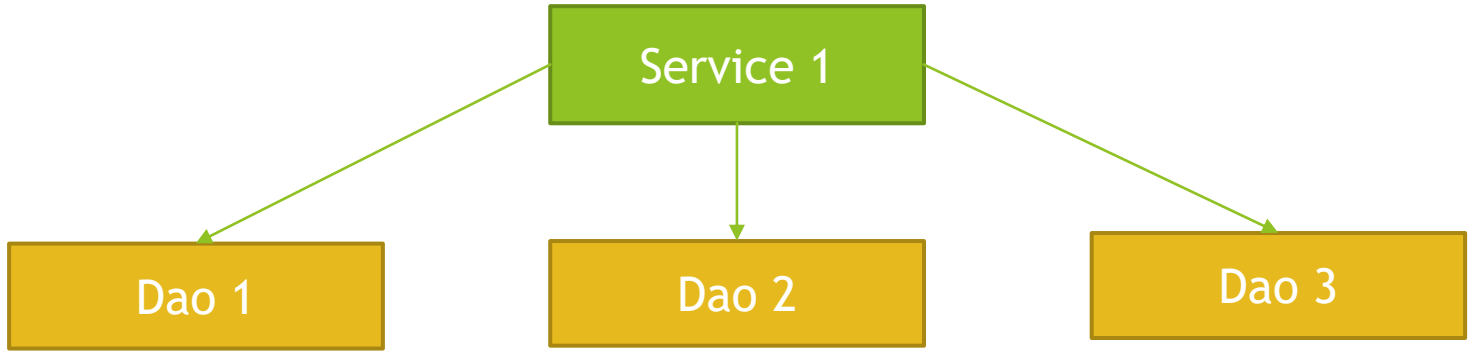
Spring Framework: étape par étape pour devenir professionnel

Présenté par : **ZAROUAL Mohamed**

Consultant Java/JEE

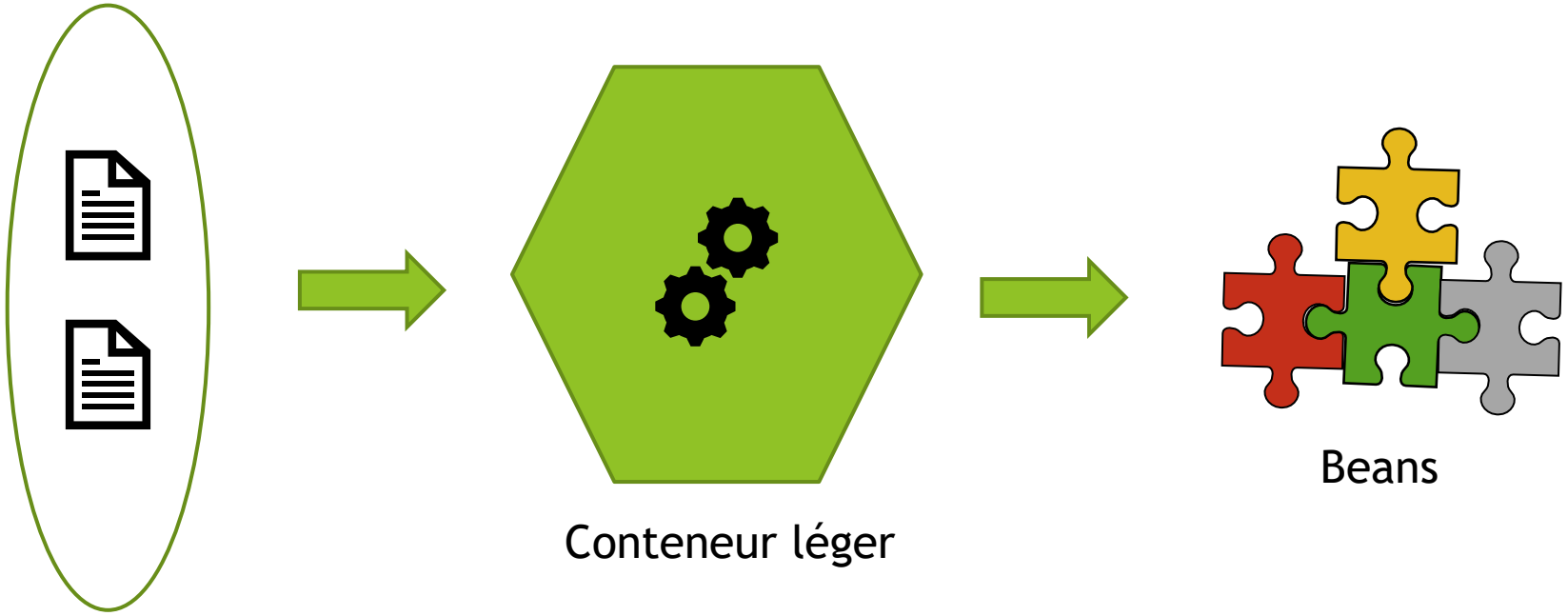
Présentation du cours



Présentation du cours

- Spring : le conteneur léger
  - Design pattern : IoC
  - Gestion des beans : création, initialisation et destruction

Présentation du cours



Fichiers de configuration

Conteneur léger

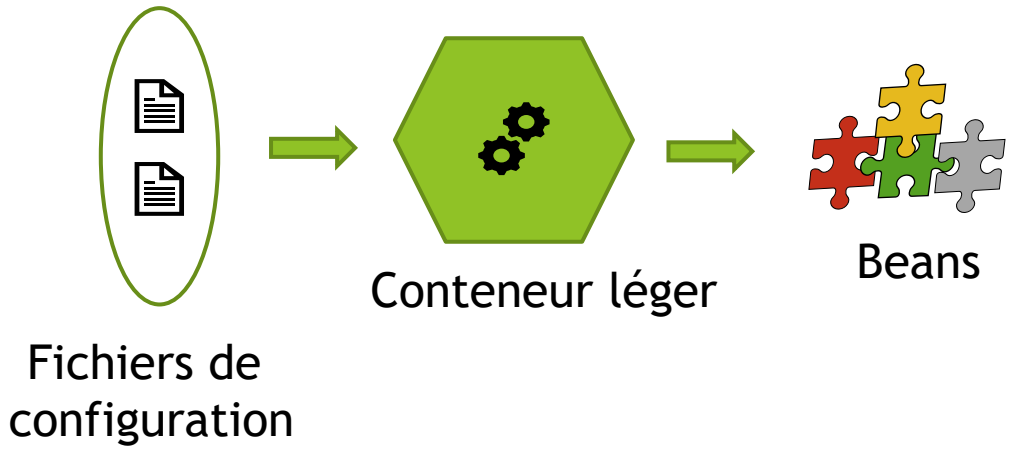
Beans

Présentation du cours

Un Bean
est une instance d'une classe gérée dans le conteneur de Spring

Définition d'un bean :

- Classe
- ID
- des informations relatives à sa configuration
  - Paramètres du constructeur
  - Portée : Singleton/Prototype
  - …

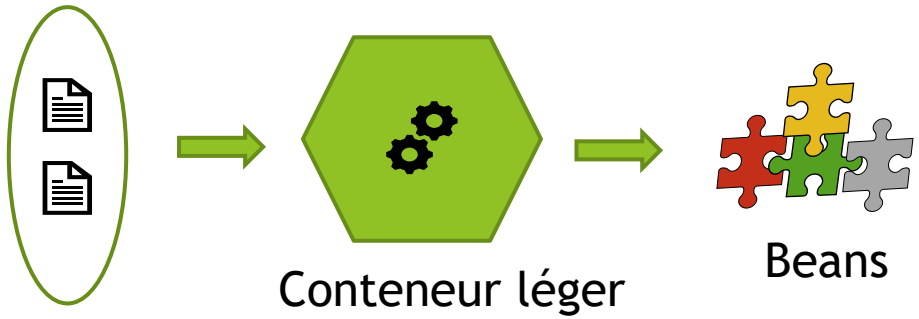


Présentation du cours

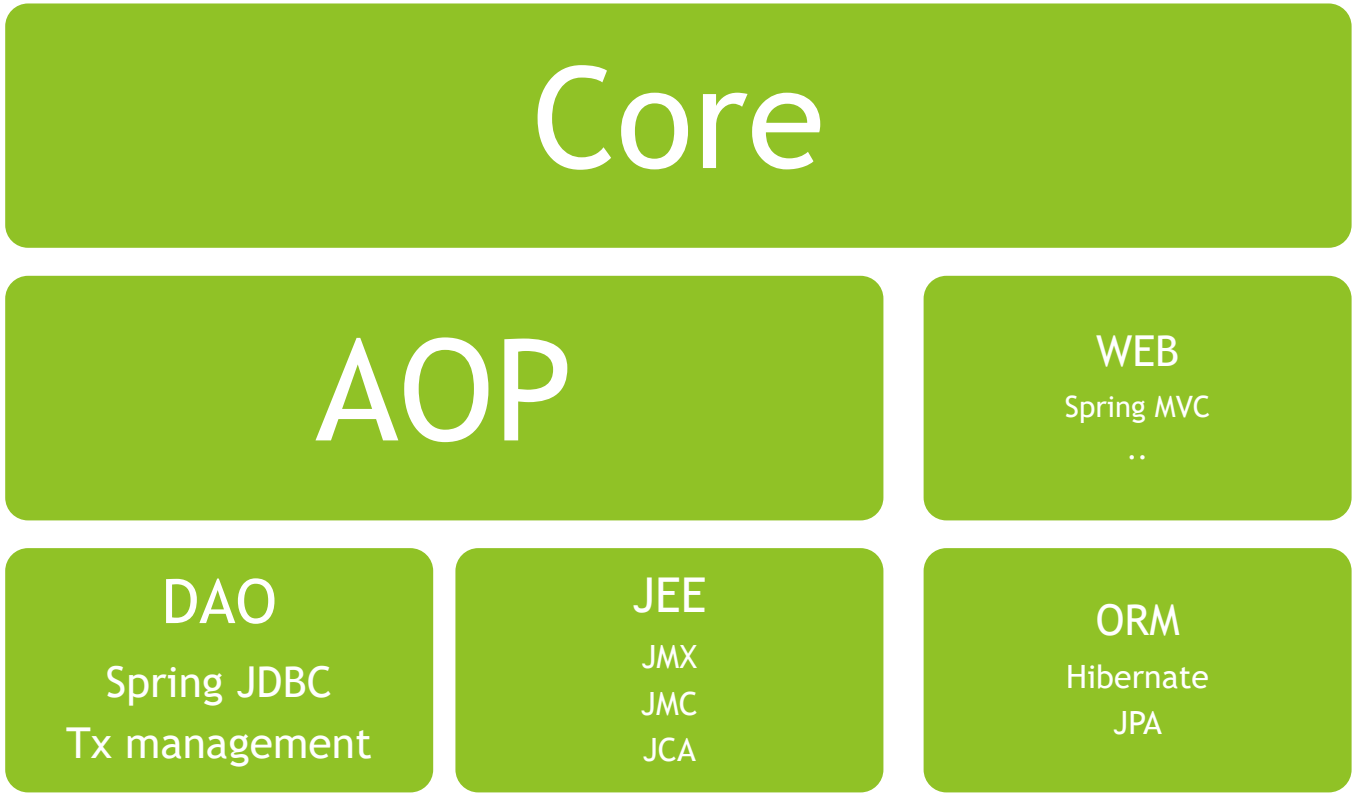
L'**inversion de contrôle**

est un design pattern dont le processus définit les dépendances d'un objet sans avoir à les créer.

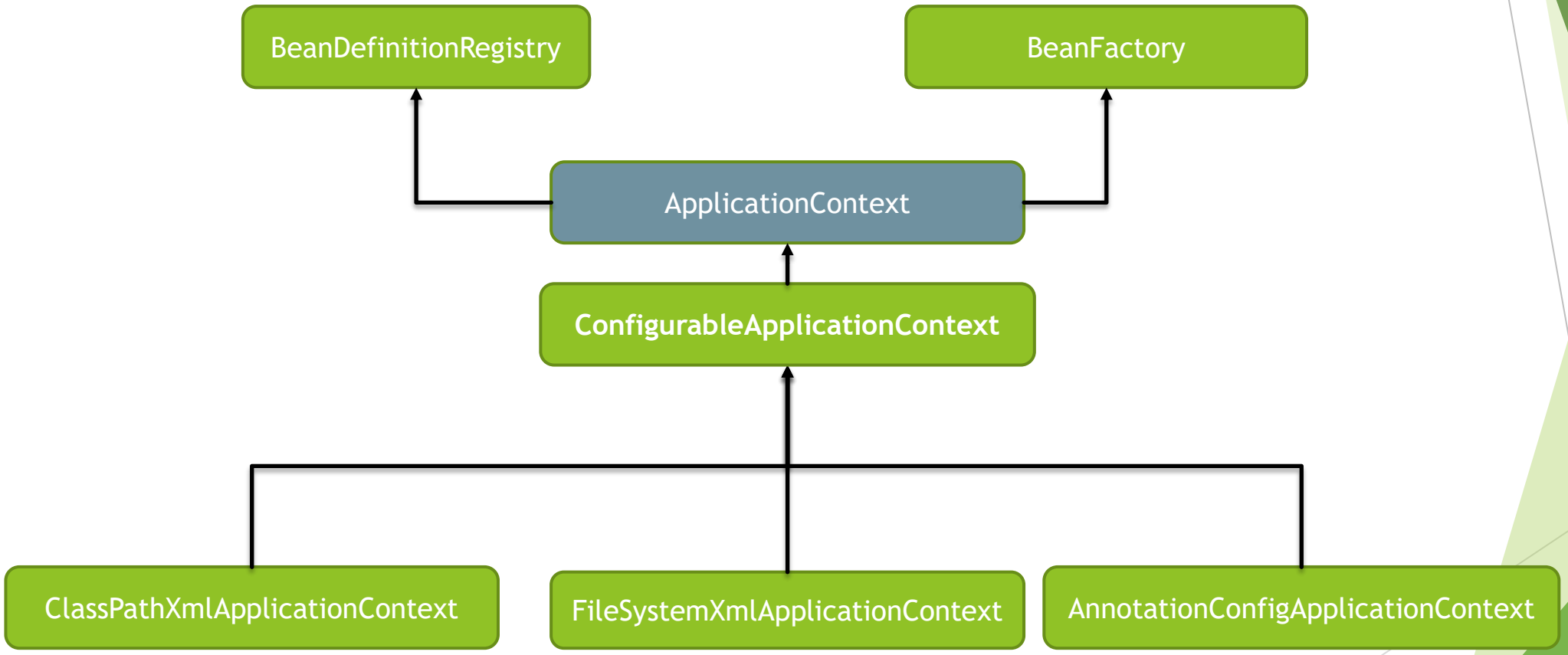
- La recherche de dépendance
- L'injection de dépendances



Spring : Multitude de modules



Spring : ApplicationContext



Injection des dépendances

Configuration XML

injection des dépendances par constructeur

- Par défaut, Spring sélectionne le premier constructeur supportant cette configuration
- Balise : **constructor-arg**
- L'attribut **ref** : informe Spring que la valeur est un bean spring.
- L'attribut **value** : informe Spring que la valeur est une valeur simple

Configuration XML

injection des dépendances par modificateur/setter

- Un setter est obligatoire pour injecter la dépendance
- Balise : **property**
- L'attribut **name** : informe Spring sur la dépendance à injecter
- L'attribut **ref** : informe Spring que la valeur est un bean spring.
- L'attribut **value** : informe Spring que la valeur est une valeur simple

Configuration XML

injection des dépendances qui ne sont pas des beans

- Un setter est obligatoire pour injecter la dépendance
- Balise : **property**
- L'attribut **name** : informe Spring sur la dépendance à inejcter
- L'attribut **value** : informe Spring que la valeur est une valeur simple
  - int
  - double
  - boolean
  - Char
  - Properties
  - Local
  - url
  - File
  - Class<T>
  - Array
  - Map, Set, List

Configuration XML

Injection par constructeur Vs Injection par setter

- **Par constructeur**
  - définir un contrat fort
- **Par setter**
  - est la plus utilisée avec Spring

Configuration XML

Injection par constructeur Vs Injection par setter

Il n’y a donc pas de réponse unique à la question

Configuration XML

Autowiring : *Injection automatique*

- l'injection explicite des dépendances implique plusieurs lignes dans le fichier de configuration
- l'autowiring permet de simplifier grandement le fichier de configuration
- L'autowiring est pour les objets
- Stratégies de l'autowiring
  - No
  - byName
  - byType
  - Constructor
  - autodetect

Configuration XML

Autowiring : *Injection automatique*

- Autodetect : Déprécié depuis la v3

Configuration XML
Bean naming

- ID
- NAME
- Alias

Configuration XML
Bean Alias

- Alias

Configuration XML
création d'un bean d'une classe singleton

- Une classe singleton ?
- **factory-method** : le nom de la méthode **static** à invoquer

Configuration XML
création d'un bean moyennant une factory

- Utile quand le bean à instancier est géré dans une API qui expose la factory
- **factory-bean** : informe Spring sur le bean géré dans le conteneur et qui encapsule la factory
- **factory-method** : le nom de la méthode **non static** à invoquer

Configuration XML

Séparer la configuration dans plusieurs fichier de configuration

- Si votre application est constituée de plusieurs modules
  - Un fichier de configuration XML par module
    - Authenitifaction-config.xml
    - Administration-config.xml
    - Produits-config.xml
- Si votre application est constituée de plusieurs couches
  - Un fichier de configuration XML par couche
    - dao-config.xml
    - Service-config.xml
    - Security-config.xml

Configuration XML

Simplifier la configuration en utilisant les namespace

- Spring propose plusieurs espaces de nommage (namespaces):
  - aop,
  - jee,
  - lang,
  - tx
  - util.
- Le but est de réduire la quantité de code à produire dans le fichier de configuration XML

Configuration XML
Définitions abstraites de Beans

- L'héritage est une notion très importante dans la POO
- La même chose est possible dans la configuration des beans
  - permet de créer de nouveaux beans à partir d'un bean template
  - Permet de réduire le quantité du code à écrire lors de la configuration

Configuration XML
Création d'un prototype

- Spring crée une seule instance de chaque bean : Singleton
- Prototype : à chaque fois qu'une instance du bean sera demandée, le conteneur va créer une nouvelle instance
  - L'attribut **scope** de la balise **bean**
    - Valeur possibles
      - Singleton (par défaut)
      - prototype

Configuration XML
Création d'un prototype

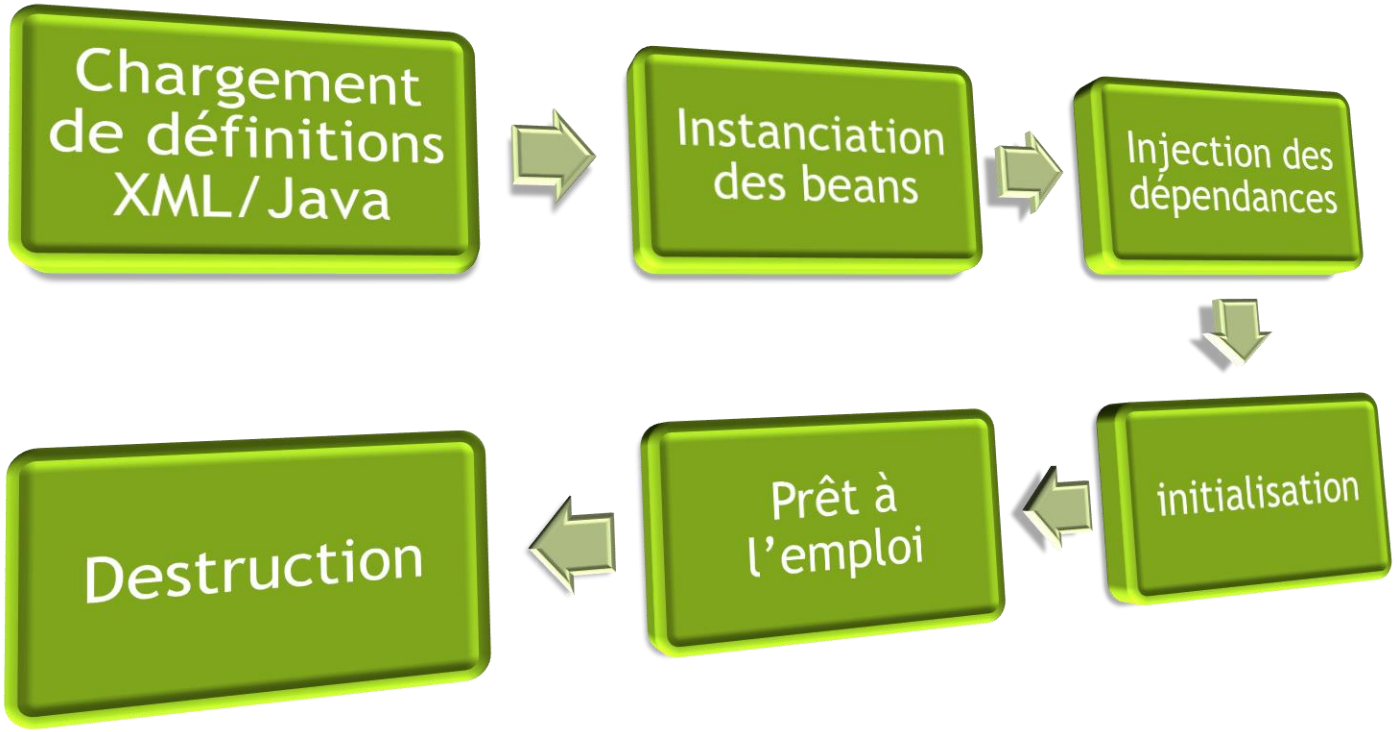
- Spring crée une seule instance de chaque bean : Singleton
- Prototype : à chaque fois qu'une instance du bean sera demandée, le conteneur va créer une nouvelle instance
  - L'attribut **scope** de la balise **bean**
    - Valeur possibles
      - Singleton (par défaut)
      - prototype

Configuration XML
Inner bean

- Inner bean : est bean déclaré dans la définition d'un autre bean
- Inner bean n'est pas visible de l'extérieur
  - Inutile de renseigner les attributs id et name

Configuration XML

Le cycle de vie d'un bean



Configuration XML
Le cycle de vie d'un bean

- Pour l'initialisation, il y a plusieurs façons de faire :
  - **Utilisation de l'attribut init-method de la balise <bean>**
    - La méthode ne prend pas d'argument
    - La méthode peut avoir n'importe quel modificateur de porté. de préférence private
  - **Implémentation l'interface InitializingBean**
    - définir le comportement dans la méthode afterPropertiesSet.
    - Cette méthode n'est pas recommandée puisqu'elle introduit un couplage fort avec Spring

Configuration XML
Le cycle de vie d'un bean

- Quant à la destruction :
  - **Utilisation de l'attribut destroy-method de la balise <bean>**
    - La méthode ne prend pas d'argument
    - La méthode peut avoir n'importe quel modificateur de porté. de préférence private
  - **Implémentation de l'interface DisposableBean**
    - Définir le comportement dans la méthode destroy().
    - Cette méthode n'est pas recommandée puisqu'elle introduit un couplage fort avec Spring

Configuration XML
Définition des profiles

- **Si la configuration de votre application est différente d'un environnement à l'autre**
  - **Par exemple, la définition de la datasource :**
    - Configuration pour l'environnement de production (base de données Oracle)
    - Configuration pour l'environnement de développement (base de donnée H2)

Configuration XML
Définition des profiles

- **Si certains beans de votre application ont un comportement différent en fonction de l'environnement d'exécution**
  - Par exemple, appel à une ressource externe :
    - Un bean pour l'environnement de Production : consommer un web service
    - Un autre bean(mocks) pour l'environnement de Développement : récupérer un JSON dans le file système

Configuration par annotation(java)

Introduction

- Première version de Spring sortie en 2004 ne proposait que la configuration en XML.
- Avec la version java 1.5 et l'introduction des annotations, Spring avait adopté la configuration en Java et ceci dans la version 3.0

Configuration par annotation(java)
Introduction

- Pour la déclaration des beans, spring propose plusieurs annotations :
  - **@Component** : permet de préciser que le bean est un composant
  - **@Repository** : permet de préciser que le bean est un repository (dao)
  - **@Service** : permet de préciser que le bean est un service
  - **@Controller** : permet de préciser que le bean est un contrôleur Spring MVC
  - **@Autowired** permet de demander une injection automatique par type.
  - **@configuration** permet de demander au conteneur d'utiliser cette classe pour instancier des beans.
  - **@Bean** s'utilise sur une méthode d'une classe annotée avec @Configuration qui crée une instance d'un bean.

Configuration par annotation(java)

Injection des dépendances avec @Autowired et @Qualifier

- **@Autowired** permet de demander une injection automatique par type.
  - Parce que c'est rare de trouver deux instances du même type.
  - Spring va chercher l'instance du type demandé et l'injecte automatiquement.
    - Et si jamais Spring trouve qu'il y a plusieurs beans de même type, il va prendre celui dont l'id correspond au nom de la dépendance.
      - Si aucun des beans trouvés ne correspond au nom de la dépendance : une exception UnsatisfiedDependencyException est levée
        - **@Qualifier** entre en jeux !

Configuration par annotation(java)

Injection d'un dépendance par constructeur

- Spring va utiliser la même logique pour injecter les dépendances par constructeur
- Il va chercher d'abord un bean du même type A
  - S'il en trouve plusieurs, il va chercher un bean dont l'id correspond au nom du paramètre
    - S'il ne trouve pas de bean avec le nom du paramètre, une exception **UnsatisfiedDependencyException** est levée
      - Utiliser le @**Qualifier** pour résoudre ce problème

Configuration par annotation(java)

Injection d'un dépendance par constructeur

- Depuis Spring 4.3.x, Si une classe ne contient qu'une seul constructeur, il n y a plus besoin d'utiliser @Autwired.
  - @component (ou une des autres stéreotypes) suffit

Configuration par annotation(java)

injection des dépendances qui ne sont pas des beans

- @Autowired ne permet d'injecter les String et les types primitifs.
- Pour ce cas, Spring propose une annotation spécialement pour ce type de dépendances : @Value

Configuration par annotation(java)
Création d'un prototype

- il est possible de préciser la porté d'un bean spring via l'annotation **@Scope**
  - **@Scope**(value = ConfigurableBeanFactory.**SCOPE_PROTOTYPE**)

Configuration par annotation(java)

Implémentation d'un convertisseur

- Converter<S,T>
  - S : source
  - T : Target (cible)

Configuration par annotation(java)
@Primary

- Interface (FormationDao)
  - FormationDaoAImpl : injectée dans 200 beans
  - 6 mois après, une nouvelle implémentation : FormationDaoBImpl

- Le contexte d'application n'arrive pas à injecter les beans parce qu'il en trouve deux !
  - Solution
    - 1- Aller dans les 200 beans et ajouter @Qualifier en indiquant le nom du bean associer à la première implémentation
    - 2- utiliser @Primary

Configuration par annotation(java)

Le cycle de vie d'un bean

- **@Bean**
  - @Bean (**initMethod** = "init", **destroyMethod** = "destroy")
- **@PostConstrut** et **@PreDestroy**

Configuration par annotation(java)

Séparer la configuration en plusieurs classes de configuration

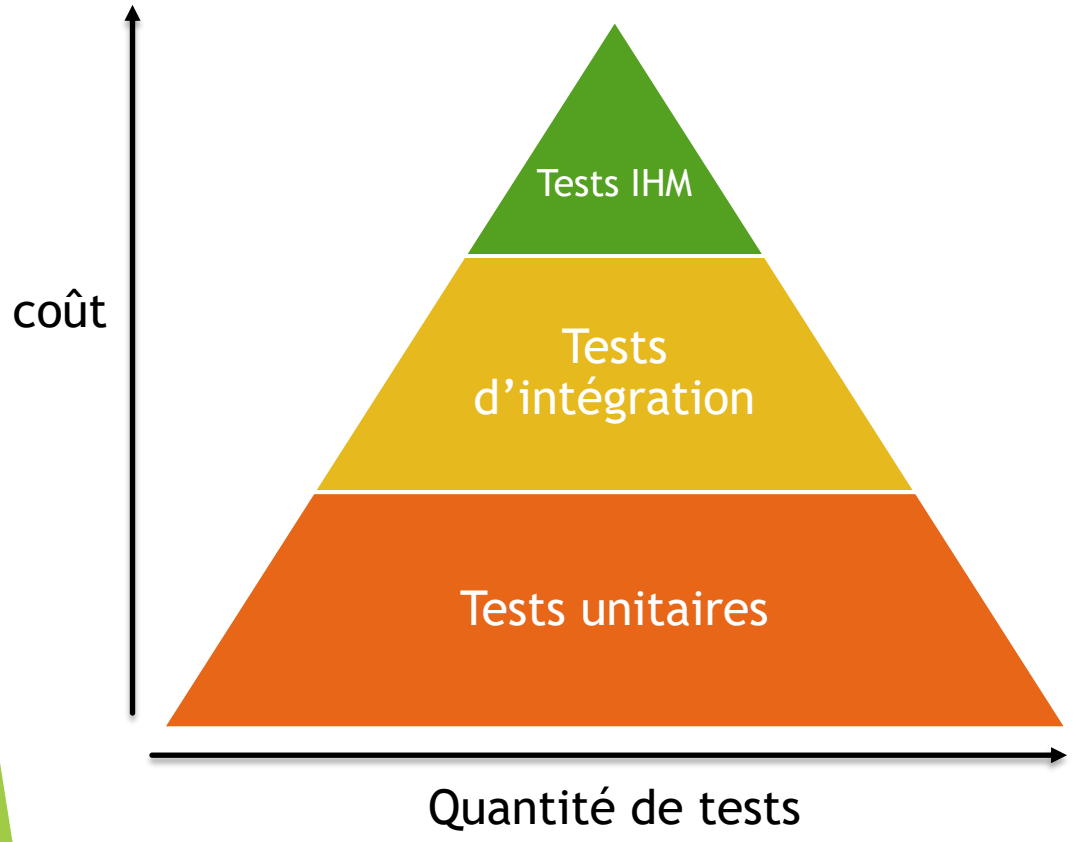
- Dans une application, le plus souvent on a un module par couche applicative.
  - Un module pour la couche DAO,
  - une autre pour la couche service, etc.
- *Séparer la configuration en plusieurs classes de configuration* revient à dire , une classe de configuration par couche.

Configuration par annotation(java)
Définition des profiles

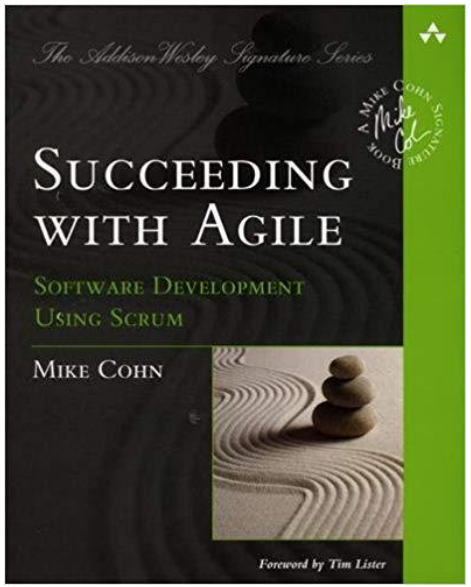
- Pour configurer un profil on utilise l'annotation @**Profile**
- l'annotation @**Profile** peut être utilisée sur différentes annotations
  - @Service
  - @Repository
  - @Component
  - @Configuration
  - @Bean
  - ….

Tester une application Spring

Pyramide de tests



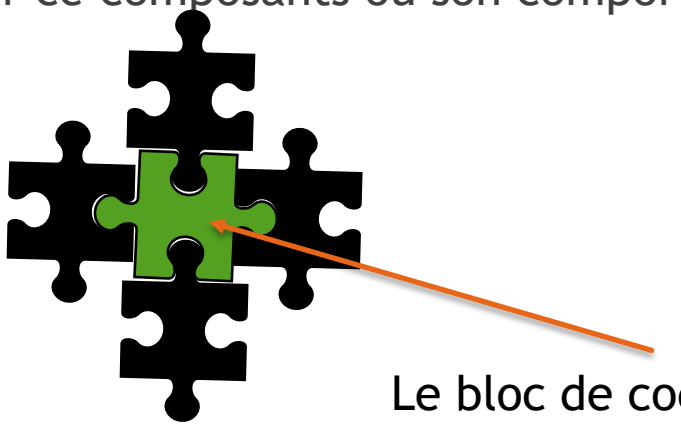
- décrite par Mike Cohn dans le livre Succeeding with Agile



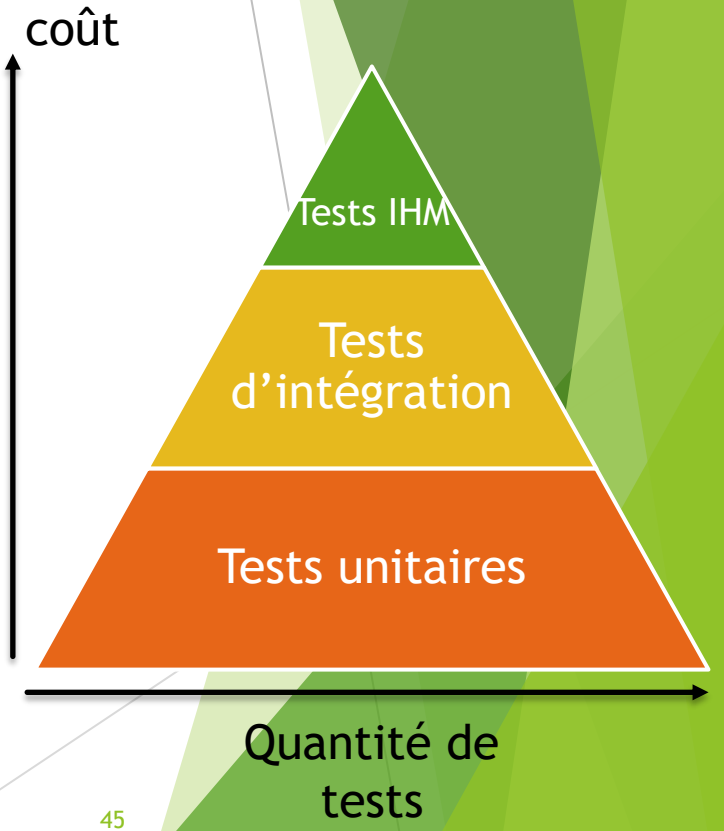
Tester une application Spring

Qu'est ce que c'est un test unitaire

- Concrètement, un test unitaires est l'action de tester une unité de code (méthode, classe, module) pour s'assurer de son bon fonctionnement.
- tester une unité de code consiste à vérifier, en fonction de certaines données fournies en entrée que les résultats sont conformes aux spécifications du module.
- Si l'unité de code objet de test dépend d'un autre composant, nous allons devoir simuler ce composants ou son comportement.



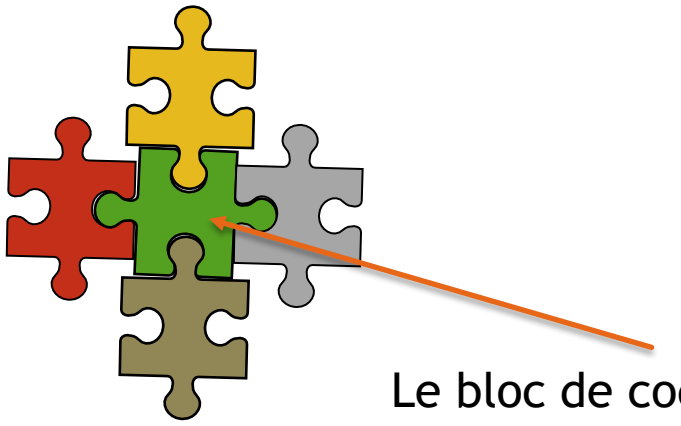
Le bloc de code à tester



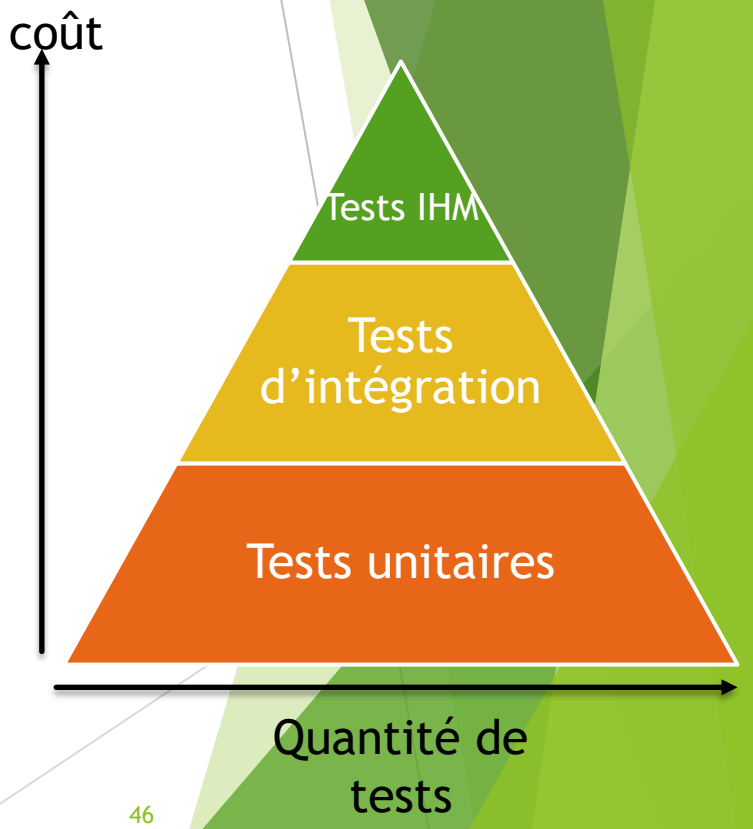
Tester une application Spring

Qu'est ce que c'est un test d'intégration

- Un test d'intégration vise à s'assurer du bon fonctionnement d'un composant applicatif (méthode, classe, module) en présence de plusieurs unités de programme, testés unitairement au préalable.
- le comportement des composants dont dépend notre test n'est pas simulé.

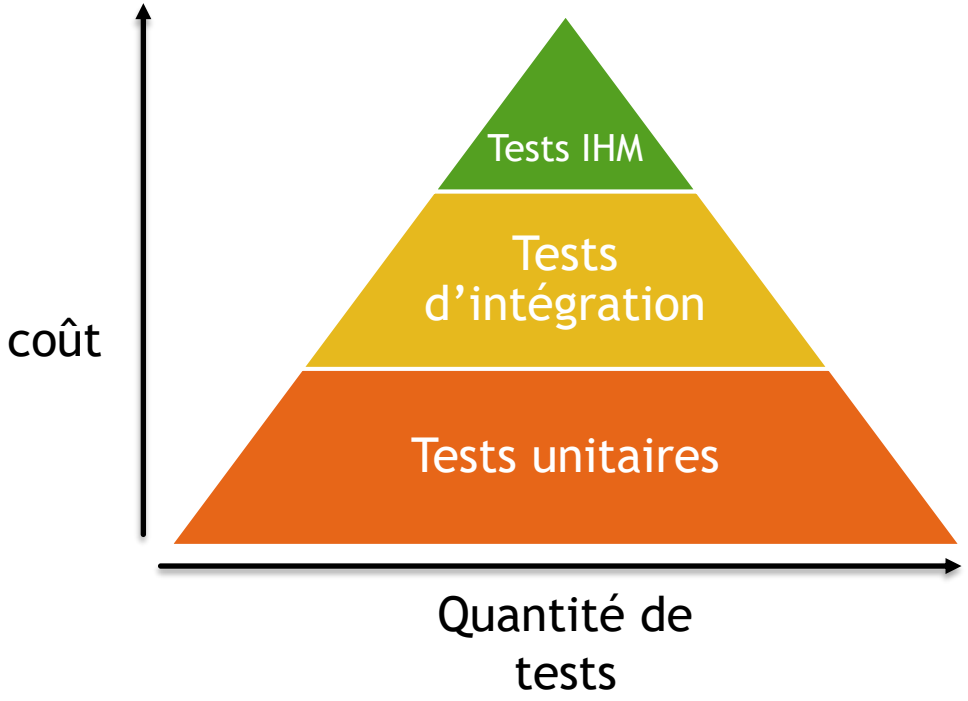


Le bloc de code à tester



Tester une application Spring
Qu'est ce que c'est un test IHM

- Validation de l'intégration du composant dans le système global (configuration, connectivité), composants graphiques faits maison



Tester une application Spring
Qu'est ce que c'est JUnit

- JUnit est un framework open source pour le développement et l'exécution de tests unitaires en Java

Tester une application Spring
Qu'est ce que c'est Mockito

- C'est un framework Java permettant de générer automatiquement des objets 'mockés'.
- Couplé avec JUnit, il permet de tester le comportement des objets réels associés à un ou des objets 'mockés' facilitant ainsi l'écriture des tests unitaires.
- Il propose des fonctionnalités très utiles au-delà de la simple simulation d'une valeur de retour comme :
  - la simulation de cas d'erreurs en levant des exceptions,
  - la validation des appels de méthodes,
  - la validation de l'ordre de ces appels,
  - la validation des appels avec un timeout.

Tester une application Spring
Qu'est ce que c'est AssertJ

- assertJ est une bibliothèque d'assertion plus complète et fortement typée, inspirée d'une autre bibliothèque, fest-assert.

Tester une application Spring
Module spring-test

- **SpringJUnit4ClassRunner** : permet de cacher un ApplicationContext durant tous les tests.
  - Tous les tests sont exécutés dans le même context en utilisant les même dépendances.
- @**ContextConfiguration** : Préciser l'endroit où se trouvent les beans à utiliser durant les tests
  - **Classes**
  - **locations**

Tester une application Spring
Module spring-test

- Si les attributs locations et classes attribute ne sont pas renseignés par défaut, Spring va chercher d'abord
  - Un fichier xml [nom de la classe de test]-context.xml
  - S'il ne le trouve pas, il va chercher toutes les inner classes annotées @Configurations définies dans la classe de test

Tester une application Spring
Module spring-test : @ActiveProfiles

- Spring-test permet d'activer le profile associer à la classe de test moyennant l'annotation @ActiveProfile
- @ActiveProfiles permet de lister les profils pour lesquels il faut lancer le test.

La POA
(programmation orientée aspect)

La POA (programmation orientée aspect)
Le contexte

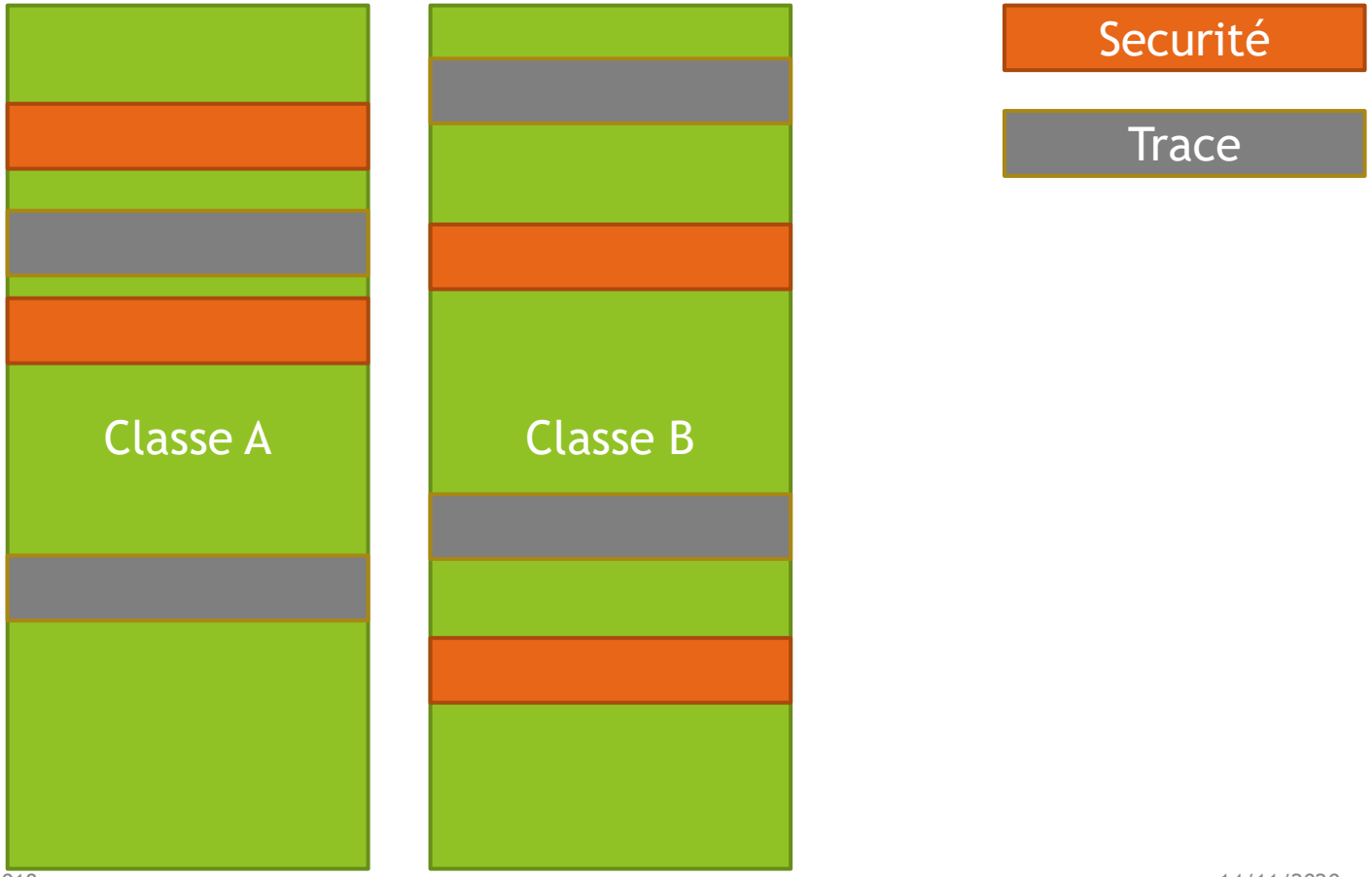
Préoccupation d'ordre fonctionnel

- Logique métier

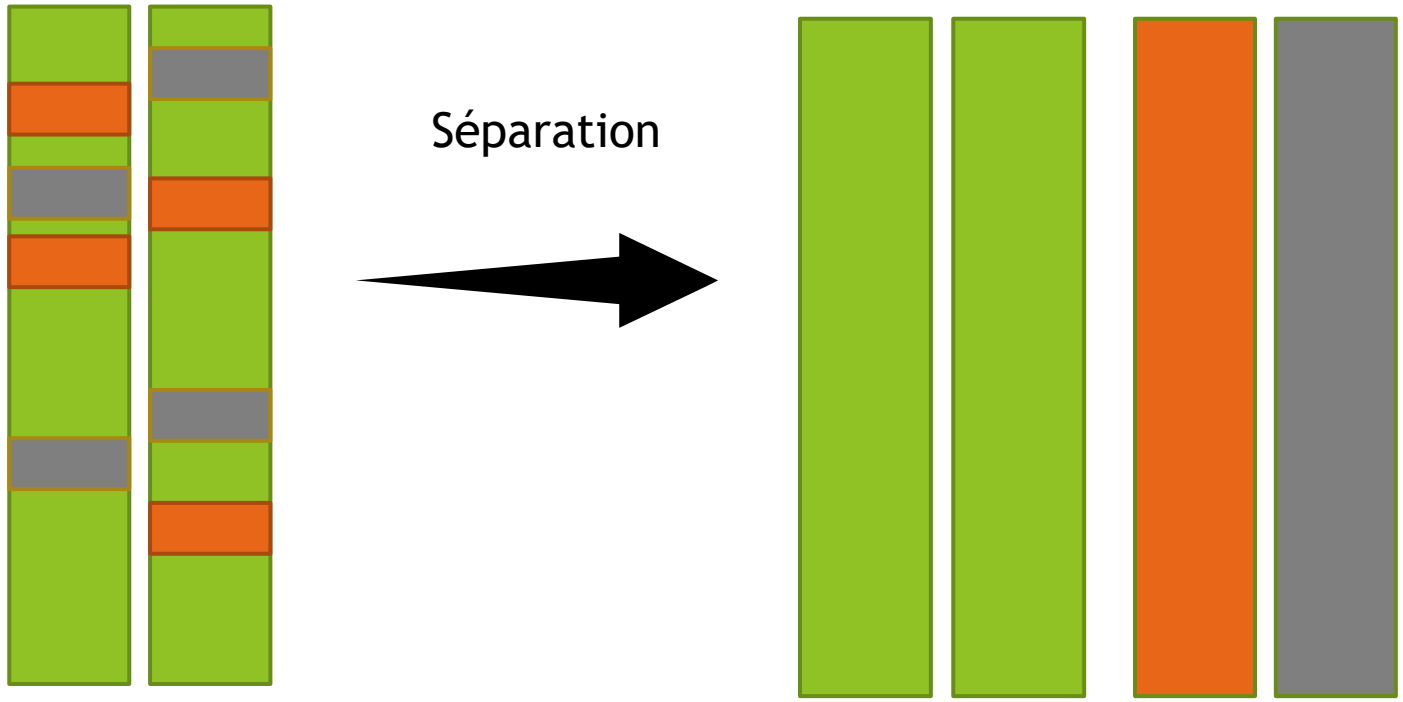
Préoccupation d'ordre technique

- Sécurité
- Gestion des transactions
- Journalisation
- …….

La POA (programmation orientée aspect)
Le contexte



La POA (programmation orientée aspect)
Objectif



La POA (programmation orientée aspect)
Limites de la POO

- Avec l'approche orientée objet,
  - Les fonctionnalités transversales sont implémentées dans chaque classe concernée
  - Une évolution de ces fonctionnalités transversales implique la modification de plusieurs classes.

La POA (programmation orientée aspect)
POA

- La POA (programmation orientée aspect) :
  - Proposer un moyen de centraliser dans une nouvelle entité le code d'une fonctionnalité transversale et ainsi la séparation entre les aspects métier et les aspects technique

La POA (programmation orientée aspect) Terminologie

- la POA vient avec un certain nombre de mots clefs qui définissent et concrétisent ses concepts.
  - Aspect
  - Pointcut (coupure)
  - JoinPoint (Point de jonction)
  - advice (Greffon)
  - Weaver (Tisseur) : AspectJ, Spring AOP, Aspect Werkz

La POA (programmation orientée aspect)
Spring AOP

- AspectJ est la première librairie ayant fourni les composants pour la creation des aspects en 1995
- Spring AOP est un complément de la version en cours d'AspectJ

La POA (programmation orientée aspect)
Mon premier aspect

- Créer une classe Aspect en utilisant l'annotation @Aspect
- Déclarer un bean de type Aspect (@Component ou @Bean)
- Declare un advice (greffon)
- Associate un pointcut (coupure) à une expression
- Activer la configuration AOP => @EnableAspectJAutoProxy

La POA (programmation orientée aspect)
Pointcut

execution([**Modificateur**] [**Type_De_Retour**] [**Nom_De_La_Classe**].[**Nom_De_La_Methode**] ([**Arguments**]) throws [**Type_D_Exception**])

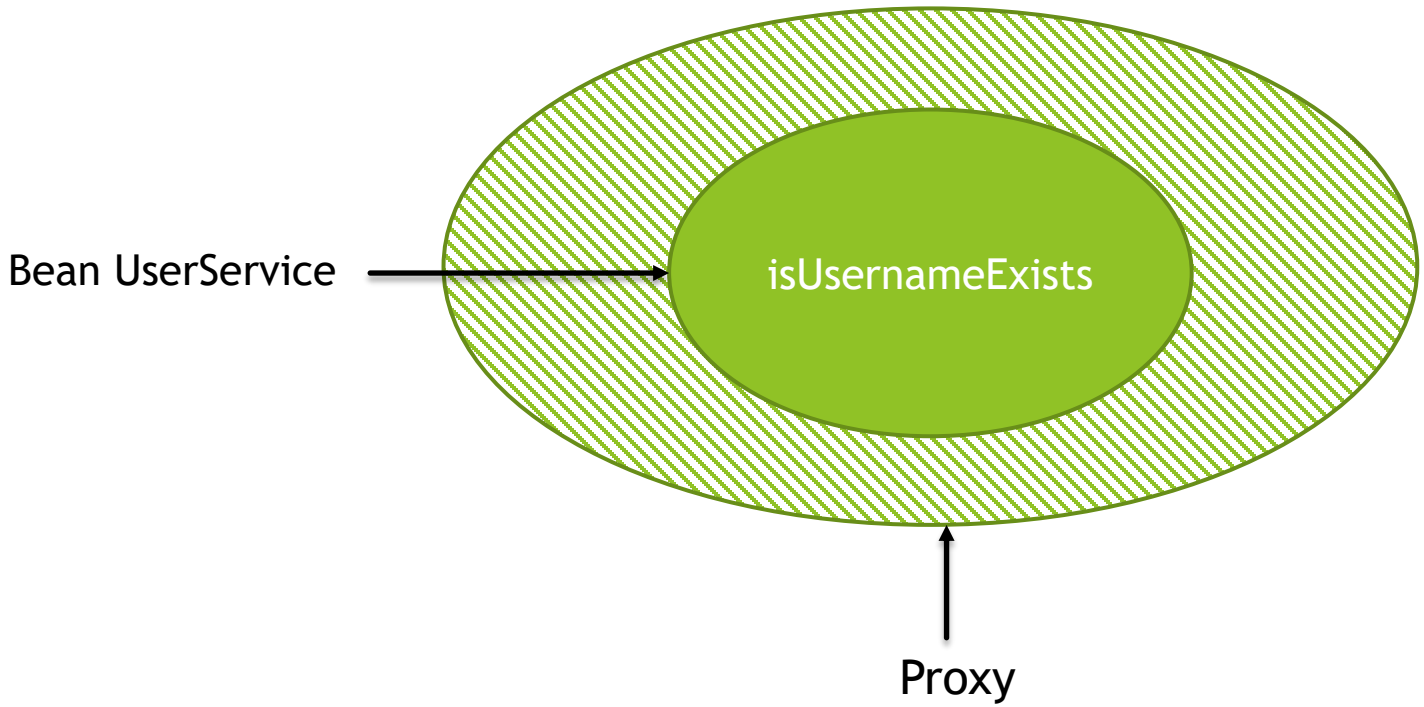
La POA (programmation orientée aspect)
Pointcut

- [**Modificateur**] : Il n'est pas obligatoire
  - Par défaut : public
- [**Type_De_Retour**] : est obligatoire.
  - Si il ne fait pas partie des critères, mettez *****
  - Si il est absent, l'application plante avec une exception IllegalArgumentException
- [**Nom_De_La_Methode**] :
  - Package + Classe + Nom de la méthode
  - Il n'est pas obligatoire
- [**Arguments**] est obligatoire.
  - Si il ne fait pas partie des critères, mettez *****
  - Si il est absent, l'application plante avec une exception IllegalArgumentException

La POA (programmation orientée aspect)
Pointcut

- On peut créer des coupures avec @Pointcut
- On peut regrouper les pointcuts dans une classe

La POA (programmation orientée aspect)



La POA (programmation orientée aspect)

- Proxies Dynamic JDK
  - L’objet cible implémente au moins une interface
- CGLIB
  - l'objet cible n'implémente aucune interface

La POA (programmation orientée aspect)
Advice (greffon)

- Before
- After returning
- After throwing
- After
- Around

La POA (programmation orientée aspect)
JoinPoint

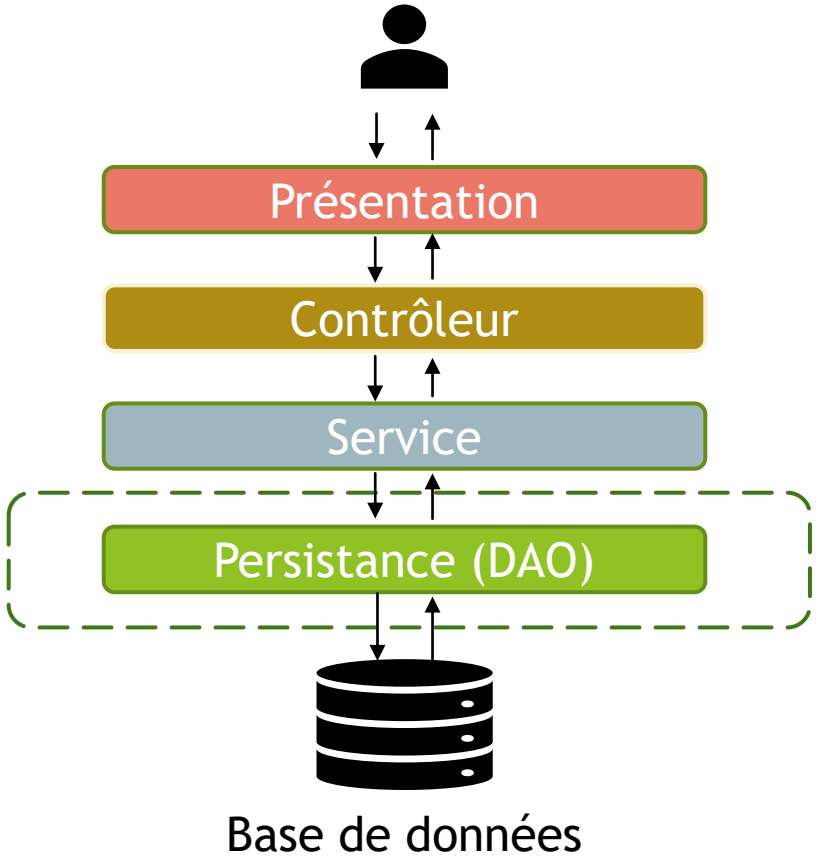
- La récupération des informations sur la méthode interceptée peut s'effectuer de manière fortement typé
  - Target()
  - Args()

La POA (programmation orientée aspect)
Conclusion

- Les aspects Spring AOP ne peuvent s'appliquer que sur des beans Spring .
- Les aspects Spring AOP ne peuvent être appliqués que sur des méthodes public et non static

Accès aux données

Accès aux données
la couche de persistance



Accès aux données
Le pattern DAO

- Ce pattern permet de faire le lien entre
  - la couche d'accès aux données (Système de stockage : ex, Base de données)
  - la couche métier d'une application : vos classes Java

Table Développeur

ID	Nom	prenom
1	ZAROUAL	Mohamed

DeveloppeurDaoImpl

```java
public class Developpeur {

    private Integer id;
    private String nom;
    private String prenom;

    public Integer getId() {
        return id;
    }

    public void setId(final Integer id) {
        this.id = id;
    }

    public String getNom() {
        return nom;
    }

    public void setNom(final String nom) {
        this.nom = nom;
    }

    public String getPrenom() {
        return prenom;
    }

    public void setPrenom(final String prenom) {
        this.prenom = prenom;
    }

}
```

Accès aux données
la classe DAO

- Une classe DAO a les tâches suivantes
  - Créer les données
  - Récupèrer les données
  - Mettre à jour les données
  - Supprimer les données

```
public interface FormationDao {

    void create(Formation formation);

    void update(Formation formation);

    void delete(Formation formation);

    Formation find(Long id);

    List<Formation> findAll();
}
```

Accès aux données
Requête SQL via JDBC

- Etablir une connexion avec la base de données
- Créer un *Statement*
- Exécuter la requête,
- boucler sur le *ResultSet* pour récupérer les résultats
- Fermer la connexion, le *Statement* et le *ResultSet*
- Gérer les transactions
- Gérer les exceptions

Accès aux données
Spring JDBC

- Définir la *DataSource*
- Spécifier la requête SQL
- Exécuter la requête,
- Définir comment traiter chaque élément dans les résultats

Accès aux données
Récupérer le résultat d'une requête

- **RowMapper<T>**
  - Mapper une ligne vers un objet
- **ResultSetExtractor<T>**
  - Le résultat est extrait de plusieurs tables mais devant être traité comme un seul objet
- **RowCallbackHandler**
  - Ecrire le résultat dans un fichier
  - Filtrer les résultats avant de les mettre dans une collection

Accès aux données
Requête SQL de mise à jour

- La méthode update() est utilisée pour
  - Ajouter
  - Mettre à jour
  - Supprimer

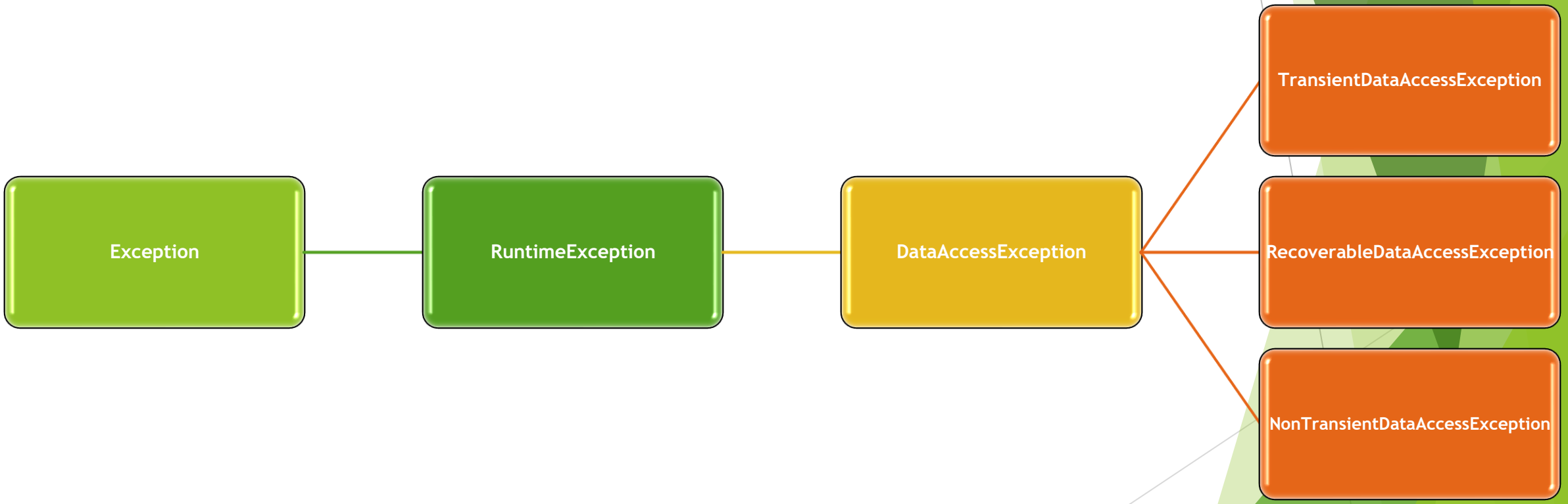
Accès aux données
Paramètres nommés

- NamedParameterJdbcTemplate
  - Map
  - MapSqlParameterSource

Accès aux données
Les exceptions

- public class SQLException extends Exception implements Iterable<Throwable>
- public int getErrorCode()

Accès aux données
Les exceptions



Accès aux données
Les exceptions

- NonTransientDataAccessException
  - CleanupFailureDataAccessException,
  - DataIntegrityViolationException,
  - DataRetrievalFailureException,
  - DataSourceLookupFailureException,
  - InvalidDataAccessApiUsageException,
  - InvalidDataAccessResourceUsageException,
  - NonTransientDataAccessResourceException,
  - PermissionDeniedDataAccessException,
  - UncategorizedDataAccessException

Accès aux données
Les exceptions

- TransientDataAccessException
  - ConcurrencyFailureException,
  - QueryTimeoutException,
  - TransientDataAccessResourceException

Accès aux données
Les exceptions

- RecoverableDataAccessException

Gestion des transactions

Gestion des transactions
Propriétés d'une transaction

Atomicité

- une transaction se fait au complet ou pas du tout

Cohérence

- La transaction amènera le système d'un état valide à un autre état valide

Isolation

- Aucune dépendance possible entre les transactions

Durabilité

- Les transactions assurent que les modifications qu'elles induisent perdurent, même en cas de défaillance du système

Gestion des transactions
La gestion des transactions par Spring-tx

- transaction manager (gestionnaire de transactions)
  - Seule la configuration du transaction-manager qui change
  - Implémentation de PlatformTransactionManager.

Technologie	Gestionnaire de transactions
JDBC	DataSourceTransactionManager
Hibernate	HibernateTransactionManager
JTA	JtaTransactionManager
JPA	JpaTransactionManager

Gestion des transactions
Configuration d'un transaction manager

- Configuration de la gestion des transactions
  - Ajout d'un bean **PlatformTransactionManager**
  - Activation de la gestion des transactions avec **@EnableTransactionManagement**
  - Implémentation de **TransactionManagementConfigurer** si l'application utiliser plusieurs datasource/transaction-manager

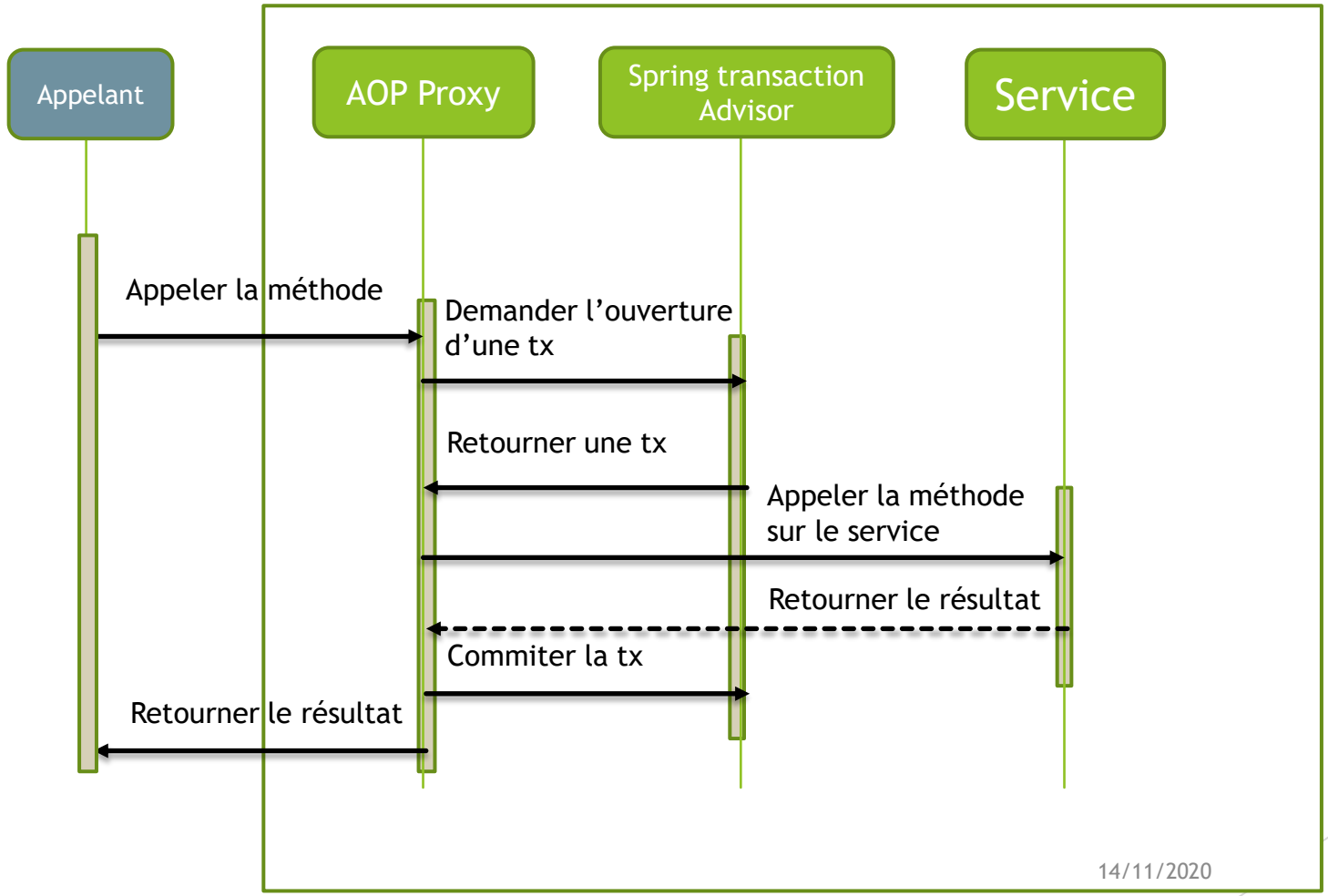
Gestion des transactions
La gestion des transaction par Spring-tx

- On distingue deux principales approches :
  - Par programmation
    - utilisation de l'API TransactionTemplate
  - Par déclaration
    - utilisation de proxys grâce à Spring AOP

Gestion des transactions
L'approche par programmation

- Par programmation
  - Création d'une formation
  - Ajouter une langue à la formation

Gestion des transactions
L'approche déclarative



@Transactional

Gestion des transactions
L'approche déclarative

- Annoter la méthode du service avec @Transactional
  - Attention, les méthodes annotées avec @Transactional devraient être public
  - @Transactional sur une classe, elle s'applique sur toutes les méthodes
  - @Transactional sur une classe asbtract, elle s'applique sur toutes les classes filles héritant de classe abstract
  - @Transactional sur une interface, elle s'applique sur toutes les classes implémentant l'interface

Gestion des transactions
Les attributs de @Transactional

- propagation : précise le mode de propagation de la transaction grâce à une énumération de type Propagation.
  - REQUIRED
  - MANDATORY
  - NEVER
  - NOT_SUPPORTED
  - REQUIRES_NEW
  - SUPPORTS
  - NESTED

Gestion des transactions
Les attributs de @Transactional

- isolation: : précise le niveau d'isolation de la transaction grâce à une énumération de type Isolation.

Gestion des transactions
Les attributs de @Transactional

- Il existe 3 types de violation
  - La lecture sale (Dirty read)
    - La transaction T1 modifie une ligne, la transaction T2 lit ensuite cette ligne,
    - Puis T1 effectue une annulation de la modification (rollback),
    - T2 a donc vu une ligne qui n'a jamais vraiment existé.
  - La lecture non reproductible
    - T1 extrait une ligne,
    - T2 met à jour cette ligne,
    - T1 extrait à nouveau la même ligne,
    - T1 a extrait deux fois la même ligne et a vu deux valeurs différentes.
  - La lecture fantôme
    - T1 lit quelques lignes remplissant certaines conditions de recherche,
    - T2 insère plusieurs lignes remplissant ces mêmes conditions de recherche,
    - Si T1 répète la lecture elle verra des lignes qui n'existaient pas auparavant. Ces lignes sont appelées des lignes fantômes.

Gestion des transactions
Les attributs de @Transactional

	Lecture sale	Lecture non reproductible	Lecture fantôme
READ_UNCOMMITTED	Oui	Oui	Oui
READ_COMMITTED	Non	Oui	Oui
REPEATABLE_READ	Non	Non	Oui
SERIALIZABLE	Non	Non	Non

Gestion des transactions
Les attributs de @Transactional

- readOnly : indique si la transaction est en lecture seule (false) ou lecture/écriture(true)
- **rollbackFor/ rollbackForClassname** : ensemble d'exceptions héritant de Throwable qui provoquent un rollback de la transaction si elles sont levées durant les traitements
- **noRollbackFor/ noRollbackForClassname** : ensemble d'exceptions héritant de Throwable qui ne provoquent pas un rollback de la transaction si elles sont levées durant les traitements
- **timeout** : entier qui précise le timeout de la transaction

Gestion des transactions
@Transactional : Service Vs DAO

- **Au niveau du service**
- **Au niveau du DAO**

Gestion des transactions
L'approche par programmation

- Ajout d'un bean de type **TransactionTemplate**
- Dans le service, injecter le bean **TransactionTemplate**
- Appeler la méthode execute():
  - TransactionCallback
  - TransactionCallbackWithoutResult

Gestion des transactions
Configurer la TransactionTemplate

- setIsolationLevel() : : précise le niveau d'isolation de la transaction grâce à une énumération de type Isolation.
  - ISOLATION_DEFAULT
  - ISOLATION_READ_UNCOMMITTED
  - ISOLATION_READ_COMMITTED
  - ISOLATION_REPEATABLE_READ
  - ISOLATION_SERIALIZABLE

Gestion des transactions
Configurer la TransactionTemplate

- setPropagationBehavior() : précise le mode de propagation de la transaction grâce à une énumération de type Propagation.
  - PROPAGATION_MANDATORY
  - PROPAGATION_NESTED
  - PROPAGATION_NEVER
  - PROPAGATION_NOT_SUPPORTED
  - PROPAGATION_REQUIRED
  - PROPAGATION_REQUIRES_NEW
  - PROPAGATION_SUPPORTS

Gestion des transactions
Configurer la TransactionTemplate

- setTimeout() : entier qui précise le timeout de la transaction
- **setReadOnly() :** indique si la transaction est en lecture seule (false) ou lecture/écriture(true)

Accès aux données
Tests d'intégration

- Configurer une base de données mémoire (H2)
- Injecter la classe DAO objet du test
- Rédiger les tests

Accès aux données
Tests d'intégration

- @Sql
- @SqlConfig
- @SqlGroup

Accès aux données
Tests d'intégration

- @Commit
- @Rollback
- @AfterTransaction
- @BeforeTransaction
- @DirtiesContext

Spring / JPA / Hibernate

Spring / JPA / Hibernate
Introduction

- ORM (object-relational mapping)
- JPA (Java Persistence API)
  - **Persistence**
  - **EntityManagerFactory**
  - **EntityManager**
- Hibernate

Spring / JPA / Hibernate Configuration

- Configurer les classes @Entity
- Plus besoin de configurer un META-INF/persistence.xml
- LocalContainerEntityManagerFactoryBean
- JpaTransactionManager
- Configurer la couche Dao