Université Nice Sophia Antipolis

Université Nice Sophia Antipolis

Injection de dépendances Framework Spring

Philippe Collet

Master 1 IFI

2016-2017

Plan

- Injection de dépendances, conteneurs légers…
- Spring : introduction au framework
- Configuration XML
- Configuration par annotations
- Configuration par classes

Dépendances entre objets

- Toute application Java est une composition d'objets qui collaborent pour rendre le service attendu
  - Les objets **dépendent** les uns des autres

- Plus les applications sont grandes, plus on crée une architecture globale,
  - avec une organisation en **gros** objets, dépendants entre eux
  - Et de **petits** objets, internes au gros objets, ou servant de données transmises entre eux

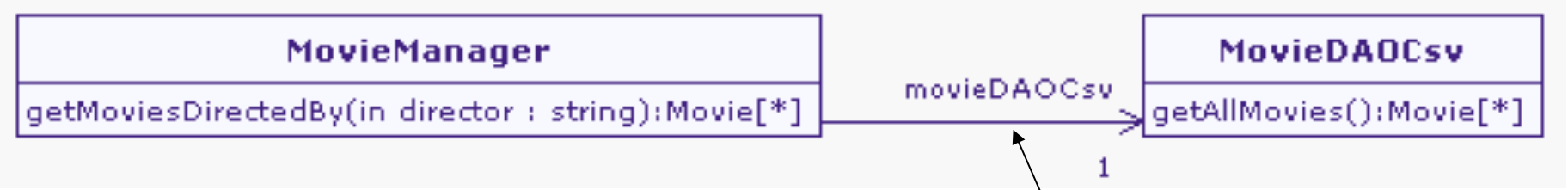
Découplage : rappel

- **Couplage** : degré de dépendance entre objets
- Plusieurs bonnes pratiques pour traiter ce sujet ont été répertoriées comme patrons de conception *Factory*, *Builder*, *Decorator*
  - Gestion de la création des objets
  - Gestion de la résolution des dépendances
  - …
- Mais elles sont implémentées par le développeur de l'application !

Un Exemple

- Basé sur l'article de référence de Martin Fowler sur l'injection de dépendance
  - *Inversion of Control Containers and the Dependency Injection pattern:* http://martinfowler.com/articles/injection.html
- Une classe de service de films `MovieManager` collabore avec une classe d'accès aux films stockés dans un fichier csv `MovieDAOCsv`
- `MovieManager` comporte le seul service de recherche de film sur le nom de son réalisateur `getMoviesByDirector`
- `MovieDAOCsv` comporte une méthode qui retourne la liste de tous les films `getAllMovies`

Approche naïve

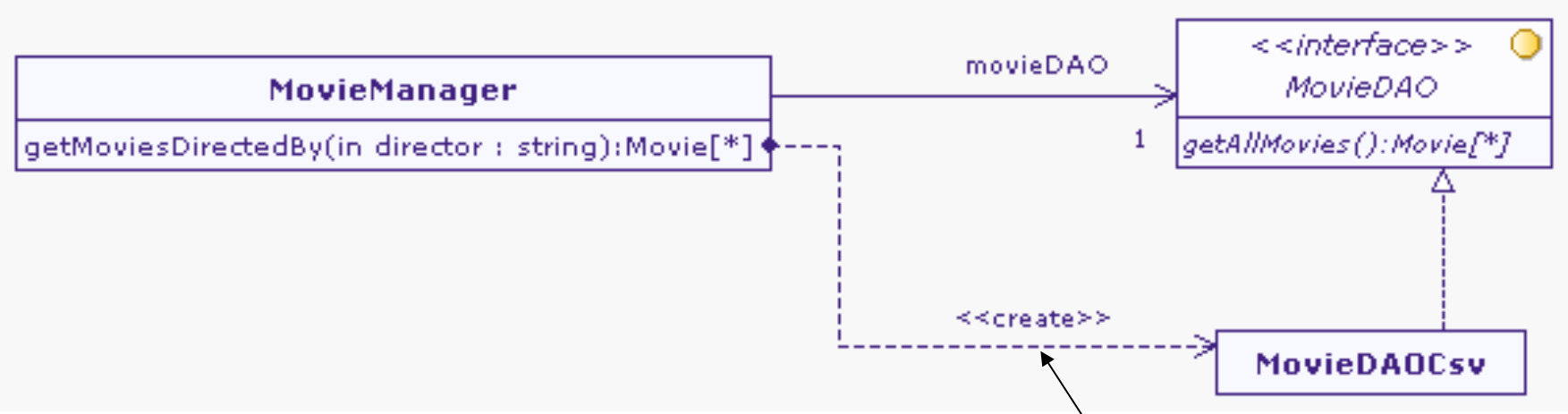



```java
public class MovieManager {

    private MovieDAOCsv movieDAOCsv;

    public MovieManager() {
        movieDAOCsv = new MovieDAOCsv("mymovies.txt");
    }

    public List<Movie> getMoviesDirectedBy(String director) {
        List<Movie> allMovies = movieDAOCsv.getAllMovies();
        // ...
```

Couplage fort

```java
    public static void main(String[] args) {
        MovieManager movieManager = new MovieManager();
        List<Movie> moviesByAllen = movieManager.getMoviesDirectedBy("Allen");
        // ...
```

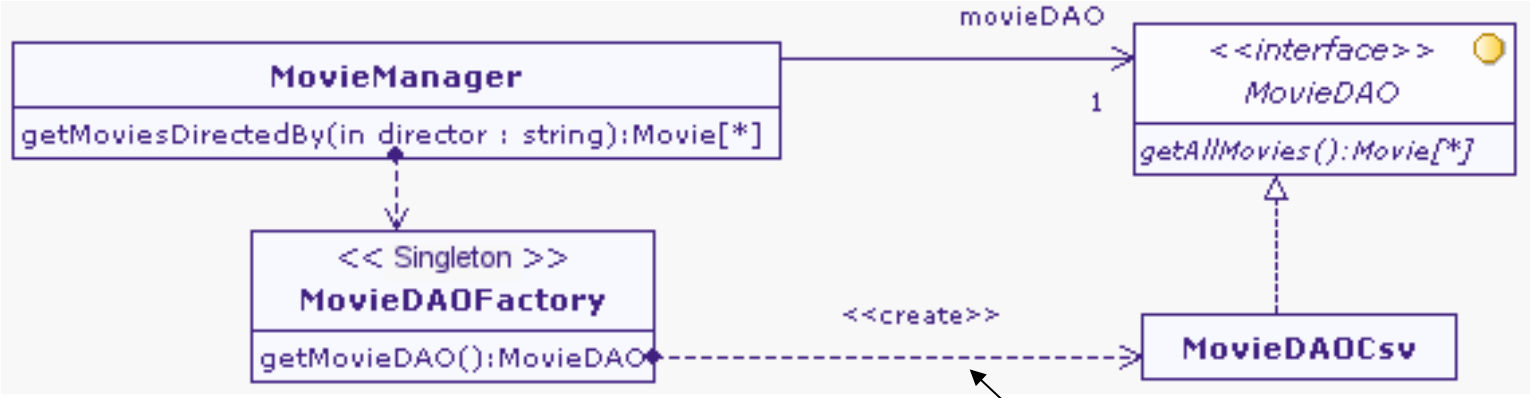
Utilisation d'une interface




```java
public class MovieManager {

    private MovieDAO movieDAO;

    public MovieManager() {
        movieDAO = new MovieDAOCsv("mymovies.txt");
    }


    public List<Movie> getMoviesDirectedBy(String director) {
        List<Movie> allMovies = movieDAO.getAllMovies();
        // ...
```

Couplage plus faible

Utilisation d'une fabrique



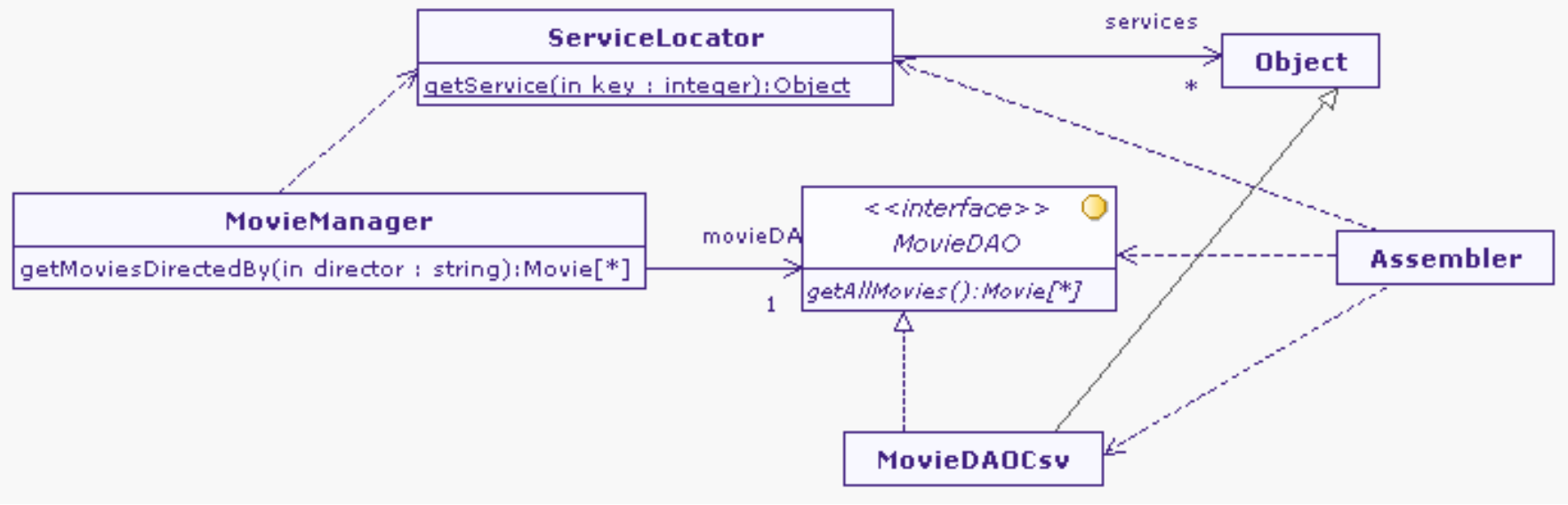

```
public class MovieManager {

    private MovieDAO movieDAO;

    public MovieManager() {
        movieDAO = MovieDAOFactory.getInstance().getMovieDAO();
    }

    public List<Movie> getMoviesDirectedBy(String director) {
        List<Movie> allMovies = movieDAO.getAllMovies();
```

Découplage

```
public class MovieDAOFactory {

    //  Code singleton plus bas ...

    public MovieDAO getMovieDAO() {
        return new MovieDAOCsv("mymovies.txt");
    }
```

Mais la fabrique est une classe de notre application !

Recherche de dépendances



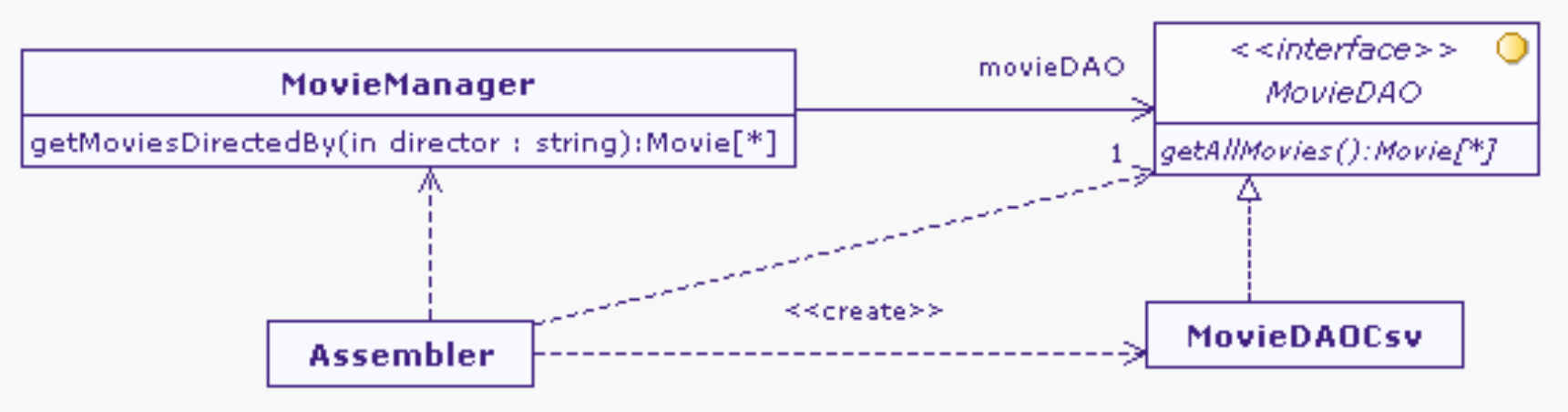
- Principe :
  - un objet extérieur est chargé de créer les instances et les ajouter à une liste de services gérés par un fournisseur en les associant à des clés (chaîne de caractères)
  - l'application peut alors récupérer les instances à partir de la chaîne qui les identifie

Recherche de dépendances

- Dans notre cas :
  - un assembleur crée l'instance du DAO, l'ajoute à la liste des services gérés par `ServiceLocator` en l'associant à la chaîne "`movieDao` »
  - le constructeur de `MovieManager` utilise le `ServiceLocator` pour obtenir une instance de DAO

```
public class MovieManager {

    private MovieDAO movieDAO;

    public MovieManager() {
        movieDAO = (MovieDAO) ServiceLocator.getService("movieDao");
    }
```

Injection de dépendances



- Principe : un objet extérieur est chargé de composer l'application
  - il crée les instances
  - il les injecte dans les classes qui les utilisent

Injection de dépendances

- Dans notre cas:
  - un assembleur crée l'instance du DAO `MovieDAOCsv` et **l'injecte** dans `MovieManager` via un *setter*

```
public class MovieManager {

    private MovieDAO movieDAO;

    public MovieManager() {
    }

    public void setMovieDAO(MovieDAO movieDAO) {
        this.movieDAO = movieDAO;
    }
```

Inversion de contrôle

- Dans cette dernière approche, l'application ne résout pas les dépendances, c'est **l'objet extérieur d'assemblage** qui en a la charge
- Si l'application s'exécute dans un framework, celui est représenté par un conteneur à l'exécution :
  - celui-ci prend le contrôle du flot d'exécution et gère notamment la résolution des dépendances
  - l'application s'insère dans le cadre qu'il fixe
  - Il y a donc **inversion de contrôle**
- Pour une application JavaEE ou Spring, c'est le conteneur qui prend ce contrôle
- Le patron d'architecture **IoC** (Inversion of Control) est commun à de nombreux frameworks de composants

Conteneur IoC

- Raccourci qui désigne un conteneur basé sur l'injection de dépendances
- Dispose d'un noyau d'inversion de contrôle qui instancie et assemble les composants par injection de dépendances

- Plusieurs types d'IoC
  - IoC type 1
    - Injection à travers une interface spécifique
  - IoC type 2 :
    - Injection via les mutateurs
  - IoC type 3 :
    - Injection via les constructeurs

Conteneur IoC de Spring

- IoC type 2 et 3

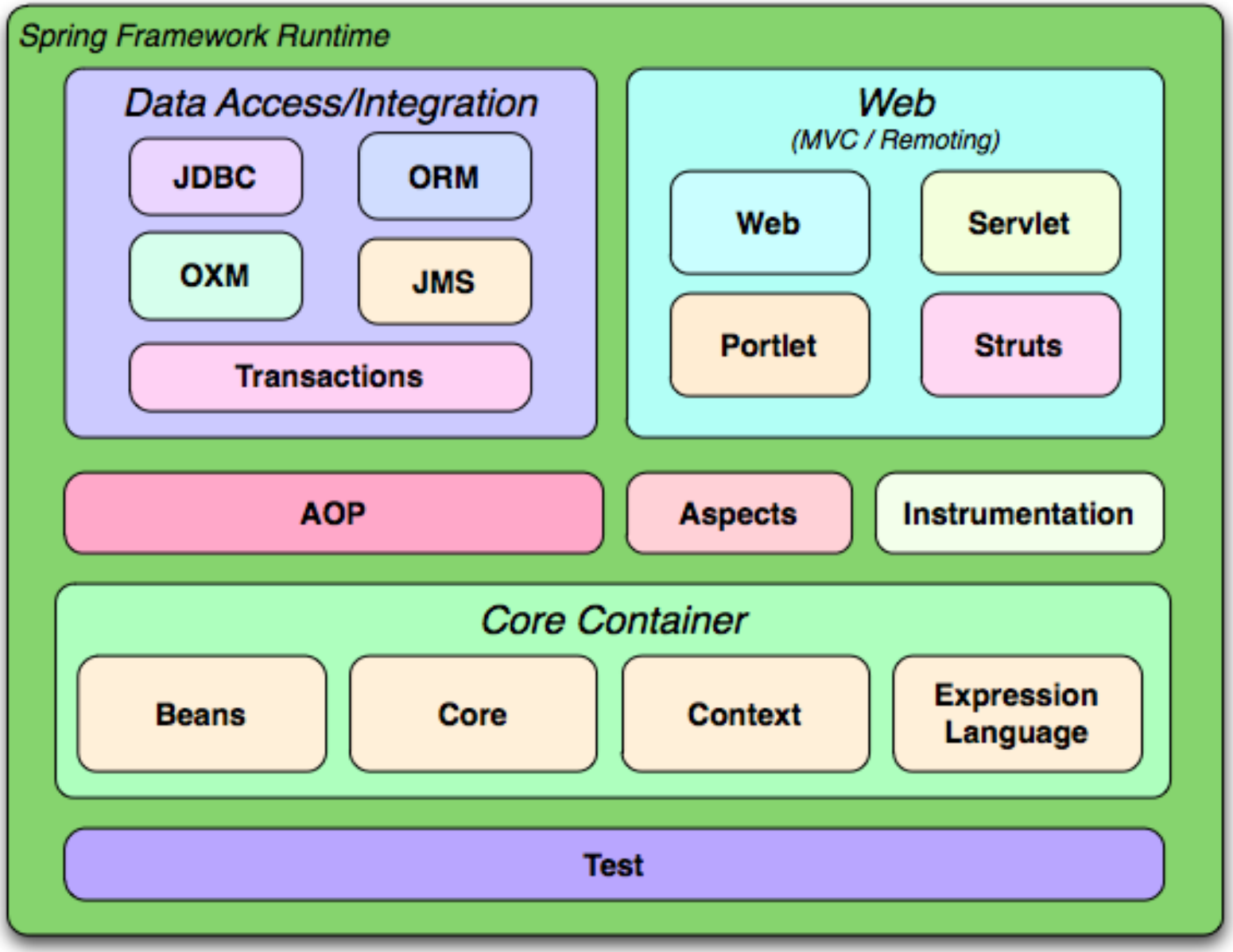
- Le conteneur gère des **beans**
- Un **bean** est un simple objet (*POJO: Plain Old Java Object*) qui respecte la convention de nommage des mutateurs (`setNomAttribut`)

- Le conteneur utilise des méta-données de configuration pour instancier, configurer et assembler les beans
- La configuration peut se faire par fichier XML, annotations dans le code Java, et classes de configuration

Spring

- Framework Open Source qui fournit une solution légère pour la construction de grandes applications Java
  - Simplifie et structure les développements
  - Favorise les bonnes pratiques
    - utilisation des patrons de conception
    - modularité
    - découplage
  - Facilite la mise en œuvre des tests

Vue d'ensemble - modules



Historique

• Août 2003	Spring 1.0 M1
• Mars 2004	Spring 1.0
• Septembre 2004	Spring 1.1
• Mai 2005	Spring 1.2
• Octobre 2006	Spring 2.0
• Novembre 2007	Spring 2.5
• Décembre 2009	Spring 3.0
• Décembre 2011	Spring 3.1
• Décembre 2012	Spring 3.2
• Décembre 2013	Spring 4.0
• Juillet 2015	Spring 4.2
• Juin 2016	Spring 4.3
• Septembre 2017	Spring 5.0

Résumé

- Spring est un conteneur dit « léger » qui peut prendre en charge:
  - l'instanciation d'objets et leurs référencements
  - la gestion transactionnelle
  - une politique de sécurité
- Spring joue aussi un rôle transverse dans l'intégration
  - de frameworks d'interface Web
  - de Web Services ou micro-services
  - de frameworks de persistance
  - avec des systèmes externes
- Spring peut faciliter la mise en œuvre de tests
- Spring est peu intrusif

Projets connexes

- Spring IO : dépendances cohérentes pour le framework, construction d'applications
- SpringBoot : kit de développement simplifié pour le développement
- Spring Cloud : patterns de distribution, implémentation de micro-services
- Spring Cloud Data Flow : orchestration de micro-services
- Spring Data : couche d'accès aux données (BD relationelles ou pas, map-reduce…)
- Spring Security : support d'authentification et d'autorisation
- ...

http://www.springsource.org/projects

Librairies Spring

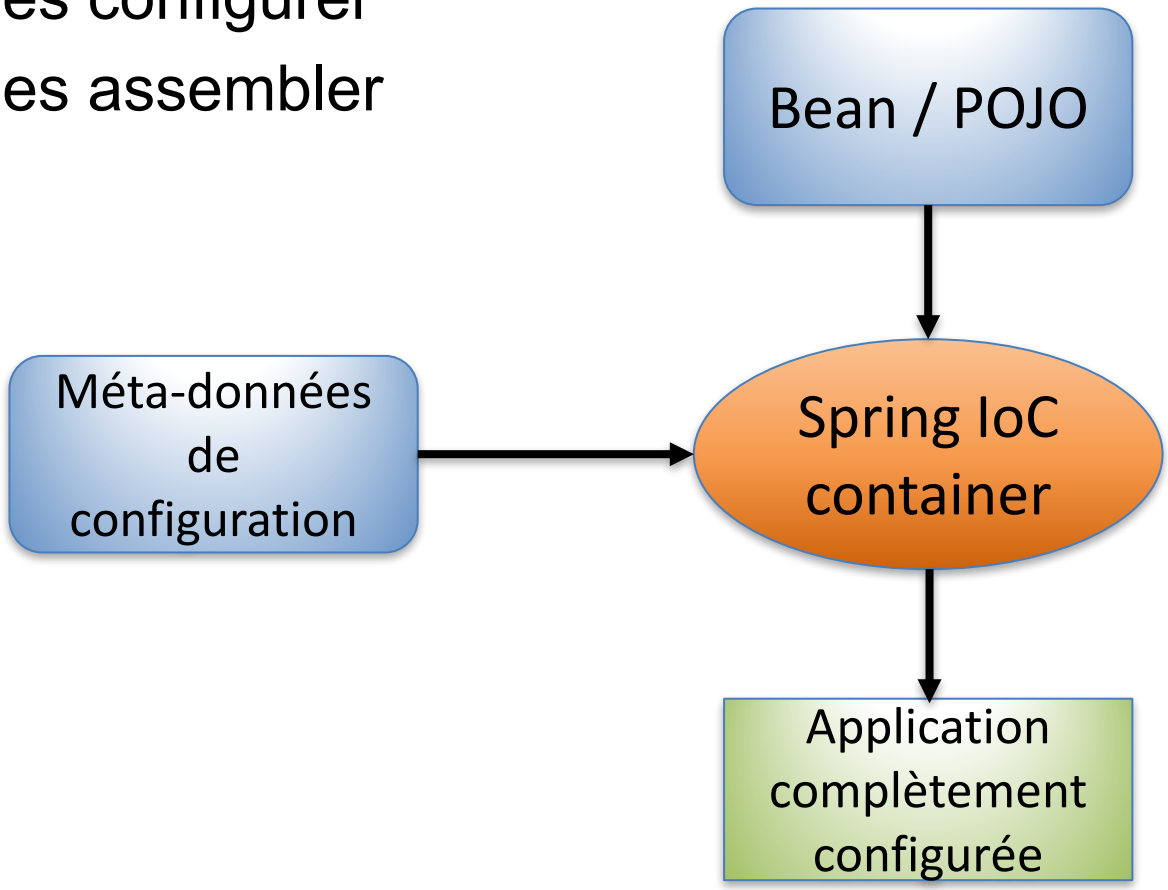
- Gérées par dépendances Maven
  - Beaucoup de dépendances à placer

- Ou par une seule dépendance à SpringBoot

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.4.2.RELEASE</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
```

Méta-données de configuration

- Les méta-données permettent à Spring
  - d'instancier les beans
  - de les configurer
  - de les assembler



Fichier de configuration XML

- Nom par défaut : `applicationContext.xml`
  - Déclaration du schéma utilisé : balises utilisables
  - Déclaration de bean Spring

```
<?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>
```

Déclaration du schéma utilisé

Configuration des beans Spring

Instanciation du conteneur

- Instanciation programmatique ou déclarative
  - Application Java : utilisation de l'API Spring

```
String [] confFiles=new String [] {"applicationContext.xml"};
ApplicationContext context=
    new ClassPathXmlApplicationContext(confFiles);
```

  - Application Web : configuration dans le fichier web.xml

```
<context-param>
   <param-name>contextConfigLocation</param-name>
   <param-value>/WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
   <listener-class>
         org.springframework.web.context.ContextLoaderListener
   </listener-class>
</listener>
```

Beans

- Le conteneur gère des beans
  - Création à partir des méta-données
- Un bean est défini notamment par
  - un (ou plusieurs) identifiant(s) unique(s)
    - Unicité au sein du conteneur
  - un nom complet de classe
    - Ex : `org.spring.hellospring.HelloWorld`
  - un comportement dans le conteneur
    - Notion de scope, cycle de vie, …
  - des références vers ses dépendances

Nommage des beans

- Dans le conteneur, un bean possède des identifiants
  - un identifiant unique avec l'attribut `id`
  - des alias avec l'attribut `name`
- Le nommage n'est pas obligatoire
  - Dans ce cas, le conteneur génère un nom unique
- Par convention, on utilise les conventions Java à partir du nom de la classe :
  - Ex : `userDAO`, `authenticationService`, ...

Instanciation des beans

- La déclaration d'un bean induit la méthode d'instanciation (constructeur / fabrique)
- Il peut être instancié via son constructeur

```
<bean id="service" class="com.spring.formation.ExampleService" >
    <constructor-arg value="chaine"/>
</bean>
```

- Utilisation du type ou de l'index des arguments

```
<bean id="service1" class="com.spring.formation.ExampleService" >
    <constructor-arg type="java.lang.String" value="chaine"/>
    <constructor-arg type="int" value="2"/>
</bean>

<bean id="service2" class="com.spring.formation.ExampleService" >
    <constructor-arg index="1" value="2"/>
    <constructor-arg index="0" value="chaine"/>
</bean>
```

Injection des dépendances

- par **constructeur**

```java
public class Director {

    private String name;

    public Director(String name) {
        this.name = name;
    }
```

```java
public class Movie {

    private Director director;
    private String name;
    private short year;

    public Movie(String name, Director director, short year) {
        this.director = director;
        this.name = name;
        this.year = year;
    }
```

Résolution des arguments

- sans ambiguïté
- grâce au type
- selon l'index

Passage

- par valeur
- par référence

```xml
<bean id="director" class="sample.Director">
      <constructor-arg value="Allen"/>
</bean>

<bean id="movie" class="sample.Movie">
      <constructor-arg type="java.lang.String" value="Annie Hall"/>
      <constructor-arg ref="director"/>
      <constructor-arg index="2" value="1977"/>
</bean>
```

Injection des dépendances

- par **mutateur**

```
<bean id="director" class="sample.Director">
    <constructor-arg value="Allen"/>
</bean>

<bean id="movie" class="sample.Movie">
    <property name="name" value="Annie Hall"/>
    <property name="director" ref="director"/>
</bean>
```

```
public class Movie {

    private Director director;
    private String name;

    public Movie() {
    }

    public void setDirector(Director director) {
        this.director = director;
    }

    public void setName(String name) {
        this.name = name;
    }
```

Portée d'un bean (scope)

- Définit la stratégie de création et de stockage d'instances des beans

- `singleton` : 1 instance par conteneur (défaut)
- `prototype` : 1 instance par récupération du bean
- `request` : 1 instance par requête http
- `session` : 1 instance par session http
- `global` : 1 instance par session http globale

Configuration par annotations

- Annotations compatibles Java 5 et plus
- Objectifs
  - Simplifier la configuration (***convention over configuration***)
  - Limiter les fichiers de configuration XML
- Annotations dans Spring
  - Annotations spécifiques à des couches d'architecture
    - Présentation / Service / DAO (Data Access…)

Configuration XML pour l'injection

- Gestion des dépendances par annotations

```
<?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 />

  <bean id="..." class="..." />

</be
```

Balise pour « activer » les annotations

Inclusion du namespace context

@Autowired ou @Inject

- Injection automatique des dépendances
- 2 annotations pour la même fonction
  - @Autowired : annotation Spring
  - @Inject : annotation JSR-330
- Spring supporte les 2 annotations
  - @Autowired dispose d'une propriété (required) que n'a pas l'annotation @Inject. Ceci est utile pour indiquer le caractère facultatif d'une injection.
- Résolution par type
  - Se base sur le type de l'objet pour retrouver la dépendance à injecter.

@Autowired sur un attribut

- Utilisation de @Autowired pour injecter helloWorld

```
import org.springframework.beans.factory.annotation.Autowired;
public class HelloWorldCaller {

   @Autowired
   // Le nom de l'attribut importe peu, seul son type compte
   private HelloWorld helloWorldService;

   public void callHelloWorld(){
           helloWorldService.sayHello();
   }
}
```

@Autowired sur un mutateur

- Utilisation de @Autowired pour injecter helloWorld

```
import org.springframework.beans.factory.annotation.Autowired;
public class HelloWorldCaller {
    private HelloWorld helloWorldService;

    @Autowired
    public void setHelloWorldService(HelloWorld helloWorldService){
            // Le nom de la méthode setter importe peu. Seul le
        // type du paramètre compte
            this.helloWorldService=helloWorldService;
    }
    public void callHelloWorld(){
            helloWorldService.sayHello();
    }
}
```

Utilisation de @Autowired

- Position de l'annotation
  - devant un attribut
  - devant une méthode (« init-method », constucteur, setters)
- Vérification de dépendance
  - Par défaut @Autowired vérifie les dépendances
  - Une exception est générée si aucun bean n'est trouvé

@Resource

- Injection à partir de l'identifiant du bean
- Nécessite Java 6 ou plus
- Se place devant un attribut ou un setter
- Par défaut, le nom est déduit du nom de la méthode (setter) ou de l'attribut

```
package com.spring.formation;

import javax.annotation.Resource;

public class MovieManager {

    @Resource(name="myMovieDAO")
    private MovieDAO movieDAO;

    public MovieManager() {}

    public void setMovieDAO(MovieDAO movieDAO) {
        this.movieDAO = movieDAO;
    }
}
```

@Value

- Injection d'une valeur dans une propriété d'un bean
- Se place devant un attribut, un setter, un paramètre de constructeur
- Exemple
  - Attribut

```
public class MovieDAOCsv implements MovieDAO {

  @Value("mymovies.txt")
  private String filename;
```

  - Constructeur

```
public class MovieDAOCsv implements MovieDAO {

  private String filename;

  @Autowired
  public MovieDAOCsv(@Value("mymovies.txt") String filename) {
    this.filename = filename;
  }
```

Callbacks du cycle de vie

- Annotations `@PostContruct` et `@PostDestroy` : même principe qu'en XML
  - Appel des méthodes à l'instanciation/la destruction du bean

```
public class ExempleService {

  @PostConstruct
  public void init() {
    // Appelée à l'initialisation du bean
    ...
  }

  @PreDestroy
  public void destroy() {
    // Appelée avant destruction du bean
    // Ex: libération des ressources
    ...
  }

}
```

@Required

- Cas particulier : ne fait pas d'injection
- À utiliser en // de la configuration XML
- Spécifie que la propriété DOIT être injectée au runtime
- Se place devant les setters des propriétés
- Permet de ne pas « oublier » une `<property>`

```
package com.spring.formation;

import org.springframework.beans.factory.annotation.Required;

public class MovieManager {

    private MovieDAO movieDAO;

    public MovieManager() {
    }

    @Required
    public void setMovieDAO(MovieDAO movieDAO) {
        this.movieDAO = movieDAO;
    }
}
```

Import

Annotation

Configuration par annotation

- Précédentes annotations uniquement pour l'injection
- Besoin d'annotations équivalentes à la balise <bean>

- Spring peut détecter automatiquement ces beans
  - Balise XML `<component-scan>`
- Plusieurs types de composants => « Stéréotype »
  - Notions de composants (« components »)

- Possibilité de mixer configuration par annotations / par XML / par classe

@Component

- Définit une classe POJO en tant que bean
  - Équivalent à la balise <bean>
- L'identifiant du bean peut être spécifié
  - Si ce n'est pas le cas, il est déduit du nom de la classe
- Alternative : @Named (JSR-330)
  - Même comportement que @Component

```
import org.springframework.stereotype.Component;

@Component("monComposant")
public class MyComponent {

    [...]

}
```

Configuration pour charger les beans

- Définition des beans par annotations
- Ajout automatique des beans dans le conteneur

```
<?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 />

    <context:component-scan base-package="com.spring.formation" />

</beans>
```

Balise pour « activer » la détection des classes « annotées » dans un package donné

Stéréotype selon l'usage

- `@Component` : composant générique
- `@Service` : Services métier
- `@Repository` : accès aux données (DAO)
- `@Controller` : contrôleur pour interface graphique (Web)

@Scope

- A utiliser conjointement à une annotation @Component ou dérivée

- Définit la portée d'un bean
  - Par défaut (si non présent) : singleton

```
package com.spring.formation;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component("monComposant")
@Scope("prototype")
public class MyComponent {

}
```

@Configuration

- Annotation sur une classe
  - La classe regroupe des informations de configurations et des définitions de Beans
- @Configuration hérite de @Component
  - Utilisation possible d'autowiring pour les attributs
- Les méthodes annotées @Bean représentent des définitions de composants

```
@Configuration
public class AppConfig {

  @Bean
  public MovieDAO movieDao() {
    return new MovieDAOCsv("movies.txt");
  }

}
```

Chargement du contexte

- Création d'un contexte spécifique aux configurations d'annotations :

```
public class MainApp {

  public static void main(String[] args) {
    ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
    MovieDAO mDAO = (MovieDAO) context.getBean("movieDAO");

    mDAO…
  }
```

@Configuration

- @Import : importation d'une autre classe de configuration

```
@Configuration
public class DaoConfig {

  @Bean
  public MovieDAO movieDao() {
    return new MovieDAOCsv("movies.txt");
  }
}
```

```
@Configuration
@Import(DaoConfig.class)
public class ManagerConfig {

  @Autowired MovieDAO movieDao;

  @Bean
  MovieManager movieManager() {
    return new MovieManagerImpl(movieDao);
  }
}
```

- @ImportResource : importation d'une configuration XML

```
@Configuration
@ImportResource("classpath:conf/dao-context.xml")
public class ManagerConfig {

  @Autowired MovieDAO movieDao;

  @Bean
  MovieManager movieManager() {
    return new MovieManagerImpl(movieDao);
  }
}
```

Enrichissement de @Configuration (Spring framework 4)

- Détection automatique de Beans : *@ComponentScan*

```
@Configuration
@ComponentScan(basePackages = { "com.formation.spring.service" })
public class AppConfig {
```

- Auto-détection sur tout le classpath : @EnableAutoConfiguration

```
@Configuration
@EnableAutoConfiguration(exclude={MovieDAOJdbc.class})
public class AppConfig {
```

Encore plus facile en SpringBoot

- @SpringBootApplication

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
// same as @Configuration @EnableAutoConfiguration @ComponentScan
public class Application {

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

D'autres choses intéressantes dans Spring

- Support pour le test unitaire (Runner Junit spécifique, mocking de contexte, etc.)
- Support pour le Web:
  - Pattern MVC (Model-View-Controller) avec composants @Controller
  - Facilité de production de Web Services (XML, JSON) et de micro-services
- Support pour les BDs et les transactions
  - Spring Data : JPA + Transactions