

---

# Compléments à "Spring par la pratique, 2nde édition"

Arnaud Cogoluègues, Zenika

Contenu mis à disposition par Arnaud Cogoluègues sous un [contrat Creative Commons](#).

## Résumé

Compléments à la deuxième édition du livre ["Spring par la pratique"](#).

## 1. Conteneur léger

### 1.1. Standards autour de l'injection de dépendances en Java

Il existe deux standards traitant de l'injection de dépendances en Java : la [JSR-330 "Dependency Injection for Java"](#) et la [JSR-299 "Java Contexts and Dependency Injection"](#) (JCDI, anciennement Web Beans). Malgré une similitude dans le nom, ces deux JSR ne sont pas vraiment équivalentes ou concurrentes : elles ont des portées différentes.

La JSR-303 définit une API très simple pour faire de l'injection de dépendances. Elle se rapproche dans son fonctionnement de [Guice](#), dont la version 2 est l'implémentation de référence. Elle est entièrement basé sur l'utilisation d'annotations. De par sa simplicité, il s'agit typiquement d'une API pouvant être implémentée par un conteneur léger et donc fonctionnant sous J2SE. Spring 3.0 implémente cette JSR.

La JSR-299 est quant à elle beaucoup plus ambitieuse puisqu'elle vise à définir les règles de cohabitation des différents composants disponibles dans JEE 6. L'idée est de pouvoir par exemple annoter des backing-beans JSF afin qu'ils puissent se voir injecter des EJB. L'injection de dépendances étant basée elle aussi sur des annotations, elle propose des moyens avancés pour affiner les injections (pas de confusion de type par exemple) et gérer des objets de contexte différents (ex. : injecter un objet de portée session dans un singleton). Une autre partie importante de cette JSR est la définition d'un modèle de notification événementielle, afin de faciliter le découplage entre composants. Spring 3.0 n'implémente pas cette JSR.

### 1.2. Support pour JSR-330 (Dependency Injection for Java)

L'injection de dépendances en Java est standardisée via la JSR-330, qui définit une API simple (5 annotations, 1 interface) mais puissante et extensible. Même si la JSR n'apporte rien qui n'existait déjà dans Spring (sous la forme d'API propriétaires), Spring 3.0 supporte complètement cette spécification.

#### 1.2.1. Autowiring avec @Inject

L'annotation @Inject indique d'injecter une dépendance, de la même manière que l'annotation Spring @Autowired :

```
package splp.v2.task;

import javax.inject.Inject;

public class MyService {
```

```

@Inject
private MyDao myDao;

public MyDao getMyDao() {
    return myDao;
}
}

```

Contrairement à `@Autowired`, `@Inject` ne dispose pas d'un attribut `required`. Elle indique donc systématiquement une dépendance obligatoire. A noter que `@Inject` est aussi reconnue pour l'injection dans les tests unitaires :

```

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class Jsr330Test {

    @Inject
    private MyService myService;

    @Test public void inject() {
        Assert.assertNotNull(myService.getMyDao());
    }
}

```

`@Inject` étant une annotation standard, elle est donc à préférer à `@Autowired`.

### 1.2.2. Affiner l'injection automatique avec `@Qualifier`

La JSR-330 introduit l'annotation `@javax.inject.Qualifier` qui permet d'affiner l'injection automatique, notamment quand il y a des conflits de types. Reprenons l'exemple du livre (page 47). Un service d'alerte peut envoyer des messages de deux façons différentes, selon le type d'abonnement du client :

```

public class AlerteClientManager {

    @Inject
    private MessageSender senderPourAbonnementGold;

    @Inject
    private MessageSender senderPourAbonnementStandard;

    (...)
}

```

Il y a deux implémentations de `MessageSender` : `SmsMessageSender` (pour les abonnements gold) et `EmailMessageSender` (pour les abonnements standards). Comme ces deux implémentations sont cachées derrière leur interface pour l'injection automatique, Spring n'est pas capable de savoir lequel injecter dans la bonne propriété. Il existe plusieurs façons de remédier à ce problème et elles sont toutes bâties sur l'utilisation de l'annotation `@Qualifier` de la JSR-330 (et non pas celle de Spring, qui marche d'une façon différente !).

La première solution consiste à définir notre propre annotation `@Sender`, qui permettra de qualifier les dépendances :

```

package splp.v2.jsr330;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)

```

```
@Qualifier
public @interface Sender {
    String value();
}
```

Remarquez que notre annotation porte elle-même l'annotation `@Qualifier`. On peut ensuite qualifier les dépendances avec :

```
package splp.v2.jsr330;

import javax.inject.Inject;

public class AlerteClientManager {

    @Inject
    @Sender("goldMessageSender")
    private MessageSender senderPourAbonnementGold;

    @Inject
    @Sender("standardMessageSender")
    private MessageSender senderPourAbonnementStandard;

    (...)

}
```

Dans la déclaration XML des `MessageSenders`, les deux beans doivent porter le nom de leur qualification :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>

    <bean id="alerteClientManager" class="splp.v2.jsr330.AlerteClientManager" />

    <bean id="standardMessageSender" class="splp.v2.jsr330.EmailMessageSender" />

    <bean id="goldMessageSender" class="splp.v2.jsr330.SmsMessageSender" />

</beans>
```

La deuxième solution utilise aussi l'annotation `@Sender` et est 100% annotation : les beans ne sont plus déclarés dans un fichier XML, on utilise le component scanning pour les instancier et les `MessageSenders` se voient apposer l'annotation `Sender` directement sur leur classe. Le fichier de contexte Spring :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>

    <context:component-scan base-package="splp.v2.jsr330" />

</beans>
```

Et les deux implémentations de `MessageSender` :

```
@Sender("standardMessageSender")
@Component
public class EmailMessageSender implements MessageSender {

    (...)

}

@Sender("goldMessageSender")
@Component
public class SmsMessageSender implements MessageSender {

    (...)

}
```

La 3ème et dernière solution est aussi 100% annotation et ne repose pas sur des chaînes de caractères pour identifier les dépendances. Elle n'utilise pas l'annotation `@Sender` mais se base plutôt sur deux annotations, une pour chaque type de `MessageSender` :

```
import javax.inject.Qualifier;

@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface SenderStandard { }

import javax.inject.Qualifier;

@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface SenderGold { }
```

Les deux implémentations de `MessageSender` se voient apposer l'annotation correspondante :

```
@SenderStandard
@Component
public class EmailMessageSender implements MessageSender {
    (...)
}

@SenderGold
@Component
public class SmsMessageSender implements MessageSender {
    (...)
}
```

Enfin, les dépendances sont qualifiées de la même façon sur le service d'alerte :

```
@Component
public class AlerteClientManager {

    @Inject
    @SenderGold
    private MessageSender senderPourAbonnementGold;

    @Inject
    @SenderStandard
    private MessageSender senderPourAbonnementStandard;

    (...)
}
```

Remarquez que cette dernière solution est la plus sûre en termes de typage (on ne se fie pas juste à des chaînes de caractères et cela évite de se tromper à cause de fautes de frappes).

## 1.3. Configuration avec des classes Java

Spring est connu pour sa configuration XML mais il est possible depuis la version 3.0 d'effectuer une configuration à base de classes Java. Le contenu du contexte Spring n'est plus alors décrit par du XML, mais par des classes Java et généralement complété par des annotations (pour l'autowiring par exemple).

### 1.3.1. Méta-données en Java

Les méta-données en Java se basent sur deux annotations :

- `@Configuration` : à poser sur une classe, pour indiquer qu'elle fournit des méta-données ;
- `@Bean` : à poser sur une méthode d'une classe `@Configuration` afin d'indiquer qu'elle produit un bean.

Voici un exemple de configuration Java produisant un service métier et un DAO :

```
package splp.v2.annotconfig.conf;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import splp.v2.annotconfig.app.dao.MyDao;
import splp.v2.annotconfig.app.dao.impl.MyDaoImpl;
import splp.v2.annotconfig.app.service.MyService;
import splp.v2.annotconfig.app.service.impl.MyServiceImpl;

@Configuration
public class BusinessConfiguration {

    @Bean
    public MyService myService() {
        return new MyServiceImpl();
    }

    @Bean
    public MyDao myDao() {
        return new MyDaoImpl();
    }

}
```

C'est donc ce code qui va être chargé de créer les beans. Pour faire un parallèle avec une configuration XML, la classe `BusinessConfiguration` serait le fichier XML et chaque méthode annotée avec `@Bean` une balise `bean`.

Spring honore aussi les annotations apposées sur les Beans produits, par exemple pour les méthodes d'initialisation ou encore l'autowiring. Ainsi, le service précédent peut se voir injecter le DAO grâce à l'annotation `@Inject` :

```
public class MyServiceImpl implements MyService {

    @Inject
    private MyDao myDao;

    (...)

}
```

Voyons maintenant comment exploiter les méta-données Java dans une application autonome puis dans une application Web.

### 1.3.2. Utilisation de `AnnotationConfigApplicationContext`

La classe `AnnotationConfigApplicationContext` permet de créer un contexte Spring à partir d'une configuration Java (c'est-à-dire une ou plusieurs classes annotées avec `@Configuration`). Voici le démarrage d'un contexte Spring dans un test unitaire avec la configuration Java précédente :

```
package splp.v2.annotconfig;
(...)
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class AnnotConfigTest {

    @Test public void createAnnotContextWithClass() {
        ApplicationContext ctx = new AnnotationConfigApplicationContext(BusinessConfiguration.class);
        MyServiceImpl service = ctx.getBean(MyServiceImpl.class);
        Assert.assertNotNull(service.getMyDao());
    }

}
```

```
}
}
```

L'`AnnotationConfigApplicationContext` est aussi capable de scanner des répertoires pour trouver les classes annotées avec `@Configuration` :

```
package splp.v2.annotconfig;
(...)
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class AnnotConfigTest {
    @Test public void createAnnotContextWithComponentScanning() {
        ApplicationContext ctx = new AnnotationConfigApplicationContext("splp.v2.annotconfig.conf");

        MyServiceImpl service = ctx.getBean(MyServiceImpl.class);
        Assert.assertNotNull(service.getMyDao());
    }
}
```

### 1.3.3. Utilisation dans une application Web

La classe `AnnotationConfigWebApplicationContext` permet de créer un contexte Spring pour le Web soit pour l'application Web (via le `ContextLoaderListener`), soit pour une Servlet Spring. Il faut pour cela préciser quelques paramètres supplémentaires dans le `web.xml` : la classe du contexte et les configurations Java à utiliser. Voyons pour le contexte Spring d'une application Web :

```
<web-app (...)>

  <context-param>
    <param-name>contextClass</param-name>
    <param-value>
      org.springframework.web.context.support.AnnotationConfigWebApplicationContext
    </param-value>
  </context-param>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>splp.v2.annotconfig.conf.BusinessConfiguration</param-value>
  </context-param>

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

Il est aussi possible de préciser des packages qui seront parcourus pour trouver des classes annotées avec `@Configuration` :

```
<web-app (...)>

  (...)

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>splp.v2.annotconfig.conf</param-value>
  </context-param>

  (...)
</web-app>
```

### 1.3.4. Préciser des portées avec la configuration Java

La configuration Java permet aussi de gérer les portées ("scope"). Reprenons l'exemple du livre (page 67), qui consiste à injecter un bean de portée session (contenant les informations utilisateur) dans un singleton. Voici la

configuration XML :

```
<bean id="businessService" class="splp.sample.BusinessServiceImpl">
  <property name="infoUtilisateur" ref="infoUtilisateur" />
</bean>

<bean id="infoUtilisateur" class="splp.sample.InfoUtilisateurImpl" scope="session">
  <aop:scoped-proxy proxy-target-class="false"/>
</bean>
```

Il est possible de faire l'équivalent en configuration Java en ajoutant une annotation `@Scope` sur la méthode `@Bean` correspondant à la création de l'objet `InfoUtilisateur` :

```
package splp.v2.chap03.scope.javaconfig;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Scope;
import org.springframework.context.annotation.ScopedProxyMode;

import splp.sample.BusinessService;
import splp.sample.BusinessServiceImpl;
import splp.sample.InfoUtilisateur;
import splp.sample.InfoUtilisateurImpl;

@Configuration
public class BusinessConfiguration {

    @Bean @Scope(value="session", proxyMode=ScopedProxyMode.INTERFACES)
    public InfoUtilisateur infoUtilisateur() {
        return new InfoUtilisateurImpl();
    }

    @Bean
    public BusinessService businessService() {
        BusinessServiceImpl service = new BusinessServiceImpl();
        service.setInfoUtilisateur(infoUtilisateur());
        return service;
    }
}
```

## 2. Spring MVC

### 2.1. Support pour la JSR-303 (Bean Validation)

La JSR-303 définit un système de validation déclaratif, basé sur des annotations. L'idée est d'annoter les propriétés de classes Java avec des contraintes (telles que `@NotNull`, `@Length`, etc.) puis de faire passer des instances de ces classes dans un validateur. Spring 3.0 supporte la JSR-303 dans le sens où il propose une abstraction pour effectuer des validations de façon programmatique (i.e. appeler directement un validateur), mais aussi de façon automatique dans Spring MVC (i.e. les objets liés à des formulaires de Spring MVC peuvent passer directement dans un validateur JSR-303). C'est cette intégration que nous allons étudier.

La première chose à faire pour bénéficier du support JSR-303 est d'ajouter une implémentation dans le class path, par exemple Hibernate Validator. La deuxième chose est d'activer le support de Spring MVC, en utilisant par exemple le namespace `mvc` :

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

```

    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

    <mvc:annotation-driven />

</beans>

```

A noter que `mvc:annotation-driven` active aussi le support pour les annotations Spring MVC (`@RequestMapping`, `@ModelAttribute`, etc.).

Prenons une entité simple, destinée à être éditée dans un formulaire HTML et ajoutons-lui des annotations de validation :

```

package splp.v2.mvc.app.domaine;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class Personne {

    @NotNull @Size(min=1,max=50)
    private String nom;

    @NotNull @Size(min=1,max=50)
    private String prenom;

    (...)
}

```

Voici un contrôleur Web envoyant vers l'affichage d'un formulaire d'édition d'une `Personne` (remarquez l'initialisation de l'objet avec la méthode annotée `@ModelAttribute`) :

```

package splp.v2.mvc.app.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import splp.v2.mvc.app.domaine.Personne;

@Controller
public class MainController {

    @RequestMapping(value="/main.do", method=RequestMethod.GET)
    public void main() { }

    @ModelAttribute("personne")
    public Personne init() {
        // initialisation de l'objet à éditer
        Personne personne = (...)
        return personne;
    }
}

```

Voici le formulaire correspondant :

```

<%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>
<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<form:form modelAttribute="personne">
  <p><form:errors path="*" /></p>
  <p>Nom : <form:input path="nom" /></p>
  <p>Prénom : <form:input path="prenom" /></p>
  <p><input type="submit" /></p>
</form:form>

```

Et voici la méthode du contrôleur appelée lors de la soumission du formulaire :

```

package splp.v2.mvc.app.web;

import javax.validation.Valid;

import org.springframework.validation.BindingResult;
import org.springframework.web.servlet.ModelAndView;

(...)

@Controller
public class MainController {

    (...)

    @RequestMapping(value="/main.do", method=RequestMethod.POST)
    public ModelAndView submit(@Valid Personne personne, BindingResult bindingResult) {
        if(bindingResult.hasErrors()) {
            // gestion des erreurs de validation
            (...)
            return new ModelAndView("main", "personne", personne);
        } else {
            return new ModelAndView("confirm", "personne", personne);
        }
    }
}

```

Spring MVC lance la validation grâce à l'annotation `@Valid` (issue de la JSR-303) apposée sur le paramètre `Personne` passée à la méthode. Les propriétés de l'objet sont directement bindés avec les valeurs du formulaire et l'objet donc validé par le framework. Les éventuelles erreurs de binding et/ou de validation peuvent être exploitées via l'objet `BindingResult` passé à la méthode.

L'utilisation de la JSR-303 facilite la ré-utilisation des règles de validation, puisque celles-ci sont directement mises sur la classe concernée. Spring MVC peut automatiquement lancer la validation, comme nous venons de le voir, mais il est aussi possible de la lancer programmatiquement ou encore de laisser un outil d'ORM (Hibernate, JPA) la lancer avant la persistance de l'entité.

### 3. Bases de données embarquées

Depuis la version 3.0.0.M4, Spring propose un espace de nom `jdbc` pour définir des beans `DataSources` issues de bases de données de données embarquées (HSQLDB, H2 et Derby). Ce support est particulièrement utile dans le cas de tests unitaires ou quand une application a besoin d'un espace de stockage (non-permanent) lors de son fonctionnement et que l'on veut réduire sa configuration au minimum.

Voici l'entête à déclarer dans un fichier de configuration 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"
        xmlns:jdbc="http://www.springframework.org/schema/jdbc"
        xsi:schemaLocation="http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc-3.0.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

</beans>

```

Pour déclarer une `DataSource` dans un contexte Spring, il suffit d'utiliser la balise `embedded-database` :

```
<jdbc:embedded-database id="dataSource" />
```

Le code précédent définit un bean `dataSource` de type `javax.sql.DataSource`. Spring est capable d'exécuter des scripts dès l'initialisation du `DataSource` afin par exemple de créer des tables et d'y insérer des données. Cela se fait avec la balise `script` :

```
<jdbc:embedded-database id="dataSource">
  <jdbc:script location="classpath:/splp/v2/jdbc/create_table.sql" />
  <jdbc:script location="classpath:/splp/v2/jdbc/insert_data.sql" />
</jdbc:embedded-database>
```

Par défaut, Spring utilise la base de données Java [HSQLDB](#). Il supporte aussi deux autres bases de données : [H2](#) et [Derby](#). Le choix se fait avec l'attribut `type` qui accepte les valeurs `HSQL`, `H2` ou `DERBY` :

```
<jdbc:embedded-database id="dataSourceHsql" type="HSQL" />
<jdbc:embedded-database id="dataSourceH2" type="H2" />
<jdbc:embedded-database id="dataSourceDerby" type="DERBY" />
```

Voici quelques remarques à propos des bases de données embarquées créées par Spring :

- Selon la base de données utilisées, il faut ajouter le binaire (JAR) correspondant dans le class path.
- Pour HSQLDB et H2, Spring crée des instances en mémoire, tandis que pour Derby, le système de fichiers est utilisé.
- La base de données s'appelle par défaut `testdb`.

L'API des bases de données embarquées de Spring est aussi directement utilisable, avec la classe `EmbeddedDatabaseBuilder` et l'interface `EmbeddedDatabase`. `EmbeddedDatabaseBuilder` propose une API "coulante" pour configurer et créer la base de données embarquée. Comme `EmbeddedDatabase` étend `DataSource`, un objet de ce type peut être injecté à tout objet qui attend une `DataSource`. Le code suivant illustre l'utilisation de l'API :

```
EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
EmbeddedDatabase db = builder
    .setType(EmbeddedDatabaseType.H2)
    .script("classpath:/splp/v2/jdbc/create_table_hsql.sql")
    .build();
JdbcTemplate tpl = new JdbcTemplate(db);
tpl.queryForInt("select count(1) from property");
db.shutdown();
```

## 4. Scheduling

Depuis la version 3.0.0.M4, Spring propose un nouvel espace de nom destiné à faciliter la configuration de tâches planifiées. Voici l'entête à déclarer dans un fichier de configuration 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"
  xmlns:task="http://www.springframework.org/schema/task"
  xsi:schemaLocation="http://www.springframework.org/schema/task
    http://www.springframework.org/schema/task/spring-task-3.0.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

</beans>
```

L'espace de nom propose la balise `scheduler` afin de définir un `ThreadPoolTaskExecutor` basé sur des utilitaires disponibles depuis Java 5 (`ThreadPoolExecutor`). Une fois le `scheduler` déclaré, il est possible de lui assigner des tâches planifiées avec les balises `scheduled-tasks` et `scheduled`. Ces tâches sont des POJO, déclaré en tant que beans Spring. Voici un exemple de configuration :

```
<task:scheduler id="taskScheduler" />

<task:scheduled-tasks scheduler="taskScheduler">
  <task:scheduled ref="task1" method="execute" cron="*/1 * * * * " />
  <task:scheduled ref="task2" method="execute" fixed-delay="2000" />
  <task:scheduled ref="task3" method="execute" fixed-rate="3000" />
</task:scheduled-tasks>

<!-- déclaration des beans Task -->
(...)
```

Les attributs `ref` et `method` de la balise `scheduled` font référence au bean de la tâche et à la méthode à appeler respectivement. Voici un exemple de tâche :

```
package splp.v2.task;

public class MyTask {

    public void execute() {
        // traitement
    }
}
```

Le scheduler est capable de gérer l'ordonnancement des tâches deux trois façons :

- avec l'attribut `cron` qui nécessite une expression cron.
- avec l'attribut `fixed-delay` qui nécessite une durée (en ms). Cette durée correspond au temps d'attente entre deux exécutions de la tâche, à compter de la fin de l'exécution de la précédente instance. Cela signifie qu'il ne peut y avoir qu'une seule exécution de la tâche en même temps.
- avec l'attribut `fixed-rate` qui nécessite une durée (en ms). Cette durée correspond au temps d'attente entre deux exécutions de la tâche.

Par défaut, le scheduler a un seul thread dans son pool. Il fait croître la taille pool selon les besoins et tue les threads au bout d'un certain d'inactivité (60 secondes). Il est possible de régler la taille du pool de thread avec l'attribut `pool-size` de la balise `scheduler` :

```
<task:scheduler id="taskScheduler" pool-size="10" />
```