Introduction au Framework Spring

1. De quoi s'agit-il ?

SPRING est effectivement un conteneur dit « léger », c'est-à-dire une infrastructure similaire à un serveur d'application J2EE. Il prend donc en charge la création d'objets et la mise en relation d'objets par l'intermédiaire d'un fichier de configuration qui décrit les objets à fabriquer et les relations de dépendances entre ces objets.

Le gros avantage par rapport aux serveurs d'application est qu'avec SPRING, vos classes n'ont pas besoin d'implémenter une quelconque interface pour être prises en charge par le framework (au contraire des serveurs d'applications J2EE et des EJBs). C'est en ce sens que SPRING est qualifié de conteneur « léger ».

Outre cette espèce de super fabrique d'objets, SPRING propose tout un ensemble d'abstractions permettant de gérer entre autres :

- Le mode transactionnel
- L'appel d'EJBs
- La création d'EJBs
- La persistance d'objets
- La création d'une interface Web
- L'appel et la création de WebServices

Pour réaliser tout ceci, SPRING s'appui sur les principes du design pattern IoC et sur la programmation par aspects (AOP).

2. Pourquoi vous avez besoin de SPRING ? Le pattern IoC

Le coeur de SPRING et ce qui fait sa très grande force est la mise en œuvre du design pattern « Inversion Of Control » ou encore « Dependency Injection » décrit par Martin Fowler (**http://www.martinfowler.com/articles/injection.html**). Vous trouverez une présentation de ce design pattern en français sur notre site à cette adresse**http://smeric.developpez.com/java/uml/avalon/ioc/**; aussi, je me contenterai d'une description succincte dans cet article.

2.1. Le pattern IoC

L'idée du pattern IoC est très simple, elle consiste, lorsqu'un objet A a besoin d'un objet B, à déléguer à un objet C la mise en relation de A avec B. Bon, ok, cela ressemble à une vieille équation d'algèbre incompréhensible alors un petit exemple de code vaut mieux qu'une formule fumeuse.

Prenons l'exemple d'un code classique où une classe a besoin de sa fabrique pour gérer les accès à la base de données. Le code suivant est assez classique, sachant qu'un effort a déjà été fait pour centraliser la création des différentes fabriques au sein d'une fabrique générale.

Product

```
package product;

import factory.DAOFactory;
import product.dao.ProductDAO;

public class Product{

    private ProductDAO dao;

    private long id;
    private String name;
    private String description;

    public Product(){
            dao = (ProductDAO)DAOFactory.getFactory("ProductDAO");
    }

    public String getName(){return name;}

    //etc
}
```

DAOFactory

```
package factory;

import product.dao.ProductDAOImpl;
import product.dao.ClientDAOImpl;
import product.dao.CommandDAOImpl;

import java.util.*;

public class DAOFactory{

    private Hashtable factories;
    private static DAOFactory self = null;

    protected DAOFactory(){
            factories = new Hashtable();
            factories.put("ProductDAO", new ProductDAOImpl());
            factories.put("ClientDAO", new ClientDAOImpl());
            factories.put("CommandDAO", new CommandDAOImpl());
    }

    public static Object getFactory(String factoryName) throws
NullPointerException {
            return DAOFactory.self().get(factoryName);
    }

    protected Object get(String factoryName) throws NullPointerException {
            return factories.get(factoryName);
```

```
    }

    public static synchronized DAOFactory self(){
            if(self == null){
                    self = new DAOFactory();
            }
            return self;
    }
}
```

Quels constats simples peut-on faire sur cette conception ?

- Le code dépend d'un appel à la classe DAOFactory
- Les implémentations des différents DAO sont déclarées en dur dans la classe DAOFactory

Mais alors, quels problèmes cela pose t-il ?

Pb1 - Comment faire du test quand les classes implémentant les DAO ne sont pas encore développées ?

Pb2 - Comment faire pour que l'application puisse fonctionner avec des classes DAO différentes dans différents contextes (contexte serveur centralisé, contexte distribué, contexte « phase de test », contexte « production ») ?

Solution au problème 1

La solution est ici de modifier le code de la classe DAOFactory, ok. Mais comment faire pour partager ce code modifié alors que plusieurs développeurs n'ont pas les mêmes exigences en termes de création de fabriques ?

Peut-on aussi accepter d'être obligé de modifier du code qui marche lors du passage à différentes phases de test ?

Solution au problème 2

Le cas présenté est simple mais imaginez que certaines classes DAO soient des classes distribuées, déployées au sein d'un serveur d'application. La création de ces classes devrait se faire via une requête JNDI. Ok, on peut donc modifier le code de la classe DAOFactory pour ces classes en particulier. Mais alors, comment faire du test lorsque ces classes ne sont pas encore développées ? Et d'une manière générale, ne peut-on pas trouver plus simple et moins consommateur en temps de réaliser des tests hors contexte d'un serveur d'application et donc remplacer, lors des tests, les classe XXXDAOImpl par des classes Java classiques (des « mocked » objets, les bouchons en français dans le texte) ?

Avec SPRING, tous ces problèmes sont externalisés et gérés par le framework. Il suffit donc d'écrire le code suivant :

Product

```
package product;

import factory.DAOFactory;
import product.dao.ProductDAO;

public class Product{

    private ProductDAO dao;
```

```
    private long id;
    private String name;
    private String description;

    public void setProductDAO(ProductDAO dao){
            this.dao = dao;
    }

    public String getName(){return name;}

    //etc
}
```

On voit ici que notre classe Product ne dépend d'aucune fabrique et qu'elle ne fait que déclarer un « setter » pour sa propriété « dao ».

Le framework SPRING, via le fichier de configuration suivant, prend en charge la mise en relation des différents objets :

```
<beans>
    <bean id="productDAO" class="product.dao.ProductDAOImpl"></bean>
    <bean class="product.Product">
            <property name="productDAO">
                    <ref bean="productDAO"/>
            </property>
    </bean>
</beans>
```

Pour répondre aux problèmes soulevés précédemment, il suffit de gérer des fichiers de configuration spécifiques pour les tests et un fichier de configuration spécifique pour la mise en production de l'application (chacun déclarant des XXXDAOImpl adaptés aux contexte).

Que les objets soient créés par un simple « new » ou via une recherche dans un annuaire JNDI, ce qui peut être le cas pour des composants EJBs, n'a aucun impact sur le code applicatif.

Vous le comprendrez mieux après avoir lu le chapitre sur les EJBs, mais avec SPRING, il est possible de créer un code applicatif qui ne manipule que des classes Java classiques (ce que l'on appelle des POJOs : Plain Old Java Objects) sans être obligé de maîtriser tous les aspects techniques des technologies que vous mettez en oeuvre ; SPRING se charge de la « clue » technique.

3. Configurer une application

3.1. Les principes

Nous en avons parlé très brièvement dans le chapitre précédent, un utilisateur classique3 de SPRING va utiliser un fichier de configuration XML pour décrire les différents objets devant être gérés par le conteneur de SPRING.

Avec SPRING, on va gérer des JavaBeans ou Beans; cela n'a rien de vraiment contraignant, le terme JavaBeans est juste utilisé par SPRING car SPRING utilise les « setters » pour mettre des objets en relation. Vos objets sont donc des JavaBeans !

Ceci étant dit, on va donc déclarer des « BeanFactory », des fabriques de JavaBean, dans le fichier de configuration pour expliquer à SPRING que l'on a un certain nombre d'objets à créer au cours de notre application.

```
<beans>
    <!-- Première fabrique-->
    <bean id="productDAO" class="product.dao.ProductDAOImpl"></bean>
    <!-- Seconde fabrique-->
    <bean class="product.Product">
            <property name="productDAO">
                    <ref bean="productDAO"/>
            </property>
    </bean>
</beans>
```

Les « property » dans un bean servent donc à déclarer les relations de dépendance entre beans. Dans notre exemple, on déclare donc que les objets de type « Product » ont besoin d'un objet référencé dans le fichier de configuration sous le nom « productDAO ». Lors de la création d'un « Product », SPRING affecte automatiquement sa propriété « productDAO » en utilisant la méthode setProductDAO et en lui passant, ici, un objet de type ProductDAOImpl.

3.1.1. Cycle de vie d'un Bean

Par rapport à la problématique d'instanciation d'objet dans une application, on a des classes qui donnent naissance à plusieurs instances et d'autres classes, déclarées comme des singletons, qui n'autorisent que la création d'une instance unique.

Par défaut, SPRING suppose que les fabriques déclarées sont des fabriques de type « singleton ».

Dans notre exemple, si on imagine qu'un autre bean « type X » fasse comme le bean « Product », c'est-à-dire fait référence au bean « productDAO », lors de la création d'une instance de « type X », SPRING retournera le même productDAO que celui retourné à l'objet de type « Product ».

Dans le cas de notre bean « Product », vous devez sûrement vous dire qu'il est peu probable que l'application n'utilise qu'une seule instance de Product. Effectivement, ce bean n'est sûrement pas un singleton. Pour ce type de bean (dit « Prototype »), SPRING propose de déclarer un attribut « singleton » que l'on peut positionner à « false ».

```
<bean class="product.Product" singleton="false">
        <property name="productDAO">
            <ref bean="productDAO"/>
        </property>
</bean>
```

Donc résumons un peu, SPRING gère 2 types de Beans, des **Singletons** et des **Prototypes**. Les singletons sont tous créés à l'initialisation du framework (sauf spécification d'un attribut lazy-init=true) et les prototypes sont eux créés à la demande, lors d'un appel comme :

```
« springfactory ».getBean(«product») ;
```

3.1.2. Déclaration des propriétés

C'est ici qu'entre en action notre pattern IoC. Les propriétés d'un bean permettent à ce bean de déclarer un besoin en terme de configuration à SPRING. On l'a vu dans notre exemple, le bean « Product » déclare une propriété de nom « productDAO » et fait référence à un autre bean « ProductDAO » pour identifier la valeur voulue pour cette propriété (utilisation de la notion de « ref »). Pour chaque propriété déclarée, la classe doit proposer un « setter » respectant la convention de nommage des JavaBeans.

SPRING, outre la référence à un autre bean déclaré dans le fichier de configuration, propose plusieurs formes pour la déclaration d'une propriété. Chaque forme étant adaptée à un besoin particulier comme :

- Positionnement d'une valeur par défaut
- Passage de paramètres de configuration (login, password, nom JNDI, URL, classes d'une driver)
- Initialisation de tableaux ou de listes

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
    <property name="driverClassName">
        <value>com.mysql.jdbc.Driver</value>
    </property>
    <property name="url">
        <value>jdbc:mysql://localhost/spring</value>
    </property>
    <property name="username">
        <value>sample</value>
    </property>
    <property name="password">
        <value>sample</value>
    </property>
</bean>
```

```
<bean id="mySessionFactory"
class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="mappingRessources">
            <list>
                    <value>sample.hbm.xml</value>
            </list>
    </property>
</bean>
```

3.2. Accéder au fichier de configuration

L'accès et la lecture du fichier de configuration correspondent à l'initialisation du « container » SPRING. En fonction du contexte d'utilisation de SPRING, plusieurs moyens sont mis à votre disposition pour charger le fichier de configuration de vos beans et ainsi initialiser vos singletons (vous vous souvenez bien sûr de nos fameux singletons initialisés dès le chargement du framework !).

3.2.1. Cas d'une application type client « Swing »

```
ClassPathResource res = new ClassPathResource("applicationContext.xml");
// Notre fabrique SPRING permettant l'accès aux beans déclarés
XmlBeanFactory factory = new XmlBeanFactory(res);
```

Le fichier de configuration « applicationContext.xml » doit être accessible via le CLASSPATH.

3.2.2. Cas d'une application web (war)

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/daoContext.xml /WEB-
INF/applicationContext.xml</param-value>
</context-param>

<listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<!-- Ou utilisation de la servlet ContextLoaderServlet-->
<servlet>
    <servlet-name>context</servlet-name>
    <servlet-
class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

Dans le cas de l'usage de la servlet ContextLoaderServlet, le fichier par défaut recherché est /WEB-INF/applicationContext.xml.

3.2.3. Cas d'un EJB (« ejbjar »)

Dans ce cas, il faut définir la variable d'environnement java:com/env/ejb/BeanFactoryPath dans le fichier ejb-jar.xml pour que SPRING y trouve automatiquement un fichier applicationContext.xml et l'utilise pour initialiser les beans à utiliser dans le contexte de l'ejbjar.

La configuration précédente est utilisée lorsque les EJBs que vous créés utilisent les classes abstraites fournies par SPRING, classes qui facilitent la création d'EJB dans le contexte SPRING.

```
public class MyComponentEJB extends AbstractStatelessSessionBean
implements IMycomponent{

}
```

Le problème de cette solution est que chaque EJB de votre ejbjar va posséder sa propre copie du fichier de configuration et donc sa propre copie de la fabrique SPRING. Dans le cas d'une initialisation coûteuse de tous vos singletons, cette solution peut être totalement rédhibitoire.

Une autre solution est donc envisageable et permet à tous les ejbs d'un ejbjar de partager la même fabrique SPRING. Pour cela, vous devez créer un fichier beanRefFactory.xml contenant la définition de vos beans, rendre ce fichier accessible via le classpath de votre ejbjar ou via le path définit dans la variable d'environnement ejb/BeanFactoryPath et enfin écrire les lignes de code suivantes dans le code de votre ejb :

```
public void setSessionContext(SessionContext sessionContext){
    super.setSessionContext(sessionContext);
    setBeanFactoryLocator(ContextSingletonBeanFactoryLocator.getInstance());
}
```

3.3. Utiliser la configuration au « runtime »

Comme nous l'avons présenté précédemment, les beans déclarés « singleton » sont créés dès le chargement du fichier de configuration. Ensuite, votre besoin peut être de récupérer un premier bean, comme on créé un premier objet dans un programme principal Java classique.

Pour cela, SPRING offre une API permettant de manipuler le fichier de configuration et en particulier de récupérer un objet, singleton ou prototype. Dans le cas de la récupération d'un singleton, SPRING retourne toujours la même instance, celle créée au départ et dans le cas d'un prototype, SPRING se charge du « new » de la nouvelle instance que vous demandez et se charge bien sûr de mettre cette nouvelle instance en relation avec les autres objets dont elle a besoin via le bon vieux principe de l'inversion de contrôle gérée par SPRING.

```
ClassPathResource res = new ClassPathResource("applicationContext.xml");
// La fabrique SPRING est chargée, les singletons sont créés
XmlBeanFactory factory = new XmlBeanFactory(res);

// On utilise la méthode getBean en passant le nom du bean pour créer
// ou récupérer un bean déclaré dans le fichier de configuration
IVehiculeServices vservices =
(IVehiculeServices)factory.getbean("myVehiculeServices");
```

4. Le mode transactionnel

Si votre application utilise une ou plusieurs bases de données, vous avez nécessairement besoin de déclarer des transactions. Bon, ok on peut essayer de s'en sortir avec les fonctionnalités des drivers JDBC mais dès que l'on doit gérer des services transactionnels qui s'imbriquent, ça commence à se compliquer et le code de vos services commence à être pas mal pollué par la gestion des transactions.

Dans le monde J2EE et en particulier dans le monde des EJBs, une solution a été trouvée pour résoudre cette problématique. Quand vous voulez qu'une méthode d'un EJB soit transactionnelle, c'est-à-dire par exemple que lors de l'appel à cette méthode, une transaction est automatiquement démarrée et quand votre méthode se termine, un « commit » est automatiquement réalisé, il suffit de déclarer la fameuse méthode « transaction required » dans le fichier de configuration de votre EJB.

Tout cela est très bien sauf que vous êtes obligé d'écrire un EJB et donc vous êtes obligé d'utiliser un serveur d'application J2EE.

Eh bien vous me croirez ou non mais SPRING y sait faire pareil en mieux !

Le « mieux » c'est parce que SPRING vous permet de déclarer **n'importe quel méthode de n'importe quelle classe** comme étant **transactionnelle**. Je veux dire que vous pouvez prendre votre petite classe Java classique toute simple et déclarer que toutes ses méthodes sont transactionnelles sans une seule ligne de code.

Pour réaliser cette prouesse, SPRING utilise les principes de la programmation par Aspects et la capacité fournie par Java de permettre la création de « dynamic proxies ». Je ne parlerai pas dans ce premier article des détails de SPRING AOP aussi, si vous ne connaissez pas les principes de l'AOP (Aspect Oriented Programming) et des « dynamic proxies », sachez que les proxy c'est comme « Canada Dry » ; un proxy, ça ressemble à votre objet mais ce n'est pas votre objet.

Non, sans rire, l'idée est que l'on peut créer dynamiquement un objet qui possède les mêmes caractéristiques que l'objet que vous avez demandé à la fabrique SPRING et tout cela sans que l'objet ayant fait la demande en sache quoi que ce soit. Tout appel à une méthode de l'objet demandé peut alors être « interceptée », avant et après l'appel, pour pouvoir exécuter du code particulier.

Dans le cas qui nous intéresse ici, le transactionnel, tout appel à une méthode déclarée transactionnelle, va donc être intercepté et c'est SPRING qui va se charger d'exécuter du code pour démarrer la transaction et terminer la transaction.

Prenons un exemple simple. Une classe **VehiculeServices** implémente l'interface **IVehiculeServices** et offre ainsi un ensemble de services au dessus d'une base de données qui gère des véhicules. Toutes les méthodes des classes implémentant l'interface IVehiculeServices doivent être transactionnelles et on ne veut écrire que du code SQL simple dans les classes implémentant cette interface, notre classe VehiculeServices ici.

Dernier point, mon singleton de type IVehiculeServices doit être connu sous le nom « myVehiculeServices ».

Les étapes à suivre sont les suivantes :

- Déclaration d'un « TransactionInterceptor » nous permettant de dire quelles sont les méthodes que l'on veut déclarer transactionnelles. Ici toutes les méthodes des classes implémentant l'interface IVehiculeServices.
- Déclaration de la classe qui implémente notre interface, ici la classe VehiculeServices.
- Déclaration d'un proxy sur notre classe VehiculeServices. Dans cette déclaration, nous devons dire pour quelle classe on veut créer un proxy et aussi quel type d'interception on veut réaliser autour des différentes méthodes. Dans notre cas, on va donc passer notre « TransactionInterceptor ».

```
<bean id="myTransactionInterceptor"
class="org.springframework.transaction.interceptor.TransactionInterceptor">
    <property name="transactionManager">
            <ref bean="myTransactionManager"/>
    </property>
    <property name="transactionAttributeSource">

    <value>IVehiculeServices.*=PROPAGATION_REQUIRED</value>
    </property>
</bean>
<bean id="myVehiculeServicesTarget" class="VehiculeServices">
    <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
    </property>
</bean>
<bean id="myVehiculeServices"
class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
            <value>IVehiculeServices</value>
    </property>
    <property name="interceptorNames">
            <list>
                    <value>myTransactionInterceptor</value>
                    <value>myvehiculeServicesTarget</value>
            </list>
    </property>
```

```
</bean>
```

Avec tout cela, si on fait appel à :

```
s = (IVehiculeService)factory.getBean("myVehiculeServices");
```

SPRING va retourner un proxy sur un objet de la classe VehiculeServices. Mais pour l'objet ayant fait l'appel, cet objet est toujours un IVehiculeServices, sauf que maintenant, les méthodes sont transactionnelles.

4.1. Le gestionnaire de transactions

Si vous avez bien lu le fichier de configuration précédent, vous aurez constaté la définition d'une propriété « transactionManager ».

Cette propriété permet à notre « TransactionInterceptor » de savoir quel objet gère effectivement les transactions. SPRING offre plusieurs gestionnaires de transaction adaptés en fonction des besoins et donc du contexte d'utilisation de SPRING :

- Vous utilisez simplement SPRING avec un driver JDBC et un gestionnaire de DataSource comme celui offert par jakarta-commons, utilisez DataSourceTransactionManager
- Vous utilisez SPRING avec le framework de persistance Hibernate, utilisez HibernateTransactionManager
- Vous utilisez SPRING dans le contexte d'un serveur d'application J2EE et/ou vous utilisez JTA, utilisez JtaTransactionManager

Exemple avec Hibernate et MySQL

```
<bean id="mySessionFactory"
class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="mappingResources">
            <list>
                    <value>sample.hbm.xml</value>
            </list>
    </property>
    <property name="hibernateProperties">
            <props>
                    <prop
key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
                    <prop key="hibernate.show_sql">true</prop>
            </props>
    </property>
    <property name="dataSource"><ref bean="myDataSource"/></property>
</bean>
<bean id="myTransactionManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">
    <property name="sessionFactory"><ref
bean="mySessionFactory"/></property>
```

```
</bean>
```

5. Les EJBs

Grâce à SPRING AOP, vous allez pouvoir faire appel à des EJBs sans le savoir et surtout sans polluer votre code de « jndi-lookup » ou autre « narrow ». L'idée est comme avec la définition des méthodes transactionnelles, de définir un proxy qui va prendre en charge pour vous l'appel aux EJBs locaux ou remotes.

Un bon principe proposé par SPRING, mais qui n'a rien d'obligatoire et qui est pris comme exemple dans la suite de cet article, et qu'il est recommandé de créer une interface « métier » Java classique pour vos EJBs et d'utiliser cette interface dans la déclaration de l'interface remote de l'EJB ainsi que dans l'implémentation de l'EJB (c'est ce qu'on appelle le pattern « EJB business method interface »).

5.1. Les EJBs locaux

Supposons donc que vous ayez un EJB session stateless qui implémente l'interface **IProductServices** et un contrôleur qui possède une propriété **myProductServices** qui s'attend à recevoir un IProductServices.
La configuration suivante permet de déclarer un proxy sur votre EJB sans que votre contrôleur n'en sache rien !! Je veux dire que votre contrôleur ne sait même pas qu'il fait appel à un EJB.

```
<bean id="myProductServices"

    class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName"><value>myProductServices</value></property>
    <property name="businessInterface"><value>IProductServices</value></property>
</bean>
<bean id="myController" class="myController">
    <property name="myProductServices"><ref bean="myProductServices"/></property>
</bean>
```

On voit ici la puissance de l'IoC ! Non ? Vous ne voyez rien !?

Mais si regardez, imaginez que vous soyez en cours de développement du contrôleur et que la personne qui doit développer l'EJB n'a pas encore finalisé son code. Pour faire vos tests unitaires vous avez besoin d'un « bouchon » sur l'EJB, une classe qui va simuler cet EJB (les anglais parlent de mocked objects). Il vous suffit alors de créer une classe Java classique type**MockProductServices** qui implémente l'interface IProductServices et de déclarer cette classe à la place de LocalStatelessSessionProxyFactoryBean (et sans les 2 propriétés). Pour votre contrôleur, rien ne change !

Lorsque vous en serez à la phase d'intégration de votre processus de développement, tous les « mocked objects » seront remplacés par les « vrais » objets et tout ceci sans aucune modification du code, la très grande classe non ?

5.2. Les EJBs remotes

Ben, pour les EJBs remote c'est pareil. Le seul truc auquel il faut faire « attention » ceux sont les exceptions définies dans l'interface remote de votre EJB. Vous avez ici plusieurs possibilités :

- Vous ne déclarez aucune exception dans l'interface métier de votre EJB et vous utilisez cette interface dans le fichier de configuration de SPRING.
- Vous déclarez des remote exceptions dans l'interface métier de votre EJB et vous utilisez cette interface dans le fichier de configuration de SPRING.
- Vous déclarez des remote exceptions dans l'interface métier de votre EJB et vous créez une seconde interface métier côté « client » sans les remotes exceptions et vous utilisez cette interface dans le fichier de configuration de SPRING. Dans ce cas, c'est le proxy SPRING qui se charge d'intercepter les exceptions remote et de les transformer en « RuntimeException ».

6. Conclusion

Bon et bien j'espère que cette brève introduction à SPRING vous aura mis l'eau à la bouche. L'objectif n'était pas d'être exhaustif sur chacun des thèmes abordés, aussi, vous découvrirez par vous-même qu'il existe d'autres manières d'aborder ces différents points et vous verrez que je n'ai pas mentionné tous les détails.

Vous pouvez légitimement vous demander si vous avez ou non besoin de SPRING, encore un autre framework !

De part mon expérience dans le monde des nouvelles technologies (13 ans maintenant), je vous dirais que **je ne vois pas** quels arguments vous pourriez avancer si ce n'est que vous avez déjà un framework « SPRING like » et donc qu'il est inutile d'investir dans un autre framework similaire. Sachez aussi que SPRING intègre / s'intègre aux frameworks suivants :

- Struts/Tiles, Velocity, FreeMarker, WebWork, Spring Web flow
- Hibernate, iBatis
- J2EE/EJB
- Hessian, Burlap, RMI, Spring's http invoker

Dans de prochains articles, j'aborderai plus en détail la gestion des exceptions proposée par SPRING, la couche d'accès aux données, l'intégration avec Hibernate, SPRING AOP et enfin la partie Web (avec peut être une présentation de SPRING Web Flow).

Sommaire