

# Développement des applications Web basées sur les microservices (Partie Backend)

Responsable du cours:

Dr. Ing. Mariam LAHAMI

Dr. Ing. Sihem LOUKIL

Email: [mariam.lahami@enis.tn](mailto:mariam.lahami@enis.tn)

[sihem.loukil@enis.tn](mailto:sihem.loukil@enis.tn)



**SPRING**  
Framework

# Introduction au Framework Spring

- Introduction à Spring
- Architecture de Spring
- Les bonnes pratiques dans Spring

# Introduction à Spring (1 / 3)

## Problématiques

- Les développements Java/JEE, notamment ceux qui utilisent les EJB, sont réputés complexes, tant en terme de développement que de tests et de maintenance.
  - JEE impose **une plateforme d'exécution lourde**, qui pose des problèmes d'interopérabilité entre les différentes implémentations.
  - Les développements JEE se caractérisent par leur **forte dépendance** et s'avèrent souvent difficiles à tester

# Introduction à Spring (2/3)

## Motivation

- Simplifier et structurer les développements JEE de manière à respecter les meilleures pratiques d'architectures logicielles.

Utiliser le framework Spring pour concevoir des applications **performantes, faiblement couplées**, facilement **testables** et dont le code est **réutilisable**

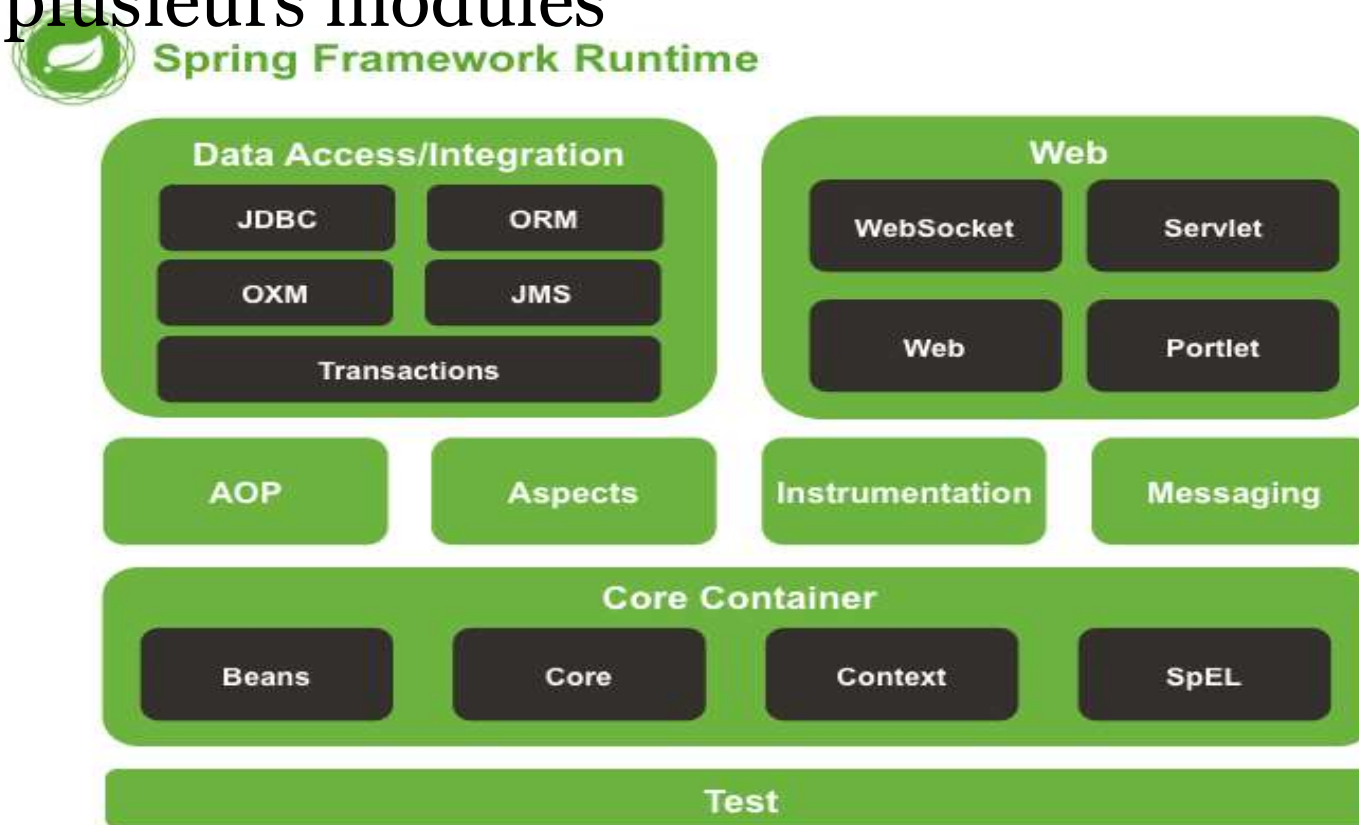
# Introduction à Spring (3/3)

## Qu'est-ce que Spring ?

- Framework offrant un cadre de développement en se basant sur des «bonnes pratiques»
- A l'origine orienté Java et Java EE
  - Aujourd'hui d'autres implémentations existent : .NET et Python
- Un conteneur « léger»
  - Facilite le développement avec des POJO (Plain Old Java Object) qui n'ont pas besoin d'être exécutés dans un conteneur/serveur d'application spécifique
  - Permet d'avoir des composants «faiblement couplés»
  - Améliore la qualité du code et facilite les tests

# Architecture Spring

- Spring se compose d'un noyau (Core Container) et de plusieurs modules



**Source :** <http://docs.spring.io/spring-framework/docs/current/springframework-reference/html/overview.html>

# Les bonnes pratiques dans Spring:

## Décomposition en couches (1/2)

- **Principe:** la division en couches est une technique répandue pour décomposer logiquement un système complexe
- **Avantages:**
  - possibilité d'isoler une couche, afin de faciliter sa compréhension et son développement ;
  - très bonne gestion des dépendances ;
  - possibilité de substituer les implémentations de couches ;
  - réutilisation facilitée ;
  - testabilité favorisée (grâce à l'isolement et à la possibilité de substituer les implémentations).

# Les bonnes pratiques dans Spring

## Décomposition en couches(2/2)

Présentation (e.g., JSP)

**Vue**

Coordination (e.g., Spring MVC, JSF, Struts)

**Contrôleur**

Métier (POJO)

**Service**

Accès aux données (e.g., JDBC, Hibernate, JPA)

**DAO**

Persistance (e.g., base de données)

**Entrepôt de données**



# Les bonnes pratiques dans Spring:

## Programmation par contrat(1/9)

- **Principe:** consiste à séparer la spécification d'une couche logicielle de sa réalisation.
  - La spécification donne lieu à la création d'une **interface** et la réalisation fournit une **classe qui implante cette interface**.
  - On peut produire plusieurs implantations différentes d'une même interface.

# Les bonnes pratiques dans Spring:

## Programmation par contrat(2/9)

- Avantages:
  - **Réduire les dépendances**
  - **Faciliter les tests**
  - **Simplifier le code**
  - **Organisation du développement**

# Les bonnes pratiques dans Spring: Programmation par contrat(3/9)

- Exemple:

```
interface IUserDAO{  
    public void addUser (User user);  
    Public getUserById(Long Id);  
}
```

```
class UserImplJDBC implements IUserDAO{  
    //add unimplemented method  
}
```

```
class UserImplJPA implements IUserDAO{  
    //add unimplemented method  
}
```

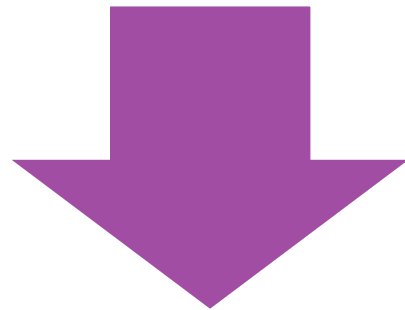


## Les bonnes pratiques dans Spring: Programmation par contrat(4/9)

- Avantage majeur: construire des applications **faiblement couplées**



Couplage faible



Couplage fort

# Les bonnes pratiques dans Spring

## Programmation par contrat(5/9)

### **Couplage fort vs couplage faible**

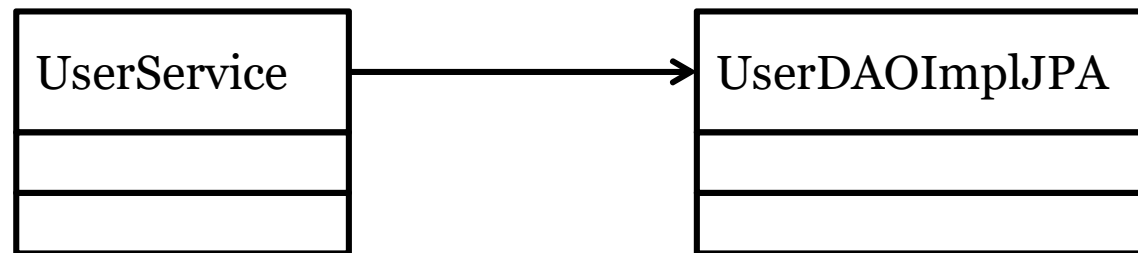
- Couplage fort: Quand une classe A est liée à une classe B, on dit que la classe A est **fortement couplée** à la classe B.
  - La classe A ne peut fonctionner qu'en présence de la classe B.
  - Si une nouvelle version de la classe B (soit B2), est créée, on est obligé de modifier dans la classe A.
  - Modifier une classe implique:
    - Il faut disposer du code source.
    - Il faut recompiler, déployer et distribuer la nouvelle application aux clients.
    - Ce qui engendre un cauchemar au niveau de la maintenance de l'application

# Les bonnes pratiques dans Spring

## Programmation par contrat(6/9)

### **Couplage fort vs couplage faible**

- Exemple de couplage fort: Chaque service métier contient des références vers des objets d'accès aux données.



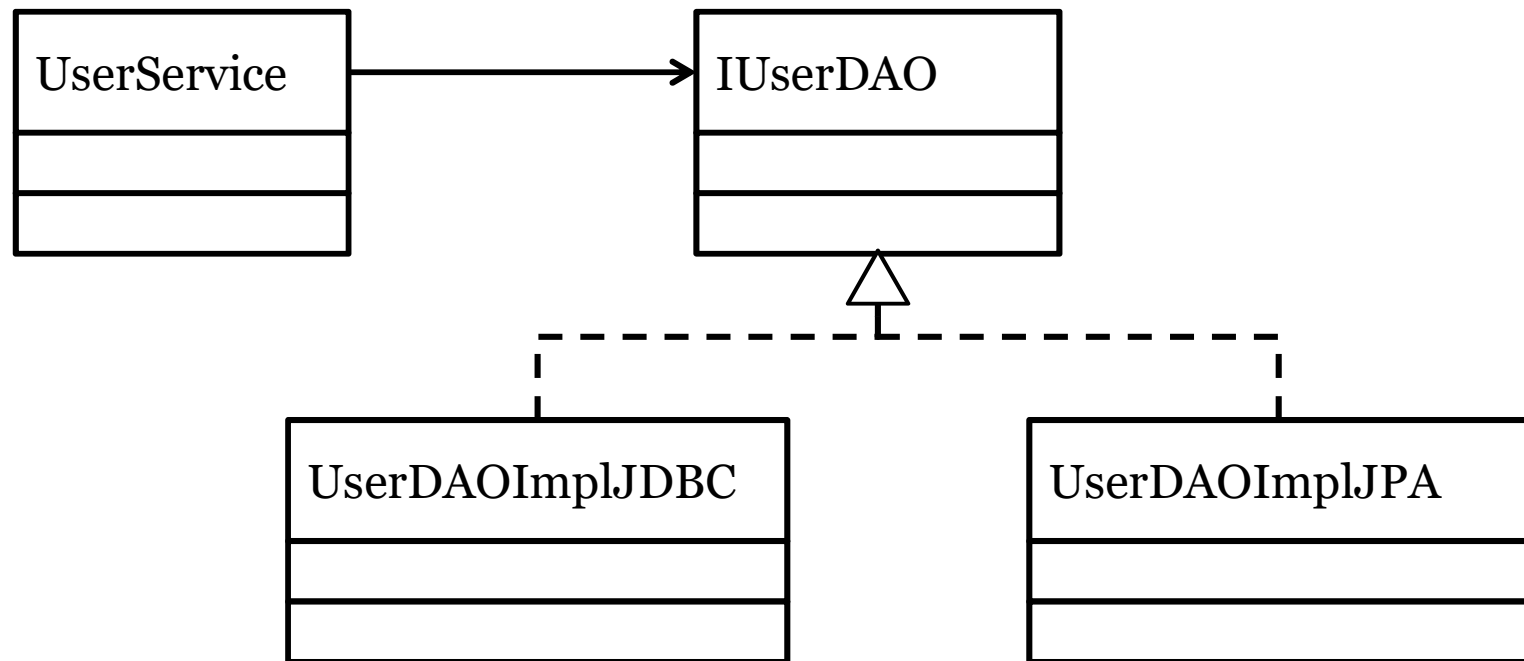
*Dépendances entre un service métier et un DAO  
(couplage fort)*

# Les bonnes pratiques dans Spring

## Programmation par contrat(7/9)

### Couplage fort vs couplage faible

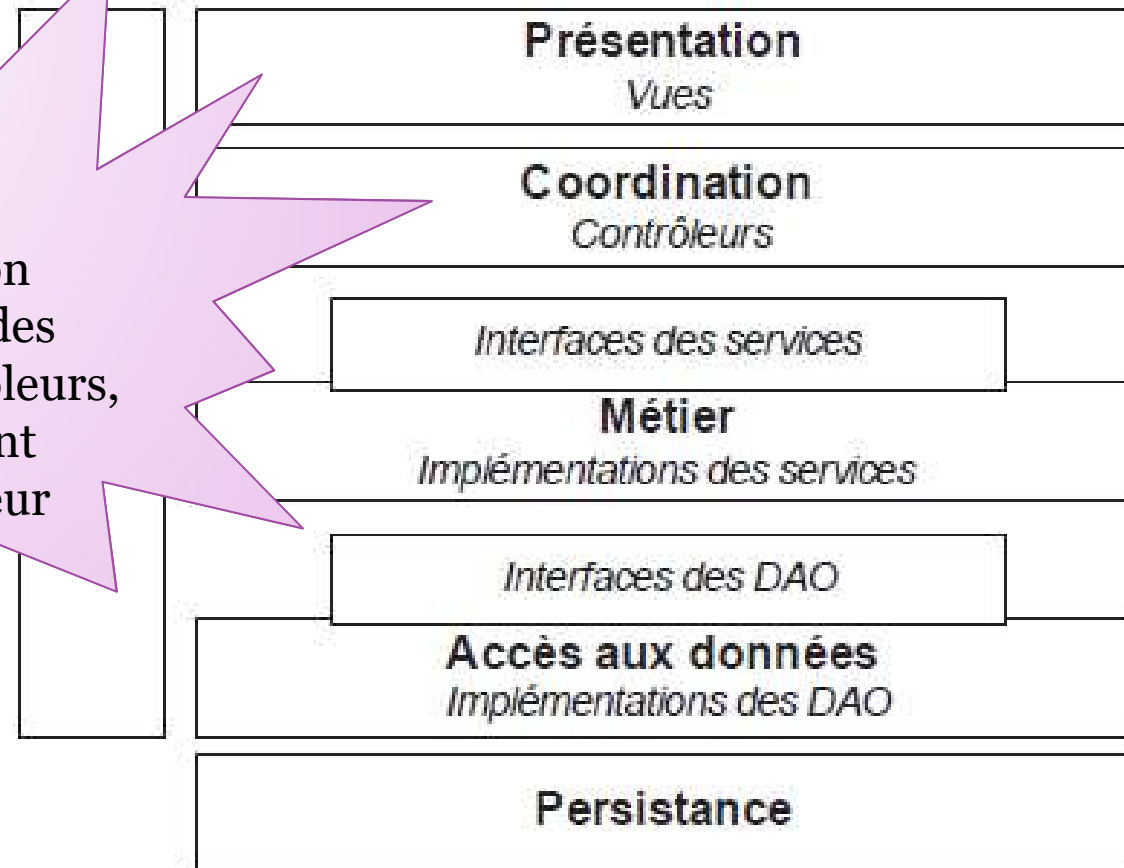
- Couplage **faible**: consiste à utiliser **les interfaces** afin de réduire les dépendances entre les classes.



*Dépendances entre un service métier et un DAO (couplage faible)*

# Les bonnes pratiques dans Spring: Programmation par contrat(9/9)

Dans une application Spring, l'ensemble des composants (contrôleurs, services et DAO) sont gérés par le conteneur léger.



**Communication intercouche par le biais des interfaces**



# Les bonnes pratiques dans Spring:

## Inversion de contrôle (1/2)

Basée sur **l'injection de dépendances**

- un mécanisme permettant de dynamiser la gestion de dépendance entre objets

 Rôle du conteneur : gérer cycle de vie des composants+ leurs dépendances

- But :
  - facilite l'utilisation des composants qu'on n'a pas développés
  - minimise l'instanciation statique d'objets (avec l'opérateur **new**)

# Les bonnes pratiques dans Spring:

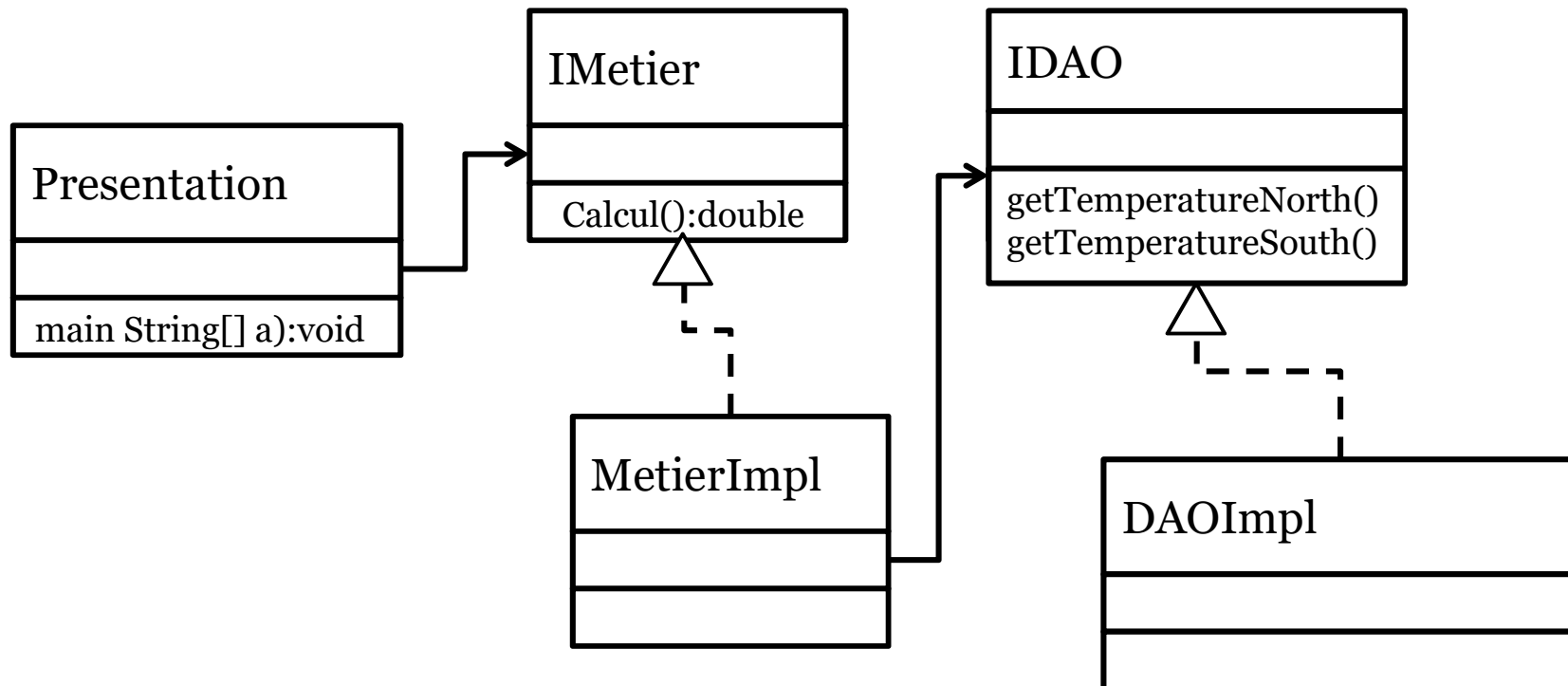
## Inversion de contrôle (2/2)

### **La notion d'injection de dépendance**

- Il existe trois types d'injection :
  - **L'injection par constructeurs:** un objet se voit injecter ses dépendances au moment où il est créé, c'est-à dire via les arguments de son constructeur.
  - **L'injection par mutateurs (setters):** un objet est créé, puis ses dépendances lui sont injectées par les setters.
  - **L'injection par champ :** méthode magique de plus en plus populaire

# Injection des dépendances Via un exemple (1/4)

- Soit l'exemple suivant:



# Injection des dépendances Via un exemple (2/4)

- Soit l'exemple suivant:

```
public interface IDao {  
    public double getTemperatureNorth();  
    public double getTemperatureSouth();  
}
```



```
public class DaoImpl implements IDao {  
    public double getTemperatureNorth() {  
        return 5.75;  
    }  
    public double getTemperatureSouth() {  
        return 30.6;  
    }  
}
```

# Injection des dépendances Via un exemple(3/4)

- Soit l'exemple suivant:

```
public interface IMetier {  
    public double calculMoy();  
}
```



```
public class MetierImpl2 implements IMetier {  
    private IDao dao;  
    public void setDao(IDao dao) {  
        this.dao = dao;  
    }  
    public double calculMoy() {  
        return (dao.getTemperatureNorth()+dao.getTemperatureSouth())/2;  
    }  
}
```

# Injection des dépendances Via un exemple (4/4)

## Injection par instantiation statique

```
public class Presentation {  
  
    public static void main(String[] args) {  
        Dao dao=new DaoImpl();  
        MetierImpl metier=new MetierImpl();  
        metier.setDao(dao);  
        System.out.println(metier.calculMoy());  
    }  
}
```

# Injection des dépendances avec Spring

- L'injection de dépendances peut être effectuée via
  - Un fichier XML
  - Une classe de configuration en utilisant les annotations @ Configuration et @Bean
  - des annotations (telles que @Autowired, @Component, @Service, @Resource, @Repository...)

# Injection des dépendances avec Spring

- Utilisation des annotations lors l'injection des dépendances

- Sur un champ

```
@Component("metier")
public class MetierImpl implements IMetier {
    @Autowired
    private IDao dao;
    ...
}
```

- Sur un setter

```
@Autowired
public void setDao(IDao dao) {
    this.dao = dao;
}
```

- Sur un constructeur

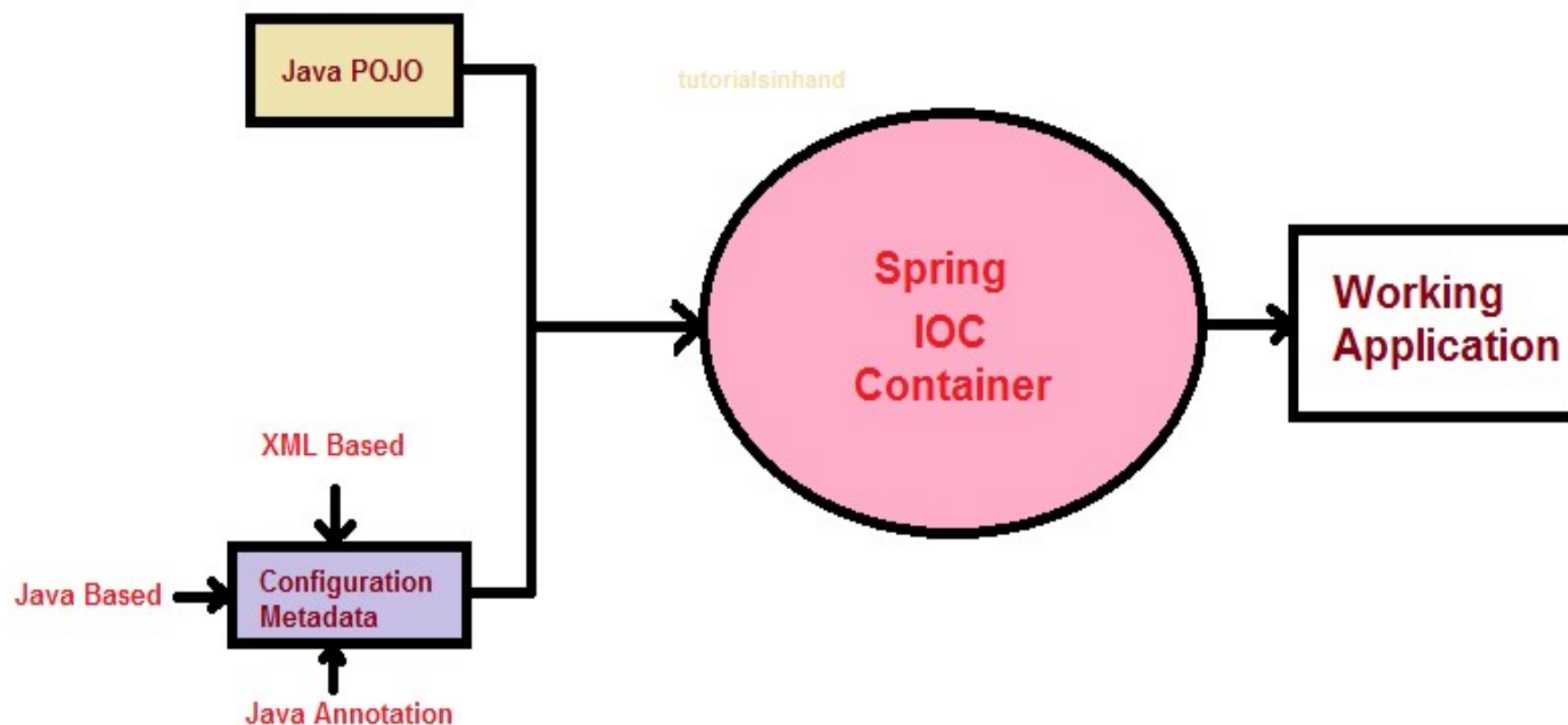
```
@Autowired
public MetierImpl(IDao dao) {
    this.dao = dao;
}
```



# Synthèse

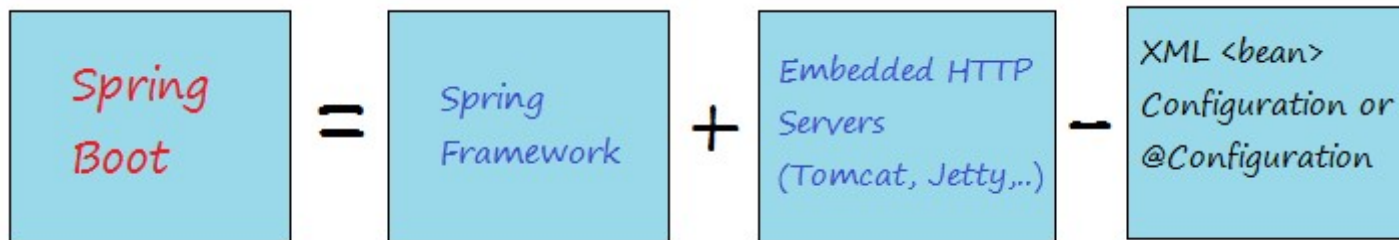
- La construction des objets de notre application va être déléguée à ce composant que l'on appelle **un conteneur IoC** (*IoC container*).
  - Aussi appelé le contexte Spring
- Ce conteneur accueille un ensemble d'objets dont il a la responsabilité de gérer le cycle de vie.
- Le Spring Framework est avant tout un conteneur IoC.  
=> Spring Framework est responsable de la création des objets et assure que les dépendances entre eux sont correctement créées.

# Synthèse



Working of Spring Container

# Spring Boot



# C'est quoi Spring Boot ?

- C'est un Framework qui a été conçu essentiellement pour développer des architectures à base de **micro-services**
  - L'idée est de découper une application en petites unités implémentées sous forme de micro-services
  - Chaque micro-service est responsable d'une fonctionnalité élémentaire, développé, testé et déployé indépendamment des autres
  - Chaque micro-service peut être conçu à l'aide de n'importe quel langage et technologie  
=>Couplage faible




# Avantages de Spring Boot

- Faciliter le développement d'applications complexes
- Faciliter l'injection des dépendances
- Faciliter la gestion des dépendances avec Maven
- Réduire les fichiers de configuration et supporter l'auto-configuration
- Fournir un conteneur léger embarqué (Tomcat)

# Création d'un projet Spring Boot (1/4)

← → ↻ 🏠 <https://start.spring.io/>

Applications Boîte de réception... Courrier :: Boîte de... Bienvenue sur Face... Google Sci-Hub: removing... ReDCAD Wel

 **Spring Initializr**  
Bootstrap your application

Project

Language

Spring Boot

Project Metadata

Maven Project

Gradle Project

Java

Kotlin

Groovy

2.2.0 RC1

2.2.0 (SNAPSHOT)

2.1.10 (SNAPSHOT)

2.1.9

Group  
com.example

Artifact  
demo

> Options

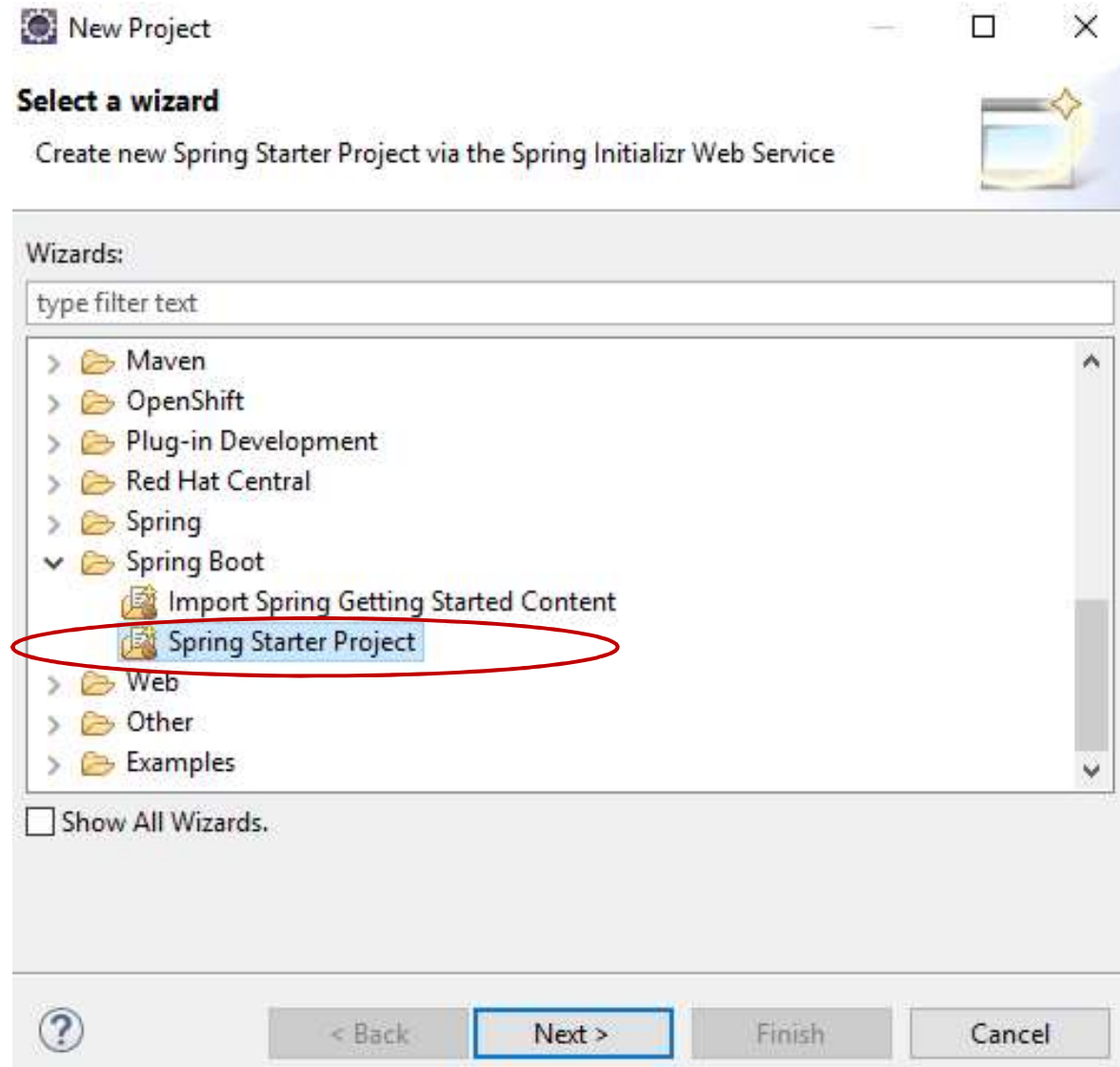
© 2013-2019 Pivotal Software  
start.spring.io is powered by  
[Spring Initializr](#) and [Pivotal Web Services](#)

Generate - Ctrl + ⌘

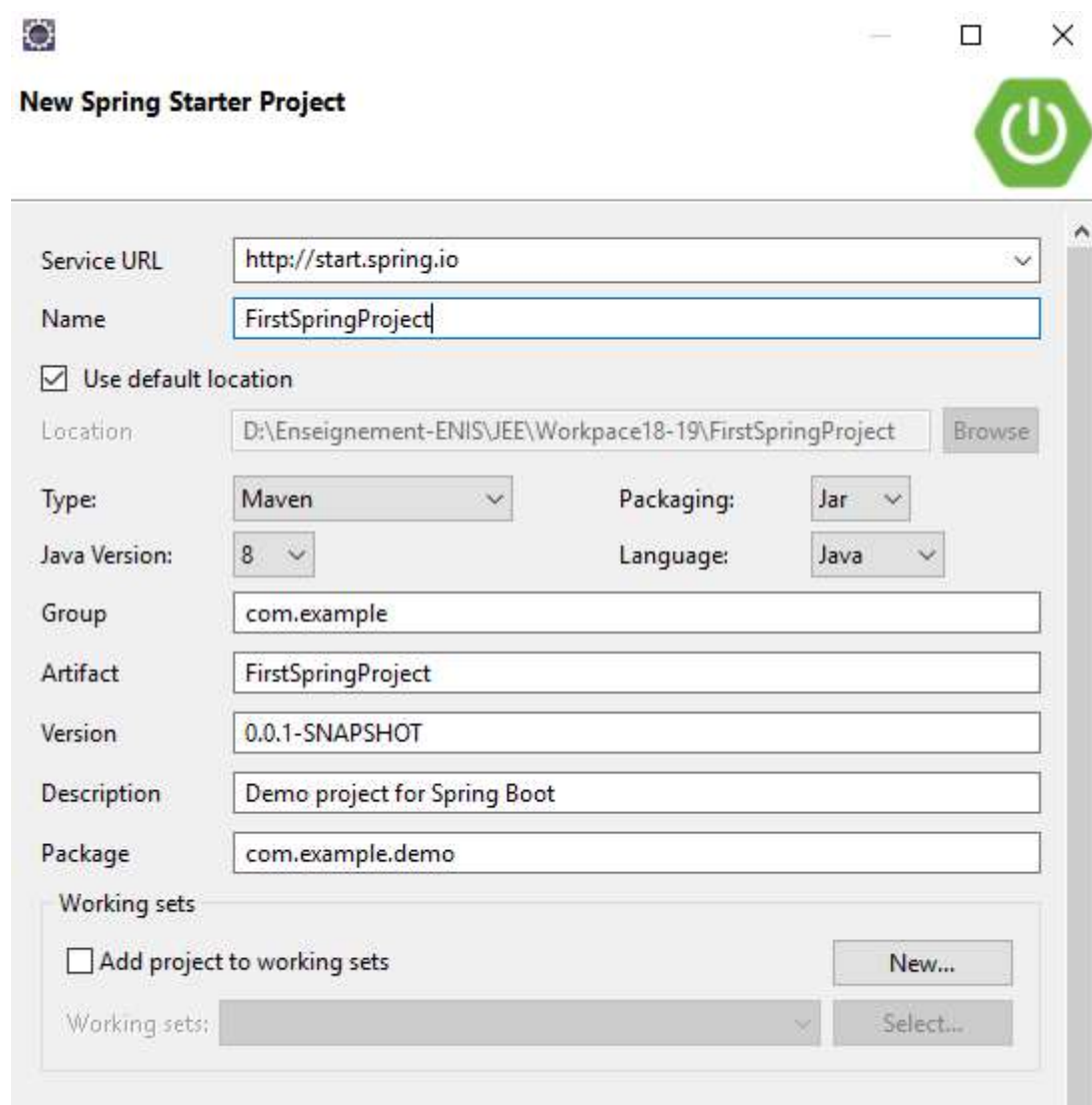
Explore - Ctrl + Space

Share...

# Création d'un projet Spring Boot (2/4)



# Création d'un projet Spring Boot(3/4)



The screenshot shows the 'New Spring Starter Project' dialog box in the Eclipse IDE. The dialog has a title bar with a standard window icon and a green power button icon. The main content area contains several fields and options for configuring a new Spring project.

**Service URL:**

**Name:**

☒ Use default location

**Location:**

**Type:**  **Packaging:**

**Java Version:**  **Language:**

**Group:**

**Artifact:**

**Version:**

**Description:**

**Package:**

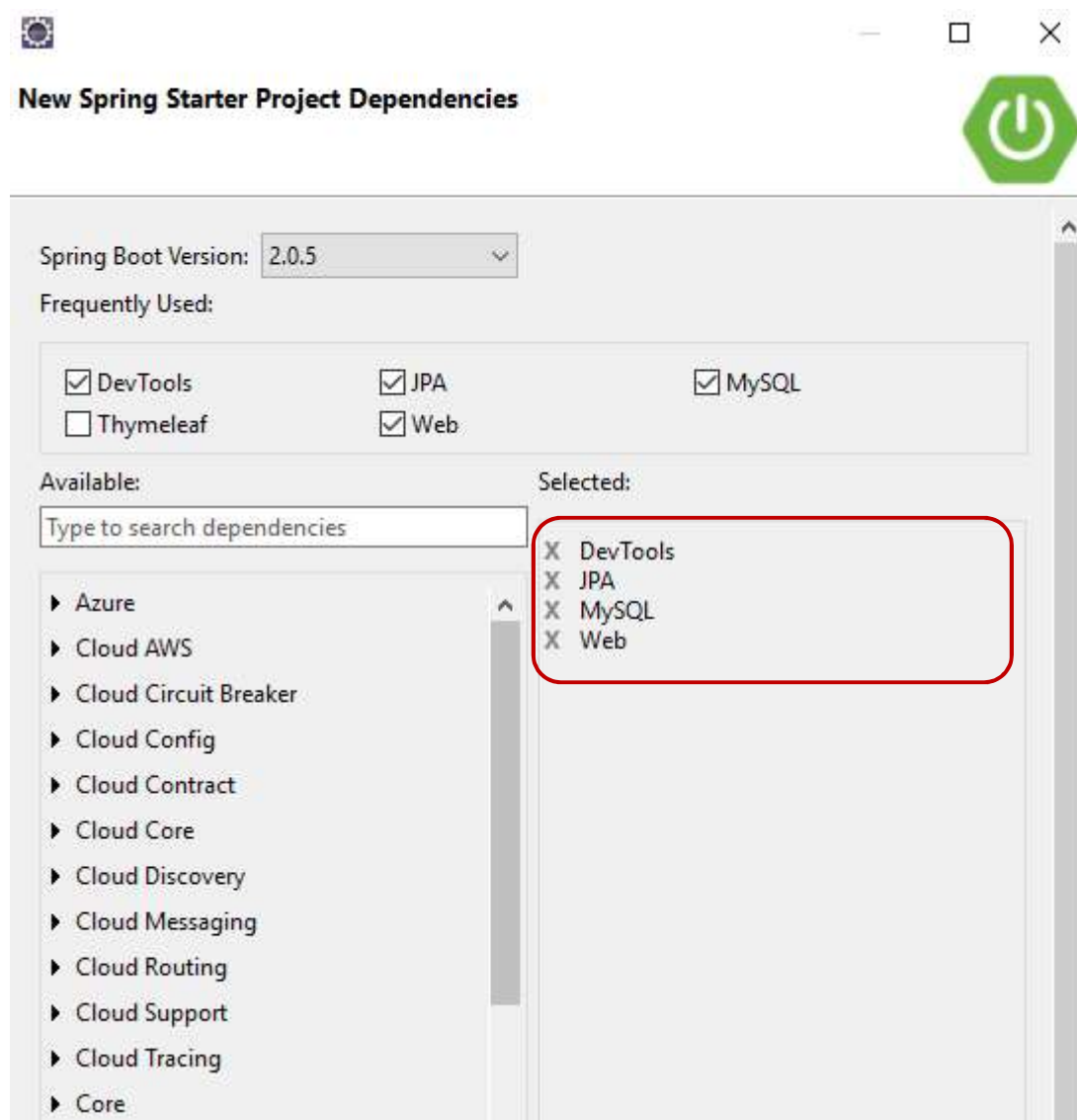
**Working sets:**

☐ Add project to working sets
















Working sets:



# Création d'un projet Spring Boot (4/4)



# Structure d'un projet Spring Boot

- ▼  FirstSpringProject [boot] [devtools]
  - ▼  src/main/java
    - >  com.example.demo
  - ▼  src/main/resources
    -  static
    -  templates
    -  application.properties
  - >  src/test/java
  - >  JRE System Library [JavaSE-1.8]
  - >  Maven Dependencies
  - >  src
  -  target
  -  mvnw
  -  mvnw.cmd
  -  pom.xml

# Outils d'intégration continue : Maven ou Gradle

- Ces outils jouent un rôle important lors de la gestion et l'automatisation de production des logiciels
  - Gérer les dépendances
  - Lancer la compilation des sources
  - Lancer les tests unitaires
  - Générer les packages (war, jar, ear, etc.)
  - Déployer l'application dans le serveur
  - ...

# Le fichier pom.xml

- Voici un extrait du fichier:

```
dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-devtools</artifactId>
<scope>runtime</scope>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<scope>runtime</scope>
</dependency>
...</dependencies>
```

# Le fichier Application.properties

- Rôle : mettre la configuration des sources de données, serveur, etc.
- Exemple:

```
spring.datasource.url=jdbc:mysql://localhost:3306/laboratoire?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driverClassName=com.mysql.cj.jdbc.Driver
spring.jpa.hibernate.ddl-auto=create
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
spring.jpa.show-sql = true
```

# Le point d'entrée d'une application Spring Boot (1/2)

- Une classe annotée par **@SpringBootApplication**

```
@SpringBootApplication
public class FirstSpringProjectApplication {

    public static void main(String[] args) {
        SpringApplication.run(FirstSpringProjectApplication.class, args);
    }
}
```

- **SpringApplication** : la classe de démarrage d'une application Spring et qui va créer une instance de la classe `ApplicationContext`
- **ApplicationContext** : l'interface centrale d'une application Spring permettant de fournir des informations de configuration à l'application.

# Le point d'entrée d'une application Spring Boot(2/2)

- **@SpringBootApplication** est équivalente à ces 3 annotations ensemble
  - @Configuration: déclare la classe comme étant une classe de configuration et indique qu'elle peut contenir des méthodes annotées par @Bean.
  - @EnableAutoConfiguration: activer l'autoconfiguration et générer les configurations nécessaires en fonction des dépendances ajoutées
  - @ComponentScan: chercher les beans dans le package de votre application

# Premier Essai

- Créer un projet Spring Boot en ajoutant **DevTools** comme dépendance
- Modifier le code comme suit :

```
@SpringBootApplication
public class NonWebSpringProjectApplication implements
CommandLineRunner{

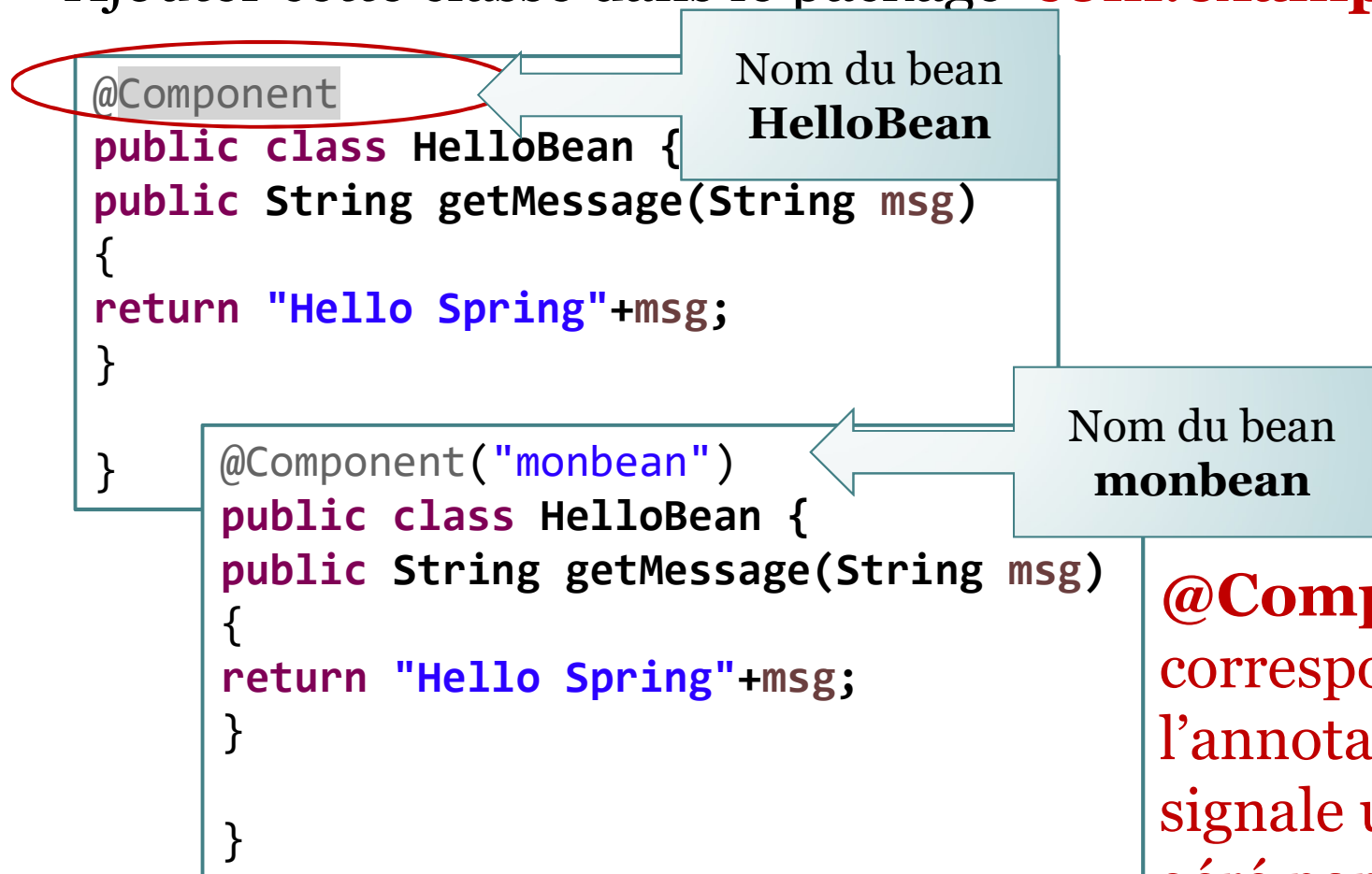
    public static void main(String[] args) {
        SpringApplication.run(NonWebSpringProjectApplication.class, args);
    }
    public void run(String... args) throws Exception {
        System.out.println("Hello Spring");
    }
}
```

- Exécuter l'application aller dans Run As et cliquer sur :
  - Spring Boot App
  - Java Application



## Deuxième Essai

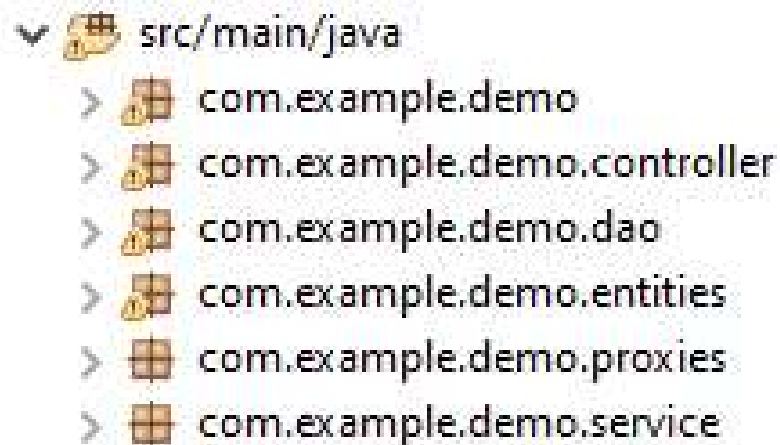
- Ajouter cette classe dans le package **com.example.demo.bean**



**@Component :**  
correspond à  
l'annotation de base qui  
signale un composant  
géré par Spring.

## A retenir

- Le package contenant le point d'entrée de notre application (la classe contenant le public static void main) est **com.example.demo**
- Tous les autres packages bean, dao, model, etc doivent être dans le package demo



## Deuxième Essai

- Injection de dépendance du bean HelloBean avec l'annotation **@Autowired** et appel de méthode dans la classe principale

```
@SpringBootApplication
```

```
public class NonWebSpringProjectApplication implements  
CommandLineRunner{
```

```
@Autowired  
HelloBean bean;
```

```
public static void main(String[] args) {  
    SpringApplication.run(NonWebSpringProjectApplication.class, args);  
}  
public void run(String... args) throws Exception {  
    System.out.println(bean.getMessage(" From ENIS Engineer"));  
}  
}
```

# Troisième Essai

```
public interface IHello {  
    public String getMessage(String msg);  
}
```

```
@Component("eng")  
public class HelloBean implements  
    IHello {  
    public String getMessage(String msg)  
    {  
        return "Hello Spring"+msg;  
    }  
}
```

```
@Component("fr")  
public class BonjourBean implements  
    IHello {  
    public String getMessage(String msg)  
    {  
        return "Bonjour"+msg;  
    }  
}
```

# Troisième Essai

```
@SpringBootApplication
public class NonWebSpringProjectApplication implements
CommandLineRunner{

    @Autowired
    IHello eng;
    public static void main(String[] args) {
        SpringApplication.run(NonWebSpringProjectApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println(eng.getMessage(" From ENIS Engineer "));
    }
}
```

## Quatrième Essai : création d'un projet Web avec Spring Boot

- Créer un projet **Spring starter project** tout en ajoutant la dépendance **Spring Web**
- Faire un clic droit sur le projet et aller dans Run As et cliquer sur Spring Boot App
- La console indique :

```
Starting service [Tomcat]
Starting Servlet engine: [Apache Tomcat/9.0.52]
Initializing Spring embedded WebApplicationContext
Root WebApplicationContext: initialization completed in 3368 ms
Tomcat started on port(s): 8080 (http) with context path ''
Started Essai4Application in 5.696 seconds (JVM running for 7.239)
```
- Allons donc à <http://localhost:8080/>

## Quatrième Essai : exécution du projet

- Résultat : erreur 404



- On a crée un projet web, mais on n'a aucune page HTML ni JSP.
- Spring Boot, comme Spring MVC, implémente le patron de conception MVC 2, donc il nous faut au moins un contrôleur et une vue

## Quatrième Essai : définition d'un contrôleur

- Le contrôleur est une classe Java annoté par Controller ou RestController :
  - reçoit une requête du contrôleur frontal et communique avec le modelé pour préparer et retourner une réponse

```
@Controller
@ResponseBody
public class HelloController {

    @GetMapping(value = "/hello")
    public String sayHello() {
        return "Hello World!";
    }
}
```

**com.example.demo.controller**