Chapitre 4
Le conteneur Spring

1. Introduction

Ce chapitre présente une utilisation simplifiée de Spring afin que nous ayons un aperçu de Spring sans nous perdre dans les détails. Nous verrons par la suite des éclairages sur des parties qu'il vaut mieux connaître pour bien se repérer dans une application plus complexe. L'exemple qui l'illustre permet déjà d'expérimenter une grande partie des problématiques relatives à l'utilisation de ce framework.

2. Les origines

Nous allons aborder les principaux composants de Spring.

Spring est un framework qui simplifie la programmation. Il est composé d'un cœur, Spring Core, qui permet une gestion simple des instances de classe en mémoire et de bibliothèques de classes qui utilisent ce cœur.

Celles-ci s'appellent des beans Spring. Le framework fournit un ensemble de beans préprogrammés qui couvrent un très large spectre de cas d'utilisation que l'on rencontre quand nous codons une application complexe.

Ils sont extensibles et faciles à utiliser.

Le cœur du framework permet de charger au démarrage un ensemble de singletons et facilite ensuite leur accès en injectant automatiquement leur référence (emplacement mémoire) dans les objets qui les utilisent.

Spring permet d'avoir un contrôle très fin sur la gestion des objets en mémoire.

Spring s'interface avec énormément de frameworks et de produits. Le principal intérêt de Spring est l'instanciation et la mise à disposition automatisée de beans. Ces objets peuvent être de deux types : des singletons, comme nous l'avons déjà évoqué, mais aussi des objets « dupliqués » nommés « prototypes ».

Contrairement à l'objet `Prototype`, l'objet `Singleton` est un objet partagé instancié une seule fois dans un même conteneur Spring et dont l'utilisation est partagée. Spring gère en interne une liste des singletons instanciés. Si un bean membre d'un autre bean est à injecter et qu'il est déjà chargé en mémoire, Spring copie sa référence, sinon il l'instancie et le met dans sa liste. Il l'injecte alors automatiquement. Généralement, Spring crée les singletons au démarrage du conteneur Spring lors du lancement de l'application. Il existe quelques exceptions à cette règle que nous verrons par la suite.

Un objet `Prototype` est un objet qui est instancié à chaque fois que Spring l'injecte dans le membre d'un objet Spring. Tous les mécanismes de facilitation offerts par Spring sont disponibles pour les objets `Prototype` mais l'utilisation de ce type d'objets est relativement rare.

L'intérêt du bean prototype est de bénéficier de tous les avantages de Spring pour les objets qui ne sont pas des singletons.

Nous verrons que Spring permet également de contrôler le cycle de vie (création, destruction…) et offre la possibilité d'intercepter les appels des méthodes des objets managés afin d'en prendre le contrôle via la programmation par aspect qui y est intégrée.

Nous verrons qu'il existe quatre façons de configurer les beans. Ces typologies de configuration sont interchangeables et il est possible de les mixer : la configuration peut être implicite ou explicite. Dans le premier cas, Spring découvre les beans au lancement du conteneur en parcourant les classes, dans le second cas, Spring ne prend en compte que l'emplacement et la fonction exacts des beans ainsi déclarés.

Spring facilite l'intégration de l'application avec son écosystème et l'appel aux web services standards ou Hessian, Burlap, Rmi, RPC entre autres. Spring permet aussi d'utiliser des EJB.

3. Les modules fondamentaux

Les beans spécialisés présentés ci-après sont généralement présents depuis la version 0.9 de Spring et ont reçu des améliorations au fil des versions. Ils sont surtout utilisés quand nous souhaitons étendre Spring. Vous les verrez surtout dans les frameworks. Quand nous utilisons les annotations nous ne voyons plus ces objets, mais ils sont utilisés en interne dans Spring.

3.1 Composition d'un bean

Nous verrons plus en détail le fonctionnement d'un bean dans le chapitre Programmation orientée aspect avec Spring consacré à l'AOP.

Pour simplifier, un bean peut être considéré comme un proxy qui augmente un objet Java. Le proxy permet :

- un détournement des appels aux méthodes de l'objet pour ajouter des comportements ;
- l'ajout de nouvelles méthodes ;
- une gestion des liens vers les objets référencés dans l'objet principal ;
- une possibilité de valoriser les propriétés de l'objet via de multiples façons : chaînes, fichiers (via une factory) ;
- une gestion de messages dans des bundles (via le contexte) ;

– une gestion d'événements interobjets : création, destruction, événements utilisateurs.

Le bean est orienté données (POJO) ou traitements en essayant de séparer ces aspects dans des beans spécialisés.

Il contient donc :

– la classe d'implémentation réelle du bean ;
– des éléments de configuration comportementale du bean, singleton/prototype ;
– les beans liés à l'injection des dépendances…
– des valeurs de propriété à définir à la construction.

Quand nous voulons utiliser un bean Spring, nous ajoutons un membre de classe et nous indiquons à Spring que nous voulons l'utiliser en typant la variable avec l'interface correspondant au Bean. Spring a différente stratégie pour trouver le bon Bean à injecter.

Un bean est identifié par son nom et son identifiant. Les identifiants peuvent posséder des alias, mais cette utilisation est plutôt une source de confusion. Plusieurs beans peuvent porter le même identifiant, on indiquera alors à Spring lequel est à prendre en priorité (**`@Primary`**).

3.2 Le singleton et le prototype

Un singleton, qui est le type par défaut des beans, ne possède qu'une instance d'implémentation au sein d'un contexte alors que le prototype peut en avoir plusieurs.

3.3 Les objets fondamentaux du package core

Ce package gère l'injection de dépendances. Il utilise un certain nombre de classes qu'il faut connaître.

3.3.1 Le PropertyEditor

Spring utilise les éditeurs de propriétés pour gérer la conversion entre les valeurs String et les types personnalisés Object.

Il existe une liaison automatique et une liaison personnalisée.

L'infrastructure JavaBeans standard détecte automatiquement les objets `PropertyEditor` s'ils se trouvent dans le même package que la classe qu'ils gèrent.

Ces classes `PropertyEditor` doivent porter le même nom que la classe, avec le suffixe `Editor`.

Spring les réutilise massivement pour initialiser ses beans depuis les fichiers XML. Il est aussi très utilisé par Spring MVC.

Liaison automatique

Il y a une détection automatique si la classe du bean et l'objet **`PropertyEditor`** représenté par la classe avec le suffixe **`Editor`** sont dans le même package.

Si les classes `Exemple.java` et `ExempleEditor.java` sont trouvées dans le même package :

```
@Data
@ToString
public class Exemple {
       private String chaine;
       private Integer entier;
}

public class ExempleEditor extends PropertyEditorSupport {
    @Override
    public String getAsText() {
       Exemple exemple = (Exemple) getValue();
        return exemple == null ? "" :
exemple.getChaine()+">"+exemple.getEntier();
    }

    @Override
    public void setAsText(String text) throws
```

```
                    IllegalArgumentException {
        if (StringUtils.isEmpty(text)) {
            setValue(null);
        } else {
             Exemple exemple = new Exemple();
           StringTokenizer st = new StringTokenizer(text,">");
           exemple.setChaine(st.nextToken());
           exemple.setEntier(Integer.parseInt(st.nextToken()));
            setValue(exemple);
        }
    }
}
```

Avec un contrôleur :

```
@Slf4j
@Controller
public class MonRestController {

       @GetMapping(value = "/test/{id}")
       public @ResponseBody String test1(@PathVariable String id) {
             return id;
       }

       @GetMapping(value = "/test2/{ex}")
       public @ResponseBody String test2(@PathVariable Exemple ex) {
             String ret=ex.toString();

             return ret;
       }
}
```

Sur un appel :

```
http://localhost:8080/test2/aaa%3E1
```

Nous avons :

```
Exemple(chaine=aaa, entier=1) appel :
```

Liaison personnalisée

Si le binder se trouve dans un autre package que celui de la classe alors il faut initialiser le binder dans le contrôleur :

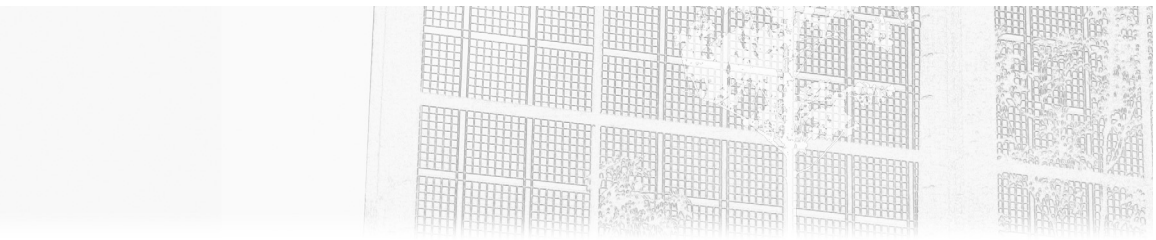
```
@Slf4j
@Controller
public class MonRestController {

	@GetMapping(value = "/test3/{ex}")
	public @ResponseBody String test3(@PathVariable Exemple2 ex) {
		String ret=ex.toString();

		return ret;
	}

	@InitBinder
	public void initBinder(WebDataBinder binder) {
	    binder.registerCustomEditor(Exemple2.class,
	        new CustomEditorEditor());
	}
}
```

3.4 Les PropertyValues

Une `PropertyValues` contient plusieurs `PropertyValue`.

La `PropertyValue` d'un objet contient les informations et la valeur d'une propriété de bean individuelle. Elle permet, entre autres, de gérer les propriétés indexées de manière optimisée. Elle est utilisée dans les convertisseurs que nous verrons au chapitre Configuration avancée. Elle permet aussi de spécifier si la valeur est à ignorer lorsque cette dernière n'existe pas dans l'objet ciblé.

Notez que la valeur n'a pas besoin d'être d'un type prédéfini car une implémentation `BeanWrapper` doit pouvoir gérer toutes les conversions nécessaires. En effet, cet objet ne connaît rien des objets auxquels il sera rattaché.



Chapitre 3
La présentation avec les JSP

1. Introduction

Les servlets ne sont pas adaptées pour gérer efficacement l'affichage comme vous avez pu le constater dans le chapitre précédent. La plateforme Jakarta EE propose une solution nommée **JSP** (*Jakarta Server Pages*). Cette technologie permet de créer facilement un contenu dynamique au format HTML ou XML. Elle correspond au V (vue) de l'architecture MVC. La servlet s'occupe de faire le traitement métier et lorsque celui-ci est terminé, elle délègue l'affichage à une JSP. La suite du chapitre prendra comme exemple un contenu HTML.

Les JSP sont, tout simplement, des pages HTML d'extension `.jsp` dans lesquelles il est possible d'ajouter différents types de contenus (non HTML) qui seront traités par le conteneur de servlets pour générer un rendu spécifique lié au contexte d'exécution de la requête. Ces types de contenus peuvent être :

- des scripts sous la forme de code Java,
- des scripts sous la forme d'EL (*Jakarta Expression Language*),
- des actions standards,
- des tags standards (JSTL - *Jakarta Standard Tag Library*),
- ou des tags personnalisés.

Le chapitre explique le principe de fonctionnement des JSP puis présente les différents types de contenus listés précédemment.

Différentes technologies sont abordées dans ce chapitre.

Tout d'abord les JSP. La version actuelle est la **3.0**. Vous pouvez consulter la spécification à l'adresse suivante :
https://jakarta.ee/specifications/pages/3.0/jakarta-server-pages-spec-3.0.pdf

Ensuite, l'EL (*Jakarta Expression Language*) est dans sa version 4.0. La spécification est consultable à l'adresse suivante :
https://jakarta.ee/specifications/expression-language/4.0/jakarta-expression-language-spec-4.0.pdf

Pour finir, la JSTL (*Jakarta Standard Tag Library*), dans sa version 2.0, dont la spécification est consultable à l'adresse suivante :
https://jakarta.ee/specifications/tags/2.0/jakarta-tags-spec-2.0.pdf

Aujourd'hui, la technologie des JSP n'est plus le standard promu par Oracle pour le développement des interfaces mais cette technologie est encore largement utilisée (et pour longtemps). Oracle préconise l'utilisation de la technologie JSF. Par ailleurs, l'avènement des services web et des frameworks JavaScript propose une autre alternative pour développer des applications. Les chapitres suivants aborderont ces autres possibilités.

2. Le projet

2.1 La création du projet

La suite du chapitre se base sur un projet exemple nommé **Projet_JSP**.

- Commencez par créer un projet de type Gradle Project avec les mêmes caractéristiques que dans le premier chapitre.
- Complétez l'arborescence de répertoires pour qu'elle ressemble à cela :

- Ajoutez un fichier nommé `web.xml` dans le répertoire `src/main/webapp/WEB-INF` avec un contenu équivalent au projet initial (*PremierProjetWeb*). Veillez simplement à renommer le nom du projet au sein de la balise `<display-name>`.
- Modifiez le fichier `build.gradle` avec le contenu suivant (observez la section `dependencies` permettant de référencer l'API des servlets et l'API des JSP) :

```
plugins {
    id 'java'
    id 'war'
}

repositories {
    jcenter()
}


dependencies {
    compileOnly "jakarta.servlet:jakarta.servlet-api:5.0.0"
compileOnly group: 'jakarta.servlet.jsp', name:
'jakarta.servlet.jsp-api', version: '3.0.0'
}
```

- Sélectionnez le menu contextuel **Gradle - Refresh Gradle Project** de votre projet pour télécharger les dépendances.

Le projet est prêt pour manipuler les JSP.

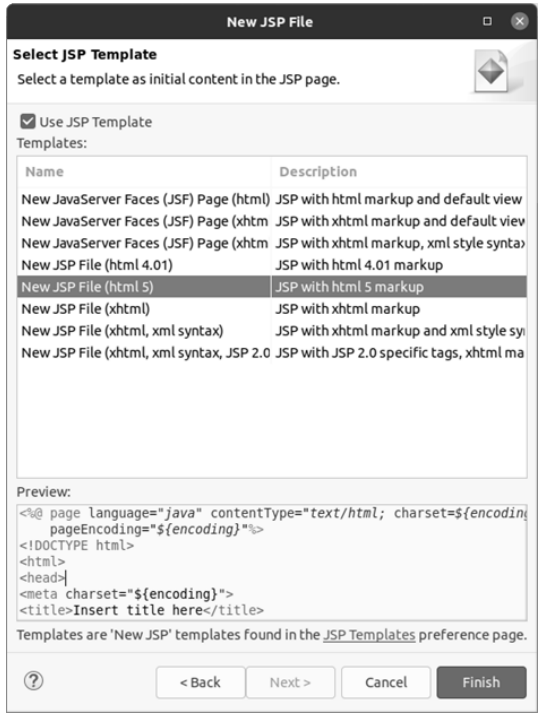
2.2 La création d'une JSP

Pour créer une JSP, suivez les étapes suivantes :

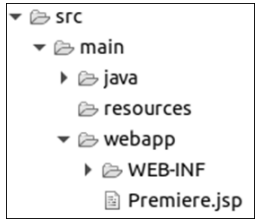
- Faites un clic droit sur le répertoire `webapp` de votre projet puis cliquez sur le menu **New** - **JSP File**.

L'écran suivant apparaît :

- Donnez un nom à votre JSP (`Premiere.jsp`) dans l'exemple et cliquez sur **Next** afin de pouvoir sélectionner le template à utiliser pour la création :



- Sélectionnez le template **New JSP File (html 5)** et cliquez sur **Finish**, votre première page JSP est créée et opérationnelle :



Le contenu de la JSP est le suivant :

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>

</body>
</html>
```

C'est une page HTML classique mis à part la première ligne. C'est ce que l'on appelle une directive. Cette ligne permet de définir certaines caractéristiques de la JSP. Pour plus d'informations, veuillez vous référer à la section Les directives.

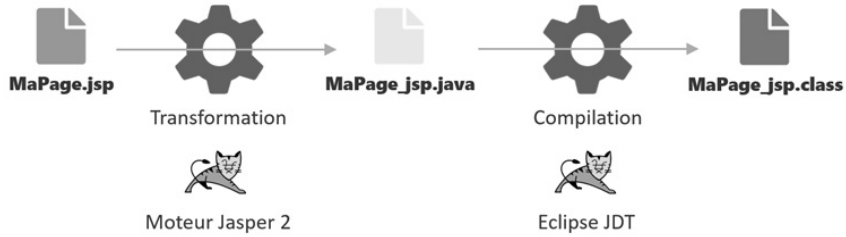
Cette page peut être appelée au travers de l'URL suivante :
http://localhost:8080/Projet_JSP/PremiereJSP.jsp

Le résultat n'est pas spectaculaire car aucun contenu dynamique n'est présent. Cependant, en affichant le code source de la page HTML sur le navigateur, vous pouvez noter l'absence de la directive ce qui indique que le fichier a été traité sur le serveur avant d'être remis au navigateur. La section suivante explique ce traitement.

3. Le principe d'exécution

Les JSP existent pour simplifier la création d'un contenu dynamique car les servlets ne sont pas adaptées pour cette tâche. Lorsqu'une requête HTTP implique l'exécution d'une JSP, voici les actions qui sont déclenchées :

– Si la JSP n'a encore jamais servi, celle-ci est transformée en classe Java (*Translation phase*) avant d'être compilée. La transformation est réalisée par le moteur **Jasper 2** et la compilation est réalisée par défaut par le compilateur Java **Eclipse JDT** (et non pas `javac`). Pour plus d'informations, veuillez vous référer à la documentation officielle à l'adresse suivante :
https://tomcat.apache.org/tomcat-10.0-doc/jasper-howto.html



Cette classe doit implémenter l'interface `jakarta.servlet.Servlet`. Dans l'environnement Tomcat, elle dérive indirectement de la classe `jakarta.servlet.http.HttpServlet`. Une JSP n'est donc ni plus ni moins qu'une servlet. La classe doit aussi implémenter l'interface `jakarta.servlet.jsp.HttpJspPage`.

Cette interface force l'écriture de la méthode `_jspService(...)`. Cette méthode est l'équivalent des méthodes `doXXX(...)` des servlets. Elle prend en paramètre un objet de type `HttpServletRequest` et un objet de type `HttpServletResponse`. Elle a pour rôle la création d'une réponse à l'utilisateur.

Ensuite, la méthode `service(...)` de cette nouvelle classe est appelée. Elle appellera la méthode `_jspService(...)`.

Le cycle de vie d'une JSP est le même que celui d'une servlet. La seule différence est le nom des méthodes pour l'initialisation et la destruction. Elles s'appellent `jspInit()` et `jspDestroy()`. Il est possible de les redéfinir dans la page JSP si nécessaire. Pour plus de détails, veuillez vous référer au chapitre traitant des servlets et à la section Le paramétrage d'une JSP un peu plus loin dans ce chapitre.

Pour bien comprendre cette étape, voici un extrait de la classe Java générée pour la JSP `premiereJSP.jsp`. La classe s'appelle `Premiere_jsp` et dérive de la classe `org.apache.jasper.runtime.HttpJspBase` qui dérive de la classe `HttpServlet` et qui implémente l'interface `HttpJspPage`.

```
/*
 * Generated by the Jasper component of Apache Tomcat
 * Version: Apache Tomcat/10.0.7
 * Generated at: 2021-09-06 19:7:26 UTC
 * Note: The last modified time of this file was set to
 *       the last modified time of the source file after
 *       generation to assist with modification tracking.
 */
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class Premiere_jsp
```