Livre Blanc

Framework Open Source

Mise en oeuvre de Tapestry, Spring et Hibernate

Version 1 - Avril 2006

TABLE DES MATIÈRES

1 AVANT PROPOS

Ce document a pour but de présenter un exemple de mise en œuvre conjointe de trois parmi les *framework* Open Source Java les plus populaires du moment : **Tapestry**, **Spring** et **Hibernate**. Il s'agit d'un retour d'expérience concret issu d'un projet réalisation d'une application de gestion chez un client dans le monde de l'assurance.

Après une présentation succincte de chacun des *framework*, nous expliquons la façon dont nous les avons utilisés dans le cadre du projet, puis nous donnons une analyse des avantages et inconvénients de ce type d'architecture dans le cadre du développement d'une application web de gestion.

2 PRÉSENTATION

Lors d'une récente mission, nous avons été à développer une architecture technique basée sur les *framework* Open Source Tapestry, Spring et Hibernate.

Le *framework* ainsi constitué devait être utilisé pour développer une application de gestion Web tournant sur un serveur JSP/Servlet (en l'occurrence Tomcat) et utilisant une base de données Oracle. Il devait également pouvoir être utilisé dans d'autres projet du même type avec des bases de données MySQL ou Oracle.

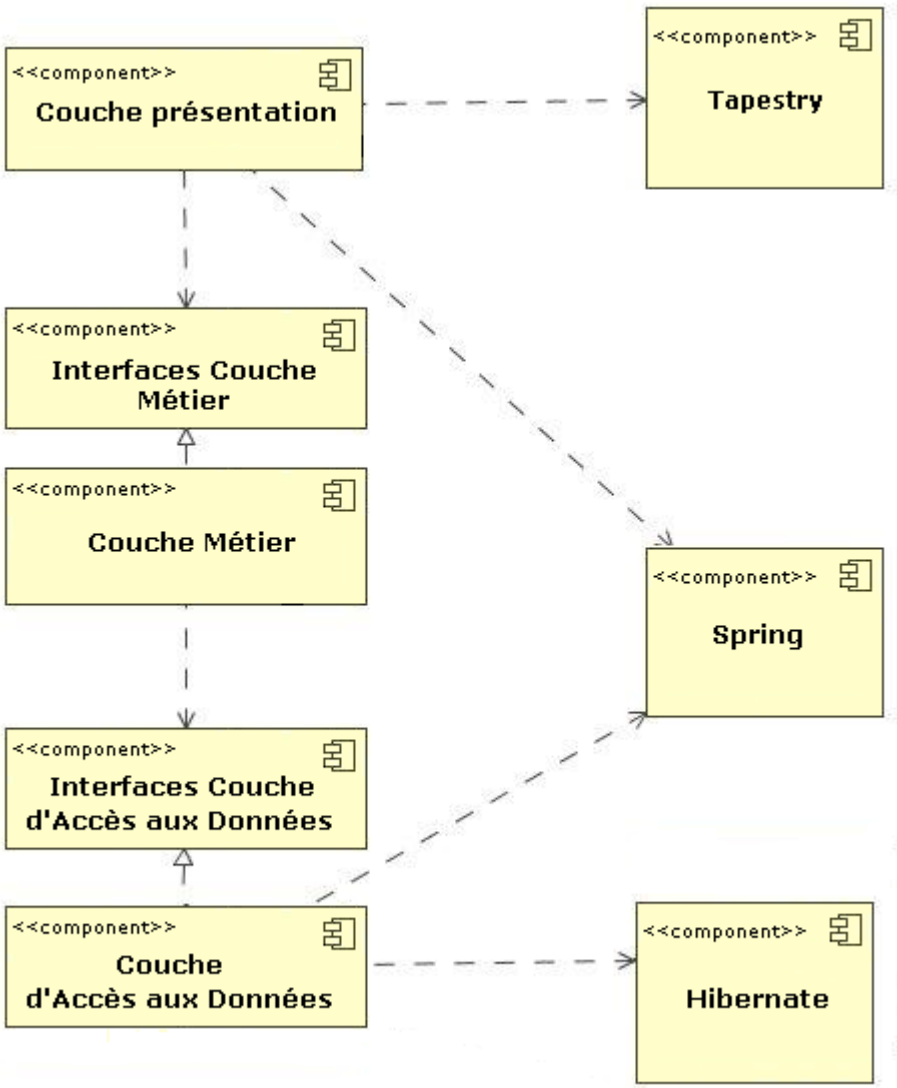
2.1 Objectifs du framework

Par abus de langage, nous appelons « le framework » le cadre de développement constitué par les *framework* Tapestry, Spring et Hibernate.

Les objectifs recherchés lors de la mise en place de ce framework étaient :

- Pouvoir développer des applications de gestion tournant sur un serveur JSP/Servlet et utilisant une ou des bases de données.
- Avoir des temps de développements les plus optimum possibles
- Avoir des performances honorables sur l'application développée. Nous ne recherchions pas à avoir des temps de réponse optimisés, étant donné la faible audience visée.
- Avoir une application robuste. Ne pas avoir une phase de recette technique longue à la fin des développements.
- Pouvoir capitaliser les éléments réutilisables des développements pour d'autres applications basées sur les mêmes *framework.*
- Pouvoir séparer les couches afin de permettre de faire simplement évoluer les développements.
- Avoir à disposition les briques utiles au développement d'applications web.
- Avoir un cadre de travail permettant le travail collaboratif et poussant la production de code de qualité.

2.2 Architecture des applications



- **La couche présentation est assurée par Tapestry**

  - Les vues sont en HTML avec des balises Tapestry "jwcid" pour y insérer des données.

  - Les contrôleurs font le lien entre les actions des utilisateurs et la couche métier. Ils gèrent également l'enchaînement des pages.

- **La couche métier**

  - Elle fournit des interfaces utilisables par la couche présentation.

  - Elle accède à la couche DAO.

  - L'implémentation des interfaces est contrôlée par Spring.

- **La couche modèle**

  - Cette couche contient le modèle de données, les fichiers de *mapping* Hibernate.

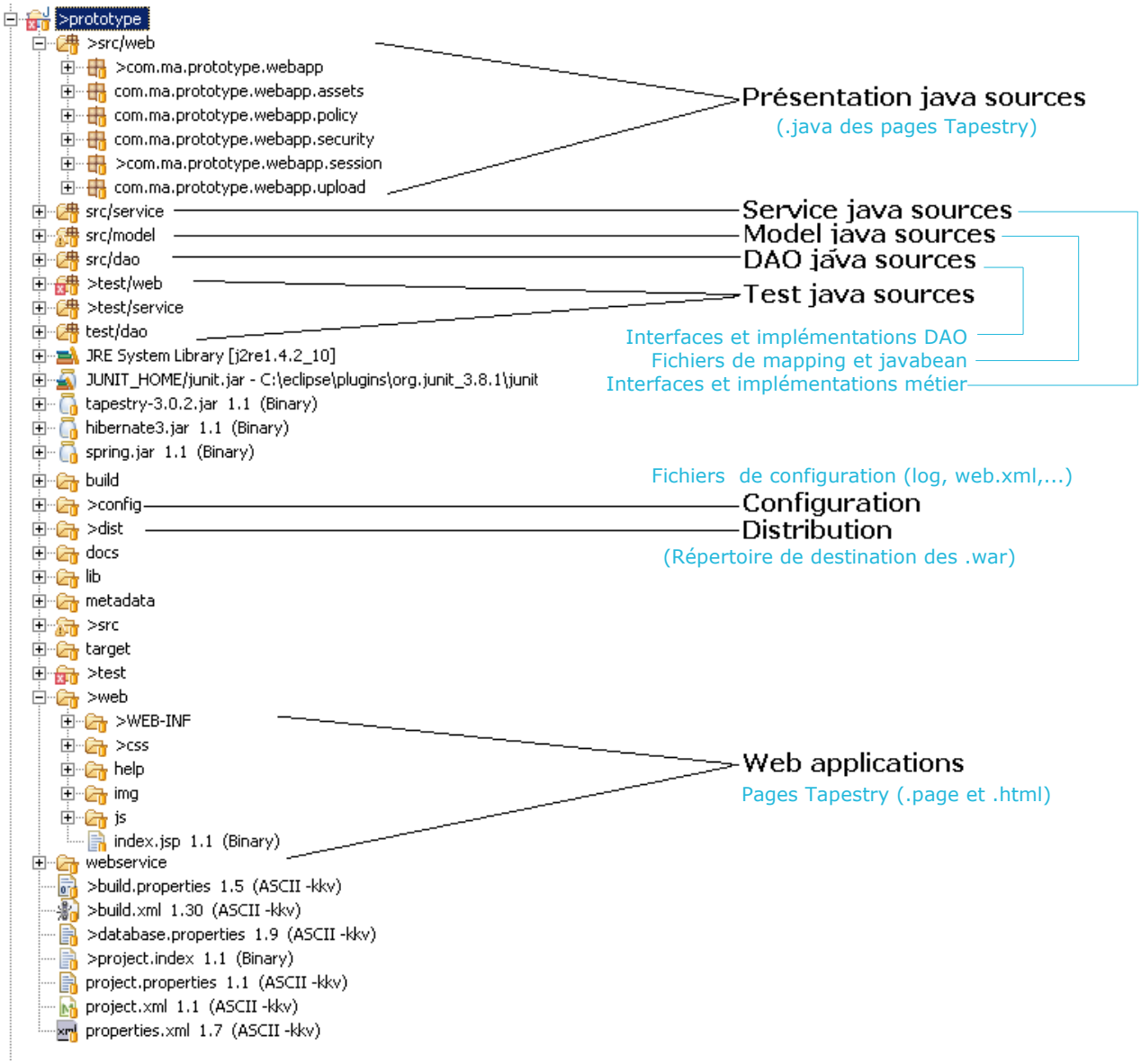
- Les tables sont représentées sous forme de Javabean.

- **La couche d'accès aux données (DAO)**

  - Elle est chargée d'enregistrer et restituer les données de la base.

  - Hibernate génère les requêtes SQL.

Nous nous sommes inspiré d'AppFuse notamment pour l'architecture du projet que voici :



2.3 Tapestry

Tapestry est un *framework* Open Source supporté par la Fondation Apache. Il s'intègre parfaitement avec une variété de conteneurs J2EE comme Tomcat, Jetty ou JBoss.

Tapestry est un des *framework* les plus proches de la séparation contenant/contenu selon l'architecture MVC (*Model*, *View*, *Controler*). En cela, et parce qu'il repose sur un ensemble de composants réutilisables, il permet de créer des pages et des applications web complètement dynamiques et avec une productivité très importante.

Nom	Tapestry
Site internet	http://jakarta.apache.org/tapestry/
Version étudiée	3.0.2
Licence	Apache Software License
Plateformes supportées	Toutes
Tarification	Gratuit

2.4 Spring

Spring est un framework permettant de développer des applications J2EE. Il prend en charge la création et la mise en relation d'objets par l'intermédiaire de fichiers de configuration qui décrivent les objets à fabriquer et les relations de dépendances entre ces objets.

Nom	Spring
Site internet	http://www.springframework.org/
Version étudiée	1.2
Licence	Apache Software License
Plateformes supportées	Toutes
Tarification	Gratuit

2.5 Hibernate

Hibernate est un *framework* Open Source gérant la persistance des objets en base de données relationnelle. Il fait le lien entre la représentation objet des données et sa représentation relationnelle basé sur un schéma SQL.

Nom	Hibernate
Site internet	http://www.hibernate.org/
Version étudiée	3.0
Licence	LGPL
Plateformes supportées	Toutes
Tarification	Gratuit

3 Mise en œuvre

Pour mettre en place ce projet nous nous sommes basé sur projet AppFuse de Matt Raible. Matt Raible est un américain qui a entrepris d'écrire une application exemple qui puisse même servir de fondation pour une application réelle. AppFuse est un projet destiné à accélérer le développement d'application web. De plus, il a implémenté un exemple pour chaque framework MVC : Struts, JSF, Tapestry, SpringMVC, WebWork... Et pour chaque couche de persistance : Hibernate, JDO, Ibatis... Et notamment une version utilisant Spring, Tapestry et Hibernate. Le tout documenté sur le wiki[1] de Matt Raible. Il y a également une démo en ligne sur : https://appfuse.dev.java.net/

Nous allons maintenant voir le rôle de chaque composant et la façon dont il a été mis en œuvre. Pour plus de clarté, nous prendrons comme exemple une page qui affiche la liste des utilisateurs pour un pays.

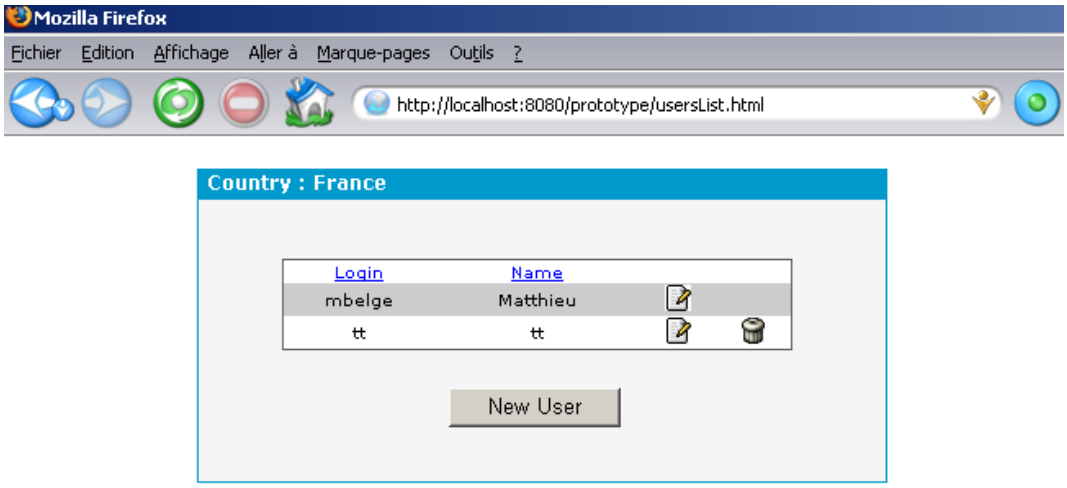


Illustration 1: Liste des utilisateurs pour la France

3.1 Tapestry

Tapestry gère la couche présentation, pour afficher la liste des utilisateurs comme dans l'illustration ci-dessus, nous avons besoin de différentes pages.

- **Page *template***

  Les templates (pages HTML) contiennent la partie statique des pages. En plus de contenir la partie statique, le *template* est également utilisé pour l'emplacement des *tags* de présentation pour la partie dynamique de la page. Les tags sont représenté par l'attribut **"jwcid"**. Cet attribut informe Tapestry qu'une instance de JWC doit être créée et ajoutée à l'emplacement spécifié. L'attribut jwcid est principalement placé dans des blocs comme la balise , de cette manière la page peut être toujours visible même en dehors du contexte de Tapestry. La convention de nommage des *templates*

1 http://raibledesigns.com/wiki/Wiki.jsp?page=Articles

est le nom du composant suivi de l'extension .Html. La liste des composants Tapestry ainsi que leur description se trouve à l'adresse suivante :
http://jakarta.apache.org/tapestry/3.0.3/doc/ComponentReference/index.html

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" lang="fr">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<title><span key="title"/></title>
<link rel="stylesheet" href="css/Prototype.css" type="text/css" />
</head>
<body jwcid="@Body">
<div id="page">
 <div id="contenu">
  <div class="colonnes">
   <dl>
    <dt>
     <span key="country"/> :
     <span jwcid="@Insert" value="ognl:countryToUpdate.name"/>
    </dt>
    <dd>
     <form jwcid="@Form">

      <table jwcid="@contrib:Table" class="list"
       rowsClass="ognl:beans.evenOdd.next"
       row="ognl:row"
       source="ognl:coutryToUpdate.users"

       columns="login:login, name:name, !select, !delete">
       <span jwcid="selectColumnHeader@Block"/>
       <tr jwcid="selectColumnValue@Block">
        <a jwcid="@LinkSubmit" selected="ognl:userToUpdate" tag="ognl:row"
            listener="ognl:listeners.updateUser">
         <img jwcid="@Image" image="ognl:assets.update" alt="message:button.update"/>
        </a>
       </tr>
       <span jwcid="deleteColumnHeader@Block"/>
       <tr jwcid="deleteColumnValue@Block">

        <span jwcid="@Conditional"condition="ognl:userManager.isRemovable(row.login)">
         <a jwcid="@DirectLink" listener="ognl:listeners.removeUser"
             parameters="ognl:row.login">
          <img jwcid="@Image" image="ognl:assets.delete"/>
         </a>
        </span>
       </tr>
      </table>
      <p class="button">
       <input type="submit" jwcid="@Submit" value="message:button.newUser"
            listener="ognl:listeners.newUser"/>
      </p>
     </form>
    </dd>
   </dl>
  </div>
 </div>
</div>
</body>
</html>
```

Tag "Insert" pour afficher la propriété "name" de la Country

Tag formulaire *(obligatoire pour utiliser des tags comme @Submit)*

Composant Tapestry qui génère un tableau à partir d'une source *(ArrayList, tableau)*

Nom de colonne

Propriété à afficher

Ligne sélectionnée de type User passer au listener par l'objet userToUpdate

Tag conditionnel, condition à respecter pour afficher le contenu de la balise

Méthode de la couche métier "UserManager" *(test si l'utilisateur peut être supprimé)*

Soumission du formulaire

UserList.html

- **Page spécification**

  La spécification fait le lien entre la présentation (*template*) et la classe Java. Les objets, leur type et leur emplacement y sont spécifiés. La page spécification définit la classe qui sera utilisée et décrit les composants qui seront repris dans la page HTML. Par défaut la page spécification utilise comme *template* la page html du même nom (ex : UserList.page et UserList.html) toutefois il est possible de désigner un autre template à la page (ex : <context-asset name="$template" path="WEB-INF\accueil.html"/>). La convention de nommage des spécifications est le nom du composant suivie de l'extension .page (pour les page-specification) et .jwc (pour les componant-specification).

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE page-specification
   PUBLIC "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
   "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<page-specification class="com.ma.prototype.webapp.UserList">

   <property-specification name="userManager" type="com.ma.prototype.service.UserManager">
      global.appContext.getBean("userManager")
   </property-specification>

   <property-specification name="countryToUpdate" persistent="yes"
type="com.ma.prototype.model.Country"/>
   <property-specification name="userToUpdate" persistent="yes"
type="com.ma.prototype.model.User"/>

   <property-specification name="row"/>
   <bean name="evenOdd" class="org.apache.tapestry.bean.EvenOdd"/>
   <context-asset name="upArrow" path="img/arrow_up.png"/>
   <context-asset name="downArrow" path="img/arrow_down.png"/>
   <context-asset name="update" path="img/edit.png"/>
   <context-asset name="delete" path="img/delete.png"/>
</page-specification>
```

Classe java qui permet d'afficher la page

Bean qui sera utilisé dans la classe et la page html

Objet de type Country qui sera utilisé dans la page de présentation

Ligne de la table ("@contrib:Table")

Images utilisées dans la page de présentation par le tag "@Image"

UserList.page

- **Classe java**

  C'est une classe Java qui contient les propriétés et les méthodes nécessaires pour les pages Tapestry. La classe charge les objets métier et les transmet à la page. La méthode *pageBeginRender* prépare la page à afficher. L'objet *Visit* de Tapestry gère les variables de session.

```
public abstract class UserList extends BasePage implements PageRenderListener
{
   public abstract UserManager getUserManager();
   public abstract void setUserManager(UserManager manager);
   public abstract Country getCountryToUpdate();
   public abstract void setCountryToUpdate(Country country);
   public abstract User getUserToUpdate();
   public abstract void setUserToUpdate(User user);

   public void pageBeginRender(PageEvent request)
   {
      Map visit = (Map) request.getRequestCycle().getEngine().getVisit();
      if (visit == null)
      {
         this.setSession(request.getRequestCycle());
         visit = (Map) request.getRequestCycle().getEngine().getVisit();
      }
   }

   public void newUser(IRequestCycle cycle)
   {
      User newUser = new User();
      newUser.setCountry(getCountryToUpdate());
      newUser.setCountryId(getCountryToUpdate().getCountryId());
      UserList nextPage = (UserList) cycle.getPage("userModif");
      nextPage.setUserToUpdate(newUser);
      nextPage.setCountryToUpdate(getCountryToUpdate());
      cycle.activate(nextPage);
   }
}
```

Appel des classes métier

Déclaration des objets de la page

Initialisation de la page

Variable de session

Déclaration de la page suivante avec le nom de la page dans le fichier .application *(userModif)*

Initialisation de la page *(on lui passe le User et le Country)*

Activation de la page *"userModif"* si aucune page n'est précisée, la page courante est rechargée

UserList.java

- **Application spécification**

  L'application spécification est utilisée pour décrire l'application au *framework*. Ce fichier fournit le nom de l'application, le moteur, le nom et le chemin des pages qui seront appelées par l'application.

  La page "Home" a une signification particulière, elle est obligatoire dans toutes les applications Tapestry, lors du lancement de l'application, celle-ci charge et affiche la page "Home". Le fichier ".application" est nécessaire dans toutes les applications utilisant Tapestry.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
 "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
 "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<application name="prototype" engine-
class="org.apache.tapestry.engine.BaseEngine">

   <description>add a description</description>

   <page name="Home" specification-path="Home.page"/>
   <page name="userList" specification-path="UserList.page"/>
   <page name="userModif" specification-path="UserModif.page"/>

</application>
```

Nom de l'application

Noms des pages utilisées dans l'application

prototype.application

- **Les composants**

  Tapestry propose de nombreux composants de base comme les champs de formulaire les tableaux, etc.

  La liste des composants de références ainsi que leur documentation sont disponibles à l'adresse suivante :
  http://jakarta.apache.org/tapestry/3.0.3/doc/ComponentReference/index.html

  Dans notre exemple, la table qui affiche la liste des utilisateurs est un composant tapestry "@contrib:Table". Il suffit de lui passer la source, c'est à dire l'ArrayList ou le tableau que l'on souhaite afficher puis le nom ou le rang des colonnes à afficher.

  Il est également possible de créer ses propres composants totalement réutilisables. Pour cela, on procède de la même façon que pour créer une page Tapestry à l'exception de l'extension de la page qui doit être *.jwc* et non plus *.page* et le composant doit être référencé dans un fichier *.library* à la place du fichier *.application*. Par exemple monComposent.jwc est référencé dans le fichier maLibrairie.library pour insérer ce composant dans une page tapestry on l'appelle de la façon suivante :

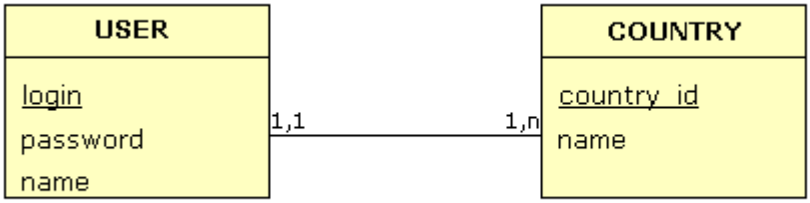
3.2 Hibernate

Hibernate gère la persistance des objets en base de données. Il génère les requêtes SQL et fait le lien entre la base de données et les classes Java. Pour cela il lui faut des fichiers de *mapping* (fichier.hbm.xml) pour lui indiquer quelle classe correspond à quel objet. La partie accès aux données est gérée dans la couche DAO. Il permet au développeur de gagner du temps en gérant à sa place les requêtes.

3.2.1 Le *mapping*

Les fichiers de *mapping* hibernate sont des fichiers *xml* qui représentent la base de données. Le fichier de *mapping* indique à Hibernate à quelle table dans la base de données il doit accéder, et quelles colonnes de cette table il devra utiliser.

Voici un exemple de fichier de *mapping* Hibernate pour les tables USER et COUNTRY :



Les fichiers de *mapping* contiennent le nom de la table ainsi que le nom de la classe correspondante, le nom et le type de chaque propriété ainsi que le nom de la colonne dans la table, la taille du champ n'est pas obligatoire. Les clés primaires sont représentés par la balise *<id>.* Pour les clés composées, on utilise la balise *<composite-id>* suivie du nom de la clé et de la classe. Chaque clé composée est également une entité, une classe Java.

```
<composite-id name="comp_id" class="com.ma.prototype.model.EntreprisePK">
      <key-property
         name="entrepriseId"
         column="ENTREPRISE_ID"
         type="java.lang.String"
         length="5"
      />
      <key-property
         name="countryId"
         column="COUNTRY_ID"
         type="java.lang.String"
         length="5"
      />
</composite-id>
```

Exemple de clé composée

Dans le fichier de *mapping* de la table User, la relation 1,1 avec la table Country est traduite de la façon suivante avec une propriété "*many-to-one*" le fichier User contient également en propriété la clé primaire de la table Country. La classe User contiendra une propriété de type Country.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >

<hibernate-mapping>

<class
   name="com.ma.prototype.model.User"
   table="USER"
>

   <id
      name="login"
      type="java.lang.String"
      column="LOGIN"
   >
      <generator class="assigned" />
   </id>

  <property
      name="password"
      type="java.lang.String"
      column="PASSWORD"
      length="32"
   />

   <property
      name="name"
      type="java.lang.String"
      column="NAME"
      length="255"
   />

   <property
      name="countryId"
      type="java.lang.String"
      column="COUNTRY_ID"
      length="5"
   />

   <!-- bi-directional many-to-one association to Country -->
   <many-to-one
      name="country"
      class="com.ma.prototype.model.Country"
      not-null="true"
      insert="false"
      update="false"
   >
      <column name="COUNTRY_ID" />
   </many-to-one>

</class>
</hibernate-mapping>
```

User.hbm.xml

Nous allons maintenant voir le *mapping* de la table Country. Dans le fichier de *mapping* de la table Country la relation 1,n "*one-to-many*" avec la table User est traduite par une propriété "*bag*", la classe Country contiendra une liste de type User.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" >

<hibernate-mapping>

<class
   name="com.ma.prototype.model.Country"
   table="COUNTRY"
>

   <id
      name="countryId"
      type="java.lang.String"
      column="COUNTRY_ID"
   >
      <generator class="assigned" />
   </id>

   <property
      name="name"
      type="java.lang.String"
      column="NAME"
      length="200"
   />

   <!-- bi-directional one-to-many association to User -->
   <bag
      name="users"
      lazy="true"
      inverse="true"
      cascade="none"
   >
      <key>
         <column name="COUNTRY_ID" />
      </key>
      <one-to-many
         class="com.ma.prototype.model.User"
      />
   </bag>

</class>
</hibernate-mapping>
```

Country.hbm.xml

Hibernate utilise ensuite le *mapping* lors de la création des requêtes dans la couche d'accès aux données (DAO). Les fichiers de *mapping* doivent être crées pour chaque table utilisée, il existe des outils permettant de les générer automatiquement tel que MiddleGen ou encore XDoclet. Dans notre cas nous avons utilisé MiddleGen.

- **MiddleGen**

MiddleGen est un outil permettant de générer les fichiers de mapping Hibernate directement à partir d'un schéma de base de données existant. C'est un outil simple d'utilisation et rapide qui fait gagner beaucoup de temps puisqu'il évite d'écrire chaque fichier manuellement. Cependant il ne gère pas correctement les clés composées.

3.2.2 DAO

La partie DAO gère l'accès aux données, Hibernate fait le lien entre les classes Java et les tables en base. Le langage Hibernate est appelé HQL (Hibernate Query Language). Le langage HQL est un langage qui ressemble beaucoup au langage SQL à l'exception prêt que le HQL est orienté objet, il comprend donc les notions d'héritage, d'association... Voici un exemple de fichier DAO qui contient des requêtes Hibernate.

```
public class UserDAOHibernate extends BaseDAOHibernate implements UserDAO {

   /* Retourne un "User" par rapport à son identifiant "login" */
   public User getUser(String login) {
      User user = (User) getHibernateTemplate().get(User.class, login);
      return user;
   }

   /* Retourne la liste complète des "User" */
   public List getUsers(User user) {
      return getHibernateTemplate().find("from User");
   }

   /* Retourne une liste des "User" par rapport au pays passé en paramètre */
   public List getUsersByCountry(String countryId)
   {
      return getHibernateTemplate().find("from User where countryId = ?", countryId);
   }

   /* Retourne un tableau contenant le login le nom de l'utilisateur et le nom du pays par rapport
      au pays passé en paramètre */
   public List getUsersCountrys(String countryId)
   {
      return getHibernateTemplate().find("select u.login, u.name, u.country.name from User u
      where u.countryId = ?", countryId);
   }
```

UserDAOHibernate.java

Hibernate traduit la requête HQL en SQL compréhensible par la base de données, la requête "select u.login, u.name, u.country.name from User u where u.countryId = ?" qui utilise le nom des objets est traduite de la façon suivante "select u.login, u.name, c.name from user u, country c where u.country_id = c.country_id and u.country_id = ?" qui utilise les noms de la base de données. Nous précisons juste le nom de l'objet que l'on souhaite et Hibernate se charge de créer la jointure lui même.

Pour les actions de base *(insert, update, delete)* sans paramètre particuliers, on utilise le DAO de base d'Hibernate *(BaseDAO)*. Ces méthodes prennent en paramètre l'objet à modifier (*ex : getHibernateTemplate.save(myUser);*) pour insérer un nouvel utilisateur.

Pour indiquer à Hibernate quel est le SGBD utilisé, il faut lui renseigner la propriété *hibernate.dialect* dans le fichier le configuration. Ici nous utilisons une base de données Oracle <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>. Dans notre projet, les fichiers de configuration sont géré par Spring.

3.3 Spring

Spring prend en charge la création et la mise en relation d'objets par l'intermédiaire de fichiers de configuration qui décrivent les objets à fabriquer et les relations de dépendances entre ces objets. Il permet ainsi de faire le lien entre les diverses couches du projet pour que la partie métier communique avec la partie DAO, la partie DAO avec la partie Modèle....

ACEGI est un framework de sécurité lié à Spring, nous l'avons utilisé pour gérer l'authentification et les autorisations. La mise en place d'Acegi se fait dans le fichier de configuration applicationContext-security.xml géré par Spring et les filtres à appliquer sont indiqués dans le fichier web.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app >
   ...
   <filter>
      <filter-name>acegiSessionFilter</filter-name>
      <filter-class>com.ma.prototype.session.AcegiFilter</filter-class>
   </filter>
         <filter>
      <filter-name>Acegi Authentication Processing Filter</filter-name>
      <filter-class>net.sf.acegisecurity.util.FilterToBeanProxy</filter-class>
      <init-param>
         <param-name>targetClass</param-name>
         <param-value>net.sf.acegisecurity.ui.webapp.AuthenticationProcessingFilter</param-
          value>
      </init-param>
   </filter>
   <filter>
      <filter-name>Acegi HTTP Request Security Filter</filter-name>
      <filter-class>net.sf.acegisecurity.util.FilterToBeanProxy</filter-class>
      <init-param>
         <param-name>targetClass</param-name>
         <param-value>
                  net.sf.acegisecurity.intercept.web.SecurityEnforcementFilter
        </param-value>
      </init-param>
   </filter>
   <filter>
      <filter-name>acegiRemoteUserFilter</filter-name>
      <filter-class>net.sf.acegisecurity.wrapper.ContextHolderAwareRequestFilter</filter-class>
   </filter>
   ...
</web-app>
```

Filtre sur les URL

Web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

   <bean id="jdbcAuthenticationDao"
     class="com.ma.prototype.dao.hibernate.security.HibernateBUAuthenticationDAO"
   autowire="byName"/>

   <bean id="filterInvocationInterceptor"
     class="net.sf.acegisecurity.intercept.web.FilterSecurityInterceptor">
      <property name="authenticationManager"><ref
        local="authenticationManager"/></property>
      <property name="accessDecisionManager"><ref
        local="accessDecisionManager"/></property>
       <property name="objectDefinitionSource">
                <ref local="objectDefinitionSource"/>
      </property>
   </bean>

        <bean id="objectDefinitionSource"
          class="com.ma.prototype.session.ObjectDefinitionSource">
        <property name="secureUrl">
                <list>
           <value>not/login.html*=Foo</value>
           <value>/*.html*=USER</value>
                </list>
        </property>
        <property name="convertUrlToLowercaseBeforeComparison">
                <value>true</value>
        </property>
        </bean>

   <bean id="managerSecurity"
       class="net.sf.acegisecurity.intercept.method.aopalliance.MethodSecurityInterceptor">
     <property name="authenticationManager"><ref bean="authenticationManager"/></property>
     <property name="accessDecisionManager"><ref local="accessDecisionManager"/></property>
  </bean>

  ...
</beans>
```

Accès aux pages (login.html* = Foo => accessible à tout le monde, /*.html*=USER => accessible uniquement aux personnes authentifiées)

ApplicationContext-security.xml

La configuration d'Hibernate et la relation entre la partie DAO et la partie modèle sont gérées par Spring dans le fichier *applicationContext-hibernate.xml.* Ce fichier contient également les paramètres de connexion à la base de données.

```
<?xml version="1.0" encoding="UTF-8" ?>
 <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">
 <beans default-lazy-init="false" default-dependency-check="none" default-autowire="no">

 <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource" >
 <property name="driverClassName">
        <value>oracle.jdbc.driver.OracleDriver</value>
 </property>
 <property name="username">
        <value>user</value>
 </property>
 <property name="password">
        <value>user</value>
 </property>
 <property name="url">
        <value>jdbc:oracle:thin:@127.0.0.1:1521:DATA</value>
 </property>
 </bean>

 <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource">
 <ref bean="dataSource" />
  </property>
 <property name="mappingResources">
 <list>
  <value>com/ma/prototype/model/User.hbm.xml</value>
  <value>com/ma/prototype/model/Country.hbm.xml</value>
  ...
  </list>
  </property>

 <property name="hibernateProperties">
 <props>
  <prop key="hibernate.dialect">org.hibernate.dialect.OracleDialect</prop>
  <prop key="hibernate.use_outer_join">true</prop>
  ...
  </props>
  </property>
  </bean>

 <bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
 <property name="sessionFactory">
  <ref local="sessionFactory" />
  </property>
  </bean>

  <bean id="uesrDAO" class="com.ma.prototype.dao.hibernate.UserDAOSPImpl" />
  <bean id="countryDAO" class="com.ma.prototype.dao.hibernate.CountryDAOHibernate" />
  ...
  </beans>
```

Paramètres de connexion à la base de données

Fichiers de mapping

- SGBD utilisé
- Configuration d'Hibernate

Définitions des beans DAO (Nom et classe implémenté)

ApplicationContext-hibernate.xml

Enfin, la relation entre la partie métier et la partie DAO se fait dans le fichier de configuration *applicationContect-service.xml* que voici :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
   "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
   <bean id="txProxyTemplate" abstract="true"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
      <property name="transactionManager"><ref bean="transactionManager"/></property>
      <property name="transactionAttributes">
         <props>
            <prop key="insert*">PROPAGATION_REQUIRED</prop>
            <prop key="update*">PROPAGATION_REQUIRED</prop>
            <prop key="remove*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
         </props>
      </property>
   </bean>

   <bean id="userManager" parent="txProxyTemplate">
      <property name="target">
         <bean class="com.ma.prototype.service.impl.UserManagerImpl">
                  <property name="dao"><ref bean="baseDAO"/></property>
                  <property name="userDAO"><ref bean="userDAO"/></property>
         </bean>
       </property>
   </bean>

   <bean id="countryManager" parent="txProxyTemplate">
      <property name="target">
         <bean class="com.ma.prototype.service.impl.CountryManagerImpl">
                  <property name="dao"><ref bean="baseDAO"/></property>
                  <property name="countryDAO"><ref bean="countryDAO"/></property>
         </bean>
      </property>
   </bean>
   ...
</beans>
```

Définition des beans Service (Nom et classe implémenté, liste des DAO utilisés)

ApplicationContext-service.xml

4 ANALYSE

4.1 Réponse des *framework* aux objectifs

Les *framework* que l'on a mis en place ont globalement bien répondu aux objectifs que l'on avait fixés.

4.1.1 Avoir des temps de développements les plus optimum possibles

Les temps de développement ont été globalement bons. Le mise en place des *framework* a été très rapide grâce à l'utilisation de AppFuse et MiddleGen.

L'appropriation des différents *framework* par les développeurs ne connaissant pas ces *framework* a été plus ou moins longue.

- Hibernate et Spring ont été rapides à prendre en main, les exemples, les tutoriels et la documentation sont nombreux.
- La prise en main de Tapestry a été plus complexe. Le framework s'appuie sur des concepts propres que l'on ne retrouve pas dans d'autres frameworks de présentation ou dans la norme Servlet. Il y a un langage de *template* propre à s'approprier quand on ne connaît que les JSP et les Taglibs. La documentation de Tapestry est assez vaste mais pas forcément très bien faite. On a du mal à trouver rapidement une réponse à la question précise que l'on se pose.

Une fois les *framework* pris en main, les temps de développement ont été très bons, principalement grâce à la simplicité d'écriture en base de données offert par Hibernate et l'intégration entre Spring et Hibernate. La gestion poussée des formulaires de Tapestry a également permis d'être performant sur la mise en place des interfaces graphiques.

4.1.2 Avoir des performances honorables sur l'application développée

Hibernate génère lui même les requêtes SQL à partir des objets utilisés. Par défaut Hibernate remonte toutes les lignes d'une table. Lors de la phase de développement ces problèmes ne sont pas gênant, les base de données étant souvent peu rempli. Mais dès qu'on passe sur des bases qui ont un plus gros volumes les temps de réponse deviennent inacceptables. Une phase d'optimisation portant sur les requêtes les plus longues et sur les fichiers configuration a été nécessaire. Le résultat final a permis d'obtenir les performances voulues.

4.1.3 Avoir une application robuste

L'utilisation de ces 3 *framework* requiert l'utilisation d'un grand nombre de fichiers de configuration XML qui ont l'avantage de permettre une configuration simple mais le désavantage de ne pas remonter certaines erreurs à la compilation.

Cependant l'application n'a pas eu de comportement surprenant lors de la montée en

charge.

4.1.4 Pouvoir capitaliser les éléments réutilisables

Le modèle de composants de Tapestry a permis de développer des composants génériques réutilisés à différents endroits de l'application et capitalisés pour être utilisés dans d'autres applications.

4.1.5 Développement en couches

Le pattern DAO utilisé a permis de bien isoler la couche métier de l'accès aux données.

Tapestry est un framework MVC (Modèle, Vue, Contrôleur) qui permet de séparer la logique de présentation de la logique métier.

Spring a permis de brancher la sécurité et les aspects transactionnels indépendamment du reste de l'application avec de la programmation orientée aspect (AOP).

4.1.6 Avoir à disposition des briques utiles

Une des caractéristiques de Spring est de ne pas réinventer la roue. Ce *framework* intègre un grand nombre de librairies bas niveaux mais très utiles qui sont directement utilisables dans les développements.

Tapestry utilise son propre système de vue, avec son propre langage de *template* ce qui empêche l'utilisation des librairies de Taglib qui sont souvent utiles dans les développement web. Une partie d'entre elles ont du être ré implémentés en composants Tapestry pour les besoins du projet.

4.1.7 Disposer d'un cadre de travail

Les trois *framework* offrent un réel cadre de travail (ce qui est quand même la première définition de framework). Les développeurs sont poussés à développer de la meilleure manière.

Le découpage des paramètres de configuration en XML en autant du fichiers que l'on veut permet un travail collaboratif efficace si l'on utilise un CVS.

4.2 Bilan sur Spring, Tapestry et Hibernate

4.2.1 Tapestry est puissant mais trop isolé

Tapestry est très bien adapté pour faire des applications de gestion. La gestion des formulaires est plus poussée que dans Struts avec une gestion objet des champs du formulaire.

Le modèle en composants est très puissant pour réutiliser simplement du code. Le fait que l'on puisse embarquer les *templates* dans des JAR permet de constituer des bibliothèques sous forme d'archives facilement déployables.

Le principal reproche que l'on fera au *framework* est qu'il n'a pas la dynamique des *framework* basés sur JSP pouvant s'appuyer sur des Taglibs externes. Il faut redévelopper des

composants qui existent depuis longtemps en JSP.

4.2.2 Hibernate est un réel gain de temps

Hibernate permet de gagner énormément de temps sur la partie écriture et récupérations d'objet par ID. La souplesse de la configuration permet de « tuner » l'application sans avoir à retoucher au code. Quant au langage de requête, il est suffisamment puissant pour exprimer des requêtes très complexes et profiter de la puissance de la base de données.

Pour cela, il offre plus de sécurité qu'une persistance géré par un serveur J2EE qui offre moins de souplesse en cas de problèmes de performances.

4.2.3 Spring offre une grande évolutivité

Spring offre une intégration avec les autres *framework* très appréciable, notamment l'intégration avec Hibernate. Le développement de comportements par la programmation aspect permet de pouvoir faire évoluer rapidement les applications tout au long de leur cycle de vie. Par rapport à un conteneur J2EE, Spring n'offre pas toutes les fonctionnalités mais permet de développer beaucoup plus rapidement en ne proposant que les fonctionnalités les plus utiles (méthodes transactionnelles, gestion de la sécurité via Acegi ...).