Java Quartz avec Spring

par TheBigJim

Date de publication : 10 septembre 2010

Ce tutorial a pour but de montrer comment faire de l'ordonnancement de tâches (comme le fait cron) avec Java. D'abord nous utiliserons l'API Quartz avec Spring 2.5, puis nous utiliserons Spring 3 avec le namespace task. La méthode est pragmatique, quasimment prête à l'emploi.

I - Introduction

I-A - Quoi ? Qu'est-ce ?

Quartz est une API java d'ordonnancement de tâches qui permet d'automatiser le lancement de traitements oneshot ou bien réguliers.
Il en va de même pour Spring 3 et le namespace task.

I-B - Objectif

Montrer comment:

- utiliser Quartz, dans un environnement Spring 2.5;
- utiliser Spring 3 et le namespace task.

I-C - Outils

Java implicitement, Eclipse notre IDE java, maven pour les dépendances nécessaires au développement de notre tutorial.

I-D - Connaissances prérequises souhaitées mais non obligatoires

- Eclipse IDE;
- Spring framework.

II - Préparation de notre environnement

II-A - Création du projet maven

Démarrer Eclipse et lancer le wizard de création de projet via le menu **File/New/Other** ou via le raccourci clavier **CTRL+N**;

Dans la liste qui apparaît, sélectionner Maven/Maven Project et cliquer sur le bouton **Next**;

Cocher la case Create a simple project et cliquer sur le bouton **Next**;

Identifier votre projet en renseignant les zones *Group Id* et *Artifact Id*, puis cliquer sur le bouton **Finish**: votre projet est créé.

II-B - Quartz et Spring 2.5

II-B-1 - Les dépendances à déclarer

Editer le fichier pom.xml et ajouter les dépendances suivantes juste avant la balise </project> :

```
<dependencies>
 <dependency>
  <groupId>opensymphony</groupId>
  <artifactId>quartz</artifactId>
  <version>1.6.1</version>
 </dependency>
```

```
 <dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring</artifactId>
  <version>2.5.6</version>
 </dependency>
 <dependency>
  <groupId>commons-collections</groupId>
  <artifactId>commons-collections</artifactId>
  <version>3.1</version>
 </dependency>
</dependencies>
```

Sauvegarder le fichier.

II-B-2 - Le fichier de configuration Spring

Dans le dossier /src/main/resources, créer un fichier nommé applicationContext.xml dont le contenu est pour le moment:

```
<?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.xsd">
</beans>
```

II-B-3 - Developpement du job

II-B-3-a - Architecture de notre exemple

- Niveau 4 : Le main (programme principal) qui lancera le job;
- Niveau 3 : Le job Quartz qui lancera le traitement à réaliser via le business manager;
- Niveau 2 : Le business manager : les traitements métiers (besoin fonctionnel) utilisateurs de la couche de niveau 1;
- Niveau 1 : Le ou les objets permettant la satisfaction du ou des traitements métier.

II-B-3-b - Création du business manager (Niveau 2)

Le business manager de notre exemple n'est qu'un simple objet intégrant une méthode (satisfaisant notre besoin fonctionnel) qui sera appelée par notre job.

```
package quartz.exemple.manager;

public class BusinessManager {
 public void runAction() {
  System.out.println("In business manager, I call the business action") ;
 }
}
```

II-B-3-c - Création du job (Niveau 3)

Un job n'est qu'un simple **POJO** intégrant une méthode d'exécution du job. Pour satisfaire à notre architecture, ce job intègre aussi un objet **Manager** que l'on nommera génériquement pour l'exemple **BusinessManager**, **BusinessManager** dont on appelle une méthode d'exécution satisfaisant un besoin fonctionnel. Pour permettre l'injection par Spring du manager, la définition d'un accesseur (setter) est nécessaire.

```
package quartz.exemple.job;
import quartz.exemple.manager.BusinessManager;

public class QuartzExempleJob {

 private BusinessManager businessManager;

 public void execute(){
  System.out.println("In quartz job, I call the business manager") ;
  businessManager.runAction();
 }

 public void setBusinessManager(BusinessManager businessManager) {
  this.businessManager = businessManager;
 }
}
```

II-B-3-d - Configuration du job dans Spring

Dans le fichier applicationContext.xml, nous déclarons :

- le business manager

```
<bean id="businessManager" class="quartz.exemple.manager.BusinessManager"/>
```

- le POJO job et l'injection du business manager dans ce job

```
<bean id="QuartzExempleJob" class="quartz.exemple.job.QuartzExempleJob">
 <property name="businessManager" ref="businessManager"/>
</bean>
```

- le lien entre le POJO job et l'API quartz (objet + méthode appelée)

```
<bean name="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
 <property name="targetObject" ref="QuartzExempleJob"/>
 <property name="targetMethod" value="execute"/>
</bean>
```

- les paramètres de notre job (fréquence d'exécution) via une **cron expression**

```
<bean id="jobCronScheduler" class="org.springframework.scheduling.quartz.CronTriggerBean">
 <property name="jobDetail" ref="jobDetail"/>
 <property name="cronExpression" value="1 * * ? * *"/>
</bean>
```

- le lancement de notre job dans lequel on spécifie soit un lancement (implicite) automatique au démarrage par Spring (lazy-init=false), soit un lancement (explicite) manuel à la demande de l'utilisateur (lazy-init=true)

```
<bean id="schedulerBean"
 class="org.springframework.scheduling.quartz.SchedulerFactoryBean"
 lazy-init="false">
 <property name="triggers">
  <list>
   <ref bean="jobCronScheduler"/>
  </list>
 </property>
</bean>
```

II-B-3-e - Création du main (Niveau 4)

Notre main ne fait que charger le fichier de configuration Spring.

```
package quartz.exemple;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App
{
 public static void main( String[] args )
 {
  System.out.println( "Loading Spring applicationContext.xmlfile...." );
  ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");
  System.out.println( "Spring applicationContext.xml loaded");
 }
}
```

II-C - Spring 2.5, Spring 3, ça change quoi ?

Eh bien pas grand chose, seules les dépendances maven diffèrent.
Toutefois Spring 3 dispose d'un namespace task qui permet de se passer de Quartz et qui fait quasiment la même chose.

II-D - Spring 3

Dans cette partie, nous utilisons des annotations qui existent déjà dans Spring 2.5. Spring 3 apporte quant à lui le namespace task, configurable dans le fichier de configuration ou par les annotations. Avant de commencer cette partie, créer un nouveau projet Maven sous Eclipse.

II-D-1 - Les dépendances Spring 3 à déclarer

Editer le fichier pom.xml et ajouter les dépendances suivantes juste avant la balise </project> :

```
<dependencies>
 <dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>3.0.0.RELEASE</version>
 </dependency>
 <dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>3.0.0.RELEASE</version>
 </dependency>
</dependencies>
```

Sauvegarder le fichier.

II-D-2 - Le fichier de configuration Spring

Dans le dossier /src/main/resources, créer un fichier nommé applicationContext.xml dont le contenu est pour le moment:

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

```
 xmlns:context="http://www.springframework.org/schema/context"
 xmlns:task="http://www.springframework.org/schema/task"
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/
beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/task http://www.springframework.org/schema/task/spring-
task-3.0.xsd
  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-
context-3.0.xsd">
</beans>
```

II-D-3 - Développement du job

Pour éviter les répétitions, l'exemple est ici volontairement simplifier par rapport à ce qui a été abordé avec Quartz. L'adaptation à l'architecture de l'exemple Quartz étant triviale. Nous ne ferons donc que créer le job et son ordonnancement dans différentes configurations, notamment via les fichiers de configuration et/ou les annotations.

II-D-3-a - Exclusivement par fichier de configuration

II-D-3-a-i - Le Job, un objet simple

Créer un package job, puis créer la classe TheJob dans ce package.

```
package job;

public class TheJob {

 public void runAction() {
  System.out.println("I'm the Job");
 }
}
```

II-D-3-a-ii - La configuration de l'ordonnancement sous Spring

La déclaration de la classe de notre job (un bean) est ici classique.
Pour l'ordonnancement, nous faisons appel au namespace task dans lequel nous indiquons :
- le bean ainsi que la méthode qui sera appelée au lancement de la tâche;
- l'expression **cron** qui va bien pour l'ordonnancement de la tâche.

```
<bean id="job" class="job.Job"/>

 <task:scheduled-tasks>
  <task:scheduled ref="job" method="runAction" cron="3/10 * * * * ?"/>
 </task:scheduled-tasks>
```

II-D-3-a-iii - Lancement de notre job

Ici c'est du classique, on crée une classe Main qui chargera notre fichier applicationContext.xml afin que Spring démarre notre job.

```
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

 public static void main(String[] args) {
  new ClassPathXmlApplicationContext("applicationContext.xml");
 }
```

```
}
```

II-D-3-b - Par les annotations

II-D-3-b-i - Le Job

Modifier la classe **TheJob** en ajoutant l'annotation **@Repository("theJob")** où "theJob" sera le nom du bean déclaré. Cette annotation n'est pas exclusive à Spring 3, Spring 2.5 l'implémente aussi.
Ce qui donne:

```
package job;

import org.springframework.stereotype.Service;

@Repository("theJob")
public class TheJob {

 public void runAction() {
  System.out.println("I'm the Job");
 }
}
```

II-D-3-b-ii - La configuration de l'ordonnancement sous Spring

Modifier le fichier **applicationContext.xml** en remplaçant la balise **bean** par la balise **context**.
La balise **context:component-scan** indique à Spring d'explorer le package donné (spécifié dans la propriété base-package) à la recherche d'annotations. En l'occurence, notre job est annoté avec l'annotation **@Repository("theJob")**, ce qui en fera un bean nommé **theJob**. Cette balise n'est pas exclusive à Spring 3, Spring 2.5 l'implémente aussi.
Ce qui donne:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xmlns:task="http://www.springframework.org/schema/task"
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/
beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/task http://www.springframework.org/schema/task/spring-
task-3.0.xsd
  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-
context-3.0.xsd">

 <context:component-scan base-package="job"/>

 <task:scheduled-tasks>
  <task:scheduled ref="theJob" method="runAction" cron="3/10 * * * * ?"/>
 </task:scheduled-tasks>

</beans>
```

II-D-3-b-iii - Lancement de notre job

Même chose que précédemment.

```
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {
```

```
 public static void main(String[] args) {
  new ClassPathXmlApplicationContext("applicationContext.xml");
 }
}
```

II-D-3-c - Un peu plus loin avec les annotations

Ici, le fichier de configuration Spring sera alimenté avec le strict minimum.

II-D-3-c-i - Le Job

Modifier la classe **TheJob** en ajoutant l'annotation **@Component("theJob") (déclaration du bean)** sur la classe et l'annotation **@Scheduled(cron="3/10 * * * * ?")** sur la méthode runAction où **cron** est une expression cron d'ordonnancement.
Ce qui donne:

```
package job;

import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

@Component("theJob")
public class TheJob {

 @Scheduled(cron = "3/10 * * * * ?")
 public void runAction() {
  System.out.println("Le Job c'est moi");
 }
}
```

II-D-3-d - La configuration de l'ordonnancement sous Spring

Modifier le fichier **applicationContext.xml** en remplaçant la balise **task:scheduled-tasks** par la balise **task:annotation-driven**.
Par ce biais, l'ordonnancement de notre job est configuré via les annotations.
Ce qui donne:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:context="http://www.springframework.org/schema/context"
 xmlns:task="http://www.springframework.org/schema/task"
 xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/
beans/spring-beans-3.0.xsd
  http://www.springframework.org/schema/task http://www.springframework.org/schema/task/spring-
task-3.0.xsd
  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-
context-3.0.xsd">

 <context:component-scan base-package="job"/>
 <task:annotation-driven/>

</beans>
```

II-D-3-e - Lancement de notre job

Même chose que précédemment.

```
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App {

 public static void main(String[] args) {
  new ClassPathXmlApplicationContext("applicationContext.xml");
 }
}
```

II-E - Pour ceux qui souhaitent ne pas utiliser Spring

API Quartz	Spring Quartz Framework
Job	QuartzJobBean
JobDetail	JobDetailBean
CronTrigger	CronTriggerBean
SimpleTrigger	SimpleTriggerBean
SchedulerFactory	SchedulerFactoryBean

II-F - Conclusion

Ce tutoriel est perfectible, n'hésitez pas à remonter vos remarques. Enfin, j'espère qu'il vous aura aidé.