Java Spring

Introduction

http://spring.io

Exigences du projet informatique :

- Performance
- Maintenance
- Sécurité
- Portabilité

- Distribution
- Capacité à communiquer avec d'autres applications distantes
- Capacité de fournir le service à différents types de clients (desktop, mobile, sms, HTTP...)
- Coût du soft

Definition

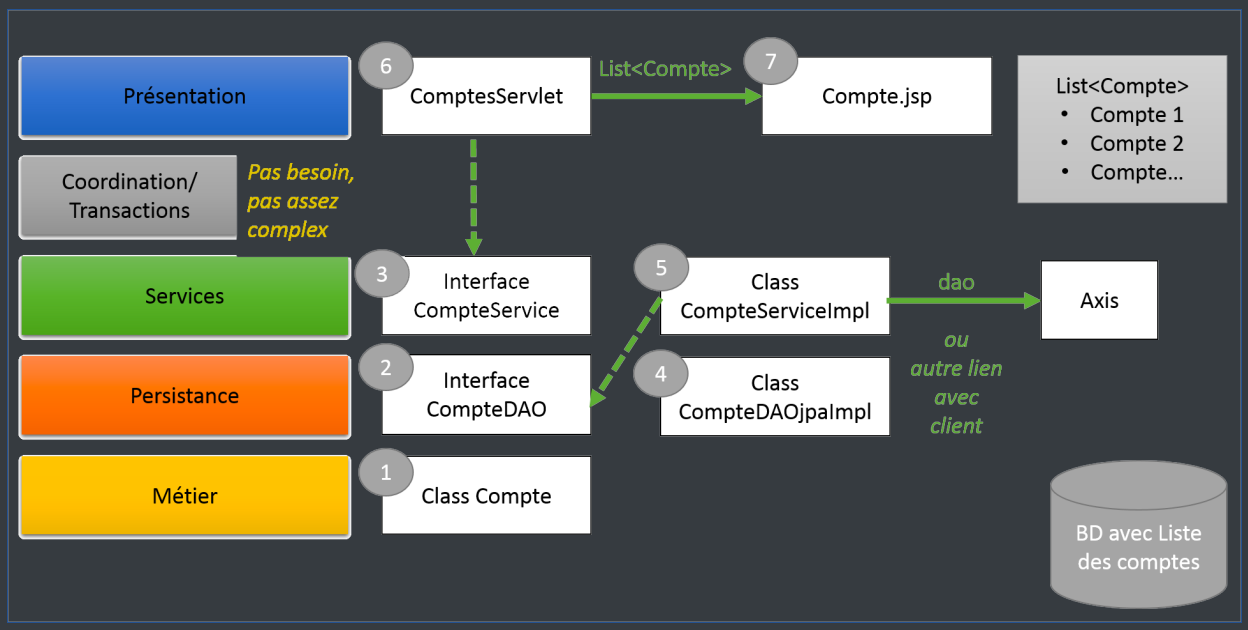
Spring : Framework pour l'inversion de contrôle. Spring est un conteneur léger.

Framework : Cadre de développement. Il contient des bonnes pratiques de développement. Il permet d'éviter de recoder des classes utilitaires. Il fournit le socle technique pour que les développeurs puissent se consentrer sur le code métier.

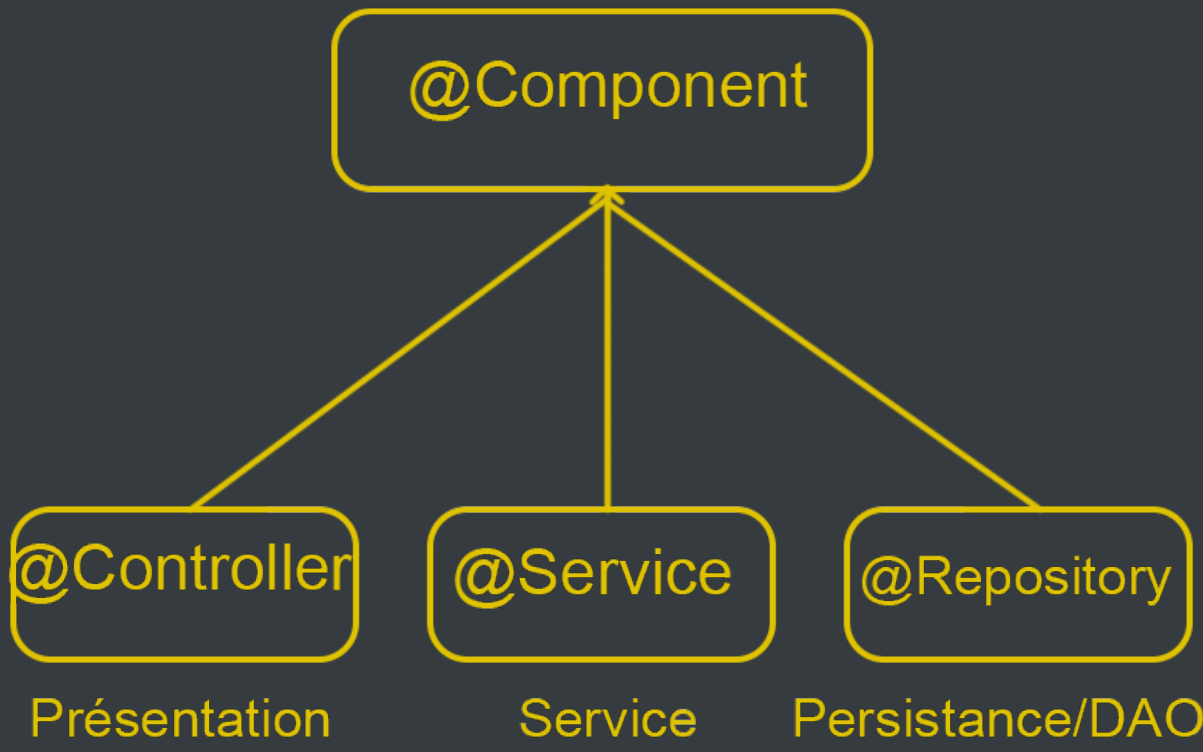
Rappels de conception

- Peu de couplage de classe
- Appli fermée à la modification mais ouverte aux changements
- Programmer interface et non une implémentation
- Appli qui n'évolue pas : meurt

Schéma de conception :



@Component, @Repository, @Service & @Controller



Couplage fort

Quand la classe A est liée à la classe B, on dit que la classe A est fortement couplée à la classe B. A ne peut fonctionner qu'en présence de B. Si une nouvelle classe B2 est créée, on est obligé de modifier dans la classe A. Le couplage fort est une mauvais pratique.

Couplage faible

Pour utiliser le couplage faible, il faut utiliser les interfaces. Classe A utilise une interface IA et la classe B inplémente une interface IB. Si A est lié à l'interface IB par une association, on dit que A et la classe B sont liés par un couplage faible. Dans ce cas, B peut fonctionner avec n'importe quelle classe qui implémente l'interface IA.

Formes d'injection des dépendances

Statique :

```java
import metier.MetierImpl;
import dao.DaoImpl
public class presentation
{
    public static void main (String args[])
    {
        DaoImpl dao = new DaoImpl();
        //^ ici c'est statique, il faudrait que ça soit dynamique
        MetierImpl metier = new Metier();
        metier.setDao(dao);
        System.out.println(metier.calcul());
    }
}
```

Dynamique : géré par spring, qui permet de passer par des fichiers de configuration .txt

---

Ingection des dépendances avec Spring

L'injection des dépendances se fait au début de l'execution de l'appli. SpringIoC lit un fichier *.xml* qui déclare quelles sont les différentes classes à instance et assure les dépendances. Ajout de classe -> modification du *.xml*.

Dans une appli java standard

```xml
<beans>
    <bean id="d" class="dao.DaoImpl2"></bean>
    <bean id="metier" class="metier.MetierImpl2">
        <property name="dao" ref="d"></property>
    </bean>
</beans>
```

Ici est équivalent à un *new*, est équivalent à un attribut.

XML et DTD

Exemple de XML

```
<smartphone>
    <ecran>taille</ecran>
    <processeur>architecture</processeur>
    <batterie>capacité</batterie>
</smartphone>
```

Le DTD décrit les donénes du XML, c'est le schéma.

Architecture Spring

Data access integration

- JDBC
- ORM
- OXM
- JMS
- Transaction

Web

- Web
- Servlet
- Porlet
- Struts

Core container

- Beans
- Core
- Context

MVC

- Implémentation du modèle mv

Résumé

Spring fournit un cadre de travail:

- Une aide pour simplifier les aspects techniques
- Des patterns d'architecture
- Il fait la "plomberie"

Les sous projets sont plus spécifiques

- Réaliser un batch, un webservice
- pas le temps de tout noter D:

Il est utilisé avec un serveur d'application léger tel que Tomcat ou Jetty.

Choix de la méthode d'injection

Setter :

- Respecte la convention
- Heritage automatique
- Plus clair que par le constructeur
- Permet d'avoir des dépendances optionelles

Injection par constructeur :

- Objets immutables
- Oblige à avoir toutes les dépendances correctement définies
- Plus consise que par setter

Injection par champs :

- Même qualités que par constructeir
- Encore plus concises
- Mais gênant pour les tests unitaires

Les objets métiers

Objets java simples:

- POJO
- Bean

Ils implémentent une interface métier, ce qui permet de découpler les objets, de simpliier l'écriture

---

Annotation ou XML?

Config métier : annotation

Config infrastructure : xml

Placeholder configuration

```
<bean ...>
    <property name="monNom" value="${jdbc.username}"></property>
</bean>
```

Démarrer Spring

```
ApplicationContext ctx = new
ClassPathXmlApplicationContext("application-context.xml");
```

Démarrer appli web

```xml
<context-param>
    <param-name>ContextConfigLocalisation</param-name>
    <param-value>classpath:META-INF/spring/application-context.xml</param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

—>Cette configuration lance uniquement Spring IoC, elle ne lance pas Spring MVC

Inner Bean

Même principe que les inner class en Java. Il est déclaré à l'intérieur qu'un auytte bean. Il est injecté dans le ce deuxième bean. Seul ce deuxième bean peut le voir. Cela clarifie la configuration. Pas de limite, on peut faire des inner beans de inner beans...

```xml
<bean id="beanA" classe="example.BeanA">
    <property name = "beanB">
        <bean class = "example.BeanB">
            <property name = "prop1" value = "jdjdjd" />
            <property name = "prop2" value = "jfjfjfjf" />
        </bean>
    </property>
</bean>
```

Class anonyme

```
jdbcTemplate.querry(
    "select id, user_id, balance from account",
    new ParametrizedRowMapper<Account>()
    {
        public Account mapRow(ResultSet rs, int i) throws
SQLException
        {
            return new Account(rs.getLong("id"),
rs.getInt("user_id"), rs.getInt("balance"));
        }
    }
);
```