1.7 Les beans Spring

Relu

Relecture RB faite le 16/02/2011

Ce chapitre n'a pas la prétention d'être une formation à *Spring*. Pour plus d'informations, vous pouvez vous reporter à la documentation de springsource (HTML / PDF). Ne sont abordés ici que quelques détails permettant de mieux comprendre certains éléments des fichiers de configuration de *esup-commons*.

Esup-commons V2 utilise Spring 3

Tout au long de ce chapitre nous allons nous appuyer sur un exemple (configuration du gestionnaire d'exceptions) :

```
<bean id="exceptionServiceFactory"
      class="org.esupportail.commons.services.exceptionHandling.CachingEmailExceptionServiceFactoryImpl"
      parent="abstractApplicationAwareBean">
   <property name="smtpService" ref="smtpService" />
   <property name="recipientEmail" value="${exceptionHandling.email}" />
   <property name="exceptionViews">
      <map>
         <entry key="java.lang.Exception" value="go_exception" />
      </map>
   </property>
   <property name="logLevel" value="${exceptionHandling.logLevel}" />
   <property name="cacheManager" ref="cacheManager" />
   <property name="cacheName" value="" />
</bean>
```

Sommaire :

Les fichiers de configuration

Spring permet de créer des objets (appelés alors *beans*) en les déclarant dans un fichier de configuration *XML*.

Le fichier de configuration principal (**properties/applicationContext.xml**) est déclaré dans le *web.xml* sous la forme d'un paramètre de l'application :

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:/properties/applicationContext.xml</param-value>
</context-param>
```

Dans *esup-commons* ce fichier de configuration principal contient seulement des inclusions de fichiers de configuration spécialisés par domaine, comme par exemple:

```
<import resource="exceptionHandling/exceptionHandling.xml" />
```

Il est possible, suivant les besoins de votre application, de supprimer ou d'ajouter des fichiers de configuration spécialisés.

L'injection de données

Une des caractéristiques de base de *Spring* est de permettre l'injection de données.

> ⚠ Cette fonctionnalité est aussi possible avec *JSF*, qui est également utilisé dans *esup-commons*, mais *JSF* est moins puissant que *Spring* dans ce domaine. *esup-commons* n'utilise donc pas l'injection de données de *JSF*.

L'injection de données permet de renseigner des attributs d'un *bean* via un fichier de configuration. Le *bean* doit disposer d'un *setter* pour l'accès à ces attributs.

Voyons quelques exemples...

Injection d'une chaîne de caractères

```
<property name="recipientEmail" value="webmaster@domain.edu"/>
```

Dans ce cas, la méthode **setRecipentEmail()** sera appelée avec pour paramètre, la valeur **webmaster@domain.edu**.

Injection d'un autre bean

```
<property name="authenticationService" ref="authenticationService"/>
```

On voit ici un autre aspect important de *Spring* qui est l'utilisation quasi systématique des interfaces. La classe **CachingEmailExceptionServiceFactoryImpl** (qui correspond au *bean* **exceptionServiceFactory** et contient la définition de cette propriété **authenticationService**) a un attribut **authenticationService** de type **AuthenticationService**. **AuthenticationService** est une interface. Le *bean* **authenticationService** doit donc être d'une classe qui implémente cette interface. Ceci permet d'avoir plusieurs implémentations possibles pour cette interface et de choisir, simplement en modifiant un fichier de configuration, laquelle on utilise. Cette approche est particulièrement intéressante : elle permet, par exemple, de très facilement tester une couche de l'application en branchant des implémentations de tests des autres couches avec lesquelles le *bean* doit interagir.

Injection d'une map

```
<property name="exceptionViews">
  <map>
    <entry key="java.lang.Exception" value="go_exception" />
  </map>
</property>
```

Injection d'une liste

```
<property name="servers">
  <list>
    <ref bean="smtpServer1" />
    <ref bean="smtpServer2" />
  </list>
</property>
```

Utilisation de paramètres

Afin de centraliser la configuration, une bonne pratique consiste à utiliser un fichier de configuration regroupant les paramètres de l'application. Ceci évite notamment de devoir modifier n fichiers différents.

Exemple d'utilisation :

```
<property name="recipientEmail" value="${exceptionHandling.email}" />
```

Ici la propriété *recipientEmail* contiendra la valeur contenue dans le paramètre *exceptionHandling.email*.

Le paramètre *exceptionHandling.email* est défini dans le fichier **default.properties** et peut être surchargé dans le fichier **config.properties** :

```
exceptionHandling.email=bugs@domain.edu
```


Ce mécanisme est rendu possible par la définition d'un bean utilisant la classe *PropertyPlaceholderConfigurer* de Spring dans le fichier **properties/applicationContext.xml**

```
<bean id="propertyConfigurer" class="org.springframework.beans.factory.config.
PropertyPlaceholderConfigurer">
  <property name="locations">
    <list>
      <value>classpath:/properties/defaults.properties</value>
      <value>classpath:/properties/config.properties</value>
      <value>file:${application.config.location}</value>
    </list>
  </property>
  <property name="ignoreResourceNotFound" value="true" />
</bean>
```

Ici on définit les propriétés dans le fichier **defaults.properties**.Elles sont éventuellement écrasées par celles définies dans **config.properties**.

Ici on utilise aussi la possibilité de surcharger ces valeurs par une référence à un fichier qui sera précisé par une option au lancement de la JVM (Ex : **-Dapplication.config.location=/tmp/foo.properties**). Afin que le propertyConfigurer ne lève pas une exception au cas où cette dernière possibilité n'est pas utilisée on positionne une des ces propriétés (**ignoreResourceNotFound**) pour qu'il ignore les éventuelles ressources absentes.

L'héritage de configuration

Spring n'oblige pas à saisir, dans toutes les définitions de *beans*, les mêmes propriétés. Pour cela, il est possible d'utiliser le mot-clé **parent**.

Le « parent » a un attribut **abstract="true"** car il ne doit pas être créé en mémoire par *Spring*. Cette notation permet de se rapprocher de l'héritage *Java* qui est beaucoup utilisé dans *esup-commons*.

Exemple d'un *bean* « parent » ayant aussi lui-même un « parent » :

```
<bean
  id="abstractApplicationAwareBean"
  parent="abstractI18nAwareBean"
  abstract="true">
    <property name="applicationService" ref="applicationService" />
</bean>
```


L'attribut **scope** (voir ci-après) n'est pas héritable, il est donc inutile de le préciser pour un *bean* abstrait.

Vérification des *beans*

Les *beans* manipulés par *Spring* n'ont pas, par défaut, de dépendances particulières avec *Spring*.

Il est, par contre, possible de sciemment introduire une dépendance avec *Spring* pour obtenir des services supplémentaires.

Faire en sorte que votre *bean* implémente l'interface **InitializingBean** de *Spring* en fait partie. Cette interface vous oblige à implémenter une méthode **afterPropertiesSet** qui sera appelée par *Spring* juste après l'initialisation du *bean*. Cette méthode vous permet de vérifier que toutes les propriétés sont bien initialisées. Si ce n'est pas le cas, vous pouvez, par exemple, lever une exception ou affecter une valeur par défaut.

On trouvera par exemple :

```
public void afterPropertiesSet() {
  super.afterPropertiesSet();
  if (!StringUtils.hasText(exceptionView)) {
    exceptionView = DEFAULT\__SERVLET_\_EXCEPTION_VIEW;
    logger.info(getClass() + ": no exceptionView set, using default \[" + exceptionView + "\]");
  }
}
```

Portée des beans

Spring offre une notion de portée (**scope**).

Par défaut, un *bean* est de portée **singleton**. *Spring* crée une seule instance de ce *bean* pour toute la durée d'exécution de l'application.

Il existe aussi des portées **session** et **request** qui, respectivement, permettent d'avoir une instance du *bean* par session utilisateur (au sens d'une application web) ou par requête (au sens HTTP). Cette notion est particulièrement intéressante pour les contrôleurs web d'une application. Ces derniers sont en général des *beans* de portée **session**, comme par exemple :

```
<bean id="administratorsController"
  class="[...].formation.web.controllers.AdministratorsController"
  parent="abstractContextAwareController"
  scope="session" />
```

Usuellement, un *bean* de *scope* **request** peut faire référence, via ses propriétés, à un *bean* de *scope* **session** ou **singleton**. De même, un *bean* de *scope* **session** peut faire référence à un *bean* de *scope* **singleton**. Une bonne architecture nous amène d'ailleurs à utiliser dans ce sens l'injection de beans. Par défaut, la réciproque provoque une exception ... mais il est cependant possible de réaliser cette réciproque par le biais de l'*aop*, et cela très simplement.

Concrètement si nous voulons ici injecter le *bean* **administratorsController** qui est de scope session dans un bean de scope singleton, on utilisera la balise *aop:scoped-proxy* comme ceci dans la déclaration du *bean* **administratorsController** :

```
<bean id="administratorsController"
  class="[...].formation.web.controllers.AdministratorsController"
  parent="abstractContextAwareController"
  scope="session">
 <aop:scoped-proxy/>
</bean>
```

Pour ce faire, on aura pris soin de déclarer comme il se doit l'espace de noms aop, avec dans la balise racine du fichier de configuration de *beans Spring* ceci :

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:aop="http://www.springframework.org/schema/aop"
       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-3.0.xsd
         http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-3.0.
xsd" >
...
</beans>
```