# FAQ Spring

Date de publication : 21/10/2007

Dernière mise à jour : 17/05/2009

Cette faq a été réalisée à partir des questions fréquemment posées sur les forums de **www.developpez.com** et de l'expérience personnelle des auteurs.
Nous tenons à souligner que cette faq ne garantit en aucun cas que les informations qu'elle propose sont correctes. Les auteurs font leur maximum, mais l'erreur est humaine. Cette faq ne prétend pas non plus être complète. Si vous trouvez une erreur, ou que vous souhaitez nous aider en devenant rédacteur, lisez ceci.
Sur ce, nous vous souhaitons une bonne lecture.
**L'équipe Java**

# Ont contribué à cette FAQ :

Righetto Dominique - djo.mos - Michael Courcy - Rédaction Java (Equipe Java) - lunatix - Erik Gollot (http://ego.developpez.com) - Gildas Cuisinier (Hikage) ( Blog ) -

## Sommaire > Informations

### Quels sont les droits de reproduction de cette FAQ ?

**Auteurs : Rédaction Java ,**



**lien : Quels sont les droits de reproduction de cette FAQ ?**

### Comment bien utiliser cette FAQ ?

**Auteurs : Rédaction Java ,**

**Le but :**

**Cette faq a été conçue pour être la plus simple possible d'utilisation. Elle tente d'apporter des réponses simples et complètes aux questions auxquelles sont confrontés tous les débutants (et les autres).**

**L'organisation :**

**Les questions sont organisées par thème, les thèmes pouvant eux-même contenir des sous-thèmes. Lorsqu'une question porte sur plusieurs thèmes, celle-ci est insérée dans chacun des thèmes rendant la recherche plus facile.**

**Les réponses :**

**Les réponses contiennent des explications et des codes sources. Certaines sont complétées de fichier à télécharger contenant un programme de démonstration. Ces programmes sont volontairement très simples afin qu'il soit aisé de localiser le code intéressant. Les réponses peuvent également être complétées de liens vers d'autres réponses, vers la documentation en ligne de Sun ou vers un autre site en rapport.**

**Nouveautés et mises à jour :**

**Lors de l'ajout ou de la modification d'une question/réponse, un indicateur est placé à coté du titre de la question. Cet indicateur reste visible pour une durée de 15 jours afin de vous permettre de voir rapidement les modifications apportées.**

**J'espère que cette faq pourra répondre à vos questions. N'hésitez pas à nous faire part de tous commentaires/remarques/critiques.**

**lien : Comment participer à cette FAQ ?**

### Où trouver d'autres sources d'information ?

**Auteurs : Rédaction Java ,**

- **Les cours et didacticiels de www.developpez.com**
- **Les forums de la section Java de www.developpez.com**

- Les FAQs Java de www.developpez.com

## Comment participer à cette FAQ ?

**Auteurs : Rédaction Java ,**

**Cette faq est ouverte à toute collaboration. Pour éviter la multiplication des versions, il serait préférable que toutes collaborations soient transmises aux administrateurs de la faq.**
**Plusieurs compétences sont actuellement recherchées pour améliorer cette faq :**

**Rédacteur :**

**Bien évidemment, toute nouvelle question/réponse est la bienvenue.**

**Web designer :**

**Toute personne capable de faire une meilleur mise en page, une feuille de style ou de belles images...**

**Correcteur :**

**Malgré nos efforts des fautes d'orthographe ou de grammaire peuvent subsister. Merci de contacter les administrateurs si vous en débusquez une... Idem pour les liens erronés.**

**lien : Quels sont les droits de reproduction de cette FAQ ?**

## Remerciements

**Auteurs : Rédaction Java ,**

**Un grand merci à tous ceux qui ont pris de leur temps pour la réalisation de cette FAQ.**

**Aux rédacteurs :**

**Remerciements tout d'abord à tous ceux qui ont rédigé les questions et les réponses.**

**Aux correcteurs :**

**Remerciements également aux personnes qui ont relu les textes pour supprimer un maximum de fautes de français.**

**Aux visiteurs :**

**Remerciements enfin à tous ceux qui ont consulté cette FAQ, et qui, par leurs remarques, nous ont aidé à la perfectionner.**

**Et pour finir, un merci tout spécial à tous les membres de l'équipe qui nous ont fourni outils et logiciels nécessaires pour la réalisation de ce document.**

## Sommaire > Spring Framework

### Qu'est-ce que Spring

**Auteurs : Erik Gollot ,**

**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'appuie sur les principes du design pattern IoC et sur la programmation par aspects (AOP).**

**Spring est disponible sous licence Apache 2.0**

**lien : Site officiel de Spring Framework**

### Ressources Spring ?

**Auteurs : Gildas Cuisinier ,**

**Developpez.com propose à l'heure actuelle plusieurs cours et articles concernant Spring :**

- **Introduction au framework Spring, par Erik Gollot**
- **Spring 2.0 et namespace, par Erik Gollot**
- **Tutoriel Spring IOC, par Serge Tahé**
- **Spring : théorie & pratique, par Steve Hostettler**

**D'autre part, la documentation officielle de Spring est très claire et très précise. Une lecture attentive de celle-ci permet de bien commencer et répond à de nombreuses questions.**

**De nombreuses publications sur Spring existent, la plupart en anglais. Voici une liste non exaustive de ces livres :**

- **Spring par la pratique, Eyrolles**
- **Professional Development with the Spring Framework, Wrox Collections**
- **Pro Spring, Apress**
- **Spring in Action, Manning Publications**

-  **Expert Spring MVC And Web Flow, Apress**

**Et pour répondre à vos questions, Developpez.com possède son forum Spring.**

## Comment utiliser Spring avec Maven ?

**Auteurs : Gildas Cuisinier ,**

**Il est possible d'utiliser Spring dans un projet basé sur Maven de deux manières**

**La première est d'inclure l'intégralité de Spring dans le projet :**

```
<dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring</artifactId>
            <version>2.0.5</version>
            <scope>runtime</scope>
</dependency>
```

**De cette manière un jar unique contenant tous les modules ( Core, AOP, DAO, MVC, EJB, ... ) sera récupéré.**

**L'autre méthode consiste à spécifier uniquement les modules utiles au projet :**

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>2.0.5</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>2.0.5</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>2.0.5</version>
    <scope>runtime</scope>
</dependency>
```

**Cela permet de réduire la taille du projet lorsque tous les modules ne sont pas utilisés**

**lien : Dépôt Maven pour Spring Framework**

## Comment résoudre le problème de dépendance avec JTA ?

**Auteurs : Gildas Cuisinier ,**

**Lors de l'utilisation de Spring avec Maven, il se peut qu'un problème de dépendance avec JTA apparaisse :**

```
required artifacts missing:
 javax.transaction:jta:jar:1.0.1B
```

**La cause de ce problème est l'incompatibilité de JTA avec le dépôt Maven, et donc il ne peut pas être disponible sur celui-ci. Il est donc nécessaire de télécharger manuellement JTA sur le site de Sun : JTA**

**Une fois le fichier, par exemple jta-1_0_1B-classes.zip, téléchargé, il est nécessaire de l'ajouter dans le dépôt local :**

```
mvn install:install-file -Dfile=<chemin vers le fichier téléchargé> -DgroupId=javax.transaction -
DartifactId=jta -Dversion=<version du jar téléchargé> -Dpackaging=jar
```

**<version du jar téléchargé> correspondra à la version du jar téléchargé, dans le cas de jta-1_0_1B-classes.zip, la version est 1.0.1B**

**lien : Jar de JTA sur le site de Sun Microsystem**

## Peut-on utiliser Spring dans une application Web ?

**Auteurs : Gildas Cuisinier ,**

**Oui, il est tout à fait possible d'utiliser Spring dans une application Web. En effet Spring possède des intégrations avec différents framework tels que Struts, JSF ou même les servlets J2EE.**
**D'autre part, Spring possède son propre module MVC Web, aux fonctionnalités similaires à Struts mais plus évolué et possèdant toutes possibilités de Spring.**

**De plus, Spring fournit des moyens d'exporter les services métiers sous forme de Web Services, ou d'autres moyens équivalents ( HttpInvocker, Burlaps, RMI, EJB ), et d'utiliser ceux-ci facilement dans des clients.**

## Peut-on utiliser Spring dans une application desktop ?

**Auteurs : Gildas Cuisinier ,**

**Oui, il est possible d'utiliser Spring dans une application standalone pour gérer les couches métiers, l'accès au données ou l'utilisation de Web Services.**

**De plus, un sous projet de Spring a pour but de fournir une plateforme afin d'aider les développeurs dans la création d'application Swing de haute qualité : Spring RCP**

**lien : Spring Rich Client**

## Qu'est-ce que l'AOP ?

**Auteurs : Gildas Cuisinier ,**

**Dans les développements, il n'est pas rare d'avoir certaines fonctionnalités qui sont utilisées à plusieurs endroits dans un code. C'est le cas par exemple de l'écriture des traces, de la sécurité, de la gestion des transactions. Prenons le cas de la sécuritée :**

```
private void maMethodeSecurisée()
{
   if(GestionSecurite.peutExecuter("maMethodeSecurise", nomUtilisateur){
  // Ici le code de la méthode propre
   } else throw new
 ExecutionInterditeException("Vous n'avez pas les droits d'éxecution de cette méthode");

}
```

**Donc ici, chaque méthode sécurisée est responsable à la fois de la logique qu'elle doit implémentée, mais aussi de la gestion de la sécurité.**
**Cela provoque deux problèmes : le premier, il est préférable qu'une classe ne possède qu'une seule responsabilité. D'un autre coté, comme chaque classe à une "copie" du code de sécurité, le moindre changement doit être répercuté à plusieurs endroit.**

**C'est pour résoudre ces problèmes de fonctionnalités transversales que la notion de programmation orienté aspect existe. Cette fonctionnalité transversale est implémentée dans ce qu'on appelle donc un aspect qui est composé de :**

- **Un greffon ( ou advice ), qui est le code qui doit être exécuté**
- **Des points d'action ( ou pointcut ), qui définissent quand ce code doit être exécuté**

**Les points d'actions peuvent être défini sur la base d'expression régulières ( sur base des types de retour et paramètres, nom du package, nom de la méthode ) et les greffons peuvent être exécutés avant, après ou "autour" du point d'action.**

**Ensuite, le framework d'AOP va effectué un tissage, c'est à dire 'introduire' le code du greffon à tout les endroits qui correspondent au points d'action. Ce tissage pouvant être réalisé de plusieurs manières : à la compilation ( ou le code source sera remanié pour y introduire les apppels ), au lancement ou encore au chargement des classes.**

**Parmis les frameworks Java pour l'AOP, les plus connus sont :**

- **AspectJ**
- **AspectWerkz**
- **JAC**
- **JBossAOP**
- **Spring AOP**

**lien : Spring : Théorie et pratique**
**lien : Introduction à Spring AOP**

# Comment configurer et utiliser un conteneur Spring ?

**Auteurs : Gildas Cuisinier ,**

**Afin de configurer un conteneur Spring, il est nécessaire de créer un fichier xml contenant la définition des divers Beans qui seront gérés par Spring.**

**Voici un exemple simple de ce fichier, version Spring 2.0 utilisant des namespaces :**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans-2.0.xsd">

 <bean name="unBean" class="com.developpez.spring.unBean">
        <property name="unePropriete" value="Hello world"/>
 </bean>
</beans>
```

**Le même fichier, mais pour Spring 1.2 qui utilise une DTD à la place des namespaces :**

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
beans.dtd">
<beans>
 <bean name="unBean" class="com.developpez.spring.unBean">
        <property name="unePropriete" value="Hello world"/>
 </bean>
</beans>
```

**Une fois le fichier créé, il est nécessaire d'instancier une fabrique de Beans ( Bean Factory ) ou un contexte d'application ( ApplicationContext ). Il existe diverses implémentations de ces interfaces, mais les deux principales chargent un ( ou plusieurs ) fichier situé soit par un chemin sur le disque, soit dans le classpath :**

```
  // Instanciation d'un contexte Spring à partir d'un fichier XML dans le classpath
        ApplicationContext ap = new ClassPathXmlApplicationContext("com/developpez/spring/
applicationContext.xml");

        // Instanciation d'un contexte Spring à partir d'un fichier sur le disque
        ApplicationContext ap2 = new FileSystemXmlApplicationContext("c:\\applicationContext.xml");

        // Instanciation d'un BeanFactory à partir d'un fichier xml dans le classpath
        Resource resource = new ClassPathResource("com/developpez/spring/beans.xml");
        BeanFactory factory = new XmlBeanFactory(resource);

    // Instanciation d'un BeanFactory à partir d'un fichier xml sur le disque
        Resource resource2 = new FileSystemResource("c:\\beans.xml");
        BeanFactory factory2 = new XmlBeanFactory(resource);
```

**Une fois une instance créée, la récupération d'un Bean Spring se fait via la méthode getBean :**

```
MonBean bean = (MonBean )ap.getBean("monBean");
// ou
```

```
MonBean bean2 = (MonBean) factory.getBean("monBean");
```

**lien : Documentation officiel de Spring - Fabrique de Beans**

## Comment définir un bean ?

**Auteurs : Gildas Cuisinier ,**

**La définition d'un bean dans le fichier de configuration se fait comme ceci :**

```
<beans>
<bean id="monBean" class="com.developpez.spring.MonBean"/>
</beans>
```

## Comment définir plusieurs noms pour un Bean

**Auteurs : Gildas Cuisinier ,**

**Il est possible de donner plusieurs noms à un Bean via l'attribut *name*, par exemple le bean :**

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans-2.0.xsd">
    <bean id="monBean" class="com.developpez.spring.MonBean" name="monBeanAlias,aliasMonBean" />
</beans>
```

**possède le nom *monBean* ( déf séparés par une virgule**



## Comment définir un Bean en tant que singleton ?

**Auteurs : Gildas Cuisinier ,**

**Si rien n'est spécifié, Spring considère tout Bean comme un singleton. Autrement dit, lorsque deux demandes sont faites pour un même Bean, c'est la même instance de l'objet qui est fournie par le BeanFactory/ApplicationContext.**

**Si vous désirez qu'un nouvel objet soit instancié à chaque fois il faut le spécifier dans la définition du Bean :**

```
<beans>
<bean id="monBean" class="com.developpez.spring.MonBean" singleton="false"/>
</beans>
```

**Depuis la version 2.0 de Spring, c'est via l'attribut Scope que cela est géré :**

```
<!-- Bean Singleton -->
<bean id="monBean" class="com.developpez.spring.MonBean" scope="singleton"/>
</beans>

<!-- Bean non Singleton -->
<bean id="monBean" class="com.developpez.spring.MonBean" scope="prototype"/>
```

```
</beans>
```

**lien : Documentation officielle, Scope**

## Comment injecter des propriétés de type Properties

**Auteurs : Gildas Cuisinier ,**

**Spring fournit des balises afin de pouvoir gérer facilement les propriétés de type Properties :**

```
<bean id="monBean" class="com.developpez.spring.MonBean" name="monBeanAlias,aliasMonBean">
        <property name="proprietes">
            <props>
                <prop key="application.nom">Mon Application</prop>
                <prop key="application.version">1.0.0</prop>
                <prop key="application.auteur">Gildas Cuisinier</prop>
            </props>
        </property>
    </bean>
```

**équivaut à un fichier properties :**

```
application.nom=Mon Application
application.version=1.0.0
application.auteur=Gildas Cuisinier
```

## Comment injecter un Bean dans un autre ?

**Auteurs : Gildas Cuisinier ,**

**Afin d'illustrer l'injection de Bean dans un autre grâce à Spring partons d'un exemple :**

**Soit un service, qui possède entre autre une méthode pour lister les utilisateurs :**

```
public class UserService {

    private UserDao userDao;
    public void setUserDao(UserDao userDao) {
        this.userDao = userDao;
    }

    public List listUsers(){
        return userDao.listUsers();
    }
}
```

**Le service utilise un DAO pour les utilisateurs. Les méthodes de ce DAO sont définies dans une interface :**

```
public interface UserDao {
    public List listUsers();
}
```

**Et une implémentation de cette interface :**

```
public class UserDaoImpl implements UserDao {
    public List listUsers() {

        List result = new ArrayList();

        result.add(new User("hikage", "password"));
        result.add(new User("spring", "spring"));

        return result;
    }
}
```

**Afin de spécifier à Spring d'injecter une référence du DAO au service, il faut déclarer tout d'abord un bean pointant sur l'implémentation du DAO :**

```
  <bean id="userDao" class="com.developpez.spring.UserDaoImpl"/>
```

**Ensuite, dans la déclaration du service en lui-même, il est nécessaire de créer une propriété :**

```
<bean id="userService" class="com.developpez.spring.UserService">
       <property name="userDao" ref="userDao"/>
   </bean>
```

**Dès lors, lorsque l'on récupérera une instance du service, Spring injectera automatiquement une instance de l'implémentation.**

**lien : Documentation officielle, References to other beans**

## Comment injecter des dépendances via le constructeur ?

**Auteurs : Gildas Cuisinier ,**

**Pour injecter des propriétés via le constructeur d'une classe, il faut utiliser les balises *constructor-arg*.**

**Soit la classe :**

```
public class BeanConstructor {

    private String chaine;
    private Integer entier;

    public BeanConstructor(String chaine, Integer entier) {
        this.chaine = chaine;
        this.entier = entier;
    }
}
```

**Il est possible de spécifier les arguments via leur index :**

```
<bean id="beanConstructeur2" class="com.developpez.hikage.BeanConstructor">
        <constructor-arg index="0" value="Test"/>
```

```
        <constructor-arg index="1" value="23"/>
    </bean>
```

**Cependant, imaginons maintenant que la classe possède un constructeur supplémentaire :**

```
public BeanConstructor(String chaine, String chaine2)
```

**Dans ce cas, il est préférable de spécifier à Spring de quel type sont les paramètres, afin de s'assurer de passer par le bon constructeur :**

```
<bean id="beanConstructeur" class="com.developpez.hikage.BeanConstructor">
        <constructor-arg type="java.lang.String" value="Test"/>
        <constructor-arg type="java.lang.Integer" value="23"/>
    </bean>
```

## Comment externaliser des propriétés dans un fichier Properties ?

**Auteurs : Gildas Cuisinier ,**

**Soit un fichier properties qui contient les informations suivantes :**

```
datasource.driver=org.mysql.driver.Driver
datasource.url=jdbc://mysql:localhost/maBase
datasource.username=dbUser
datasource.password=dbPassword
```

**Il est possible de récupérer ces informations sous forme de variables dans le fichier de configuration de Spring en ajoutant un Bean :**

```
<bean name="propertyPlaceholder" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
            <property name="locations">
                <value>classpath:com/developpez/hikage/properties/datasource.properties</value>
            </property>
        </bean>
```

**Ensuite, pour accèder aux variables il suffit de les encadrer dans ${ } :**

```
<bean id="datasource" class="com.developpez.hikage.Datasource">
        <property name="urlConnection" value="${datasource.url}"/>
        <property name="drivers" value="${datasource.drivers}"/>
        <property name="username" value="${datasource.username}"/>
        <property name="password" value="${datasource.password}"/>
    </bean>
```

## Comment injecter une constante statique dans une propriété?

**Auteurs : Gildas Cuisinier ,**

**Les informations nécessaires à la configuration sont parfois stockées dans des champs statiques d'une classe ou d'une interface. Spring fournit une classe utilitaire pour récupérer ces informations pour ainsi les injecter dans un bean.**

**Par exemple, le nom d'une application est stockée dans une classe *Constants* :**

```
public class Constants {
    public static final String APPLICATION_NAME = "FAQ Spring";
}
```

**La classe *FieldRetrievingFactoryBean* fournit un moyen d'injecter cette information dans la propriété *applicationName* de la classe *ApplicationInformation* :**

```
 <bean id="applicationInformation" class="com.developpez.spring.ApplicationInformation">
       <property name="applicationName">
           <bean class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">

  <property name="staticField" value="com.developpez.spring.Constants.APPLICATION_NAME"/>
           </bean>
       </property>
    </bean>
```

**Depuis la version 2.0 de Spring, via le namespace util, il existe un raccouçi pour faire la même opération :**

```
<bean id="applicationInformationV2" class="com.developpez.spring.ApplicationInformation">
       <property name="applicationName">
         <util:constant static-field="com.developpez.spring.Constants.APPLICATION_NAME"/>
       </property>
    </bean>
```

**lien : Documentation officielle de Spring, Schema Utils**

## Comment injecter des propriétés de types simples ?

**Auteurs : Gildas Cuisinier ,**

**Soit la classe :**

```
public class MonBean {

    private String chaine;
    private Integer entier;
    private Long  entierLong;
    private Boolean booleen;
    private Float flottant;

public void setChaine(String chaine) { this.chaine = chaine ; }
public String getChaine() { return this.chaine ; }

public void setEntier(Integer entier) { this.entier = entier ; }
public Integer getEntier() { return this.entier ; }

public void setEntierLong(Long entierLong) { this.entierLong = entierLong ; }
public String getEntierLong() { return this.entierLong ; }

public void setBooleen( Boolean booleen ) { this.booleen = booleen ; }
public Boolean getBooleen() { return this.booleen; }

public void setFlottant(Float flottant) { this.flottant = flottant ; }
public Float getFlottant() { return this.flottant } ;

}
```

**Voici un exemple de fichier de configuration qui spécifie les valeurs des types simples :**

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans-2.0.xsd">

<bean id="monBean" class="com.developpez.spring.MonBean">
        <property name="chaine" value="Visitez Developpez.com"/>
        <property name="booleen"  value="true"/>
        <property name="entier" value="10"/>
        <property name="flottant" value="10.3"/>
        <property name="entierLong" value="204"/>
    </bean>

</beans>
```

## Comment configurer une liste ?

**Auteurs : Gildas Cuisinier ,**

**Soit la classe :**

```
public class BeanList {

    private List list;

    public List getList() {
        return list;
    }

    public void setList(List list) {
        this.list = list;
    }
}
```

**Il est possible de définir cette liste via les balises :**

```
<bean id="beanList" class="com.developpez.hikage.BeanList">
        <property name="list">
            <list>
                <value>Valeur 1</value>
                <value>Valeur 2</value>
                <value>Valeur 3</value>
                <ref bean="monBean"/>
            </list>
        </property>
    </bean>
```

## Comment configurer une Map ?

**Auteurs : Gildas Cuisinier ,**

**Soit une classe :**

```
public class BeanMap {
```

```
    private Map map;

    public Map getMap() {  return map;}

    public void setMap(Map map) {   this.map = map;   }
}
```

**Pour configurer celle-ci via le fichier XML, il faut utiliser les balises :**

```
<bean id="beanMap" class="com.developpez.hikage.BeanMap">
        <property name="map">
   <map>
    <entry>
                <key><value>Clé de type String</value></key>
                <value>Valeur associé de type String</value>
            </entry>
            <entry>
             <!-- Clé de type Object -->
                <key><ref bean="monBeanCle"></ref></key>
      <!-- Valeur dde type Object -->
                <ref bean="monBeanValeur"></ref>
            </entry>
   </map>
        </property>
    </bean>
```

**Il est aussi possible d'écrire ces lignes de manière abrégée :**

```
<bean id="beanMap" class="com.developpez.hikage.BeanMap">
        <property name="map">
            <map>
                <entry key="Ma clé de type String" value="Valeur associée de type String" />
                <entry key-ref="monBean" value-ref="monBean"/>
            </map>
        </property>
    </bean>
```

## Comment fournir le context Spring à un Bean ?

**Auteurs : Gildas Cuisinier ,**

**Il est parfois nécessaire à un Bean de possèder une référence à l'ApplicationContext ( ou le BeanFactory ) qui s'occupe de son cycle de vie. Pour cela, il suffit simplement que celui-ci implémente une de ces deux interfaces pour que Spring injecte automatiquement la référence correspondante :**

```
public interface ApplicationContextAware  {
    void setApplicationContext(org.springframework.context.ApplicationContext
 applicationContext) throws org.springframework.beans.BeansException;
}

public interface BeanFactoryAware  {
    void setBeanFactory(org.springframework.beans.factory.BeanFactory beanFactory) throws
 org.springframework.beans.BeansException;
}
```

**lien : Javadoc ApplicationContextAware**

**lien : Javadoc BeanFactoryAware**

## Comment effectuer des vérifications après la configuration des propriétés?

**Auteurs : Gildas Cuisinier ,**

**Dans le cas de l'injection de dépendance par accesseurs, il n'est pas possible de manière standard de vérifier que tous les arguments obligatoires sont fournis. De plus, il est parfois nécessaire d'effectuer certaines opérations de configuration manuellement. Pour cela, Spring fournit une interface :**

```
public interface InitializingBean  {

    void afterPropertiesSet() throws java.lang.Exception;
}
```

**Dès qu'un bean implémente cette interface, Spring appellera la méthode afterPropertiesSet après l'appel de tous les accesseurs, fournissant ainsi un moyen de vérifier l'existance de tous les paramètres ainsi que leur validité, ou encore de configurer certaines ressources manuellement.**

## Comment accéder au contexte Spring depuis un composant non géré par Spring ?

**Auteurs : Gildas Cuisinier , Righetto Dominique ,**

**Pour donner accès au contexte Spring aux composants non gérés par Spring, il suffit d'implémenter un bean géré par Spring, implémentant ApplicationContextAware, qui va exposer des méthodes d'accès au contexte Spring.**

```
public class ApplicationContextHolder implements ApplicationContextAware {

 /** Contexte Spring qui sera injecte par Spring directement */
 private static ApplicationContext context = null;


 /**
 * Méthode de ApplicationContextAware, qui sera appellée automatiquement par le conteneur
 */
 public void setApplicationContext(ApplicationContext ctx)
   throws BeansException {
  context = ctx;
 }


 /**
  * Methode statique pour récupérer le contexte
  */
 public static ApplicationContext getContext() {
  return context;
 }

}
```

**Il est ensuite nécessaire de configurer cette classe dans le contexte Spring :**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="
```

```
      http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/
spring-beans-2.5.xsd">

 <!-- ============================   BEANS   ============================ -->
 <!-- Bean d'acces au Contexte -->
 <bean id="applicationContextHolder" class="com.developpez.spring.ApplicationContextHolder" />

 <!-- Definition des autres beans -->
</beans>
```

**Le principe est que lors de l'instanciation de la classe ApplicationContextHolder par le conteneur Spring, celui-ci va détecter qu'elle implémente l'interface ApplicationContextAware, et donc lui injecter le contexte.**
**Le setteur de la classe va stocker ce contexte dans une variable de classe ( variable déclarée statique ). Et la classe propose une méthode, statique, de récupération de ce contexte.**

**Exemple d'utilisation :**

```
ApplicationContext context = ApplicationContextHolder.getContext();
MonBean bean = (MonBean) context.getBean("monBean");
```

**lien : FAQ Comment fournir le context Spring à un Bean ?**

## Que propose le namespace p

**Auteurs : Gildas Cuisinier ,**

**Le namespace p a pour but de réduire la taille du fichier de configuration de Spring, en proposant des raccourcis pour la définition des propriétés.**

**Par exemple, un fichier de configuration sans le namespace p :**

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean name="monBean" class="com.developpez.spring.Personne">
        <property name="nom" value="Cuisinier"/>
        <property name="prenom" value="Gildas"/>
        <property name="age" value="23"/>
        <property name="pays" value="Belgique"/>
        <property name="parrain" ref="beanParrain"/>
    </bean>

    <bean name="beanParrain" class="com.developpez.spring.Personne"/>
</beans>
```

**Grâce au namespace p, voici le même fichier de configuration adapté :**

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean name="monBean" class="com.developpez.spring.Personne"
    p:nom="Cuisinier" p:prenom="Gildas" p:age="23 p:pays="Belgique" p:parrain-ref="beanParrain"/>
```

```
    </bean>

    <bean name="beanParrain" class="com.developpez.spring.Personne"/>
</beans>
```

**La synthaxe étant p:<propriete> pour les valeurs directes et p:<propriete-ref> pour l'injection d'une référence d'un bean.**

**Documentation officielle sur le site Spring**

## Comment fermer un contexte d'application en détruisant les Beans ?

**Auteurs :**

**Lors de l'utilisation de Spring en dehors d'un contexte Web, il est intéressant de fermer proprement un contexte d'application, en libérant les ressources et les beans.**

**La méthode close() de l'interface ConfigurableApplicationContext permet de faire cela. Il faut donc que l'implémentation du contexte d'application Spring implémente cette interface, et heureusement c'est le cas de ClasspathXmlApplicationContext, qui est sans doute la version la plus utilisée.**

```
ConfigurableApplicationContext context = new ClasspathApplicationContext("com/developpez/hikage/
context/applicationContext.xml");

// Termine le contexte Spring
context.close();
```

**Attention cependant, seul les beans de scope singleton seront détruits.**

**Une autre méthode de ConfigurableApplicationContext qui peut s'avérer utile est registerShutdownHook(). Cette méthode permet de spécifier à la JVM de fermer automatiquement le contexte Spring lorsque l'application s'arrête.**

**lien : Documentation de référence**

Sommaire > Remoting

## Comment exporter/utiliser un service avec Burlap ?

**Auteurs : Gildas Cuisinier ,**

**Burlap est un protocole basé sur XML et sur http pour exporter un service. Il est normalement indépendant du langage, mais actuellement principalement utilisé pour des applications Java à Java.**

**Afin d'exporter un service avec Burlap, il est nécessaire que celui-ci soit composé d'une interface et d'une implémentation.**
**Dès lors il est très simple d'exporter un service :**

```
<!-- Configuration du service en lui même -->
<bean id="monService" class="com.developpez.spring.remoting.MonServiceImpl">
 <!-- Définition des propriétés -->
</bean>

<bean name="/MonServiceBurlap" class="org.springframework.remoting.caucho.BurlapServiceExporter">
 <!-- Spécification de l'implémentation du service -->
 <property name="service">
  <ref bean="monService"/>
 </property>

 <!-- Spécification de l'interface du service à exporter -->
 <property name="serviceInterface">
  <value>com.developpez.spring.remoting.MonService</value>
 </property>

</bean>
```

**Le nom "/MonServiceBurlap" permet d'identifier l'URL d'accès à ce service, et pour cela, il est évidemment nécessaire que ces définitions soient faites dans le fichier de configuration de contexte d'une DispatcherServlet ( voir Spring MVC pour plus d'information ).**

**Par exemple, pour une Servlet de nom remoting, c'est dans un fichier WEB-INF/remoting-servlet.xml. Le fichier web.xml correspondant pourrait être :**

```
<servlet>
 <servlet-name>remoting</servlet-name>
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
 <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
 <servlet-name>remoting</servlet-name>
 <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

**Dès lors, si l'application Web est déployée et possède comme url d'accès http://localhost:8080/MonApplication, le service Burlap sera accessible via http://localhost:8080/MonApplication/remoting/MonServiceBurlap.**

**Au niveau du client, il existe un mécanisme similaire pour se connecter au service :**

```
<bean id="monServiceProxy" class="org.springframework.remoting.caucho.BurlapProxyFactoryBean">
 <!-- Configuration de l'url d'accès  -->
 <property name="serviceUrl">
  <value>http://localhost:8080/MonApplication/remoting/MonServiceBurlap</value>
```

```
  </property>

  <!-- Configuration de l'interface du service que le proxy doit implémenté  -->
  <property name="serviceInterface">
   <value>com.developpez.spring.remoting.MonService</value>
  </property>
 </bean>
```

**Dès lors, il est possible d'injecter monProxyService de manière tout à fait standard à tout Bean nécessitant une référence au service ( via son interface ).**

## Comment exporter/utiliser des services Web avec HttpInvoker ?

**Auteurs : Gildas Cuisinier ,**

**Contrairement à Burlaps ou Hessian qui utilisent des sérialisations binaire et XML indépendantes à Java, HttpInvoker utilise la sérialisation Java standard.**
**Cela implique que HttpInvoker ne fonctionne bien evidemment qu'entre deux applications Java, mais aussi que tous les objets qui sont transférés suivent les règles de sérialisation Java : Implémenter l'interface java.io.Serializable ainsi que de définir un serialVersionUID correctement.**

**De plus, HttpInvoker n'est disponible qu'avec Spring, il est donc nécessaire que le client et le serveur soient tout les deux basés sur Spring.**

**Afin d'exporter un service, la classe HttpInvokerServiceExporter sera utilisée :**

```
<!-- Configuration du service en lui même -->
<bean id="monService" class="com.developpez.spring.remoting.MonServiceImpl">
 <!-- Définition des propriétés -->
</bean>

<bean name="/
MonServiceHttpInvoker" class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
 <!-- Spécification de l'implémentation du service -->
 <property name="service">
  <ref bean="monService"/>
 </property>

 <!-- Spécification de l'interface du service à exporter -->
 <property name="serviceInterface">
  <value>com.developpez.spring.remoting.MonService</value>
 </property>

</bean>
```

**Le nom "/MonServiceHttpInvoker" permet d'identifier l'URL d'accès à ce service, et pour cela, il est évidemment nécessaire que ces définitions soient faites dans le fichier de configuration de contexte d'une DispatcherServlet ( voir Spring MVC pour plus d'information ).**

**Par exemple, pour une Servlet de nom remoting, c'est dans un fichier WEB-INF/remoting-servlet.xml. Le fichier web.xml correspondant pourrait être :**

```
<servlet>
 <servlet-name>remoting</servlet-name>
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
 <load-on-startup>1</load-on-startup>
```

```
</servlet>

<servlet-mapping>
 <servlet-name>remoting</servlet-name>
 <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

**Dès lors, si l'application Web est déployée et possède comme url d'accès http://localhost:8080/MonApplication, le service HttpInvoker sera accessible via http://localhost:8080/MonApplication/remoting/MonServiceHttpInvoker.**

**Au niveau du client, il existe un mécanisme similaire pour se connecter au service :**

```
<bean id="monServiceProxy" class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
 <!-- Configuration de l'url d'accès  -->
 <property name="serviceUrl">
  <value>http://localhost:8080/MonApplication/remoting/MonServiceHttpInvoker</value>
 </property>

 <!-- Configuration de l'interface du service que le proxy doit implémenté  -->
 <property name="serviceInterface">
  <value>com.developpez.spring.remoting.MonService</value>
 </property>
</bean>
```

**Dès lors, il est possible d'injecter monProxyService de manière tout à fait standard à tout Bean nécessitant une référence au service ( via son interface ).**

## Comment exporter/utiliser un service avec Hessian ?

**Auteurs : Gildas Cuisinier ,**

**Hessian est un protocole binaire, basé sur http pour exporter un service. Il est normalement indépendant du langage, mais actuellement principalement utilisé pour des applications Java à Java.**

**Afin d'exporter un service avec Hessian, il est nécessaire que celui-ci soit composé d'une interface et d'une implémentation.**
**Dès lors il est très simple d'exporter un service :**

```
<!-- Configuration du service en lui même -->
<bean id="monService" class="com.developpez.spring.remoting.MonServiceImpl">
 <!-- Définition des propriétés -->
</bean>

<bean name="/MonServiceHessian" class="org.springframework.remoting.caucho.HessianerviceExporter">
 <!-- Spécification de l'implémentation du service -->
 <property name="service">
  <ref bean="monService"/>
 </property>

 <!-- Spécification de l'interface du service à exporter -->
 <property name="serviceInterface">
  <value>com.developpez.spring.remoting.MonService</value>
 </property>

</bean>
```

**Le nom "/MonServiceHessian" permet d'identifier l'URL d'accès à ce service, et pour cela, il est évidemment nécessaire que ces définitions soient faites dans le fichier de configuration de contexte d'une DispatcherServlet ( voir Spring MVC pour plus d'information ).**

**Par exemple, pour une Servlet de nom remoting, c'est dans un fichier WEB-INF/remoting-servlet.xml. Le fichier web.xml correspondant pourrait être :**

```
<servlet>
 <servlet-name>remoting</servlet-name>
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
 <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
 <servlet-name>remoting</servlet-name>
 <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

**Dès lors, si l'application Web est déployée et possède comme url d'accès http://localhost:8080/MonApplication, le service Hessian sera accessible via http://localhost:8080/MonApplication/remoting/MonServiceHessian.**

**Au niveau du client, il existe un mécanisme similaire pour se connecter au service :**

```
<bean id="monServiceProxy" class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
 <!-- Configuration de l'url d'accès  -->
 <property name="serviceUrl">
  <value>http://localhost:8080/MonApplication/remoting/MonServiceHessian</value>
 </property>

 <!-- Configuration de l'interface du service que le proxy doit implémenté  -->
 <property name="serviceInterface">
  <value>com.developpez.spring.remoting.MonService</value>
 </property>
</bean>
```

**Dès lors, il est possible d'injecter monProxyService de manière tout à fait standard à tout Bean nécessitant une référence au service ( via son interface ).**

## Comment intégrer Apache Axis2 et Spring ?

**Auteurs : Righetto Dominique ,**

**Pour intégrer Apache Axis2 et Spring de manière à ce que ce dernier gére les instances des web services, il faut déclarer le bean qui représente le web service dans le fichier de configuration du web service, c'est à dire dans le fichier "services.xml" via le tag "SpringBeanName"**

```
<parameter name="SpringBeanName" locked="false">monBeanWebService</parameter>
```

**ET indiquer à Axis que Spring est le fournisseur d'instances, via le tag "ServiceObjectSupplier" dans le fichier "services.xml"**

```
<service name="MonService">
    <description>
```

```
        Mon super service
    </description>
    <messageReceivers>
        <messageReceiver
            mep="http://www.w3.org/2004/08/wsdl/in-out"
      class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
    </messageReceivers>

  <parameter name="ServiceObjectSupplier" locked="false">org.apache.axis2.extensions.spring.receivers.SpringServ
    <parameter name="SpringBeanName" locked="false">monBeanWebService</parameter>
    <parameter name="useOriginalwsdl">true</parameter>
</service>
```

**Les lignes importantes dans ce fichier sont :**

```
<parameter name="ServiceObjectSupplier" locked="false">org.apache.axis2.extensions.spring.receivers.SpringServle
<parameter name="SpringBeanName" locked="false">monBeanWebService</parameter>
```

**"monBeanWebService" est un bean (une classe publique non abstraite non finale avec des méthodes publiques qui représentent les services exposés) qui est déclaré dans un fichier de configuration Spring.**

Sommaire > Intégration d'API

## Comment créer une tache plannifiée avec l'API Timer et Spring ?

**Auteurs : Gildas Cuisinier ,**

**Il est tout à fait possible d'utiliser l'API Timer de Java afin de gérer une plannification de tâches avec Spring.**
**Pour cela, la première étape est de créer un Bean qui possèdera une méthode qui représentera la tâche en question :**

```
public class JobTest {

    public void lancementProgramme(){
        System.out.println("Lancement");
    }
}
```

**Ce bean doit bien evidemment être défini dans le contexte Spring :**

```
<bean id="tachePlannifie" class="com.developpez.spring.Tache">
   </bean>
```

**Ce Bean peut être configurer via Spring comme n'importe quel Bean afin d'avoir accès par exemple à une base de données, ou autre. Notons par la même occasion que cet objet est un simple POJO et n'est aucunement lié à un framework spécifique. Ce Bean n'hérite d'aucune classe, n'implémente aucune interface et il n'y a pas de standard défini en ce qui concerne le nom de la méthode qui représente la tâche.**

**L'étape suivant est de définir un Bean qui va justement faire la liaison entre un gestionnaire de tâches et notre précedent Bean afin de spécifier quelle méthode doit justement être utilisée :**

```
<bean id="tachePlannifieTask" class="org.springframework.scheduling.timer.MethodInvokingTimerTaskFactoryBean">
       <property name="targetObject"><ref bean="tachePlannifie"/> </property>
        <property name="targetMethod"><value>lancementProgramme</value></property>
   </bean>
```

**Maintenant qu'on a définit "Quoi faire", il est nécessaire de spécifier "Quand le faire". Pour cela, il faut utiliser une classe ScheduledTimerTask :**

```
<bean id="tachePlannifieScheduledTask" class="org.springframework.scheduling.timer.ScheduledTimerTask">
        <!-- Attendre une minute avant le premier lancement -->
        <property name="delay" value="60000"/>
        <!-- Et relancer ensuite toutes les 10 minutes -->
        <property name="period" value="600000"/>
        <property name="timerTask">
            <ref bean="tachePlannifieTask"/>
        </property>
    </bean>
```

**Ici, une minutes d'attente ( 60 000 millisecondes ) avant le premièr lancement sera réalisée, et ensuite la tâche sera relancée toutes les 10 minutes ( 600 000 millisecondes ).**

**Une fois que le quand et le quoi sont défini, il est nécessaire de provoquer le lancement de cette tâche, et cela est géré par le TimerFactoryBean :**

```
<bean id="timerFactory" class="org.springframework.scheduling.timer.TimerFactoryBean">
        <property name="scheduledTimerTasks">
            <list>
                <ref bean="tachePlannifieScheduledTask"/>
            </list>
        </property>
    </bean>
```

**Cette fabrique possède une propriété scheduledTimerTasks de type liste de ScheduledTimerTask. Elle permet donc de gérer plusieurs tâches plannifiées.**

**lien : JDK Timer support, Documentation officielle**

## Comment intégrer EhCache et Spring ?

**Auteurs : Righetto Dominique ,**

**Pour intégrer EhCache et Spring afin de pouvoir injecter directement un bean Cache, on définit le bean ci-dessous dans un des fichiers de définition du contexte Spring :**

```
<bean id="customCache" class="org.springframework.cache.ehcache.EhCacheFactoryBean">
  <property name="cacheManager">
   <bean class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">
    <property name="configLocation"
     value="classpath:ehcache.xml" />
    <property name="shared" value="false" />
   </bean>
  </property>
  <property name="cacheName" value="SampleConfigOne" />
 </bean>
```

**On place la valeur de la propriété cacheName sur le nom d'une des définitions de cache spécifiées dans le fichier ehcache.xml.**

**Dans cette déclaration, on précise que le fichier de définition des caches se trouve dans le classpath au niveau de la racine (on pourrait également utiliser classpath*:ehcache.xml pour lui indiquer de chercher ce fichier dans tous les classpath). On précise également que le CacheManager n'est pas partagé via la propriété shared.**

**Ensuite on peut injecter ce bean normalement dans tout bean qui en aurait besoin via :**

```
<bean >
     <property name="cache" value="customCache" />
</bean>
```

**lien : Site du projet EhCache**

## Comment créer une tâche planifiée avec Spring et Quartz ?

**Auteurs : Gildas Cuisinier ,**

**Quartz est une API destinée à la création de tâches plannifiées. Elle possède certains avantages par rapport à la méthode basée sur le Timer :**

- **Planification plus fine, avec une synthaxe basée sur le Crontab Linux**

- **Possibilité de rendre les tâches persistantes, afin d'avoir un historique des exécutions entres les différents lancements d'une application**
- **Possibilité d'intégration avec JTA pour la gestion des transactions**

**Spring possède une intégration avec celle-ci, et permet l'exécution de tâches basées sur des POJOs.**

**La première étape est donc de créer une tâche, par la définition d'un bean :**

```
ublic class MonJob {

    public void jobMethod(){
        // Code de la méthode
    }
}
```

```
  <bean id="MonJob" class="com.developpez.spring.quartz.MonJob"/>
```

**La tâche est donc bien un POJO, et le nom de la méthode du job est au choix du développeur.**

**L'étape suivante est d'utiliser une classe fournie par Spring, qui va crée le JobDetail ( la tâche qui sera compréhensible par Quartz ) :**

```
<bean name="tache" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
        <property name="targetObject" ref="MonJob"/>
        <property name="targetMethod" value="jobMethod"/>
    </bean>
```

**Il suffit donc de définir la propriete *targetObject* avec une référence vers le bean défini juste avant, et la propriété *targetMethod* avec le nom de la méthode à utiliser.**

**Une fois la tâche définie, il est nécessaire de définir le moment où elle doit être exécutée. Pour cela, deux classes existent : une pour une planification simple, une autre pour une planification plus complexe :**

```
  <!-- Méthode simple -->
  <bean id="triggerTache" class="org.springframework.scheduling.quartz.SimpleTriggerBean">
        <property name="jobDetail" ref="tache"/>
        <!-- Démarre la première fois après 10 secondes -->
        <property name="startDelay" value="10000"/>
        <!-- Ensuite toute les 50 secondes -->
        <property name="repeatInterval" value="50000"/>
    </bean>

      <!-- Méthode complexe -->
    <bean id="triggerTache2" class="org.springframework.scheduling.quartz.CronTriggerBean">
        <property name="jobDetail" ref="tache"/>
        <!-- Exécution toute les 5 secondes -->
        <property name="cronExpression" value="0/5 * * * * ?"/>
    </bean>
```

**Le premier trigger est simple, il suffit de définir le moment de la première exécution et ensuite la période d'éxecution. Le deuxième est un peu plus complexe et se base sur une syntaxe Crontab : *seconde minute heure jour-dans-le-mois mois jour-dans-la-semaine annee***

**La dernière étape est de définir un gestionnaire de planification :**

```
<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="triggerTache2"/>
        </list>
    </property>
</bean>
```

**lien : Format des expressions Cron**

## Comment créer un contexte Spring dans une application Web?

**Auteurs : Gildas Cuisinier ,**

**Dans une application Web, il n'est pas utile de créer l'ApplicationContext à la main via une des implémentations de cette interface.**
**En effet, Spring fourni un Listener afin de gérer cela :**

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

**Par défaut, ce Listener va charger un fichier /WEB-INF/applicationContext.xml, et stocker l'ApplicationContext dans le ServletContext.**
**Il est possible de spécifier un ou plusieurs fichiers à charger à la place du fichier par defaut. Pour cela, il faut utiliser un paramètre de contexte :**

```
<context-param>
    <param-name>contextConfigLocation</param-name>
        <!-- Chargement de tout fihcier xml commençant par applicationContext -->
    <param-value>/WEB-INF/applicationContext*.xml</param-value>
</context-param>
```

**Ensuite, dans les servlets ( ou toute autre classe possèdant une référence au ServletContext ), il est possible de récupérer l'applicationContext via :**

```
WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

**lien : Documentation officielle sur l'intégration de Spring dans les applications Web**

## Comment utiliser Spring avec JSF ?

**Auteurs : Gildas Cuisinier , djo.mos ,**

**Une fois qu'un context Spring a été crée dans l'application Web, il est possible d'injecter un Bean Spring dans une Bean JSF. Pour ce faire, il faut tout d'abord ajouter un variable-resolver dans le faces-config.xml :**

```
faces-config>
  <application>
    <variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver</variable-resolver>
    </application>
</faces-config>
```

**Une fois cela fait, il est possible de lier un Bean Spring comme une propriété du Bean JSF :**

```
<managed-bean>
  <managed-bean-name>monBeanJSF</managed-bean-name>
 <managed-bean-class>com.developpez.spring.jsf.MonBeanJSF</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
```

```
    <property-name>userService</property-name>
    <value>#{userService}</value>
  </managed-property>
</managed-bean>
```

**Dans cet exemple, #{userService} fait référence à un Bean défini dans Spring, avec le nom userService.**

**Méthode Spring 2.5**

**Spring 2.5 offre enfin une nouvelle méthode d'intégration avec JSF compatible avec la version 1.2 (qui ne repose pas sur le VariableResolver, déprécié depuis JSF 1.2).**
**Pour cela, il faut ajouter dans faces-config ce fragment xml :**

```
<application>
<el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver
  </el-resolver>
</application>
```

**Le reste fonctionne exactement de la même façon qu'avec le DelegatingVariableResolver.**

## Comment maintenir une session Hibernate afin de pouvoir parcourir des objets dans une vue ?

**Auteurs : Michael Courcy ,**

**Lors de l'utilisation d'Hibernate dans une application Web, il est courant d'avoir des problèmes lors de l'utilisation des objets Hibernate dans une vue ( JSP, Velocity ou autre ). La cause de cela, est la stratégie par défaut de Hibernate, qui ne charge pas toutes les relations d'une entité automatiquement, mais seulement lorsqu'on tente réellement de les utiliser.**

**Le problème est qu'il est nécessaire d'avoir une Session hibernate ouverte, et qu'habituellement, celle-ci est fermée dans la couche service.**

**Pour résoudre le problème, si la SessionFactory est configurée dans un fichier de contexte Spring, un filtre est à notre disposition pour garder la session ouverte : OpenSessionInViewFilter.**
**Il suffit donc de l'ajouter dans le fichier web.xml :**

```
<filter>
 <filter-name>Hibernate Session In View Filter</filter-name>
       <filter-class>org.springframework.orm.hibernate3.support.OpenSessionInViewFilter</filter-
class>
</filter>
<filter-mapping>
 <filter-name>Hibernate Session In View Filter</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
```

**lien : Javadoc OpenSessionInViewFilter**

Sommaire > Accès aux données

## Comment intégrer Toplink et Spring ?

**Auteurs : Righetto Dominique ,**

**Pour intégrer Oracle Toplink et Spring il faut déclarer un bean "sessionFactory" dans lequel on va préciser :**

- **L'endroit où se trouve le fichier de session qui contient lui même l'endroit où se trouve le descripteur de mapping**
- **La source de données**
- **Le logger**

```
<bean id="sessionFactory" class="org.springframework.orm.toplink.LocalSessionFactoryBean">
  <property name="configLocation" value="toplink-sessions.xml"/>
  <property name="dataSource" ref="dataSource"/>
  <property name="sessionLog">
   <bean class="org.springframework.orm.toplink.support.CommonsLoggingSessionLog"/>
  </property>
</bean>
```

**Dans cet exemple on indique que :**

- **Le fichier de session Toplink se trouve à la racine du classpath de l'application**
- **La source de données est un bean déja défini et nommé "dataSource"**
- **Le logger est CommonsLogging et il est géré par Spring**

**Voici le contenu de mon fichier de session "toplink-sessions.xml" qui indique que le fichier de mapping "toplink-Mapping.xml" est lui aussi à la racine du classpath de l'application :**

```
<?xml version="1.0" encoding="UTF-8"?>
<toplink-sessions version="10g Release 3 (10.1.3.0.0)" xmlns:xsd="http://www.w3.org/2001/
XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <session xsi:type="server-session">
      <name>Session</name>
      <event-listener-classes/>
      <primary-project xsi:type="xml">toplink-Mapping.xml</primary-project>
   </session>
</toplink-sessions>
```

**Note : Il faut enlever dans le fichier de mapping Toplink les informations de connexion à la base de données. En effet, Spring s'occupant de fournir la source de données ( via le dataSource ), il détecte un conflit et ne peut pas créer une instance de la sessionFactory.**

**Ensuite pour utiliser ce bean "sessionFactory" il suffit d'hériter de la classe "org.springframework.orm.toplink.support.TopLinkDaoSupport" et d'utiliser le ToplinkTemplate via "getTopLinkTemplate()" pour exécuter des traitements.**

```
public class MonDaoImpl extends TopLinkDaoSupport implements MonDao {
//Placer les méthodes utilisant "getTopLinkTemplate()" ici...
}
```

**Sans oublier de configurer le bean correctement, en lui fournissant une instance de la sessionFactory :**

```
<bean id="monDao" class="com.drighetto.dao.impl.MonDaoImpl">
 <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

**lien : Section Toplink dans la documentation officielle**

**lien : Documentation Toplink**

## Comment intégrer JPA et Spring ?

**Auteurs : Righetto Dominique , djo.mos ,**

**Voici les déclarations à effectuer pour intégrer JPA et Spring (cet exemple se base sur l'utilisation de l'implémentation Oracle Toplink de JPA) :**

**Etape 1 : Déclaration dans le contexte Spring des dépendances JPA**

**Activation du tissage lors du runtime pour le contexte Spring afin que tous les beans implémentant l'interface "LoadTimeWeaverAware" (comme le bean LocalContainerEntityManagerFactoryBean) reçoivent une référence vers le tisseur (cf documentation Spring pour plus de précisions).**

**Remarque, le load time weaving n'est nécessaire qu'avec l'implémentation Toplink**

```
<context:load-time-weaver />
```

**Déclaration du "PersistenceUnitManager" permettant de personnaliser la sélection des unités de persistance et des sources de données.**
**(*Cette étape n'est pas obligatoire* )**

```
<bean id="persistenceUnitManager" class="org.springframework.orm.jpa.persistenceunit.DefaultPersistenceUnitMana
 <!-- On spécifie ici les lieux où trouver les fichiers de persistence -->
 <property name="persistenceXmlLocations">
  <list>
   <value>classpath*:META-INF/persistence.xml</value>
  </list>
 </property>
 <!-- On spécifie ici les sources de données à utiliser, locale ou distante -->
 <property name="dataSources">
  <map>
   <entry key="localDataSource" value-ref="dataSource" />
   <!--<entry key="remoteDataSource" value-ref="remote-db" />-->
  </map>
 </property>
 <!-- On spécifie ici la sources de données par défaut si aucune source de données n'est disponible --
>
 <property name="defaultDataSource" ref="dataSource" />
</bean>
```

**Déclaration de l' "EntityManagerFactory" permettant de fournir les instances des gestionnaires d'entités :**

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
 p:dataSource-ref="dataSource"
 p:persistenceUnitManager-ref="persistenceUnitManager">
 <!-- On spécifie ici l'adaptateur Spring pour l''implémentation JPA utilisée -->
 <property name="jpaVendorAdapter">
```

```
    <bean class="org.springframework.orm.jpa.vendor.TopLinkJpaVendorAdapter" p:databasePlatform="oracle.toplink.es
      p:showSql="false" />
    </property>
    <!-- On spécifie ici le tisseur utilisée pour la modification du ByteCode, cf documentation de Spring pour plu
>
    <property name="loadTimeWeaver">
     <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver" />
    </property>
    <!-- On spécifie ici le dialecte utilisé en fonction de l'implémentation JPA utilisée -->
    <property name="jpaDialect">
     <bean class="org.springframework.orm.jpa.vendor.TopLinkJpaDialect" />
    </property>
   </bean>
```

**Déclaration du "TransactionManager" qui est le gestionnaire de transactions**

```
<bean id="txManager"
class="org.springframework.orm.jpa.JpaTransactionManager"
  p:entityManagerFactory-ref="entityManagerFactory">
  <!-- On spécifie ici le dialecte utilisé en fonction de l' implémentation JPA utilisée -->
 <property name="jpaDialect">
                  <bean class="org.springframework.orm.jpa.vendor.TopLinkJpaDialect" />
 </property>
</bean>
```

**Activation de la prise en compte des annotations de type @Required,@Autowired,@PostConstruct,@PreDestroy,@Resource,@PersistenceContext,@PersistenceUnit :**

```
<context:annotation-config />
```

**Déclaration d'un traducteur d'exception :**

```
 <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />
```

**Etape 2 : Implémentation d'une classe utilisant JPA**

**Chaque classe qui désire utiliser JPA pour accéder aux données peut :**

- **Soit hériter de la classe "org.springframework.orm.jpa.support.JpaDaoSupport"**
- **Soit se faire injecter un attribut de type "org.springframework.orm.jpa.JpaTemplate"**

**Dans le cas de l'héritage voici le type de déclaration à effectuer pour déclarer la classe (dépendance sur l' "EntityManagerFactory")**

```
<bean id="myDao" class="com.drighetto.springjpa.dao.impl.DaoJpaImpl"
 p:entityManagerFactory-ref="entityManagerFactory" />
```

***Note : Pour le moment l'utilisation de JPA avec Spring ne supporte que l'isolation par défaut pour l'isolation des transactions... Dans la classe org.springframework.orm.jpa.DefaultJpaDialect (la classe "org.springframework.orm.jpa.vendor.TopLinkJpaDialect" hérite de cette classe) dans la méthode beginTransaction() une vérification est faite sur l'isolation placée et si celle-ci n'est pas placée à défaut alors l'exception suivante est levée "Standard JPA does not support custom isolation levels - use a special JpaDialect for your JPA implementation".***

**Voici le fichier de contexte dans son ensemble :**

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
 ****************************************
 Application context for the project
 ****************************************
-->
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:p="http://www.springframework.org/schema/p"
 xmlns:aop="http://www.springframework.org/schema/aop"
 xmlns:context="http://www.springframework.org/schema/context"
 xmlns:jee="http://www.springframework.org/schema/jee"
 xmlns:tx="http://www.springframework.org/schema/tx"
 xmlns:util="http://www.springframework.org/schema/util"
 xsi:schemaLocation="
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-
aop-2.5.xsd
   http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-
beans-2.5.xsd
   http://www.springframework.org/schema/context http://www.springframework.org/schema/context/
spring-context-2.5.xsd
   http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-
jee-2.5.xsd
   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-
tx-2.5.xsd
   http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-
util-2.5.xsd">



 <!-- ========================= RESOURCE DEFINITIONS ========================= -->

 <!--
  Activates a load-time weaver for the context. Any bean within the context that
  implements LoadTimeWeaverAware (such as LocalContainerEntityManagerFactoryBean)
  will receive a reference to the autodetected load-time weaver.
 -->
 <context:load-time-weaver />

 <!-- DataSource -->
 <bean id="dataSource"
  class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close"
  p:driverClassName="oracle.jdbc.driver.OracleDriver"
  p:url="jdbc:oracle:thin:@localhost:1521:xe" p:username="MyTestUser"
  p:password="MyTestUser" />

 <!-- JNDI DataSource for JEE environments -->
 <!--
  <jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/petclinic"/>
 -->

 <!-- JPA PersistenceUnitManager used to customize the selection of the persistence unit and the datasources --
>
 <bean id="persistenceUnitManager"
  class="org.springframework.orm.jpa.persistenceunit.DefaultPersistenceUnitManager">
  <!-- Multiple value can be specified here -->
  <property name="persistenceXmlLocations">
   <list>
    <value>classpath*:META-INF/persistence.xml</value>
   </list>
  </property>
```

```
  <property name="dataSources">
   <map>
    <entry key="localDataSource" value-ref="dataSource" />
    <!--<entry key="remoteDataSource" value-ref="remote-db" />-->
   </map>
  </property>
  <!-- if no datasource is specified, use this one -->
  <property name="defaultDataSource" ref="dataSource" />
 </bean>

 <!-- JPA EntityManagerFactory -->
 <bean id="entityManagerFactory"
  class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
  p:dataSource-ref="dataSource"
  p:persistenceUnitManager-ref="persistenceUnitManager">
  <property name="jpaVendorAdapter">
   <bean
    class="org.springframework.orm.jpa.vendor.TopLinkJpaVendorAdapter"
    p:databasePlatform="oracle.toplink.essentials.platform.database.oracle.OraclePlatform"
    p:showSql="false" />
  </property>
  <property name="loadTimeWeaver">
   <bean
    class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver" />
  </property>
  <property name="jpaDialect">
   <bean
    class="org.springframework.orm.jpa.vendor.TopLinkJpaDialect" />
  </property>
 </bean>

 <!-- Transaction manager for a single JPA EntityManagerFactory (alternative to JTA) -->
 <bean id="txManager"
  class="org.springframework.orm.jpa.JpaTransactionManager"
  p:entityManagerFactory-ref="entityManagerFactory">
  <property name="jpaDialect">
   <bean
    class="org.springframework.orm.jpa.vendor.TopLinkJpaDialect" />
  </property>
 </bean>



 <!-- ========================= CONFIG DEFINITIONS ========================= -->

 <!--
  Activates various annotations to be detected in bean classes: Spring's
  @Required and @Autowired, as well as JSR 250''s @PostConstruct,
  @PreDestroy and @Resource (if available) and JPA's @PersistenceContext
  and @PersistenceUnit (if available).
 -->
 <context:annotation-config />

 <!-- Exception translation bean post processor -->
 <bean
  class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />



 <!-- ================================== TRANSACTIONS DEFINITIONS ================================== -
>
  <!-- For the moment JPA only support ISOLATION_DEFAULT for the transaction isolation -->

  <!-- Define pointcut for txAdvices -->
  <aop:config>
   <!-- DAO Layer -->
```

```
  <aop:advisor advice-ref="txAdviceDao"
   pointcut="execution(* com.drighetto.springjpa.dao.impl.*.*(..))" />
  <!-- Service Layer -->
  <aop:advisor advice-ref="txAdviceService"
   pointcut="execution(* com.drighetto.springjpa.services.*.*(..))" />
 </aop:config>

 <!-- the transactional advice for DAO layer -->
 <tx:advice id="txAdviceDao" transaction-manager="txManager">
  <!-- the transactional semantics... -->
  <tx:attributes>
   <!-- Read methods don't use a transaction -->
   <tx:method name="read*" propagation="SUPPORTS"
    read-only="true" />
   <!-- Exclude Getter/Setter -->
   <tx:method name="set*" propagation="SUPPORTS"
    read-only="true" />
   <tx:method name="get*" propagation="SUPPORTS"
    read-only="true" />
   <!-- All others methods must use a existing transaction -->
   <tx:method name="*" isolation="DEFAULT" timeout="10"
    propagation="MANDATORY" read-only="false"
    rollback-for="org.springframework.dao.DataAccessException" />
  </tx:attributes>
 </tx:advice>

 <!-- the transactional advice for Service layer -->
 <tx:advice id="txAdviceService" transaction-manager="txManager">
  <!-- the transactional semantics... -->
  <tx:attributes>
   <!-- Read methods don't use a transaction -->
   <tx:method name="display*" propagation="SUPPORTS"
    read-only="true" />
   <!-- Exclude Getter/Setter -->
   <tx:method name="set*" propagation="SUPPORTS"
    read-only="true" />
   <tx:method name="get*" propagation="SUPPORTS"
    read-only="true" />
   <!-- All others methods create a transaction -->
   <tx:method name="*" isolation="DEFAULT" timeout="10"
    propagation="REQUIRES_NEW" read-only="false"
    rollback-for="org.springframework.dao.DataAccessException" />
  </tx:attributes>
 </tx:advice>




 <!-- ================================ BEANS DEFINITIONS ================================ -->

 <!-- DAO -->
 <bean id="myDao" class="com.drighetto.springjpa.dao.impl.DaoJpaImpl"
  p:entityManagerFactory-ref="entityManagerFactory" />

 <!-- Service -->
 <bean id="myService"
  class="com.drighetto.springjpa.services.Processor"
  p:myDao-ref="myDao" />

</beans>
```

**lien : Section JPA sur la documentation officielle**