



# Spring Core

---

David THIBAU - 2021

david.thibau@gmail.com



# Agenda

---

- Introduction
  - Le framework Spring
  - Pattern Ioc et Injection de dépendance
  - Autres patterns de Spring
  - Le conteneur Spring
- Configuration via annotations
  - Les classes *@Configuration*
  - L'annotation *@Component* et autre stéréotypes
  - L'annotation *@Autowired*
- Spring Boot
  - Introduction
  - Développement avec Spring Boot
  - Propriétés de configuration
  - Profils



# Introduction

---

## **Le framework Spring**

Le pattern IoC et l'injection de  
dépendance

Les autres design patterns de Spring

Le conteneur Spring



# Historique et version

---

- ❖ Spring est un projet **OpenSource** supporté par la joint-venture **Pivotal Software** (VMWare, ...)
- ❖ Rod Johnson et Jorgen Holler ont démarré le projet en 2002 comme une alternative à la spécification J2EE supporté par Sun puis Oracle.
- ❖ Actuellement, c'est le framework Java le plus utilisé !!



# Projets Spring

---

*Spring* est en fait un ensemble de projets adaptés à toutes les problématiques actuelles basé sur la même fondation : ***Spring Core***.

Tous ces projets ont comme objectifs :

- ✓ Permettre d'écrire du code propre, modulaire et testable
- ✓ Éviter d'avoir à coder les aspects techniques (intégration aux autres systèmes)
- ✓ Être portable : Nécessite juste une JVM



# Principaux projets

---

***Spring core*** : Les fondations. Repose sur le pattern IoC, services de bas niveau

***Spring Security*** : Tout ce qui est nécessaire pour sécuriser une application (web) java

***Spring Data*** : Approche commune pour persister des données (SQL, NOSQL)

***Spring Integration*** : Comment faire communiquer des applications legacy

***Spring Batch*** : Faciliter et optimiser les batchs

***Spring Cloud*** : Architecture micro-services déployés sur Cloud public/privé

...



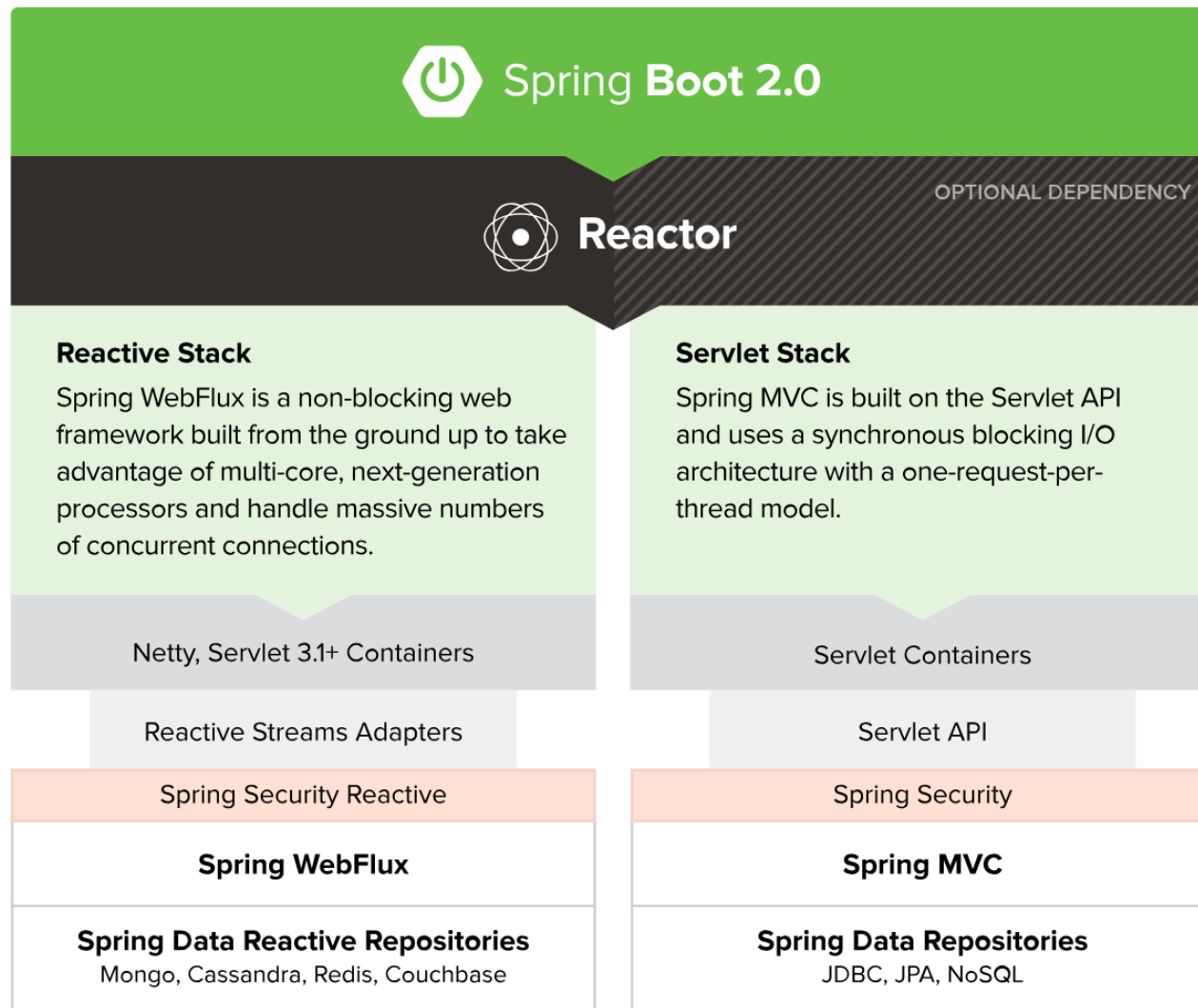
# Usage Spring

---

Le principal usage de Spring est de construire des applications web

- Traditionnelles : Les pages HTML sont générés côté serveur
- Modernes : UI est construit avec des frameworks Javascript (Angular, ReactJS, ...)  
Spring ne fournit que l'API REST backend

# Spring stacks







# Pattern IoC et injection de dépendances



# Pattern IoC

## *Inversion Of Control*

---

❖ Le problème :

*Comment faire fonctionner la couche contrôleur (HTTP) avec la base de données alors que ces 2 couches sont développées par des équipes différentes ?*

❖ La réponse du paradigme Objet :

*Utiliser des interfaces !*



# Illustration

---

- ❖ Dans un contrôleur web, nous voulons fournir une méthode qui liste toutes les films d'un metteur en scène particulier.
- ❖ Cette méthode s'appuie sur une couche DAO (Data Access Object), qui fournit un méthode permettant de récupérer tous les films de la BD : *MovieLister* :

```
public class MovieLister...  
    public List<Movie> moviesDirectedBy(String arg) {  
        List<Movie> allMovies = finder.findAll();  
        List<Movie> ret = new ArrayList<Movie>() ;  
        for (Movie movie : allMovies ) {  
            if (!movie.getDirector().equals(arg))  
                ret.add(movie);  
        }  
        return ret;  
    }  
}
```



# Leçon apprise : Utiliser des interfaces !

---

- ❖ Comme nous voulons que notre méthode soit indépendant de la façon dont stocker les films, nous nous basons sur une interface qui définit la méthode dont on a besoin :

```
public interface MovieFinder {  
    List<Movie> findAll();  
}
```



# Implémentation ?

---

- ❖ Même si le code est bien découplé via l'utilisation d'interface, comment peut-on insérer une classe concrète qui implémente l'interface *MovieFinder* ?

Par exemple, dans le constructeur de la classe *MovieLister*.

```
class MovieLister...  
    private MovieFinder finder;  
    public MovieLister() {  
        finder = new ColonDelimitedMovieFinder("movies1.txt");  
    }
```

=> Argh !!! Le contrôleur est alors dépendant de l'implémentation !!

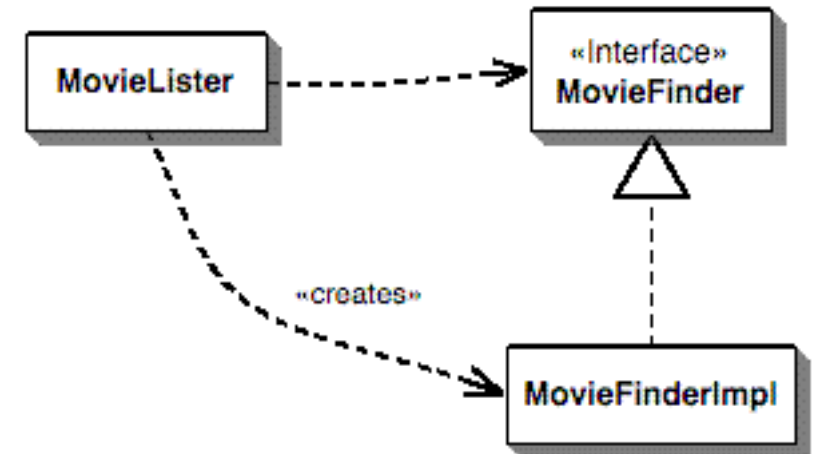
# Solution

=> La classe *MovieLister* est dépendante de l'interface **et** de l'implémentation !!

L'objectif était de ne dépendre que de l'interface.

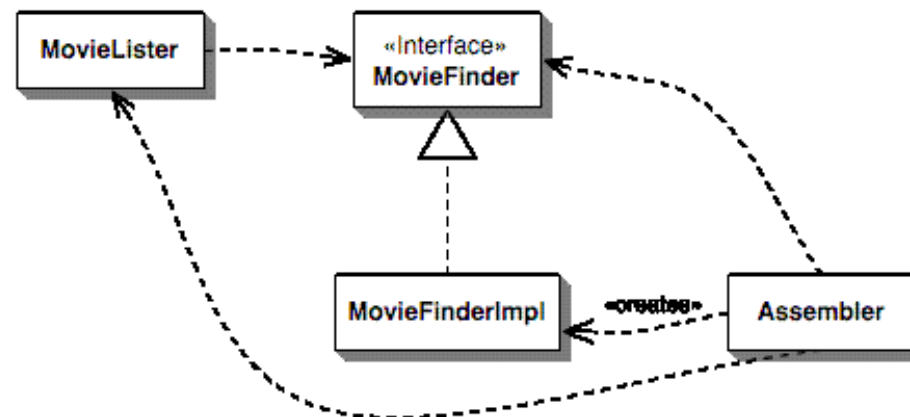
Alors, comment spécifier l'implémentation à instancier ?

=> Solution : Deleguer l'instanciation au framework :  
**Pattern IoC**



# Injection de dépendance

- ❖ ***L'injection de dépendance*** est juste une spécialisation du pattern IoC
- ❖ Le framework instantie les objets ET initialise ces attributs
- ❖ Dans l'exemple précédent, il initialise l'attribut *MovieLister* avec une implémentation.





# Types d'injection de dépendances

---

❖ Il y a 3 principaux types d'injections de dépendances :

– Par constructeur

```
public MovieLister(MovieFinder finder) {  
    this.finder = finder;  
}
```

– Par setter

```
public void setFinder(MovieFinder finder) {  
    this.finder = finder;  
}
```

– Par interface :

```
public interface InjectFinder {  
    void injectFinder(MovieFinder finder);  
}
```





# Configuration

---

- ❖ La configuration du framework (conteneur) instanciant les objets (beans) consiste à spécifier quelle implémentation à injecter dans chaque objet.
- ❖ Cela est effectué via :
  - Un fichier externe (XML)
  - Une classe Java de configuration
  - Des annotations dans le code



# Example XML

---

```
<beans>
```

```
  <bean id="MovieLister" class="spring.MovieLister">
```

```
    <property name="finder">
```

```
      <ref local="MovieFinder"/>
```

```
    </property>
```

```
  </bean>
```

```
  <bean id="MovieFinder" class="spring.ColonMovieFinder">
```

```
    <property name="filename">
```

```
      <value>movies1.txt</value>
```

```
    </property>
```

```
  </bean>
```

```
</beans>
```



# Test

---

```
public void testWithSpring()
    throws Exception {

    ApplicationContext ctx =
        new FileSystemXmlApplicationContext("spring.xml");

    MovieLister lister = (MovieLister)ctx.getBean("MovieLister");

    Movie[] movies = lister.moviesDirectedBy("Sergio Leone");
    assertEquals("Once Upon a Time in the West",movies[0].getTitle());
}
```



# Avantages de l'injection de dépendances

---

- ❖ L'injection de dépendance apporte d'importants bénéfices :
  - Les composants applicatifs sont **plus facile à écrire**
  - Les composants sont **plus faciles à tester**. Il suffit d'instancier les objets collaboratifs et de les injecter dans les propriétés de la classe à tester dans les méthodes de test.
  - Le **typage** des objets est **préservé**.
  - Les **dépendances sont explicites** (à la différence d'une initialisation à partir d'un fichier *properties* ou d'une base de données)



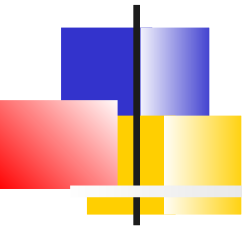
# Services techniques

---

De plus, au moment de l'instanciation des beans, le framework peut leur ajouter des capacités (aspects)

Avec un framework IoC comme Spring, un développeur peut :

- Écrire une méthode s'exécutant dans une transaction base de données sans utiliser l'API de transaction
- Rendre une méthode accessible à distance sans utiliser une API remote
- Protéger l'accès d'une méthode
- ...



# Spring et les Design Patterns



# Design patterns

---

Un **patron de conception** (plus souvent appelé **design pattern**) est un arrangement caractéristique de modules ou classes, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel.

- Il décrit une solution standard, utilisable dans la conception de différents logiciels.
- Il est issu de l'expérience des concepteurs de logiciels.
- Il décrit les grandes lignes d'une solution, qui peuvent ensuite être modifiées et adaptées en fonction des besoins.
- Ils ont une influence sur l'architecture logicielle d'un système informatique.



# Introduction

---

Spring utilise de nombreux Design Patterns, citons : loc, la fabrique (Factory), le singleton, l'AOP (Aspect Oriented Programming) et la programmation par modèle (Template).

D'autre part, Spring impose la programmation par contrat à savoir l'usage intensif des interfaces indépendamment des implantations réelles des objets.





# Le Singleton

---

Le modèle de conception ***Singleton*** permet de garantir l'unicité d'un objet au sein d'une JVM. Les buts recherchés sont:

- réduire le temps d'instanciation d'une classe.
- réduire la consommation de mémoire inutile.

La classe Singleton doit fournir la même instance quelque soit le module appelant, elle ne doit pas avoir d'informations d'historique ou de sessions.

# Constructeur privé

Rendre son constructeur privé permet d'assurer que le seul moyen d'obtenir l'instance est de recourir à sa méthode statique *getInstance()* pour obtenir l'instance.

Attention, cependant au multithreading, les méthodes d'un singleton sont stateless

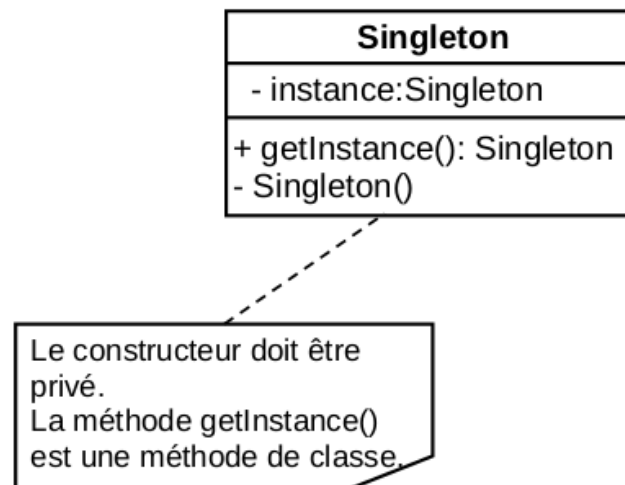


Diagramme du Singleton



# Spring et le Singleton

---

Le framework Spring prend en charge la gestion des objets *Singleton* sans que le développement ait à les mettre en œuvre explicitement.

Dans une application Spring, la plupart des beans applicatifs sont des singletons



# Exemple Factory

---

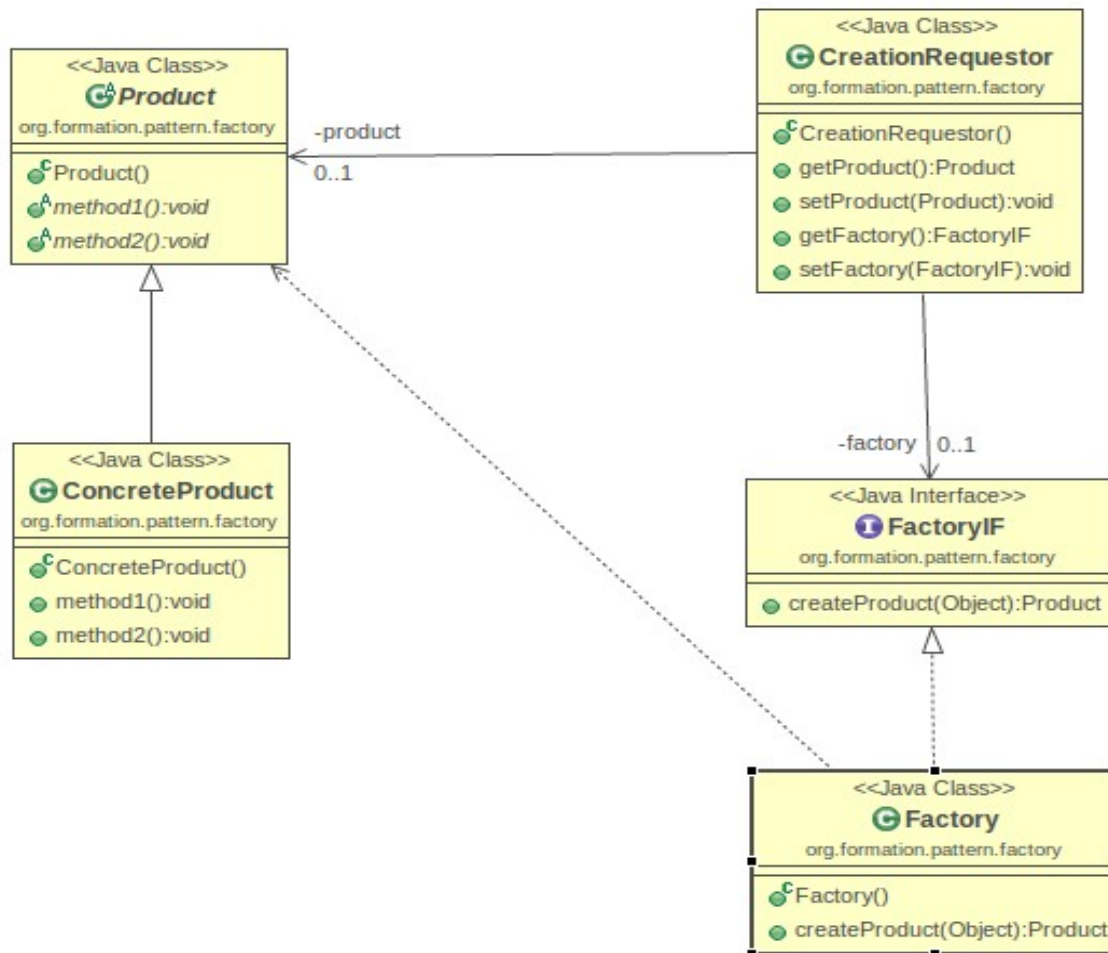
Nom : *Factory Method* [GoF95]

Problème à résoudre : Écriture d'une classe réutilisable avec des types de données arbitraires.

La classe doit pouvoir instancier d'autres classes sans en être dépendantes.

Solution : Elle délègue le choix de la classe à instancier à un autre objet et référence la nouvelle classe créée à travers une interface

# Solution





# Classes impliquées

---

***Product*** : Classe abstraite (ou interface) mère de toutes les implémentations

***Concrete Product*** : Classe concrète instanciée par la fabrique

***Creation Requestor*** : Classe utilisant des produits mais ne dépendant d'aucune implémentation

***Factory Interface*** : Interface déclarant la méthode de création.

***Factory*** : Classe implémentant l'interface *Factory* qui instancie une implémentation spécifique de *Product*



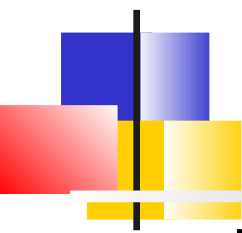
# Conséquences

---

*CreationRequestor* est indépendant des implémentations concrètes créés

L'ensemble des classes concrètes pouvant être instanciées peut changer dynamiquement

Spring est principalement une fabrique à beans. On s'adresse à lui pour récupérer l'objet que l'on veut manipuler



# Programmation par modèle

---

Le principe de ce motif de conception est la séparation de la partie fixe d'un procédé et sa partie variante.

Spring utilise grandement ce motif de conception dans les contextes

- d'accès aux données (JdbcTemplate)
- d'accès à des ressources distantes (RestTemplate)
- ...
-



# Diagramme

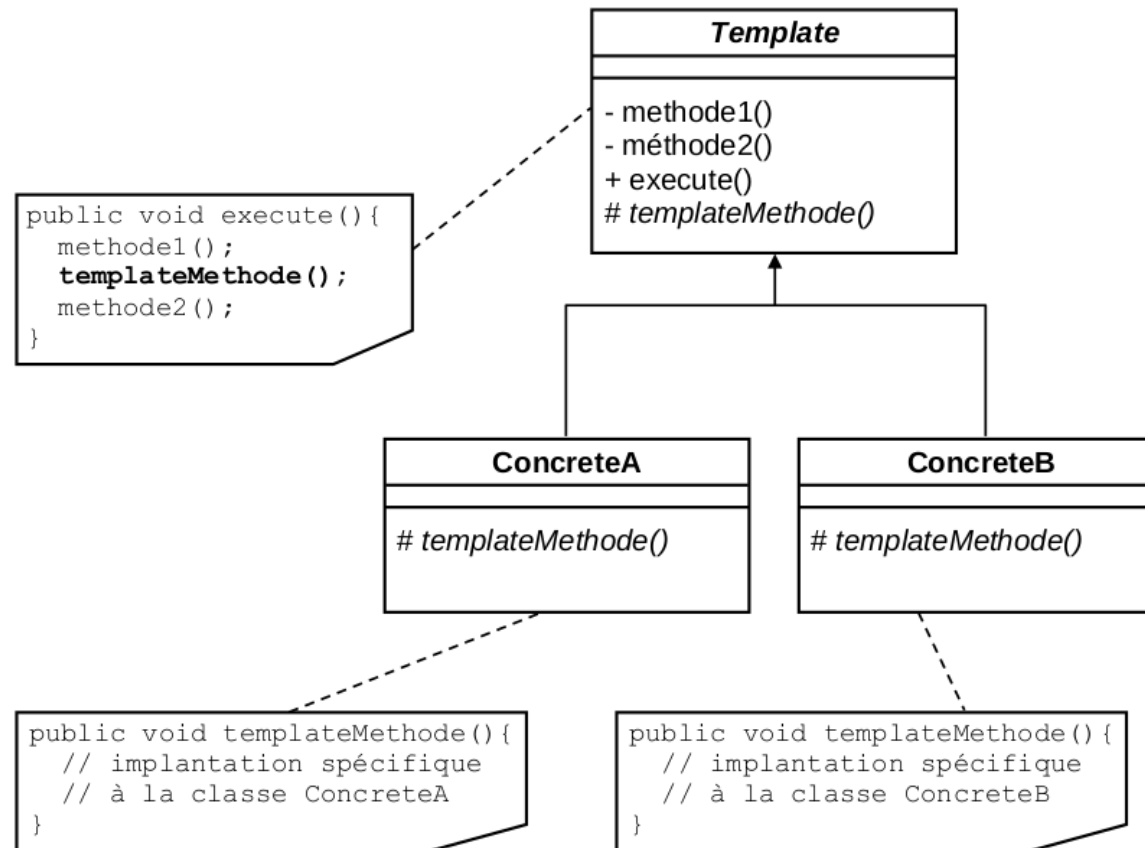


Diagramme du patron



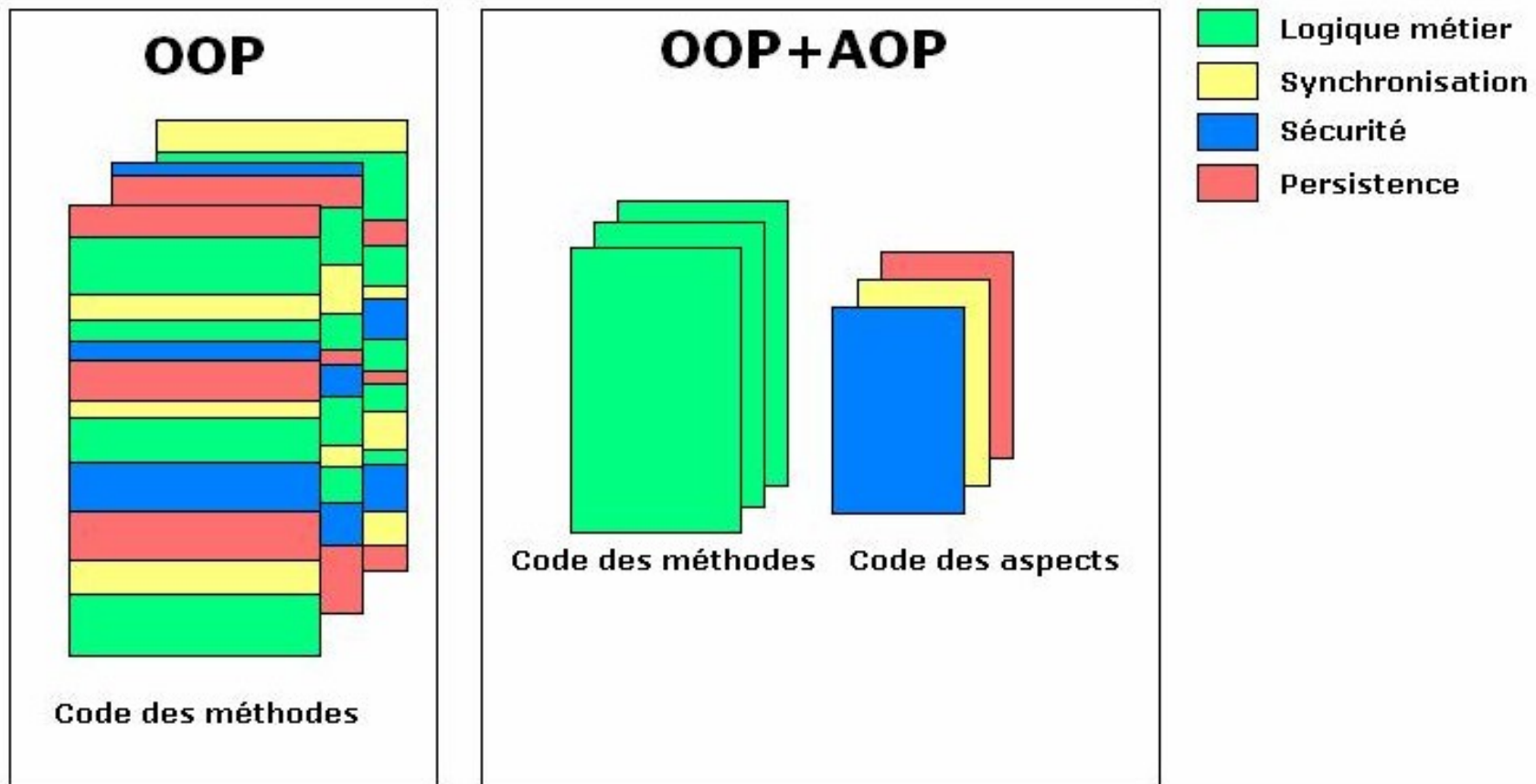
## Diagramme (2)

---

La partie invariante est implantée dans la classe abstraite *Template* par le biais de la méthode ***execute()***;

La partie spécifique est assurée par les diverses implantations de la méthode ***templateMethode()*** dans chaque sous-classe.

# Principes de l'AOP et Cross-cutting concerns





# Terminologie AOP

---

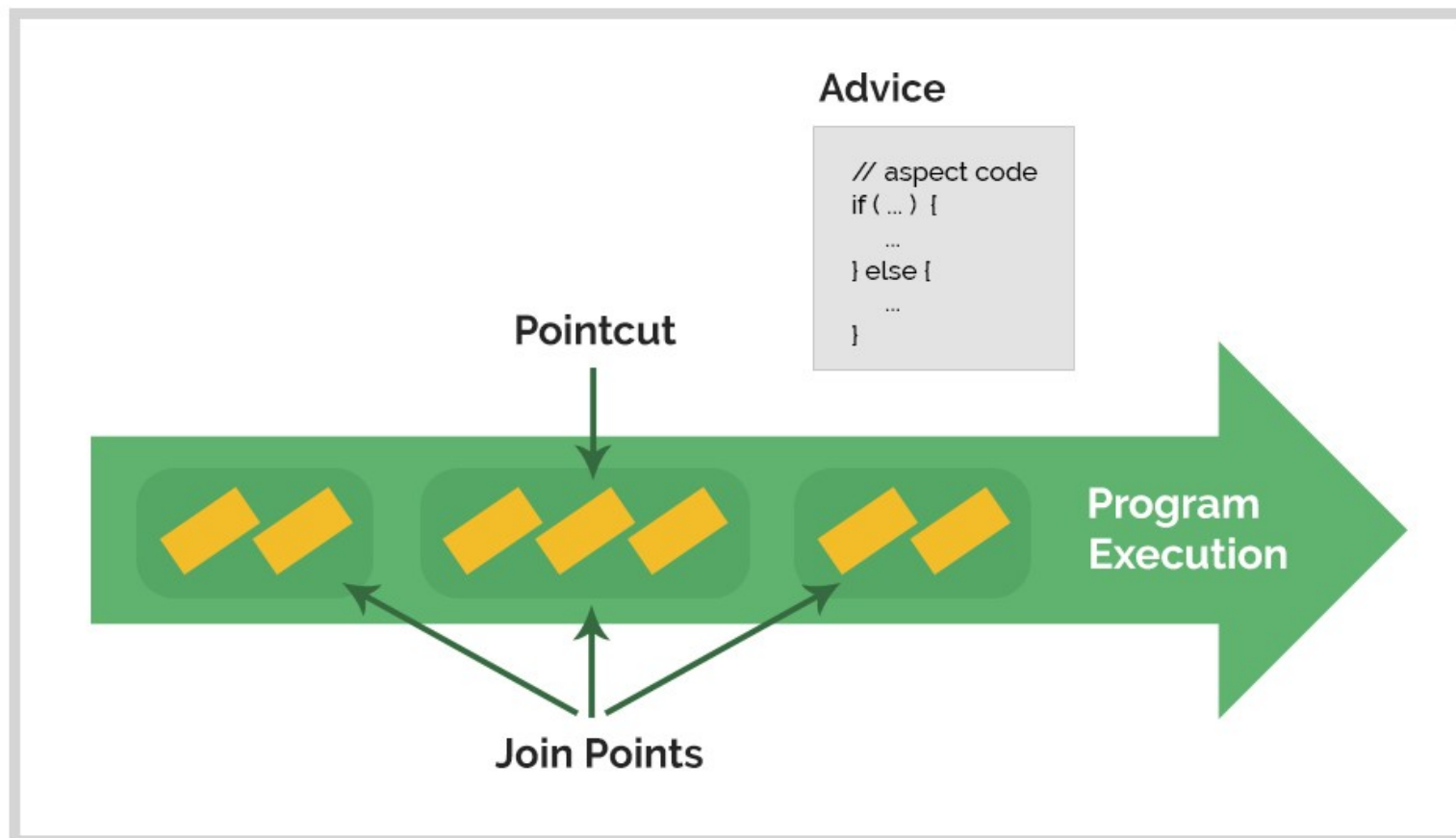
Un **aspect** est la modularisation d'une préoccupation qui concerne plusieurs classes.

Un **point de jonction** est un point lors de l'exécution d'un programme, tel que l'exécution d'une méthode ou le traitement d'une exception.

Un **pointcut** est un prédicat qui aide à faire correspondre un conseil à appliquer par un aspect à un point de jonction particulier.

Un **advice** est une action prise par un aspect à un point de jonction particulier. Les différents types d'advice incluent les advice «autour», «avant» et «après».

# Terminologie





# Exemple Intercepteur transactionnel

---

```
public Object invoke(Invocation invocation)
    throws Throwable
{
    if (tm != null)
    {
        Transaction tx = tm.getTransaction();
        if (tx != null) invocation.setTransaction(tx);
    }
    return getNext().invoke(invocation);
}
```



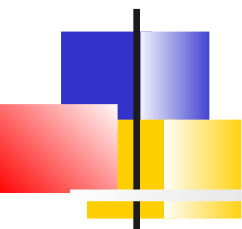
# Services techniques

---

Spring fournit de nombreux aspects qui peuvent être appliqués aux beans applicatifs via la configuration XML ou des annotations.

Citons en particulier :

- Sécurité :  
Protéger l'accès d'une méthode
- Transaction.  
Ouverture Commit/Rollback



# Introduction

---

Le framework Spring

Le pattern IoC et l'injection de  
dépendance

Les autres design patterns de Spring

**Le conteneur Spring**





# BeanFactory

---

- ❖ Le concept central de Spring est l' « **usine à bean** » : **bean factory**
- ❖ Les responsabilités de la bean factory :
  - Instanciation des objets
  - Permet de retrouver un objet créé par son nom
  - Gestion des relations entre les objets => Injection de dépendance
- ❖ *Le terme bean provient des objectifs initiaux de Spring. En fait, il n'est pas nécessaire que ces objets gérés soit des JavaBeans*



# Types de factory

---

- ❖ Un *BeanFactory* contient un certain nombre de définitions de beans, chacune identifiée de manière unique par un nom.
- ❖ Typiquement, un *BeanFactory* charge les définitions de bean à partir d'une source de configuration (comme un document XML)
- ❖ En fonction de la définition du bean, la fabrique retournera soit une nouvelle instance, soit une instance précédemment créée
- ❖ Les *BeanFactories* peuvent être reliées entre elles par des relations d'héritage permettant la mutualisation (et la surcharge) de configuration.



# *BeanDefinition*

---

A l'intérieur du conteneur, les beans sont représentés par la classe ***BeanDefinition*** qui encapsule :

- Le nom qualifié de la classe associée
- La configuration comportemental du bean (scope, méthodes de call-back, ...).
- Les références aux autres beans (les collaborateurs ou dépendances)
- D'autres données de configuration (le dimensionnement d'un pool par exemple)



# Nommage des beans

---

Un bean a un ou plusieurs identifiants (unique à l'intérieur du conteneur) : les **noms** ou **alias**

La convention est d'utiliser la convention Java standard pour les attributs d'une instance.

Les autres noms sont appelés des alias.



# Types de configuration

---

- ❖ Différents sources de configuration sont possibles pour une BeanFactory :
  - **Fichier de configuration XML** :  
Changement sans recompilation, Utilisation de *namespace* spécifique
  - **Fichier properties** : Si la configuration est vraiment simpliste
  - **Annotations dans les sources Java** : A privilégier
  - **Classe de configuration Java** : A privilégier



# Cycle de vie des beans

---

Les implémentations de BeanFactory prennent en charge les méthodes de cycle de vie standard des beans.

14 méthodes d'initialisation sont définies par Spring

1. Initialisation du nom du bean : ***setBeanName***
2. Initialisation de son ClassLoader : ***setBeanClassLoader***
3. Initialisation de la BeanFactory ***setBeanFactory***
4. Initialisation de l'objet Environment ***setEnvironment***
5. Initialisation du résolveur de @Value : ***setEmbeddedValueResolver***
6. Initialisation du chargeur de ressource ***setResourceLoader***
7. Initialisation du diffuseur d'évènements ***setApplicationEventPublisher***
8. Initialisation de la source des libellés ***setMessageSource***
9. Initialisation du contexte applicatif ***setApplicationContext***
10. Initialisation du contexte de Servlet ***setServletContext***
11. Appel des méthodes d'initialisation ***postProcessBeforeInitialization*** (BeanPostProcessors)
12. Appel de la méthode ***afterPropertiesSet***
13. Une méthode personnalisée spécifiée par ***init-method***
14. Appel des méthodes ***postProcessAfterInitialization*** de BeanPostProcessors



# Interfaces *Aware*

---

- ❖ Les beans peuvent implémenter des interfaces de type *Aware* permettant de manipuler le conteneur :
  - Chargeur de ressources (***ResourceLoaderAware***)
  - Générateur d'évènement (***ApplicationEventPublisherAware***)
  - Gestionnaire de message (***MessageSourceAware***)
  - L'application context (***ApplicationContextAware***)
  - Le ServletContext dans le cas d'une application web (***ServletContextAware***)



# Rappel séquence d'instanciation d'un bean

---

- Si le bean implémente *BeanNameAware*, appel de la méthode ***setBeanName()***
- Si le bean implémente *BeanClassLoaderAware* appel de ***setBeanClassLoader()***
- Si le bean implémente *BeanFactoryAware*, appel de ***setBeanFactory()***
- Si le bean implémente *ResourceLoaderAware*, appel de ***setResourceLoader()***
- Si le bean implémente *ApplicationEventPublisherAware*, appel de ***setApplicationEventPublisher()***
- Si le bean implémente *MessageSourceAware*, appel de ***setMessageSource()***
- Si le bean implémente *ApplicationContextAware* appel de ***setApplicationContext()***
- Si le bean implémente *ServletContextAware*, appel de ***setServletContext()***
- Appel des méthodes ***postProcessBeforeInitialization()*** des *BeanPostProcessors*
- Si le bean implémente *InitializingBean*, appel de ***afterPropertiesSet()***
- Si *init-method* défini, appel de la ***méthode d'initialisation spécifique***
- Appel des méthodes ***postProcessAfterInitialization()*** des *BeanPostProcessors*





# Cycles de vie des objets gérés

---

❖ Les objets gérés suivent 3 types de cycle de vie :

- **Singleton**: Il existe une seule instance de l'objet (qui est donc partagé). Idéal pour des services « stateless »
- **Prototype** : A chaque fois que l'objet est demandé via son nom, une instance est créée.
- **Custom object “scopes”** : Ce sont des objets qui interagissent avec des éléments ne faisant pas partie du container.
  - Certains sont fournis par Spring en particulier les objets liés à la requête ou à la session HTTP
  - Il est assez aisé de mettre en place un système de scope pour les objets (Implémentation de `org.springframework.beans.factory.config.Scope`).



# *ApplicationContext*

---

- ❖ Dans la pratique les interfaces *BeanFactory* sont à éviter, il est en effet plus facile d'utiliser les interfaces ***ApplicationContext***
- ❖ Ce sont des objets readOnly qui sont rechargeables
- ❖ Un *ApplicationContext* fournit :
  - Les méthodes de *BeanFactory* pour accéder aux beans
  - La possibilité de charger des fichiers ressources
  - La possibilité de publier des événements vers des listeners enregistrés
  - La possibilité de résoudre les messages (internationalisation)
  - La gestion des contextes hiérarchiques



# *ApplicationContext*

---

**String[] getBeanDefinitionNames()** : Récupérer tous les noms des beans présents dans le contexte applicatif

**java.lang.Object getBean(java.lang.String name)** : Récupérer une instance d'un bean via son nom

**<T> T getBean(java.lang.Class<T> requiredType)** : Renvoie l'instance de bean qui correspond de manière unique au type d'objet donné, si possible.

**java.lang.Class<?> getType(java.lang.String name)** : Détermine le type du bean à partir de son nom.

**boolean isSingleton(java.lang.String name)** : Indique si le bean est un singleton

...



# Modèle événementiel

---

- ❖ Les beans implémentant l'interface *ApplicationListener* reçoivent les événement de type *ApplicationEvent* :
  - ***ContextRefreshedEvent*** : L'*ApplicationContext* est initialisé ou rechargé
  - ***ContextClosedEvent*** : L'*ApplicationContext* est fermé et les beans détruits
  - ***RequestHandledEvent*** : Spécifique au *WebApplicationContext*, indique qu'une requête HTTP vient d'être servie.
- ❖ Il est également possible de définir ses propres événements



# Accès aux ressources

---

- ❖ Spring définit l'interface **Resource** qui offre des méthodes utiles pour le chargement de ressource URL (*getURL()*, *exists()*, *isOpen()*, *createRelative()*, ...)
- ❖ Plusieurs implémentations utiles sont également fournies :
  - *ClassPathResource* : Chargée à partir d'un classpath
  - *ServletContextResource* : Chargée à partir d'un chemin relatif à la racine d'une application web




# *ResourceLoader*

---

- ❖ Chaque contexte Spring implémente l'interface *ResourceLoader* qui détermine la stratégie de chargement des ressources.
- ❖ Par exemple, le code

```
Resource template = ctx.getResource("some/resource/path/myTemplate.txt");
```

- retourne un *ClassPathResource* pour une *ClassPathXmlApplicationContext*
- retourne un *ServletContextResource* pour un contexte de type *WebApplicationContext*



# Configuration via les annotations

---

## **Classes de configuration**

*@Component* et autre stéréotypes  
Injection de dépendances et *@Autowired*  
Environnement et *SpEL*



# Comparaison avec XML

---

- ❖ A la place du XML, il est possible d'utiliser des annotations dans la classe du bean.
- ❖ Chaque approche a ses avantages et ses inconvénients
  - Les annotations profitent de leur contexte de placement ce qui rend la configuration plus concise
  - XML permet d'effectuer le câblage sans nécessiter de recompilation du code source
  - Les classes annotées ne sont plus de simple POJOs
  - Avec les annotations, la configuration est décentralisée et devient plus difficile à contrôler
- ❖ Les annotations sont traitées avant la configuration XML ainsi la configuration XML peut surcharger la configuration par annotations





# Concept

---

La configuration s'effectue via des classes Java annotées par **@Configuration**

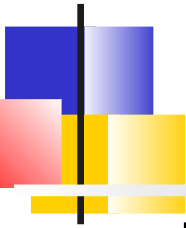
Ces classes sont constituées principalement de méthodes annotées par **@Bean** qui définissent l'instanciation et la configuration des objets gérés par Spring

**@Configuration**

```
public class AppConfig {
```

**@Bean**

```
public MyService myService() {  
    return new MyServiceImpl();  
}  
}
```



# *AnnotationConfigApplicationContext*

---

La classe de Spring

## ***AnnotationConfigApplicationContext***

traite la configuration par annotation

- Les classes annotées via *@Configuration* et les méthodes annotées avec *@Bean* sont enregistrées comme des définitions de bean .
- Les classes annotées *@Component* sont également enregistrées comme définitions de bean
- Les méta-données d'injection de dépendance *@Autowired* ou *@Inject* sont également enregistrées



# Construction

---

Usage de classes *@Configuration*

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

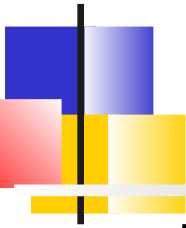


# Méthode *register()*

---

La méthode ***register(Class<?>)*** est pratique lorsque les classes annotées sont nombreuses :

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



# Composition de configuration

---

L'annotation **@Import** permet d'importer une autre classe  
*@Configuration*

```
@Configuration
```

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

```
@Configuration
```

```
@Import(ConfigA.class)
```

```
public class ConfigB {  
    public @Bean B b() { return new B(); }  
}
```

```
-
```

```
ApplicationContext ctx = new  
    AnnotationConfigApplicationContext(ConfigB.class);
```



# Déclaration de bean

---

Il suffit d'annoter une méthode avec **@Bean** pour définir un bean du nom de la méthode.

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



# Injection de dépendances

---

L'expression de dépendances à l'intérieur d'une classe de configuration s'effectue tout simplement par des appels de méthodes

```
@Configuration
public class AppConfig {
    @Bean
    public Foo foo() {
        return new Foo(bar());
    }
    @Bean
    public Bar bar() {
        return new Bar();
    }
}
```



# Attributs de *@Bean*

---

*@Bean* définit 3 attributs :

***name, value()*** : les alias du bean

***init-method*** : méthodes de call-back d'initialisation

***destroy-method*** : Call-back de destruction

@Configuration

```
public class AppConfig {  
  
    @Bean(name={"foo","super-foo"}, initMethod = "init")  
    public Foo foo() {  
        return new Foo();  
    }  
    @Bean(destroyMethod = "cleanup")  
    public Bar bar() {  
        return new Bar();  
    }  
}
```





# Méthode *scan()*

---

La méthode ***scan()*** permet de parcourir un package particulier :

```
public static void main(String[] args) {  
    AnnotationConfigApplicationContext ctx =  
        new AnnotationConfigApplicationContext();  
    ctx.scan("com.acme");  
    ctx.refresh();  
    MyService myService =  
        ctx.getBean(MyService.class);  
}
```



# Annotations *@Enable*

---

Les classes *@Configuration* sont généralement utilisées pour configurer des ressources externes à l'applcatif (une base de données par exemple, des composants d'un module Spring)

Pour faciliter la configuration de ces ressources, Spring fournit des annotations ***@Enable*** qui configurent les valeurs par défaut de la ressource

Les classes configuration n'ont plus alors qu'à personnaliser la configuration par défaut



# Exemples *@Enable*

---

***@EnableWebMvc*** : Configuration par défaut de Spring MVC

***@EnableCaching*** : Permet d'utiliser les annotations *@Cacheable*, ...

***@EnableScheduling*** : Permet d'utiliser les annotations *@Scheduled*

***@EnableJpaRepositories*** : Permet de scanner les classes *Repository* d'accès aux données d'une BD

...



# Exemple


---

```
@Configuration
@EnableWebMvc
public class SpringMvcConfig implements WebMvcConfigurer {

    @Override
    public void configureMessageConverters(
        List<HttpMessageConverter<?>> converters) {

        converters.add(new MyHttpMessageConverter());
    }

    //...
}
```



# Configuration via les annotations

---

Classes de configuration

**@Component et autre stéréotypes**

Injection de dépendances et *@Autowired*

Environnement et *SpEL*



# Introduction

---

Les annotations peuvent également être utilisées pour déclarer des beans applicatifs

Spring est alors capable de détecter automatiquement les définitions en parcourant le classpath et en appliquant des critères de filtre

L'annotation principale pour la définition d'un bean est **@Component**



# Exemple

---

**@Component**

```
public class SimpleMovieLister {  
    private MovieFinder movieFinder;  
    @Autowired  
    public SimpleMovieLister(MovieFinder movieFinder) {  
        this.movieFinder = movieFinder;  
    }  
}
```

**@Component**

```
public class JpaMovieFinder implements MovieFinder {  
    ...  
}
```



# Usage basique de *@Component*

---

## Référence directe des composants

```
public static void main(String[] args) {  
    ApplicationContext ctx = new  
    AnnotationConfigApplicationContext(MyServiceImpl.class,  
    Dependency1.class, Dependency2.class);  
    MyService myService = ctx.getBean(MyService.class);  
    myService.doStuff();  
}
```

## Scan d'un package

```
public static void main(String[] args) {  
    AnnotationConfigApplicationContext ctx = new  
    AnnotationConfigApplicationContext();  
    ctx.scan("com.acme");  
    ctx.refresh();  
    MyService myService = ctx.getBean(MyService.class);  
}
```





# *@ComponentScan*

---

Spring peut détecter automatiquement les classes correspondant à des beans

Il suffit d'ajouter l'annotation **@ComponentScan** en indiquant un package.

Cela s'effectue normalement sur une classe *@Configuration*

```
@Configuration
@ComponentScan(basePackages = "org.example")
public class AppConfig {
    ...
}
```



# Stereotypes

---

Spring propose des annotations alternatives à *@Component* qui donnent une indication sur le rôle du bean (stéréotype) :

**@Repository** : Bean d'accès à des données persistantes

**@Service** : Bean implémentant de la logique métier

**@Controller** : Bean répondant à des requêtes *HTTP*



# @Scope

---

Le **scope** d'un bean définit son cycle de vie et sa visibilité.

L'annotation **@Scope** permet de préciser un des scopes prédéfinis de Spring ou un scope custom

Les scopes prédéfinis de Spring sont :

- **singleton** : Le bean est créé 1 fois au démarrage de Spring. Scope par défaut
- **prototype** : Le bean est créé à chaque utilisation
- **request** : Le bean est associé à une requête HTTP
- **session** : Le bean est associé à une session HTTP
- **application** : Le bean est associé à une application HTTP
- **websocket** : Le bean est associé à une session WebSocket




# Exemple

---

```
@Bean
@Scope(value = WebApplicationContext.SCOPE_REQUEST,
        proxyMode = ScopedProxyMode.TARGET_CLASS)
public HelloMessageGenerator requestScopedBean() {
    return new HelloMessageGenerator();
}
```

Ou en raccourci

```
@Bean
@RequestScope
public HelloMessageGenerator requestScopedBean() {
    return new HelloMessageGenerator();
}
```



# Configuration via les annotations

---

Classes de configuration  
*@Component* et autre stéréotypes  
**Injection de dépendances et**  
***@Autowired***  
Environnement et *SpEL*



# @Autowired

---

L'annotation **@Autowired** se place sur des méthodes *setter*, des méthodes arbitraires, des constructeurs

Il demande à Spring d'injecter un bean du type de l'argument

Généralement un seul bean est candidat à l'injection

@Autowired a un attribut supplémentaire *required*, (*true* par défaut)



# Examples

---

```
public class SimpleMovieLister {
    private MovieFinder movieFinder;
    @Autowired
    public void setMovieFinder(MovieFinder movieFinder) { this.movieFinder = movieFinder;
}
// ...
}
...
public class MovieRecommender {
    private MovieCatalog movieCatalog;
    private CustomerPreferenceDao customerPreferenceDao;
    @Autowired
    public void prepare(MovieCatalog movieCatalog, CustomerPreferenceDao customerPreferenceDao)
    {
        this.movieCatalog = movieCatalog;
        this.customerPreferenceDao = customerPreferenceDao;
    }
    // ...
}
```



# Examples

---

```
public class MovieRecommender {  
    @Autowired  
    private MovieCatalog movieCatalog;  
    private CustomerPreferenceDao customerPreferenceDao;  
    @Autowired  
    public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {  
        this.customerPreferenceDao = customerPreferenceDao;  
    }  
    // ...  
}  
...  
public class MovieRecommender {  
    @Autowired  
    private MovieCatalog[] movieCatalogs;  
    // ...  
}
```





# Injection implicite

---

```
@Controller
public class MovieLister {
    private final MovieFinder finder ;

    // @Autowired n'est pas nécessaire car MovieFinder est final
    public MovieLister(MovieFinder finder) {
        this.finder = finder ;
    }

    public List<Movie> moviesDirectedBy(String arg) {
        List<Movie> allMovies = finder.findAll();
        List<Movie> ret = new ArrayList<>() ;
        for (Movie movie : allMovies ) {
            if (!movie.getDirector().equals(arg))
                ret.add(movie);
        }
        return ret;
    }
}
```



# @Qualifier

---

**@Qualifier** permet de sélectionner un candidat à l'auto-wiring parmi plusieurs possibles

L'annotation prend comme attribut une String dont la valeur doit correspondre à un élément de configuration d'un bean



# Exemple

---

```
public class MovieRecommender {  
    @Autowired  
    @Qualifier("main")  
    private MovieCatalog movieCatalog;  
    // ...  
}  
---  
<beans>  
    <context:annotation-config/>  
    <bean class="example.SimpleMovieCatalog">  
        <qualifier value="main"/>  
        <!-- .... --></bean>  
    <bean class="example.SimpleMovieCatalog">  
        <qualifier value="action"/>  
        <!-- .... --></bean>  
    <bean id="movieRecommender" class="example.MovieRecommender"/>
```



# @Resource

---

**@Resource** permet d'injecter un bean par son nom.

L'annotation prend l'attribut ***name*** qui doit indiquer le nom du bean

Si l'attribut *name* n'est pas précisé, le nom du bean à injecter correspond au nom de la propriété



# Examples

---

```
public class MovieRecommender {  
    @Resource(name="myPreferenceDao")  
    private CustomerPreferenceDao  
        customerPreferenceDao;  
    @Resource  
    private ApplicationContext context;  
    public MovieRecommender() {  
    }  
    // ...  
}
```

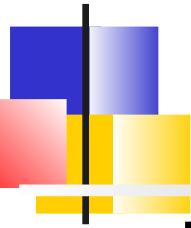


# Méthodes de call-back

---

Spring supporte également les méthodes de call-back **@PostConstruct** et **@PreDestroy**

```
public class CachingMovieLister {  
    @PostConstruct  
    public void populateMovieCache() {  
        // populates the movie cache upon initialization...  
    }  
    @PreDestroy  
    public void clearMovieCache() {  
        // clears the movie cache upon destruction...  
    }  
}
```



# Annotations standard JSR 330

---

Depuis la version 3.0, Spring supporte les annotations de JSR 330.

Pour cela le classpath doit contenir les jars implémentant la JSR

```
<dependency>
```

```
    <groupId>javax.inject</groupId>
```

```
    <artifactId>javax.inject</artifactId>
```

```
    <version>1</version>
```

```
</dependency>
```



# Équivalence


---

**@javax.inject.Inject** correspond à  
*@Autowired*

**@javax.inject.Named** correspond à  
*@Component*

**@javax.inject.Singleton** est équivalent  
à *@Scope("singleton")*





# Configuration via les annotations

---

Classes de configuration  
*@Component* et autre stéréotypes  
Injection de dépendances et *@Autowired*  
**Environnement et *SpEL***



# *Environment*

---

L'interface ***Environment*** est une abstraction modélisant 2 aspects :

- Les **propriétés** : Ce sont des propriétés de configuration des beans. Ils proviennent des fichier .properties, d'argument de commande en ligne ou autre ...
- Les **profils** : Groupe nommé de Beans, les beans sont enregistrés seulement si le profil est activé



# Exemple

---

**@Configuration**

**@Profile("development")**

```
public class StandaloneDataConfig {
```

```
    @Bean
```

```
    public DataSource dataSource() {
```

```
        return new EmbeddedDatabaseBuilder()
```

```
            .setType(EmbeddedDatabaseType.HSQL)
```

```
            .addScript("classpath:com/bank/config/sql/schema.sql")
```

```
            .addScript("classpath:com/bank/config/sql/test-data.sql")
```

```
            .build();
```

```
    }
```

```
}
```

**@Configuration**

**@Profile("production")**

```
public class JndiDataConfig {
```

```
    @Bean(destroyMethod="")
```

```
    public DataSource dataSource() throws Exception {
```

```
        Context ctx = new InitialContext();
```

```
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
```

```
    }
```

```
}
```



# Activation d'un profil

---

## Programmatically :

```
AnnotationConfigApplicationContext ctx = new  
    AnnotationConfigApplicationContext();  
ctx.getEnvironment().setActiveProfiles("development");  
ctx.register(SomeConfig.class,  
    StandaloneDataConfig.class, JndiDataConfig.class);  
ctx.refresh();
```

## Propriété Java

```
-Dspring.profiles.active="profile1,profile2"
```



# Propriétés des beans

---

Les beans possèdent des propriétés de configuration qui fixent les valeurs de certains de leurs attributs

- Les propriétés peuvent provenir de diverses sources: fichiers de propriétés, propriétés système JVM, variables d'environnement système, JNDI, paramètres de contexte de servlet, etc.

Le rôle de l'objet d'environnement est de résoudre les propriétés.

Les beans peuvent implémenter l'interface `EnvironmentAware` ou `@Inject` la classe `Environment` afin d'interroger l'état du profil ou de résoudre directement les propriétés.

Cependant, dans la plupart des cas, les beans ne devraient pas avoir besoin d'interagir directement avec l'environnement, mais peuvent avoir à la place des valeurs de propriété spécifiées par une expression SpEl `$ {...}`



# *Spring El*

---

- ❖ Spring propose un langage similaire à *Unified EL* pouvant être utilisé dans les fichiers de configurations XML ou les annotations
- ❖ Les expressions sont basées sur la notation `# {...}`
- ❖ Elles permettent d'initialiser les propriétés des beans



# Examples

---

```
<bean class="mycompany.RewardsTestDatabase">
  <property name="databaseName"
    value="#{systemProperties.databaseName}"/>
  <property name="keyGenerator"
    value="#{strategyBean.databaseKeyGenerator}"/>
</bean>
```

-----

```
@Repository
public class RewardsTestDatabase {
  @Value("#{systemProperties.databaseName}")
  public void setDatabaseName(String dbName) { ... }
  @Value("#{strategyBean.databaseKeyGenerator}")
  public void setKeyGenerator(KeyGenerator kg) { ... }
}
```



# Fonctionnalités

---

- ❖ Opérateurs logiques, de comparaison, mathématiques, *instanceof*, expressions régulières
- ❖ Opérateur ternaire ( ? : )
- ❖ Accès aux propriétés, aux tableaux, listes et maps, Construction de tableaux et listes
- ❖ Invocation de méthodes, de constructeurs
- ❖ Affectation de valeur
- ❖ Variables, Références aux beans par leurs noms
- ❖ Enregistrement de fonction customisée
- ❖ Sélection ou projection de Collection





# Exemples

---

**// Propriétés**

placeOfBirth.City

**// Listes et tableaux**

"Members[0].Inventions[6]"

**// Maps**

Officers['president']

**//Liste (de liste)**

{{'a','b'},{'x','y'}}

**//Tableaux**

new int[]{1,2,3}

**//Méthodes**

'abc'.substring(2, 3)

**// Collection selection**

Members.?[Nationality == 'Serbian']

**//Collection projection**

Members.![placeOfBirth.city]



# Configuration XML

---

- ❖ Affectation de la propriété d'un bean (via une méthode par exemple)

```
<bean id="numberGuess" class="org.springframework.samples.NumberGuess">  
  <property name="randomNumber" value="${ T(java.lang.Math).random() *  
    100.0 }"/>  
</bean>
```

- ❖ La variable '*systemProperties*' est prédéfinie et peut donc être utilisée

```
<bean id="taxCalculator" class="org.springframework.samples.TaxCalculator">  
  <property name="defaultLocale" value="$  
    { systemProperties['user.region'] }"/>  
</bean>
```

- ❖ Possibilité de faire référence à un autre bean

```
<bean id="shapeGuess" class="org.springframework.samples.ShapeGuess">  
  <property name="initialShapeSeed" value="${ numberGuess.randomNumber }"/>  
</bean>
```



# Annotations

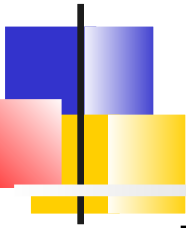
---

- ❖ L'annotation **@Value** peut être placée sur les champs, méthodes ou paramètres des méthodes-constructeurs

```
@Value("${ systemProperties['user.region'] }")
private String defaultLocale ;
--

@Value("${ systemProperties['user.region'] }")
public void setDefaultLocale(String defaultLocale)
--

@Autowired
public void configure(MovieFinder movieFinder,
@Value("${ systemProperties['user.region'] }") String
    defaultLocale)
```



# Externalisation de propriétés

---

Définir le chemin vers le fichier *.properties* :

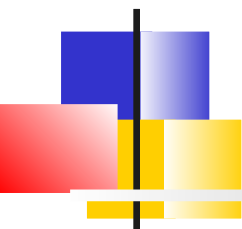
```
<util:properties id="jdbcProperties"  
    location="classpath:org/config/jdbc.properties"/>
```

Ou

```
@PropertySource("classpath:application.properties")
```

L'utiliser via *SpringEl*

```
@Value("${jdbcProperties.url}")  
private String jdbcUrl;
```



# Spring Boot

---

## **Introduction**

Développement avec SpringBoot  
Propriétés de configuration  
Profils



# Introduction

---

Spring Boot a été conçu pour simplifier le démarrage et le développement de nouvelles applications Spring

- Utilisation facile des technologies existantes (Web, DB, Cloud)
- Peut démarrer le projet sans «aucune configuration de beans» préalable



# Essence

---

Spring Boot est un ensemble de bibliothèques exploitées par un système de gestion de build et de dépendance (Maven ou Gradle)



# Auto-configuration

---

Le concept principal de SpringBoot est la configuration automatique

SpringBoot est capable de détecter automatiquement la nature de l'application et de configurer les beans requis

- Cela permet de démarrer rapidement et de remplacer progressivement la configuration par défaut pour les besoins de l'application





# Gestion des dépendances

---

Spring Boot simplifie la gestion des dépendances et de leurs versions:

Il organise les fonctionnalités de Spring en modules.

=> Des groupes de dépendances peuvent être ajoutés à un projet en important des modules "starter".

- Il fournit un POM parent dont les projets héritent et qui gère les versions des dépendances.
- Il offre l'interface Web "Spring Initializr", qui peut être utilisée pour générer des configurations Maven ou Gradle



# Starter Modules

---

## Exemples de starter modules

- ***spring-boot-starter-web***: Les librairies de Spring MVC + configuration automatique d'un serveur embarqué (Tomcat, Undertow, Netty).
- ***spring-boot-starter-data-\**** : Dépendances nécessaires pour accéder aux données d'une certaine technologie(JPA, NoSQL, ...). Il configure automatiquement la source de données pour interagir avec le système de persistance
- ***spring-boot-starter-security*** : SpringSecurity + configuration automatique d'un gestionnaire d'authentification basique
- ***spring-boot-starter-actuator*** : Des dépendances et des beans permettant de surveiller une application (métriques, security audit, HTTP traces) accessible par HTTP ou JMX



# Autoriser l'auto-configuration

---

```
package com.infoq.springboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Application {

    @RequestMapping("/")
    public String home() {
        return "Hello";
    }

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



# *@EnableAutoConfiguration*

---

**@EnableAutoConfiguration** demande à SpringBoot de détecter la configuration appropriée (principalement à partir des dépendances)

La classe *Application* est exécutable, ce qui signifie que l'application et son conteneur intégré peuvent être démarrés en tant qu'application Java

- Les plugins Maven de Boot permettent de produire un exécutable "fat jar" (package mvn)



# Personnalisation de la configuration

---

La configuration par défaut peut être remplacée par différents moyens

- Les fichiers de configuration externes (.properties ou .yaml) sont utilisés pour définir les valeurs des variables de configuration.  
Nous pouvons mettre en place différents fichiers selon les profils (correspondant aux environnements)
- Utilisation de classes spécifiques au bean que vous souhaitez personnaliser (par exemple, *AuthenticationManagerBuilder*)
- Beans en Java qui remplacent les beans par défaut
- La configuration automatique peut également être désactivée pour une partie de l'application



# Spring Boot

---

Introduction

**Développement avec SpringBoot**

Propriétés de configuration

Profils



# Structure Projet

---

## Recommandations :

- Placer la classe Main dans le package racine du projet
- L'annoter avec ***@SpringBootApplication***
  - L'annotation englobe :
    - ***@EnableAutoConfiguration***
    - ***@ComponentScan***
    - ***@Configuration***



# Sous-packages typiques

---

com

+ - example

+ - myproject

+ - Application.java

|

+ - domain

| + - Customer.java

| + - CustomerRepository.java

|

+ - service

| + - CustomerService.java

|

+ - web

+ - CustomerController.java





# Execution

---

Le projet est généralement packagé en un *jar*. Il peut être démarré par:

- `java -jar target/myproject-0.0.1-SNAPSHOT.jar`
- Ou pour le debug :  
`java -Xdebug -Xrunjdwp:server=y,transport=dt_socket,address=8000, suspend=n -jar target/myproject-0.0.1-SNAPSHOT.jar`

Les plugins Maven ou Gradle fournissent également un moyen pour démarrer l'application

```
mvn spring-boot:run
gradle bootRun
```



# Rechargement de code

---

Les applications Spring Boot étant une simple application Java, le rechargement de code à chaud doit être pris en charge.

=> Cela élimine le besoin de redémarrer l'application à chaque changement de code



# Dev Tools

---

Le module ***spring-boot-devtools*** peut être ajouté via une dépendance

Il fournit :

- Ajout de propriétés de configuration utile pour le développement  
Ex :  
*spring.thymeleaf.cache=false*
- Redémarrage automatique lorsqu'une classe ou un fichier de configuration change
- *LiveReload Server* : Permet de recharger automatiquement le navigateur



# Traces

---

Spring utilise ***Common Logging*** en interne mais permet de choisir son implémentation

Des configurations sont fournies pour :

- Java Util Logging
- Log4j2
- Logback (default)



# Format des traces

---

Une ligne contient :

- Timestamp à la ms
- Severity level : ERROR, WARN, INFO, DEBUG ou TRACE.
- Process ID
- Un séparateur --- .
- Le nom de la thread entouré par [].
- Le nom du Logger <=> Nom de la classe.
- Un message
- Une note entourée par []



# Configurer les traces avec Spring

---

Par défaut, Spring affiche les messages de type ERROR, WARN, et INFO sur la console. Différentes techniques permettent de modifier la configuration par défaut :

- *java -jar myapp.jar -debug* : Active les messages DEBUG
- Propriétés ***logging.file.name*** et ***logging.file.path*** pour spécifier un fichier de trace
- Les niveaux de sévérité peuvent être configurés pour chaque logger

```
logging.level.root=WARN
logging.level.org.springframework.web=DEBUG
logging.level.org.hibernate=ERROR
```

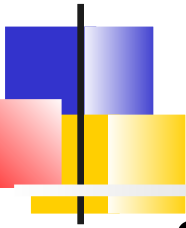
***Lab : Set up IDE***



# Spring Boot

---

Introduction  
Développement avec SpringBoot  
**Propriétés de configuration**  
Profils



# Propriétés de Configuration

---

Spring Boot vous permet d'externaliser la configuration pour s'adapter à différents environnements

Vous pouvez utiliser des fichiers de propriétés ou YAML, des variables d'environnement ou des arguments de commande en ligne.

Les valeurs de propriété peuvent être injectées directement dans les beans

- Avec l'annotation **@Value**
- Ou mappez dans un objet de configuration avec **@ConfigurationProperties**



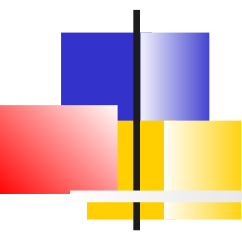


# Précédence

---

Certains façons de configurer ont priorité sur d'autres, voici les principaux niveaux de priorité:

1. *spring-boot-devtools.properties* si *devtools* est activé
2. Les propriétés de Test
3. La ligne de commande Ex : *--server.port=9000*
4. Les variables d'environnement
5. Les propriétés spécifiques à un profil
6. *application.properties* , *yaml*



# *application.properties (.yml)*

---

Spring recherche un fichier `application.properties` ou `application.yml` aux emplacements suivants:

- A la racine du classpath
- Un package *config* dans le classpath
- Le répertoire courant
- Dans un répertoire *config*



# Valeurs filtrées

---

Les valeurs d'une propriété sont filtrées.

Ils peuvent ainsi faire référence à une propriété déjà définie.

```
app.name=MyApp
```

```
app.description=${app.name} is a Boot app.
```

Plus généralement, le fichier de propriétés supporte *SpringEl*



# Format YAML

---

**YAML** (*Yet Another Markup Language*) est une extension de JSON, il est très pratique et très compact de spécifier des données de configuration hiérarchiques.

```
environments:
  dev:
    url: http://dev.bar.com
    name: Developer Setup
  prod:
    url: http://foo.bar.com
    name: My Cool App
```

Est identique à :

```
environments.dev.url=http://dev.bar.com
environments.dev.name=Developer Setup
environments.prod.url=http://foo.bar.com
environments.prod.name=My Cool App
```



# Listes

---

Les listes sont représentées avec le caractère -

my:

servers:

- dev.bar.com
- foo.bar.com

Devient :

```
my.servers[0]=dev.bar.com
```

```
my.servers[1]=foo.bar.com
```



# Injection avec *@Value*

---

La première façon de lire une valeur configurée dans son code d'application consiste à utiliser l'annotation ***@Value***.

```
@Value("${my.property}")  
private String myProperty ;
```

Dans ce cas, aucun contrôle n'est effectué sur la valeur réelle de la propriété



# Validation au démarrage

---

Il est possible de forcer la vérification du type de valeurs de propriété au démarrage:

Utilisez une classe annotée avec **@ConfigurationProperties** et annotez-la avec **@Validated**



# Exemple

---

```
@ConfigurationProperties(prefix="connection")
```

```
@Validated
```

```
public class ConnectionProperties {
```

```
    private String username;
```

```
    private InetAddress remoteAddress;
```

```
    // ... getters and setters
```

```
}
```

```
.....
```

```
@Configuration
```

```
@EnableConfigurationProperties(ConnectionProperties.class)
```

```
public class MyConfiguration {
```

```
}
```





# Contraintes de validation

---

Il est également possible d'ajouter les annotations standard ***javax.validation*** sur les attributs d'une classe de configuration

```
@ConfigurationProperties(prefix="connection")
```

```
@Validated
```

```
public class ConnectionProperties {
```

```
    @NotNull
```

```
    private RemoteAddress remoteAddress;
```

```
    // ... getters and setters
```

```
    public static class RemoteAddress {
```

```
        @NotEmpty
```

```
        public String hostname;
```

```
        // ... getters and setters
```

```
    }
```

```
}
```

***TP: Propriétés de configuration***



# Spring Boot

---

Introduction  
Développement avec SpringBoot  
Propriétés de configuration  
**Profils**



# Introduction

---

Les profils permettent de  
séparer les parties de la  
configuration et de les rendre  
disponibles uniquement dans  
certains environnements:

Production, test, débogage, etc.



# Annotations *@Profile*

---

Tout classe *@Component* ou *@Configuration* peut être marquée avec ***@Profile*** pour limiter son chargement

```
@Configuration
@Profile("production")
public class ProductionConfiguration {

    // ...

}
```



# Activation de profils

---

Les profils sont généralement activés par la propriété ***spring.profiles.active***

Plusieurs profils peuvent être activés simultanément

Ceci est généralement défini avec la ligne de commande:

***--spring.profiles.active=dev,hsqldb***



# Profils dans *.yml*

---

Avec `application.yml`, il est possible de déclarer plusieurs profils en grâce à la notation `---`.

```
server:  
  address: 192.168.1.100
```

```
---
```

```
spring:  
  profiles: development
```

```
server:  
  address: 127.0.0.1
```

```
---
```

```
spring:  
  profiles: production
```

```
server:  
  address: 192.168.1.120
```

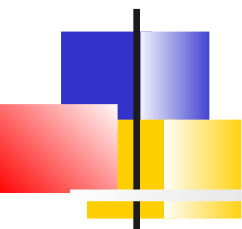


# Propriétés spécifiques à un profil avec fichier *properties*

---

Les propriétés spécifiques au profil peuvent être définies dans des fichiers nommés:

***application-{profile}.properties***



# Persistence

---

Principes de SpringData  
SpringBoot et SpringJPA  
Spring Boot et NoSQL, Exemple  
MongoDB  
Spring Data Rest





# Introduction

---

La mission de ***Spring Data*** est de fournir un modèle de programmation simple et cohérent pour l'accès aux données quelque soit la technologie sous-jacente (Relationnelle, NoSQL, Cloud, Moteur de recherche)

*Spring Data* est donc le projet qui encadre de nombreux sous-projets collaborant avec les sociétés éditrices de la solution de stockage



# Apports de *SpringData*

---

Les apports sont :

- Une abstraction de la notion de **repository** et de **mapping** objet
- La **génération dynamique de requêtes** basée sur des règles de nommage des méthodes
- Des classes **d'implémentations** de bases pouvant être utilisées : *\*Template.java*
- Un support pour **l'audit** (Date de création, dernier changement)
- La possibilité d'intégrer du code **spécifique** au repository
- Configuration **Java ou XML**
- Intégration avec les contrôleurs de **Spring MVC** via **SpringDataRest**



# Interfaces *Repository*

---

L'interface centrale de Spring Data est ***Repository***  
(C'est une classe marqueur)

L'interface prend en arguments de type

- la **classe persistante** du domaine
- son **id**.

La sous-interface ***CrudRepository*** ajoute les méthodes CRUD

Des abstractions spécifiques aux technologies sont également disponibles *JpaRepository*, *MongoRepository*, ...



# Interface *CrudRepository*

---

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {

    <S extends T> S save(S entity);

    T findOne(ID primaryKey);

    Iterable<T> findAll();

    Long count();

    void delete(T entity);

    boolean exists(ID primaryKey);

    // ... more functionality omitted.
}
```



# Déduction de la requête

---

Après avoir étendu l'interface, il est possible de définir des méthodes permettant d'effectuer des requêtes vers le repository

A l'exécution Spring fournit un bean implémentant l'interface et les méthodes fournies.

Spring doit déduire les requêtes à effectuer :

- Soit à partir du **nom** de la méthode
- Soit de l'annotation **@Query**



# Exemple

---

```
public interface MemberRepository
    extends JpaRepository<Member, Long> {

    /**
     * Tous les membres ayant un email particulier.
     * @param email
     * @return
     */
    public List<Member> findByEmail(String email);

    /**
     * Chargement de la jointure one2Many.
     * @param id
     * @return
     */
    @Query("from Member m left join fetch m.documents where m.id =:id")
    public Member fullLoad(Long id);
```



# Méthodes de sélection de données

---

Lors de l'utilisation du nom de la méthode, celles-ci doivent être préfixées comme suit :

- Recherche : *find\*By\**
- Comptage : *count\*By\**
- Suppression : *delete\*By\**
- Récupération : *get\*By\**

La première *\** peut indiquer un flag (comme *Distinct* par exemple)

Le terme **By** marque la fin de l'identification du type de requête

Le reste est parsé et spécifie la clause **where** et éventuellement **orderBy**



# Résultat du parsing

---

Les noms des méthodes consistent généralement de propriétés de l'entité combinées avec *AND* et *OR*

Des opérateurs peuvent également être précisés :  
*Between, LessThan, GreaterThan, Like*

Le flag *IgnoreCase* peut être attribué individuellement aux propriétés ou de façon globale

```
findByLastnameIgnoreCase(...))
```

```
findByLastnameAndFirstnameAllIgnoreCase(...))
```

La clause *order* de la requête peut être précisée en ajoutant *OrderBy(Asc/Desc)* à la fin de la méthode





# Expression des propriétés

---

Les propriétés ne peuvent faire référence qu'aux propriétés directes des entités

Il est cependant possible de référencer des propriétés imbriquées :

```
List<Person> findByAddressZipCode(ZipCode zipCode);
```

Ou si ambiguïté

```
List<Person> findByAddress_ZipCode(ZipCode zipCode);
```



# Gestion des paramètres

---

En plus des paramètres concernant les propriétés, SpringBoot est capable de reconnaître les paramètres de types **Pageable** ou **Sort** pour appliquer la pagination et le tri dynamiquement

Les valeurs de retours peuvent alors être :

- *Page* connaît le nombre total d'éléments en effectuant une requête *count*,
- *Slice* ne sait que si il y a une page suivante

```
Page<User> findByLastname(String lastname, Pageable pageable);  
Slice<User> findByLastname(String lastname, Pageable pageable);  
List<User> findByLastname(String lastname, Sort sort);  
List<User> findByLastname(String lastname, Pageable pageable);
```



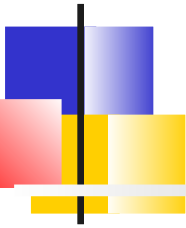
# Limite

---

Les mots clés ***first*** et ***top*** permettent de limiter les entités retournées

Elles peuvent être précisées avec un numérique

```
User findFirstByOrderByLastnameAsc();  
Slice<User> findTop3ByLastname(String lastname,  
                                Pageable pageable);
```



# Mots-clés supportés pour JPA

---

And, Or Is, Equals, Between,  
LessThan, LessThanEqual,  
GreaterThan, GreaterThanEqual,  
After, Before, IsNull,  
IsNotNull, NotNull, Like,  
NotLike, StartingWith,  
EndingWith, Containing, OrderBy,  
Not, In, NotIn, True, False,  
IgnoreCase



# Utilisation des *NamedQuery* JPA

---

Avec JPA le nom de la méthode peut correspondre à une *NamedQuery*.

```
@Entity
@NamedQuery(name = "User.findByEmailAddress",
    query = "select u from User u where u.emailAddress = ?1")
public class User {
}

public interface UserRepository extends JpaRepository<User, Long> {
    User findByEmailAddress(String emailAddress);
}
```



# Utilisation de *@Query*

---

La requête peut également être exprimée dans le langage d'interrogation du repository via l'annotation ***@Query*** :

- Méthode la plus prioritaire
- A l'avantage de se situer sur la classe *Repository*

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    @Query("select u from User u where u.emailAddress = ?1")  
    User findByEmailAddress(String emailAddress);  
  
    @Query("select u from User u where u.firstname like %?1")  
    List<User> findByFirstnameEndsWith(String firstname);  
  
    @Query("select u.id, LENGTH(u.firstname) as fn_len from User u where u.lastname like ?1%")  
    List<Object[]> findByAsArrayAndSort(String lastname, Sort sort);  
  
    @Query("select u from User u where u.firstname = :firstname or u.lastname = :lastname")  
    User findByLastnameOrFirstname(@Param("lastname") String lastname,  
                                   @Param("firstname") String firstname);  
}
```



# SpringBoot et Spring Data-JPA



# Apports Spring Boot

---

*spring-boot-starter-data-jpa* fournit les dépendances suivantes :

- Hibernate
- Spring Data JPA .
- Spring ORMs

Par défaut, toutes les classes annotée par *@Entity*, *@Embeddable* ou *@MappedSuperclass* sont scannées et prises en compte

L'emplacement de départ du scan peut être réduit avec ***@EntityScan***





# Configuration source de données / Rappels

---

Pour accéder à une BD relationnelle, Java utilise la notion de ***DataSource*** (interface représentant un pool de connections BD)

Une data source se configure via :

- Une URL JDBC
- Un compte base de donnée
- Un driver JDBC
- Des paramètres de dimensionnement du pool



# Support pour une base embarquée

---

Spring Boot peut configurer automatiquement les bases de données H2, HSQL et Derby.

Il n'est pas nécessaire de fournir d'URL de connexion, la dépendance Maven suffit :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <scope>runtime</scope>
</dependency>
```



# Base de production

---

Les bases de production peuvent également être auto-configurées.

Les propriétés requises à configurer sont :

`spring.datasource.url=jdbc:mysql://localhost/test`

`spring.datasource.username=dbuser`

`spring.datasource.password=dbpass`

`spring.datasource.driver-class-name=com.mysql.jdbc.Driver`

Voir *DataSourceProperties* pour l'ensemble des propriétés disponibles

L'implémentation du pool sous-jacent privilégie celle de Tomcat dans Spring Boot 1 et Hikari dans Spring Boot 2. Cela peut être surchargée par la propriété *spring.datasource.type*



# Configuration du pool

---

Des propriétés sont également spécifiques à l'implémentation de pool utilisée.

Par exemple pour Hikari :

```
# Timeout en ms si pas de connexions dispo.  
spring.datasource.hikari.connection-timeout=10000
```

```
# Dimensionnement du pool  
spring.datasource.hikari.maximum-pool-size=50  
spring.datasource.hikari.minimum-idle= 10
```



# Propriétés

---

Les bases de données JPA embarquées sont créées automatiquement.

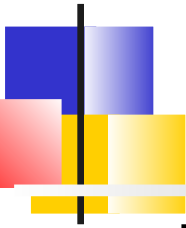
Pour les autres, il faut préciser la propriété  
***spring.jpa.hibernate.ddl-auto***

Ou utiliser les propriétés natives d'Hibernate

- Elles peuvent être spécifiées en utilisant le préfixe *spring.jpa.properties.\**

Ex :

```
spring.jpa.properties.hibernate.globally_quoted_identifiers=true
```



# Configuration des Templates

---

Les beans ***JdbcTemplate*** et ***NamedParameterJdbcTemplate*** sont auto-configurés et peuvent donc être directement injectés

Leur comportement peut être personnalisé par les propriétés *spring.jdbc.template.\**

Ex :

```
spring.jdbc.template.max-rows=500
```



# Example

---

@Repository

```
public class UserDaoImpl implements UserDao {
```

```
    private final String INSERT_SQL = "INSERT INTO USERS(name, address, email) values(:name,:email)";
```

```
    private final String FETCH_SQL_BY_ID = "select * from users where record_id = :id";
```

@Autowired

```
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
```

```
public User create(final User user) {
```

```
    KeyHolder holder = new GeneratedKeyHolder();
```

```
    SqlParameterSource parameters = new MapSqlParameterSource()
```

```
        .addValue("name", user.getName())
```

```
        .addValue("email", user.getEmail());
```

```
    namedParameterJdbcTemplate.update(INSERT_SQL, parameters, holder);
```

```
    user.setId(holder.getKey().intValue());
```

```
    return user;
```

```
}
```

```
public User findUserById(int id) {
```

```
    Map parameters = new HashMap();
```

```
    parameters.put("id", id);
```

```
    return namedParameterJdbcTemplate.queryForObject(FETCH_SQL_BY_ID, parameters, new UserMapper());
```

```
}
```

```
}
```



# Code JDBC ou JPA

---

On peut également se faire injecter les beans permettant de coder à un niveau plus bas :

- Au niveau JDBC, en se faisant injecter la *DataSource*
- Au niveau JPA, en se faisant injecter l'*entityManager* ou l'*entityManagerFactory*





# *OpenInView*

---

Lors d'une application Web, Spring Boot enregistre par défaut l'intercepteur *OpenEntityManagerInViewInterceptor* afin d'appliquer le pattern “**Open EntityManager in View**” permettant d'éviter les *LazyException* dans les vues

Si ce n'est pas le comportement voulu :  
`spring.jpa.open-in-view = false`



# Application Web

---

Spring MVC  
Support pour les APIs Rest  
Spring Boot pour les APIs Rest



# Introduction

---

*SpringBoot* est adapté pour le développement web

Le module starter ***spring-boot-starter-web*** permet de charger le framework Spring MVC

Spring MVC permet de déclarer des beans de type

- ***@Controller*** ou ***@RestController***
- Dont les méthodes peuvent être associées à des requêtes http via ***@RequestMapping***

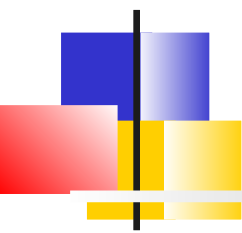


# Spring MVC

---

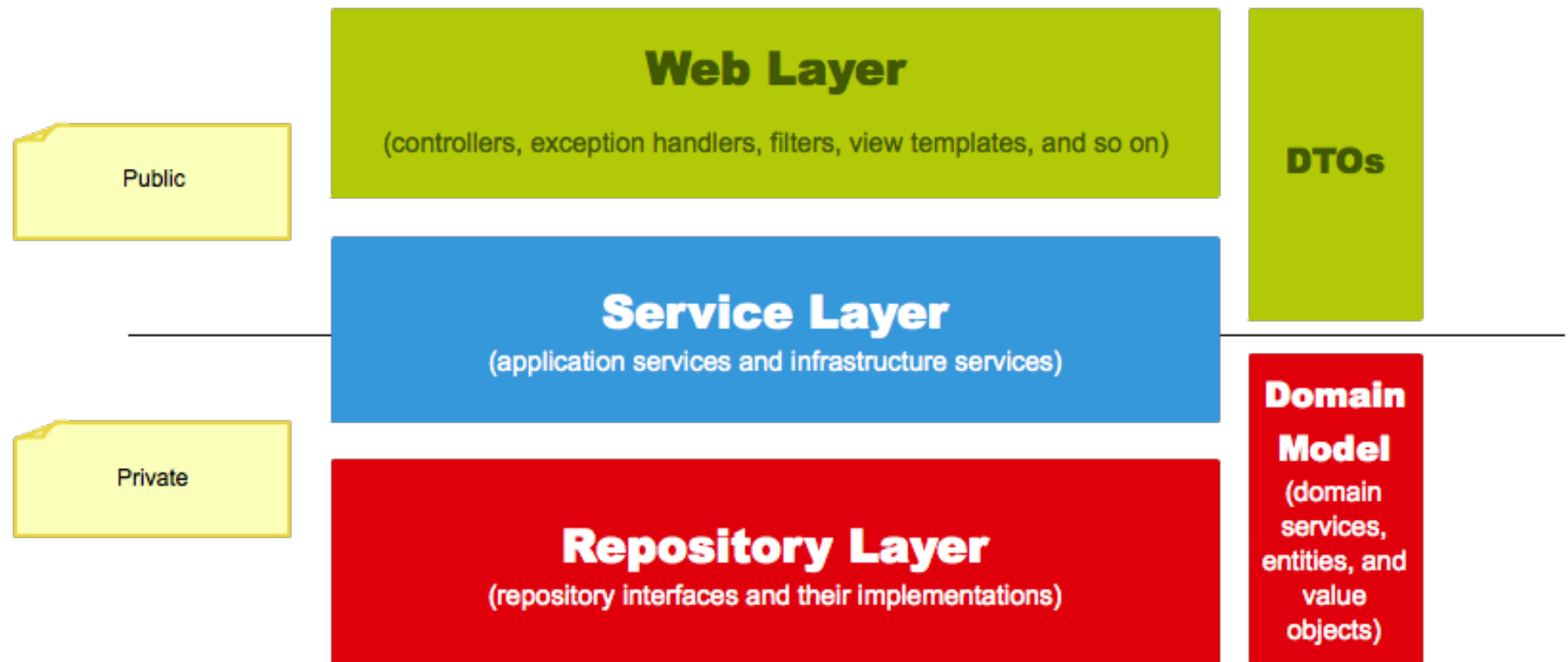
Le framework MVC (Spring-Web Model-View-Controller) est conçu autour du servlet ***DispatcherServlet*** qui distribue des requêtes aux contrôleurs

- L'association contrôleur / requête est effectuée via l'annotation ***@RequestMapping***
  - Les contrôleurs classiques ont la responsabilité de préparer les données du modèle via des interfaces de type ***Map***.  
Le traitement de la requête est ensuite transféré à une technologie de rendu (*JSP, Velocity, Freemarker, Thymeleaf*) qui sélectionne un gabarit de page et génère du HTML
  - Les contrôleurs REST ont la responsabilité de construire une réponse HTTP (code de retour, entêtes, ... ) dont le corps est généralement un document ***json***



# Architecture classique projet

---





# *@Controller, @RestController*

---

Les annotations **@Controller**, **@RestController** se positionnent sur de simples POJOs dont les méthodes publiques sont généralement accessible via HTTP

## **@Controller**

```
public class HelloWorldController {  
  
    @RequestMapping("/helloWorld")  
    public String helloWorld(Model model) {  
        model.addAttribute("message", "Hello World!");  
        return "helloWorld";  
    }  
}
```



# @RequestMapping

---

## @RequestMapping

- Placer au niveau de la classe indique que toutes les méthodes du gestionnaires seront relatives à ce chemin
- Au niveau d'une méthode, l'annotation précise :
  - **path** : Valeur fixe ou gabarit d'URI
  - **method** : Pour limiter la méthode à une action HTTP
  - **produce/consume** : Préciser le format des données d'entrée/sortie



# Compléments *@RequestMapping*

---

Des variantes pour limiter à une méthode :

*@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping*

Limiter à la valeur d'un paramètre ou d'une entête :

*@GetMapping(path = "/pets", headers = "myHeader=myValue", params = "myParam=myValue")*

Utiliser des expressions régulières

*@GetMapping(value = "/ex/bars/{numericId:[\\d]+}")*

Utiliser des propriétés de configuration

*@RequestMapping("\${context}")*





# Types des arguments de méthode

---

Une méthode d'un contrôleur peut prendre des arguments de type :

- La requête ou réponse HTTP (ServletRequest, HttpServletRequest, spring.WebRequest, ...)
- La session HTTP (HttpSession)
- La locale, la time zone
- Les streams d'entrée/sortie
- La méthode HTTP
- L'utilisateur authentifié par HTTP (Principal)
- Une Map, org.springframework.ui.Model ou org.springframework.ui.ModelMap représentant le modèle exposé à la vue
- Errors ou validation.BindingResult : Les erreurs d'une précédente soumission de formulaire

Si l'argument est d'un autre type, il nécessite des **annotations** afin que Spring puisse effectuer les conversions nécessaires à partir de la requête HTTP



# Annotations sur les arguments

---

Les annotations sur les arguments permettent d'associer un argument à une valeur de la requête HTTP :

- **@PathVariable** : Une partie de l'URI
- **@RequestParam** : Un paramètre HTTP
- **@RequestHeader** : Une entête
- **@RequestBody** : Contenu de la requête en utilisant un *HttpMessageConverter*
- **@RequestPart** : Une partie d'une requête multi-part
- **@SessionAttribute** : Un attribut de session
- **@ModelAttribute** : Un attribut de requête
- **@ModelAttribute** : Un attribut du modèle (requête, session, etc.)
- **@Valid** : S'assure que les contraintes sur l'argument sont valides



# Gabarits d'URI

---

Un gabarit d'URI permet de définir des noms de variable :

<http://www.example.com/users/{userId}>

L'annotation **@PathVariable** associe la variable à un argument de méthode

```
@GetMapping("/owners/{ownerId}")
```

```
public String findOwner(@PathVariable String  
    ownerId, Model model) {
```



# Compléments

---

- Un argument **@PathVariable** peut être de type simple, Spring fait la conversion automatiquement
- Si *@PathVariable* est utilisée sur un argument *Map <String, String>*, l'argument est renseigné avec toutes les variables du gabarit
- Un gabarit peut être construit à partir de la combinaison des annotations de type et de méthode



# Paramètres avec *@RequestParam*

---


```
@Controller
@RequestMapping("/pets")
@SessionAttributes("pet")
public class EditPetForm {

    // ...

    @GetMapping
    public String setupForm(@RequestParam("petId") int petId, ModelMap model) {
        Pet pet = this.clinic.loadPet(petId);
        model.addAttribute("pet", pet);
        return "petForm";
    }

    // ...

}
```



# Types des valeurs de retours des méthodes

---

Les types des valeurs de retour possibles sont :

- Pour le modèle MVC :
  - *ModelAndView*, *Model*, *Map*
  - Des Vues : *View*, *String*
- *void* : Si le contrôleur a lui-même généré la réponse
- Pour le modèle REST :
  - Une classe Modèle ou DTO converti via un *HttpConverter* (REST JSON) qui fournit le corps de la réponse HTTP
  - Une *ResponseEntity*<> permettant de positionner les codes retour et les entêtes HTTP



# Formats d'entrée/sorties

---

Il est également de spécifier une liste de type de média permettant de filtrer sur l'entête *Content-type* de la requête HTTP

En entrée, précise le format attendu

```
@PostMapping(path = "/pets", consumes = "application/json")  
public void addPet(@RequestBody Pet pet, Model model) {
```

Ou en sortie, précise le format généré :

```
@GetMapping(path = "/pets/{petId}",  
    produces = MediaType.APPLICATION_JSON_UTF8_VALUE)  
@ResponseBody  
public Pet getPet(@PathVariable String petId, Model model) {
```



# @RequestBody et convertisseur

---

L'annotation **@RequestBody** utilise des *HTTPMessageConverter* qui se basent sur l'entête *content-type* de la requête

- *StringHttpMessageConverter*
- *FormHttpMessageConverter*  
(*MultiValueMap<String, String>*)
- *ByteArrayHttpMessageConverter*
- *MappingJackson2HttpMessageConverter* : JSON
- *MappingJackson2XmlHttpMessageConverter* : XML
- ...





# Spring MVC pour les APIs Rest



# *@RestController*

---

Si les contrôleurs n'implémentent qu'une API Rest, ils peuvent être annotés par ***@RestController***

Ces contrôleurs ne produisent que des réponses au format JSON, XML  
=> Pas nécessaire de préciser *@ResponseBody*



# Exemple pour des données JSON

---

```
@RestController
@RequestMapping(value="/users")
public class MyRestController {

    @RequestMapping(value="/{user}", method=RequestMethod.GET)
    public User getUser(@PathVariable Long user) {
        // ...
    }

    @GetMapping(value="/{user}/customers")
    List<Customer> getUserCustomers(@PathVariable Long user) {
        // ...
    }

    @DeleteMapping(value="/{user}")
    public User deleteUser(@PathVariable Long user) {
        // ...
    }

    @PostMapping
    public ResponseEntity<Member> register(@Valid @RequestBody Member member) {

        member = memberRepository.save(member);

        return new ResponseEntity<>(member,HttpStatus.CREATED);
    }
}
```



# Sérialisation JSON

---

Un des principales problématiques des interfaces REST et la conversion des objets du domaine Java au format JSON.

Des frameworks spécialisés sont utilisés (Jackson, Gson) mais en général le développeur doit régler certaines problématiques :

- Boucle infinie pour les relations bidirectionnelles entre classes du modèle
- Adaptation aux besoins de l'interface de front-end
- Optimisation du volume de données échangées



# Contrôler la sérialisation

---

Une API comme Jackson propose une sérialisation par défaut pour les classes modèles basée sur les getter/setter.

Pour adapter la sérialisation par défaut à ses besoins, 3 alternatives :

- Créer des classes DTO spécifiques
- Utiliser les annotations proposées par Jackson
- Implémenter ses propres *ObjectMapper*



# Annotations Jackson

---

**@JsonProperty, @JsonGetter, @JsonSetter,  
@JsonAnyGetter, @JsonAnySetter, @JsonIgnore,  
@JsonIgnoreProperty, @JsonIgnoreType :**

Permettant de définir les propriétés JSON

**@JsonRootName** : Arbre JSON

**@JsonSerialize, @JsonDeserialize** : Indique des  
dé/sérialiseurs spécialisés

**@JsonManagedReference, @JsonBackReference,  
@JsonIdentityInfo** : Gestion des relations  
bidirectionnelles

....



# Spring Boot et APIs Rest



# Auto-configuration

---

*SpringBoot* effectue des configurations automatiques pour Spring MVC. Les principaux apports sont :

- Démarrage automatique des serveur embarqué
- Configuration par défaut pour servir des ressources statiques (index.html, favicon, Webjars)
- Détection et configuration automatique du langage de templating
- Configuration automatique des *HttpMessageConverters* permettant un comportement par défaut des sérialiseurs
- Fourniture automatique de *RestTemplateBuilder* pour effectuer des appels REST





# Personnalisation de la configuration

---

- Ajouter un bean de type ***WebMvcConfigurer*** et implémenter les méthodes voulues :
  - Configuration MVC (ViewResolver, ViewControllers)
  - Configuration du CORS
  - Configuration d'intercepteurs
  - ...



# Exemple Cross-origin

---

Le *crosss-origin resource sharing*, i.e pouvoir faire des requêtes vers des serveurs différents que son serveur d'origine peut facilement se configurer globalement en surchargeant la méthode *addCorsMapping* :

```
@Configuration
public class MyConfiguration implements WebMvcConfigurer {
    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**");
    }
}
```

A noter qu'il est également possible de configurer le cors individuellement sur les contrôleurs via l'annotation **@CrossOrigin**

Voir : <https://spring.io/guides/gs/rest-service-cors/>



# Exemple *Intercepteurs*

---

```
@SpringBootApplication
public class MyApplication implements WebMvcConfigurer {

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

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        System.out.println("Adding interceptors");
        registry.addInterceptor(new MyInterceptor()).addPathPatterns("/
**");
        super.addInterceptors(registry);
    }
}
```



# Gestion des erreurs

---

*Spring Boot* associe ***/error*** à la page d'erreur globale de l'application

- Un comportement par défaut en REST ou en Web permet de visualiser la cause de l'erreur

Pour remplacer le comportement par défaut :

- Modèle MVC
  - Implémenter ***ErrorController*** et l'enregistrer comme Bean
  - Ajouter un bean de type ***ErrorAttributes*** qui remplace le contenu de la page d'erreur
- Modèle REST
  - L'annotation ***ResponseStatus*** sur une exception métier lancée par un contrôleur
  - Utiliser la classe ***ResponseStatusException*** pour associer un code retour à une Exception
  - Ajouter une classe annotée par ***@ControllerAdvice*** pour centraliser la génération de réponse lors d'exception



# Exemple

---

**@ResponseStatus(value = HttpStatus.NOT\_FOUND)**

```
public class MyResourceNotFoundException extends RuntimeException {  
    public MyResourceNotFoundException() {  
        super();  
    }  
    public MyResourceNotFoundException(String message, Throwable cause) {  
        super(message, cause);  
    }  
    public MyResourceNotFoundException(String message) {  
        super(message);  
    }  
    public MyResourceNotFoundException(Throwable cause) {  
        super(cause);  
    }  
}
```

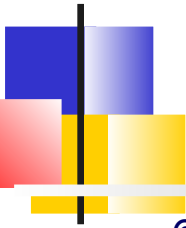


# *ResponseStatusException*

---

```
@GetMapping(value =("/{id}")
public Foo findById(@PathVariable("id") Long id, HttpServletResponse response)
{
    try {
        Foo resourceById = RestPreconditions.checkFound(service.findOne(id));

        eventPublisher.publishEvent(new SingleResourceRetrievedEvent(this,
response));
        return resourceById;
    }
    catch (MyResourceNotFoundException exc) {
        throw new ResponseStatusException(
            HttpStatus.NOT_FOUND, "Foo Not Found", exc);
    }
}
```



# Exemple *@ControllerAdvice*

---

**@ControllerAdvice**

```
public class NotFoundAdvice extends ResponseEntityExceptionHandler {
```

```
    @ExceptionHandler(value = {MemberNotFound.class, DocumentNotFound.class})
```

```
    ResponseEntity<Object> handleNotFoundException(HttpServletRequest request,
        Throwable ex) {
        return new ResponseEntity<Object>(
            "Entity was not found", new HttpHeaders(), HttpStatus.NOT_FOUND);
    }
```

**@Override**

```
    protected ResponseEntity<Object>
        handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
            HttpHeaders headers, HttpStatus status, WebRequest request) {
        return new ResponseEntity<Object>(
            ex.getMessage(), new HttpHeaders(), HttpStatus.BAD_REQUEST);
    }
}
```



# Appels de service REST

---

*Spring* fournit la classe ***RestTemplate*** facilitant les appels aux services REST.

*Spring Boot* ne fournit pas de bean auto-configuré de type *RestTemplate* mais il auto-configuré un ***RestTemplateBuilder*** permettant de les créer







# Mise en place de Swagger

---

La mise en place de Swagger dans un environnement *SpringBoot* est assez direct :

- Ajouter les dépendances dans *pom.xml*
- Se créer une classe configuration définissant un bean de type *Docket* (filtre des annotations) et autorisant les configurations par défaut
- Utiliser les annotations Swagger

=> Accéder à */swagger-ui.html*



# Dépendances

---

```
<dependency>
```

```
  <groupId>io.springfox</groupId>
```

```
  <artifactId>springfox-swagger2</artifactId>
```

```
  <version>2.9.2</version>
```

```
</dependency>
```

```
<dependency>
```

```
  <groupId>io.springfox</groupId>
```

```
  <artifactId>springfox-swagger-ui</artifactId>
```

```
  <version>2.9.2</version>
```

```
</dependency>
```



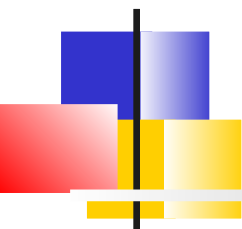
# Configuration

---

```
@Configuration
@Profile("swagger")
@EnableSwagger2
public class SwaggerConfiguration {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.any())
            .paths(PathSelectors.any())

            .build();
    }
}
```



# Annexe JPA



# Classes entités persistantes

---

- ❖ Les entités sont de simple Java Beans (constructeur sans argument et getter/setter) qui en plus contiennent
  - Les méta-données de mapping exprimées via des annotations
    - Type des champs
    - Adaptation au modèle physique
    - Associations entres entités



# Conception

---

- Bien que rigoureusement non obligatoire, il est souhaitable que la classe implémente l'interface ***java.io.Serializable***
  - cela permet aux instances de transiter sur le réseau entre le conteneur et un client distant
- Par défaut, la classe entité est **associée à une table** du nom de sa classe
- Un bean entité est toujours associé à une **clé primaire** (primary key) qui permet d'identifier ce bean de façon unique dans la base de données



# Annotations des entités

---

L'annotation **@javax.persistence.Entity** indique que les instances de cette classe pourront être prises en charge par un service de persistance *EntityManager*

Il doit avoir au moins une annotation **@Id** indiquant la clé primaire

Par défaut, tous les attributs ayant des méthodes get/set sont **persistants**





# Exemple

---

**@Entity**

```
public class Personne implements Serializable{  
  
    // l'attribut id désigne la clé primaire, le mapping est défini sur les attributs du bean  
    // les attributs id, nom, prenom sont mappés respectivement sur les colonnes id, nom, prenom  
  
    @Id private int id;  
  
    private String nom, prenom;  
  
    public Personne(){}  
  
    public int getId(){return id;}  
  
    public void setId(int pk){id=pk;}  
  
    public String getNom(){return nom;}  
  
    public void setNom(String n){nom=n;}  
  
    public String getPrenom(){return prenom;}  
  
    public void setPrenom(String p){prenom=p;}  
  
}
```



# *@Transient*

---

- Par défaut , les attributs d'une classe entité sont persistants

L'annotation **@Transient** permet d'indiquer qu'un attribut n'est pas persistant

Cette annotation se place soit sur l'attribut, soit sur une méthode get



# Associations Entités

---

Les associations entre entités sont indiqués par les annotations :

- @OneToOne
- @OneToMany, @ManyToOne
- @ManyToMany

Elles donnent lieu à des clés étrangères, des tables d'association ou des clés primaires partagés dans la base de données.

# Association OneToMany - uni.

## Table d'association contrôlé par Hibernate

@Entity

```
public class Theme {
```

@Id

```
private Long id;
```

```
private String label;
```

@OneToMany

```
private Set<MotClef> motclefs = new HashSet<MotClef>();
```

@Entity

```
public class MotClef {
```

@Id

```
private Long id;
```

```
private String mot;
```

```
public MotClef(){}
```

# Gestionnaire de persistance



---

- ❖ Le gestionnaire de persistance, i.e. ***EntityManager*** est l'API permettant de faire les opérations de lecture et d'écriture sur la BD

# Fonctions du gestionnaire de persistance



---

- ❖ Les principales fonctions de l'entity manager sont :
  - Chargement d'entités via leur id
  - insertions d'entités
  - suppression d'entités
  - synchronisation entre entités et base de données
  - Requêtes et recherche d'entité



# Chargement d'entité

---

Les méthodes permettant de rechercher un bean entité à partir de sa clé primaire sont :

- ***<T> T find(Class<T> entityClass, Object primaryKey) :***

retourne l'entité trouvé ou *null* si l'entité n'est pas trouvée

Ex : **`em.find(Theme.class, 11) ;`**

- ***<T> T getReference(Class<T> entityClass, Object primaryKey)***

retourne le bean entité trouvé ou lance

*EntityNotFoundException* si l'entité n'est pas trouvée



# Création/Suppression

---

- la méthode qui permet de rendre persistant un bean transient est :  
***void persist(Object entity);***
  - le bean transmis en paramètre est inséré en base de données (INSERT SQL)
- la méthode qui permet de supprimer bean entité est :  
***void remove(Object entity);***
  - le bean transmis en paramètre est supprimé de la base de données (DELETE SQL)