise à interrompre le thread entre la lecture de la personne P et sa mise à jour effective dans la liste des personnes afin d'augmenter la probabilité de conflits.
- incrémente le nombre d'enfants de P (ligne 52) et sauvegarde P (ligne 54). Si le thread n'a pas la bonne version de P, une exception sera déclenchée par la couche [dao]. On récupère alors le code de l'exception (ligne 59) pour vérifier que c'est bien le code 3 (mauvaise version de P). Si ce n'est pas le cas, on relance l'exception à destination de la méthode appelante, au final la méthode de test [test4]. Si on a l'exception de code 3, alors on recommence le cycle [lecture -> mise à jour -> sauvegarde]. Si on n'a pas d'exception, alors la mise à jour a été faite et le travail du thread est terminé.

Que donnent les tests ?

Dans la première configuration testée :

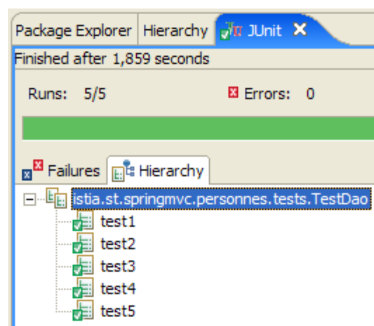
- on commente l'instruction d'attente dans la méthode [saveOne] de [DaoImpl] (ligne 83, page 9).

```
// on attend 10 ms
//wait(10);
```

- la méthode [test4] crée 100 threads (ligne 8, page 14).

```
// création de N threads de mise à jour du nombre d'enfants
final int N = 100;
```

On obtient les résultats suivants :



Les cinq tests ont été réussis.

Dans la seconde configuration testée :

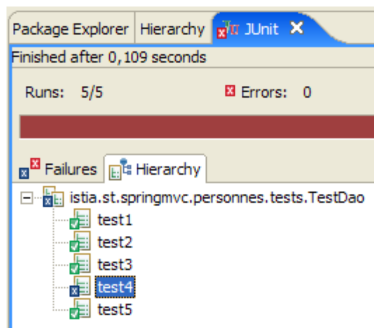
- on décommente l'instruction d'attente dans la méthode [saveOne] de [DaoImpl] (ligne 83, page 9).

```
// on attend 10 ms
wait(10);
```

- la méthode [test4] crée 2 threads (ligne 8, page 14).

```
// création de N threads de mise à jour du nombre d'enfants
final int N = 2;
```

On obtient les résultats suivants :




```
junit.framework.AssertionFailedError: expected:<2> but was:<1>
at istia.st.springmvc.personnes.tests.TestDao.test4(TestDao.java:199)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
```

Le test [test4] a échoué. On a créé deux threads chargés chacun d'incrémenter de 1 le nombre d'enfants d'une personne P qui au départ en avait 0. On attendait donc 2 enfants après exécution des deux threads, or on n'en a qu'un.

Suivons les logs écran de [test4] pour comprendre ce qui s'est passé :

```
1.  thread n° 0 [1145536368171] : lancé
2.  thread n° 0 [1145536368171] : 0 -> 1 pour la version 1
3.  thread n° 0 [1145536368171] : début attente
4.  thread n° 1 [1145536368171] : lancé
5.  thread n° 1 [1145536368171] : 0 -> 1 pour la version 1
6.  thread n° 1 [1145536368171] : début attente
7.  thread n° 0 [1145536368187] : fin attente
8.  thread n° 1 [1145536368187] : fin attente
9.  thread n° 0 [1145536368187] : a terminé et passé le nombre d'enfants à 1
10. thread n° 1 [1145536368187] : a terminé et passé le nombre d'enfants à 1
```

- ligne 1 : le thread n° 0 commence son travail
- ligne 2 : il a récupéré une copie de la personne P et trouve son nombre d'enfants à 0
- ligne 3 : il rencontre le [Thread.sleep(10)] de sa méthode [run] et s'arrête donc au temps [1145536368171] (ms)
- ligne 4 : le thread n° 1 récupère alors le processeur et commence son travail
- ligne 5 : il a récupéré une copie de la personne P et trouve son nombre d'enfants à 0
- ligne 6 : il rencontre le [Thread.sleep(10)] de sa méthode [run] et s'arrête donc
- ligne 7 : le thread n° 0 récupère le processeur au temps [1145536368187] (ms), c.a.d. 16 ms après l'avoir perdu.
- ligne 8 : idem pour le thread n° 1
- ligne 9 : le thread n° 0 a fait sa mise à jour et passé le nombre d'enfants à 1
- ligne 10 : le thread n° 1 a fait de même

La question est de savoir pourquoi le thread n° 1 a-t-il pu faire sa mise à jour alors que normalement il ne détenait plus la bonne version de la personne P qui venait d'être mise à jour par le thread n° 0.

Tout d'abord, on peut remarquer une anomalie entre les lignes 7 et 8 : il semblerait que le thread n° 0 ait perdu le processeur entre ces deux lignes au profit du thread n° 1. Que faisait-il à ce moment ? Il exécutait la méthode [saveOne] de la couche [dao]. Celle-ci a le squelette suivant (cf page 9) :

```
1. public void saveOne(Personne personne) {
2....
3.   // modification - on cherche la personne
4.....
5.   // a-t-on la bonne version de l'original ?
6....
7.   // on attend 10 ms
8.   wait(10);
9.   // c'est bon - on fait la modification
10. ...
11.}
```

- le thread n° 0 a exécuté [saveOne] et est allé jusqu'à la ligne 8 où là, il a été obligé de lâcher le processeur. Entre-temps, il a lu la version de la personne P et c'était 1 parce que la personne P n'avait pas encore été mise à jour.
- le processeur étant devenu libre, c'est le thread n° 1 qui en a hérité. Il a, à son tour, exécuté [saveOne] et est allé jusqu'à la ligne 8 où là il a été obligé de lâcher le processeur. Entre-temps, il a lu la version de la personne P et c'était 1 parce que la personne P n'avait toujours pas été mise à jour.
- le processeur étant devenu libre, c'est le thread n° 0 qui en a hérité. A partir de la ligne 9, il a fait sa mise à jour et passé le nombre d'enfants à 1. Puis la méthode [run] du thread n° 0 s'est terminée et le thread a affiché le log qui disait qu'il avait passé le nombre d'enfants à 1 (ligne 9).
- le processeur étant devenu libre, c'est le thread n° 1 qui en a hérité. A partir de la ligne 9, il a fait sa mise à jour et passé le nombre d'enfants à 1. Pourquoi 1 ? Parce qu'il détient une copie de P avec un nombre d'enfants à 0. C'est le

log (ligne 5) qui le dit. Puis la méthode [run] du thread n° 1 s'est terminée et le thread a affiché le log qui disait qu'il avait passé le nombre d'enfants à 1 (ligne 10).

D'où vient le problème ? Il vient du fait que le thread n° 0 n'a pas eu le temps de valider sa modification et donc de changer la version de la personne P avant que le thread n° 1 n'essaie de lire cette version pour savoir si la personne P avait changé. Ce cas de figure est peu probable mais pas impossible. Il a fallu forcer le thread n° 0 à perdre le processeur pour le faire apparaître avec simplement deux threads. Sans cet artifice, la configuration précédente n'avait pas réussi à faire apparaître ce même cas avec 100 threads. Le test [test4] avait été réussi.

Quelle est la solution ? Il y en a sans doute plusieurs. L'une d'elles, simple à mettre en oeuvre, est de synchroniser la méthode [saveOne] :

```
public synchronized void saveOne(Personne personne)
```

Le mot clé [synchronized] assure qu'un seul thread à la fois peut exécuter la méthode. Ainsi le thread n° 1 ne sera-t-il autorisé à exécuter [saveOne] que lorsque le thread n° 0 en sera sorti. On est alors sûr que la version de la personne P aura été changée lorsque le thread n° 1 va entrer dans [saveOne]. Sa mise à jour sera alors refusée car il n'aura pas la bonne version de P.

Ce sont les quatres méthodes de la couche [dao] qu'il faudrait synchroniser. Nous décidons cependant de garder cette couche telle qu'elle a été décrite et de reporter la synchronisation sur la couche [service]. A cela plusieurs raisons :

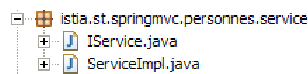
- nous faisons l'hypothèse que l'accès à la couche [dao] se fait toujours au travers d'une couche [service]. C'est le cas dans notre application web.
- il peut être nécessaire de synchroniser également l'accès aux méthodes de la couche [service] pour d'autres raisons que celles qui nous feraient synchroniser celles de la couche [dao]. Dans ce cas, il est inutile de synchroniser les méthodes de la couche [dao]. Si on est assurés que :
  - tout accès à la couche [dao] passe par la couche [service]
  - qu'un unique thread à la fois utilise la couche [service]

  alors on est assurés que les méthodes de la couche [dao] ne seront pas exécutés par deux threads en même temps.

Nous découvrons maintenant la couche [service].

2.6 La couche [service]

La couche [service] est constituée des classes et interfaces suivantes :



- [IService] est l'interface présentée par la couche [dao]
- [ServiceImpl] est une implémentation de celle-ci

L'interface [IService] est la suivante :

```
1.  package istia.st.springmvc.personnes.service;
2.
3.  import istia.st.springmvc.personnes.entites.Personne;
4.
5.  import java.util.Collection;
6.
7.  public interface IService {
8.    // liste de toutes les personnes
9.    Collection getAll();
10.   // obtenir une personne particulière
11.   Personne getOne(int id);
12.   // ajouter/modifier une personne
13.   void saveOne(Personne personne);
14.   // supprimer une personne
15.   void deleteOne(int id);
16. }
```

Elle est identique à l'interface [IDao].

L'implémentation [ServiceImpl] de l'interface [IService] est la suivante :

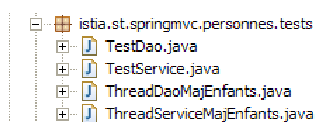
```
1.  package istia.st.springmvc.personnes.service;
2.
3.  import istia.st.springmvc.personnes.dao.IDao;
```

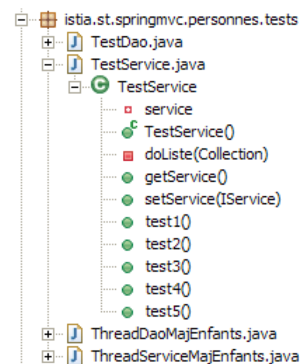
```
4.  import istia.st.springmvc.personnes.entites.Personne;
5.
6.  import java.util.Collection;
7.
8.  public class ServiceImpl implements IService {
9.
10.   // la couche [dao]
11.   private IDao dao;
12.
13.   public IDao getDao() {
14.     return dao;
15.   }
16.
17.   public void setDao(IDao dao) {
18.     this.dao = dao;
19.   }
20.
21.   // liste des personnes
22.   public synchronized Collection getAll() {
23.     return dao.getAll();
24.   }
25.
26.   // obtenir une personne en particulier
27.   public synchronized Personne getOne(int id) {
28.     return dao.getOne(id);
29.   }
30.
31.   // ajouter ou modifier une personne
32.   public synchronized void saveOne(Personne personne) {
33.     dao.saveOne(personne);
34.   }
35.
36.   // suppression d'une personne
37.   public synchronized void deleteOne(int id) {
38.     dao.deleteOne(id);
39.   }
40. }
```

- lignes 10-19 : l'attribut [IDao dao] est une référence sur la couche [dao]. Il sera initialisé par Spring IoC.
- lignes 22-24 : implémentation de la méthode [getAll] de l'interface [IService]. La méthode se contente de déléguer la demande à la couche [dao].
- lignes 27-29 : implémentation de la méthode [getOne] de l'interface [IService]. La méthode se contente de déléguer la demande à la couche [dao].
- lignes 32-34 : implémentation de la méthode [saveOne] de l'interface [IService]. La méthode se contente de déléguer la demande à la couche [dao].
- lignes 37-39 : implémentation de la méthode [deleteOne] de l'interface [IService]. La méthode se contente de déléguer la demande à la couche [dao].
- toutes les méthodes sont synchronisées (mot clé synchronized) assurant qu'un seul thread à la fois pourra utiliser la couche [service] et donc la couche [dao].

2.7 Tests de la couche [service]

Un test JUnit est écrit pour la couche [service] :



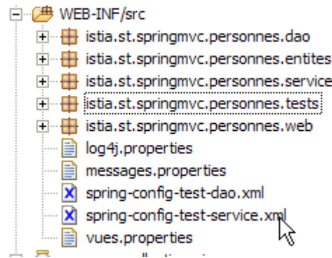


[TestService] est le test JUnit. Les tests faits sont strictement identiques à ceux faits pour la couche [dao]. Le squelette de [TestService] est le suivant :

```
1.  package istia.st.springmvc.personnes.tests;
2.
```

```
3.  ...
4.
5.  public class TestService extends TestCase {
6.
7.   // couche [service]
8.   private IService service;
9.
10.  public IService getService() {
11.    return service;
12.  }
13.
14.  public void setService(IService service) {
15.    this.service = service;
16.  }
17.
18.  // constructeur
19.  public TestService() {
20.    service = (IService) (new XmlBeanFactory(new ClassPathResource(
21.        "spring-config-test-service.xml"))).getBean("service");
22.  }
23.
24.  // liste des personnes
25.  private void doListe(Collection personnes) {
26. ...
27.  }
28.
29.  // test1
30.  public void test1() throws ParseException {
31.    // liste actuelle
32.    Collection personnes = service.getAll();
33.    int nbPersonnes = personnes.size();
34.    // affichage
35.    doListe(personnes);
36.    // ajout d'une personne
37.    Personne p1 = new Personne(-1, "X", "X", new SimpleDateFormat(
38.        "dd/MM/yyyy").parse("01/02/2006"), true, 1);
39.    service.saveOne(p1);
40.    int id1 = p1.getId();
41.    // vérification - on aura un plantage si la personne n'est pas trouvée
42.    p1 = service.getOne(id1);
43.    assertEquals("X", p1.getNom());
44. ...
45.  }
46.
47.  // modification-suppression d'un élément inexistant
48.  public void test2() throws ParseException {
49. ...
50.  }
51.
52.  // gestion des versions de personne
53.  public void test3() throws ParseException, InterruptedException {
54. ...
55.  }
56.
57.  // optimistic locking - accès multi-threads
58.  public void test4() throws Exception {
59.    // ajout d'une personne
60.    Personne p1 = new Personne(-1, "X", "X", new SimpleDateFormat(
61.        "dd/MM/yyyy").parse("01/02/2006"), true, 0);
62.    service.saveOne(p1);
63.    int id1 = p1.getId();
64.    // création de N threads de mise à jour du nombre d'enfants
65.    final int N = 100;
66.    Thread[] taches = new Thread[N];
67.    for (int i = 0; i < taches.length; i++) {
68.      taches[i] = new ThreadServiceMajEnfants("thread n° " + i, service,
69.          id1);
70.      taches[i].start();
71.    }
72. ...
73.  }
74.
75.  // tests de validité de saveOne
76.  public void test5() throws ParseException {
77.  ...
78.  }
79. }
```

- lignes 8-16 : la classe de test détient une référence de type [IService] (ligne 8) sur la couche [service]. C’est donc bien une interface qui est testée. La référence d’une implémentation concrète de celle-ci est obtenue dans le constructeur.
- lignes 19-22 : le constructeur du test JUnit. Il récupère la référence de l’implémentation de la couche [service] à tester dans le fichier de configuration Spring [spring-config-test-service.xml] :



Le fichier [spring-config-test-service.xml] est dans [WEB-INF/src] parce que Spring va le chercher dans le *ClassPath* du projet Eclipse. Son contenu est le suivant :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.   <!-- implémentation de la couche [dao] -->
5.   <bean id="dao" class="istia.st.springmvc.personnes.dao.DaoImpl" init-method="init"/>
6.   <!-- implémentation de la couche service -->
7.   <bean id="service" class="istia.st.springmvc.personnes.service.ServiceImpl">
8.     <property name="dao">
9.       <ref local="dao"/>
10.    </property>
11.  </bean>
12. </beans>
```

L'instance de la couche [dao] est définie ligne 5. Elle est de type [DaoImpl]. L'instance de la couche [service] est elle définie lignes 7-11. Elle est de type [ServiceImpl]. Ce type a un attribut [dao] dont la valeur doit être une référence de la couche [dao]. Cet attribut est initialisé lignes 8-10 avec la référence de la couche [dao] créée ligne 5. Au final, la couche [service] est implémentée avec une instance [ServiceImpl] détenant une référence sur la couche [dao] elle-même implémentée par une instance [DaoImpl].

La méthode [test1] teste les quatre méthodes de l'interface [IService] de façon identique à la méthode de test de la couche [dao] de même nom. Simplement, on accède à la couche [service] (lignes 39, 42) plutôt qu'à la couche [dao].

La méthode [test4] cherche à mettre en lumière les problèmes d'accès concurrents aux méthodes de la couche [service]. Elle est, là encore, identique à la méthode de test [test4] de la couche [dao]. Il y a cependant quelques détails qui changent :

- on s'adresse à la couche [service] plutôt qu'à la couche [dao] (ligne 62)
- on passe aux threads une référence à la couche [service] plutôt qu'à la couche [dao] (ligne 68)

Le type [ThreadServiceMajEnfants] est lui aussi quasi identique au type [ThreadDaoMajEnfants] au détail près qu'il travaille avec la couche [service] et non la couche [dao] :

```
1.  package istia.st.springmvc.personnes.tests;
2.
3.  import istia.st.springmvc.personnes.dao.DaoException;
4.  import istia.st.springmvc.personnes.entites.Personne;
5.  import istia.st.springmvc.personnes.service.IService;
6.
7.  public class ThreadServiceMajEnfants extends Thread {
8.
9.   // nom du thread
10.  private String name;
11.  // référence sur la couche [service]
12.  private IService service;
13.  // l'id de la personne sur qui on va travailler
14.  private int idPersonne;
15.
16.  public ThreadServiceMajEnfants(String name, IService service, int idPersonne) {
17.    this.name = name;
18.    this.service = service;
19.    this.idPersonne = idPersonne;
20.  }
21.
22.  public void run() {
23. ...
24.  }
25.
26.  // suivi
27.  private void suivi(String message) {
28.    System.out.println(name + " : " + message);
29.  }
30.
31. }
```

- ligne 12 : le thread travaille avec la couche [service]

Nous faisons les tests avec la configuration qui a posé problème à la couche [dao] :

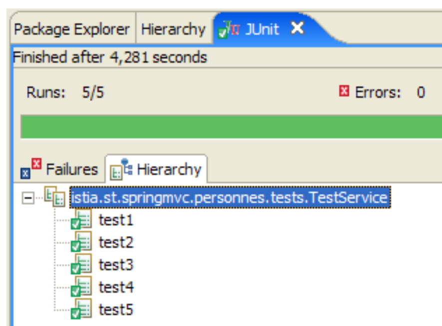
- on décommente l'instruction d'attente dans la méthode [saveOne] de [DaoImpl] (ligne 83, page 9).

```
    // on attend 10 ms
    wait(10);
```

- la méthode [test4] crée 100 threads (ligne 65, page 20).

```
    // création de N threads de mise à jour du nombre d'enfants
    final int N = 100;
```

Les résultats obtenus sont les suivants :




```
thread n° 93 [1145541451687] : 98 -> 99 pour la version 99
thread n° 93 [1145541451687] : début attente
thread n° 44 [1145541451687] : fin attente
thread n° 93 [1145541451687] : fin attente
thread n° 93 [1145541451703] : L'original de la personne
[[4,99,X,X,01/02/2006,true,99]] a changé depuis sa lecture initiale
thread n° 93 [1145541451703] : 99 -> 100 pour la version 100
thread n° 93 [1145541451703] : début attente
thread n° 44 [1145541451703] : a terminé et passé le nombre d'enfants à 99
thread n° 93 [1145541451718] : fin attente
thread n° 93 [1145541451718] : a terminé et passé le nombre d'enfants à 100
```

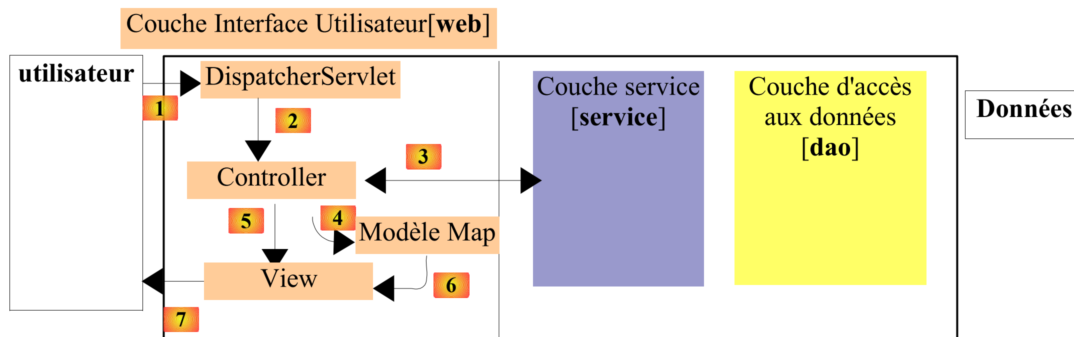
Les dernières lignes des logs écran

Tous les tests ont été réussis

C'est la synchronisation des méthodes de la couche [service] qui a permis le succès du test [test4].

2.8 La couche [web]

Rappelons l'architecture 3tier de notre application :



La couche [web] va offrir des écrans à l'utilisateur pour lui permettre de gérer le groupe de personnes :

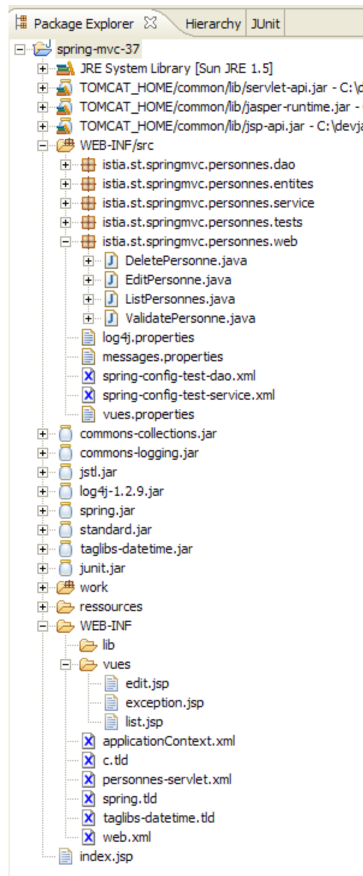
- liste des personnes du groupe
- ajout d'une personne au groupe
- modification d'une personne du groupe
- suppression d'une personne du groupe

Pour cela, elle va s'appuyer sur la couche [service] qui elle même fera appel à la couche [dao]. Nous avons déjà présenté les écrans gérés par la couche [web] (page 3). Pour décrire la couche web, nous allons présenter successivement :

- sa configuration
- ses contrôleurs
- ses vues
- quelques tests

2.8.1 Configuration de l'application web

Le projet Eclipse / Tomcat de l'application est le suivant :



- dans le paquetage [istia.st.springmvc.personnes.web], on trouve les contrôleurs et le validateur du formulaire d'ajout / modification d'une personne
- le résolveur de vues sera une instance de [ResourceBundleViewResolver]. Les vues seront définies dans le fichier [vues.properties] dans [WEB-INF/src]. Les pages JSP / JSTL associées sont dans [WEB-INF/vues].
- les messages d'erreurs sont définis dans [messages.properties] dans [WEB-INF/src].
- le dossier [lib] contient les archives tierces nécessaires à l'application. Elles sont visibles sur la copie d'écran ci-dessus.

[web.xml]

Le fichier [web.xml] est le fichier exploité par le serveur web pour charger l'application. Son contenu est le suivant :

```
1.  <?xml version="1.0" encoding="ISO-8859-1"?>
2.
3.  <!DOCTYPE web-app PUBLIC
4.    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5.    "http://java.sun.com/dtd/web-app_2_3.dtd">
6.  <web-app>
7.    <!-- le chargeur du contexte spring de l'application -->
8.    <listener>
9.      <listener-class>
10.       org.springframework.web.context.ContextLoaderListener</listener-class>
11.   </listener>
12.   <!-- la servlet -->
13.   <servlet>
14.     <servlet-name>personnes</servlet-name>
15.     <servlet-class>
16.       org.springframework.web.servlet.DispatcherServlet</servlet-class>
17.   </servlet>
18.   <!-- le mapping des url -->
19.   <servlet-mapping>
20.     <servlet-name>personnes</servlet-name>
21.     <url-pattern>*.html</url-pattern>
22.   </servlet-mapping>
23.   <!-- le document d'entrée -->
```

```
24.   <welcome-file-list>
25.     <welcome-file>index.jsp</welcome-file>
26.   </welcome-file-list>
27. </web-app>
```

- lignes 19-22 : les url [*.html] seront dirigées vers la servlet [personnes]
- lignes 13-17 : la servlet [personnes] est une instance de la classe [DispatcherServlet] de Spring. Les url [*.html] seront donc traitées par [DispatcherServlet].
- lignes 8-11 : le listener [ContextLoaderListener] est chargé afin d'exploiter le fichier [WEB-INF/applicationContext.xml]
- lignes 24-26 : l'application a une page d'entrée par défaut [index.jsp] qui se trouve à la racine du dossier de l'application web.

[index.jsp]

Cette page est présentée si un utilisateur demande directement le contexte de l'application sans préciser d'url, c.a.d. ici [/spring-mvc-37]. Son contenu est le suivant :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.
4.  <c:redirect url="/list.html"/>
```

[index.jsp] redirige le client vers l'url [/spring-mvc-37/list.html]. Cette url affiche la liste des personnes du groupe.

[applicationContext.xml]

Ce fichier est exploité par le listener [ContextLoaderListener] avant même que [DispatcherServlet] ne soit instancié. Son contenu est le suivant :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- implémentation de la couche [dao] -->
5.    <bean id="dao" class="istia.st.springmvc.personnes.dao.DaoImpl" init-method="init"/>
6.    <!-- implémentation de la couche service -->
7.    <bean id="service" class="istia.st.springmvc.personnes.service.ServiceImpl">
8.      <property name="dao">
9.        <ref local="dao"/>
10.     </property>
11.   </bean>
12. </beans>
```

[ContextLoaderListener] va instancier les beans de ce fichier. Ils correspondent aux couches [dao] (ligne 5) et [service] (lignes 7-11). Notre application web va donc utiliser une couche [dao] de type [DaoImpl] et une couche [service] de type [ServiceImpl]. Ce sont les deux couches que nous venons d'étudier et de tester.

[personnes-servlet.xml]

Ce fichier est exploité par [DispatcherServlet] une fois celui-ci instancié, mais avant qu'il ne serve des requêtes. Il configure la couche [web] alors que le fichier [applicationContext.xml] configure les couches [dao] et [service]. Les beans déjà instanciés dans [applicationContext.xml] sont disponibles lors de l'instanciation des beans de [personnes-servlet.xml]. Le contenu de ce fichier est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
    beans.dtd">
3.  <beans>
4.    <!-- les mappings de l'application-->
5.    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
6.      <property name="mappings">
7.        <props>
8.          <prop key="/list.html">Personnes.ListController</prop>
9.          <prop key="/delete.html">Personnes.DeleteController</prop>
10.         <prop key="/edit.html">Personnes.EditController</prop>
11.       </props>
12.     </property>
13.   </bean>
14.   <!-- LES CONTROLEURS -->
15.   <bean id="Personnes.ListController"
16.     class="istia.st.springmvc.personnes.web.ListPersonnes">
17.     <property name="service">
18.       <ref bean="service"/>
19.     </property>
20.   </bean>
```

```
21.   <bean id="Personnes.DeleteController"
22.     class="istia.st.springmvc.personnes.web.DeletePersonne">
23.     <property name="service">
24.       <ref bean="service"/>
25.     </property>
26.   </bean>
27.   <bean id="Personnes.EditController"
28.     class="istia.st.springmvc.personnes.web.EditPersonne">
29.     <property name="sessionForm">
30.       <value>true</value>
31.     </property>
32.     <property name="commandName">
33.       <value>personne</value>
34.     </property>
35.     <property name="validator">
36.       <ref bean="Personnes.Validator"/>
37.     </property>
38.     <property name="formView">
39.       <value>edit</value>
40.     </property>
41.     <property name="service">
42.       <ref bean="service"/>
43.     </property>
44.   </bean>
45.   <!-- le validateur -->
46.   <bean id="Personnes.Validator"
47.     class="istia.st.springmvc.personnes.web.ValidatePersonne"/>
48.   <!-- le résolveur de vues -->
49.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
50.     <property name="basename">
51.       <value>vues</value>
52.     </property>
53.   </bean>
54.   <!-- le gestionnaire d'exceptions -->
55.   <bean
56.     class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
57.     <property name="exceptionAttribute">
58.       <value>exception</value>
59.     </property>
60.     <property name="defaultStatusCode">
61.       <value>200</value>
62.     </property>
63.     <property name="defaultErrorView">
64.       <value>exception</value>
65.     </property>
66.   </bean>
67.   <!-- le fichier des messages -->
68.   <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
69.     <property name="basename">
70.       <value>messages</value>
71.     </property>
72.   </bean>
73. </beans>
```

- lignes 5-13 : l’application n’accepte que trois url [/list.html, /delete.html, /edit.html]
- ligne 8 : l’url [/list.html] est traitée par le contrôleur [Personnes.ListController] défini lignes 15-20. Ce contrôleur gère une demande de type GET.
- ligne 9 : l’url [/delete.html] est traitée par le contrôleur [Personnes.DeleteController] défini lignes 21-26. Ce contrôleur gère une demande de type GET.
- ligne 10 : l’url [/edit.html] est traitée par le contrôleur [Personnes.EditController] défini lignes 27-44. Ce contrôleur gère un formulaire et est dérivé de [SimpleFormController]. Le formulaire est validé par le validateur de données défini lignes 46-47.
- les trois contrôleurs ont besoin de la couche [service] pour remplir leur mission. Aussi leur injecte-t-on (lignes 17-19, 23-25, 41-43) le bean [service] défini dans [applicationContext.xml].
- lignes 49-53 : le résolveur de vues est [ResourceBundleViewResolver]. Les vues seront définies dans [vues.properties].
- lignes 55-66 : le gestionnaire d’exceptions. Il est là au cas où une exception non gérée par l’application remonterait jusqu’à [DispatcherServlet]. C’est alors la vue par défaut " exception " qui sera affichée (ligne 64).
- lignes 68-72 : le gestionnaire des messages est [ResourceBundleMessageSource]. Les messages seront définis dans [messages.properties].

2.8.2 Les vues de l’application web

Les vues de l’application sont définies dans [vues.properties] :

```
1.  #list
2.  list.class=org.springframework.web.servlet.view.JstlView
```

```
3.  list.url=/WEB-INF/vues/list.jsp
4.  #r-list
5.  r-list.class=org.springframework.web.servlet.view.RedirectView
6.  r-list.url=/list.html
7.  r-list.contextRelative=true
8.  r-list.http10Compatible=false
9.  #edit
10. edit.class=org.springframework.web.servlet.view.JstlView
11. edit.url=/WEB-INF/vues/edit.jsp
12. #exception
13. exception.class=org.springframework.web.servlet.view.JstlView
14. exception.url=/WEB-INF/vues/exception.jsp
```

- lignes 2-3 : la vue nommée "list" est associée à la page JSP / JSTL [/WEB-INF/vues/list.jsp]
- lignes 10-11 : la vue nommée "edit" est associée à la page JSP / JSTL [/WEB-INF/vues/edit.jsp]
- lignes 13-14 : la vue nommée "exception" est associée à la page JSP / JSTL [/WEB-INF/vues/exception.jsp]
- lignes 5-8 : la vue nommée "r-list" est associée à une redirection vers l'url [/list.html] (ligne 6) relative au contexte (ligne 7). Ce sera donc l'url [/spring-mvc-37/list.html].

La vue [list.jsp]

Elle sert à afficher la liste des personnes :

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	1	Charles	Lemarchand	01/01/1986	false	0	Modifier	Supprimer

Ajout

Son code est le suivant :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ taglib uri="/WEB-INF/taglibs-datetime.tld" prefix="dt" %>
4.  <%@ page isELIgnored="false" %>
5.
6.  <html>
7.    <head>
8.      <title>Spring MVC - personnes</title>
9.    </head>
10.   <body background="<c:url value="/ressources/standard.jpg"/>">
11.     <h2>Liste des personnes</h2>
12.     <table border="1">
13.       <tr>
14.         <th>Id</th>
15.         <th>Version</th>
16.         <th>Pr&eacute;nom</th>
17.         <th>Nom</th>
18.         <th>Date de naissance</th>
19.         <th>Mari&eacute;</th>
20.         <th>Nombre d'enfants</th>
21.         <th></th>
22.       </tr>
23.       <c:forEach var="personne" items="${personnes}">
24.         <tr>
25.           <td><c:out value="${personne.id}"/></td>
26.           <td><c:out value="${personne.version}"/></td>
27.           <td><c:out value="${personne.prenom}"/></td>
28.           <td><c:out value="${personne.nom}"/></td>
29.           <td><dt:format pattern="dd/MM/yyyy">${personne.dateNaissance.time}</dt:format></td>
30.           <td><c:out value="${personne.marie}"/></td>
31.           <td><c:out value="${personne.nbEnfants}"/></td>
32.           <td><a href="<c:url value="/edit.html?id=${personne.id}"/>">Modifier</a></td>
33.           <td><a href="<c:url value="/delete.html?id=${personne.id}"/>">Supprimer</a></td>
34.         </tr>
35.       </c:forEach>
36.     </table>
37.     <br>
38.     <a href="<c:url value="/edit.html?id=-1"/>">Ajout</a>
39.   </body>
40. </html>
```

- cette vue reçoit une clé dans son modèle :

- la clé **${personnes}** associée à un objet de type [ArrayList] d'objets de type [Personne]

- lignes 23-35 : on parcourt la liste ${personnes} pour afficher un tableau HTML contenant les personnes du groupe.
- ligne 32 : l'url pointée par le lien [Modifier] est paramétrée par le champ [id] de la personne courante afin que le contrôleur associé à l'url [/edit.html] sache quelle est la personne à modifier.
- ligne 33 : il est fait de même pour le lien [Supprimer].
- ligne 29 : pour afficher la date de naissance de la personne sous la forme JJ/MM/AAAA, on utilise la balise <dt> de la bibliothèque de balise [DateTime] du projet Apache [Jakarta Taglibs] :

Taglibs

Home
Jakarta Taglibs
JCP Standardized Tag Libraries

DateTime Tag Library

The DateTime custom tag library contains tags which can be used to handle date and time related functions. Tags are provided for formatting a Date for output, generating a Date from HTML forum input, using time zones, and localization.

Le fichier de description de cette bibliothèque de balises est défini ligne 3.

La vue [edit.jsp]

Elle sert à afficher le formulaire d'ajout d'une nouvelle personne ou de modification d'une personne existante :

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	1	Charles	Lemarchand	01/01/1986	false	0	Modifier	Supprimer

Ajout

la vue [list.jsp]

Ajout/Modification d'une personne

Id	2
Version	1
Prénom	Mélanie
Nom	Humbort
Date de naissance (JJ/MM/AAAA)	12/01/1985
Marié	○ Oui ⦿ Non
Nombre d'enfants	1

Valider Annuler

la vue [edit.jsp]

Le code de la vue [edit.jsp] est le suivant :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ taglib uri="/WEB-INF/spring.tld" prefix="spring" %>
4.  <%@ page isELIgnored="false" %>
5.
6.  <html>
7.    <head>
8.      <title>Spring-mvc : Personnes</title>
9.    </head>
10.   <body background="../ressources/standard.jpg">
11.     <h2>Ajout/Modification d'une personne</h2>
12.     <spring:bind path="personne">
13.       <c:if test="${status.error}">
14.         <h3>Les erreurs suivantes se sont produites :</h3>
15.         <ul>
16.           <c:forEach items="${status.errorMessages}" var="erreur">
17.             <li><c:out value="${erreur}"/></li>
18.           </c:forEach>
```

```
19.        </ul>
20.        <hr>
21.      </c:if>
22.    </spring:bind>
23.    <form method="post" action="<c:url value="/edit.html"/>">
24.      <table border="1">
25.        <tr>
26.          <td>Id</td>
27.          <td>${personne.id}</td>
28.        </tr>
29.        <tr>
30.          <td>Version</td>
31.          <td>${personne.version}</td>
32.        </tr>
33.        <tr>
34.          <td>Pr&eacute;nom</td>
35.          <spring:bind path="personne.prenom">
36.            <td>
37.              <input type="text" value="${status.value}" name="${status.expression}" size="20">
38.            </td>
39.            <td>${status.errorMessage}</td>
40.          </spring:bind>
41.        </tr>
42.        <tr>
43.          <td>Nom</td>
44.          <spring:bind path="personne.nom">
45.            <td>
46.              <input type="text" value="${status.value}" name="${status.expression}" size="20">
47.            </td>
48.            <td>${status.errorMessage}</td>
49.          </spring:bind>
50.        </tr>
51.        <tr>
52.          <td>Date de naissance (JJ/MM/AAAA)</td>
53.          <spring:bind path="personne.dateNaissance">
54.            <td>
55.              <input type="text" value="${status.value}" name="${status.expression}">
56.            </td>
57.            <td>${status.errorMessage}</td>
58.          </spring:bind>
59.        </tr>
60.        <tr>
61.          <td>Mari&eacute;</td>
62.          <td>
63.            <c:choose>
64.              <c:when test="${personne.marie}">
65.                <input type="radio" name="marie" value="true" checked>Oui
66.                <input type="radio" name="marie" value="false">Non
67.              </c:when>
68.              <c:otherwise>
69.                <input type="radio" name="marie" value="true">Oui
70.                <input type="radio" name="marie" value="false" checked>Non
71.              </c:otherwise>
72.            </c:choose>
73.          </td>
74.        </tr>
75.        <tr>
76.          <td>Nombre d'enfants</td>
77.          <spring:bind path="personne.nbEnfants">
78.            <td>
79.              <input type="text" value="${status.value}" name="${status.expression}">
80.            </td>
81.            <td>${status.errorMessage}</td>
82.          </spring:bind>
83.        </tr>
84.      </table>
85.      <br>
86.      <input type="hidden" value="${personne.id}" name="id">
87.      <input type="submit" value="Valider">
88.      <a href="<c:url value="/list.html"/>">Annuler</a>
89.    </form>
90.  </body>
91. </html>
```

- cette vue reçoit une clé dans son modèle :
  - la clé **${personne}** qui est un objet de type [Personne] qui représente soit une personne à ajouter (id= -1), soit une personne à modifier (id<> -1)
- lignes 12-22 : on rencontre ici une balise <spring:bind> connue. Ce qui l'est moins, c'est la valeur de son attribut **path**, ici [personne]. Nous avions rencontré des balises <spring:bind> où l'attribut **path** avait pour valeur un champ du conteneur de saisies mais jamais le conteneur lui-même. Certaines erreurs sont liées non pas à un champ du conteneur, mais au traitement du conteneur lui-même. La balise <spring:bind> avec un attribut *path* ayant pour valeur le conteneur de saisies permet d'avoir accès à ce type d'erreurs. Entre les balises d'ouverture et de fermeture de la balise, la variable ${status} nous donne les informations suivantes :

    - ${status.error} : un booléen qui indique s'il existe des erreurs liées au conteneur lui-même
    - ${status.errorMessages} : la liste de ces erreurs
- lignes 16-18 : la boucle d'affichage des erreurs de niveau conteneur
- ligne 23 : le formulaire sera posté à l'url /edit.html
- ligne 27 : le champ [id] de l'objet [Personne] est affiché
- ligne 31 : le champ [version]
- lignes 35-40 : saisie du prénom de la personne au sein d'une balise <spring:bind> :
  - lors de l'affichage initial du formulaire (GET), ${status.value} affiche la valeur actuelle du champ [prenom] de l'objet [Personne] et ${status.errorMessage} est vide.
  - en cas d'erreur après le POST, on réaffiche la valeur saisie ${status.value} ainsi que le message d'erreur éventuel ${status.errorMessage}
- lignes 44-49 : saisie du nom de la personne
- lignes 53-58 : saisie de la date de naissance de la personne
- lignes 63-72 : saisie de l'état marié ou non de la personne avec un bouton radio. On utilise la valeur du champ [marie] de l'objet [Personne] pour savoir lequel des deux boutons radio doit être coché.
- lignes 77-82 : saisie du nombre d'enfants de la personne
- ligne 86 : un champ HTML caché nommé [id] et ayant pour valeur le champ [id] de la personne en cours d'édition, -1 pour un ajout, autre chose pour une modification.
- ligne 87 : le bouton [Valider] de type [Submit] du formulaire
- ligne 88 : un lien permettant de revenir à la liste des personnes. Il a été libellé [Annuler] parce qu'il permet de quitter le formulaire sans le valider.

La vue [exception.jsp]

Elle sert à afficher une page signalant qu'il s'est produit une exception non gérée par l'application et qui est remontée jusqu'à [DispatcherServlet]. La vue est générée par le gestionnaire d'exceptions déclaré dans [personnes-servlet.xml].

Par exemple, supprimons une personne qui n'existe pas dans le groupe :

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants	
1	1	Joachim	Major	13/01/1984	true	2	Mo
2	1	Mélanie	Humbort	12/01/1985	false	1	Mo
3	1	Charles	Lemarchand	01/01/1986	false	0	Mo

Ajout

la vue [list.jsp] - il n'y a pas de personne d'id=7

http://localhost:8080/spring-mvc-37/delete.html?id=7

Hotmail Personnaliser les liens Windows Media Windows

Erreur

L'exception suivante s'est produite : Personne d'id [7] inconnue

Retour à la liste

la vue [exception.jsp] – on a demandé la suppression de la personne d'id=7 en tapant à la main l'url dans le navigateur.

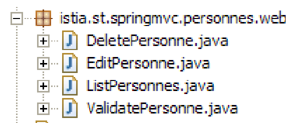
Le code de la vue [exception.jsp] est le suivant :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Spring MVC - personnes</title>
8.    </head>
9.    <body background="<c:url value="/ressources/standard.jpg"/>">
10.     <h2>Erreur</h2>
11.     L'exception suivante s'est produite :
12.     <c:out value="${exception.message}"/>
13.     <br><br>
14.     <a href="<c:url value="/list.html"/>">Retour à la liste</a>
15.   </body>
16. </html>
```

- cette vue reçoit une clé dans son modèle :
  - la clé **${exception}** associée à un objet de type [Exception] qui est l'exception qui est remontée jusqu'à [DispatcherServlet]
- ligne 12 : le texte de l'exception est affiché
- ligne 14 : on propose à l'utilisateur un lien pour revenir à la liste des personnes

2.8.3 Les contrôleurs de l'application web

Les contrôleurs sont définis dans le paquetage [istia.st.springmvc.personnes.web] :



Le contrôleur [ListPersonnes]

La configuration de ce contrôleur dans [personnes-servlet.xml] est la suivante :

```
1.  <!-- les mappings de l'application-->
2.  <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
3.    <property name="mappings">
4.      <props>
5.        <prop key="/list.html">Personnes.ListController</prop>
6....
7.      </props>
8.    </property>
9.  </bean>
10. <!-- LES CONTROLEURS -->
11. <bean id="Personnes.ListController"
12.   class="istia.st.springmvc.personnes.web.ListPersonnes">
13.   <property name="service">
14.     <ref bean="service"/>
15.   </property>
16.</bean>
```

Ci-dessus, on voit que l'url [/list.html] va être traitée par le contrôleur [ListPersonnes] (ligne 12). Ce contrôleur doit afficher la liste des personnes du groupe.

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	1	Charles	Lemarchand	01/01/1986	false	0	Modifier	Supprimer

Ajout

Son code est le suivant :

```
1.  package istia.st.springmvc.personnes.web;
2.
3.  import java.util.HashMap;
4.  import java.util.Map;
5.
6.  import istia.st.springmvc.personnes.service.IService;
7.
8.  import javax.servlet.http.HttpServletRequest;
9.  import javax.servlet.http.HttpServletResponse;
10.
11. import org.springframework.web.servlet.ModelAndView;
12. import org.springframework.web.servlet.mvc.Controller;
13.
14. public class ListPersonnes implements Controller {
15.
16.   // service
17.   IService service;
18.
19.   public IService getService() {
20.     return service;
21.   }
22.
23.   public void setService(IService service) {
24.     this.service = service;
25.   }
26.
27.   // gestion de la requête
```


www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

```
28.   public ModelAndView handleRequest(HttpServletRequest request,
29.       HttpServletResponse response) throws Exception {
30.     // le modèle de la vue [list]
31.     Map model = new HashMap();
32.     model.put("personnes", service.getAll());
33.     // résultat
34.     return new ModelAndView("list", model);
35.   }
36.
37. }
```

- lignes 17-25 : le contrôleur a une référence sur la couche [service]. Le contrôleur est instancié au moment de la création par [DispatcherServlet] des beans du fichier [personnes-servlet.xml]. Les lignes 11-15 de ce fichier ci-dessus montrent que la propriété [service] du contrôleur est initialisée avec un bean nommé [service]. Celui-ci a été créé par [ContextLoaderListener] lorsqu'il a exploité le fichier [applicationContext.xml] (cg paragraphe 2.8.1, page 24).
- ligne 14 : la classe implémente l'interface [Controller] qui n'a qu'une méthode [handleRequest].
- lignes 28-35 : implémentation de la méthode [handleRequest].
- ligne 31 : un modèle vide est créé
- ligne 32 : on demande à la couche [service] la liste des personnes du groupe et on met celle-ci dans le modèle sous la clé " personnes ".
- ligne 34 : on retourne un [ModelAndView] où " list " est le nom de la vue à afficher et le modèle de cette vue celui qui vient d'être construit.

La vue " list " va être affichée. Elle est associée à la page [list.jsp] décrite page 26.

Le contrôleur [DeletePersonne]

La configuration de ce contrôleur dans [personnes-servlet.xml] est la suivante :

```
1. <!-- les mappings de l'application-->
2.   <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
3.     <property name="mappings">
4.       <props>
5.         <prop key="/delete.html">Personnes.DeleteController</prop>
6. ...
7.       </props>
8.     </property>
9.   </bean>
10.  <!-- LES CONTROLEURS -->
11. ...
12.  <bean id="Personnes.DeleteController"
13.    class="istia.st.springmvc.personnes.web.DeletePersonne">
14.    <property name="service">
15.      <ref bean="service"/>
16.    </property>
17.  </bean>
```

Ci-dessus, on voit que l'url [/delete.html] doit être traitée par le contrôleur [DeletePersonne] (ligne 13). Cette url est celle des liens [Supprimer] de la vue [list.jsp] :

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	1	Charles	Lemarchand	01/01/1986	false	0	Modifier	Supprimer

Ajout

dont le code est le suivant :

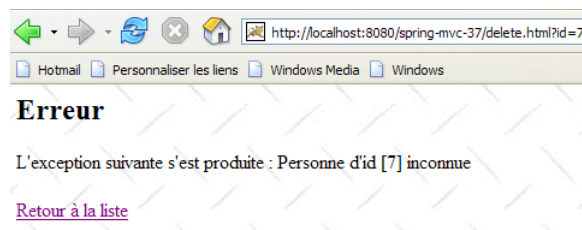
```
1.  ...
2.  <html>
3.    <head>
4.      <title>Spring MVC - personnes</title>
5.    </head>
6.    <body background="<c:url value="/ressources/standard.jpg"/>">
7.  ...
8.        <c:forEach var="personne" items="${personnes}">
9.          <tr>
10. ...
11.           <td><a href="<c:url value="/edit.html?id=${personne.id}"/>">Modifier</a></td>
```

```
12.         <td><a href="<c:url value="/delete.html?id=${personne.id}"/>">Supprimer</a></td>
13.       </tr>
14.     </c:forEach>
15.   </table>
16.   <br>
17.   <a href="<c:url value="/edit.html?id=-1"/>">Ajout</a>
18.  </body>
19. </html>
```

Ligne 12, on voit l'url [/delete.html?id=XX] du lien [Supprimer]. Le contrôleur [DeletePersonne] qui doit traiter cette url doit supprimer la personne d'id=XX puis faire afficher la nouvelle liste des personnes du groupe. Son code est le suivant :

```
1.  package istia.st.springmvc.personnes.web;
2.
3.  import istia.st.springmvc.personnes.service.IService;
4.
5.  import javax.servlet.http.HttpServletRequest;
6.  import javax.servlet.http.HttpServletResponse;
7.
8.  import org.springframework.web.bind.RequestUtils;
9.  import org.springframework.web.servlet.ModelAndView;
10. import org.springframework.web.servlet.mvc.Controller;
11.
12. public class DeletePersonne implements Controller {
13.   // service
14.   IService service;
15.
16.   public IService getService() {
17.     return service;
18.   }
19.
20.   public void setService(IService service) {
21.     this.service = service;
22.   }
23.
24.   // on gère la requête
25.   public ModelAndView handleRequest(HttpServletRequest request,
26.       HttpServletResponse response) throws Exception {
27.     // on récupère l'id de la personne à supprimer
28.     int id = RequestUtils.getIntParameter(request, "id", 0);
29.     // on supprime la personne
30.     service.deleteOne(id);
31.     // on redirige vers la liste des personnes
32.     return new ModelAndView("r-list");
33.   }
34. }
```

- ce qui a été dit pour [ListPersonnes] peut être repris à l'identique ici.
- lignes 25-32 : implémentation de la méthode [handleRequest] de l'interface [Controller]
- ligne 28 : l'url traitée est de la forme [/delete.html?id=XX]. On récupère la valeur [XX] du paramètre [id].
- ligne 30 : on demande à la couche [service] la suppression de la personne ayant l'id obtenu. Nous ne faisons aucune vérification. Si la personne qu'on cherche à supprimer n'existe pas, la couche [dao] lance une exception que laisse remonter la couche [service]. Nous ne la gérons pas non plus ici. Elle remontera donc jusqu'à [DispatcherServlet] qui appellera alors le gestionnaire d'exceptions. Celui-ci fera afficher sa page par défaut. Nous avons montré ce cas dans l'étude de la vue [exception.jsp], page 29 :



- ligne 32 : si la suppression a eu lieu (pas d'exception), on retourne un [ModelAndView] où "r-list" est le nom de la vue. Il n'y a pas de modèle. Dans [vues.properties], la vue [r-list] est définie comme suit :

```
1.  #r-list
2.  r-list.class=org.springframework.web.servlet.view.RedirectView
3.  r-list.url=/list.html
4.  r-list.contextRelative=true
5.  r-list.http10Compatible=false
```

C'est donc une redirection vers l'url [/list.html] qui est faite : la liste des personnes du groupe va être réaffichée sans la personne qui vient d'être supprimée.

Le contrôleur [EditPersonne]

La configuration de ce contrôleur dans [personnes-servlet.xml] est la suivante :

```
1.<!-- les mappings de l'application-->
2.  <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
3.    <property name="mappings">
4.      <props>
5...
6.        <prop key="/edit.html">Personnes.EditController</prop>
7.      </props>
8.    </property>
9.  </bean>
10. <!-- LES CONTROLEURS -->
11.....
12. <bean id="Personnes.EditController"
13.   class="istia.st.springmvc.personnes.web.EditPersonne">
14....
15. </bean>
```

Ci-dessus, on voit que l’url [/edit.html] doit être traitée par le contrôleur [EditPersonne] (lignes 6 et 13). Cette url est celle des liens [Modifier] et celui du lien [Ajout] de la vue [list.jsp] :

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	1	Charles	Lemarchand	01/01/1986	false	0	Modifier	Supprimer

Ajout

dont le code est le suivant :

```
1.  ...
2.  <html>
3.    <head>
4.      <title>Spring MVC - personnes</title>
5.    </head>
6.    <body background="<c:url value="/ressources/standard.jpg"/>">
7.  ...
8.        <c:forEach var="personne" items="${personnes}">
9.          <tr>
10. ...
11.           <td><a href="<c:url value="/edit.html?id=${personne.id}"/>">Modifier</a></td>
12.           <td><a href="<c:url value="/delete.html?id=${personne.id}"/>">Supprimer</a></td>
13.         </tr>
14.       </c:forEach>
15.     </table>
16.     <br>
17.     <a href="<c:url value="/edit.html?id=-1"/>">Ajout</a>
18.   </body>
19. </html>
```

Ligne 11, on voit l’url [/edit.html?id=XX] du lien [Modifier]. Le contrôleur [EditPersonne] qui doit traiter cette url doit gérer un formulaire d’édition de la personne d’id=XX.

Si id= -1, il s’agit d’un ajout et le formulaire est présenté vide. Ce sera le cas si l’utilisateur utilise le lien [Ajout] de la ligne 17 ci-dessus.

Ajout/Modification d'une personne

Id	-1
Version	0
Prénom	
Nom	
Date de naissance (JJ/MM/AAAA)	
Marié	○ Oui ⦿ Non
Nombre d'enfants	0

[Valider] Annuler

Si id<> -1, il s'agit d'une modification et le formulaire doit être présenté pré-rempli avec les informations de la personne à modifier. Ce sera le cas si l'utilisateur utilise le lien [Modifier] de la ligne 11 de la page [list.jsp] :

Ajout/Modification d'une personne

Id	3
Version	1
Prénom	Charles
Nom	Lemarchand
Date de naissance (JJ/MM/AAAA)	01/01/1986
Marié	○ Oui ⦿ Non
Nombre d'enfants	0

[Valider] Annuler

Le contrôleur [EditPersonne] dérive de [SimpleFormController]. Sa configuration dans [personnes-servlet.xml] est donc celle de ce contrôleur :

```
1.    <bean id="Personnes.EditController"
2.      class="istia.st.springmvc.personnes.web.EditPersonne">
3.      <property name="sessionForm">
4.        <value>true</value>
5.      </property>
6.      <property name="commandName">
7.        <value>personne</value>
8.      </property>
9.      <property name="validator">
10.       <ref bean="Personnes.Validator"/>
11.     </property>
12.     <property name="formView">
13.       <value>edit</value>
14.     </property>
15.     <property name="service">
16.       <ref bean="service"/>
17.     </property>
18.   </bean>
19.   <!-- le validateur -->
20.   <bean id="Personnes.Validator"
21. class="istia.st.springmvc.personnes.web.ValidatePersonne"/>
```

- lignes 12-14 : la vue associée au formulaire se nomme " edit ". Nous savons qu'elle correspond à la page JSP [edit.jsp] étudiée page 27.
- la configuration ne précise pas le paramètre [commandClass] qui fixe le conteneur dont les champs servent à :
    1. alimenter le formulaire lorsque celui-ci est affiché au moment du GET
    2. recevoir les valeurs postées au moment du POST

  Lorsque le paramètre [commandClass] est défini, le conteneur des saisies est créé au moment du GET par appel du constructeur par défaut de la classe associée, ce qui fait que le formulaire est affiché vide. Cette méthode conviendrait ici pour l'ajout d'une personne [/edit.html?id=-1]. Pour la modification d'une personne existante [/edit.html?id=XX], au moment du GET, on doit afficher un formulaire pré-rempli avec les informations de la personne que l'utilisateur veut modifier. La méthode précédente ne convient alors pas. En l'absence du paramètre [commandClass], le conteneur doit être fourni par la méthode [formBackingObject] du contrôleur. C'est ce qui sera fait ici. La méthode [formBackingObject] fournira un objet [Personne] vide si l'url à traiter est [/edit.html?id=-1], où l'objet [Personne] d'id=XX si l'url à traiter est [/edit.html?id=XX]. Dans ce dernier cas, l'objet [Personne] sera demandé à la couche [service].

- lignes 6-8 : le conteneur des saisies sera dans le modèle de la vue " edit " sous le nom " personne "

- lignes 3-5 : l'objet [Personne] créé au moment du GET par la méthode [formBackingObject] du contrôleur restera en session pour être réutilisé au moment du POST.
- lignes 9-11 : lorsque le formulaire sera posté, il sera vérifié par le validateur [ValidatePersonne] défini lignes 20-21. Il s'agira de vérifier la validité des différents champs de l'objet [Personne] posté.

- le contrôleur [EditPersonne] a besoin d'avoir accès à la couche [service] (lecture puis sauvegarde d'une personne à modifier par exemple). Le contrôleur détient une référence sur la couche [service] qui est initialisée par les lignes 15-18.

Le code du contrôleur est le suivant :

```
1.  package istia.st.springmvc.personnes.web;
2.
3.  import java.text.SimpleDateFormat;
4.  import istia.st.springmvc.personnes.dao.DaoException;
5.  import istia.st.springmvc.personnes.entites.Personne;
6.  import istia.st.springmvc.personnes.service.IService;
7.
8.  import javax.servlet.http.HttpServletRequest;
9.  import javax.servlet.http.HttpServletResponse;
10.
11. import org.springframework.beans.propertyeditors.CustomDateEditor;
12. import org.springframework.validation.BindException;
13. import org.springframework.web.bind.RequestUtils;
14. import org.springframework.web.bind.ServletRequestDataBinder;
15. import org.springframework.web.servlet.ModelAndView;
16. import org.springframework.web.servlet.mvc.SimpleFormController;
17. import org.springframework.web.servlet.view.RedirectView;
18.
19. public class EditPersonne extends SimpleFormController {
20.   // service
21.   IService service;
22.
23.   public IService getService() {
24.     return service;
25.   }
26.
27.   public void setService(IService service) {
28.     this.service = service;
29.   }
30.
31.   // enregistrement d'éditeurs de propriétés
32.   protected void initBinder(HttpServletRequest request,
33.       ServletRequestDataBinder binder) throws Exception {
34.     // format attendu pour la date e naissance
35.     SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
36.     // format strict
37.     dateFormat.setLenient(false);
38.     // on enregistre un éditeur de propriétés String (dd/MM/yyyy) -> Date
39.     // CustomDateEditor est fourni par Spring - il sera utilisé par Spring
40.     // pour transformer
41.     // la chaîne saisie dans le formulaire en type java.util.Date
42.     // la date ne pourra être vide (2ième paramètre de CustomDateEditor)
43.     binder.registerCustomEditor(java.util.Date.class, null,
44.         new CustomDateEditor(dateFormat, false));
45.   }
46.
47.   // préparation [Personne] à afficher
48.     protected Object formBackingObject(HttpServletRequest request) {
49.     // on récupère l'id de la personne
50.     int id = RequestUtils.getIntParameter(request, "id", -1);
51.     // ajout ou modification ?
52.     Personne personne = null;
53.     if (id != -1) {
54.       // modification - on récupère la personne à modifier
55.       personne = service.getOne(id);
56.     }else{
57.       // ajout - on crée une personne vide
58.       personne = new Personne();
59.       personne.setId(-1);
60.     }
61.     // on rend l'objet [Personne]
62.     return personne;
63.   }
64.
65.   // exécution de la commande
66.   protected ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse response, Object
    command, BindException errors) throws Exception {
67.     // sauvegarde la personne ajoutée ou modifiée
68.     Personne personne = (Personne) command;
69.     int idPersonne = personne.getId();
70.     try {
```

```
71.       // sauvegarde de la personne
72.       service.saveOne(personne);
73.       // on redirige vers la liste des personnes
74.       return new ModelAndView("r-list");
75.     } catch (DaoException ex) {
76.       // on note l'erreur
77.       String message=idPersonne==-1 ? "personne.ajout.echec" : "personne.modification.echec";
78.       errors.reject(message,new Object[]{ex.getMessage()},"Echec de la mise à jour: {0}");
79.       // on réaffiche le formulaire
80.       return showForm(request,response, errors);
81.     }
82.   }
83.
84. }
```

- ligne 19 : la classe [EditPersonne] dérive de [SimpleFormController]
- lignes 21-29 : l'attribut [service] initialisé lors de l'instanciation du contrôleur par [DispatcherServlet]. Nous en avons parlé un peu plus haut. Cet attribut est une référence sur la couche [service] de l'architecture 3tier de l'application.
- lignes 32-45 : la méthode [initBinder] est utilisée ici pour la date de naissance de la personne. Nous voulons être capables de faire la conversion [String] (valeur saisie au format JJ/MM/AAAA) vers le type [java.util.Date]. Pour cela, il nous faut enregistrer un éditeur de propriété particulier. Nous avons rencontré ce cas dans un autre exemple de l'article 2 (paragraphe 5.11). Nous n'y revenons pas.
- lignes 48-64 : la méthode [formBackingObject] qui, au moment du GET initial sur le formulaire, rend une référence sur le conteneur de saisies, c.a.d. sur un objet [Personne].
- le GET a pour cible une url du type [/edit.html?id=XX]. Ligne 50, nous récupérons la valeur de [id]. Ensuite il y a deux cas :
  1. id est différent de -1. Alors il s'agit d'une modification et il faut afficher un formulaire pré-rempli avec les informations de la personne à modifier. Ligne 55, cette personne est demandée à la couche [service].
  2. id est égal à -1. Alors il s'agit d'un ajout et il faut afficher un formulaire vide. pour cela, une personne vide est créée lignes 58-59.

  La méthode [formBackingObject] rend l'objet [Personne] créé. Ce sera le conteneur de saisies associé au formulaire [edit.jsp].

- lignes 66-82 : l'une des méthodes [submit] que l'on peut redéfinir pour traiter le POST du formulaire. Rappelons qu'entre-temps, celui a été vérifié par le validateur de données et qu'il a été déclaré correct sinon on ne serait pas arrivé là. Nous avons choisi ici, une méthode [submit] (il en existe plusieurs avec différentes signatures) permettant de signaler des erreurs. C'est le paramètre [BindException errors] qui va enregistrer celles-ci.
  - ligne 68 : on récupère le conteneur de saisies de type [Personne]. Il contient la personne modifiée ou la personne ajoutée.
  - ligne 69 : on récupère l'id de la personne
  - ligne 72 : la personne est sauvegardée. La sauvegarde peut échouer. Dans un cadre multi-utilisateurs, la personne à modifier a pu être supprimée ou bien déjà modifiée par quelqu'un d'autre. Dans ce cas, la couche [dao] va lancer une exception qu'on gère ici.
  - ligne 74 : s'il n'y a pas eu d'exception, on redirige le client vers l'url [/list.html] pour lui présenter le nouvel état du groupe. Pour cela, on renvoie un [ModelAndView] demandant l'affichage de la vue " r-list " qui s'avère être une redirection (cf vues.properties).
  - lignes 77-78 : s'il y a eu exception, on crée le message d'erreur adéquat et on l'ajoute à la liste des erreurs [BindException errors] qui nous a été passée en paramètre (ligne 78). On utilise pour cela la méthode [reject] de l'interface [BindException] qui lie l'erreur au conteneur de saisies plutôt qu'à un champ particulier de celui-ci.
  - ligne 80 : on redemande le réaffichage du formulaire initial. La méthode [showForm] renvoie un [ModelAndView] où la vue est celle nommée [formView] dans la configuration du contrôleur et où le modèle contient le conteneur de saisies qui a été refusé et qu'elle trouve dans la session (sessionForm=true) ainsi que la liste des erreurs [BindException errors] qu'on lui passe en paramètre. Nous savons qu'alors la page [edit.jsp] va être utilisée pour afficher ce modèle. Rappelons le code qui affiche la liste des erreurs de niveau conteneur :

```
<spring:bind path="personne">
  <c:if test="${status.error}">
    <h3>Les erreurs suivantes se sont produites :</h3>
    <ul>
      <c:forEach items="${status.errorMessages}" var="erreur">
        <li><c:out value="${erreur}"/></li>
      </c:forEach>
    </ul>
    <hr>
  </c:if>
</spring:bind>
```

Nous avons dit qu'au moment du POST, les saisies étaient contrôlées pas un validateur de données avant d'être passées à la méthode [onSubmit] ci-dessus. Nous voyons maintenant ce validateur de données.

Le validateur [ValidatePersonne]

Celui-ci est lié par configuration au contrôleur [EditPersonne] :

```
1.  <bean id="Personnes.EditController"
2.      class="istia.st.springmvc.personnes.web.EditPersonne">
3.  ...
4.      <property name="validator">
5.        <ref bean="Personnes.Validator"/>
6.      </property>
7.  ...
8.    </bean>
9.    <!-- le validateur -->
10.   <bean id="Personnes.Validator"
11. class="istia.st.springmvc.personnes.web.ValidatePersonne"/>
```

[ValidatePersonne] doit s'assurer que l'objet [Personne] qui est posté par le formulaire [edit.jsp] est correct. Son code est le suivant :

```
1.  package istia.st.springmvc.personnes.web;
2.
3.  import istia.st.springmvc.personnes.entites.Personne;
4.
5.  import org.springframework.validation.Errors;
6.
7.  public class ValidatePersonne implements
8.     org.springframework.validation.Validator {
9.
10.   public boolean supports(Class classe) {
11.     return classe.isAssignableFrom(Personne.class);
12.   }
13.
14.   public void validate(Object obj, Errors erreurs) {
15.     // on récupère la personne postée
16.     Personne personne = (Personne) obj;
17.     // on vérifie le prénom
18.     String prénom = personne.getPrenom();
19.     if (prénom==null || prénom.trim().length() == 0) {
20.       erreurs.rejectValue("prenom", ,
21.           "Le prénom est nécessaire !");
22.     }
23.     // on vérifie le nom
24.     String nom = personne.getNom();
25.     if (nom==null || nom.trim().length() == 0) {
26.       erreurs.rejectValue("nom", "personne.nom.necessaire",
27.           "Le nom est nécessaire !");
28.     }
29.     // on vérifie le nombre d'enfants
30.     int nbEnfants = personne.getNbEnfants();
31.     if (nbEnfants < 0) {
32.       erreurs.rejectValue("nbEnfants", "personne.nbEnfants.invalide",
33.           "Donnée incorrecte !");
34.     }
35.   }
36. }
```

- lignes 7-8 : [ValidatePersonne] implémente l'interface [Validator] et doit donc, à ce titre, implémenter les méthodes [supports] et [validate].
- lignes 10-12 : la méthode [supports] : [ValidatePersonne] sert à contrôler la validité des objets [Personne] ou dérivés.
- lignes 14-35 : la méthode [validate]. Des vérifications basiques sont faites sur le prénom saisi (lignes 18-22), le nom (lignes 24-28), le nombre d'enfants (lignes 30-34).

Le fichier [messages.properties] des messages d'erreurs

Les messages d'erreur associés aux codes d'erreur tels que "personne.prenom.necessaire", "personne.nom.necessaire", ... qu'on trouve dans la méthode [validate] du validateur ainsi que dans la méthode [onSubmit] du contrôleur [EditPersonne] sont définis dans le fichier [messages.properties]. Ceci est fixé par configuration dans [personnes-servlet.xml] :

```
<!-- le fichier des messages -->
<bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
  <property name="basename">
    <value>messages</value>
  </property>
</bean>
```

Le contenu de [messages.properties] est le suivant :

```
personne.prenom.necessaire=Le nom est obligatoire !
personne.nom.necessaire=Le prénom est obligatoire !
personne.nbEnfants.invalide=Donnée incorrecte !
typeMismatch=Donnée incorrecte !
personne.ajout.echec=Echec de l'ajout : {0}
personne.modification.echec=Echec de la modification : {0}
```

2.9 Les tests de l’application web

Un certain nombre de tests ont été présentés au paragraphe 2.1, page 3. Nous invitons le lecteur à les rejouer. Nous montrons ici d’autres copies d’écran qui illustrent les cas de conflits d’accès aux données dans un cadre multi-utilisateurs :

[Firefox] sera le navigateur de l’utilisateur U1. Celui demande l’url [http://localhost:8080/spring-mvc-37/list.html] :

http://localhost:8080/spring-mvc-37/list.html

Hotmail Personnaliser les liens Windows Media Windows

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	1	Charles	Lemarchand	01/01/1986	false	0	Modifier	Supprimer

Ajout

[IE] sera le navigateur de l’utilisateur U2. Celui demande la même Url :

Adresse http://localhost:8080/spring-mvc-37/list.html OK Liens

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	1	Charles	Lemarchand	01/01/1986	false	0	Modifier	Supprimer

Ajout

L’utilisateur U1 entre en modification de la personne [Lemarchand] :

http://localhost:8080/spring-mvc-37/edit.html?id=3

Hotmail Personnaliser les liens Windows Media Windows

Ajout/Modification d'une personne

Id	3
Version	1
Prénom	Charles
Nom	Lemarchand
Date de naissance (JJ/MM/AAAA)	01/01/1986
Marié	Oui ● Non
Nombre d'enfants	0

Valider Annuler

L’utilisateur U2 fait de même :

Adresse http://localhost:8080/spring-mvc-37/edit.html?id=3

Ajout/Modification d'une personne

Id	3
Version	1
Prénom	Charles
Nom	Lemarchand
Date de naissance (JJ/MM/AAAA)	01/01/1986
Marié	○ Oui ⦿ Non
Nombre d'enfants	0

Valider Annuler

L'utilisateur U1 fait des modifications et valide :

http://localhost:8080/spring-mvc-37/edit.html?id=3

Hotmail Personnaliser les liens Windows Media Windows

Ajout/Modification d'une personne

Id	3
Version	1
Prénom	Charles
Nom	Lemarchand
Date de naissance (JJ/MM/AAAA)	01/01/1986
Marié	⦿ Oui ○ Non
Nombre d'enfants	1

Valider Annuler

validation

http://localhost:8080/spring-mvc-37/list.html

Hotmail Personnaliser les liens Windows Media Windows

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	2	Charles	Lemarchand	01/01/1986	true	1	Modifier	Supprimer

Ajout

réponse du serveur

L'utilisateur U2 fait de même :

Adresse http://localhost:8080/spring-mvc-37/edit.html?id=3

Ajout/Modification d'une personne

Id	3
Version	1
Prénom	Charles
Nom	LEMARCHAND
Date de naissance (JJ/MM/AAAA)	01/01/1986
Marié	○ Oui ⦿ Non
Nombre d'enfants	0

Valider Annuler

validation

Adresse http://localhost:8080/spring-mvc-37/edit.html

Ajout/Modification d'une personne

Les erreurs suivantes se sont produites :

- Echec de la modification : L'original de la personne [[3,1,Charles,LEMARCHAND,01/01/1986,false,0]] a changé depuis sa lecture initiale

Id	3
Version	1
Prénom	Charles
Nom	LEMARCHAND
Date de naissance (JJ/MM/AAAA)	01/01/1986
Marié	○ Oui ⦿ Non
Nombre d'enfants	0

Valider Annuler

le conflit de version a été détecté

L'utilisateur U2 revient à la liste des personnes avec le lien [Annuler] du formulaire :

Adresse http://localhost:8080/spring-mvc-37/list.html OK Liens

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	2	Charles	Lemarchand	01/01/1986	true	1	Modifier	Supprimer

Ajout

Il trouve la personne [Lemarchand] telle que U1 l'a modifiée. Maintenant U2 supprime [Lemarchand] :

Adresse http://localhost:8080/spring-mvc-37/list.html OK Liens

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	2	Charles	Lemarchand	01/01/1986	true	1	Modifier	Supprimer

Ajout

- Suppression

Adresse http://localhost:8080/spring-mvc-37/list.html OK Liens

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer

Ajout

- réponse du serveur

U1 a toujours sa propre liste et veut modifier [Lemarchand] de nouveau :

http://localhost:8080/spring-mvc-37/list.html

Hotmail Personnaliser les liens Windows Media Windows

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	2	Charles	Lemarchand	01/01/1986	true	1	Modifier	Supprimer

Ajout

- modification

http://localhost:8080/spring-mvc-37/edit.html?id=3

Hotmail Personnaliser les liens Windows Media Windows

Erreur

L'exception suivante s'est produite : Personne d'id [3] inconnue

Retour à la liste

- réponse du serveur : il n'a pas trouvé [Lemarchand]

U1 utilise le lien [Retour à la liste] pour voir de quoi il retourne :

http://localhost:8080/spring-mvc-37/list.html

Hotmail Personnaliser les liens Windows Media Windows

Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer

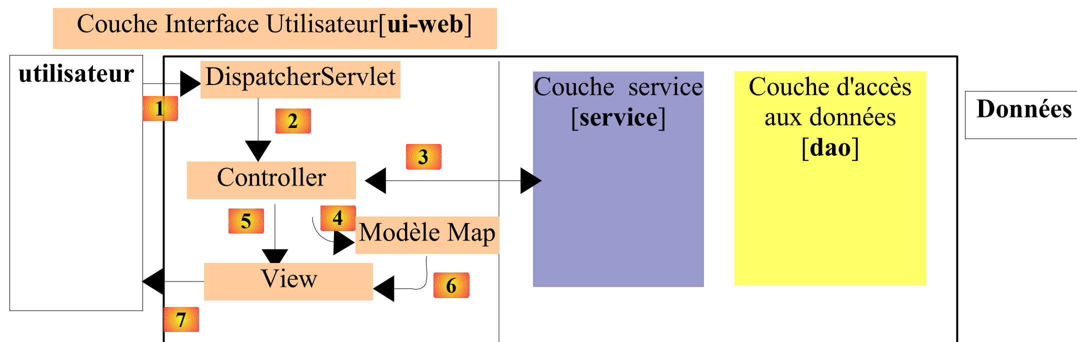
Ajout

Il découvre qu'effectivement [Lemarchand] ne fait plus partie de la liste...

2.10 Mise en archives de l'application web

Nous avons développé un projet Eclipse / Tomcat d'une application 3tier :

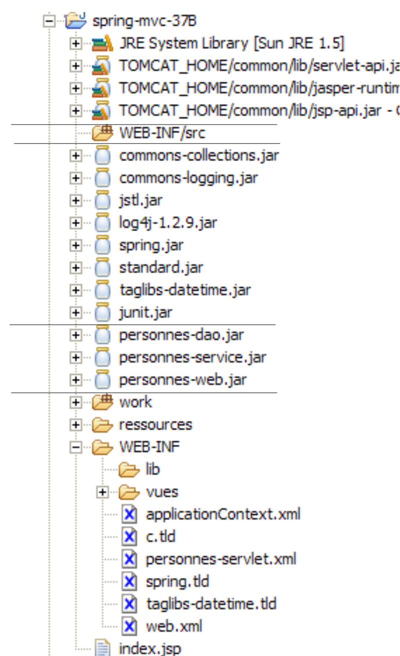
www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com



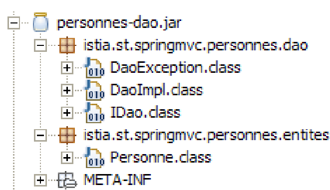
Dans une nouvelle version à venir, le groupe de personnes va être placée dans une table de base de données.

- Cela va induire une réécriture de la couche [**dao**]. Cela, on peut aisément le comprendre.
- La couche [**service**] va être également modifiée. Actuellement, son unique rôle est d'assurer un accès synchronisé aux données gérées par la couche [dao]. Dans ce but, nous avons synchronisé toutes les méthodes de la couche [service]. Nous avons expliqué pourquoi cette synchronisation était placée dans cette couche plutôt que dans la couche [dao]. Dans la nouvelle version, la couche [service] aura encore l'unique rôle de synchronisation des accès mais celle-ci sera assurée par des transactions de base de données plutôt que par une synchronisation de méthodes Java.
- La couche [**web**] va elle rester inchangée.

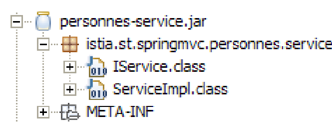
Afin de faciliter le passage d'une version à l'autre, nous créons un nouveau projet Eclipse / Tomcat [spring-mvc-37B], copie du projet précédent [spring-mvc-37] mais où les couches [web, service, dao] on été mises dans des archives .jar :



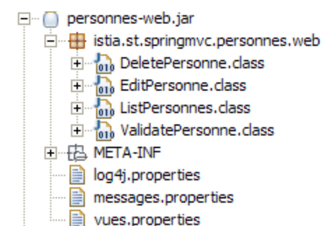
Le dossier [WEB-INF/src] est désormais vide. Il contenait le code source des classes Java ainsi que les fichiers [*.properties] qui configuraient la couche [web]. Ces éléments sont passés dans les trois archives [personnes-*.jar] :



archive de la couche [dao]

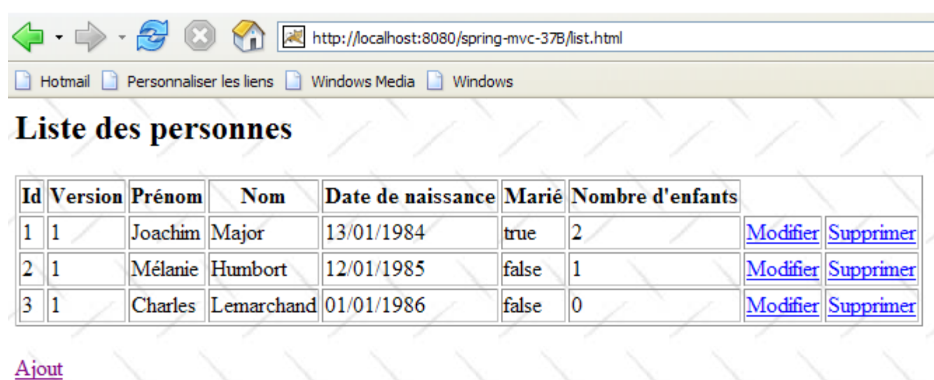


archive de la couche [service]



archive de la couche [web]

Le projet Eclipse / Tomcat [spring-mvc-37B] a été configuré pour inclure les trois archives [personnes-*.jar] dans son *ClassPath*. Pour le tester, nous demandons l'url [http://localhost:8080/spring-mvc-37B/list.html] :



Liste des personnes

Id	Version	Prénom	Nom	Date de naissance	Marié	Nombre d'enfants		
1	1	Joachim	Major	13/01/1984	true	2	Modifier	Supprimer
2	1	Mélanie	Humbort	12/01/1985	false	1	Modifier	Supprimer
3	1	Charles	Lemarchand	01/01/1986	false	0	Modifier	Supprimer

Ajout

Le lecteur est invité à faire des tests complémentaires. On pourra noter que si on a eu cet affichage, c'est que le fichier des vues [vues.properties] a été correctement exploité. Or il est dans l'archive [personnes-web.jar]. On sait que ce fichier est cherché dans le *ClassPath* de l'application web et que l'archive [personnes-web.jar] fait partie de celui-ci.

Dans la version avec base de données, nous allons changer les couches [service] et [dao]. Nous voulons montrer qu'il suffira alors, de remplacer dans le projet précédent les archives [personnes-dao.jar] et [personnes-service.jar] par les nouvelles archives pour que notre application fonctionne désormais avec une base de données. Nous n'aurons pas à toucher à l'archive de la couche [web].

3 Conclusion

Dans cet article nous avons mis en oeuvre Spring MVC dans une architecture 3tier [web, metier, dao] sur un exemple basique de gestion d'une liste de personnes. Cela nous a permis d'utiliser les concepts qui avaient été présentés dans les précédents articles.

Dans la version étudiée, la liste des personnes était maintenue en mémoire. Le prochain article présentera une version où cette liste sera maintenue dans une table de base de données.

4 Le code de l'article

Comme pour les précédents articles, le lecteur trouvera le code des exemples de ce document sous la forme d'un fichier zippé sur le site de l'article. Les règles de déploiement du fichier zippé sont à relire dans l'article 1. Une fois un projet importé dans [Eclipse] :

- copier le contenu du dossier [lib] du zip dans le dossier [WEB-INF/lib] du projet
- s'assurer que le dossier [work] existe sinon le créer : [clic droit sur projet / Projet Tomcat / Créer le dossier work]
- nettoyer le projet [Project / clean / clean selected projects]

Table des matières

1RAPPELS......... 2

2SPRING MVC DANS UNE ARCHITECTURE 3TIER – EXEMPLE 1......... 2

2.1Présentation......... 2
2.2Le projet Eclipse / Tomcat......... 4
2.3La représentation d'une personne......... 5
2.4La couche [dao]......... 7
2.5Tests de la couche [dao]......... 11
2.6La couche [service]......... 18
2.7Tests de la couche [service]......... 19
2.8La couche [web]......... 22
2.8.1Configuration de l'application web......... 23
2.8.2Les vues de l'application web......... 25
2.8.3Les contrôleurs de l'application web......... 30
2.9Les tests de l'application web......... 38
2.10Mise en archives de l'application web......... 40

3CONCLUSION......... 42

4LE CODE DE L'ARTICLE......... 42