

# Spring MVC

**Christophe Fontaine**

**[cfontaine@dawan.fr](mailto:cfontaine@dawan.fr)**

**02/01/2023**

# Objectifs

---



- Construire des applications Java EE robustes basées sur Spring MVC et Spring ORM
- Implémentation de services web REST

**Durée :** 5 jours

**Pré-requis:** Maîtrise de Java

Connaissance des Servlets et de JSP

# Bibliographie

- **Java Spring**  
**Le socle technique des applications Java EE**

Hervé Le Morvan

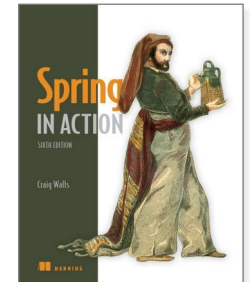
Éditions ENI - 3<sup>ème</sup> édition - janvier 2021



- **Spring in Action**

Craig Walls

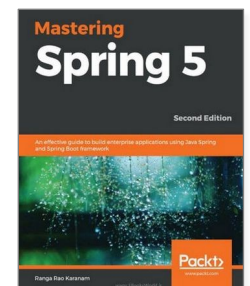
Manning - 6<sup>nd</sup> edition - Janvier 2022



- **Mastering Spring 5**

Ranga Rao Karanam

Packt Publishing - 2<sup>nd</sup> edition - Juillet 2019



- **Spring Framework Reference Documentation**

<https://docs.spring.io/spring-framework/docs/current/reference/html/>

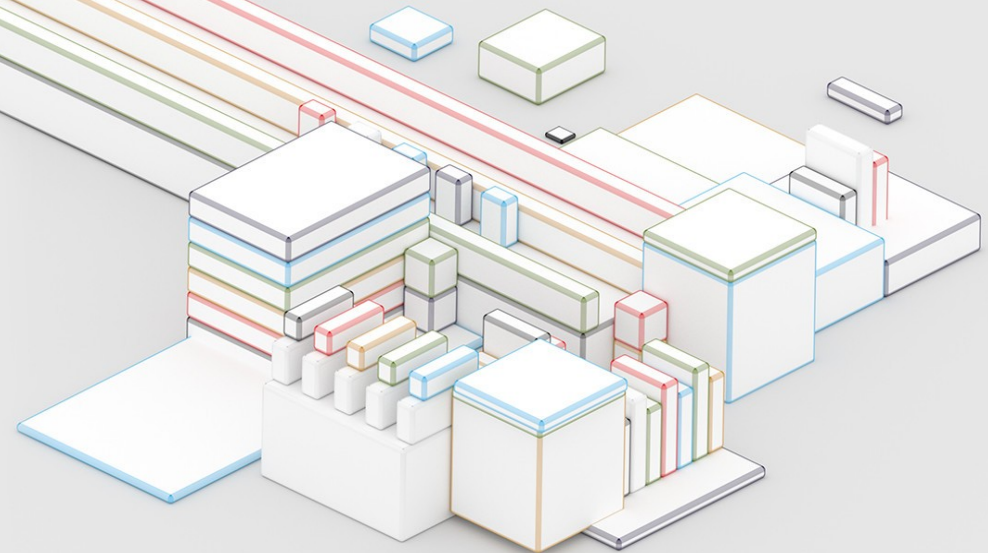
# Plan

---



- Découvrir Spring MVC
- Configurer des beans: Spring Core
- Spring Web MVC :
  - Implémenter des Contrôleurs
  - Persister des données
  - Formulaire Spring et validation
  - Templating
- Implémenter des web services REST
- Réaliser un mapping des données avec Spring ORM

# Découvrir Spring MVC

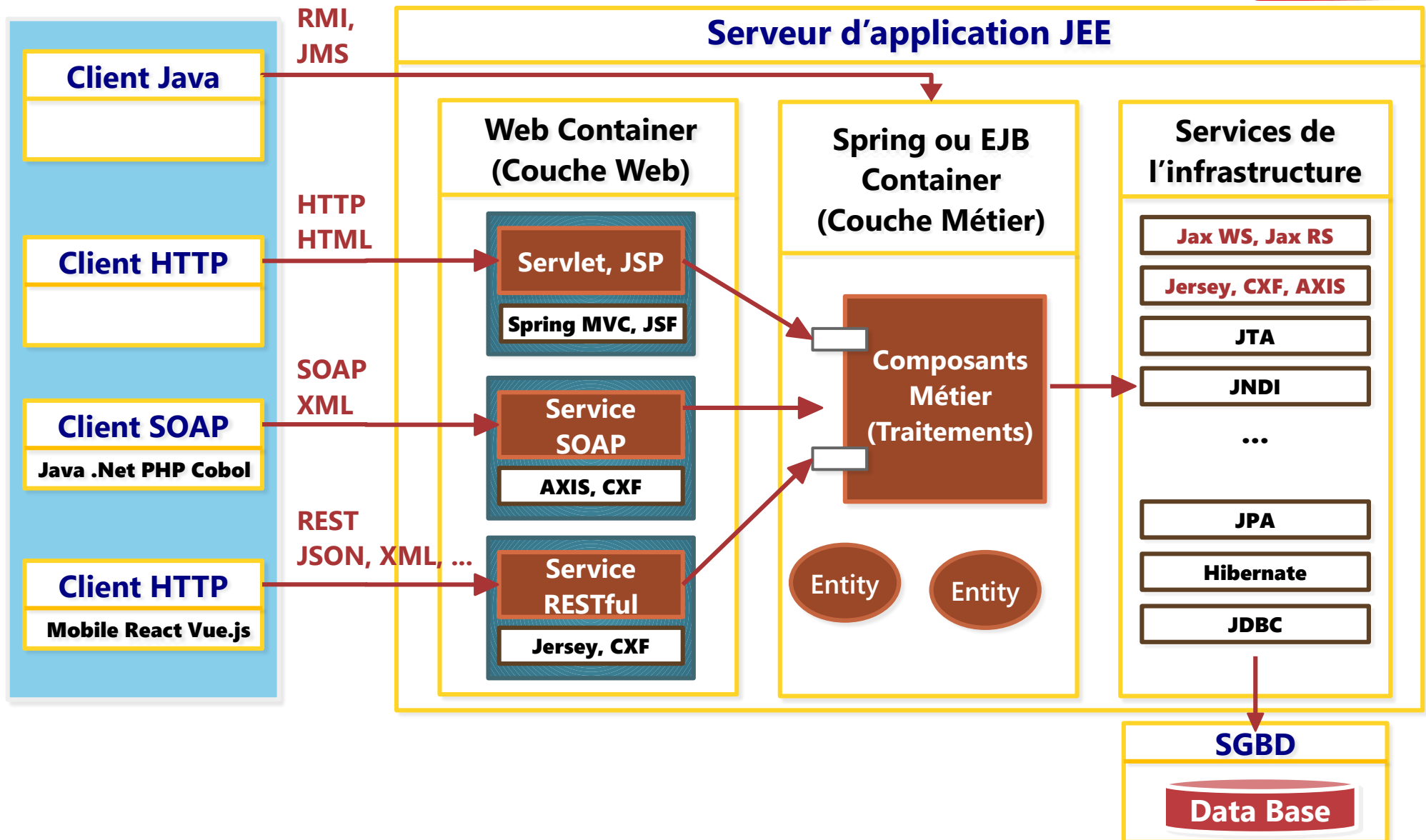


- Java EE est la version "entreprise" de Java, elle a pour but de faciliter le développement d'applications distribuées
- Java EE est **avant tout une norme** :  
Il définit ce qui doit être fourni mais ne dit pas comment cela doit être fourni
- La plateforme Java EE est composée :
  - d'un ensemble de spécification pour l'infrastructure dans laquelle s'exécute les composants → serveur d'application
  - d'un ensemble de spécification décrivant des services techniques : par exemple, comment accéder à un annuaire, à une base de données...

Exemple de services :

- JNDI (Java Naming and Directory Interface)  
est une API d'accès aux services de nommage et aux annuaires d'entreprises tels que DNS, NIS, LDAP...
- JTA (Java Transaction API)  
est une API définissant des interfaces standard avec un gestionnaire de transactions

# Architecture Java EE





# Serveurs d'applications

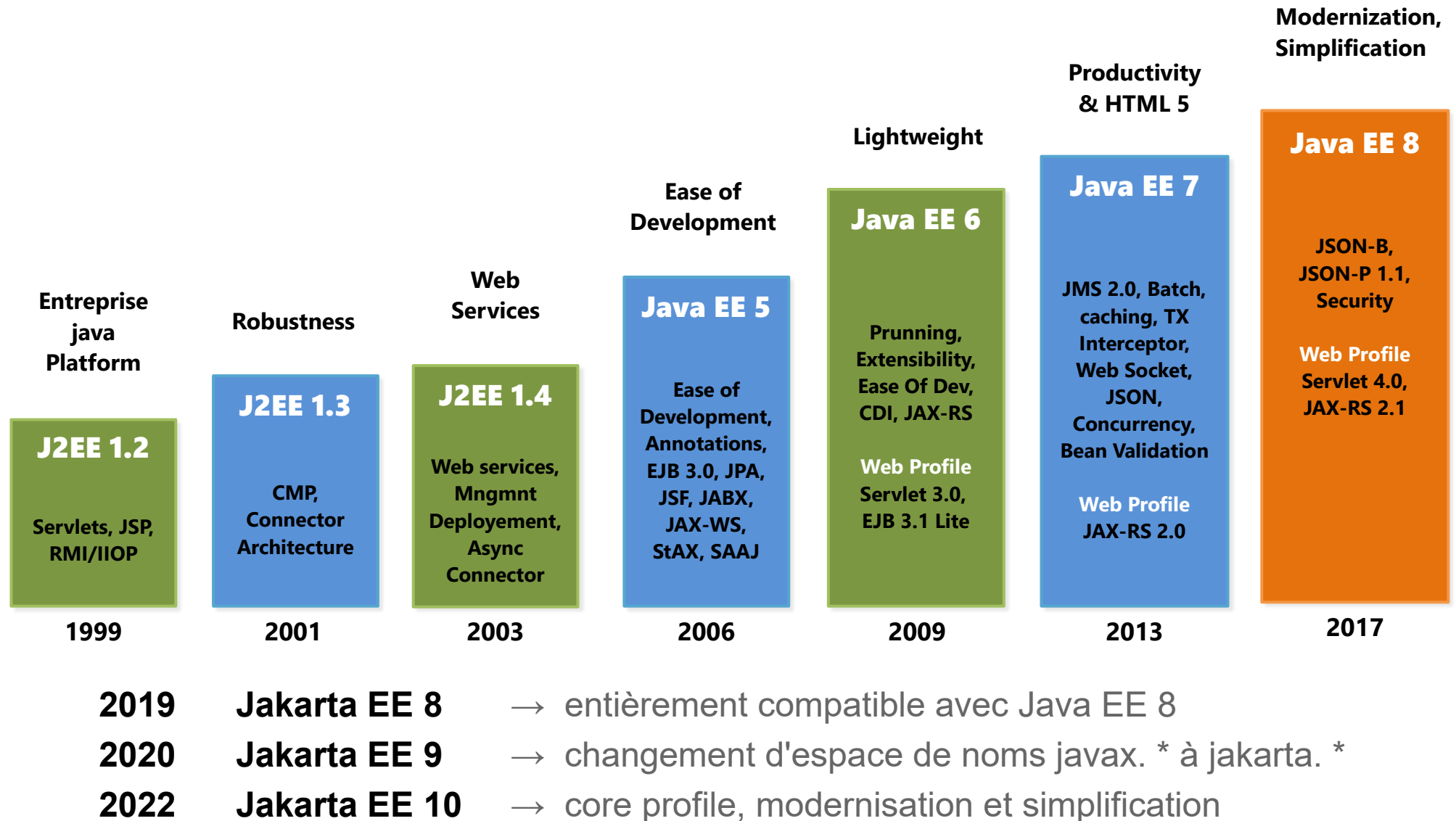
- Les applications JEE sont hébergées par des serveurs certifiés JEE : Web-Profile ou Full-Profile  
<http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html>

- **Exemple de serveur d'application**

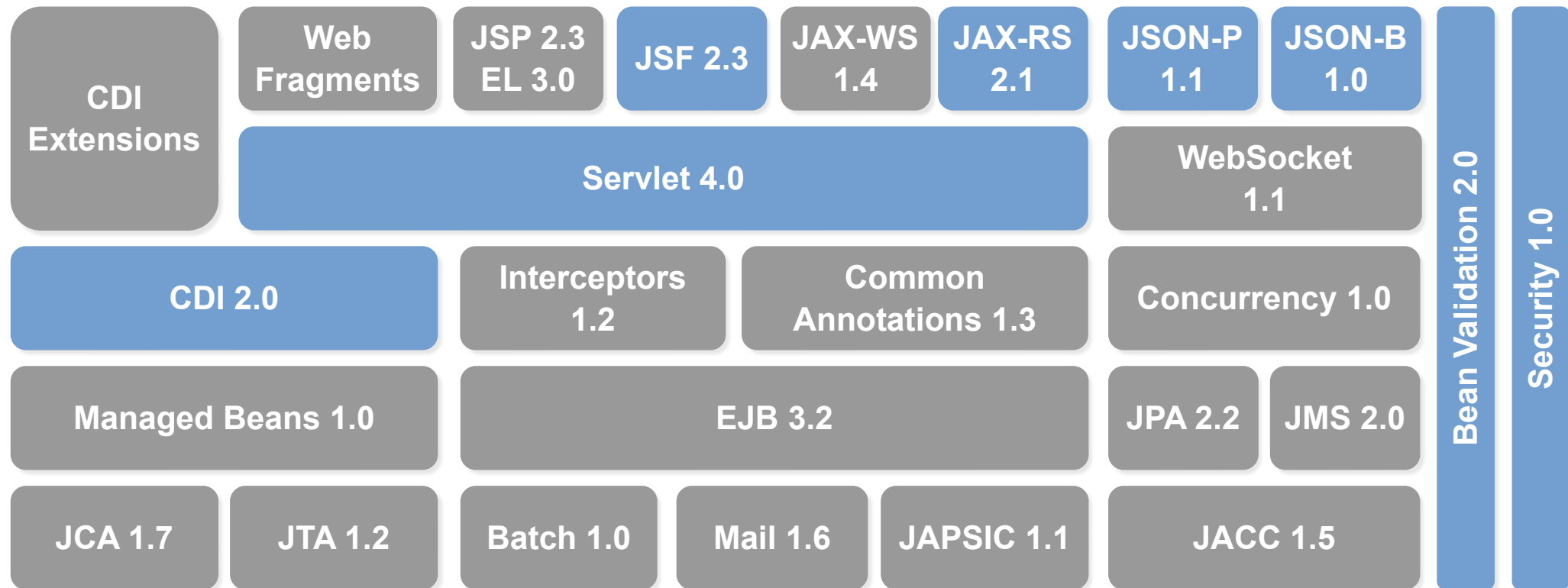
- **Eclipse GlassFish** (implémentation de référence)
- **Oracle WebLogic Server** (propriétaire)
- **IBM WebSphere** (propriétaire)
- **WildFly** (Red Hat & Jboss)
- **Apache Tomcat** (conteneur servlet/jsp)
- **Jetty** (moteur de servlet)



# Evolution de Java EE



# Java EE 8 : APIs



# Concept d'inversion de contrôle



- Patron d'architecture qui fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application mais du framework
- L'IoC est illustré par le principe de Hollywood :  
*« Ne nous appelez pas, c'est nous qui vous appellerons »*
- IoC permet de découpler les dépendances entre objets (Couplage → degré de dépendance entre objets)
- Avec l'IoC, le framework prend en charge l'exécution principale du programme, il coordonne et contrôle l'activité de l'application

# Concept d'inversion de contrôle

---



- Le rôle du développeur est de créer les blocs de code en utilisant l'API fournie par le framework, sans relation dure entre eux

Ces blocs de codes sont laissés à la discrétion du framework qui se chargera de les appeler

- Utilisation la plus connue : l'inversion des dépendances décrit dans l'article :

**dependency inversion principle de Robert C. Martin en 1994**

# Container léger



« SPRING est effectivement un conteneur dit “ léger ”, c’est-à-dire une infrastructure similaire à un serveur d'applications Java EE

Il prend donc en charge la création d’objets et la mise en relation d’objets par l’intermédiaire d’un fichier de configuration qui décrit les objets à fabriquer et les relations de dépendances entre ces objets

Le gros avantage par rapport aux serveurs d’application est qu’avec SPRING, les classes n’ont pas besoin d’implémenter une quelconque interface pour être prises en charge par le framework (au contraire des serveur d'applications Java EE et des EJBs)

C’est en ce sens que SPRING est qualifié de conteneur “ léger ” »

**Erik Gollot, Introduction au framework Spring**

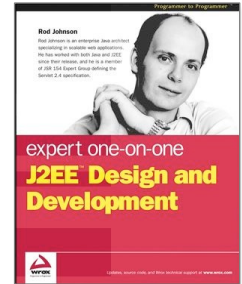
- Spring est un framework applicatif pour faciliter et améliorer la productivité de développement d'applications
- Il intègre :
  - l'inversion de contrôle via l'injection de dépendances
  - la programmation orientée aspect
  - une couche d'abstraction

La couche d'abstraction permet d'intégrer facilement des bibliothèques ou des frameworks déjà existants

# Historique de Spring

2002

**Rod Johnson** publie son livre  
**ExpertOne-on-One J2EE Design and Development**  
qui explique les raisons de la création de Spring



2004

**Spring 1.0** sort sous licence Apache 2  
Création de l'entreprise **interface21**

2006

**Spring 2.0** → Java 5, Groovy

2007

**Spring 2.5** → Java 6, Java EE 5, configuration par annotations  
Interface 21 devient **SpringSource**

2009

**Spring 3.0** → Java EE 6, configuration java  
Achat de SpringSource par VMWare (420 M\$)

2013

**Spring 4.0** → Java 8 et Java EE 7, inclut Spring Boot  
Création de **Pivotal**, joint venture entre VMWare et EMC

2017

**Spring 5.0** → Java 9, Kotlin, Reactive programming

2019

Acquisition de Pivotal Software par VMWare (2,7 milliards)

2022

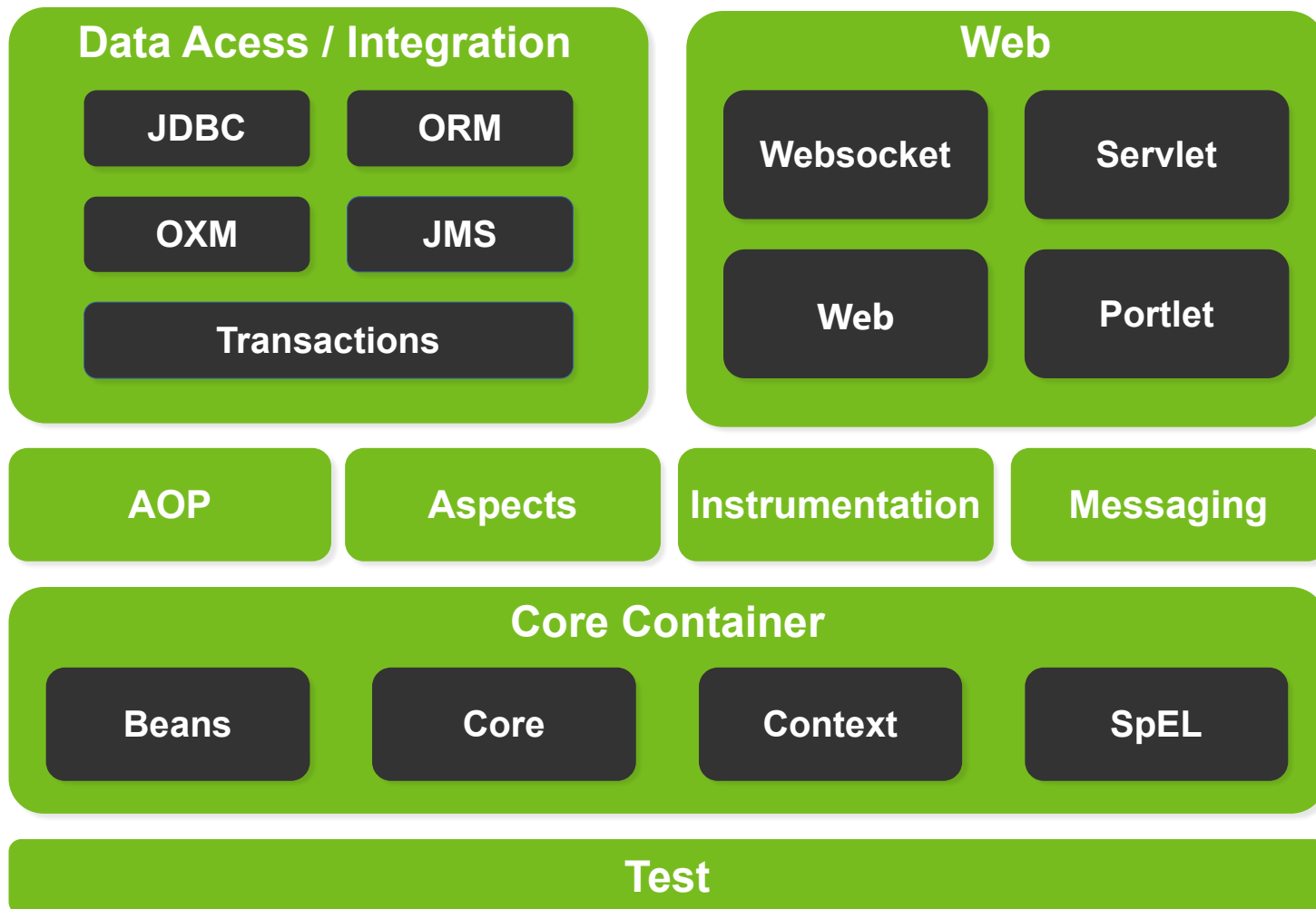
**Spring 6.0** → **Java 17**, Jakarta EE 9, compilation native, observabilité



# Structure du framework



## Spring Framework Runtime



# Galaxie Spring



Le cadre de Spring consistait à apporter un conteneur léger servant à l'loc  
Aujourd'hui, Spring représente un grand nombre de modules logiciels :



**Spring Framework** → contient les fonctionnalités de base de Spring  
(représente la version 1 de spring) **version: 5.3.24**



**Spring Boot** → pour simplifier le démarrage et le développement  
de nouvelles applications Spring **version: 2.7.7**



**Spring Security** → sécurité au niveau d'une application JEE  
(authentification et habilitation des utilisateurs) **version: 5.8.1**



**Spring Data** → a pour but de faciliter l'utilisation de solutions de type  
No SQL. Il est composé de plusieurs sous-projets, un pour les  
différentes solutions supportées **version: 2021.2.6**

# Galaxie Spring



**Spring Batch** → plan de production pour l'enchaînement de traitements par lots liés par des dépendances **version: 4.3.7**



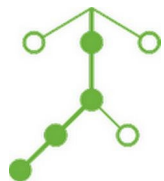
**Spring Cloud** → fournit des outils permettant aux développeurs de créer rapidement certains des modèles courants dans les systèmes distribués **version: 2021.0.5**



**Spring Web Flow** → développement d'interfaces web riches (ajax, jsf,...), utilise Spring MVC **version: 2.5.1**



**Spring Web Service** → permet de développer des services web de type SOAP **version: 3.1.4**



**Spring LDAP** → a pour but de simplifier l'utilisation d'annuaires de type LDAP **version: 2.4.0**

# Galaxie Spring



## Autre module :

 **Spring Shell (2.1.4)**

 **Spring HATEOAS (1.5.2)**

 **Spring REST Docs (2.0.7)**

 **Spring AMQP (2.4.8)**

 **Spring Integration (5.5.15)**

 **Spring Statemachine (3.2.0)**

 **Spring Session (2021.2.0)**

 **Spring Cloud Data Flow (2.9.6)**

 **Spring CredHub (2.2.0)**

 **Spring Flo (0.8.8)**

 **Spring for Apache Kafka (2.9.4)**

 **Spring Vault (2.3.1)**

 **Spring for GraphQL (1.1.1)**

# Développer avec Spring

- Tous les IDE supportant Java SE/Java EE



à condition de récupérer les bibliothèques nécessaires :  
Framework Spring, JMS, log, AOP...

- **Pivotal fournit un environnement complet**

- **Spring Tools Suite 4**  
↳ un IDE basé sur Eclipse



- **Pivotal tc Server**  
↳ Un serveur basé sur Tomcat

# Configuration de Spring Tool Suite 4



- À partir de **eclipse 4.18**, le JRE qui va exécuter eclipse est intégrée sous forme de plugin (openjdk 17)

Il n'est plus nécessaire de le configurer dans le fichier **eclipse.ini** avec l'option -vm

```
-vm
```

```
C:\Program Files\Java\jdk1.8.0_351
```

- **Ajouter le plugin pour support de JSP, HTML, CSS ...**

Dans help → eclipse market place → search → find

↳ tapez **jsp** et installer :

## Eclipse Enterprise Java and Web Developer Tools 3.28

# Configuration de Spring Tool Suite 4



- Dans windows → préférence
  - filtre sur **encoding**  
CSS Files, HTML Files, JSP Files : sélectionner UTF-8
  - filtre sur **text editors**  
cocher : Insert spaces for tabs  
cocher : remove multiple spaces and backspace/delete
  - filtre sur **spelling**  
décocher : enable spelling
  - filtre sur **jre**  
Installed JREs → Add → choisir : Standard VM  
JRE home : C:\Programmes\Java\jdk1.8.0\_351\jre  
JRE Name : jdk1.8.0\_351

# Configuration de Spring Tool Suite 4



- filtre sur **compiler**  
JDK Compliance → Compiler compliance level → 1.8
- filtre sur **formatter**  
java → code style → Formatter  
new → profil name: Eclipse  
indentation : tab policy choisir space only
- filtre sur **server**  
ajouter le serveur **Apache Tomcat 9**  
préciser le chemin du répertoire de Tomcat
- filtre sur **web browser**  
Choisir son navigateur  
(par défaut : le navigateur par défaut du système)



# Configuration du projet Spring Core



- Créer un projet Maven simple
  - ↳ cocher Create a simple project (skip archetype selection)
- Ajouter dans l'élément **<project>** de pom.xml :
  - la dépendance **spring-webmvc**

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.3.24</version>
  </dependency>
</dependencies>
```

on obtient par le jeu des dépendances liées :

spring-aop-5.3.24.jar	spring-expression-5.3.24.jar
spring-beans-5.3.24.jar	spring-jcl-5.3.24.jar
spring-context-5.3.24.jar	spring-web-5.3.24.jar
spring-core-5.3.24.jar	spring-webmvc-5.3.24.jar

# Configuration du projet Spring Core

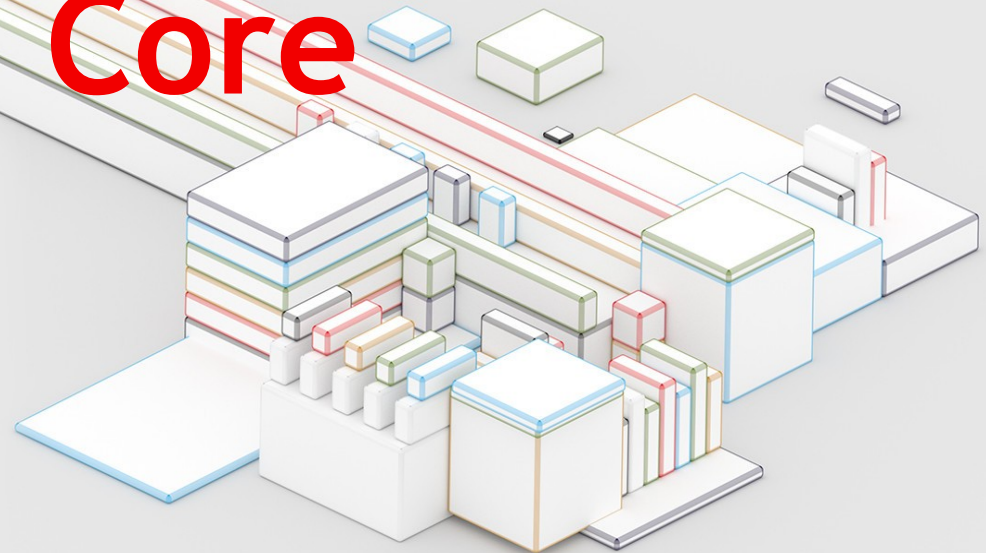


- pour compiler en java 8 et utiliser UTF-8 avec maven (avant **<dependencies>**)

```
<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
  <project.build.sourceEncoding>
    UTF-8
  </project.build.sourceEncoding>
</properties>
```

- Maven update  
clique gauche sur le projet → Maven → Update Maven...

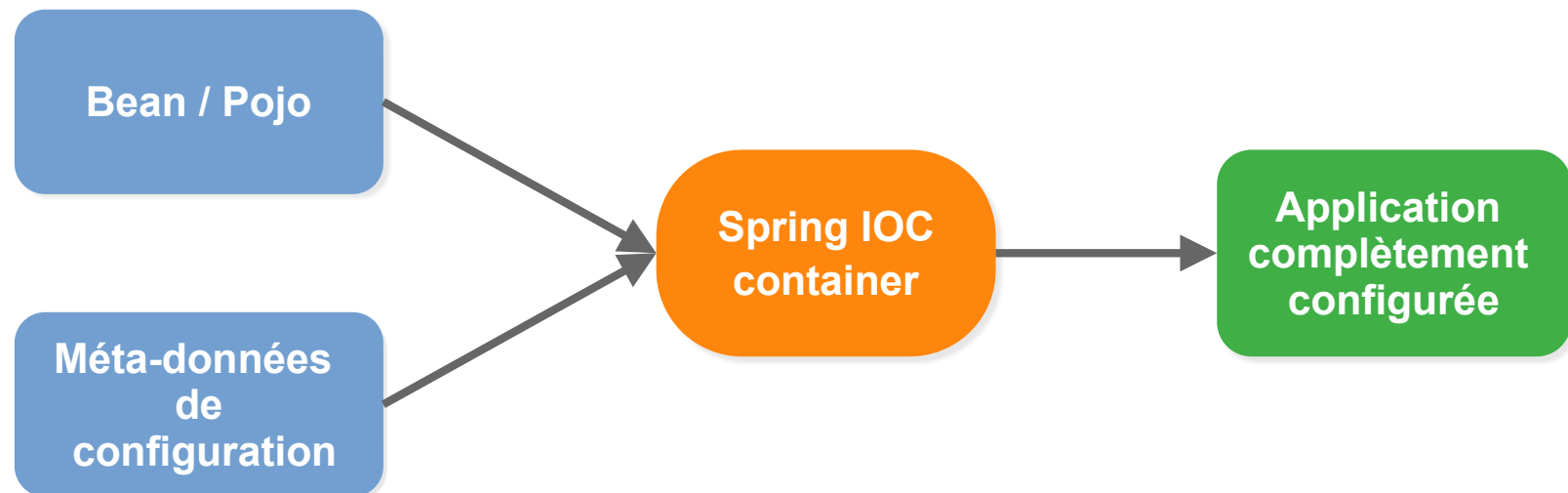
# Configurer des beans : Spring Core



# Le conteneur d'inversion de contrôle

## Le conteneur d'inversion de contrôle est le cœur de spring

- Le conteneur d'IOC utilise l'injection de dépendance pour gérer les composants qui constituent une application
- Les objets gérés par le conteneur d'IOC sont des **beans**
- Il reçoit ses instructions pour l'instanciation, la configuration et l'assemblage des beans en lisant les **métadonnées de configuration**



# Le conteneur d'inversion de contrôle

---



L'interface **ApplicationContext** représente le conteneur d'ioc

- Plusieurs implémentations sont fournies avec Spring :
  - pour les applications autonomes, on utilise :
    - `ClassPathXmlApplicationContext`
    - `FileSystemXmlApplicationContext`
    - `AnnotationConfigApplicationContext`
    - ...
  - pour les applications web, on utilise
    - `WebXmlApplicationContext`

# Méta-données de configuration

---



**Les méta-données peuvent être fournies :**

- en **XML** : méthode traditionnel
  - très souple et très puissante
- avec les **annotations java** (depuis Spring 2.5) :
  - plus rapide à utiliser et plus simple
- en **Java** (depuis Spring 3.0) :
  - permet de coder en Java un équivalent de la configuration XML
  - plus puissant, moins simple à modifier et moins répandu

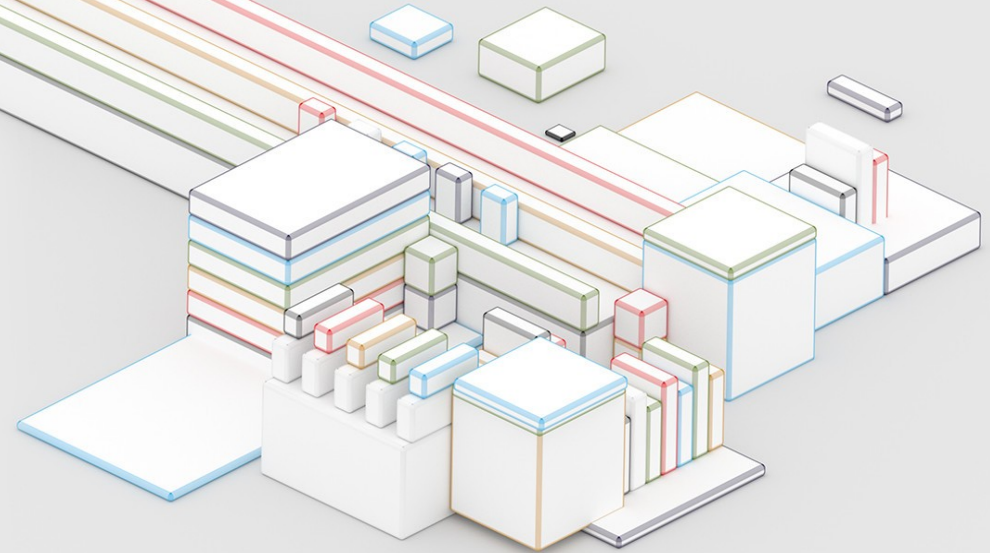
# Beans



## Un bean est défini par

- Un (ou plusieurs) identifiant(s) unique(s)  
Unicité au sein du conteneur
- Un nom complet de classe d'implémentation réelle du bean  
ex : `fr.dawan.formation>HelloWorld`
- Un comportement dans le conteneur  
Notion de : scope, lifecycle callbacks ...
- Ses références à d'autres beans, appelées dépendances ou collaborateurs

# Configurer en XML





# Fichier de configuration XML

- La configuration de Spring comprend au moins une définitions de bean que le conteneur doit gérer
- En XML, les beans sont configurés avec l'élément **<bean>** à l'intérieur de l'élément **<beans>** de niveau supérieur

```
<?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">
  <bean id="..." class="...">
  </bean>
</beans>
```

nom complet de la classe du bean

identifiant unique du bean

# Nommage des beans



- Les identifiants du bean doivent être unique dans le conteneur
- En XML, on utilise un des deux attributs
  - **id** : pour spécifier un nom
  - **name** : pour en définir plusieurs  
(séparé par , ; ou un espace)
- Si l'on ne fournit pas de nom au bean, le conteneur va en générer un unique → si l'on veut faire référence au bean, on est obligé de fournir un nom
- La convention pour le nommage des beans est la même que la convention de nommage identifiants des java  
ex : `userDAO`, `authenticationService`, ...

# Instancier un conteneur

- Pour une application java :

```
ApplicationContext context = new ClassPathXmlApplicationContext(  
    new String[] { "services.xml", "dao.xml" });
```

- Pour récupérer les instances des beans depuis le conteneur

```
T getBean(String name, Class<T> requiredType)
```

- On peut diviser la définition des beans en plusieurs fichiers XML et on les charge soit :
  - en passant les différents fichiers au constructeur
  - en utilisant l'élément **<import/>**, l'attribut **resource** permet de définir le chemin du fichier

```
<beans>  
  <import resource="services.xml"/>  
  <import resource="dao.xml"/>  
  ...
```

# Instancier des beans

- **Injection de dépendance**

C'est un processus où les objets définissent leur dépendance, uniquement à l'aide d'arguments de constructeur et des mutateurs

Le conteneur injecte ensuite ces dépendances lorsqu'il crée le bean

- **Injection de dépendance basé sur les mutateurs**

Le conteneur va appeler les setters sur les beans, après avoir appelé le constructeur sans argument

En xml, on utilise l'élément **<property>**

```
<bean id="contact1" class="fr.dawan.Contact">  
  <property name="nom" value="DOE"/>  
  <property name="prenom" value="John"/>  
</bean>
```

# Instancier des beans



- **Injection de dépendance basé sur le constructeurs**

Le conteneur va appeler un constructeur avec des arguments, chacun représentant une dépendance

En xml, on utilise l'élément **<constructor-arg>**

- **Résolution des arguments du constructeur**

S'il n'y a pas d'ambiguïté, l'ordre des arguments fournis au constructeur lors de l'instanciation du bean est celui de la définition du bean

```
<bean id="contact1" class="fr.dawan.Contact">  
  <constructor-arg value="DOE"/>  
  <constructor-arg value="John"/>  
</bean>
```

# Instancier des beans

- Sinon, on doit indiquer soit :
  - le type de l'argument avec l'attribut **type**

```
<bean id="exampleBean" class="fr.dawan.ExampleBean">  
  <constructor-arg type="int" value="7500000" />  
  <constructor-arg type="java.Lang.String" value="42"/>  
</bean>
```

- l'index de l'argument avec l'attribut **index** (commence à 0)

```
<bean id="exampleBean" class="fr.dawan.ExampleBean">  
  <constructor-arg index="0" value="7500000"/>  
  <constructor-arg index="1" value="42"/>  
</bean>
```

- le nom de l'argument avec l'attribut **name**

```
<bean id="exampleBean" class="fr.dawan.ExampleBean">  
  <constructor-arg name="years" value="7500000"/>  
  <constructor-arg name="ultimateAnswer" value="42"/>  
</bean>
```

# Instancier des beans



- L'attribut **value** : spécifie la valeur d'une propriété ou d'un argument de constructeur sous forme d'une chaîne de caractère (Spring fera la conversion vers le type réel)
- **Valeur de chaine vide**  
Spring traite les arguments vides comme des chaînes vides

```
<bean id="contactA" class="fr.dawan.Contact">  
  <property name="email" value=""/>  
</bean>
```

- L'élément **<null/>** gère les valeurs NULL

```
<bean id="contactA" class="fr.dawan.Contact">  
  <property name="email">  
    <null/>  
  </property>  
</bean>
```

Correspond à : `exampleBean.setEmail(null)`

# Portée des beans (scope)



On définit la portée d'un bean avec l'attribut **scope** de l'élément `<bean>`

- **Singleton** (par défaut) : crée une instance unique pour chaque conteneur IoC
- **Prototype** : crée une instance à chaque demande (getBean)

Les autres portées sont disponibles uniquement pour un **ApplicationContext** compatible Web (**WebApplicationContext**)

- **Request** : Crée une instance par requête HTTP
- **Session** : Crée une instance par session HTTP
- **Application** : Crée une instance dont la durée de vie est celle du **ServletContext**



# Processus de résolution de dépendance



Le container réalise l'injection de dépendance de la manière suivante:

- **ApplicationContext** est créé et initialisé avec la configuration des métadonnées qui décrit tous les beans
- Pour chaque bean, ses dépendances sont exprimées sous la forme de propriétés et d'arguments de constructeur  
Ces dépendances seront fournies au bean quand il sera créé
- Les beans **singleton** sont configurés pour être **pré-instanciés** (par défaut) et sont créés lors de la création du conteneur  
Cela permet de découvrir les erreurs de configuration au moment de l'initialisation

# Processus de résolution de dépendance



- Sinon, le bean est créé uniquement lorsqu'il est demandé
- La création d'un bean entraîne potentiellement la création d'un graphe de beans, au fur et à mesure que les dépendances du bean sont créées et attribuées

- **Lazy-initialized beans**

En ajoutant l'attribut **lazy-init** égale à **true** à l'élément **<bean>**, on indique au conteneur d'ioc de créer une instance du singleton à la première demande plutôt qu'au démarrage

```
<bean id="Lazy" class="fr.dawan.Bean" lazy-init="true"/>  
<bean name="not.Lazy" class="fr.dawan.AutreBean"/>
```

On peut rendre par défaut tous les beans lazy-initialized avec l'attribut **default-lazy-init="true"** sur l'élément **<beans>**

# Injection explicite des dépendances

Elle se configure à l'aide de

- l'attribut **ref** d'un élément `<property/>` ou `<constructor-arg/>`

```
<bean id="daoUser" class="fr.dawan.dao.UserDao" />
<bean id="userService" class="fr.dawan.service.UserService">
  <property name="userDao" ref="daoUser" />
</bean>
```

- l'élément **ref** et l'attribut **bean** à l'intérieur d'un élément `<constructor-arg/>` ou `<property/>`

```
<ref bean="daoUser"/>
```

- un élément `<bean/>` à l'intérieur d'un élément `<property/>` ou `<constructor-arg/>`, un bean interne n'est pas visible en dehors (Il est inutile de le nommer)

```
<bean id="userService" class="fr.dawan.service.UserService">
  <property name="userDAO">
    <bean class="fr.dawan.dao.UserDao" />
  </property>
</bean>
```

# Injection automatique des dépendances (autowiring)



Le conteneur Spring peut établir une liaison automatique entre les beans collaborateurs

En xml, on ajoute l'attribut **autowire** de l'élément **<bean>**

il y a **4 modes** d'autowiring :

- **no** : (par défaut) aucune liaison automatique, on lie les beans explicitement avec **<ref>**
- **byName** : liaison entre le nom de la propriété et celui du bean
- **byType** : liaison entre le type de la propriété et celui du bean
- **constructor** : liaison en utilisant le constructeur du bean

```
<bean id="contact1" class="fr.dawan.Contact" autowire="...">  
  <property name="nom" value="DOE"/>  
  <property name="prenom" value="John"/>  
</bean>
```

# Injection automatique des dépendances (autowiring)



- L'attribut **default-autowire** du tag **<beans>** permet de préciser le mode d'autowiring par défaut
- La définition explicite prend toujours le pas sur l'autowiring
- L'autowiring ne peut être utilisé que pour des objets : il n'est pas possible de l'utiliser avec des types primitif, String ou des tableaux
- L'autowiring permet de simplifier la configuration des beans mais aussi de maintenir cette configuration à jour lors de l'ajout d'une nouvelle dépendance dans un bean qui utilise l'autowiring
- Pour que l'autowiring fonctionne correctement, le conteneur doit être en mesure de déterminer de façon **non ambiguë** l'instance qui sera injectée

# Cycle de vie des beans

- L'attribut **init-method** de l'élément **<bean>** permet de spécifier le nom de la méthode qui sera appelée lorsque le bean sera instancié

```
<bean id = "exempleBean" class = "fr.dawan.ExempleBean"
      init-method = "init"/>

public class ExempleBean {
    public void init() {
        // ...
    }
}
```

- L'attribut **destroy-method** de l'élément **<bean>** permet de spécifier le nom de la méthode qui sera appelée juste avant qu'un bean soit retiré du conteneur

```
<bean id = "exempleBean" class = "fr.dawan.ExempleBean"
      destroy-method = "destroy"/>

public class ExempleBean {
    public void destroy() {
        // ...
    }
}
```

# Cycle de vie des beans



- Les méthodes doivent retourner **void** et n'ont pas d'arguments
- **Méthodes d'initialisation et de destruction par défaut**  
Si l'on a trop de beans qui ont des méthodes d'initialisation et / ou de destruction portant le même nom  
On peut déclarer **init-method** et **destroy-method** sur l'élément **<beans>**

```
<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-3.0.xsd"
  default-init-method = "init" default-destroy-method = "destroy">
  <bean id = "... " class = "... ">
    </bean>
</beans>
```

# Héritage de configuration

- **Définition d'un JavaBean**

```
public class Client extends Contact {  
    private String numero;  
    // ...  
}
```

- **Définition dans le fichier de configuration**

L'attribut **parent** sur l'élément **<bean>** enfant permet de définir le bean parent

```
<bean id="contact1" class="fr.dawan.Contact">  
    <property name="nom" value="USER1"/>  
    <property name="prenom" value="Mohamed"/>  
</bean>  
<bean id="client1" class="fr.dawan.Client" parent="contact1">  
    <property name="numero" value="DA-123456"/>  
</bean>
```

- **Redéfinition d'attributs**

```
<bean id="client1" class="fr.dawan.Client" parent="contact1">  
    <property name="nom" value="DERKAOUI"/>  
    <property name="numero" value="DA-123456"/>  
</bean>
```



# Template de configuration

- Héritage de configuration sans instanciation du parent  
L'attribut **abstract** sur l'élément **<bean>** permet d'indiquer qu'il ne peut être instancié

```
<bean id="contact1" class="fr.dawan.Contact" abstract="true">  
  <property name="nom" value="DOE"/>  
  <property name="prenom" value="John"/>  
</bean>  
<bean id="client1" class="fr.dawan.Client" parent="contact1">  
  <property name="numero" value="DA-123456"/>  
</bean>
```

- Pas d'héritage Java mais des caractéristiques communes

```
<bean id="personne1" abstract="true">  
  <property name="nom" value="DOE"/>  
  <property name="prenom" value="John"/>  
</bean>  
<bean id="client1" class="fr.dawan.Client" parent="personne1">  
  <property name="numero" value="DA-123456"/>  
</bean>  
<bean id="formateur1" class="fr.dawan.Formateur" parent="personne1">  
  <property name="matricule" value="JDOE0415"/>  
</bean>
```

# Configuration de collections

- Les éléments `<list/>`, `<set/>`, `<map/>` et `<props/>` définissent les propriétés et les arguments des types de collection Java **List**, **Set**, **Map** et **Properties**

```
public class Contact {  
    private String nom;  
    private String prenom;  
    // ...
```

```
    private List<Object> objects;  
    public void setObjects(List<Object> objects) {  
        this.objects = objects;  
    }
```

// ou

```
    private Object[] objects;  
    public void setObjects(Object[] objects) {  
        this.objects = objects;  
    }
```

```
<bean id="contact1" class="fr.dawan.Contact">  
    <constructor-arg value="MARRON"/>  
    <constructor-arg value="Benjamin"/>
```

```
<property name="objects">
```

```
    <list>
```

```
0    <value>A</value>
```

```
    -----  
    <bean class="java.net.URL">
```

```
1        <constructor-arg value="http"/>
```

```
        <constructor-arg value="www.dawan.fr"/>
```

```
        <constructor-arg value="/"/>
```

```
    </bean>
```

```
2    <null/>
```

```
    </list>
```

```
</property>
```

```
</bean>
```

# Configuration de collections

```
private Set<Object> objects;  
public void setObjects(Set<Object> objects) {  
    this.objects = objects;  
}
```

<set>

```
<value>A</value>  
-----  
<bean class="java.net.URL">  
    <constructor-arg value="http"/>  
    <constructor-arg value="www.dawan.fr"/>  
    <constructor-arg value="/" />  
</bean>
```

</set>

```
private Map<Object,Object> objects;  
public void setObjects(Map<Object,Object> obj) {  
    this.objects = obj;  
}
```

<map>

```
<entry key="type" value="A"/>  
-----  
<entry>  
    <value>URL</value>  
    <bean class="java.net.URL">  
        <constructor-arg value="http"/>  
        <constructor-arg value="www.dawan.fr"/>  
        <constructor-arg value="/" />  
    </bean>  
</entry>
```

</map>

# Configuration de collections

```
private Properties myProps;  
public void setObjects(Properties objects) {  
    this.myProps = objects;  
}
```

```
<props>  
  <prop key="type">A</prop>  
  <prop key="URL">http://www.dawan.fr</prop>  
</props>
```

- **Typier les éléments**

```
private List<Integer> lst;  
public void setObjects(List<Integer> lst) {  
    this.lst = lst;  
}
```

```
<list type="int">  
  <value>5</value>  
  <value>10</value>  
  <value>15</value>  
</list>
```

```
<list>  
  <value type="int">5</value>  
  <value type="int">10</value>  
  <value type="int">15</value>  
</list>
```

On n'a pas besoin de typer dans le fichier de configuration

```
<list>  
  <value>5</value>  
  <value>10</value>  
  <value>15</value>  
</list>
```

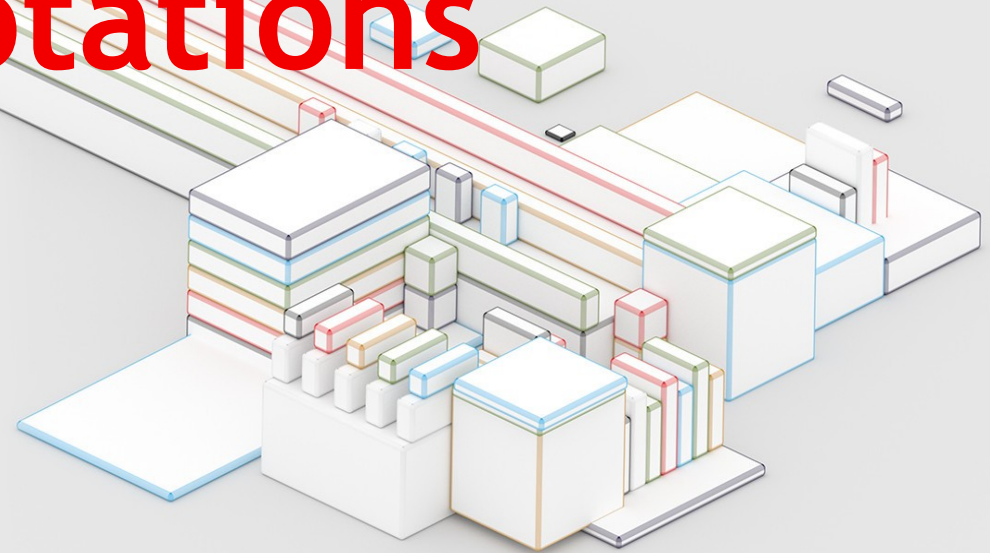
# Collection et Héritage

- Compléter une collection du bean parent

```
<bean id="contact1" class="org.dawan.formation.Contact">
  <property name="nom" value="MARRON" />
  <property name="prenom" value="Benjamin" />
  <property name="objects">
    <list>
      <value>A</value>
      <value>B</value>
      <value>C</value>
    </list>
  </property>
</bean>
<bean id="client1" class="org.dawan.formation.Client" parent="contact1">
  <property name="numero" value="DA-123456" />
  <property name="objects">
    <list merge="true">
      <value>E</value>
      <value>D</value>
    </list>
  </property>
</bean>
```

Fonctionne pour les types de collections (**list**, **set**, **map**)

# Configurer avec les annotations



# Configuration avec les annotations

- Pour informer le conteneur de l'utilisation des annotations pour réaliser une partie de la configuration, on ajoute **<context:annotation-config/>** dans l'élément **<beans>**

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

  <context:annotation-config/>

</beans>
```

# Configuration avec les annotations



- L'élément **<context:component-scan>** permet d'indiquer les packages qu'il devra scanner pour rechercher les classes annotées

```
<context:component-scan base-package="fr.dawan.formation"/>
```

- L'attribut **base-package** permet de préciser le nom du package
- La configuration par le fichier XML est prioritaire sur la configuration faite avec les annotations



# Composants



- Une entité Spring va être marquée par une annotation :
  - **@Component** pour un composant générique
- De manière plus spécifique nous pouvons catégoriser les composants en :
  - **@Controller** pour les contrôleurs
  - **@Service** pour les services
  - **@Repository** pour les gestionnaires des données
- L'attribut **value** de ces annotations permet de spécifier le nom du bean

# L'annotation @Autowired

- L'annotation **@Autowired** permet de faire de l'injection automatique de dépendances **basée sur le type**
- Elle s'utilise sur une propriété, un setter ou un constructeur
- L'attribut **required** permet de préciser si l'injection d'une instance dans la propriété est obligatoire (par défaut à **true**)
- Depuis Spring 4.3, **@Autowired** sur le constructeur n'est plus nécessaire, si le bean ne définit qu'un seul constructeur (Si plusieurs constructeurs sont disponibles, on doit en annoter un pour indiquer celui qui doit être utiliser)

```
public class PersonneService {  
    @Autowired  
    private PersonneDao personneDao;  
    public void setPersonneDao(PersonneDao personneDao) {  
        this.personneDao = personneDao;  
    }  
}
```

# L'annotation @Qualifier



- L'annotation **@Qualifier** permet de qualifier le candidat à une injection automatique **avec son nom**  
C'est utile lorsque plusieurs instances sont du type à injecter
- Elle s'utilise avec l'annotation **@Autowired**
- Elle peut s'appliquer sur
  - sur un attribut

```
@Autowired
@Qualifier("per1")
private Personne personne;
```

- sur un setter

```
@Autowired
public void setPersonne(@Qualifier("per1") Personne personne){
    this.personne = personne;
}
```

# L'annotation @Qualifier



- sur un paramètre d'une méthode

```
@Autowired
public void initialiser(@Qualifier("per1") Personne personne) {
    this.personne = personne;
}
```

- sur un constructeur

```
@Autowired
public Constructeur(@Qualifier("per1") Personne personne) {
    this.personne = personne;
}
```

- On peut aussi utiliser l'annotation **@Primary**, pour définir le bean qui sera utilisé, s'il y a plusieurs candidats pour injection avec **@Autowired**

# L'annotation @Resource



- L'annotation **@Resource** permet de demander l'injection d'un bean par son nom
- Elle s'utilise sur un champ ou une méthode
- Pour demander l'injection d'un bean précis, il faut fournir son identifiant comme valeur de l'attribut **name**
- Sans attribut, l'annotation agit comme **@Autowired**
- L'annotation détermine le bean concerné en recherchant
  - celui dont le nom correspond à l'attribut **name** de **@Resource**
  - sur le nom de la propriété
  - sur le type du bean

```
public class PersonneDao {  
    @Resource(name="maDataSource")  
    private DataSource dataSource;  
}
```

# Les annotations de gestion du cycle de vie des beans



- L'annotation **@PostConstruct**

Elle permet de marquer une méthode comme devant être exécutée à l'initialisation d'une nouvelle instance

```
public class MonBean {  
    @PostConstruct  
    public void initialiser() {  
        ...  
    }  
}
```

- L'annotation **@PreDestroy**

Elle permet de marquer une méthode comme devant être exécutée à la destruction d'une instance

```
public class MonBean {  
    @PreDestroy  
    public void detruire() {  
        ...  
    }  
}
```

# L'annotation @Scope



- L'annotation **@Scope** permet de préciser la portée du bean
- Elle s'applique sur une classe
- Les valeurs utilisables sont :
  - **singleton (par défaut)** : crée une instance unique pour chaque conteneur IoC
  - **prototype** : crée une instance à chaque demande
  - **request (web)** : crée une instance par requête HTTP
  - **session (web)** : crée une instance par session HTTP
  - **application (web)** : crée une instance dont la durée de vie est celle du `ServletContext`

```
@Controller  
@Scope("prototype")  
public class MonController { // ... }
```

# L'annotation @Lazy

- Par défaut, Spring crée tous les beans singleton au démarrage du contexte d'application
- L'annotation **@Lazy** permet de retarder le chargement des singletons à leur première utilisation
- On peut la placer, au niveau du bean  
équivalant à **lazy-init = "true"** en xml
- La valeur par défaut est **true**

```
@Lazy(true)
@Bean
public Formation getFormation(){
    return new Formation();
}
```



# Annotation standard (JSR 330)

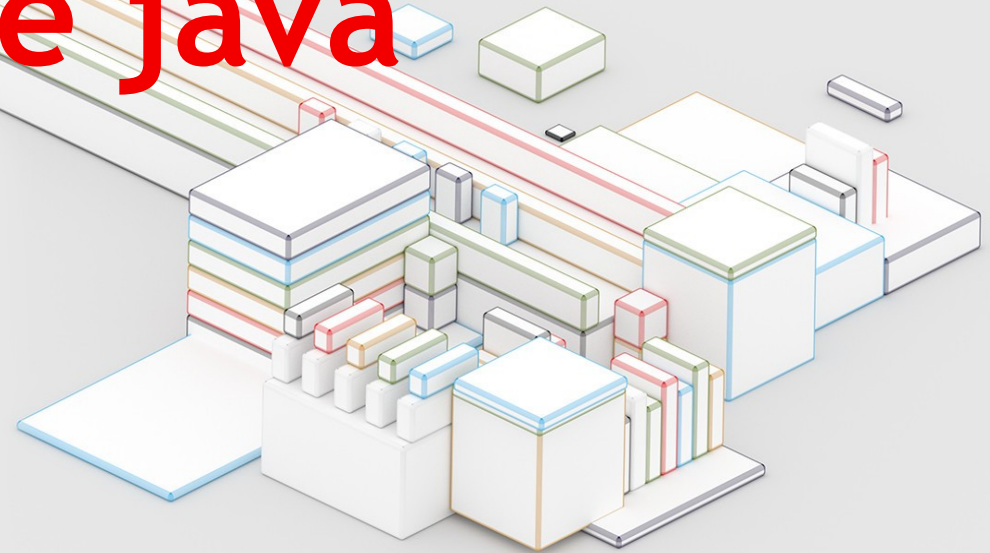


- Les annotations standard sont supporté depuis Spring 3.0  
Pour les utiliser, il faut ajouter la dépendance **javax.inject**

```
<dependency>  
  <groupId>javax.inject</groupId>  
  <artifactId>javax.inject</artifactId>  
  <version>1</version>  
</dependency>
```

Spring	javax.inject.*	
@Autowired	@Inject	@Inject n'a pas d'attribut required
@Component	@Named	ou @ManagedBean
@Scope("singleton")	@Singleton	La porté par défaut avec JSR-330 est l'équivalent de prototype avec Spring Mais un bean JSR-330 déclaré dans le conteneur Spring est un singleton par défaut Pour utiliser une porté autre que singleton, on doit utiliser l'annotation @Scope de Spring
@Qualifier	@Named	

# Configurer avec du code java




# Configuration avec Java

- Depuis Spring **3.0**, on peut faire la configuration avec du code java
- L'annotation **@Configuration** est utilisée sur une classe pour indiquer que son objectif est de fournir des définitions de bean
- L'annotation **@Bean** indique que la méthode instancie, configure et initialise un nouvel objet géré par le conteneur ioc Spring (équivalent de **<bean/>**)

```
@Configuration
public class AppConfig {
    @Bean
    public MyService myService() {
        return new MyService();
    }
}
```

correspond à

```
<beans>
    <bean id="myService" class="fr.dawan.MyService"/>
</beans>
```



# Instancier un conteneur



- On utilise pour implémentation de ApplicationContext : **AnnotationConfigApplicationContext**  
Les classes **@Configuration** peuvent être utilisées en paramètre du constructeur

```
public static void main(String[] args) {  
    ApplicationContext ctx = new  
        AnnotationConfigApplicationContext(AppConfig.class);  
    MyService myService = ctx.getBean(MyService.class);  
    myService.doStuff();  
}
```

- Avec l'annotation **@ComponentScan** placé sur la classe, on indique les packages où rechercher les composants

```
@Configuration  
@ComponentScan(basePackages = "fr.dawan.formation")  
public class AppConfig { // ... }
```

# Déclarer un bean



- Pour déclarer un bean, il faut annoter une méthode avec l'annotation **@Bean**
- Cette méthode permet d'inscrire une définition de bean dans `ApplicationContext` du type spécifié par la valeur de retour de la méthode
- Par défaut, le nom du bean aura le même nom que la méthode

```
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService() {
        return new TransferService();
    }
}
```

correspond à

```
<beans>
  <bean id="transferService" class="fr.dawan.TransferService"/>
</beans>
```

# Déclarer un bean

- On peut modifier le nom du bean avec l'attribut **name**

```
@Configuration
public class AppConfig {
    @Bean(name = { "dataSource", "dataSourceA" })
    public DataSource dataSource() {
        //...
    }
}
```

- L'annotation **@Description** permet d'ajouter une description au bean

```
@Bean
@Description("Provides a basic example of a bean")
public Foo foo() {
    return new Foo();
}
```

# Déclarer un bean

- L'annotation **@Import** permet de charger des définitions de bean depuis une autre classe de configuration

```
@Configuration
public class ConfigA {
    @Bean
    public A a() {
        return new A();
    }
}

@Configuration
@Import(ConfigA.class)
public class ConfigB {
    @Bean
    public B b() {
        return new B();
    }
}
```

# Dépendance des beans

- On peut matérialiser la dépendances à l'aide des paramètres de la méthode

```
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService(AccountRepository accountRepository)
    {
        return new TransferServiceImpl(accountRepository);
    }
}
```

- Lorsque les beans ont des dépendances les uns aux autres  
On exprime cette dépendance en ayant une méthode du bean qui en appelle une autre

```
@Bean
public Foo foo() {
    return new Foo(bar());
}

@Bean
public Bar bar() {
    return new Bar();
}
```



# L'annotation @Lazy sur une classe de configuration



- Si l'annotation **@Lazy** est présente dans une classe de configuration (**@Configuration**), cela indique que toutes les méthodes annotées avec **@Bean** sont en lazy loading
- équivalent de **default-lazy-init="true"** en xml

```
@Lazy
@Configuration
@ComponentScan(basePackages = "fr.dawan.formation")
public class LazyAppConfig {
    @Bean
    public UserService userService(){
        return new UserService();
    }
    //...
}
```

# Cycle de vie des beans

- On utilise l'annotation **@Scope** pour spécifier la portée des beans

```
@Configuration
public class MyConfiguration {
    @Bean
    @Scope("prototype")
    public Encryptor encryptor() {
        ...
    }
}
```

- L'annotation **@Bean** a des attributs **init-method** et **destroy-method** comme l'élément `<bean>` en XML

```
public class Foo {
    public void init() {
        ...
    }
    public void cleanup() {
        ...
    }
}
```

```
@Configuration
public class AppConfig {
    @Bean(initMethod = "init", destroyMethod = "cleanup")
    public Foo foo() {
        return new Foo();
    }
}
```



**Plus d'informations sur <http://www.dawan.fr>**

**Contactez notre service commercial au  
09.72.37.73.73 (prix d'un appel local)**