De Spring à Spring Boot

Développer des applications d'entreprise avec Spring 6

la mécanique de base avec Spring Framework

sommaire

– architecture logicielle : pour quoi faire ?
– vous avez dit design patterns ?
– et Spring dans tout ça ?
– IoC et injection de dépendances
– AOP : Aspect Oriented Programming
– les tests unitaires
– rappel des bonnes pratiques
– documentation
– annexes

architecture logicielle : pour quoi faire ?

architecture logicielle : pour quoi faire ?

L'architecture logicielle est :

– la spécification d’un système décrivant ses composants par leurs comportements, leurs responsabilités et leurs interfaces
– la description des interconnexions entre ces différents composants

L'architecture logicielle permet :

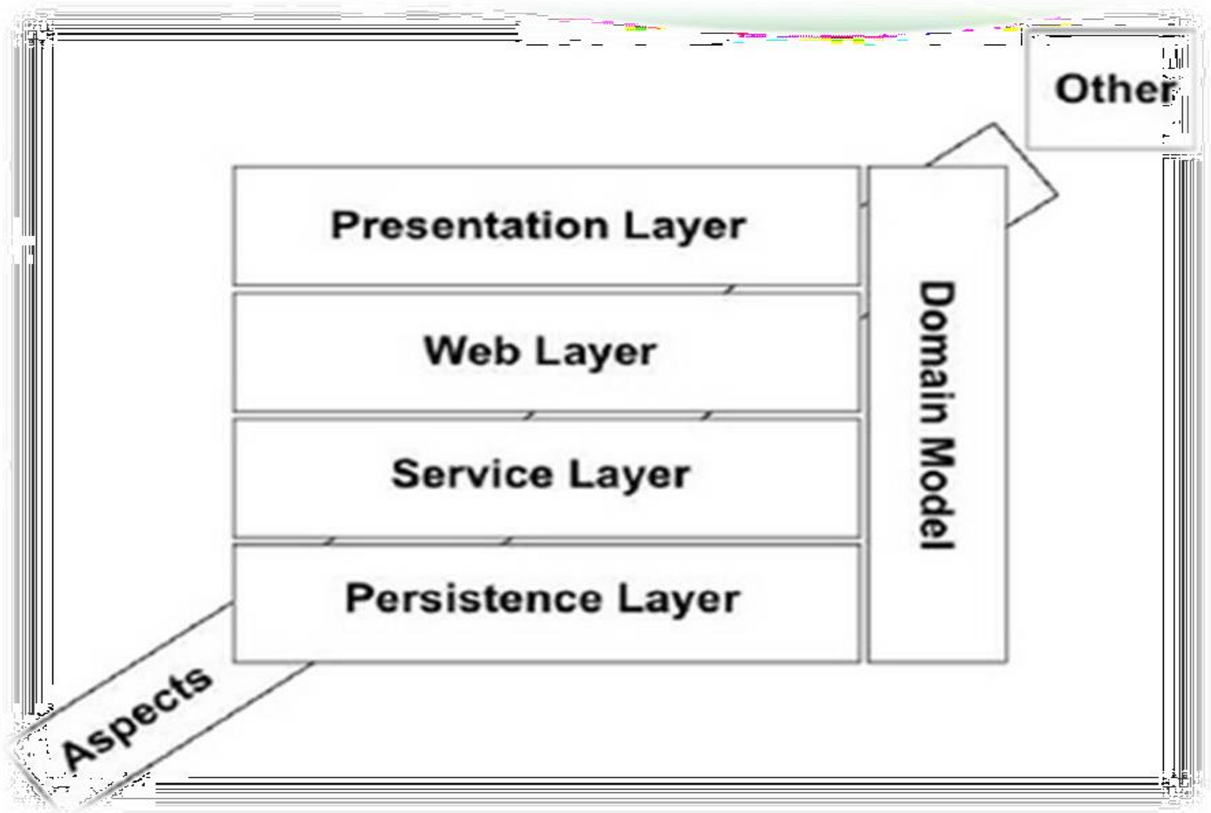
– d'assurer une meilleure évolutivité du système et d'identifier ses limites
– d'isoler les développements spécifiques notamment pour intégrer les progiciels
– de prendre en compte des problématiques standards récurrentes en informatique : performance, maintenabilité, ergonomie, sécurité...
– de capitaliser et de réutiliser des composants adressant des besoins communs
– d'identifier et appliquer des patrons de conception (**design patterns**).

architecture logicielle

architecture en couches logicielles

> *Exemple d'architecture en couches logicielles*

- The Architecture
  - Layered / N-Tiered
    - Presentation Layer
    - Web Layer
    - Service Layer
    - Persistence Layer
  - Aspects
  - Middleware
  - Other



architecture logicielle

architecture en couches logicielles

Le but d'une architecture en couches logicielles est d'isoler :

- les parties métiers
- les parties en communication avec l'extérieur (exposition/consommation de services, accès à une base de données...)

Le code métier constitue la valeur ajoutée de l'application. Elle est la plus stable d'un projet.

Les couches *provider* et *consumer* sont dépendantes de l'extérieur, et sont amenées à bouger beaucoup plus régulièrement :

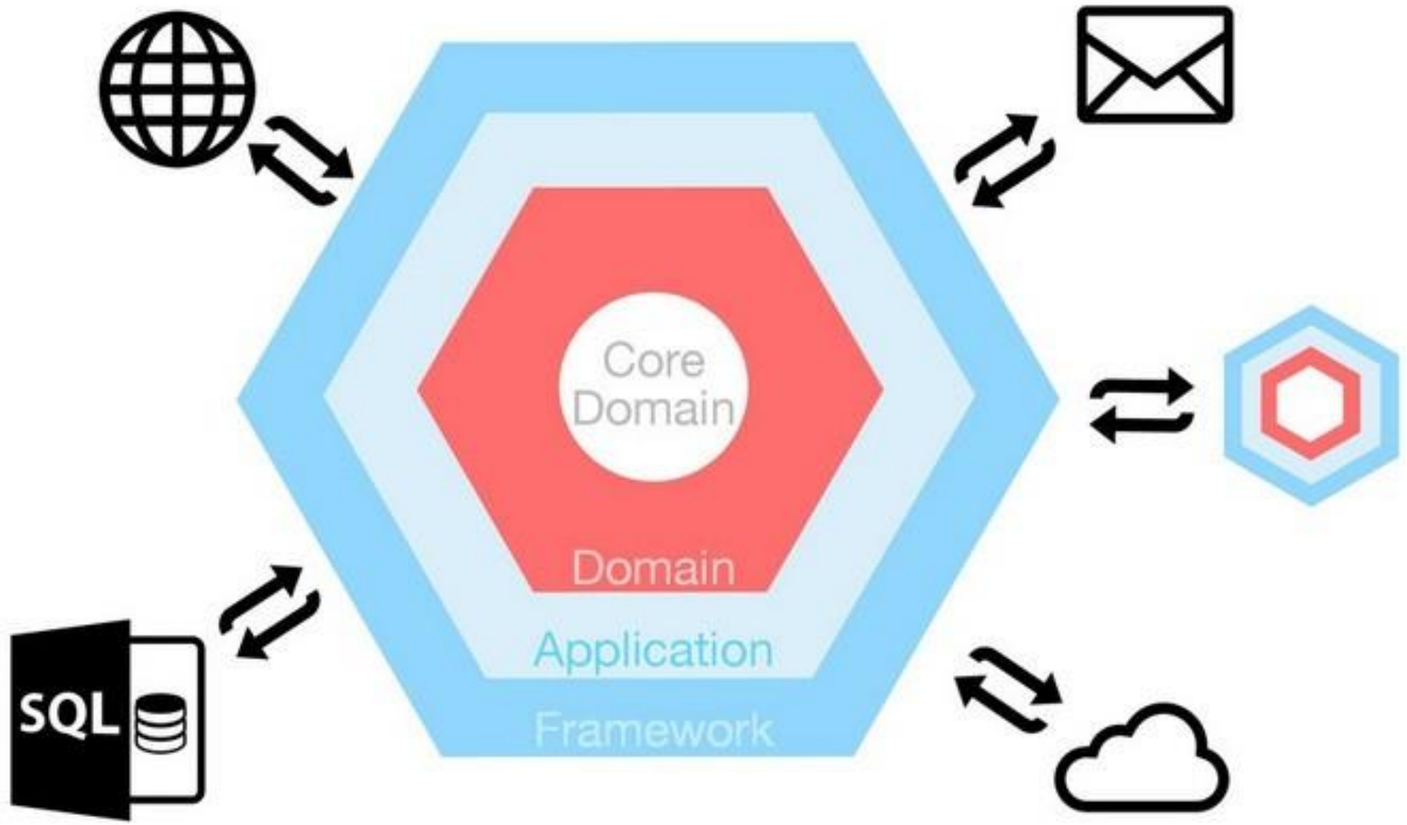
- changement de technologie (webservice SOAP => API REST, JDBC direct => mapping objet/relationnel...)
- changement des partenaires eux-memes
- ...

Ces changements périphériques doivent avoir le moins d'impact possible sur le coeur du projet.

architecture logicielle

architecture hexagonale

> *Exemple d'architecture hexagonale microservices*



architecture logicielle

architecture hexagonale

2 principes de base :

– séparation stricte entre code métier (use-cases) et code technique
– inversion de dépendance : le code technique dépend/est au service du métier et pas l'inverse.

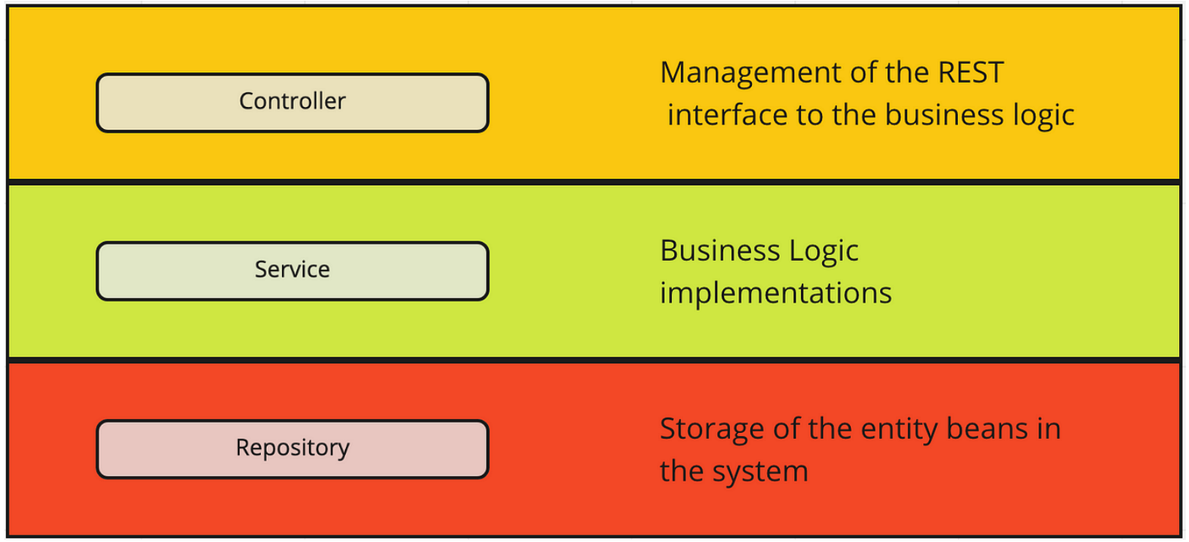
La définition du métier et de ses contraintes permettent de déterminer quel socle technique (langage...) mettre en place pour répondre à ces contraintes.

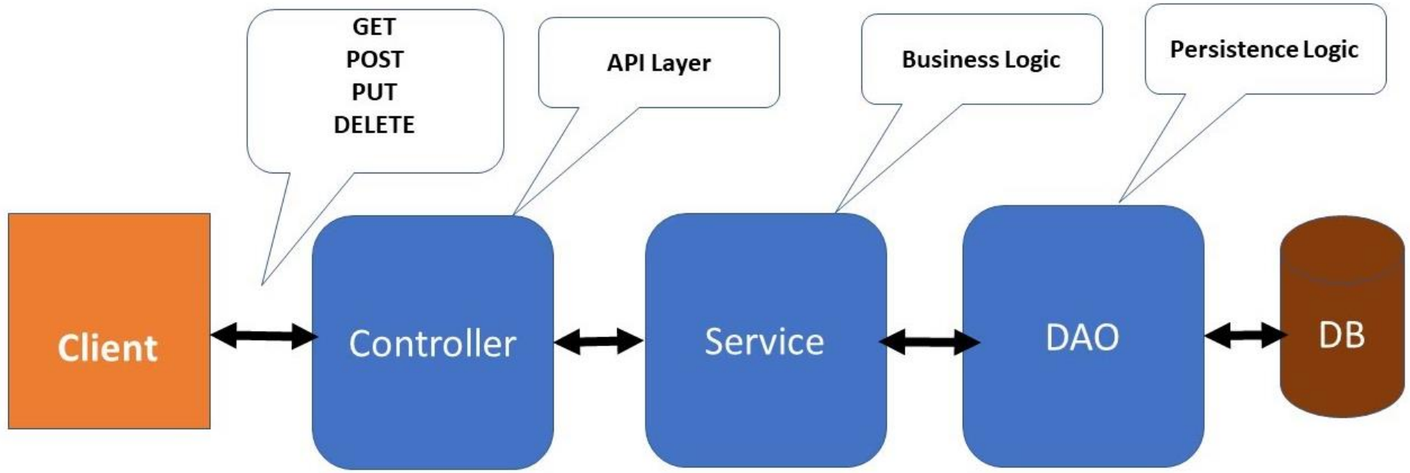
Cela implique que :

–le code métier (noms des packages, des classes, des méthodes, des variables...) doit rappeler les concepts manipulés sans biais technique. Cela permet de **mieux communiquer entre développeurs et experts métiers**
–le code technique permettant la communication vers l'extérieur se fait via des **connecteurs**, facilement remplaçables sans impacter le code métier
–**les tests métiers sont dissociés des tests techniques (des connecteurs) et sont plus faciles à écrire**. Lors d'un changement de connecteur, seuls les tests techniques liés à celui-ci auront besoin d'être réécrits.

Le DDD (Domain Driven Design) se marie bien avec l'architecture hexagonale.

Application d'entreprise





Framework

Framework fait partie de la mise en œuvre de l'architecture.

Un **Framework** se compose d'une ou plusieurs bibliothèques. Le l'application s'enregistre auprès du framework (souvent en implémentant une ou plusieurs interfaces), et le framework appelle l'application.

Un Framework existe souvent pour traiter un domaine à usage général particulier (tel que des applications Web ou des flux de travail, etc.).

Framework et bibliothèque

* A Framework is an architecture
  * A well-defined structure to solve a problem

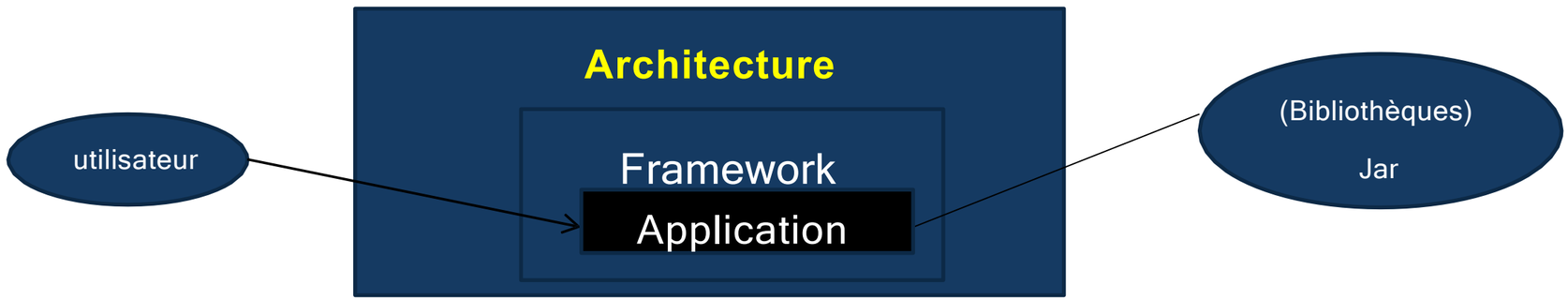
Les bibliothèques sont des composants qui contiendront le code réutilisable pour résoudre un problème.

* Library
  * Framework vs. Library
    * Invoking vs. being invoked

• L'application est un ensemble organisé d'instructions/commandes basées sur l'architecture et construites sur un Framework

• Le Framework rend l'organisation plus facile, structurée

• À partir des instructions du framework, l'application exécute l'aide à la sélection des bibliothèques



Composants façonnant une application EE

- Enterprise Application
  - Requirement/Business
  - Architecture of the application (not bound to a framework)
  - Technology
    - Language
    - Framework
    - Libraries
    - UI components
    - Service Protocol (SOAP, REST)
  - Design and Development
    - Coding
      - Classes and Objects
    - Testing
    - Maintenance

Composants façonnant une application EE

- Les classes sont la structure de l'application
  - Ils définissent comment un objet doit être structuré/créé
  - État via les propriétés
  - Comportement via les méthodes

- Les objets sont les donneurs de vie d'une application
  - Plusieurs objets collaborent pour créer un système entièrement fonctionnel
  - Ils interagissent en fonction des propriétés attribuées
  - Le comportement est défini par l'état de l'objet à un moment donné

    - Par exemple. Utilisateur -> {propriétés : nom d'utilisateur, mot de passe} -> doLogin(utilisateur)
      - Connexion réussie
      - Erreur d'authentification

Composants façonnant une application EE

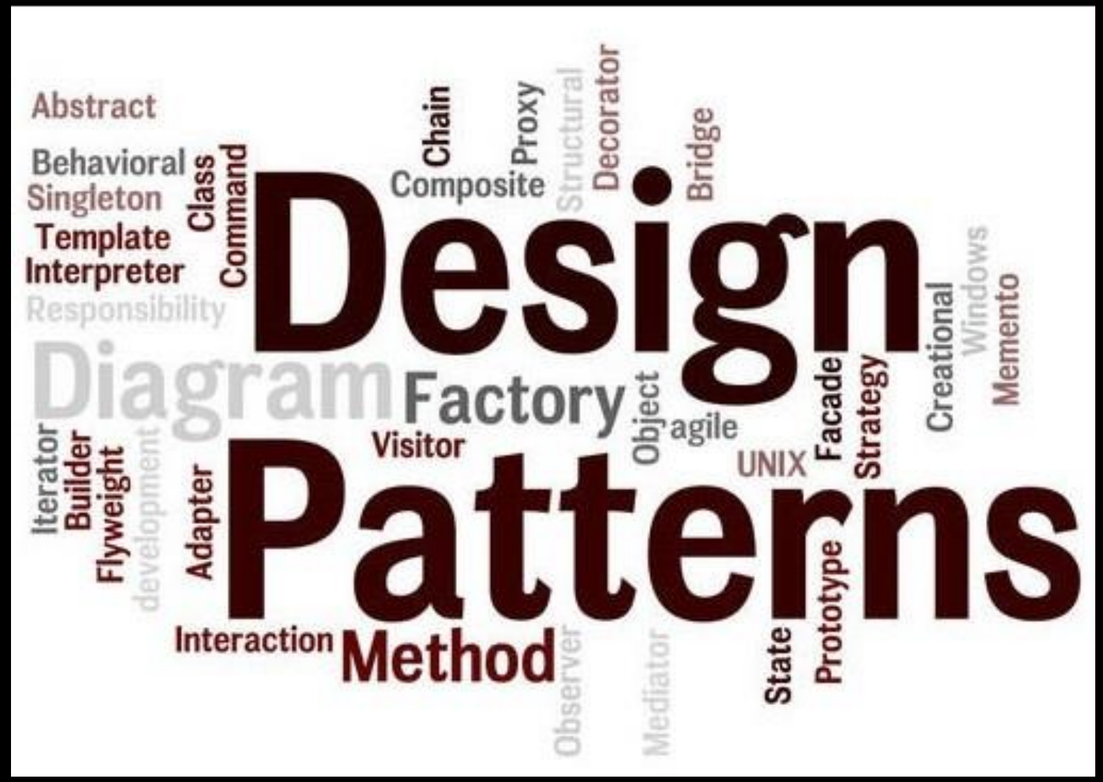
- Pour qu'une application fonctionne avec succès, nous devons faire attention à,
  - Comment les objets sont créés ?
  - Comment sont-ils organisés/assemblés ?
  - Comment sont-ils contrôlés ?
  - Comment la mémoire est-elle gérée ?

- Problèmes de développeur
  - Du point de vue du développeur, le codage pour les entreprises + ce travail d'infrastructure est fastidieux et prend du temps
  - Réécriture de code répété
  - Problèmes de débogage
  - Confusions portant à la fois sur l'infrastructure technique et le business
- Spring
  - Aborde la création/maintenance d'objets
  - Infrastructure commune via des API, empêchant le code passe-partout
  - Donne plus de temps pour traiter les affaires

Application d'entreprise

- Il représente un flux métier conçu pour un client spécifique Les
- exigences du métier sont les facteurs clés qui construisent l'architecture de l'application
- En fonction de l'architecture, un ou plusieurs frameworks spécifiques sont choisis
  - Spring, Struts – pour le web
  - Oracle, MySQL – pour les définitions de base de données
  - Hibernate - pour les spécifications liées à l'ORM
  - Jquery, Angular js – Pour une interface utilisateur interactive
  - CSS, HTML5, Bootstrap – Pour personnaliser l'apparence
  - XML, JSON – pour les normes de transfert de données
  - SOAP, REST – pour les spécifications liées au service

vous avez dit design patterns ?



vous avez dit design patterns ?

Les développements d'applications rencontrent toujours plus ou moins les mêmes problématiques à résoudre.

Un **design pattern** est un principe de solution éprouvée à un problème de conception fréquemment rencontré, sous la forme d'une organisation de classes utilisable dans la majorité des **langages de programmation objet** notamment (mais pas exclusivement).

Intérêts :

– ne pas réinventer (capitalisation d'expérience)
– écrire du code (+ ou -) facilement compréhensible par les autres programmeurs (vocabulaire commun).

Inconvénients :

– nécessite un effort de synthèse (reconnaître, abstraire...)
– demande un apprentissage.

vous avez dit design patterns ?

framework

Un design pattern peut être implémenté dans différents langages de programmation et de différentes façons.

Etant donné qu'il s'agit d'apporter une même solution à un problème récurrent, il est possible d'en faire une abstraction afin d'automatiser sa mise en oeuvre via les **frameworks**.

Un framework est une solution qui aide à structurer le code applicatif et à résoudre certains types de problèmes dans un langage de programmation donné.

Avantages :

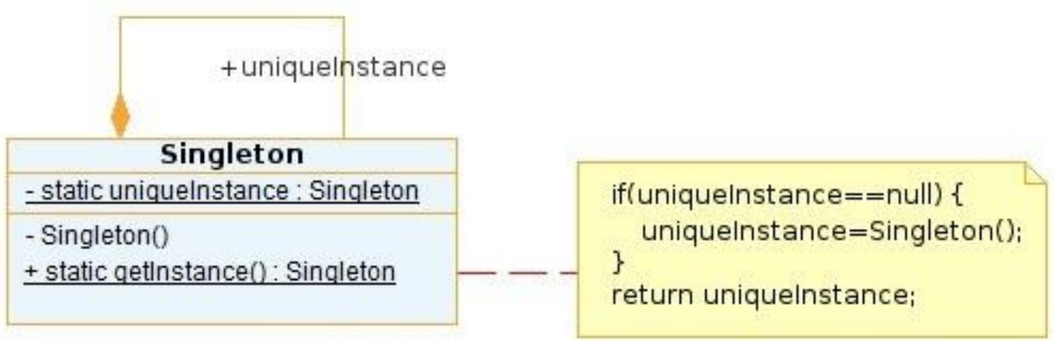
– moins de code à écrire, donc gain de temps
– développement plus robuste.

> *Exemples de frameworks :*
>
> – *Spring : patterns IoC, AOP...*
> – *Hibernate : pattern DAO*
> – *...*

vous avez dit design patterns ?

pattern singleton

– **but** : s'assurer qu'une classe n'a qu'une seule instance, et y accéder facilement
– **intérêts** :
    - un objet unique pour centraliser certaines opérations (configuration...) d'un système
    - meilleure efficacité en évitant d'avoir une multitude d'objets strictement identiques en mémoire
– **comment ça marche** :

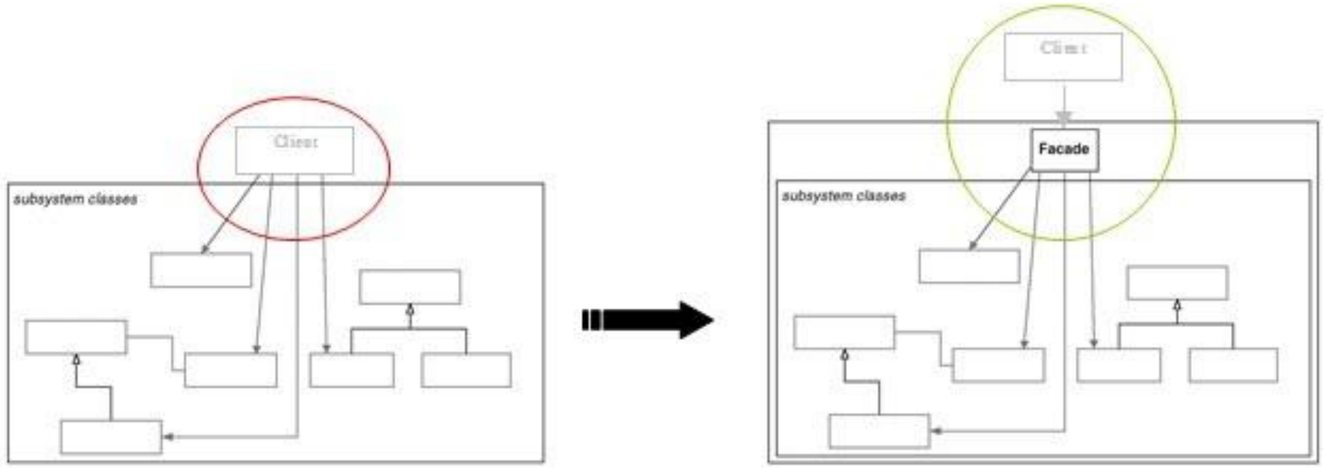


> ***exemple de cas d'usage*** *: instancier un unique pool de N connexions à la base de données, à la première tentative de connexion à la base par exemple, ou mieux, au démarrage de l'application. Il ne faut surtout pas créer autant de pools de connexions que d'appels à la base de données (performances dégradées, voire arrêt de l'application).*

vous avez dit design patterns ?

pattern façade

– **but** : fournir une interface unique et de haut niveau d'un sous-système (exemple : API)
– **intérêts** :
    - simplifier l'utilisation d'un sous-système en masquant sa complexité aux clients
    - meilleure évolutivité en cas de modification du sous-système
    - capitaliser en un endroit l'effort que devraient faire N clients
– **comment ça marche** :



> ***exemple de cas d'usage*** *: le virement entre 2 comptes bancaires nécessite plusieurs opérations (existence des comptes, solvabilité, débiter l'un, créditer l'autre...). Le sous-système fournit une façade* ***virement(débiteur, créditeur, montant)*** *et se débrouille pour remplir sa tâche, quelle que soit sa complexité.*

vous avez dit design patterns ?

pattern Inversion of Control (IoC)

Il existe 2 façons principales d'injecter les dépendances en Java, sans design pattern : par **setter** et par **constructeur**.

> *Exemple d'injection par setter*

```
ComponentA ca = new ComponentA();
ServiceB sb = new ServiceBImpl();
ca.setServiceB(sb);
```

> *Exemple d'injection par constructeur*

```
ServiceB sb = new ServiceBImpl();
ComponentA ca = new ComponentA(sb);
```

Problème : pour changer d'implémentation (utiliser **ServiceBImplBis**), il faut modifier le code, rebuilder, etc...

vous avez dit design patterns ?

pattern Inversion of Control (IoC)

L'inversion de contrôle est un **patron d'architecture**, implémenté en général grâce au design pattern d'injection de dépendances.

– **buts** :

-laisser un composant externe (framework) prendre en charge l'exécution principale d'un programme (ce n'est plus l'application qui gère les appels au framework, mais le framework qui appelle les composants)

- injecter dans les objets le code dont ils ont besoin

– **intérêts** :

- découpler les composants entre eux

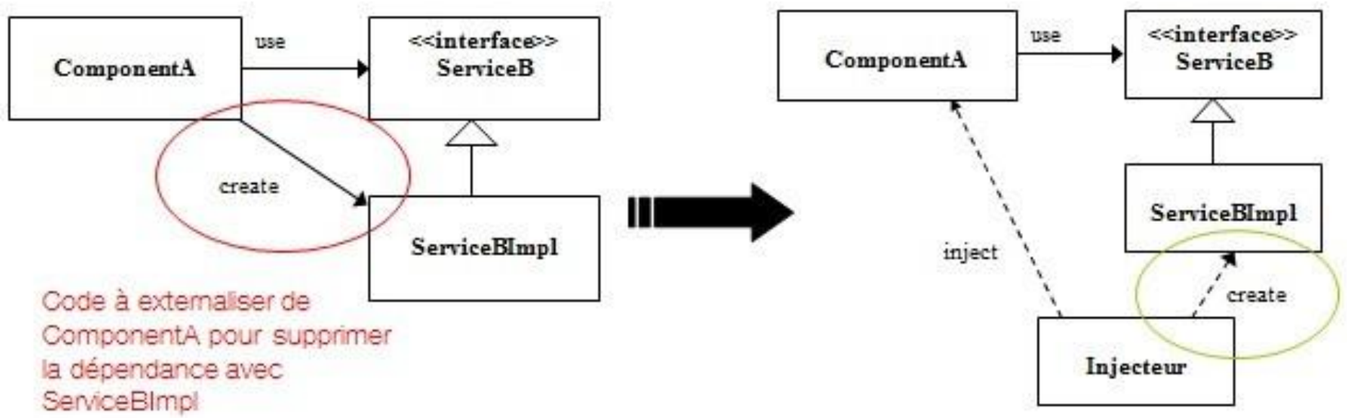
-ne plus exprimer les dépendances entre composants de façon statique directement dans le code, mais de façon dynamique, à l'exécution, grâce à un fichier de configuration ou des métadonnées

- permettre de modifier les classes à utiliser selon un contexte d’utilisation (tests, changement de plateforme, version du produit...).

vous avez dit design patterns ?

pattern Inversion of Control (IoC)

– comment ça marche :



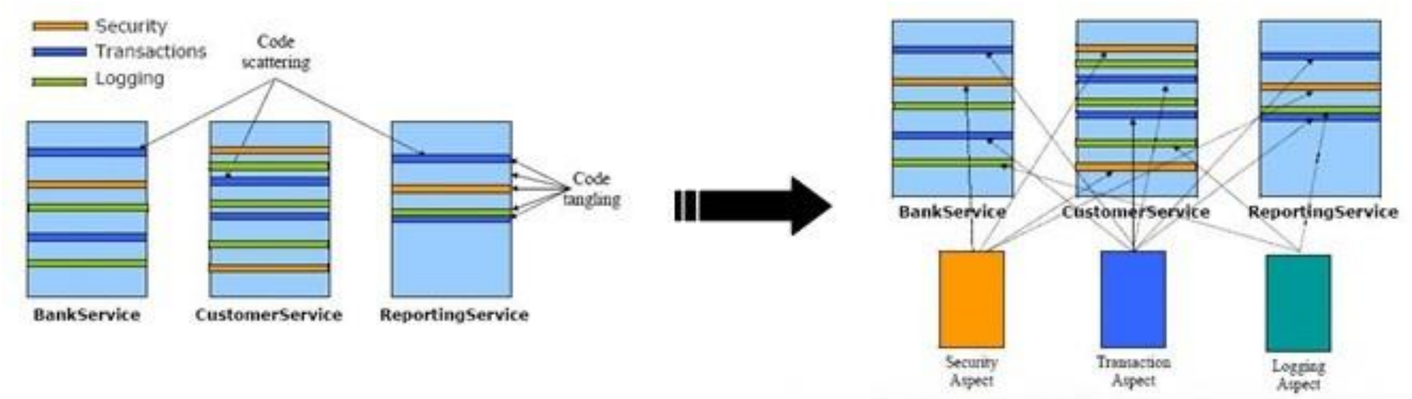
exemple de cas d'usage *: mise en oeuvre pour le code de l'application, pour lier les composants entre eux.*

Un composant *'Injecteur'* se charge d'instancier et d'injecter l'implémentation souhaitée de l'interface **ServiceB** dans **ComponentA.**

vous avez dit design patterns ?

Programmation Orientée Aspects (AOP)

– comment ça marche :



> ***exemple de cas d'usage*** *: mise en oeuvre pour la plomberie de l'application (gestion des logs, de la sécurité, des transactions...).*

La **programmation par Aspect (AOP : Aspect Oriented Programming)** permet d'externaliser dans des composants dédiés, les mécanismes techniques transverses (plomberie) dupliqués partout dans le code métier. Celui-ci est épuré et le code technique est injecté aux moments souhaités (logs pour toutes les entrées/sorties de méthodes...).

et Spring dans tout ça ?

et Spring dans tout ça ?

historique

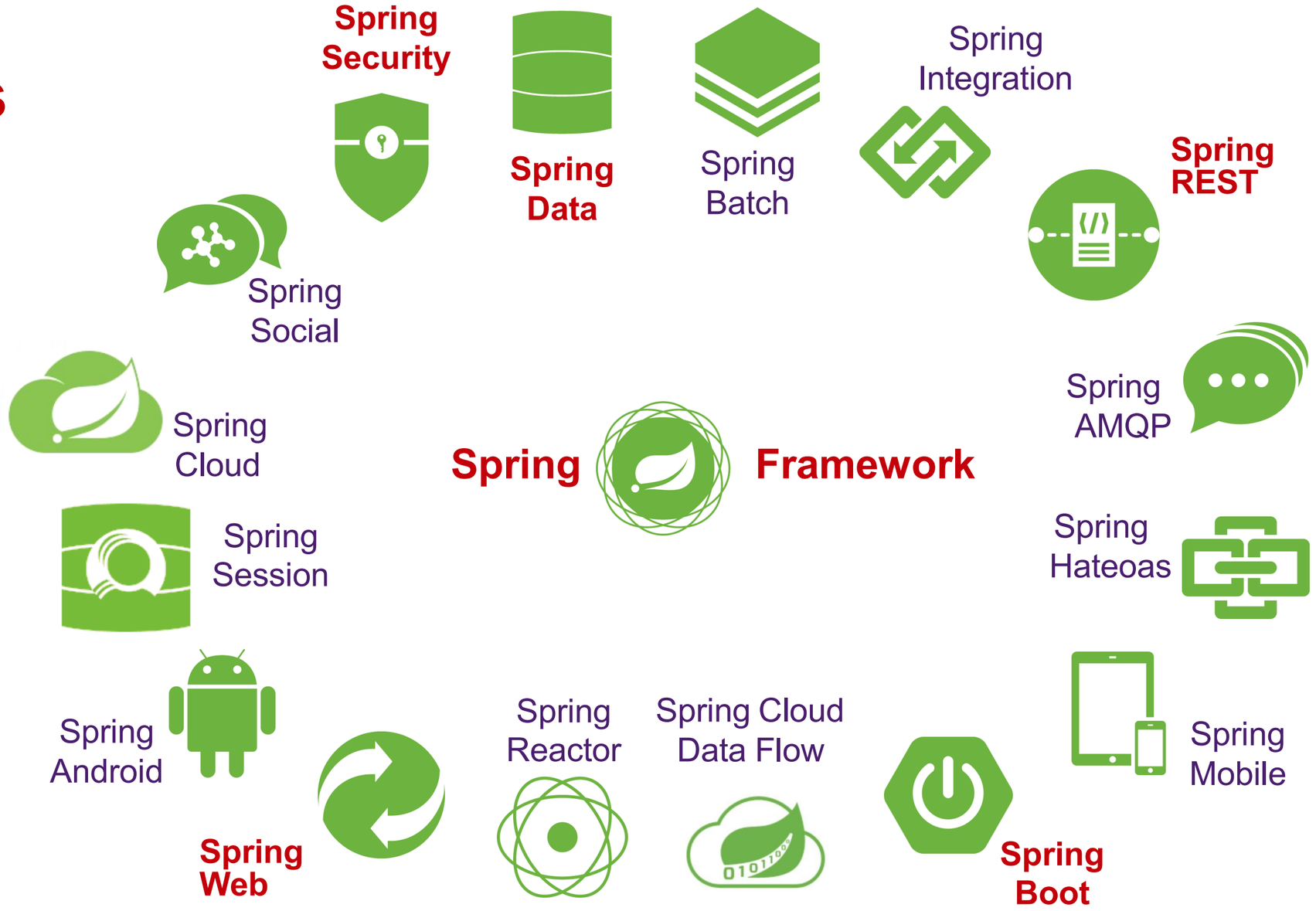
Spring Framework existe depuis +20 ans et évolue régulièrement :

- Spring 1.0 en mars 2003
- Spring 2.0 en octobre 2006
- Spring 3.0 en décembre 2009
- Spring 4.0 en décembre 2013
- Spring 5.0 en septembre 2017 (Java 8 minimum, programmation Reactive, support de Kotlin)
- Spring 6.0 en Novembre 2022 (Java 17, Jakarta API, HttpClient, …)

La galaxie **Spring** s'est étoffée de nombreux projets, dédiés chacun à leur domaine :

- **Spring Boot** : création et déploiement rapide d'applications Spring
- **Spring Security** : gestion de l'authentification et des habilitations
- **Spring Web MVC** : création d'API REST ou d'IHM selon le modèle MVC
- **Spring HATEOAS** : création de représentations REST selon le principe HATEOAS
- **Spring Cloud** : outils pour les systèmes distribués, notamment les microservices
- **Spring Data** : gestion de la persistance sur bases de données relationnelles, NoSQL...
- **Spring Batch** : simplification des batchs
- **Spring Integration** : support des patterns d'intégration d'entreprise
- **...**

Spring Projects



et Spring dans tout ça ?

objectifs

Spring est un **framework généraliste** (pas dédié à une couche précise d'une application).

Son but est de laisser le développeur se concentrer sur le code métier à **valeur ajoutée** de l'application, Spring Framework étant responsable de la *plomberie* :

– configuration des composants applicatifs, en les rendant indépendants les uns des autres et facilement interchangeables
– intégration des services et des ressources : accès aux bases de données, transactions, sécurité, monitoring, messaging...
–support des principales normes, interfaces et frameworks standards correspondants : JDBC, JPA (Hibernate), API Rest, webservices Soap (CXF), MOM, mBeans JMX...

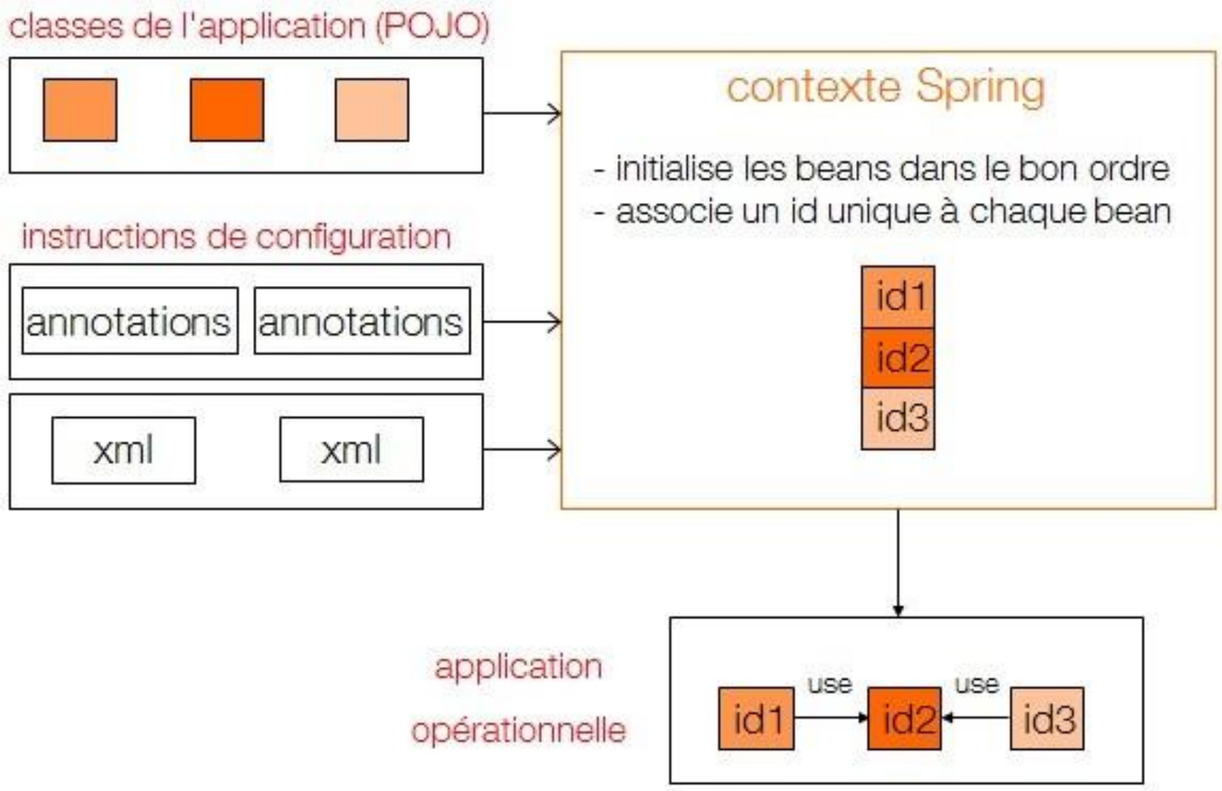
Spring Framework facilite les **tests** :

–en découplant les objets de leur contexte, il est plus simple de les tester individuellement, en remplaçant les classes en dépendances par des bouchons (mocks)
– intégration avec le framework de tests unitaires JUnit et le framework Mockito pour créer des bouchons.

et Spring dans tout ça ?

le contexte Spring

Spring manipule des **beans managés** (dont on lui aura confié la gestion), qui sont des instances de classes de l'application.



et Spring dans tout ça ?

chargement du contexte Spring

Spring se charge de créer et d'initialiser les beans de l'application suivant 3 types de configurations possibles :

– grâce aux **annotations Java** dans le code (recommandé)
– via des fichiers XML (voir en annexes)
– en mixant les 2 (voir en annexes).

Les beans sont créés dans le bon ordre car Spring analyse les dépendances dans la configuration.

En Spring Boot, une annotation **@SpringBootApplication** placée sur la classe principale de l'application joue ce rôle (entre autes) de chargement des beans.

Pour en savoir plus, rendez-vous au chapitre : Le chef d'orchestre Spring Boot.

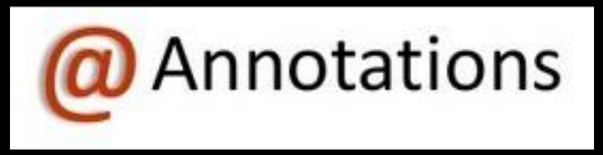
et Spring dans tout ça ?

scope des beans Spring

Chaque bean Spring est associé à un **scope** définissant quand il sera instancié :

- **singleton** (scope par défaut) : une instance unique est créée dans l'application
  - *cas d'usage : classes stateless de traitement (ex : distributeur de tickets)*
- **prototype** : une nouvelle instance est créée à chaque fois que le bean est demandé
  - *cas d'usage : classes statefull (ex : les tickets créés par le distributeur)*
- **request** : une instance est créée à chaque requête HTTP
  - *cas d'usage : créer une instance de log à chaque requête HTTP*
- **session** : une instance est créée pour chaque session HTTP
  - *cas d'usage : 1 bean utilisateur pendant toute sa session (caddie...)*
- **Application** : équivalent à **une variable d'application**
  - *cas d'usage : 1 bean utilisateur partagé par toutes les instances d'une application*

IoC et injection de dépendances



IoC et injection de dépendances

définition des beans

Il faut déterminer quels sont les beans managés (et leur scope) de l'application :

– **@Named** : annotation standardisée Java (JSR-330)
– **@Component** : annotation générique propriétaire et historique de Spring. Son fonctionnement est similaire à **@Named**
– **@Controller** : spécialisation pour les contrôleurs MVC (couche présentation), permettant l'ajout de l'annotation **@RequestMapping** pour le mapping d'URL sur les méthodes de la classe (cf : Spring MVC)
– **@Repository** : spécialisation pour les DAO, les unchecked exceptions sont transformées en **DataAccessException** Spring
– **@Service** : spécialisation pour les couches de service (applicatif, métier...) sans comportement additionnel par rapport à **@Component**.

Par défaut, le bean sera référencé par le nom de sa classe dans le contexte Spring.
Il est possible de définir un autre nom, et même son scope:

```
@Named("<beanName>")
@Scope("<scopeName>")
```

IoC et injection de dépendances

définition des beans

Ces annotations sont à placer sur les classes d'implémentation, mais pas sur les interfaces.

Par exemple, le contexte Spring référencera le bean **ManageMarketImpl** sous ce même nom :

```
@Component
@Scope("singleton")  /* optionnel car scope par défaut */
public class ManageMarketImpl implements ManageMarket {
    ...
}
```

Dans l'exemple suivant, le contexte Spring référencera le bean **ManageMarketBusinessImpl** sous le nom **manageMarketBusiness** :

```
@Component("manageMarketBusiness")
public class ManageMarketBusinessImpl implements ManageMarketBusiness {
    ...
}
```

IoC et injection de dépendances

injection de dépendances

Ensuite, il faut faire l'injection de dépendances, grâce aux annotations :

–**@Inject** : injection par **type**. Annotation standardisée Java (JSR-330). Spring va chercher parmi les beans managés, celui qui a un type correspondant à celui attendu dans la classe Java
– **@Autowired** : injection par **type**. Annotation propriétaire et historique de Spring. Son fonctionnement est similaire à **@Inject**
–**@Resource** : injection par **nom**. Annotation standardisée Java (JSR-250). La recherche va être effectuée à partir du nom de l'attribut portant l'annotation, qui doit correspondre à un nom de bean référencé. Si aucun nom ne correspond, une recherche par type est lancée.

Ces annotations peuvent s'appliquer sur un **attribut**, sur un **setter** ou sur un **constructeur**.

> ***Eviter l'utilisation de @Resource*** *car plus assujettie aux erreurs en cas de refactoring ou de renommage.*

IoC et injection de dépendances

injection de dépendances

Il est possible de préciser les beans à utiliser au lieu de garder ceux par défaut (en cas de conflit...).

Si on utilise l'injection par type (**@Inject** et **@Autowired**), il peut y avoir conflit (**NoUniqueBeanDefinitionException**) si 2 beans de même type existent dans le contexte Spring.

Pour éviter cela :

- **@Qualifier("<nomBean>")** : préciser quel bean utiliser en particulier (le nom spécifié peut avoir été défini grâce à **@Named("<nomBean>")** ou être le nom de la classe par défaut)
- ça revient dans ce cas à utiliser **@Resource** puisqu'on précise un nom.

IoC et injection de dépendances

injection de dépendances

> ***Exemple de conflit par type** : un service doit envoyer un message à un utilisateur, soit par mail, soit par SMS (suivant le contexte) :*
>
> *2 classes **EnvoiMail** et **EnvoiSMS** implémentent la même interface **EnvoiMsg**. Les classes appelantes doivent spécifier, suivant leur contexte, quelle implémentation choisir.*

```
@Autowired
@Qualifier("EnvoiMail")
private EnvoiMsg mail;

@Autowired
@Qualifier("EnvoiSMS")
private EnvoiMsg sms;

public void envoyerMessage(boolean byMail, String msg){
  if (byMail) mail.send(msg);
  else sms.send(msg);
}
```

IoC et injection de dépendances

injection de dépendances

Si on utilise l'injection par nom **@Resource**, si le nom de l'attribut ne correspond à aucun nom de bean du contexte Spring, il faut préciser lequel utiliser :

– **@Resource(name="<nomBean>")** : permet d'utiliser un bean dont le nom dans le contexte Spring n'est pas celui de l'attribut

```
// il n'existe pas de bean nommé 'mail' dans le contexte Spring,
// mais il en existe un nommé 'EnvoiMail'
@Resource(name="EnvoiMail")
private EnvoiMsg mail;
```

IoC et injection de dépendances

injection par attribut

L'injection par attribut ne nécessite pas la présence du setter correspondant dans la classe.

Elle peut s'appliquer sur tout type d'attribut, peu importe sa visibilité (**private**, **public**...).

```
@Component
public class ManageMarketImpl implements ManageMarket {

    // un bean de type 'ManageMarketBusiness' doit exister
    @Autowired
    //si besoin, ajouter : @Qualifier("monBeanSpecifique")
    private ManageMarketBusiness manageMarketBusiness;
    ...
}
```

```
@Component
public class ManageMarketImpl implements ManageMarket {

    // un bean de nom 'manageMarketBusiness' doit exister
    @Resource
    //si besoin, préciser le bean : @Resource(name="monBeanSpecifique")
    private ManageMarketBusiness manageMarketBusiness;
    ...
}
```

IoC et injection de dépendances

injection par setter

L'injection par setter se rapproche de l'injection de dépendances dans les versions antérieures de Spring (quand on faisait la configuration en XML).

Il peut représenter un intérêt si on a besoin de faire quelquechose (configuration, initialisation...) en plus d'injecter simplement le bean.

```
@Component
public class ManageMarketImpl implements ManageMarket {

    private ManageMarketBusiness manageMarketBusiness;

    //setter pour injecter une instance de ManageMarketBusiness
    @Autowired
    public void setManageMarketBusiness(ManageMarketBusiness m) {
        this.manageMarketBusiness = m;
        // config, init...
        ...
    }
    ...
}
```

IoC et injection de dépendances

injection par constructeur

```
@Component
public class ManageMarketImpl implements ManageMarket {

    private ManageMarketBusiness manageMarketBusiness;

    // constructeur nécessitant une instance de ManageMarketBusiness
    @Autowired
    public ManageMarketImpl(
            /*si besoin, ajouter : @Qualifier("monBeanSpecifique")*/
            ManageMarketBusiness m) {
        this.manageMarketBusiness = m;

        ...
    }


    ...
}
```

En cas de précision nécessaire, **@Qualifier("manageMarketBusiness")** se placerait juste avant la déclaration du paramètre du constructeur.

IoC et injection de dépendances

injection de données par défaut

Il est possible d'injecter directement des données (avec conversions de types implicites) pour des propriétés, **List**, **Map**, **Set**, grâce à l'annotation **@Value.**

```
public class ManageMarketPartnerImpl implements ManageMarketPartner {

    @Value("http://www.orange.com:8080")
    private String partner;

    @Value("10")
    private Integer pagination;

    @Value("#{'server1,server2'.split(',')}")
    private List<String> serversList;

    ...

}
```

IoC et injection de dépendances

injection de données externalisées

Il est préférable d'externaliser les données de configuration dans un fichier **.properties** et/ou dans des **variables d'environnement**.

Dans le cas d'un **.properties**, il est possible de :

–créer un fichier **.properties** (par défaut **application.properties** dans SpringBoot) accessible dans le classpath (dans le **jar**), qui définit les valeurs par défaut des propriétés
– surcharger ces propriétés grâce à un fichier **.properties** externe au **jar**, dont on aura spécifié son nom au démarrage de l'application.

Ensuite il faut créer une classe de configuration (**@Configuration**) qui permettra d'accéder à ses propriétés, de 2 façons différentes :

– en spécifiant les clés des propriétés à utiliser (**@Value("${nomDeLaCle}")**) pour chacun des attributs de la classe
–en ajoutant une annotation (**@ConfigurationProperties**) qui fera le mapping automatiquement entre le nom des clés et celui des attributs de la classe (qui doivent être identiques).

Plus d'informations sur l'**externalisation de la configuration**.

IoC et injection de dépendances

injection de données externalisées : fichier .properties

Il est possible de donner des valeurs fixes ou aléatoires (string, uuid, int, long, intervalles, valeur max...) pour les propriétés :

```
#appProps.properties

partner.url         = http://www.orange.com:8080
pagination          = 10
servers             = server1,server2
my.random.rstring   = ${random.value}
my.random.ruuid     = ${random.uuid}
my.random.rrange    = ${random.int[10,55]}
my.random.rmax      = ${random.int(15)}
```

IoC et injection de dépendances

injection de données externalisées : @Value

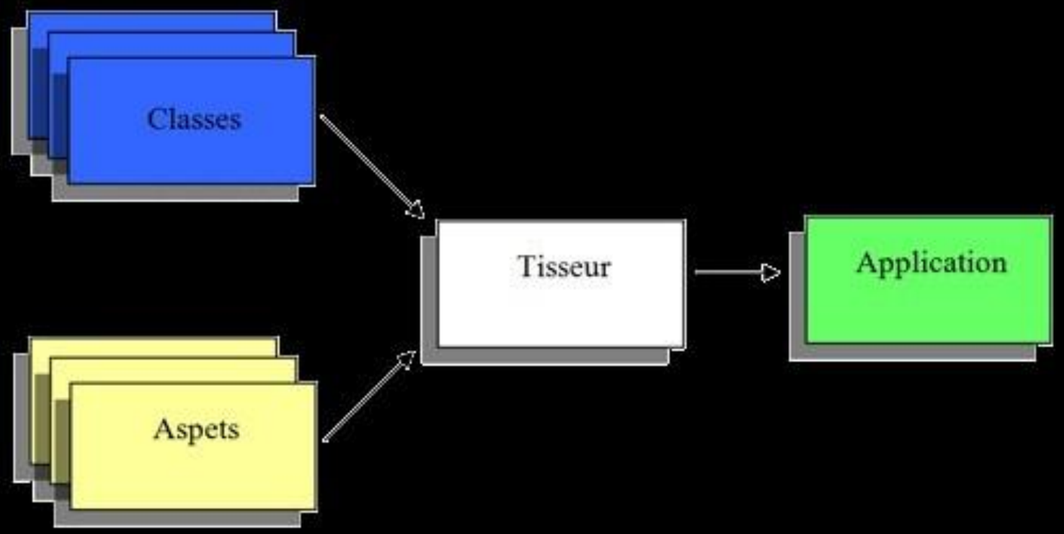
```
@Configuration
@PropertySource("classpath:appProps.properties")
public class AppConfig {

    @Value("${partner.url}")
    private String partner;

    @Value("${pagination}")
    private Integer pagination;

    @Value("#{'${servers}'.split(',')}")
    private List<String> serversList;

    @Value("${my.random.rstring}")
    private String rstring;

    @Value("${my.random.ruuid}")
    private String ruuid;

    ...

    // getters and setters
}
```

IoC et injection de dépendances

injection de données externalisées : @ConfigurationProperties

```
@Configuration
@PropertySource("classpath:appProps.properties")
@ConfigurationProperties(prefix="my.random")
public class AppRandomConfig {

    // my.random.rstring key into .properties file
    private String rstring;

    // my.random.ruuid key into .properties file
    private String ruuid;

    // my.random.rrange key into .properties file
    private String rrange;

    // my.random.rmax key into .properties file
    private String rmax;

    // getters and setters
}
```

AOP : Aspect Oriented Programming



AOP

principes

La **programmation orientée aspects (AOP)** permet d'appliquer des **traitements "transverses"** sur de nombreux objets.

L'intérêt est de **regrouper** dans un ou plusieurs composants ces traitements et de les appliquer là où on en a besoin, plutôt que de morceler ce code un peu partout.

Le principe, c'est d'avoir des **classes métiers + des aspects**, et un framework qui combine (**tisse**) le tout pour créer l'application finale.

AOP

dépendance Maven

Pour faire de l'AOP, ajouter la dépendance vers le starter Spring Boot **spring-boot-starter-aop** :

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

Cette dépendance importe deux frameworks d'AOP populaires, utilisables directement :

- **Spring-AOP** : solution qui répond à la plupart des cas d'utilisation d'AOP, applicable sur des beans Spring. Le tissage se fait au runtime. Un proxy intercepte l'appel à une méthode, exécute l'aspect et le code métier de la méthode.
- **AspectJ** : solution complète (la référence) d'AOP, sur tous types d'objets. Le tissage se fait lors de la compilation (donc plus performant à l'exécution), la classe finale intègre à la fois le code métier et le code de l'aspect.

Cet **article** compare les 2 solutions.

Nous utiliserons les annotations fournies par AspectJ par la suite.

AOP

concepts

Les principaux concepts sont :

–**Aspect** : module contenant du code appliqué de manière transverse (log, transaction..) où sont définis des greffons et des points de greffe
– **Pointcut** *(point de greffe)* : endroit du logiciel où est inséré le greffon (toutes les méthodes ayant tel nom...)
– **Advice** *(greffon)* : code activé avant, après, autour du point de greffe
– **Joinpoint** : point générique durant l'exécution d'un programme où il est autorisé d'insérer un greffon (pas en plein milieu d'une méthode par exemple). Spring AOP ne supporte que l'exécution de méthodes.

Il existe d'autres **concepts** dont nous ne parlerons pas ici.

AOP

déclarer un Aspect

Rappel : l'aspect est un module contenant du code qui sera exécuté de façon transverse.

Un **aspect** est d'abord une classe qui doit être managée par Spring (un bean). Il peut donc contenir des méthodes, des attributs... mais aussi des pointcuts, des advice...

L'annotation **@Aspect** permet de spécifier que c'est un aspect.

```
import org.aspectj.lang.annotation.Aspect;

@Component
@Aspect
public class MyLogAspect {
    ...
}
```

AOP

déclarer un Pointcut

Rappel : le pointcut permet de spécifier où insérer le greffon.

2 parties dans la déclaration :

- **@Pointcut(expression)** : détermine exactement quelles méthodes cibler pour appliquer les advice de l'aspect
- **signature** : nom + paramètres.

```
@Component
@Aspect
public class MyLogAspect {

    @Pointcut("execution(public *    com.sb.todolist..*.get*(..))")
    private void selectGetMethods() {} //signature

    ...
}
```

Plusieurs **mots-clés** sont disponibles pour l'expression du pointcut : **execution, within, this, target, args, @annotation(...)...**

On peut aussi appliquer plusieurs expressions avec des opérateurs logiques : **&&, ||, !**.

AOP

déclarer un/des Advice

Rappel : un advice est du code activé avant, après, autour... du pointcut

Les advice disponibles :

– **@Before** : s'exécute avant l'exécution des méthodes ciblées par le pointcut
– **@AfterReturning** : s'exécute après qu'une méthode se termine normalement
– **@AfterThrowing** : s'exécute après qu'une méthode lève une exception
– **@After** : s'exécute à la fin d'une méthode, en cas d'exception ou non (**finally**)
–**@Around** : permet d'exécuter du code avant et après l'exécution d'une méthode. Dans ce code, on fait un appel explicite à la méthode cible quand on le souhaite (**proceed()**).

> ***Attention** : **@Around** est le plus puissant des advice, car il permet d'appeler le code métier ou pas, de retourner sa propre valeur ou sa propre exception. L'utiliser plutôt dans les cas où on doit partager un état avant et après l'exécution de la méthode (exemple : caclul du temps passé dans un méthode).*
>
> ***Recommandation** : utiliser l'advice le moins puissant permettant d'effectuer le comportement souhaité.*

- *Les expressions Pointcut:*
  - ***execution(…)** : liée à l'exécution des méthodes*
  - **within (…)** : liée aux classes
  - **@annotation (…)** : liée aux annotations

AOP

déclarer un/des Advice

```
@Component
@Aspect
public class MyLogAspect {

    private final Logger logger = LoggerFactory.getLogger(this.getClass());

    @Pointcut("execution(public *   com.sb.todolist..*.get*(..))")
    private void selectGetMethods() {}

    @Before("selectGetMethods()")
    public void addLogEntree(JoinPoint jp) {
        logger.debug("[Executing...] {}({})", jp.getSignature(), Arrays.asList(jp.getArgs()));
    }

    @After("selectGetMethods()")
    public void addLogSortie(JoinPoint jp) { logger.debug(...); }

    // autre façon de faire, mais pas conseillée
    @Around("selectGetMethods()")
    public Object addLogEntreeSortie(ProceedingJoinPoint pjp) throws Throwable {
        logger.debug(...);
        Object retVal = pjp.proceed();
        logger.debug(...);

    }
}
```

AOP

Expressions Pointcut

Appliqué sur toutes les méthodes publiques

```
@Pointcut("execution(public * *(..))")
```

Appliqué sur toutes les méthodes publiques de classe Etudiant

```
@Pointcut("execution(* Student.*(..))")
```

Appliqué sur toutes les méthodes de setter public de la classe Student

```
@Pointcut("execution(* Student.set*(..))")
```

Appliqué sur toutes les méthodes de classe qui renvoie une valeur int

```
@Pointcut("execution(int Student. *(..))")
```

AOP

code impacté (classes API, métiers...)

```
@RestController
public class ToDoList_API {

    @GetMapping()
    public List<ToDoList> getAllToDoLists(...) {...}

    @PostMapping()
    public ToDoList createToDoList(@RequestBody ToDoList todolist){...}
```

```
@Named
public class ToDoList_ServiceImpl implements ToDoList_Service {

    public ToDoList getToDoList(long id) {...}

    public void updateToDoList(ToDoList todolist) {...}
}
```

AOP

code exécuté après tissage

```
@RestController
public class ToDoList_API {

    @GetMapping()
    public List<ToDoList> getAllToDoLists(...) {
        log.debug("[Executing...] {}({})", ...);
        ...
        log.debug("[End] {}({})", ...);
    }

    @PostMapping()
    public ToDoList createToDoList(@RequestBody ToDoList todolist){...}

}
```

exemple d'AOP : les transactions

principes des transactions

Les transactions constituent un point technique clé dans une application.

Une mauvaise gestion des transactions peut provoquer :

- des mises à jour perdues
- des lectures inconsistantes
- des sauvegardes partielles.

Les transactions doivent respecter les propriétés **ACID** :

- **Atomicité** : une transaction doit être entièrement validée ou annulée
- **Cohérence** : une transaction doit laisser les données dans un état cohérent
- **Isolation** : une transaction ne peut pas voir les modifications en cours des autres transactions
- **Durabilité** : les données sont enregistrées de manière permanente.

exemple d'AOP : les transactions

transactions locales

Les transactions locales sont gérées au moment de l'accès aux données, au niveau de la couche DAO.

La cohérence des données n'est pas assurée si la fonctionnalité demande plusieurs accès en base de données pour être accomplie.

De plus, elles nécessitent de la programmation et sont donc sources d'erreurs :

```
try {
    connexion = datasource.getConnection();
    connexion.setAutoCommit(false);
    ps = connexion.prepareStatement(sql);
    ps.setString(1, bean.getName());
    ps.executeUpdate();
    connexion.commit();
} catch (Exception e) {
    connexion.rollback();
    throw new RuntimeException("Erreur");
}
```

exemple d'AOP : les transactions

transactions globales

Les transactions doivent être globales pour encapsuler l'ensemble d'une fonctionnalité, quelque soit le nombre d'accès à la base de données.

Elles doivent donc être gérées dans la **couche applicative**, là où débutent les **use-cases**, pour englober les transactions locales.

Les serveurs d'applications JEE proposent l'utilisation de **Java Transaction API (JTA)**.

Mais là encore, elles nécessitent de la programmation et sont donc sources d'erreurs :

```
public void virementBancaire(AccountBean account1, AccountBean account2, double montant) {

    transaction.begin();
    try {
        accountDao.update(account1, -montant);
        accountDao.update(account2, montant);
        transaction.commit();
    } catch (Exception e) {
        transaction.rollback();
        throw new RuntimeException("Erreur");
    }
}
```

exemple d'AOP : les transactions

les transactions avec Spring

Spring ajoute un niveau d'abstraction pour la gestion des transactions.

Il suffit d'ajouter l'annotation **@Transactional** aux méthodes applicatives initiant des transactions :

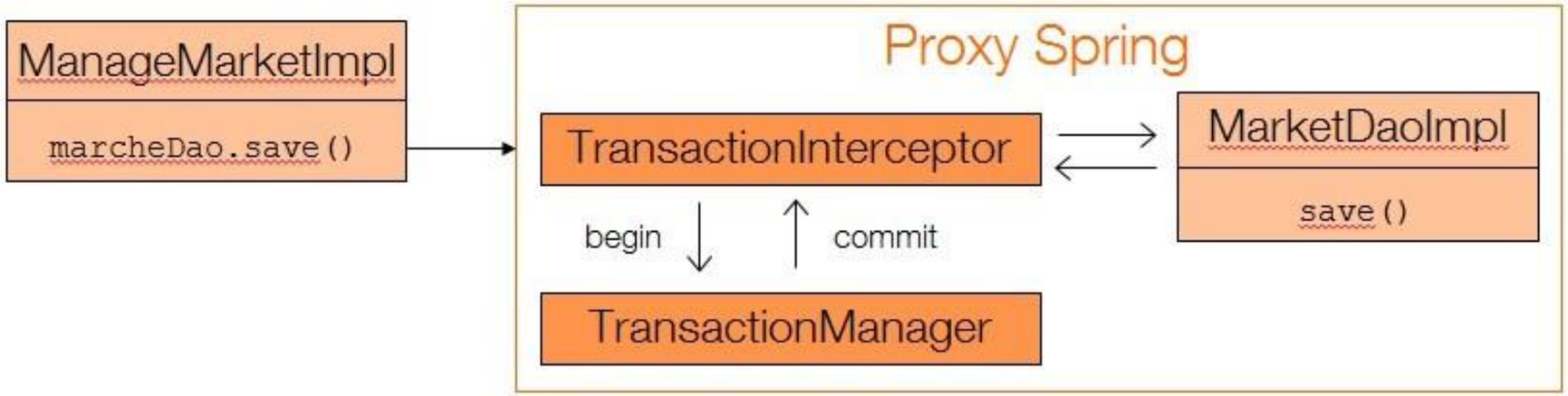
```
@Transactional
public void virementBancaire(AccountBean account1, AccountBean account2, double

    montant) { accountDao.update(account1, -montant);
    accountDao.update(account2, montant);

}
```

exemple d'AOP : les transactions

les transactions avec Spring

Exemple : ***sans @Transactional***

Exemple : ***avec @Transactional***



Annexes (pour votre lecture)
Les tests unitaires

les tests unitaires

principes

Les **tests unitaires (TU)** doivent être des tests **atomiques (ne tester qu'une chose à la fois), simples, rapides et les plus légers possibles**. Ils sont destinés a être exécutés à chaque build du projet pour détecter les erreurs au plus tôt.

En général, le test se porte sur une **méthode** particulière qu'il faut **isoler** des autres méthodes dont elle dépend, grâce à des bouchons (**mocks**) dont on maîtrise le comportement.

Sans cette isolation, le test devient un test d'intégration :

– on teste la méthode en question mais aussi tout ce qui est sous-jacent... jusqu'à la base de données ou l'appel d'un partenaire externe
– en cas d'erreur, on ne sait pas à quelle niveau elle intervient (méthode testée ou ses dépendances ?)
– ces tests sont longs à exécuter et ralentissent le processus de build.

les tests unitaires

JUnit, Mockito et Hamcrest

Le pattern d'IoC apporte **plus de modularité** à l'application et **moins de dépendances** entre les objets. Cette isolation facilite grandement l'écriture des tests unitaires et en assure l'atomicité.

Spring s'intègre parfaitement avec les frameworks :

- **JUnit 5** :
  - apporte des éléments simplifiant l'écriture de tests unitaires
  - annotations : **@BeforeAll**, **@BeforeEach**, **@Test**, **@AfterEach**, **@AfterAll**, **@DisplayName**...
  - assertions : **assertEquals**, **assertNotNull**, **assertTrue**, **assertThat**...
- **Mockito** (et **BDDMockito**) :
  - apporte des éléments simplifiant la création de mocks (bouchons)
  - annotations : **@Mock**, **@InjectMocks**...
  - directives de comportement Mockito : **when()**, **thenReturn()**, **thenThrow()**...
  - directives de comportement BDDMockito : **given()**, **willReturn()**, **willThrow()**...
- **Hamcrest** :
  - apporte des éléments simplifiant la comparaison d'objets entre eux, de vérifier telles propriétés...
  - méthodes : **equalTo()**, **contains()**, **hasSize()**...

Voici un **guide de migration de JUnit 4 à Junit 5 (Jupiter)**.

les tests unitaires

dépendance Maven pour JUnit 5

Les dépendances nécessaires aux tests sont toutes importées par Spring Boot dans le **spring-boot-starter-test** :

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

L'exclusion de **junit-vintage-engine** permet de ne pas embarquer JUnit 3 et JUnit 4.

Pour en savoir plus sur les starters Spring Boot, rendez-vous au chapitre : **Le chef d'orchestre Spring Boot**.

les tests unitaires

JUnit, Mockito et Hamcrest

```
@SpringBootTest
public class ToDoListServiceTest {

    @MockBean       // crée et injecte un mock du repo dont dépend la classe testée
    private ToDoListRepository mockTdlRepo;

    @Inject         // injecte la classe testée
    private ToDoList_ServiceImpl tdlService;

    @BeforeAll      // s'exécute 1 fois au lancement des tests /!\ static
    public static void init() {...}

    List<ToDoList> allTdls;

    @BeforeEach     // s'exécute avant chaque test
    public void setup() {
        // création d'une liste en dur de ToDoLists
        allTdls = new ArrayList<ToDoList>();
        allTdls.add(new ToDoList(...);
        allTdls.add(new ToDoList(...);
        allTdls.add(new ToDoList(...);
    }
```

les tests unitaires

JUnit, Mockito et Hamcrest

```
@DisplayName("returns all todolists")   // nom à afficher du test à exécuter
@Test                                   // test à exécuter
public void shouldReturn_All_ToDoLists() {
    //given (définition du comportement du mock pour la méthode findAll())
    //   ici : retourne la liste des todolists créé en dur dans @BeforeEach
    given(mockTdlRepo.findAll()).willReturn(allTdls);
    //when (appel de la méthode testée)
    List<ToDoList> result = tdlService.getAllToDoLists("");
    //then (vérification du résultat de la méthode testée)
    assertEquals(3, result.size());
    assertTrue(allTdls.containsAll(result));
}
```

Explications :

– **when** : la méthode testée **tdlService.getAllToDoLists("")** fait appel à la méthode **findAll()** de **ToDoListRepository**
– **given** : on définit d'abord (via le mock) le comportement de cette dépendance pour maîtriser ce qui est retourné à la méthode testée
– **when** : on vérifie l'état à la sortie de la méthode testée

les tests unitaires

JUnit, Mockito et Hamcrest

```
    @Test
    public void shouldReturn_EmptyResultDataAccessException() {
        //given (définition du comportement du mock pour la méthode findById())
        //   ici : retourne null quelquesoit le nombre (de type long) passé en paramètre
        given(mockTdlRepo.findById(ArgumentMatchers.anyLong())).willReturn(null);
        //then (vérification du résultat de la méthode testée : lancement d'une exception)
        Assertions.assertThrows(EmptyResultDataAccessException.class, () -> {
            //when (appel de la méthode testée)
            tdlService.getToDoList(1);
        });
    }

    @AfterEach      // s'exécute après chaque test
    public void teardown() { ... }

    @AfterAll       // s'exécute une fois tous les tests terminés /!\ static
    public static void destroy() {...}
}
```

rappel des bonnes pratiques

– structurer votre application en couches logicielles ou composants
– respecter le duo interface/implémentation, l'implémentation étant suffixée par **Impl**
– ne pas créer de beans Spring systématiquement :
    - oui pour les classes de traitement
    - non pour les classes du domaine/modèle d'échange
– utiliser les bons scopes pour les bons usages (en général, seul le scope singleton est utilisé)
– utiliser les annotations standardisées si possible (**@Inject** et **@Named**)
– utiliser l'héritage de beans pour mutualiser leur configuration si nécessaire
– externaliser les paramètres dans un fichier **properties**
– gérer les transactions au niveau de la couche applicative/service.

documentation

– Site officiel de Spring
– Documentation de Spring Framework
– Externalisation de la configuration
– Guide d'implémentation d'AOP avec Spring Boot et AspectJ
– Site Baeldung : de nombreux tutoriaux
– Article qui compare Spring-AOP et AspectJ
– Site officiel de Junit 5
– Guide de migration de Junit 4 à Junit 5
– Site officiel de Mockito
– Site officiel de Hamcrest

Annexes (pour votre lecture)

configuration XML

configuration XML

Avec Spring, il existe plusieurs façons de déclarer les beans et d'effectuer l'injection de dépendances entre eux :

– **en annotations Java** : à privilégier
– **en XML**
– **en XML et en annotations Java**

> ***Attention :*** *la configuration par annotations sera chargée avant celle en XML. Si des configurations sont communes entre XML et annotations, le XML écrasera les annotations.*

configuration XML

chargement du contexte Spring

Le contexte Spring (contenant les beans managés) peut être chargé depuis n'importe quel environnement (application web, test unitaire JUnit, EJB, application standalone...).

Le(s) fichier(s) XML de configuration peu(ven)t être recherché(s) programmatiquement :

– dans le classpath :

```
ApplicationContext ctx =
new ClassPathXmlApplicationContext("example/app_ctx.xml");
```

– dans le système de fichiers :

```
ctx = new FileSystemXmlApplicationContext("c:/proj/app_ctx.xml");
```

– à une adresse relative à l'environnement :

```
ctx = new XmlWebApplicationContext("WEB-INF/app_ctx.xml");
```

configuration XML

chargement du contexte Spring

Dans le cadre d'une application web, on préfère déclarer le servlet listener de Spring dans le fichier de déploiement **web.xml**, en précisant où se situe le fichier de configuration principal de Spring :

```
<!-- Spring params -->
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/spring-config/applicationContext.xml
    </param-value>
</context-param>
<!-- Spring start context Listener -->
<listener>
    <display-name>spring context loader</display-name>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

Le contexte Spring est initialisé et chargé dans le ServletContext avant toute autre servlet.

configuration XML

définition des beans

Dans le fichier de configuration Spring (exemple : **applicationContext.xml**), il faut d'abord définir les beans :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- Bean du composant applicatif -->
<bean id="manageMarketApplication"
    class="  com.sb.projet.application.impl.ManageMarketImpl"
    scope="singleton" />

<!-- Bean du composant métier -->
<bean id="manageMarketBusiness"
    class="  com.sb.projet.business.impl.ManageMarketBusinessImpl">
</bean>


</beans>
```

configuration XML

injection par setter

Il faut ensuite injecter les dépendances d'un bean à l'autre.

Pour l'injection par **setter**, il faut utiliser la balise **property**, faisant référence à un autre bean.

```
<!-- Bean du composant applicatif -->
<bean id="manageMarketApplication"  class="
    com.sb.projet.application.impl.ManageMarketImpl"
    scope="singleton" />
    <!-- nom d'un attribut (avec setter) de ManageMarketImpl.java -->
    <property name="manageMarketBusiness">
        <!-- référence à l'id d'un autre bean -->
        <ref bean="manageMarketBusiness" />
    </property>
</bean>

<!-- Bean du composant métier -->  class="
<bean id="manageMarketBusiness"
  com.sb.projet.business.impl.ManageMarketBusinessImpl">
</bean>
```

configuration XML

injection par setter

Dans le code Java, il faut donner la possibilité à Spring d'injecter un bean grâce à un setter.

```
public class ManageMarketImpl implements ManageMarket {

    //nom d'attribut correspondant au name de la balise property
    private ManageMarketBusiness manageMarketBusiness;

    //setter pour injecter une instance de ManageMarketBusiness
    public void setManageMarketBusiness(
            ManageMarketBusiness manageMarketBusiness) {
        this.manageMarketBusiness = manageMarketBusiness;
    }


    ...
}
```

configuration XML

injection par constructeur

Pour l'injection par **constructeur**, il faut utiliser la balise **constructor-arg**, faisant référence à un autre bean.

```
<!-- Bean du composant applicatif -->
<bean id="manageMarketApplication"  class="
    com.sb.projet.application.impl.ManageMarketImpl">
    <!-- référence à l'id d'un autre bean et -->
    <!-- nom d'argument du constructeur de ManageMarketImpl.java -->
    <constructor-arg ref="manageMarketBusiness" />
</bean>

<!-- Bean du composant métier -->  class="
<bean id="manageMarketBusiness"
  com.sb.projet.business.impl.ManageMarketBusinessImpl">
</bean>
```

configuration XML

injection par constructeur

Dans le code Java, on retrouve l'attribut défini en **constructor-arg** ainsi qu'une référence en paramètre du constructeur, qui servira à injecter l'autre bean.

```
public class ManageMarketImpl implements ManageMarket {

    private ManageMarketBusiness manageMarketBusiness;

    // constructeur nécessitant une instance de ManageMarketBusiness
    public ManageMarketImpl(
          ManageMarketBusiness manageMarketBusiness) {
        this.manageMarketBusiness = manageMarketBusiness;
    }


    ...
}
```

configuration XML

injection de données par défaut

De même que pour la configuration par annotation, il est possible d'injecter directement des valeurs par défaut (avec conversions de types implicites) pour des propriétés, List, Map, Set.

C'est utile pour du paramétrage par exemple, ou pour les tests.

```
<bean id="manageMarketApplication"  class="
   com.sb.projet.application.impl.ManageMarketImpl">
   <property name="partner" value="http://www.orange.com:8080" />
   <property name="pagination" value="10" />
   <property name="serversList" >
       <list>
          <value>server1</value>
          <value>server2</value>
       </list>
    </property>
</bean>
```

Attention à ne pas oublier les attributs et leur setter dans la classe !

configuration XML

injection de données externalisées

Les données de configuration peuvent être externalisées dans un fichier **properties** :

– créer un fichier **properties** accessible dans le classpath
– déclarer un **PropertyPlaceholder** qui charge ce fichier **properties**
– récupérer les valeurs du properties grâce à **${<clé>}**.

```
#appProps.properties
partner.url = http://www.orange.com:8080/
pagination  = 10
servers     = server1,server2
```

```
<context:property-placeholder
   location="classpath:appProps.properties" />

<bean id="manageMarketApplication"  class="
   com.sb.projet.application.impl.ManageMarketImpl">
   <property name="partner" value="${partner.url}" />
   <property name="pagination" value="${pagination}" />
   <property name="serversList" value="#{'${servers}'.split(',')}"/>
</bean>
```

configuration XML

gestion de l'héritage de beans

Plusieurs beans peuvent parfois partager certaines informations, avoir la même configuration...

L'héritage de beans permet la mutualisation ou la surcharge de ces propriétés communes.

```
<bean id="abstractVehicule" class="exemple.AbstractVehicule"
    abstract="true">
   <property name="companyName" value="BMW" />
   <property name="modelName" />
   <property name="color" />
</bean>

<bean id="voiture" class="exemple.Voiture"
    parent="abstractVehicule">
   <property name="nbPortes"/>
</bean>

<bean id="moto" class="exemple.Moto"
    parent="abstractVehicule">
   <property name="topcase"/>
</bean>
```

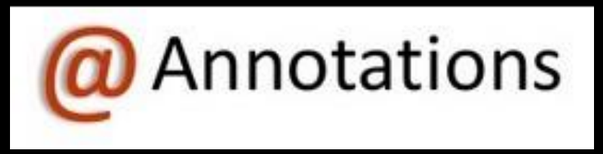
configuration XML

gestion de l'héritage de beans

```
public abstract class AbstractVehicule {

    protected String companyName = "BMW";
    protected String modelName;
    protected String color;
    ...
}

public class Voiture extends AbstractVehicule {

    private Integer nbPortes;
    ...
}

public class Moto extends AbstractVehicule {

    private boolean topcase;
    ...
}
```

annexes

configuration XML et annotations



configuration XML et annotations

utilisation des annotations uniquement (hors Spring Boot)

La déclaration des beans et l'injection de dépendances peuvent se faire en utilisant uniquement des annotations Java.

Mais **hors Spring Boot**, l'usage d'annotations n'est pas activée par défaut.

Dans le fichier de configuration Spring, **<context:component-scan ...>** permet :

– d'activer les annotations
– de scanner le package déclaré et ses sous-packages pour charger les beans présents.

Le reste de la configuration (déclaration des beans et injection de dépendances) peut maintenant être faite via annotations.

configuration XML et annotations

utilisation des annotations uniquement (hors Spring Boot)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:component-scan base-package="  com.sb.projet" />

</beans>
```

Spring va scanner toutes les classes du package **com.sb.projet** et de ses sous-packages, pour y trouver des beans annotés **@Named, @Component, @Repository**....

configuration XML et annotations

mixer XML et annotations

Il faut activer les annotations par **<context:annotation-config />** et déclarer les beans en XML (sans **@Named** ou ses dérivés au niveau Java) :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:context="http://www.springframework.org/schema/context"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
   http://www.springframework.org/schema/context
   http://www.springframework.org/schema/context/spring-context-3.0.xsd">

   <context:annotation-config />

<bean id="manageMarketApplication"  class="
   com.sb.projet.application.impl.ManageMarketImpl" />

<bean id="manageMarketBusiness"  class="
   com.sb.projet.business.impl.ManageMarketBusinessImpl"/>

</beans>
```

configuration XML et annotations

injection de dépendances

Comme pour la configuration Java, l'injection de dépendances se fait grâce aux annotations **@Inject**, **@Autowired** et **@Resource** (sans passer par le XML via **property** ou **constructor-arg**).

Leurs caractéristiques ne changent pas :

– elles peuvent s'appliquer sur un **attribut**, sur son **setter** ou sur un **constructeur**

–**@Qualifier("nomBean")** permet de lever les ambiguités lors de la recherche par type. Dans ce cas là, c'est l'**id** du bean défini dans le XML qui est utilisé.

Attention : en cas de configuration commune entre XML et annotations, c'est le XML qui est pris en compte.

configuration XML et annotations

injection de données par défaut ou externalisées

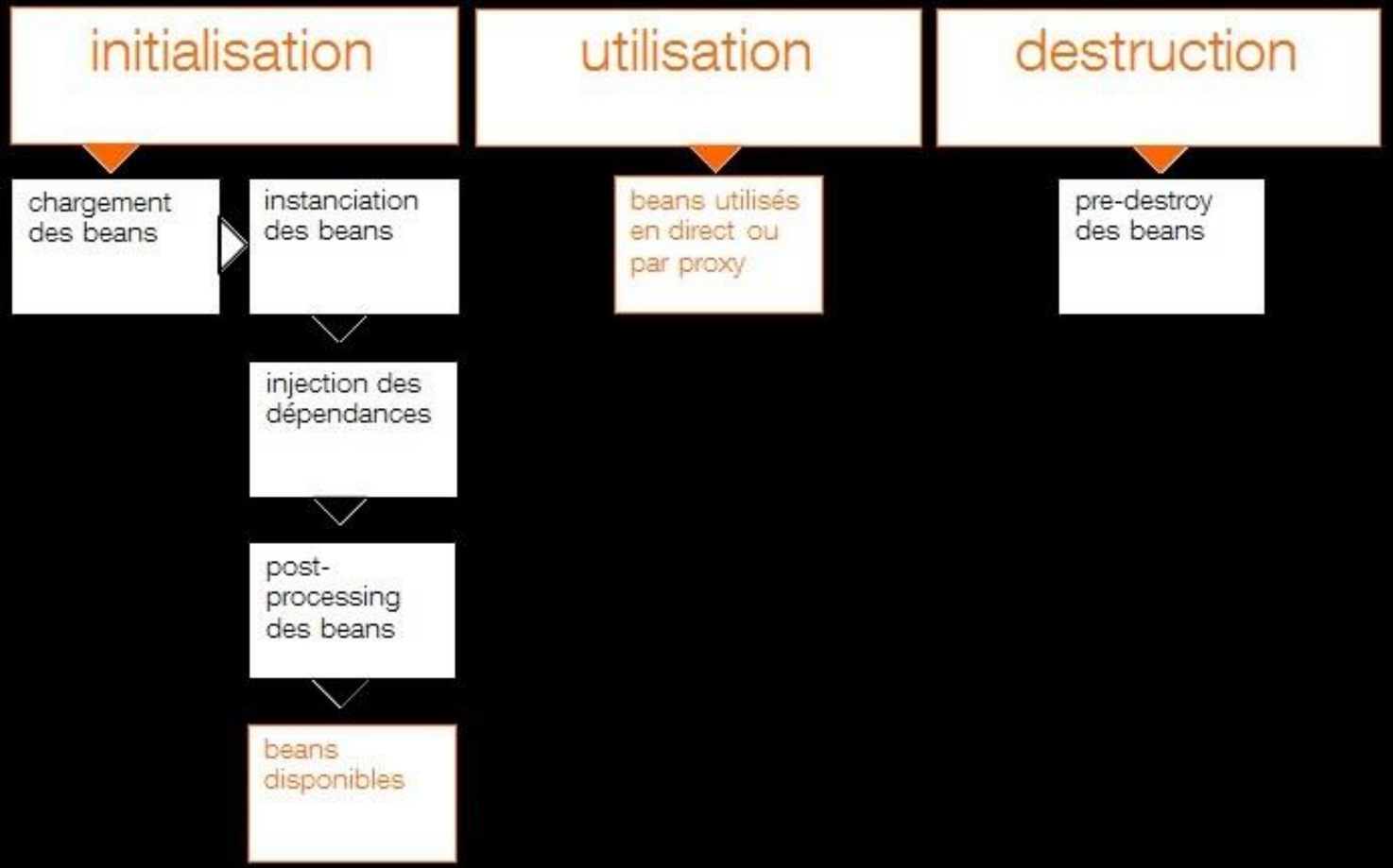
Pour injecter des données par défaut, utiliser soit la méthode purement Java, soit celle purement XML.

Pour injecter les données externalisées, déclarer le fichier **properties** en XML, puis récupérer les valeurs en annotations **@Value**.

```
<context:property-placeholder
   location="classpath:appProps.properties" />
```

```
public class ManageMarketImpl implements ManageMarket {

   @Value("${partner.url}")
   private String partner;

   @Value("${pagination}")
   private Integer pagination;

   @Value("#{'${servers}'.split(',')}")
   private List<String> serversList;

   ...
}
```

cycle de vie du contexte Spring



cycle de vie du contexte Spring

phase d'initialisation des beans

La définition de chaque bean est chargée grâce aux annotations Java (ou aux configurations XML), les beans sont alors instanciés.

Les **BeanFactoryPostProcessor** sont invoqués :

– ils permettent de modifier la définition des beans avant l'injection de dépendances

–le plus utilisé est le **PropertyPlaceholderConfigurer** : il remplace les occurrences de **${<clé>}** par leurs valeurs définies dans un fichier **properties** défini via l'annotation **@PropertySource("classpath:<fileName1>, <fileName2>...")**.

Une fois les beans instanciés, l'**injection de dépendances** peut avoir lieu.

cycle de vie du contexte Spring

phase d'initialisation des beans

Une **phase de post-processing** des beans permet d'apporter des modifications aux beans si besoin, pour définir des traitements entre la création et l'utilisation d'un bean.

Il suffit d'ajouter l'annotation **@Postconstruct** à la méthode à activer.

```
public class ManageMarketImpl {

    ...

    @PostConstruct
    public void init() throws Exception {
        ...
    }

}
```

cycle de vie du contexte Spring

phase d'initialisation des beans

Il est possible également d'invoquer un **BeanPostProcessor** :

– option très puissante qui peut modifier entièrement le comportement d'un bean
– il est difficile d'écrire son propre **BeanPostProcessor**
– Spring en fournit plusieurs implémentations.

Un **BeanPostProcessor** peut, par exemple, ajouter un proxy devant un bean, modifiant ainsi de manière transparente pour le développeur le comportement de l'application.

Cette faculté est utilisée dans les mécanismes d'AOP.

cycle de vie du contexte Spring

utilisation des beans managés

Une fois le contexte Spring chargé, les beans Spring sont disponibles et utilisables :

```
public class ManageMarketImpl implements ManageMarket {

    @Inject
    private ManageMarketBusiness manageMarketBusiness;


    ...

    public Map<Integer, MarketBean> getAllMarkets() {
        return manageMarketBusiness.findAllMarkets();
    }

    ...
}
```

cycle de vie du contexte Spring

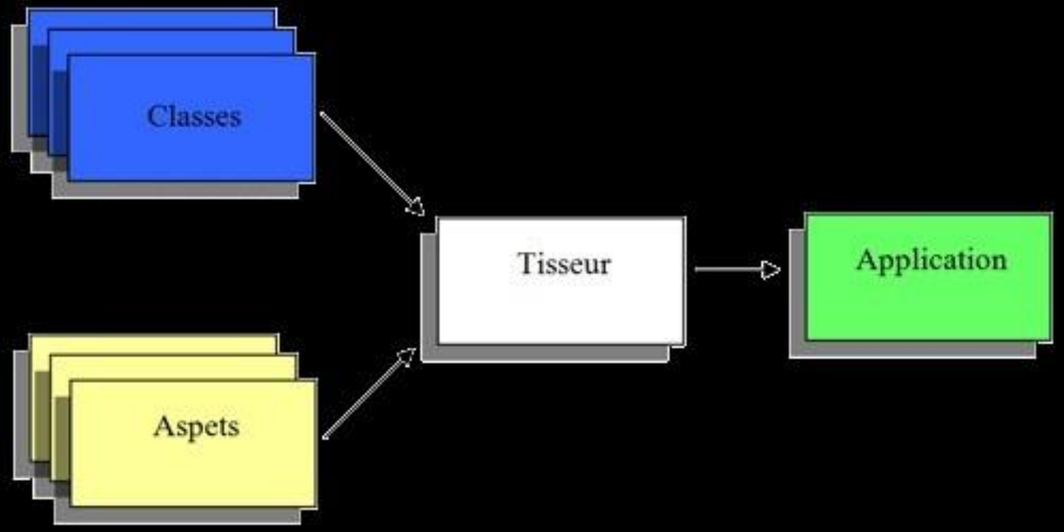
phase de destruction des beans

Cette phase permet d'ajouter des traitements à effectuer avant la destruction d'un bean.

Il suffit d'ajouter l'annotation **@PreDestroy** à la méthode à activer :

```
public class ManageMarketImpl {

    ...

    @PreDestroy
    public void clean() throws Exception {
        ...
    }

}
```

annexes
AOP en XML



AOP en XML

Il est possible de déclarer toutes ces notions en XML plutôt qu'en annotations, dans l'élément **<aop:config>**.

```
<aop:config>

    <aop:aspect id="myLogAspect" ref="aBean">
        <aop:pointcut id="businessService"
            expression="execution(* void create*(..))
                && within(  com.sb.appli..*)"/>

        <aop:before pointcut-ref="businessService" method="monitor"/>
        ...
    </aop:aspect>

</aop:config>

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