

# Spring, framework à "tout" faire

octobre 07

## Résumé

Spring est aujourd'hui l'un des framework Java les plus en vue. Basé sur des concepts simples, il permet néanmoins de modifier considérablement les processus de développement traditionnels des projets Java. Rod Johnson, son créateur, fut l'un des premiers à s'élever contre certaines normes incluses dans la plateforme Java EE, en fustigeant une complexité de développement trop grande à son goût. Partisan de la première heure d'un modèle de développement basé sur les bons vieux pojo (*Plain Old Java Object*). L'évolution de la plateforme de développement Java lui donnera raison, une refonte des modèles de programmation des EJB3 en est un exemple parfait.

Néanmoins la déferlante Spring constatée sur les projets actuels peut laisser perplexe quand à l'utilisation qui en est faite. S'il est important et justifié de lui reconnaître nombre de qualités, il semble important de cadrer son usage afin d'éviter toutes dérives liées à une sur-utilisation. Peut-être faudra-t-il lorgner du côté des EJB3 qui bien qu'arrivant après Spring, s'en inspire fortement et ont comme avantage intéressant et incontournable d'être une norme !

# Table des matières

<b>1. PRESENTATION DE SPRING</b>	<b>4</b>
1.1 Un rapide historique .....	4
1.2 Les modules Spring .....	4
<b>2. L'INJECTION DE DEPENDANCE</b>	<b>5</b>
2.1 Les dépendances .....	5
2.2 Comment initialiser les dépendances .....	6
2.3 Et Spring dans tout cela ? .....	7
<b>3. LA NOTION DE PROXY-INTERCEPTEUR</b>	<b>9</b>
3.1 Rappel : les proxies .....	9
3.2 Créer son proxy et l'utiliser avec Spring .....	9
3.3 Spring et les intercepteur .....	11
<b>4. UNE PUISSANTE BIBLIOTHEQUE DE CLASSES UTILITAIRES</b>	<b>12</b>
4.1 JdbcTemplate .....	12
4.2 HibernateTemplate ou HibernateDaoSupport .....	13
<b>5. CONCLUSION</b>	<b>14</b>

## Liste des Figures

<i>Figure 2. Modèle d'association en analyse.....</i>	<i>5</i>
<i>Figure 3. Modèle d'association en conception détaillée.....</i>	<i>6</i>
<i>Figure 4. Une Factory - le découplage.....</i>	<i>7</i>
<i>Figure 5. Fichier de paramétrage des dépendances Spring.....</i>	<i>8</i>
<i>Figure 6. Déclaration d'un intercepteur.....</i>	<i>10</i>

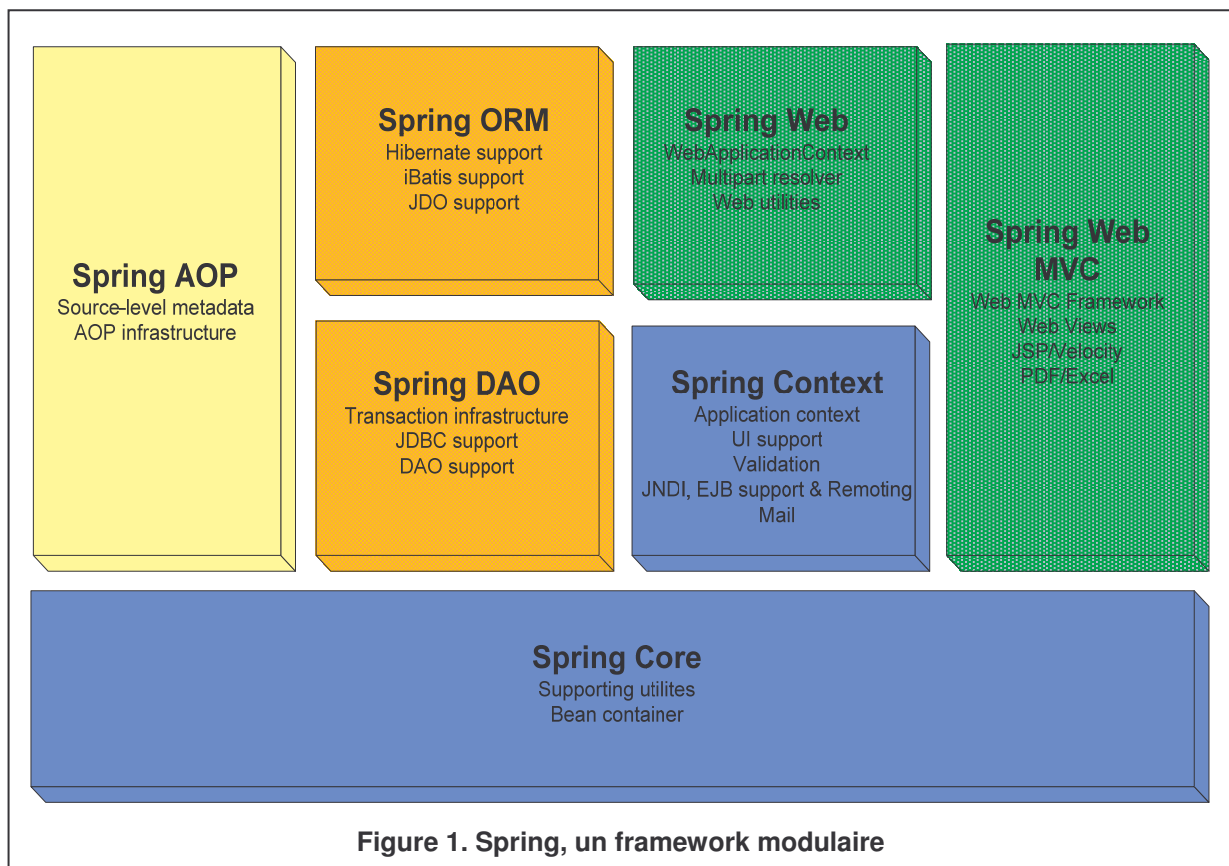
# 1. Présentation de Spring

*Don't call us, we'll call you*

## 1.1 Un rapide historique

Il y a quelques années, une poignée d'irréductibles supporters des concepts objets se sont élevés contre la lourdeur induite par les normes de la plateforme Java: le modèle de programmation et d'exécution des ejb furent alors choisis comme cible privilégiée de ces groupuscules alors disséminés un petit peu partout sur internet. Spring fut alors implicitement élu par la communauté de développeurs : Ce framework d'apparence simple, semblent répondre aux besoins de leurs projets, ainsi qu'aux processus de développement les plus en vue (développement piloté par les tests,...).

## 1.2 Les modules Spring



L'organisation de Spring est modulaire. Pourvu d'un module de base « *Spring core* », six modules d'inégales importances apparaissent (en **gras** les modules majeurs):

- **Spring Core** : module de gestion des dépendances entre beans (implémente L'injection de dépendance)
- **Spring AOP** : réservé à des développements très spécifiques

- **Spring ORM** : Classes utilitaires permettant une intégration intéressante des différents framework de mapping O/R, notamment avec Hibernate
- **Spring DAO** : Classes utilitaires facilitant à l'extrême le développement d'une couche d'accès aux données en *jdbc* pur.
- Spring Context : permet de masquer une grande partie de la technologie nécessaire pour se connecter à des *ejb*, à *JNDI*, à *JMS*,... Mais aussi l'internationalisation de nos applications
- Spring Web : comment utiliser Spring depuis une application web
- Spring MVC : Implémenter une application web en respectant le design pattern MVC (concurrent de struts)

## 2. L'injection de dépendance

La compréhension de l'injection de dépendance est incontournable pour faire du Spring. Dérivé du concept d'« *Inversion Of Control* », son utilisation est relativement simple, l'utilisation a bon escient est quand elle plus délicate.

### 2.1 Les dépendances

L'inversion de contrôle imaginé par Martin Fowler<sup>1</sup> il y a plusieurs années maintenant a été reprise par Spring sous une forme particulière : l'injection de dépendance. L'injection de dépendance est un *design pattern* finalement très simple, simplicité ouvrant néanmoins des portes particulièrement intéressantes.

La collaboration des objets les uns avec les autres est à la base des développements objets. Cette collaboration est rendue possible par la connaissance qu'un objet a d'un autre. Si un objet A a besoin de collaborer avec un objet B, il est nécessaire que l'objet A connaisse l'identité de B. On parle alors d'une dépendance entre les objets A et B. Cette dépendance, détectée lors de la phase d'analyse ou de conception d'un projet, donne naissance en phase de conception détaillée, à l'ajout d'un attribut de type B dans la classe A<sup>2</sup>.



Figure 2. Modèle d'association en analyse

---

<sup>1</sup> <http://martinfowler.com/articles/injection.html>

<sup>2</sup> Il peut arriver que cette dépendance s'implémente autrement : un paramètre d'une méthode par exemple

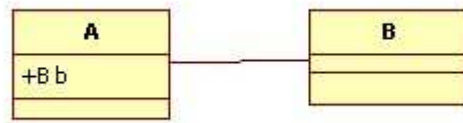


Figure 3. Modèle d'association en conception détaillée

## 2.2 Comment initialiser les dépendances

Traditionnellement, deux stratégies peuvent être envisagées afin d'initialiser ces dépendances :

- Utilisation de l'opérateur *new*
- Mise en œuvre d'une *factory* (codée par le développeur, mise à disposition par un framework quelconque)

### 2.2.1 L'opérateur *new*

L'usage de l'opérateur *new* est souvent réservé à l'initialisation des dépendances liant des classes appartenant à la même couche architecturale (associations entre objets métiers).

```

public class Pet {
    private Bone os; ;

    public Pet () {
        this.os = new Bone();
    }
}
  
```

### 2.2.2 La *factory*

La *factory* sera utilisée à des fins plus particulières : isoler l'instanciation de l'objet dépendant de l'initialisation de la dépendance elle-même (affectation de l'attribut avec la référence retournée par la *factory*). Les dépendances entre couches architecturales différentes sont particulièrement sujettes à une initialisation par l'intermédiaire d'une *factory* :

- La couche service passe par une *factory* pour travailler avec les *dao*<sup>3</sup>.

```

public class PetShopImpl {
    private PetShopDao dao; ;

    public PetShopImpl () {
        PetShopDao dao = PetShopFactory.getInstance();
    }
}
  
```

<sup>3</sup> Dao : Data Access Object

```
}

```

L'avantage de l'utilisation d'une *factory* réside dans le fait qu'elle exploite parfaitement le concept d'implémentation d'interface (ou bien celui de l'héritage). On considère que l'utilisation d'une *factory* apporte des avantages en termes de découplage : en effet, cela permet de manipuler un objet au travers d'un type que l'on a souhaité générique sans se soucier de la classe réelle de l'objet manipulé. On note sur le diagramme « Figure 4. Une *Factory* - le découplage » que le programme qui doit manipuler la couche d'accès aux données n'a plus aucun lien avec la classe qui sera physiquement instanciée, il manipulera un objet au travers d'une variable de type *ClientDao* (qui est une interface).

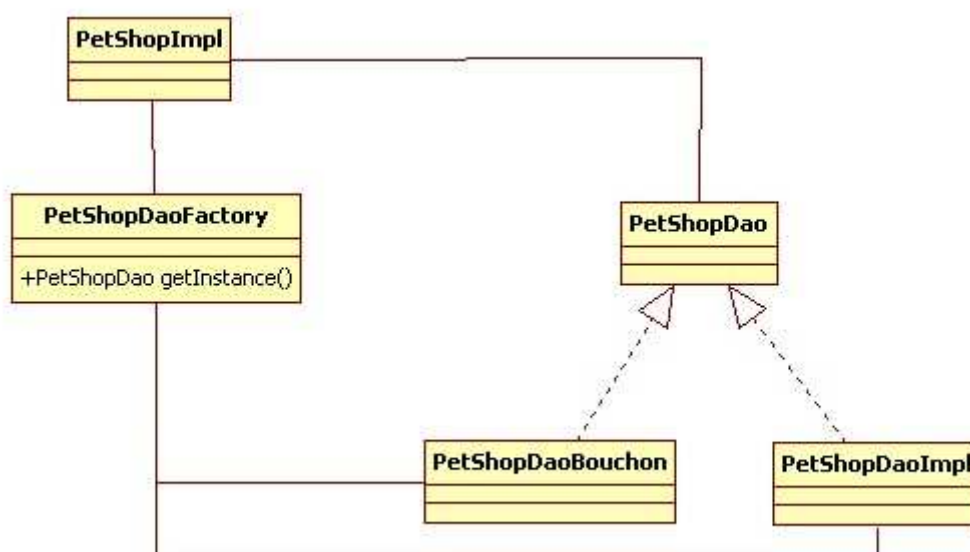


Figure 4. Une *Factory* - le découplage

Rappel : tout vient du fait que la factory utilisera un code ressemblant à :

```

public PetShopDao getInstance() {
    PetShopDao result = new PetShopDaoImpl(); // new PetShopDaoBouchon()
    return result;
}

```

### 2.3 Et Spring dans tout cela ?

Spring, et plus précisément le mécanisme d'injection de dépendance offre un nouveau moyen d'initialiser les dépendances entre vos classes. Le principe est simple : là où vous implémentiez une *factory* qui contenait la règle d'instanciation de votre dépendance (cf. code ci-dessus), Spring vous apporte une *factory* **générique** totalement **paramétrable** à l'aide d'un fichier de paramétrage au format xml.

```

<beans>
  <bean id="petdao"
        class="com.valtech.springtraining.dao.PetShopDaoImpl">
  </bean>

  <bean id="petshop"
        class="com.valtech.springtraining.service.PetShopImpl">
    <property name="dao">
      <ref bean="petdao" />
    </property>
  </bean>
</beans>

```

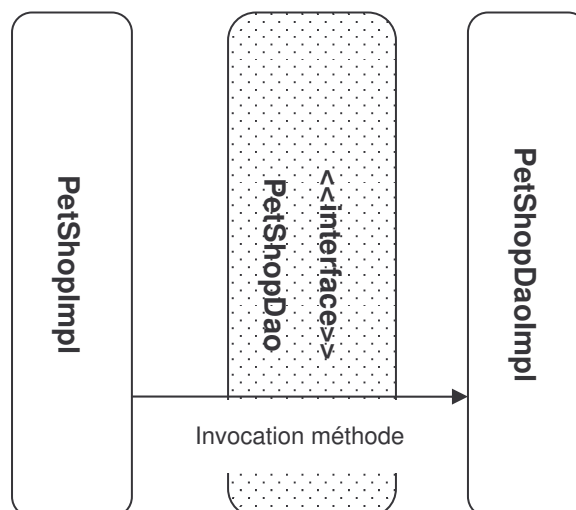
Figure 5. Fichier de paramétrage des dépendances Spring

Si l'on analyse ce fichier, on comprend assez aisément que l'on y déclare une dépendance entre un objet de type *PetShopImpl* et un *PetDaoImpl* ; La dépendance est implémentée sous la forme d'une propriété dans la classe *PetShopImpl* portant le nom *dao*. Les deux objets identifiés par « *petdao* » et « *petshop* » seront instanciés par Spring qui se chargera de faire l'initialisation de notre dépendance :

Code à la charge de Spring :

```
petshop.setDao(petdao) ;
```

Comme nous pouvons le constater sur cet exemple, le principe même d'injection de dépendances est novateur dans le sens où il normalise (au sens Spring) un paramétrage extérieur d'une *factory* générique. Mais la puissance de Spring ne se limite bien évidemment pas à ce concept mais à l'utilisation que l'on peut en faire, les chapitres suivants détaillent cette utilisation avancée et structurante.



### 3. La notion de proxy-intercepteur

Ce chapitre évoque l'une des techniques permettant d'intercepter des invocations aux objets que vous avez implémentez. D'autres techniques existent et répondent globalement à la même problématique (AOP) mais ne sont pas décrites en détail ici.

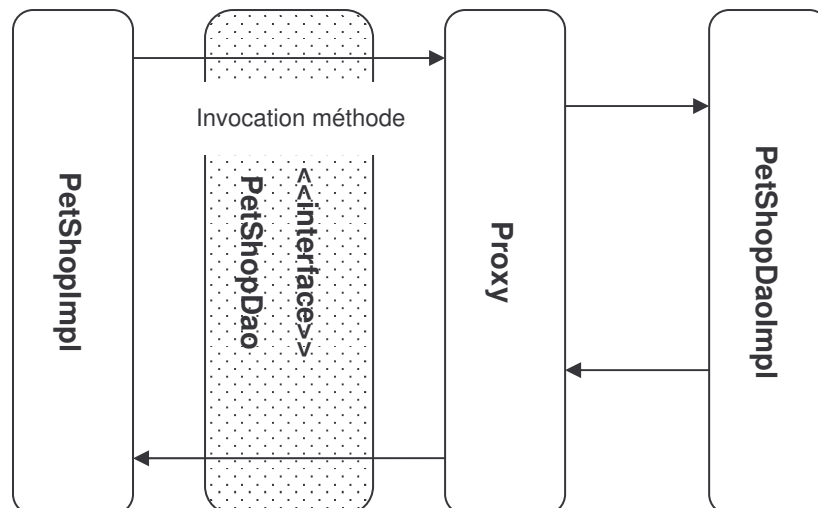
#### 3.1 Rappel : les proxies

Un *proxy* est un *design pattern* couramment utilisé dans le monde du développement.

Il désigne un mécanisme visant à masquer à un utilisateur nombre de traitement antérieur et/ou postérieur à celui qu'il pensait invoquer, et cela sans qu'il ait besoin d'apprendre une nouvelle api.

Le stub désigne l'artefact de programmation qui permet d'implémenter ce design pattern.

La mise en œuvre d'un *proxy* impose de respecter un certain nombre de bonnes habitudes au niveau de vos développement, à savoir principalement la généralisation de la programmation par interfaces (programmation par contrat).



#### 3.2 Créer son proxy et l'utiliser avec Spring

Créer un proxy est relativement simple avec Spring, il suffit d'écrire une classe qui implémente l'interface *org.aopalliance.intercept.MethodInterceptor*

```
public class DebugInterceptor implements MethodInterceptor
{
    private final Log logger = LogFactory.getLog(getClass());

    public Object invoke(MethodInvocation methodInvocation) throws Throwable
    {
        logger.info("Debut methode: " + methodInvocation.getMethod().getDeclaringClass() + "::" + methodInvocation.getMethod().getName());
        long startTime = System.currentTimeMillis();
        try
        {
            Object retVal = methodInvocation.proceed();
        }
    }
}
```

```

        return retVal;
    }
    finally
    {
        logger.info("fin methode: " + methodInvocation.getMethod().getDeclaringClass() + "::" + methodInvocation.getMethod().getName());
        logger.info("Temps d'exécution: " + (System.currentTimeMillis() - startTime) + " msecs.");
    }
}
}

```

Afin de pouvoir déclarer l'injection de cet intercepteur comme intermédiaire à votre traitement métier, il devient alors nécessaire de modifier le fichier de paramétrage de Spring :

```

<beans>
  <bean id="petdaoImpl"
    class="com.valtech.springtraining.dao.PetShopDaoImpl">
  </bean>

  <bean id="petshop"
    class="com.valtech.springtraining.service.PetShopImpl">
    <property name="dao">
      <ref bean="petdao" />
    </property>
  </bean>

  <bean id="debugInterceptor"
    class="com.valtech.interceptor.DebugInterceptor" />

  <bean id="petdao"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
      <value>com.valtech.springtraining.dao.PetShopDao</value>
    </property>
    <property name="interceptorNames">
      <list>
        <value>debugInterceptor</value>
        <value>petdaoImpl</value>
      </list>
    </property>
  </bean>
</beans>

```

Figure 6. Déclaration d'un intercepteur

La factory générique de Spring joue ici son rôle et masque à l'utilisateur l'initialisation de l'objet *petshop* grâce au fait que sa dépendance ait été initialisée sous forme d'un *proxy*. Ce dernier respectant l'interface *PetShopDao*, le développeur ne se rend pas compte qu'il utilise en fait un intermédiaire qui va utiliser le *debugInterceptor* avant de déléguer à l'objet cible qu'est *petDaoImpl*.

Aussi, dans la classe *PetShopImpl*, l'utilisation du dao telle que mentionnée ci-dessous :

```
public class PetShopImpl {
```

```

PetShopDao dao;
...
// Implementation des acceseur vers dao
public void ajouter(Pet p) {
    dao.sauve(p) ;
}
}

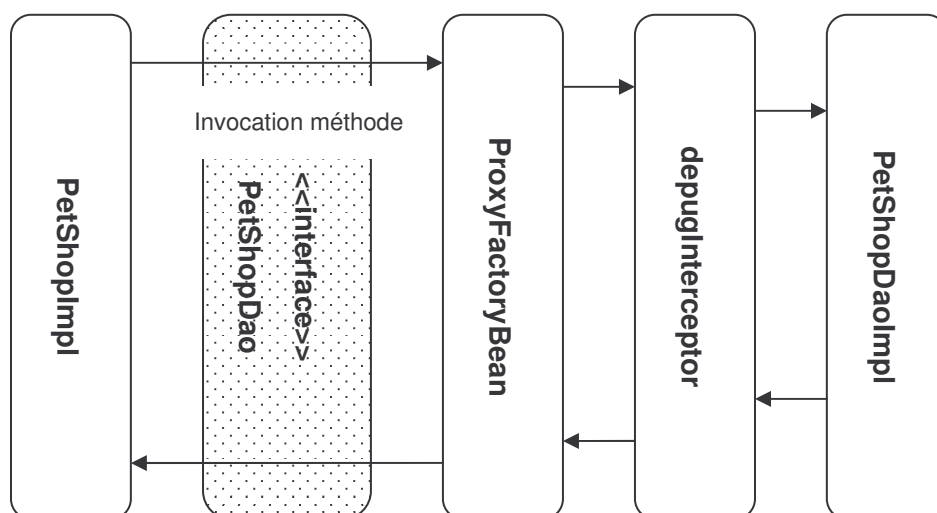
```

Aura le résultat suivant :

```

Debut methode: interface com.valtech.springtraining.dao.PetShopDao:
:sauve
// invocation de la méthode sauve de PetShopDaoImpl
Fin method: interface com.valtech.springtraining.dao.PetShopDao::sa
uve
Temps d'exécution: 10 msec.

```



### 3.3 Spring et les intercepteur

C'est la plus-value majeure de Spring ! Mais à quoi cela peut il bien servir ? Comme évoqué ci-dessus, le proxy désigne un mécanisme permettant d'exécuter des prétraitements et des post-traitements aux méthodes développées dans vos classes par vos soins. Un grand nombre de services techniques peuvent être implémentées par ces mécanismes, allégeant ainsi le code à la charge du développeur. Attention néanmoins à ne pas tomber dans le piège d'utiliser ces mécanismes afin d'implémenter la logique métier de vos applications, ils sera préférable de focaliser l'utilisation de ces mécanisme d'intercepteur à des fins techniques transverses :

- Démarcation des transactions
- Création et/ou vérification de contexte de sécurité
- Création et/ou utilisation de contexte Hibernate (session)

Bien évidemment, plusieurs de ces intercepteurs transverses existent déjà :

- *org.springframework.transaction.interceptor.TransactionProxyFactoryBean* pour la démarcation transactionnelle (capable de s'interfacer avec un manager de transaction piloté par hibernate ou un autre outil de mapping objet relationnel)



- Le framework ACEGI<sup>4</sup> pour la sécurité
- ...

La liste des intercepteurs n'est bien évidemment pas exhaustive : n'importe qui est en mesure d'implémenter son propre intercepteur et de le mettre à la disposition de la communauté. Je vous invite néanmoins à clairement identifier ceux qui vous seront utiles sur vos projets. Les plus structurants restent néanmoins le *TransactionProxyFactoryBean* pour démarquer vos transactions ainsi que *ACEGI* pour gérer la sécurité et les habilitations.

## 4. Une puissante bibliothèque de classes utilitaires

Spring comme on l'a vu, autorise une gestion de vos dépendances simplifiées et externalisées. L'utilisation conjointe de l'injection de dépendances et des intercepteurs permet de faire bénéficier vos objets de services techniques puissants et robustes.

Spring met en même temps à votre disposition un ensemble de classes utilitaires véritablement intéressantes, puissantes et simples d'utilisation. Nous allons nous attarder plus précisément aux classes *JdbcTemplate* et *hibernateDaoSupport*.

### 4.1 JdbcTemplate

A se demander pourquoi cette classe (ou ses dérivées : *NamedJdbcTemplate*) ne sont pas encore intégrées au JDK tellement leur utilisation relève du bon sens :

- Lisibilité
- Robustesse
- Concision

Exemple d'utilisation d'un *JdbcTemplate* qui extrait une liste contenant les noms de tous les utilisateurs.

```
JdbcTemplate template = new JdbcTemplate(dataSource);
List names = template.query("SELECT USER.NAME FROM USER",
    new RowMapper() {
        public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
            return rs.getString(1);
        }
    });
```

Un *NamedQuery* qui insère des données dans une table !

---

<sup>4</sup> <http://www.acegisecurity.org/>



```
String sql = "INSERT INTO TBL_USER( NAME , FIRSTNAME ) " +
            "VALUES (:name, :firstname) ";

Map<String, String> namedParameters = new HashMap<String, String>();
namedParameters.put("name", "Picasso");
namedParameters.put("firstname", "Pablo");

NamedParameterJdbcTemplate template = new NamedParameterJdbcTemplate(
    dataSource);
template.update(sql, namedParameters);
```

On remarque tous le travail effectué par les classes utilitaires de Spring :

- Gestion de la connexion via une *datasource* (avec intégration dans une éventuelle transaction courante)
- Gestion des exceptions
- Parfaite libération des ressources (*Connection, statement, resultset...*)

## 4.2 *HibernateTemplate* ou *HibernateDaoSupport*

La manipulation d'un *HibernateTemplate* permet à un développeur Hibernate aguerri d'être **très performant** et à un développeur Hibernate débutant **d'aller droit dans le mur**©. Je déconseille vivement à un développeur débutant dans le monde Hibernate qui ne serait pas encadré d'utiliser les fonctionnalités proposées par Spring pour travailler avec Hibernate (ou tout autre outil de mapping objet relationnel). Néanmoins, une utilisation maîtrisée de Spring pour faire de l'Hibernate colle parfaitement à une approche structurante permettant une utilisation industrielle d'Hibernate.

L'effet pervers de l'utilisation de cette classe est de vous masquer un certain nombre d'opération structurante pour Hibernate :

- Durée de vie de la session Hibernate
- Gestion des transactions
- Flush des opérations de persistance

Ces stratégies demeurent bien évidemment paramétrables au travers de Spring, mais une mauvaise utilisation peut avoir un impact désastreux sur les performances de l'application

Comme pour les *JdbcTemplate*, Spring se charge de gérer les exceptions, de libérer les ressources, et éventuellement de fermer la session hibernate (c'est bien là le problème !!) et offre ainsi un modèle de programmation épuré :

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        return getHibernateTemplate().find(
            "from test.Product product where product.category=?", category);
    }
}
```



## 5. Conclusion

Nous avons vu dans ce document la simplicité d'utilisation que procure Spring dans le cadre des développements. Parfaitement adapté à des processus de développement moderne notamment en ce qui concerne la testabilité de nos applications en dehors d'une infrastructure technique lourde (par opposition avec les EJB dans un serveur d'application).

Il est néanmoins important de cadrer l'utilisation de Spring afin d'éviter les dérives correspondant à des abus d'injection de dépendances. Autant il me semble pertinent de lier l'ensemble des couches architecturales : présentation, service et accès aux données, autant il me semble superflu d'utiliser l'injection de dépendances dans une de ces trois couches.

L'utilisation de Spring sur l'ensemble des couches architecturales d'une application permet en effet de le qualifier de framework à tout faire. Attention néanmoins aux abus d'usage, et surtout de considérer Spring comme la solution qui masque la complexité des framework auxquels il se couple : une maîtrise préalable du framework sous jacent est un impératif.