Spring, mise en œuvre avancée

Table des matières

Introduction

Les fondements d’une application

Un modèle du domaine
: qui représente les entités métiers.

Des services
: qui implémentent les cas d’utilisation.

Un espace de stockage
: où sont stockées les données.

Une interface client
: qui permet de réaliser les cas d’utilisation.

Exemple, pour une bibliothèque

- Modèle du domaine : Abonné, Livre, Emprunt.
- Services : rechercher un livre, emprunter un livre.
- Stockage : tables Abonnés, Livres, Emprunts.
- Interface client : écrans pour s’authentifier, rechercher un livre, emprunter un livre.

Une approche naïve consisterait à écrire des pages web dans lesquelles se trouveraient :

- le code graphique (css, html)
- le code métier.
- le code d’accès aux données (requête SQL par exemple).

Avantage : le code est centralisé à un seul endroit.

Outre que cela demande des développeurs polyvalents, cela pose 2 problèmes :

- les aspects de l’application ne peuvent pas évoluer indépendamment les uns des autres.
- aucune réutilisation possible.

Le découpage en couches

Découper l'application en *couches* (*layers*) permet de **séparer les responsabilités** :

Dans une application Java, une couche peut être vue comme un *package* qui contient des objets traitant un même aspect de l'application.

Exemple :

Soit une application *app* d'une organsisation *acme* :

- dans *com.acme.app.model* se trouvent les entités
- dans *com.acme.app.dao* se trouve le composants responsables de l'accès aux données
- dans *com.acme.app.service* et *com.acme.app.business* les composants responsables des traitements métiers
  - *com.acme.app.service* : le(s) point(s) d'entrée permettant de déclencher la réalisation des cas d'utilisation
  - *com.acme.app.business* : les opérations fonctionnelles relatives à chaque entité

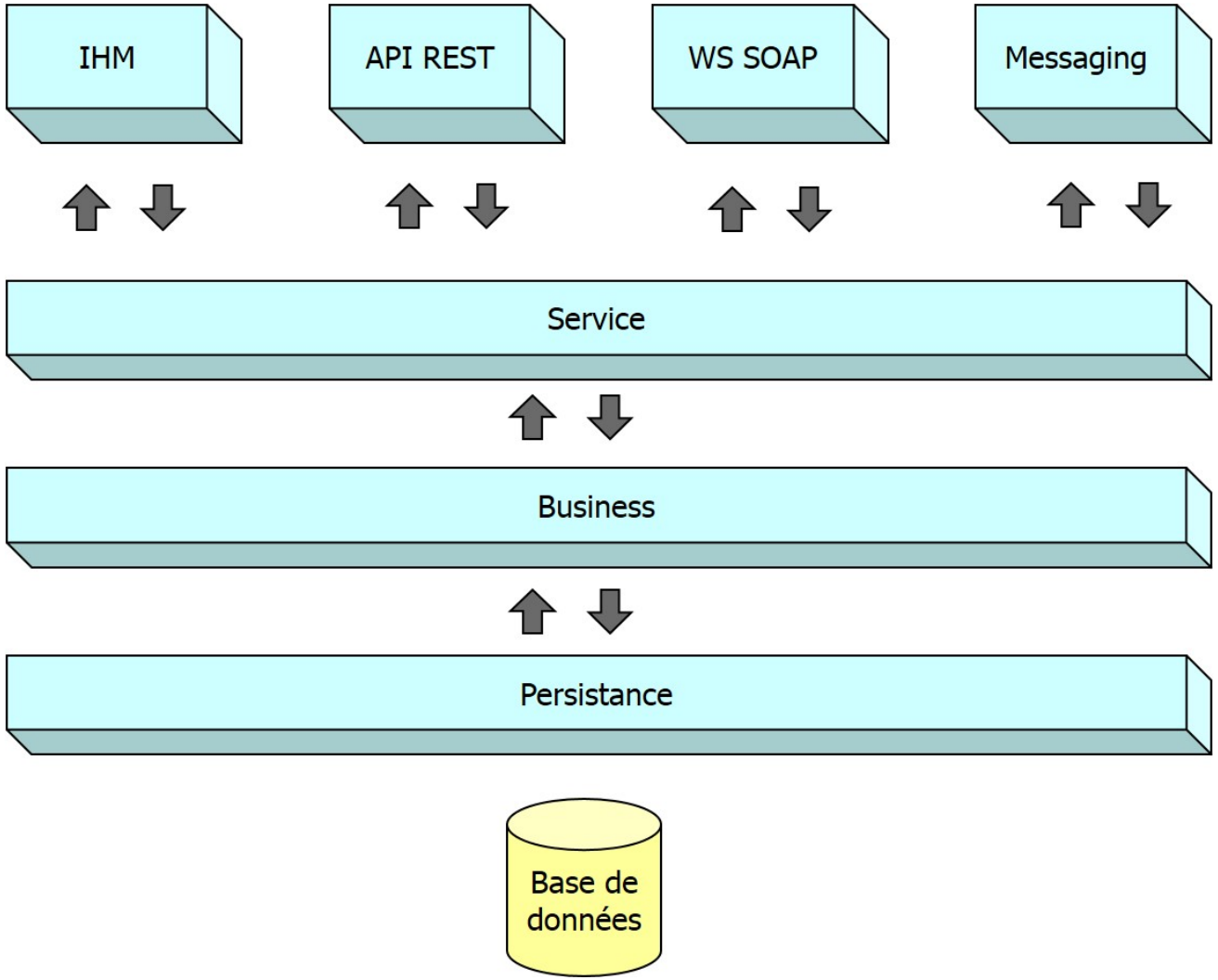
A ce stade l'application prend la forme d'un *jar*, elle est comme la partie immergée d'un iceberg.

Il faut lui ajouter une couche *web* pour la rendre accessible aux utilisateurs.

Cette couche web peut prendre deux formes :

- des web services (SOAP ou REST)
- une IHM, qui met en relation des des contrôleurs et des vues :
  - les vues sont éléments graphiques présentés à l'utilisateur
  - les contrôleurs contiennent le code Java à exécuter suite aux actions utilisateur (cliquer sur un bouton par exemple).

Schema d'une architecture découpée en couche :



Les entités sont quant à elles transverses à toutes les couches, elles *circulent* de la base de données aux vues.

Dans le cas de web services nous pouvons préférer exposer des DTO (*data transfer objects*) plutôt que les entités métiers.

IHM, WS SOAP, API REST, Messaging

ce sont les points d'entrées au système, ils assument un lien avec une technologie particulière.

Les services

ils réalisent les cas d'utilisation (ex : « emprunter un livre » pour une bibliothèque). Pour se faire chaque méthode de chaque service fait appel à la couche business. Généralement une transaction entoure les appels à la couche *service* afin que le traitement puisse conduire à passer d'un état stable du système à un autre état stable (ou à revenir à l'état initial en cas de problème).

La couche business

propose les opérations fonctionnelles associées à chaque entité. Elle est appelée par la couche *service* puisque chaque cas d'utilisation est une suite d'opération fonctionnelles.

Les couches *service* et *business* ne contiennent aucun code technique

La couche de persistance

est composée de *dao* (*data access oject*). Ces objets sont responsables des opérations de lecture et d'écriture des entités métiers. Leur développement est guidé par les besoins de la couche business. Ces classes assument un lien avec une technologie particulière (SGBD, web services, fichiers…). Si la source de données est une base de données cette responsabilité peut être déléguée à un framework (Hibernate pour les bases de données relationnelles par exemple).

Les entités métier

sont transverses à toutes les couches (sauf si l'on doit exposer une API, auquel cas nous pouvons utiliser des DTO)

Exemple :

```
public class LdapUserDao { // couche dao
    public User getUserByLogin(String login) {
        User u = // requête dans un annuaire LDAP
        return u;
    }
}

public class UserService {// couche service
    public boolean userExists(String login, String password){
        User u = new LdapUserDao().getUserByLogin(login);
        if(u==null){
            return false;
        }
        return u.getPassword().equals(password) && u.getExpDate().isAfter(Instant.now());
    }
}
```

La séparation des responsabilité est ici bien réalisée.

Toutefois nous avons un lien fort entre les deux composants de l'application : la méthode `userExists` entretient un lien fort avec `LdapUserDao`. Conséquences :

- Elle doit savoir l'instancier
- Elle contrôle son cycle de vie
- Il n'est pas possible de tester `UserService` indépendamment de `LdapUserDao`

Assurer un couplage faible entre les différentes couches par le biais des interfaces et de l'injection de dépendances : programmation par interface + injection de dépendances (DI) = couplage faible.

L'injection de dépendances permet de réaliser l'**inversion du contrôle** (IOC) : si un objet a besoin d'un autre objet, il en reçoit une instance plutôt que d'en créer une.

Nous gagnons en abstraction car le type de l'instance reçue est celui d'une interface.

L'application de cette bonne pratique conduit à modifier le code ainsi :

```
public interface UserDao{
    User getUserByLogin(String login);
} // LdapUserDaoImpl est une implémentation de UserDao

public class UserServiceImpl implements UserService {

    private final UserDao dao; // final => l'objet est immutable

    public UserServiceImpl(UserDao dao /* injection de dépendances */) {
        this.dao = dao; // obligatoire du fait de la nature final de 'dao'
    }

    @Override
    public boolean userExists(String login, String password){
        User u = this.dao.getUserByLogin(login);
        if(u==null){
            return false;
        }
        return u.getPassword().equals(password) && u.getExpDate().isAfter(Instant.now());
    }
}
```

Le couplage faible pose une question : qui est désormais responsable de la création des composants?

Réponse : la **factory**.

Dans une application, les composants sont comme des ingrédients et la *factory* est responsable d'appliquer la recette qui permet de disposer d'une application opérationnelle :

- gérer le **cycle de vie des objets** :
  - ceux qui ne maintiennent pas d'état sont des *singletons* : tous les composants qui ont besoin de l'objet peuvent utiliser la même instance.
  - ceux qui maintiennent un état sont des *prototypes* : tous les composants utilisateurs de l'objet doivent disposer d'une instance dédiée.
- procéder à l'**injection de dépendances** : créer un objet suppose de lui fournir ce dont il a besoin pour être opérationnel (cf. injection par constructeur).

La factory est donc responsable de la bonne mise en place de l'application. Cette mise en place a tout intérêt à se réaliser *une bonne fois pour toute* au démarrage de l'application.

Exemple :

```
public class ApplicationFactory{

    private static Map<String, Object> objects = new HashMap<>();

    static {
        UserDao userDao = new LdapUserDaoImpl();
        UserService userService = new UserServiceImpl(userDao);
        objects.put("userDao", userDao);
        objects.put("userService", userService);
    }

    public static Object getBean(String name){
        Object obj = objects.get(name);
        if(obj==null) {
            String errMessage = "could not find any object named "+name
            throw new IllegalArgumentException(errMessage);
        }
        return obj;
    }
}
```

La méthode *getBean* permet aux objets qui ne sont pas gérables par la *factory* (exemple : une *servlet*, un test unitaire) de récupérer auprès d'elle un objet *prêt* à l'emploi.

Exemple :

```
UserService userService = (UserService)ApplicationFactory.getBean("userService");
```

La *factory* est idéalement placée pour créer des *proxies* devant les objets, ainsi les invocations de méthodes d'un objet sur un autre objet passe par un *proxy* qui pourra enrichir le traitement métier effectué par la méthode appelée :

```
public class ApplicationFactory{

    private static Map<String, Object> objects = new HashMap<>();

    public static <T> T createObject(T impl){
        ClassLoader cl = impl.getClass().getClassLoader();
        Class<?>[] interfaces = impl.getClass().getInterfaces();
        InvocationHandler handler = new InvocationHandler(){
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                // traitement avant l'invocation de la méthode
                Object ret = method.invoke(impl, args);
                // traitement après l'invocation de la méthode
                return ret;
            }
        }
        return (T)Proxy.newProxyInstance(cl, interfaces, handler);
    }

    static {
        UserDao userDao = createObject(new LdapUserDaoImpl());
        UserService userService = createObject(new UserServiceImpl(userDao));
        objects.put("userDao", userDao);
        objects.put("userService", userService);
    }
    // méthode getBean (cf. slide précédente).
}
```

Les designs patterns

Nous venons ici d'utiliser 6 *designs patterns* :

- **Facade** pour la séparation des responsabilités.
- **Dependency injection** (d'après le pattern original *strategy*) pour le couplage faible entre les composants.
- **Factory** pour la création des composants. Cela introduit de facto les patterns associé au cycle de vie des objets :
  - **singleton** si l'objet ne maintient pas d'état.
  - **prototype** si l'objet maintient un état.
- **proxy** pour contrôler l'invocation des méthodes.

Ils sont la conséquence du découpage en couche : une application qui n'aurait qu'une seule classe remplie de méthodes capables de traiter les aspects métiers et techniques relatifs à un cas d'utilisation n'aurait pas besoin de tous ces *patterns*.

Les *designs patterns* sont des recettes standard de conception d'applications, qui s'appuient sur les principes de la programmation objet.

Ils permettent de gagner en qualité de conception. La qualité peut se mesurer (entre autres) à 4 principes :

SOC : *separation of concern*

On parle aussi de *single responsability principle*. Chaque méthode doit avoir une responsabilité claire et délimitée, une classe ne contient que des méthodes qui adressent des responsabilité de même nature (exemple : `UserDao` spécifie les les opérations de lecture, écriture et suppression des *User* dans le référentiel utilisateur).

KISS : *keep it simple, stupid*

L'objectif est de garder des choses simples : que l'on est capable de comprendre lorsque l'on revient sur le code que nous avons écrit, et que les autres peuvent comprendre aussi. Une chose est simple si sa responsabilité est clairement définie.

DRY : *don't repeat yourself*

En plus d'être source d'erreur, la répétiion de code freine l'évolutivité et la maintenabilité. Le *refactoring* sert à éliminer la duplication de code.

POJO : *plain old java object*

Nos objets Java ne doivent être guidés que par notre savoir faire en programmation orientée objet et par les spécifications fonctionnelles. Si nous utilisons une technologie tierce (un *framework* par exemple), nous ne voulons pas que celle-ci contraignent notre code applicatif : c'est au *framework* de s'adapterà nous et pas l'inverse.

Les tests

Tester conduit à comparer un résultat obtenu avec un résultat attendu. Si les deux correspondent le test est considéré comme réussi.

Tester permet d'échanger

des coûts cachés

(quel est le coût d'un dysfonctionnement en production ?)

contre

des coûts visibles

(combien de jours/hommes sont nécessaires à l'écriture des tests).

Il existe plusieurs manières de tester une application :

Ecrire des méthodes `main`, les exécuter depuis l'IDE et observer les informations affichées dans la sortie standard ou dans le *debugger*.

Si l'on dispose d'une IHM, tester l'application en réalisant les cas d'utilisation comme le fera l'utilisateur une fois l'application mise en production.

Ecrire des tests.

Les 2 premières méthodes sont limitées : elles demandent une analyse humaine et sont donc

- source d'erreurs.
- chronophages.

Les tests s'écrivent dans un *source folder* dédié (`src/test/java`) :

Pour chaque classe que l'on veut tester : une classe de test (dans le même *package*), dont ne nom est celui de la classe que l'on veut tester suffixé par *Test*.

Dans chaque classe de test et pour chaque méthode que l'on veut tester :

- une méthode de test pour le cas nominal
- une méthode de test par exception potentiellement levée.

Les méthodes de test ne retournent rien (`void`), elles sont responsables de la comparaison entre résultat attendu et résultat obtenu.

Ecrire un test n'est pas trivial car cela suppose d'avoir les idées très claires sur la responsabilité de la méthode que l'on veut tester.

Le terme « tests » recouvre en réalité deux types de tests :

Les tests d'intégration

Il s'agit ici de tester la bonne réalisation des cas d'utilisation.

- Le test voit l'application comme une *boîte noire* dont seule la couche service (la façade) est exposée.
- Un échec du test ne veut pas dire que la méthode que l'on a appelée dans le test est fautive, cela signifie qu'il y a un problème quelque part dans la séquence d'appels (couche service ? couche business ? couche d'intégration ? Il faut investiguer…)

Les tests unitaires

Il s'agit ici de tester le comportement des méthodes d'une classe.

- On utilise alors des objets bouchons (*mocks* ou *stubs*) en au lieu des dépendances habituelles de la classes pour isoler au maximum la classe que l'on veut tester du reste de l'application.
- Cela n'est possible que pour une architecture à couplage faible (programmation par interface + injection de dépendances).
- Attention : cela brise le principe d'encapsulation puisu'il faut connaître les dépendances de la classe et le comportement de la méthode pour la tester.

Les tests se lancent de deux manières :

- Dans l'IDE (par le développeur).
- Lors du *build* dans un processus d'intégration continue.

Ces deux méthodes sont complémentaires :

- le développeur ne doit *commiter* son code sur le *repository* de sources que si tous les tests réussissent sur son poste de développement.
- le *build* sur le serveur d'intégration continue se déclenche suite au *commit* + *push* sur le *repository*.

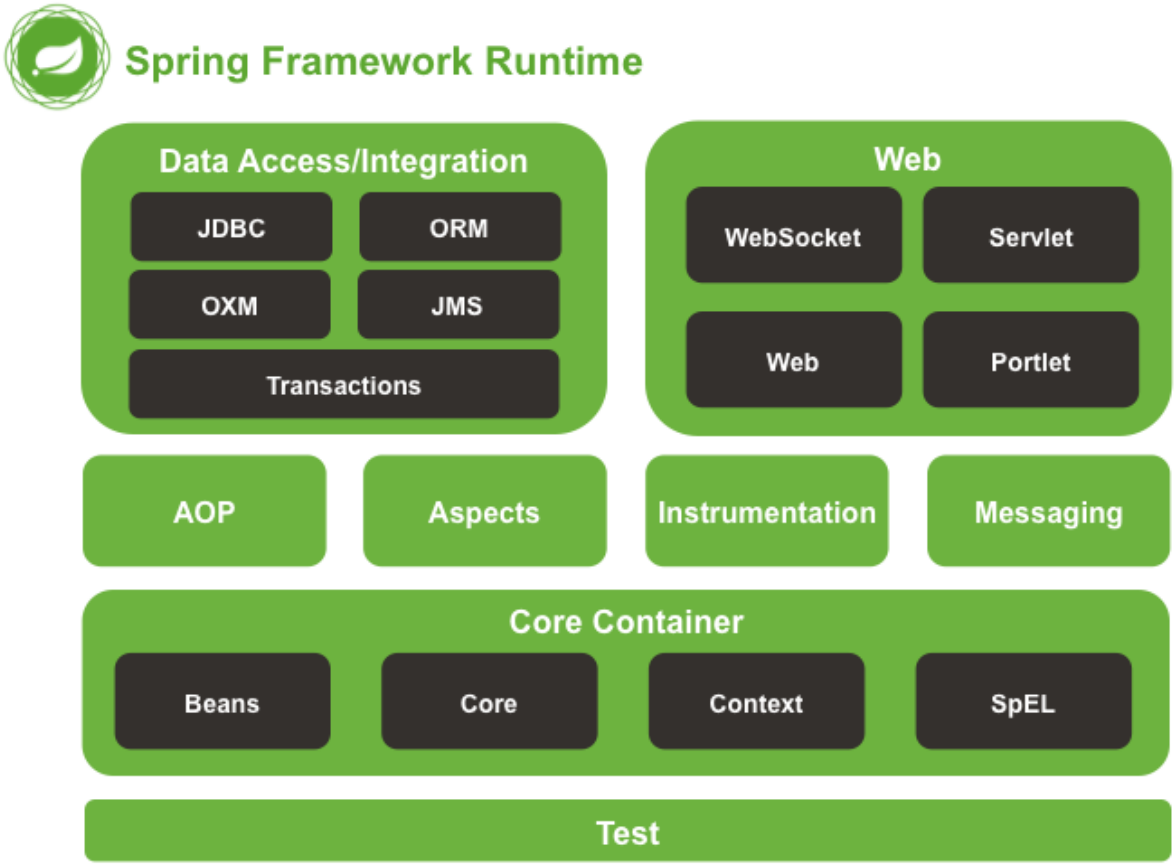
Spring : introduction

Problématiques de développement :

- La séparation des responsabilités
- La productivité des développements
- L'indépendance vis-à-vis de la plate-forme d'exécution
- Les tests

Réponse de Spring : un conteneur léger. 4 grandes parties :

- Core container
- Data access
- Web
- Integration avec d'autres technologies.


Spring Framework Runtime
Data Access/Integration
JDBC
ORM
OXM
JMS
Transactions
Web
WebSocket
Servlet
Web
Portlet
AOP
Aspects
Instrumentation
Messaging
Core Container
Beans
Core
Context
SpEL
Test

Objectifs de Spring :

We believe that:

- J2EE should be easier to use
- It is best to program to interfaces, rather than classes. Spring reduces the complexity cost of using interfaces to zero.
- JavaBeans offer a great way of configuring applications.
- OO design is more important than any implementation technology, such as J2EE.
- Checked exceptions are overused in Java. A framework shouldn’t force you to catch exceptions you’re unlikely to be able to recover from.
- Testability is essential, and a framework such as Spring should help make your code easier to test.

We aim that:

- Spring should be a pleasure to use
- Your application code should not depend on Spring APIs
- Spring should not compete with good existing solutions, but should foster integration. (For example, JDO, Toplink, and Hibernate are great O/R mapping solutions. We don’t need to develop another one.)

Core container

Permet la gestion de l'ensemble des composants d'une application :

- Prise en charge du cycle de vie des objets (*singleton*, *prototype*, *request*, *session*, etc…).
- Injection des dépendances.
- Interception (application de traitement avec et/ou après l'invocation des méthodes de nos *beans*).
- Exposition en RPC (RMI par exemple).

Ne nécessite aucune autre infrastructure qu'une machine virtuelle: la seule présence jar (spring-*.jar) dans le *classpath* suffit.

La configuration se fait au sein du projet par fichier xml ou par annotations.

Dépendance Maven : `spring-context` (qui dépend de `spring-beans`, `spring-core`, `spring-aop` et `spring-expression`), `spring-aspects`

Accès aux données et gestion des transactions

L'accès aux données est une problématique essentielle des applications, Spring propose différents outils pour faciliter ces opérations :

- `spring-jdbc` : des classes facilitant le requétage d'une base de données SQL (`JdbcTemplate` par exemple).
- `spring-orm` : couplage avec les technologies de mapping objet relationnel (JPA et Hibernate).
- `spring-tx` : une gestion des transactions dont nos méthodes peuvent profiter afin de garantir le respect des principes ACID dans notre application. Rappel des principes ACID : https://fr.wikipedia.org/wiki/Propriétés_ACID

Dépendances Maven : `spring-orm`, `spring-jdbc` et `spring-tx` (les deux dernières sont une dépendance transitive de la première).

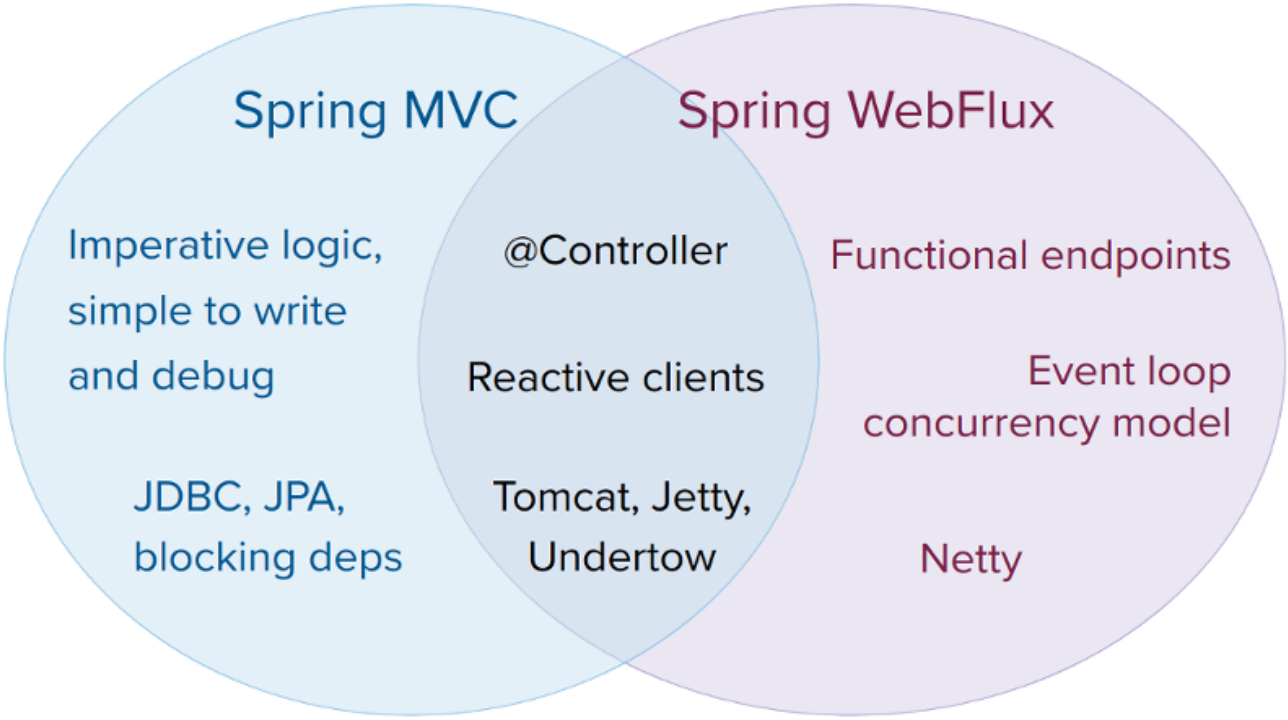
Enfin le projet Spring Data propose un couplage avec différentes technologies SQL et NoSQL afin de rendre notre application agnostique vis à vis de la base de données.

Web

Spring propose trois manières de développer des applications web :

- `spring-webmvc`, basé sur l'API *Servlet* et dans lequel chaque requête / réponse mobilise un *thread*.
- `spring-webflux`, basé sur *Netty* et permettant de développer des applications réactives non bloquantes.
- `spring-websocket`, pour la communication bi-directionnelle en mode connecté entre un navigateur et un serveur.

Spring MVC et Spring WebFlux sont deux moyens de développer des applications web (IHM ou REST) basées sur le protocole HTTP (requête/réponse).



Dépendance Maven : `spring-webmvc`, `spring-webflux`, `spring-websocket`

Integration

Rappel : *Spring should not compete with good existing solutions, but should foster integration.*

C'est ainsi que :

- `spring-webmvc` utilise Jackson ou GSON ou l'API JSON-B (à nous de choisir) pour la sérialisation ou déserialisation JSON.
- `spring-test` propose une extension Junit pour faciliter le test des *beans* Spring.
- `spring-aspects` s'appuie sur AspectJ pour la programmation orientée aspects (AOP).
- `spring-webmvc` utilise l'API Bean Validation pour la validation des objets plutôt que de redévelopper un mécanisme propriétaire.
- `spring-websocket` utilise SockJS pour proposer une abstraction vis à vis du protocole de communication.
- `spring-messaging` utilise STOMP pour l'échange de message via websocket.
- `spring-webflux` s'appuie sur l'API *reactive stream* (via Reactor ou RxJava) pour implémenter la programmation réactive.
- etc…

A chaque fois Spring favorise l'utilisation d'une technologie qui fait autorité dans son domaine plutôt que de "réinventer la roue".

Relation avec les API Java EE

Spring implémente les spécifications suivantes :

- Javax.inject : `@Named`, `@Inject`, `@Qualifer`…
- Common annotations : `@PostConstruct`, `@PreDestroy`, `@Resource`…

Et se couple avec les API suivantes :

- JPA
- JTA
- Bean validation
- JMS
- Servlet

Spring concurrence directement les API suivantes :

CDI (Context and Dependency Injection)

c'est le succès de Spring qui a conduit le JCP à crèer l'API CDI, avec laquelle Spring reste en concurrence. Spring n'implémente pas CDI.

EJB (Entreprise Java Beans)

Spring s'est proposé dès le début comme une alternative à EJB : une alternative plus simple à utiliser et moins intrusive. Si l'API EJB a corrigé tous ses défauts depuis sa version 3 (2009), le couplage obligatoire avec CDI est discutable et Spring reste une alternative plus cohérente.

Java Interceptors

Spring AOP concurrence directement cette API Java EE. Celle-ci se couple naturellement avec CDI et EJB mais ne propose pas d'interceptions à base de pointcuts.

JSF

JSF est une surcouche à Servlet qui standardise la manière de faire des pages web dynamiques avec rendu dynamique côté serveur. Spring propose quant à lui Spring MVC et Spring WebFlux, qui adressent la même problématique (mais selon une approche MVC là où JSF suit plutôt le principe MVP).

JAX-RS

JAX-RS est une surcouche à Servlet qui standardise la manière d'implémenter une API REST. Spring propose quant à lui Spring MVC et Spring WebFlux, qui adressent la même problématique.

Exemple

```
@Service ①
@Scope("singleton")
public class ShopServiceImpl implements ShopService {
    // couplage avec JPA : injection du contexte de persistance.
    @javax.persistence.PersistenceContext ②
    private javax.persistence.EntityManager em;

    @Override
    @javax.transaction.Transactional(TxType.REQUIRED) ③
    public Order processOrder(int shoppingCartId) {
        ShoppingCart sc = this.em.find(ShoppingCart.class, shoppingCartId);
        Order order = new Order();
        // opérations métiers sur sc, valorisation des propriétés de order
        this.em.persist(order);
        this.em.remove(sc);
        return order;
    }
}
```

① Spring va découvrir cette annotation et construire un objet à partir de cette classe

② Une fois l'instance créée, Spring va injecter le contexte de persistence JPA.

③ Spring découvir l'annotation `@Transactional` et, compte tenu de sa présence, inscrire dans l'application un *proxy* vers l'instance créé en 1.

```
@RestController ①
@Scope("singleton")
public class OrderEndpoint {
    @Autowired ②
    private ShopService shopService;

    @RequestMapping(path="orders", method=RequestMethod.POST) ③
    public ResponseEntity validateShoppingCart(@RequestParam int shoppingCartid) {
        Order order = this.shopService.processOrder(shoppingCartid);
        return ResponseEntity.created(order.getUri()).build(); ④
    }
}
```

① Spring va découvrir cette annotation et construire un objet à partir de cette classe

② Une fois l'instance créée, Spring va injecter une référence vers le *bean* de type `ShopService`

③ Pour indiquer à Spring que cette méhode doit être appelée à chaque requête POST sur /orders.

④ La servlet Spring récuperera ce que retourne notre méthode et fixera sur la réponse : un statut 201, un en-tête `location` dont la valeur est l'uri passé en paramètre à la méthode `created`.

Environnement de développement

- JDK >= 1.6 pour Spring 4, JDK >= 1.8 pour Spring 5.
- Framework Spring, idéalement récupérés via Maven.

Pour les dépendances Spring il faut de préférence utiliser le *bom* (*bill of material*) `spring-framework-bom` afin de ne pas avoir à repréciser la version de chaque dépendance Maven :

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-framework-bom</artifactId>
            <version>5.1.7.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Environnement d'exécution

Plusieurs solutions sont possibles :

1. Déployer un **war** sur un **serveur d'applications** Java EE (exemple : Jboss Wildfly ou EAP, IBM Websphere, Oracle WebLogic). Dans ce cas l'intention est de profiter de toutes les implémentations des API avec lesquelles Spring se couple : JTA, JPA, Servlet, JMS. Dans ce cas les *jar* Spring peuvent être ajoutées au classpath du serveur d'application afin que l'application y trouve tout ce dont elle a besoin pour fonctionner.
2. Déployer un **war** sur un **serveur web** (exemple : Tomcat, Jetty, Undertow ou Netty). Dans ce cas l'application doit porter dans le dossier `WEB-INF/lib` les librairies nécessaires à son bon fonctionnement (Spring et autres librairies tiers, Hibernate par exemple).
3. Déployer un **jar** et l'exécuter avec la commande `java -jar`. L'application est alors autonome et n'a besoin que de la JVM. Elle porte avec elle non seulement les librairies dont elle a besoin (Spring et autres librairies tierces) mais aussi son environnement d'exécution (un serveur web : Tomcat, Jetty, Undertow ou Netty). Ainsi l'application est un jar avec une méthode `main` comme point d'entrée.

Si la première solution est parfaitement valable, Spring encourage plutôt la seconde ou la troisième.

La troisième nécessite Spring boot.

Les deux dernières solutions fonctionnent particulièrement bien avec les environnements cloud PaaS car le déploiment consiste alors à seulement uploader et déployer un livrable sur un environnement managé (Java 8 et Tomcat 8 dans le cas d'Amazon Beanstalk par exemple).

Dans le cas de Spring Boot, les dépendances sont gérées différemment :

Tout d'abord le *bill of management* est différent :

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>2.1.5.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Ensuite vient la dépendance `spring-boot-starter-web` pour les applications web basé sur l'API *servlet* ou `spring-boot-starter-webflux` pour les applications web réactives :

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <!-- ou <artifactId>spring-boot-starter-webflux</artifactId> -->
    </dependency>
</dependencies>
```

Cette dernière permet de disposer dans le *classpath* de tout ce qu'il faut pour développer et déployer une application web :

- le framework Spring
- l'API *bean validation* et une implémentation (*Hibernate validator*)
- Jackson, pour la sérialisation et la déserialisation JSON.
- un serveur web embarqué :
  - *Tomcat embedded* pour `spring-boot-starter-web`
  - *Netty* pour `spring-boot-starter-webflux`

Notre application est alors un *jar*, dont le point d'entrée est :

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

Spring Data et gestion des transactions

- Principes
- Gestion des transactions
- Utilisation native de JPA
- Spring Data JPA

Principes

Spring propose un ensemble de solutions pour

1. faciliter les opérations CRUD (*create*, *read*, *update*, *delete*) de nos entités vers une base de données.
2. garantir le respect des principes ACID dans notre application.

Le premier point prend différentes formes, selon la technologie de base de données utilisée et la présence d'une solution de persistance faisant déjà autorité pour la base de données concernée.

Par exemple si nous utilisons une base de données relationnelle, nous pouvons capitaliser sur un *mapping* JPA existant. Dans ce cas Spring

- facilitera l'utilisation native de JPA en garantissant l'alignement du cycle de chaque `EntityManager` avec celui de la transaction en cours.
- proposera une surcouche à l'`EntityManager`, via les *repositories* du projet Spring Data.

En revanche si nous utilisons une base de données pour laquelle il n'existe pas d'équivalent à JPA (exemple : bases NoSQL), Spring proposera, via le projet Spring Data :

- des annotations pour *mapper* nos entités vers les tables de la base de données
- des *repositories* prêts à l'emploi pour les opérations CRUD.

Nous pouvons donc voir Spring Data comme un ensemble de librairies qui facilitent la persistance de nos objets dans une base de données (SQL ou NoSQL).

Introduction aux bases de données NoSQL : https://www.youtube.com/watch?v=qI_g07C_Q5I

Chaque librairie entretient un lien avec une technologie particulière. Ainsi il existe (liste non exhaustive) :

- `spring-data-cassandra`
- `spring-data-redis`
- `spring-data-mongodb`
- `spring-data-elasticsearch`
- `spring-data-neo4j`
- `spring-data-jdbc`
- `spring-data-r2dbc`
- `spring-data-jpa`

Spring Data est un projet satellite de Spring framework (au même titre que Spring boot, Spring security, Spring batch…). Nous reconnaissons cela au `groupId` de la dépendance Maven.

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-cassandra</artifactId>
    <version>2.1.8.RELEASE</version>
</dependency>
```

Les *repositories* proposées par Spring Data jouent le rôle de DAO. Deux types de *repositories* :

- `CrudRepository`
- `ReactiveCrudRepository`, si le *driver* de la base de données supporte les connexions réactives (exemple : MongoDB, Cassandra, Redis, ElasticSearch, R2DBC).

Liste non exhaustive des déclinaisons de Spring Data :

	repository réactif	*repository* non réactif	annotations pour le *mapping*
`spring-data-cassandra`	oui	oui	oui
`spring-data-redis`	oui	oui	oui
`spring-data-mongodb`	oui	oui	oui
`spring-data-elasticsearch`	oui	oui	oui
`spring-data-neo4j`	non	oui	non**
`spring-data-jdbc`	non	oui	oui
`spring-data-r2dbc`	oui	oui	oui
`spring-data-jpa`	non*	oui	non**

* Il n'y a pas de `ReactiveCrudRepository` pour JPA puisque JPA s'appuie sur JDBC et que JDBC est bloquant.

** car Neo4j et JPA proposent déjà un ensemble d'annotations pour déclarer un *mapping*.

Si nous utilisons une base de données relationnelle nous avons avantages à adopter une des stratégies suivantes :

- JDBC et `spring-data-jdbc`
- `r2dbc` et `spring-data-r2dbc`
- JPA au dessus de JDBC, avec `spring-orm` et éventuellement `spring-data-jpa`. A propos de JPA : voir annexe.

Gestion des transactions

Déclaration du `PlatformTransactionManager`

Il s'agit d'inscrire dans le contexte applicatif une implémentation de `PlatformTransactionManager`.

Spring propose une implémentation pour chaque technologie supportant les transactions. Citons :

- `DataSourceTransactionManager` (`spring-jdbc`) : si notre application accède à la base de données via JDBC.
- `JpaTransactionManager` (`spring-tx`) : si notre application utilise l'API JPA comme surcouche à JDBC.
- `JtaTransactionManager` (`spring-tx`) : si notre application délègue la gestion des transactions à un serveur d'applications Java EE.
- `MongoTransactionManager` (`spring-data-mongodb`) : si notre application accède à la base de données MongoDB.
- `Neo4jTransactionManager` (`spring-data-neo4j`) : si notre application accède à la base de données Neo4j.

A partir de la version 5.2, Spring proposera un support des transactions pour les bases de données relationnelles accédées via R2DBC et non JDBC (voir https://github.com/spring-projects/spring-framework/issues/22590).

Le `PlatformTransactionManager` va être responsable

1. de la démarcation transactionnelle (*begin* puis *commit* ou *rollback*).
2. de l'ouverture (au moment du *begin*) et de la fermeture (au moment du *commit* ou du *rollback*) de l'objet qui servira aux opérations CRUD :
   - une `javax.sql.Connnection` si nous utilisons l'implémentation `DataSourceTransactionManager`.
   - un `javax.persistence.EntityManager` si nous utilisons l'implémentation `JpaTransactionManager`.

Le second point est déterminant : pour une transaction donnée, toutes les opérations *CRUD* doivent se faire avec la même connection (transaction JDBC) ou avec le même `EntityManager` (transaction JPA).

Cette synchronisation est permise par le lien assumé entre :

- Un `DataSourceTransactionManager` et une `javax.sql.DataSource`. Constructeur :

```
public DataSourceTransactionManager(DataSource dataSource) {
    this();
    setDataSource(dataSource);
    afterPropertiesSet();
}
```

- Un `JpaTransactionManager` et un `javax.persistence.EntityManagerFactory`. Constructeur :

```
public JpaTransactionManager(EntityManagerFactory emf) {
    this();
    this.entityManagerFactory = emf;
    afterPropertiesSet();
}
```

Dès lors nous pouvons envisager la configuration suivante :

Persistance via JDBC :

```
public class ApplicationConfig {

    @Bean
    public DataSource datasource() {
        // code
    }
    @Bean
    public PlatformTransactionManager txManager(DataSource datasource){
        return new DataSourceTransactionManager(datasource);
    }
}
```

Persistance via JPA :

```
public class ApplicationConfig {

    @Bean
    public EntityManagerFactory emf() {
        // code
    }
    @Bean
    public PlatformTransactionManager txManager(EntityManagerFactory emf){
        return new JpaTransactionManager(emf);
    }
}
```

Notons que ces *beans* sont automatiquement inscrit si nous utilisons Spring Boot, grâce à l'annotation `@EnableAutoConfiguration` :

```
@EnableAutoConfiguration ① ②
@PropertySource("classpath:application.properties")
public class ApplicationConfig {

}
```

① si nous utilisons `spring-boot-starter-data-jdbc`, deux *beans* seront inscrits dans l'`ApplicationContext` : une `DataSource` et un `DataSourceTransactionManager`.

② si nous utilisons `spring-boot-starter-data-jpa`, trois *beans* seront inscrits dans l'`ApplicationContext` : une `DataSource`, une `EntityManagerFactory` et un `JpaTransactionManager`.

Le fichier `applicationProperties` doit bien sûr fournit bien sûr les paramètres de connexions à la base de données et, dans le cas de JPA, d'éventuels paramètres de configuration de l'`EntityManagerFactory` (dialect, etc...).

Exemple :

```
spring.datasource.url=# chaine de connexion
spring.datasource.driver-class-name=# driver de connexion
spring.datasource.username=# nom d'utilisateur
```

Documentation des propriétés : `https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html`

Le `transactionManager` peut bien sûr être injecté dans nos *beans*, via l'annotation `@Autowired`.

Les principales méthodes : `getTransaction`, `commit`, `rollback`

```
DefaultTransactionDefinition txDef = new DefaultTransactionDefinition();
/* valorisation des propriétés `propagationBehavior` (niveau de propagation), `readOnly` (pour préciser si la
transaction se fait en lecture seule ou non) */

TransactionStatus tx = txManager.getTransaction(txDef);
try{
    // opérations métiers
    txManager.commit(tx);
}catch(Exception e){
    txManager.rollback(tx);
}
```

Pour factoriser cette démarcation transactionelle dans un aspect :

```
@Component @Aspect
public class TransactionalAdvice {

    @Autowired
    private PlatformtransactionManager txManager;

    @Around("execution(* com.acme.app.service.*.*(..))")
    public Object txRequiresNew(ProceedingJoinpoint pjp) throws Throwable{
        DefaultTransactionDefinition txDef = new DefaultTransactionDefinition();
        TransactionStatus tx = txManager.getTransaction(txDef);
        try{
            Object ret = pjp.proceed();
            txManager.commit(tx);
            return ret;
        }catch(Exception e){
            txManager.rollback(tx);
        }
    }
}
```

Comme il s'agit d'un aspect, sa prise en compte suppose la présence de l'annotation `@EnableAspectJAutoProxy` sur la classe de configuration.

Avantages :

- aucune modification du code applicatif (est-ce une fin en soi ?)
- application en quelques lignes de l'`advice` à un grand nombre de méthodes (principe des *pointcuts*)

Inconvénients :

- moindre lisibilité dans le code applicatif. Nous ne voyons plus que des transactions vont s'appliquer autour des méthodes.
- nécessite de savoir lire et écrire les *pointcuts*
- risque qu'aucune transaction ne soit appliquée si le *pointcut* est mal écrit

L'annotation `@Transactional`

En général nous délégons à Spring la gestion des transactions autour de nos méthodes. Cela passe par l'inscription dans le contexte de *proxies* devant les instances de nos objets.

Jusqu'à présent nous avons évoqué la possibilité de "demander" à Spring de rendre une méthode transactionelle.

Cela signifie que toutes les opérations effectuées suite à l'invocation d'une méthode le seront dans une même transaction.

Mais cela n'est qu'un comportement parmi d'autres : le comportement correspondant au niveau de propagation `REQUIRED`

Les différents niveaux de propagation :

- `REQUIRED` : la méthode doit s'exécuter dans une transaction, s'il n'y en a pas une en cours le *proxy* en créera une.
- `MANDATORY` : la méthode doit s'exécuter dans une transaction, s'il n'y en a pas une en cours le *proxy* lèvera une exception.
- `REQUIRES_NEW` : la méthode doit s'exécuter dans une transaction dédiée, le *proxy* en créera une même si une transaction est déjà en cours.
- `NEVER` : la méthode ne doit pas s'exécuter dans une transaction, s'il y en a une en cours le *proxy* lèvera une exception.
- `NOT_SUPPORTED` : la méthode ne doit pas s'exécuter dans une transaction, s'il y en a une en cours le *proxy* la mettra en pause.
- `SUPPORTS` : la méthode peut être exécutée dans une transaction s'il y en a une en cours.

L'application des transactions par annotations se fait en ajoutant l'annotation `@Transactional` sur chacune des méthodes concernées.

Avantage : lisibilité

Inconvénient : lourd à mettre en place sur un grand nombre de méthodes.

Deux annotations sont supportées: celle de Spring et celle de JTA, toutes deux s'appellent `@Transactional`.

Leur prise en compte suppose que la classe de configuration soit annotée par `@EnableTransactionManagement`

Des *proxies* seront inscrits pour chaque classe annotée par par `@Transactional` ou ayant au moins une méthode annotée par `@Transactional`. Là encore l'attribut `proxyTargetClass` permet de préciser s'il doivent être des *proxies* par association ou par héritage.

L'annotation `javax.transaction.Transactional`

Il s'agit d'une annotation standard Java-EE, "comprise" par Spring.

Les attributs de cette annotation :

- `value` : niveau de propagation (`REQUIRED`, `MANDATORY`, etc...)
- `rollbackOn` : classes des exceptions qui doivent donner lieu à un *rollback* si elles sont levées
- `dontRollbackOn` : classes des exceptions qui ne doivent pas donner lieu à un *rollback* si elles sont levées

Avantage : aucune adhérence à Spring

Inconvénient : configuration plus limitée.

L'annotation `org.springframework.transaction.annotation.Transactional`

Cette annotation reprend sous forme d'attributs les propriété de la classe `DefaultTransactionDefinition` :

- `propagation` : niveau de propagation (`REQUIRED`, `MANDATORY`, etc...)
- `isolation` : niveau d'isolation
- `readonly` : transaction en lecture seule.
- `timeout` : délai maximum pour entre le début et la fin de la transaction.
- `rollbackFor` : classes des exceptions qui doivent donner lieu à un *rollback* si elles sont levées
- `noRollbackFor` : classes des exceptions qui ne doivent pas donner lieu à un *rollback* si elles sont levées

Avantage : des attributs supplémentaires par rapport à l'annotation `javax.transaction.Transactional`

Inconvénient : dépendance de la classe vis-à-vis de spring.

Traitements post transaction

Il arrive que nous souhaitions exécuter un traitement suite au *commit* de la transaction (envoyer un mail de confirmation par exemple).

Dans le cas d'une application par AOP ou par annotations le commit se fera après la fin de l'exécution de la méthode transactionnelle, il est toutefois possible de déclarer des traitements à exécuter **après** le *commit* de la transaction.

Cela se fait par la méthode statique `registerSynchronization` de la classe `TransactionSynchronizationManager` :

```
TransactionSynchronizationManager.registerSynchronization(new TransactionSynchronization(){
    @Override
    public void afterCommit() {
        // traitement à effectuer après le commit
    }
});
```

Transactions et tests unitaires

Dans une application n-tiers, seule la couche *service* a la responsabilité d'ouvrir une transaction. Les méthodes des *beans* des couches *business* et *dao* ont quant à elles vocation à être exécutées dans une transaction.

Ainsi, la propagation sera :

- `REQUIRED` ou `REQUIRES_NEW` pour la couche service
- `MANDATORY` pour les couches *business* et *dao*.

Dès lors, tester unitairement un *bean* de la couche *business* ne sera possible que si une transaction est en cours.

Il suffit pour cela de confier le test unitaire à Spring (annotation `@RunWith` ou `@ExtendWith`) et de décorer les méthode de test par `@Transactional`

Utilisation native de JPA

Objectif 1: disposer de l'`EntityManager` de la transaction en cours dans n'importe quel *bean* Spring :

```
@PersistenceContext
private EntityManager entityManager;
```

Objectif 2 : Déléguer à Spring la gestion des transactions autour des méthodes :

```
@Service
public class ShopServiceImpl implements ShopService {

    // injections

    @Transactional(propagation=Propagation.REQUIRED)
    public void validateShoppingCart(int shoppingCartId){
        // code
    }
}
```

Spring Data n'est pas nécessaire pour atteindre ces deux objectifs.

Il suffit d'ajouter les dépendance suivantes :

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
</dependency>
<!--
+ une dépendance vers une implémentation JPA, hibernate par exemple
+ une dépendance vers une implémentation un gestionnaire de pool de connexion
+ une dépendance vers le driver de la base de données.
-->
```

ou, si nous utilisons Spring Boot :

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-stater-data-jpa</artifactId> ①
</dependency>
<!-- + une dépendance vers le driver de la base de données. -->
```

① parmi les dépendances transitives : `spring-orm`, `HikariCP`, `hibernate-core`, `spring-data-jpa`

Et inscrire deux *beans* dans le contexte :

- 1 *bean* de type `javax.persistence.EntityManagerFactory` : cela permettra l'injection de l'`EntityManager` associé à la transaction en cours.
- 1 *bean* de type `org.springframework.transaction.PlatformTransactionManager` : cela permettra de déclarer des intercepteurs transactionnels autour de nos méthodes.

Si un fichier `META-INF/persistence.xml` existe dans le *classpath* la déclaration de l'`EntityManagerFactory` peut être faite ainsi :

```
public class ApplicationConfig {

    @Bean
    public EntityManagerFactory emf(){
        return Persistence.createEntityManagerFactory("pu1"); ①
    }

    @Bean
    public PlatformTransactionManager txManager(EntityManagerFactory emf){
        return new JpaTransactionManager(emf);
    }
}
```

① `pu1` est ici le nom de la `persistence-unit`

Le fichier `META-INF/persistence.xml` doit alors déclarer une `persistence-unit` nommée `pu1` qui précise :

- l'implémentation JPA à utiliser.
- les paramètres de connexion à la base de données.
- les propriétés spécifiques de l'implémentation JPA (`hibernate.dialect` par exemple).

Sans fichier `META-INF/persistence.xml` et avec un fichier *properties* (pour utiliser l'annotation `@TestPropertySource` dans les tests) :

```
@PropertySource("classpath:application.properties")
public class ApplicationConfig {

    @Bean
    public EntityManagerFactory emf(Environment env) {
        HikariDataSource ds = new HikariDataSource(); // ou autre implémentation de DataSource
        // valorisation des propriétés de connexion en utilisant env.getProperty
        PersistenceUnitInfo config = new PersistenceUnitInfo() { ①
            public DataSource getNonJtaDataSource() {
                return ds;
            }
            // autres méthodes à implémenter.
        }
        Properties jpaProperties = new Properties();
        // ajout des propriétés spécifiques de l'implémentation JPA (dialect par exemple)
        PersistenceProvider provider = new HibernatePersistenceProvider(); ①
        return provider.createContainerEntityManagerFactory(config, jpaProperties);
    }

    @Bean
    public PlatformTransactionManager txManager(EntityManagerFactory emf){
        return new JpaTransactionManager(emf);
    }
}
```

① `PersistenceProvider` et `PersistenceUnitInfo` proviennent du *package* `javax.persistence.spi`

Poblème : il y a environ 15 méthodes à implémenter dans l'interface `PersistenceUnitInfo`.

Une solution est de déléguer à Spring la préparation de `PersistenceUnitInfo`, via le `LocalContainerEntityManagerFactoryBean` :

```
@PropertySource("classpath:application.properties")
public class ApplicationConfig {

    @Bean
    public FactoryBean<EntityManagerFactory> emf(Environment env){
        HikariDataSource ds = new HikariDataSource(); // ou autre implémentation de DataSource
        // valorisation des propriétés de connexion en utilisant env.getProperty

        LocalContainerEntityManagerFactoryBean _emf = new LocalContainerEntityManagerFactoryBean();
        _emf.setPersistenceProviderClass(HibernatePersistenceProvider.class);
        _emf.setPackagesToScan("com.acme.app.model"); // là où se trouvent les entités mappées
        _emf.setDataSource(ds);
        Properties jpaProperties = new Properties();
        // ajout des propriétés spécifiques de l'implémentation JPA (dialect par exemple).
        _emf.setJpaProperties(props);
        return _emf;
    }

    @Bean
    public PlatformTransactionManager txManager(EntityManagerFactory emf){
        return new JpaTransactionManager(emf);
    }
}
```

`FactoryBean` est une interface de Spring, les *beans* qui l'implémentent suivent une construction en deux phases :

1. Nous transmettons des informations à la `FactoryBean<T>`
2. Spring les traite et produit un *bean* de type `T` qu'il inscrit dans le contexte (ici `T` est `EntityManagerFactory`).

Ou bien, pour disposer d'un *bean* de type `DataSource` indépendant :

```
@Configuration @PropertySource("classpath:application.properties")
public class ApplicationConfig {

    @Autowired
    private Environment env;

    @Bean(destroyMethod="close")
    public DataSource ds(){
        HikariDataSource ds = new HikariDataSource(); // ou autre implémentation de DataSource
        // valorisation des propriétés de connexion en utilisant env.getProperty
        return ds;
    }

    @Bean
    public FactoryBean<EntityManagerFactory> emf(){
        LocalContainerEntityManagerFactoryBean _emf = new LocalContainerEntityManagerFactoryBean();
        _emf.setPersistenceProviderClass(HibernatePersistenceProvider.class);
        _emf.setPackagesToScan("com.acme.app.model"); // là où se trouvent les entités mappées
        _emf.setDataSource(ds()); ①
        Properties jpaProperties = new Properties();
        // ajout des propriétés spécifiques de l'implémentation JPA (dialect par exemple).
        _emf.setJpaProperties(props);
        return _emf;
    }
    // déclaration du PlatformTransactionManager
}
```

① `this` étant un *proxy* (du fait de l'annotation `@Configuration`), l'appel à `ds()` sera reçu par le *proxy*. Celui-ci retournera ce que la méthode `ds` a retourné quand elle a été invoquée une première fois par Spring (compte tenu de l'annotation `@Bean`).

`@PropertySource("classpath:application.properties")` designe bien sûr un fichier contenant

- les propriétés de la `DataSource` (*username*, *password*, *url*, etc...).
- les propriétés spécifiques de l'implémentation JPA.

Exemple :

```
datasource.url= # chaine de connexion
datasource.driver-class-name= # driver de connexion
datasource.username= # nom d'utilisateur
datasource.password= # mot de passe

jpa.properties.hibernate.dialect= # dialect JPA
```

Ces valeurs sont accessibles via la méthode `getProperty` de l'`Environment`

L'intérêt de cette manière de faire est de pouvoir disposer d'un *bean* représentant la *datasource*, indépendamment de JPA.

Ainsi nous pouvons

- disposer de la `DataSource` par injection (annotation `@Autowired`)
- dans les tests :
  - utiliser `@TestPropertySource` pour exécuter nos tests avec une base dédiée aux tests.
  - utiliser les `@Profile` pour déclarer des `DataSource` alternatives (pour les tests par exemple).
  - utiliser l'annotation `@Sql` dans les tests unitaires, qui pose comme prérequis la présence d'un *bean* de type `DataSource`

Rappel : si nous utilisons Spring Boot (et donc la dépendance `spring-boot-starter-data-jpa`) alors ces 3 *beans* :

- `DataSource`
- `EntityManagerFactory`
- `PlatformTransactionManager`

sont inscits automatiquement dans l'`ApplicationContext` grâce à l'annotation `@EnableAutoConfiguration` et à la déclaration d'un fichier *properties* :

```
@EnableAutoConfiguration
@PropertySource("classpath:application.properties")
public class ApplicationConfig {
}
```

Le fichier `application.properties` que Spring utilisera pour créer les *beans* :

```
spring.datasource.url= # chaine de connexion
spring.datasource.driver-class-name= # driver de connexion
spring.datasource.username= # nom d'utilisateur
spring.datasource.password= # mot de passe

spring.jpa.properties.hibernate.dialect= # dialect JPA ①
```

① Chaque propriété qui doit être transmise à l'implémentation JPA doit être prefixé par `spring.jpa.properties.`. Exemple : `spring.jpa.properties.hibernate.dialect`, `spring.jpa.properties.hibernate.cache.region.factory_class`, etc...

Spring Data JPA

Le bénéfice est de pouvoir utiliser les *repositories* Spring Data plutôt que l'`EntityManager` et ainsi de disposer de méthodes plus adaptées à nos besoins courants de création, lecture, mise à jour et suppression. Citons quelques avantages :

- une méthode d'accès par l'`id` qui retourne un `Optional<T>` et non pas un `T` éventuellement nul.
- une méthode `save` qui invoque `merge` ou `persist` selon l'état de l'entité (détachée, attachée, transiente).
- une méthode de suppression par l'`id`
- un meilleur support des requêtes paginées.
- la possibilté de requéter par l'exemple (*query by example*).
- un mécanismes de requête dérivées.

L'activation de Spring Data JPA se fait en annotant la classe de configuration par `@EnableJpaRepositories` :

```
@EnableJpaRepositories
// autres annotations
public class ApplicationConfig {
    // code
}
```

Préalable si nous n'avons pas utilisé la dépendance `spring-boot-starter-data-jpa` mais seulement `spring-orm` :

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId> <!-- dépendance indirecte : spring-orm-->
    <version>2.1.8.RELEASE</version>
</dependency>
```

Spring JPA ne nécessite pas de configuration supplémentaire par rapport à une utilisation native de JPA.

Il faut donc inscrire dans l'`ApplicationContext`:

- 1 `EntityManagerFactory`
- 1 `PlatformTransactionManager`

Spring s'attend à trouver ces *beans* sous les noms `entityManagerFactory` et `transactionManager`.

Si nos *beans* sont nommées différemment (`emf` et `txManager` par exemple) nous pouvons écrire :

```
@EnableJpaRepositories(entityManagerFactoryRef= "emf", transactionManagerRef = "txManager")
// autres annotations
public class ApplicationConfig {
    // code
}
```

Définition d'un repository

Les *repositories* jouent le rôle de DAO, l'originalité est qu'il sont écrits sous la forme d'**interface** :

```
public interface PersonRepository extends JpaRepository<Person, Integer>
{
}
```

pour en disposer dans nos *beans* :

```
@Autowired
private PersonRepository repository;
```

C'est Spring qui se chargera de nous fournir une implémentation.

Techniquement `org.springframework.data.jpa.repository.JpaRepository<T, ID>` étend

- `org.springframework.data.repository.CrudRepository<T, ID>` : pour les opérations CRUD de base.
- `org.springframework.data.repository.PagingAndSortingRepository<T, ID>` : pour les requêtes paginées et triées.
- `org.springframework.data.repository.query.QueryByExampleExecutor` : pour les recherches par l'exemple.

Compte tenu de l'héritage de `CrudRepository<T, ID>`, `JpaRepository<T, ID>` propose notamment des méthodes basiques suivantes :

```
Optional<T> findById(ID id);

List<T> findAll();

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

saveAll(Iterable<S> entities);

void delete(T entity);

void deleteById(ID id);

long count();

boolean existsById(ID id);
```

Avec `spring-data-jpa`, l'implémentation donnée par Spring (`SimpleJpaRepository`) utilise bien sûr les méthodes `createQuery`, `find`, `remove`, `persist` et `merge` de l'`EntityManager`.

Compte tenu de l'héritage de `QueryByExampleExecutor`, `JpaRepository<T, ID>` propose des méthodes de recherches par l'exemple (*query by example*) :

```
<S extends T> Optional<S> findOne(Example<S> example);

<S extends T> Iterable<S> findAll(Example<S> example);

<S extends T> Iterable<S> findAll(Example<S> example, Sort sort);

<S extends T> Page<S> findAll(Example<S> example, Pageable pageable);

<S extends T> long count(Example<S> example);

<S extends T> boolean exists(Example<S> example);
```

`Pageable` permet d'exprimer une pagination. Exemple :

```
Pageable pageable = PageRequest.of(0, 10); ①
```

① 0 correspond à l'index de la page, 10 au nombre de résultats souhaités.

Définitions de requêtes par l'exemple

Les QBE *Query by example* sont un moyen efficace de réaliser des recherches multicritères.

Exemple :

```
Person p = new Person();
p.setFirstname("a");
p.setLastname("b");
```

Cela représente un "exemple" de ce que nous cherchons.

Bien sûr seules les propriétés renseignées donneront lieu à une restriction : la requête est calculée dynamiquement compte tenu de la valeur des propriétés de l'exemple.

Pour transmettre cet exemple à Spring et obtenir les `Person` dont le `lastname` vaut *a* et le `firstname` vaut *b* :

```
Person p = new Person();
p.setFirstname("a");
p.setLastname("b");

Example<Person> example = Example.of(p);
List<Person> resuts = personRepository.findAll(example); ①
```

① ou bien : `Page<Person> results = personRepository.findAll(example, PageRequest.of(0, 20));`

Les résultats seront les `Person` dont le `lastname` **est égal à** *a* et le `firstname` **est égal à** *b*

Pour contrôler les opérateurs (contient,commence par, finit par, etc...) nous pouvons transmettre à la méthode `Example.of` un *matcher* :

```
Person p = new Person();
p.setFirstname("a");
p.setLastname("b");

ExampleMatcher matcher = ExampleMatcher.matching().withStringMatcher(StringMatcher.CONTAINING);
Example<Person> example = Example.of(p, matcher);
List<Person> resuts = personRepositor.findAll(example); ①
```

① ou bien : `Page<Person> results = personRepository.findAll(example, PageRequest.of(0, 20))`

Les résultats seront les `Person` dont le `lastname` **contient** *a* et le `firstname` **contient** *b*

Définitions de requêtes personnalisées

Si nos besoins vont au delà de l'accès par l'`ID` et des requêtes par l'exemple, nous pouvons définir des requêtes spécifiques, exemple : rechercher toutes les personnes dont le nom contient telle ou telle valeur.

Là encore nous pouvons déléguer à Spring l'implémentation de cette requête, pour cela il suffit d'ajouter des méthodes à nos *repositories*.

Deux types de méthodes :

- celles dont le nom permet à Spring de *deviner* la requête.
- celles annotées par `@Query` et dont nous fournissons la requête JPA-QL.

Les méthodes dont le nom permet à Spring de deviner la requête doivent bien sûr respecter une convention. Exemple :

```
public interface PersonRepository extends JpaRepository<Person, Integer> {
    List<Person> findByLastnameContaining(String lastname);①

    Optional<Person> findByUsername(String username);①

    List<Person> findByLastnameIn(List<String> lastnames /*ou String... lastnames*/);②
}
```

① la requête par l'exemple aurait aussi pu être utilisée.

② la requête par l'exemple n'aurait pas pu être utilisée car le filtre implique plusieurs valeurs.

Le nom de la méthode se décompose en plusieurs parties. Pour la méthode `findByLastnameContaining` :

- `find` : signifie que nous voulons faire une recherche (et donc récupérer un ou plusieurs résultats).
- `By` : signifie que nous voulons appliquer un filtre
- `Lastname` : signifie que nous voulons que le filtre s'applique sur la propriété `lastname` (de la classe `Person` bien sûr)
- `Containing` : signifie que nous voulons les `Person` dont la propriété `lastname` **contient** la valeur reçue en argument.

Et le filtre peut bien sûr concerner plusieurs propriétés (exemple : `findByLastnameContainingAndFirstnameContaining`), la méthode doit avoir des paramètres correspondant à chacune d'entre elles.

si un des paramètres peut être nul il faut utiliser le type `Optional` ou ajouter l'annotation `@Nullable` sur celui-ci.

la restriction (ici sur la propriété `lastname`) s'appliquera même si la valeur transmise en argument est nulle. Ainsi `findByLastnameContaining(null)` retournera les `Person` sont le `lastname` est nul. Cela vient de ce que les requêtes sont préparées au déploiement de l'application et ne sont pas calculées à chaque appel.

Deux autres types de paramètres sont supportés :

- `Pageable`, pour définir la page souhaitée.
- `Sort`, pour orienter le classement des résultats.

Le type de retour peut quant à lui être

- `Optional<Person>` si la requête ne peut renvoyer au plus qu'un seul résultat (cf. `findByUsername`)
- `List<Person>` ou `Stream<Person>` si nous nous attendons à obtenir 0, 1 ou plusieurs résultats.
- `Page<Person>` si la méthode propose un paramètre de type `Pageable`.

Exemple d'utilisation de `Page` :

```
public interface PersonRepository extends JpaRepository<Person, Integer>
{
    Page<Person> findByLastnameContaining(String lastname, Pageable pageable);

    // méthode findByLastnameIn

    // méthode findByUsername
}
```

Pour l'appelant:

```
Page<Person> page = personRepository.findByLastnameContaining("a", PageRequest.of(0, 10));
```

Les méthodes annotées par `org.springframework.data.jpa.repository.Query` sont celles dont nous contrôlons la requête.

Exemple :

```
public interface PersonRepository extends JpaRepository<Person, Integer>
{
    @Query("select p from Person p where p.lastname like :p1 and p.firstname like :p2")
    List<Person> find(@Param("p1") String lastname, @Param("p2") String firstname);
}
```

Ici le nom de la méthode n'a aucune importance puisque c'est la requête que nous avons écrit qui sera exécutée.

Spring Data Jpa executera alors un code ressemblant à :

```
String queryString = "select p from Person p where p.lastname like :p1 and p.firstname like :p2";
TypedQuery<Person> query = entityManager.createQuery(queryString, Person.class);
query.setParameter("p1", "..." /* valeur passé au paramètre firstname de la méthode*/);
query.setParameter("p2", "..." /* valeur passé au paramètre lastname de la méthode*/);
List<Person> results = query.getResultList();
```

Si la valeur d'un paramètre peut être nulle, il faut annoter le paramètre par `@Nullable` ou utiliser le type `Optional<T>` (ici : `Optional<String>`).

```
public interface PersonRepository extends JpaRepository<Person, Integer>
{
    @Query("select p from Person p where p.lastname like :p1 and p.firstname like :p2")
    List<Person> find(@Param("p1") String lastname, @Param("p2") @Nullable String firstname);
}
```

la requête JPA est alors responsable de l'interprétation de la valeur nulle : si le paramètre `firstname` n'est pas renseigné cela veut il dire que nous souhaitons obtenir les personnes qui ont un `firstname` nul ou bien que nous ne souhaitons pas appliquer de restriction sur cette propriété ?

Enfin nous pouvons aussi disposer des type `Page<T>`, `PageRequest` et `Sort`, mais dans ce cas il faut préciser la requête de comptage :

```
public interface PersonRepository extends JpaRepository<Person, Integer>
{
    @Query(
        value="select p from Person p where p.lastname like :p1 and p.firstname like :p2",
        countQuery="select count(p) from Person p where p.lastname like :p1 and p.firstname like :p2"
    )
    Page<Person> find(@Param("p1") String lastname, @Param("p2") String firstname, Pageable pageable);
}
```

Spring Data MongoDB

Comme son nom l'indique, `spring-data-mongodb` est une couche d'accès aux données pour les opérations CRUD de nos entités vers une base de données MongoDB.

L'activation de Spring Data MongoDB se fait en ajoutant une dépendance à `spring-data-mongodb` et à un *driver* de connexion MongoDB :

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-mongodb</artifactId>
    <version>2.1.8.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver</artifactId>
    <version>3.8.2</version>
</dependency>
```

ou, avec Spring Boot :

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-mongodb</artifactId> ①
</dependency>
```

① dépendances transitives : `spring-data-mongodb` et `mongodb-driver`

Il faut ensuite annoter la classe de configuration par `@EnableMongoRepositories` et y déclarer deux *beans* :

- 1 `com.mongodb.client.MongoDbFactory` pour la connectivité vers la base de données.
- 1 `org.springframework.data.mongodb.core.MongoTemplate` nommé `mongoTemplate`.

```
@EnableMongoRepositories
// autres annotations
public class ApplicationConfig {
    @Bean
    public MongoDbFactory dbFactory() {
        return new SimpleMongoClientDbFactory("mongodb://localhost/mydatabase");
    }

    @Bean(name="mongoTemplate")
    public MongoTemplate mongoTemplate(MongoDbFactory mongoDbFactory) {
        return new MongoTemplate(mongoDbFactory);
    }
}
```

Avec Spring boot l'annotation `@EnableAutoConfiguration` suffit à inscrire automatiquement ces deux *beans*, moyennant la présence d'un fichier *properties* contenant les propriétés de connexion à la base de données :

```
@EnableMongoRepositories
@EnableAutoConfiguration
@PropertySource("classpath:application.properties")
// autres annotations
public class ApplicationConfig {
}
```

Le fichier `application.properties` :

```
spring.data.mongodb.uri=mongodb://localhost/mydatabase
```

Si le serveur supporte les transactions :

```
@EnableMongoRepositories
@EnableTransactionManagement
// autres annotations
public class MyApplication {

    @Bean
    public MongoDbFactory dbFactory() {
        return new SimpleMongoClientDbFactory("mongodb://localhost/ex0");
    }

    @Bean
    public PlatformTransactionManager txManager(MongoDbFactory  mongoDbFactory) {
        return new MongoTransactionManager(mongoDbFactory);
    }

    @Bean(name="mongoTemplate")
    public MongoTemplate mongoTemplate(MongoDbFactory mongoDbFactory) {
        return new MongoTemplate(mongoDbFactory);
    }
}
```

Avec Spring boot :

```
@EnableMongoRepositories
@EnableAutoConfiguration
@PropertySource("classpath:application.properties")
// autres annotations
public class ApplicationConfig {

    @Bean
    public PlatformTransactionManager txManager(MongoDbFactory  mongoDbFactory) {
        return new MongoTransactionManager(mongoDbFactory);
    }
}
```

Définition d'un repository

Ici les *repositories* doivent étendre l'interface `MongoRepository`

```
public interface PersonRepository extends MongoRepository<Person, Integer>
{
}
```

pour en disposer dans nos *beans* :

```
@Autowired
private PersonRepository repository;
```

C'est Spring qui se chargera de nous fournir une implémentation : `SimpleMongoRepository`.

Techniquement `org.springframework.data.mongodb.repository.MongoRepository<T, ID>` étend

- `org.springframework.data.repository.CrudRepository<T, ID>` : pour les opérations CRUD de base.
- `org.springframework.data.repository.PagingAndSortingRepository<T, ID>` : pour les requêtes paginées et triées.
- `org.springframework.data.repository.query.QueryByExampleExecutor<T>` : pour les recherches par l'exemple.

Nous disposons donc des mêmes facilités que celles vues précédemment :

- `CrudRepository<T, ID>` fournit les opérations basiques : `save`, `findById`, `delete`, `deleteById`, `existsById`, `findAll`.
- `QueryByExampleExecutor<T>` permet les recherches par l'exemple : `findOne`, `findAll`, `count`, `exists`.
- `PagingAndSortingRepository<T, ID>` permet l'expression d'une pagination et d'un ordre via les type `Page`, `Pageable` et `Sort`.

Définitions de requêtes personnalisées

Si nos besoins vont au delà de l'accès par l'`ID` et des requêtes par l'exemple, nous pouvons définir des requêtes spécifiques, exemple : rechercher toutes les personnes dont le nom contient telle ou telle valeur.

Là encore nous pouvons déléguer à Spring l'implémentation de cette requête, pour cela il suffit d'ajouter des méthodes à nos *repositories*.

Nous retrouvons les deux types de méthodes vus précédemment :

- celles dont le nom permet à Spring de *deviner* la requête.
- celles annotées par `@Query` et dont nous fournissons la requête MongoDB.

Les méthodes dont le nom permet à Spring de deviner la requête doivent bien sûr respecter une convention. Exemple :

```
public interface PersonRepository extends MongoRepository<Person, Integer> {
    List<Person> findByLastnameContaining(String lastname);①

    Optional<Person> findByUsername(String username);①

    List<Person> findByLastnameIn(List<String> lastnames /*ou String... lastnames*/);②
}
```

① la requête par l'exemple aurait aussi pu être utilisée.

② la requête par l'exemple n'aurait pas pu être utilisée car le filtre implique plusieurs valeurs.

La manière dont le nom de la méthode est interprété reprend les principes vus précédemment avec `spring-data-jpa`.

Notons toutefois le support de la géolocalisation, via le mot clé `Near` et des paramètres de type `Point` et `Distance`.

Exemple : `findByBirthplaceNear(Point point)` ou `findByBirthplaceNear(Point point, Distance distance)`

Enfin, puisque MongoDB supporte les requêtes paginées et le classement des résultats, les paramètres de type `Pageable` et `Sort` peuvent être utilisés :

```
public interface PersonRepository extends MongoRepository<Person, Integer> {
    Page<Person> findByLastnameContaining(String lastname, Pageable pageable, Sort sort);①

    Optional<Person> findByUsername(String username);①

    Page<Person> findByLastnameIn(List<String> lastnames, Pageable pageable, Sort sort);②
}
```

Le type de retour peut quant à lui être :

- `Optional<Person>` si la requête ne peut renvoyer au plus qu'un seul résultat (cf. `findByUsername`)
- `List<Person>` ou `Stream<Person>` si nous nous attendons à obtenir 0, 1 ou plusieurs résultats.
- `Page<Person>` si la méthode propose un paramètre de type `Pageable`.

l'utilisation du type de retour `Page<T>` conduit Spring a exécuter deux requête : une pour obtenir les résultats, une pour obtenir le nombre total d'éléments.

Les méthodes annotées par `org.springframework.data.mongodb.repository.Query` sont celles dont nous contrôlons la requête.

Exemple :

```
public interface PersonRepository extends MongoRepository<Person, Integer>
{
    @Query("{ 'firstname' : ?0, 'lastname': ?1 }")
    List<Person> find(String firstname, String lastname);
}
```

Ici le nom de la méthode n'a aucune importance puisque c'est la requête que nous avons écrit qui sera exécutée.

Nous pouvons aussi disposer des type `Page<T>`, `PageRequest` et `Sort` :

```
public interface PersonRepository extends JpaRepository<Person, Integer>
{
    @Query("{ 'firstname' : ?0, 'lastname': ?1 }")
    Page<Person> find(String firstname, String lastname, Pageable pageable);
}
```

Spring soumettra alors la requête deux fois :

- une pour avoir les résultats de la page demandée
- une pour avoir le comptage total et ainsi retournée un objet `Page` contenant cette information (méthode `getTotalElements()`).

Notons que l'annotation `@Query` peut propose les paramètres suivants :

- `String fields() default ""` , pour ne pas obtenir toutes les propriétés des résultats retournées. Exemple : `{ 'lastname' : 1}`.
- `String sort() default ""`, pour déclarer comment les résultats doivent être classés.
- `boolean count() default false`, pour indiquer que la requête doit être exécutée en vue d'obtenir un comptage des résultats.
- `boolean exists() default false`, pour indiquer que nous souhaitons seulement savoir s'il existent des résultats pour la requête.
- `boolean delete() default false`, pour indiquer que les résultats correspondant à la requête doivent être supprimés.

Bien sûr le type de retour de la méthode doit être cohérent avec la valeur des attributs : si `count` est valorisé à `true`, le type de retour doit être un nombre.

Spring Data REST

L'extension `spring-data-rest` permet d'exposer en REST nos *repositories* Spring Data

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-rest-webmvc</artifactId>
    <version>3.1.4.RELEASE</version>
</dependency>
```

ou, avec Spring boot :

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

Principes

Dès lors les opérations CRUD (*create, read, update, delete*) proposées par les *repositories* sont accessibles en REST.

Techniquement, `spring-data-rest-webmvc` propose une *servlet* : `RepositoryRestDispatcherServlet`, qui étend `DispatcherServlet`.

Le traitement normal de `DispatcherServlet` est conservé (recherche d'une méthode annotée par `@*Mapping`, invocation de celle-ci,etc...).

`RepositoryRestDispatcherServlet` n'intervient que si aucune méthode n'a été trouvée.

Il cherche alors parmi les *repositories* la méthode qui pourrait traiter la requête entrante (compte tenu du `path` et du verbe) et l'invoque pour obtenir de quoi fabriquer une réponse HTTP.

Spring Data Rest peut se coupler à

- `spring-data-jpa`
- `spring-data-mongodb`
- `spring-data-neo4j`
- `spring-data-gemfire`
- `spring-data-cassandra`

Soit une classe `User` et une classe `Country` et leur *repositories* :

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    @NotNull
    private String firstname, lastname, mailAddress;

    @ElementCollection
    private List<String> preferences;

    @ManyToOne
    private Country nationality;

    // getters & setters
}
```

```
@Entity
public class Country {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    @NotNull
    private String name;

    // getters & setters
}
```

```
public interface UserRepository extends JpaRepository<User, Integer> {
    Page<User> findByLastnameContaining(String lastname, Pageable pageable);

    Optional<User> findByMailAddress(String mailAddress);
}

public interface CountryRepository extends JpaRepository<Country, Integer> {}
```

Spring Data REST permettra de répondre aux requêtes suivantes :

- requête `GET` sur `/users` ⇒ `UserRepository::findAll`. La réponse obtenue : HTTP 200 dont le *body* contient :
  - les éléments (sérialisation de la `List<User>`)
  - des métadonnées (nombre de résultats, nombre de pages, numéro de la page en cours)
  - des liens : vers la page précédente et vers la page suivante.
- requête `GET` sur `/users/{id}` ⇒ `UserRepository::findById`. La réponse :
  - HTTP 404 si l'utilisateur `{id}` n'existe pas.
  - HTTP 200 si l'utilisateur `{id}` existe, le *body* de la réponse est alors une sérialisation de l'utilisateur `{id}`.
- requête `DELETE` sur `/users/{id}` ⇒ `UserRepository::deleteById`. La réponse :
  - HTTP 404 si l'utilisateur `{id}` n'existe pas.
  - HTTP 204 si si l'utilisateur `{id}` existe et a pu être supprimé.
- requête `PUT` sur `/users/{id}` avec un *body* déserialisable en `User` ⇒ `UserRepository::save`. La réponse :
  - HTTP 400 si le *body* de la requête n'est pas valide (contrainte non respectée par exemple).
  - HTTP 404 si l'utilisateur `{id}` n'existe pas.
  - HTTP 204 si l'utilisateur `{id}` existe.
- requête `POST` sur `/users` avec un *body* déserialisable en `User` ⇒ `UserRepository::save`. La réponse :
  - HTTP 400 si le *body* de la requête n'est pas valide (contrainte non respectée par exemple).
  - HTTP 201 si l'utilisateur a pu être créé.

Exemple de réponse à une requête `GET` sur `/users` :

```
{
    "_embedded":{
        "users":
        [
            {
                "firstname":"John",
                "lastname":"Doe",
                "mailAddress":"jdoe@acme.com",
                "_links" : [
                    {"self" : {"href" : "http://localhost:8080/users/1"}},
                    {"nationality" : {"href" : "http://localhost:8080/users/1/nationality"}},
                    {"preferences" : {"href" : "http://localhost:8080/users/1/preferences"}},
                ]
            }
            // autres résultats
        ]
    },
    "_links" : {
        "first" : {"href" : "http://localhost:8080/users?page=0&size=20"},
        "next" : {"href" : "http://localhost:8080/users?page=1&size=20"},
        "last" : {"href" : "http://localhost:8080/users?page=1&size=20"}
    },
    "page" : {"size" : 20, "totalElements" : 30, "totalPages" : 2, "number" : 0}
}
```

Au moment de représenter la nationalité de l'utilisateur "John Doe", Spring Data Rest a du prendre une décision :

- si un *repository* existe pour le type de la relation (ici : `Country`)
  - s'il n'existe pas de projection pour les `Country` : nous obtenons un lien vers la ressource (ici : `/users/1/nationality`).
  - si une projection existe pour les `Country`, celle-ci sera utilisée pour représenter la nationalité de l'utilisateur.
- s'il n'existe pas de *repository* pour le type de la relation (ici : `Country`), Spring data Rest n'a d'autre choix que d'en présenter les propriétés.

Les projections

Pour contrôler la manière dont une entité est représentée lorsqu'elle est incorporée nous pouvons déclarer des projections.

Une ressource est incorporée lorsqu'elle est représenté dans un élément parent.

Ainsi dans l'exemple précédent

- chaque `User` est incorporé dans la liste de résultats.
- chaque `Country` est incorporé en tant que propriété (`nationality`) des utilisateurs.

Lorsqu'une ressource est incorporée nous avons avantage à la représenter de manière synthétique.

Nous déclarons alors des projections. Exemple :

```
@RepositoryRestResource(excerptProjection = UserProjection.class)
public interface UserRepository extends JpaRepository<User, Integer>
{
    public interface UserProjection{

        @Value("#{target.firstname} #{target.lastname}") ①
        String getFullname();

        @Value("#{target.nationality.name}") ①
        String getNationality();
    }
}
```

① `target` désigne l'entité `User` qu'il faut représenter.

La projection s'appliquera automatiquement dès qu'il s'agira de représenter un `User` en tant que ressource incorporée.

Les `User` seront alors repésentés sous une forme simplifiée : seules les propriétés `fullname` et `nationality` seront exposées.

Exemple de réponse à une requête `GET` sur `/users` :

```
{
    "_embedded":{
        "users":
        [
            {
                "fullname":"John Doe",
                "nationality":"France",
                "_links" : [
                    {"self" : {"href" : "http://localhost:8080/users/1"}},
                    {"nationality" : {"href" : "http://localhost:8080/users/1/nationality"}},
                    {"preferences" : {"href" : "http://localhost:8080/users/1/preferences"}},
                ]
            }
            // autres résultats
        ]
    },
    "_links" : {
        "first" : {"href" : "http://localhost:8080/users?page=0&size=20"},
        "next" : {"href" : "http://localhost:8080/users?page=1&size=20"},
        "last" : {"href" : "http://localhost:8080/users?page=1&size=20"}
    },
    "page" : {"size" : 20, "totalElements" : 30, "totalPages" : 2, "number" : 0}
}
```

Si la requête porte sur la ressource elle-même (exemple : `GET` sur `users/1`) alors celle-ci ne sera pas incorporée et la projection ne sera pas utilisée. Dès lors :

- toutes les propriétés de type valeur (`String`, primitifs, types date) seront exposées (pour `User` : `firstname`, `lastname` et `mailAddress`).
- les relations seront représentées suivant le choix fait par Spring Data REST :
  - lien s'il y a un *repository* pour le type de la relation.
  - représentation complète s'il n'y a pas de *repository* pour le type de la relation.
  - représentation simplifiée en utilisant une projection s'il y a une projection (et donc un *repository*) pour le type de la relation.

Dès lors si une projection existe pour la classe `Country` :

```
@RepositoryRestResource(excerptProjection = CountryProjection.class)
public interface CountryRepository extends JpaRepository<Country, Integer>{

    public interface CountryProjection{

        @Value("#{target.name}")
        String getCountry();
    }
}
```

Alors un `GET` sur `/users/1` retournera :

```
{
    "firstname" : "John",
    "lastname": "Doe",
    "mailAddress": "jdoe@acme.com",
    "_embedded" : {
        "nationality" : {
            "country" : "France",
            "_links" : {
                "self" : {
                      "href" : "http://localhost:8080/countries/1{?projection}",
                      "templated" : true
                    }
            }
        }
    }
}
```

En revanche la projection n'aura pas d'influence sur la manière dont la nationalité d'un `User` est représenté lors d'un `GET` sur `/users` car dans ce cas c'est la projection définie pour `User` (`excerptProjection = UserProjection.class`) qui sera utilisée.

Notons que la projection peut aussi s'appliquer si la ressource n'est pas incorporée (exemple : `GET` sur `/users/1`) mais il faut alors :

- que la la projection soit nommée et associée explicitement au type `User`

```
@Projection(name="my-projection", types=User.class)
public interface UserProjection{
    @Value("#target.lastname")
    String getLastname();

    @Value("#target.firstname")
    String getFirstname();

    @Value("#target.nationality.name")
    String getNationality();
}
```

- que la requête précise la projection à utiliser : `GET` sur `users/1?projection=my-projection`

A propos du découplage entre les entités métier et les ressources exposées par une API REST : https://stackoverflow.com/a/38876046/10938834

Les recherches

Les `@Query` sont aussi accessibles via le *path* `search`

```
@RepositoryRestResource(excerptProjection = UserProjection.class)
public interface UserRepository extends JpaRepository<User, Integer>
{
    Page<User> findByLastnameContaining(String lastname, Pageable pageable);

    Optional<User> findByMailAddress(String mailAddress);

    @Projection(name="my-projection", types=User.class)
    public interface UserProjection{
        // code
    }
}
```

requête `GET` sur `/users/search/findByLastnameContaining?lastname=a&page=1`

conduira à invoquer la méthode `findByLastnameContaining` de `UserRepository` avec le paramètre `lastname` transmis à la méthode `findByLastnameContaining`

La réponse obtenue : HTTP 200 dont le *body* :

- les éléments (sérialisation de la `List<User>` et application d'une projection si elle existe)
- des métadonnées (nombre de résultats, nombre de pages, numéro de la page en cours)
- des liens : vers la page précédente et vers la page suivante.

Pour contrôler le *path* sous lequelle la recherche est accessible nous pouvons utiliser l'annotation `RestResource` :

```
@RepositoryRestResource(excerptProjection = UserProjection.class)
public interface UserRepository extends JpaRepository<User, Integer>
{
    @RestResource(path="byLastname")
    Page<User> findByLastnameContaining(String lastname, Pageable pageable);

    @RestResource(path="byMailAddress")
    Optional<User> findByMailAddress(String mailAddress);

    @Projection(name="my-projection", types=User.class)
    public interface UserProjection{
        // code
    }
}
```

Dès lors la recherche d'utilisateurs par leur nom se fait par un `GET` sur `users/search/byLastname`

Les erreurs

La classe `RepositoryRestExceptionHandler` propose des gestionnaires d'exceptions (`@ExceptionHandler`) qui, lorsqu'une exception survient, tente de fabriquer une réponse HTTP appropriée.

Quelques exemples :

- `DataIntegrityViolationException` ⇒ statut 409 (*conflict*)
- `ResourceNotFoundException` ⇒ statut 404 (*not found*)
- `RepositoryConstraintViolationException` ⇒ statut 400 (*bad request*)

Les évenements

Régulièrement des traitements métiers doivent être réalisés avant et/ou après les opérations de création, modification ou suppression.

Ceux-ci peuvent être implémentés dans des méthodes, celles-ci doivent être annotées par une des annotations suivantes :

- `@HandleBeforeCreate` : sur une méthode devant être appelée avant la création d'une ressource.
- `@HandleAfterCreate` : sur une méthode devant être appelée après la création d'une ressource.
- `@HandleBeforeSave` : sur une méthode devant être appelée avant la mise à jour d'une ressource.
- `@HandleAfterSave` : sur une méthode devant être appelée après la mise à jour d'une ressource.
- `@HandleBeforeDelete` : sur une méthode devant être appelée avant la suppression d'une ressource.
- `@HandleAfterDelete` : sur une méthode devant être appelée avant la suppression d'une ressource.

Rappel :

- Les créations sont initiées par des requêtes `POST` (exemple : `POST` sur `/users`).
- Les mises à jour sont initiées par des requêtes `PUT` ou `PATCH` (exemple : `PUT` sur `/users/1`).
- Les suppressions sont initiées par des requêtes `DELETE` (exemple : `DELETE` sur `/users/1`).

Ces méthodes seront bien sûr appelées par `spring-data-rest`.

Exemple :

```
@Component
@RepositoryEventHandler ①
public class UserBusinessRules
{
    private BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();

    @HandleBeforeCreate
    public void encryptPassword(User user){ ②
        user.setPassword(encoder.encode(user.getPassword()));
    }
}
```

① Pour indiquer à Spring que certaines méthodes de cette classes sont des *event handlers*.

② sera appelée avant la sauvegarde du `user` (méthode `save` de `UserRepository`). Celui-ci a été reconstruit d'après le *body* d'une requête `POST` sur `/users`).

Spring Security

- Principes
- Configuration
- Définition d'un référentiel d'utilisateur
- Authentification
- Autorisations
- Tests

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>5.1.5.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>5.1.5.RELEASE</version>
</dependency>
```

ou, avec Spring boot :

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

Principes

Spring Security permet de gérer finement la sécurité d'une application : l'**authentification** et les **autorisations**.

Cela concerne la couche web et la couche application :

Sur la couche web :

- Comment permettre à l'utilisateur de s'authentifier ?
- Qui peut accéder à quelle route (exemple : `/users/1`) ?
- Dans une page, qui peut voir quel élément (exemple : un bouton « supprimer » ?)
- Comment obtenir le contexte d'authentification de l'utilisateur ?

Sur la couche applicative :

- Qui peut effectuer quel traitement ?
- Comment obtenir le contexte d'authentification de l'utilisateur ?

Configuration

Spring Security propose une réponse à toutes ces questions.

Le point de départ est une classe de configuration, celle-ci doit

- étendre la classe abstraite `WebSecurityConfigurerAdapter`
- être annotées par `@EnableWebSecurity` (spécialisation de `@Configuration`)

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter{
}
```

A nouveau, le fait d'étendre une classe abstraite permet d'être guidé dans la configuration : chaque élément à configurer prend la forme d'une méthode à redéfinir.

Cette classe doit être chargée au démarrage de l'application, en même temps que la classe de configuration correspondant au contexte applicatif.

Ensuite, il faut inscrire le filtre (au sens `javax.servlet.Filter`) Spring security :

```
DelegatingFilterProxy filter = new DelegatingFilterProxy();
filter.setTargetBeanName("springSecurityFilterChain");
// ou DelegatingFilterProxy filter = new DelegatingFilterProxy("springSecurityFilterChain");
```

C'est l'annotation `@EnableWebSecurity` sur la classe de configuration qui permet une inscription automatique d'un *bean* nommé *springSecurityFilterChain*.

Celui-ci doit être déclaré auprès du *servlet container*. Plusieurs manières de faire :

- dans le fichier `WEB-INF/web.xml`.
- dans la méthode `onStartup` d'une implémentation de `WebApplicationInitializer` (via la méthode `servletContext.addFilter`).
- dans la méthode `getServletFilters` d'une classe héritant de `AbstractAnnotationConfigDispatcherServletInitializer`.

Avec Spring boot l'inscription du filtre est automatique si la dépendance Maven `spring-boot-starter-security` est déclarée.

Définition d'un référentiel utilisateur

Cette classe qui étend `WebSecurityConfigurerAdapter` peut redéfinir une méthode `configure` pour définir le référentiel utilisateurs :

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    // code
}
```

L'`AuthenticationManagerBuilder` propose une méthode `userDetailsService` permettant de déclarer un `UserDetailsService`.

```
public interface UserDetailsService{
    UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;
}
```

Le `UserDetails` retourné contient le nom d'utilisateur, le mot de passe et les permissions.

Spring security appelle la méthode `loadUserByUsername` au moment de l'authentification, si le `UserDetails` retourné est non nul il vérifie que les mots de passe correspondent. Si oui : l'authentification est réussie.

Implémentation possible de `UserDetails` : `User`. Constructeur :

```
public User(String username, String password, Collection<? extends GrantedAuthority> authorities) {
    // code
}
```

Dès lors nous pouvons écrire :

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    UserDetailsService uds = username -> {
        // accès à un référentiel pour retourner un objet UserDails
        // correspondant au username reçu en argument.
    };
    auth.userDetailsService(uds).passwordEncoder(new BCryptPasswordEncoder());
}
```

ou, puisqu'il y a une méthode `userDetailsService` dans la classe parente `WebSecurityConfigurerAdapter` :

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(userDetailsService()).passwordEncoder(new BCryptPasswordEncoder());
}

@Override ①
protected UserDetailsService userDetailsService() {
    UserDetailsService uds = username -> {
        // accès à un référentiel pour retourner un objet UserDails
        // correspondant au username reçu en argument.
    };
    return uds;
}
```

① Eventuellement annoté par `@Bean` si nous voulons obtenir le `UserDetailsService` par injection.

A propos de BCrypt : http://dustwell.com/how-to-handle-passwords-bcrypt.html, https://security.stackexchange.com/a/6415

Mais l'`AuthenticationManagerBuilder` propose plusieurs méthodes à qui l'on peut déléguer la création du `UserDetailsService` :

- `inMemoryAuthentication`
- `jdbcAuthentication`
- `ldapAuthentication`

Exemple avec `inMemoryAuthentication`

```
auth.inMemoryAuthentication().withUser("john.doe").password("psw").roles("USER");
```

- Exemple avec `jdbcAuthentication`

```
DriverManagerDataSource ds = new DriverManagerDataSource(); /* ou autre implémentation de DataSource.*/
/* valorisation des propriétés correspondant aux paramètres de connexion à la base de données.*/
auth.jdbcAuthentication()
    .dataSource(ds)
    .passwordEncoder(new BCryptPasswordEncoder()) ①
    .usersByUsernameQuery("select username, password, enabled from users where username=?")
    .authoritiesByUsernameQuery("select username, authority from authorities where username=?");
```

Dès lors la déclaration d'un `UserDetailsService` est à réservée aux cas non prévu par Spring (exemple : le référentiel utilisateur est une base de données NoSQL).

Authentification

La procédure d'authentification se configure via une seconde méthode `configure` de la classe abstraite `WebSecurityConfigurerAdapter`.

Implémentation par défaut de cette méthode :

```
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
            .anyRequest().authenticated()
            .and()
        .formLogin().and()
        .httpBasic();
}
```

`.formLogin()` rend accessibles une nouvelle route :

- `/login` (en `GET` pour afficher le formulaire, en `POST` pour traiter la soumission du formulaire)

Et dans tous les cas, un `POST` sur `/logout` deconnectera l'utilisateur.

Notons aussi que la protection CSRF est automatiquement activée pour toutes les requêtes POST.

Cette implémentation a le mérite de convenir aussi bien aux IHM (du fait du formulaire de login) qu'aux API REST (lecture du nom d'utilisateur et du mot de passe dans l'en-tête `Authorization`)

Mais généralement :

- nous ne gardons que le formulaire de login pour les IHM.
- nous ne gardons que l'authentification basique pour une API REST.

	IHM	REST
mode d'authentification	formulaire de login	Basic
après la deconnexion (`POST` sur `/logout`)	redirection vers une vue	Réponse HTTP 204
requête non authentifiée	redirection vers le formulaire de login	Réponse HTTP 401
requête non autorisée	Réponse HTTP 403	Réponse HTTP 403

Dès lors l'implémentation par défaut de la méthode `configure` demande à être redéfinie.

Authentification pour une IHM

Pour n'activer que l'authentification basique :

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.httpBasic();
}
```

Pour n'activer que l'authentification par un formulaire de login au format HTML.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.formLogin();
}
```

Le succès d'une authentification conduit à maintenir en session le contexte d'authentification de l'utilisateur et dès lors d'authentifier toutes les requêtes suivantes.

Si notre application est déployée sur un *cluster*, il faudra veiller à mettre en oeuvre une des 3 solutions suivantes :

- affinité de session.
- réplication des sessions sur les noeuds du *cluster*.
- stockage des informations associées aux sessions utilisateur dans une base de données.

Les deux dernières options peuvent être mises en oeuvre avec `spring-session`

La procédure de connexion peut être personnalisée.

Exemple:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.formLogin()
        .loginPage("/signin").permitAll()
    .and()
        .logout()
            .logoutUrl("/signout")
            .logoutSuccessUrl("/signout-success").permitAll();
}
```

Il est de notre responsabilité de déclarer des routes `/signin` et `/signout-success` (via des `ViewController` par exemple) et de créer les vues correspondantes.

Notre vue associé à `/signin` devra inclure un champ caché pour le token CSRF :

```
<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}" />
```

Authentification dans une API REST

Authentification `Basic`

L'authentification `Basic` est un bon moyen pour notre API de recevoir le nom et le mot de passe de l'utilisateur.

Pour l'activer le support de l'authentification basique et

- retourner une réponse HTTP 403 aux requête non autorisées sur des ressources sécurisées.
- retourner une réponse HTTP 401 aux requêtes non autentifiées sur des ressources sécurisées.
- retourner une réponse HTTP 204 à une requête de deconnexion (`POST` sur `/logout`).

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.httpBasic()
        .authenticationEntryPoint((req, resp, ex) -> resp.setStatus(401)) ①
        .and()
        .exceptionHandling().accessDeniedHandler((req, resp, ex) -> resp.setStatus(403)) ②
        .and()
        .logout().logoutSuccessHandler((req, resp, auth) -> resp.setStatus(204)) ③
        .and()
        .csrf().disable();④
}
```

① pour retourner une réponse HTTP 401 aux requêtes non autentifiées sur des ressources sécurisées.

② pour retourner une réponse HTTP 403 aux requête non autorisées sur des ressources sécurisées.

③ pour retourner une réponse HTTP 204 à une requête de deconnexion.

④ pour désactiver la protection CSRF.

Pour celui qui accède à notre API (exemple : une page web) :

```
const headers = new Headers({"Authorization": `Basic ${btoa("john.doe:azerty")}` });
fetch("https://localhost:8080/authentication", { headers: headers, method: "POST" })
```

Exemple de méthode répondant à un `POST` sur `/authentication` :

```
@PostMapping("authentication")
public Map<String, Object> onSuccessfulAuthentication(Authentication auth) {
    String username = auth.getName();

    List<String> authorities = auth.getAuthorities().stream()
        .map(a -> a.getAuthority())
        .collect(Collectors.toList()

    return Map.of(
        "username", username,
        "authorities", authorities
        );
}
```

Le filtre Spring security va accéder à l'entête `Authorization` et vérifier si le couple nom d'utilisateur / mot de passe est valide.

- Si oui le fitre :
  1. invoque de la méthode associé au *path* `/authentication`. Celle-ci peut retourner des informations utiles à l'appelant (nom et permissions de l'utilisateur par exemple).
  2. maintient en session l'`Authentication` de l'utilisateur, d'où un `Set-Cookie` contenant le `JSESSIONID` dans la réponse.
- Sinon : réponse HTTP 401

Une fois le *cookie* obtenu, c'est bien sûr via celui-ci que toutes les requêtes suivantes seront authentifiées et il n'est plus nécessaire de transmettre l'en-tête `Authorization`.

Notons que le *cookie* a avantage à être sécurisé :

- `httpOnly` pour se protéger contre les attaques XSS. Un *cookie* `httpOnly` est en effet inacessible en Javascript.
- `secured` pour ne transmettre le *cookie* que si le protocole est HTTPS.
- `sameSite` pour se protéger contre les attaques CSRF. Le navigateur ne transmet le *cookie* dans la requête que si elle est initiée depuis le même site (*host:port*) que le *endpoint* (*host:port*) sur lequel porte la requête.

Ces paramètres sont définis au niveau du *servlet container*.

- Sans Spring boot et avec le `ServletContext` : voir la méthode `getSessionCookieConfig()`
- Avec Spring boot : voir les propriétés `server.servlet.session.cookie.secure` et `server.servlet.session.cookie.http-only` du fichier `application.properties`

Authentification par *token*

Si nous ne souhaitons pas utiliser de session alors la méthode `configure` sera écrite ainsi :

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.httpBasic()
    and()
    .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
}
```

Le *endpoint* qui recoit la demande d'authentification (`/authentication` dans notre exemple) doit alors retourner un *token* (JWT par exemple), qui permet d'authentifier toutes les requêtes suivantes.

```
@PostMapping("authentication")
public Map<String, Object> authentication(Authentication auth) {
    String username = auth.getName();
    List<String> authorities = auth.getAuthorities()
        .stream()
        .map(a -> a.getAuthority())
        .collect(Collectors.toList());
    String token = Jwts.builder() ①
        /* alimentation du builder */
        .compact();
    map.put("token", token);
    return map;
}
```

① `Jwts` provient de la librairie *JSON Web Token* (https://www.jsonwebtoken.io/)

Dans toutes les requêtes HTTP qui suivent l'authentification, ce *token* doit être transmis dans l'en-tête `Authorization` en étant prefixé par `Bearer`. Exemple :

```
const token = "the-token";
const headers = new Headers({"Authorization": `Bearer ${token}` });
fetch("https://localhost:8080/sensitive-endpoint", { headers: headers });
```

A propos du stockage du *token* sur le client : https://stormpath.com/blog/where-to-store-your-jwts-cookies-vs-html5-web-storage

A propos de JWT :

- https://kev.inburke.com/kevin/things-to-use-instead-of-jwt/
- https://cryto.net/%7Ejoepie91/blog/2016/06/13/stop-using-jwt-for-sessions/
- https://cryto.net/%7Ejoepie91/blog/2016/06/19/stop-using-jwt-for-sessions-part-2-why-your-solution-doesnt-work/
- https://waiting-for-dev.github.io/blog/2017/01/24/jwt_revocation_strategies/

Reste à configurer Spring security pour que le token JWT soit détecté :

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.httpBasic()
    .and()
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
    .and()
        .oauth2ResourceServer().jwt();
}
```

Ainsi Spring security saura extraire le *token* JWT de l'en-tête `Authorization`.

Reste à le décoder, pour cela ceux possibilité :

- configurer Spring pour qu'il soumette le token à un serveur OAuth2.
- déclarer un *bean* de type `org.springframework.security.oauth2.jwt.JwtDecoder`. Cette interface n'a qu'une seule méthode :

```
Jwt decode(String token) throws JwtException;
```

Elle peut facilement être implémenté dans une méthode productrice :

```
public @Bean JwtDecoder decoder(){
    return token -> {
        // code retournant un org.springframework.security.oauth2.jwt.Jwt
    }
}
```

Accès au contexte d'authentification

La récupération du contexte de sécurité peut se faire de plusieurs manières :

- Dans n'importe quel objet :

```
Authentication authentication = SecurityContextHolder.getContext().getAuthentication();
```

- Dans un contrôleur Spring, via un paramètre de type `Principal` (`javax.security.Principal`) ou `Authentication` (`org.springframework.security.core.Authentication`) dans une méthode annotée par `@RequestMapping`

```
@RequestMapping(path="authentication", method=RequestMethod.POST)
public Map<String, String> onSuccessAuthentication(Authentication authentication) {
    String username = authentication.getName();
    Collection<GrantedAuthority> authorities = authentication.getAuthorities();
    // fabrication d'un Javscript Web Token signé.
    // soit jwt ce token
    return Map.of("jwt", jwt);
}
```

Autorisations

Règles d'accès aux routes

La sécurisation des routes d'après des *patterns* se fait dans la méthode `configure` qui a servi à paramétrer l'authentification :

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
    /*configuration de l'authentification*/
    .and()
        .authorizeRequests()
            .antMatchers("/books/**").hasRole("BOOK_ADMIN") ①
            .antMatchers("/publishers/**").access("hasRole('PUBLISHER_ADMIN')") ②
            .anyRequest().authenticated();
}
```

① `BOOK_ADMIN` est le nom d'un rôle

② `hasRole('PUBLISHER_ADMIN')` est une expression. Nous pourrions écrire `hasRole('PUBLISHER_ADMIN') and hasRole('ANOTHER_ROLE')`

L'ordre importe, `anyRequest().authenticated()` s'appliquant à toutes les routes pour lesquelles aucune règle n'a été précisée, il est important que cette instruction vienne en dernier.

La réponse émise par le fitre si les conditions ne sont pas respectée (requête anonyme ou droits insuffisants) dépend de la configuration établie précédemment :

- si la requête est anonyme : formulaire de login pour une IHM, réponse HTTP 401 pour une API REST.
- si les droits sont insuffisants : erreur 403 (auquel une vue peut éventuellement être associée si notre application web est une IHM).

Sécurisation des méthodes

La sécurisation des méthodes vise à demander à Spring de n'autoriser l'invocation de certaines méthodes que si l'utilisateur a les permissions requises.

Cela consiste à poser des annotations sur les méthodes à sécuriser, ainsi Spring pourra inscrire dans le contexte applicatif un *proxy* devant les instances des classes concernées (celles qui contiennent au moins une méthode à sécuriser).

A chaque invocation des méthodes annotées, le *proxy* vérifiera que l'utilisateur est autorisé à invoquer cette méthode.

3 types d'annotations sont supportées :

- L'annotation `@Secured` (annotation Spring)
- Les annotations `@PreAuthorize` et `@PostAuthorize` (annotations Spring, supporte les expressions SPEL)
- L'annotation `@RolesAllowed` (JSR 250)

L'activation des interceptions sur les méthodes annotées se fait en annotant la classe de configuration de la sécurité par `@EnableGlobalMethodSecurity`

Des attributs permettent de choisir le type d'annotations qui décore nos méthodes :

- `securedEnabled` (pour le support des annotations `@Secured`)
- `jsr250enabled` (pour le support de l'annotation `@RolesAllowed`)
- `prePostEnabled` (pour le support des annotations `@PreAuthorize` et `@PostAuthorize`)

Sécurisation des vues

La sécurisation des vues consiste à conditionner l'affichage de certains éléments de la vue en fonction des droits de l'utilisateur.

Si les vues utilisent la technologies JSP nous pouvons ajouter une dépendance Maven :

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-taglibs</artifactId>
</dependency>
```

Ensuite :

```
<%@ taglib uri="http://www.springframework.org/security/tags" prefix="sec" %>

<sec:authorize access="hasRole('ADMIN')">
    <a href="admin">Admin</a>
</sec:authorize>
```

Notons que cette *taglib* permet aussi de disposer du contexte d'authentification de l'utilisateur en cours.

Exemple : `<sec:authentication property="name"/>`

Tests

Pour qu'un contexte de sécurité soient créé au moment de l'exécution des tests unitaires, il suffit d'annoter la classe de test (ou les méthodes) par l'annotation `@WithMockUser`

Celle-ci est disponible dans la librairie spring-security-test :

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

Elle sera découverte par le Spring (cf. `@Runwith(SpringRunner.class)` avec Junit 4 ou `@ExtendWith(SpringExtension.class)` avec Junit 5), qui préparera un contexte de sécurité pour la/les méthodes de test concernées.

Spring Batch

- Principes
- Définition d'un job
- Définition d'un step
- Execution d'un job
- Tests
- Synthèse

```
<dependency>
    <groupId>org.springframework.batch</groupId>
    <artifactId>spring-batch-core</artifactId>
    <version>4.1.2.RELEASE</version>
</dependency>
```

ou, si nous utilisons Spring boot :

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-batch</artifactId> ①
</dependency>
```

① dépendance indirecte : `spring-batch-core`

Principes

Nous pouvons définir un *batch* comme un traitement de données non interactif, démarré suivant une planification ou suite à un évenement.

Exemple :

- traitement d'optimisation d'images en vue d'une diffusion sur le web.
- transcoding de video en vue d'une diffusion sur le web.
- paie mensuelle des employés.
- génération de relevés bancaires.
- etc...

Un *job* peut être considéré comme une suite d'étapes (*steps*).

Définition d'un Job

Les types qui représentent ces concepts dans Spring Batch sont :

- `org.springframework.batch.core.Job` : une suite d'étapes.
- `org.springframework.batch.core.Step` : une étape. Il peut s'agir :
  - d'une séquence de type *extract*, *transform*, *load*. Spring batch nomme cela *chunk-oriented step*
  - d'un traitement simple (exemple : appeler une procédure stockée ou un web service). Spring batch nomme cela *tasklet step*

Exécuter un *job* conduit à

- obtenir une instance de `JobExecution`. Celle-ci à un statut (voir la propriété `batchStatus`). Les valeurs possibles :
  - `STARTING`
  - `STARTED`
  - `STOPPING`
  - `STOPPED`
  - `COMPLETED`
  - `FAILED`
  - `ABANDONED`
  - `UNKNOWN`
- exécuter chacun des *steps*. Chaque exécution de *step* conduit à obtenir une instance `StepExecution`. Celle-ci à un statut (voir la propriété `exitStatus`). Les statuts possibles :
  - `UNKNOWN`
  - `EXECUTING`
  - `COMPLETED`
  - `NOOP`
  - `FAILED`
  - `STOPPED`

Un Job se définit habituellement dans une classe de configuration annotée par @EnableBatchProcessing :

```
@Configuration
@EnableBatchProcessing ①
public class PayrollJobConfiguration {

    @Bean
    public Step prepareWireTransfers(){
        // code
    }

    @Bean
    public Job payrollJob(JobBuilderFactory jobBuilderFactory) {
        JobBuilder builder = jobBuilderFactory.get("payrollJob");
        return builder.start(prepareWireTransfers()).build();
    }

    public static void main(String[] args){
        try(var ctx = new AnnotationConfigApplicationContext(PayrollJobConfiguration.class)){
            JobLauncher launcher = ctx.getBean(JobLauncher.class);
            Job job = ctx.getBean(Job.class); ②
            jobLauncher.run(job, new JobParameters());
        }
    }
}
```

① Inscrit dans le contexte applicatif certains *beans*, notamment : JobBuilderFactory, StepBuilderFactory, JobLauncher, JobRepository

② le *bean* de type Job est celui retourné par la méthode productrice payrollJob

L'exécution du *job* sera considérée comme réussie (COMPLETED) si le *step* prepareWireTransfers a été exécuté avec succès (COMPLETED).

Un job peut être une suite de *steps* (*sequential flow*). Exemple :

```
@Configuration
@EnableBatchProcessing
public class PayrollJobConfiguration {

    @Bean
    public Step prepareWireTransfers(){
        // code
    }

    @Bean
    public Step sendWireTransferToBank(){
        // code
    }

    @Bean
    public Job payrollJob(JobBuilderFactory jobBuilderFactory) {
        JobBuilder builder = jobBuilderFactory.get("payrollJob");
        return builder
            .start(prepareWireTransfers())
            .next(sendWireTransferToBank()) ①
            .build();
    }
}
```

① ne sera appelé que si le *step* `prepareWireTransfers` a été exécuté avec succès (`COMPLETED`)

L'exécution du *job* sera considérée comme réussie (`COMPLETED`) si les *steps* `prepareWireTransfers` et `sendWireTransferToBank` ont été exécutés avec succès (`COMPLETED`).

Les branchements conditionnels (*conditional flows*) sont supportés, grâce à la méthode `on` :

Ainsi, nous pourrions avoir la suite de traitement décrite précedemment mais avec un troisième *step* (disons `sendWireTransferToBankFallback`) qui serait appelé en cas d'échec du *step* `sendWireTransferToBank`. L'idée serait de soumettre les ordres de virement à une autre banque en cas d'echec du *step* `sendWireTransferToBank`

L'écriture du *job* serait alors :

```
@Configuration
@EnableBatchProcessing
public class PayrollJobConfiguration {

    // méthode productrice prepareWireTransfers()
    // méthode productrice sendWireTransferToBank()

    @Bean
    public Step sendWireTransferToBankFallback(){
        // code
    }

    @Bean
    public Job payrollJob(JobBuilderFactory jobBuilderFactory) {
        JobBuilder builder = jobBuilderFactory.get("payrollJob");
        return builder
            .start(prepareWireTransfers())
            .next(sendWireTransferToBank()).on("FAILED").to(sendWireTransferToBankFallback())
            .end()
            .build();
    }
}
```

Dans l'exemple précédent c'est l'issue du Step sendWireTransferToBank qui conduira à déclencher ou non le *step* sendWireTransferToBankFallback.

L'exécution du *job* sera considérée comme réussie (COMPLETED) si les *steps* prepareWireTransfers et (sendWireTransferToBank ou sendWireTransferToBankFallback) sont réussis (COMPLETED).

Notons qu'il existe deux méthodes next :

```
public JobFlowBuilder next(Step step) {
    // code
}

public JobFlowBuilder next(JobExecutionDecider decider) {
    // code
}
```

La seconde méthode permet de positionner dans le *workflow* un JobExecutionDecider.

Il s'agit d'une interface à implémenter :

```
public interface JobExecutionDecider {

    /**
     * Strategy for branching an execution based on the state of an ongoing
     * {@link JobExecution}. The return value will be used as a status to
     * determine the next step in the job.
     *
     * @param jobExecution a job execution
     * @param stepExecution the latest step execution (may be {@code null})
     * @return the exit status code
     */
    FlowExecutionStatus decide(JobExecution jobExecution, @Nullable StepExecution stepExecution);
}
```

L'avantage est de pouvoir retourner dynamiquement un statut qui orientera la suite du *workflow*.

Les différents `FlowExecutionStatus` possible : `COMPLETED`, `STOPPED`, `FAILED`, `UNKNOWN`

Définition d'un Step

Les *chunk oriented steps*

Il s'agit de Step mettant en relation un *reader*, un *processor* (facultatif) et un *writer*.

Les interfaces qui définissent ces composants sont :

- `org.springframework.batch.item.ItemReader<T>` : le composant responsable de l'extraction.
- `org.springframework.batch.item.ItemProcessor<I, O>` : le composant responsables du traitement métier.
- `org.springframework.batch.item.ItemWriter<T>` : le composant responsable de l'écriture.

Le code source des interfaces `ItemReader<T>`, `ItemProcessor<I, O>`, `ItemWriter<T>` :

```
public interface ItemReader<T>{
    @Nullable
    T read() throws Exception, UnexpectedInputException, ParseException, NonTransientResourceException;
}

public interface ItemProcessor<I,O> { ① ②
    O process(I item) throws Exception;
}

public interface ItemWriter<T>{
    void write(List<? extends T> items) throws Exception;
}
```

① `I` représente ce que retourn l'`ItemReader`

② `O` représente ce qui sera passé en paramètre à l'`ItemWriter`

Nous remarquons que :

- le *reader* retourne un *item*
- le *processor* reçoit un *item* et retourne un *item*
- le *writer* reçoit plusieurs *items*

Cela introduit la notion de *commit interval*. Il s'agit d'un nombre (disons n), et c'est seulement lorsque n *items* ont été lus par le *reader* et traités par le *processor* qu'ils sont transmis au *writer*. Les n *items* sont nommés *chunk*.

Nous pouvons imaginer que l'implémentation de ce mécanisme par Spring Batch ressemble à :

```
List<Object> itemsToWrite = new ArrayList<>();
for(int i = 0;i < commitInterval; i++){ ①
    Object item = reader.read();
    if(item == null){ ②
        break;
    }
    Object processedItem = processor.process(item);
    processedItems.add(processedItem);
}
writer.write(processedItems); ③
```

① `commitInterval` est un nombre paramétrable.

② il n'y a plus d'élément à traiter.

③ Eventuellement dans une transaction si la destination le supporte.

le cycle de vie du *reader*, du *processor* et du *writer* est celui du *step* auquel ils contribuent.

La mise en relation des composants (ItemReader, ItemWriter, Step et enfin Job) peut se faire dans la définition du Step :

```
@Configuration @EnableBatchProcessing
public class PayrollJobConfiguration {

    // méthode productrice reader() retournant un ItemReader<Employee>()
    // méthode productrice processor() retournant un ItemProcessor<Employee, WireTransfer>()
    // méthode productrice writer() retournant un ItemWriter<WireTransfer>()

    @Autowired ①
    private StepBuilderFactory stepBuilderFactory;

    @Bean
    public Step prepareWireTransfers() {
        StepBuilder builder = stepBuilderFactory.get("prepareWireTransfers");
        return builder.<Employee, WireTransfer> chunk(10) ②
            .reader(reader()).processor(processor()).writer(writer())
            .build();
    }

    @Bean
    public Job payrollJob(JobBuilderFactory jobBuilderFactory) {
        JobBuilder builder = jobBuilderFactory.get("payrollJob");
        return builder.start(prepareWireTransfers()).build();
    }
}
```

① Car un *bean* StepBuilderFactory est inscrit dans le contexte applicatif compte tenu de l'annnotation @EnableBatchProcessing

② 10 représente le *commit interval* et donc la taille du *chunk*. Il est aussi possible de passer une CompletionPolicy plutôt qu'un nombre si nous voulons que la démarcation des *chunks* suive une règle métier.

Déclaration d'un ItemReader

Soit la table Employee et une classe du même nom :

```
create table Employee(
    id serial primary key,
    firstname varchar(50),
    lastname varchar(50),
    iban varchar(50),
    bic varchar(20)
)
```

```
public class Employee{

    private int id;
    private String firstname,lastname,iban,bic;

    Employee(){}

    public Employee(int id, String firstname, String lastname, String iban, String bic){
        this.id = id;
        this.firstname = firstname;
        this.lastname = lastname;
        this.iban = iban;
        this.bic = bic;
    }
    // getters & setters
}
```

Un ItemReader<T> expose une méthode read<T>() qui agit comme un curseur.

Dès lors une classe EmployeeItemReader pourrait être écrite ainsi :

```
public class EmployeeItemReader implements ItemReader<Employee>{

    private final DataSource ds;
    private Connection connection;
    private ResultSet resultSet;
    private PreparedStatement ps;

    public EmployeeItemReader(DataSource ds){this.ds = ds;}

    public Employee read() throws Exception, UnexpectedInputException, ParseException, NonTransientResourceException {
        if(this.connection == null){
            this.connection = ds.getConnection();
            String query = "select id, firstname, lastname, iban, bic from Employee";
            this.preparedStatement = this.connection.prepareStatement(query);
        }
        if(this.resultSet == null){
            this.resultSet = this.preparedStatement.executeQuery();
        }
        if(this.resultSet.next()){
            return new Employee(rs.getInt(1), rs.getString(2), rs.getString(3), rs.getString(4));
        } else {
            this.resultSet.close();this.preparedStatement.close();this.connection.close();
            return null;
        }
    }
}
```

Et la classe de configuration du *job* deviendrait :

```
@Configuration
@EnableBatchProcessing
public class PayrollJobConfiguration {

    @Bean
    @StepScope ①
    public ItemReader<Employee> reader(){
        DataSource ds = new DriverManagerDataSource("...");
        return new EmployeeItemReader(ds);
    }
    // méthode productrice processor() retournant un ItemProcessor<Employee, WireTransfer>()
    // méthode productrice writer() retournant un ItemWriter<WireTransfer>()

    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Bean
    public Step prepareWireTransfers() {
        StepBuilder builder = stepBuilderFactory.get("prepareWireTransfers");
        return builder.<Employee, WireTransfer> chunk(10)
            .reader(reader()).processor(processor()).writer(writer())
            .build();
    }

    // méthode productrice payrollJob retournant un Job
}
```

① pour aligner le cycle de vie du *reader* avec celui du *step* auquel il contribue.

Toutefois nous constatons que le code de la classe `EmployeeItemReader` est plutôt technique :

- ouverture de la connexion
- pour chaque ligne du `ResultSet` : rapprochement entre une propriété de la classe `Employee` et une valeur lue dans une colonne.
- fermeture de la connexion

Il en serait de même si les lignes provenaient d'un fichier :

- ouverture du fichier.
- fermeture du fichier.
- pour chaque ligne : rapprochement entre une propriété de la classe `Employee` et une valeur lue dans une colonne.
- gestion des exceptions.

C'est pour cela que Spring propose des `ItemReader<T>` configurables. Quelques exemples :

- `FlatFileItemReader` si les *items* doivent être reconstruits d'après les lignes d'un fichier.
- `JdbcCursorItemReader` si les *items* doivent être reconstruits d'après les lignes retournées par une une requête en base de données.
- `StaxEventItemReader` si les *items* doivent être reconstruits d'après les éléments d'un fichier XML.
- `JmsItemReader` si les *items* doivent être lus dans une destination JMS
- `JsonItemReader` si les *items* doivent être lus dans un fichier json

Liste exhaustive : https://docs.spring.io/spring-batch/4.1.x/reference/html/appendix.html#itemReadersAppendix

Ces classes ont en commun d'implémenter ItemStreamReader<T>.

```
public interface ItemStreamReader<T> extends ItemStream, ItemReader<T> {
}
```

ItemStream :

```
public interface ItemStream {

    void open(ExecutionContext executionContext) throws ItemStreamException;

    void update(ExecutionContext executionContext) throws ItemStreamException;

    void close() throws ItemStreamException;
}
```

Nous comprenons donc que les mécanismes d'ouverture et fermeture de la connexion à la ressourcce (ouverture et fermeture d'une connexion jdbc par exemple) sont implémentés.

Chacun de ces ItemReader<T> dispose d'un *builder*.

Ainsi nous pouvons écrire :

```
JdbcCursorItemReaderBuilder<Employee> builder = new JdbcCursorItemReaderBuilder<>();

ItemReader<Employee> = builder
        .name("employeesItemReader")
        .dataSource(new DriverManagerDataSource("..."))
        .sql("select id, firstname, lastname, iban, bic from Employee")
        .beanRowMapper(Employee.class) ①
        .build();
```

① car les noms de colonnes correspondent à des noms de propriétés. Cela suppose toutefois un constructeur sans argument (pas forcément public) car Spring va invoquer les *setters* de l'objet pour valoriser les champs.

La classe de configuration devient alors :

```
@Configuration @EnableBatchProcessing
public class PayrollJobConfiguration {

    @Bean @StepScope
    public ItemReader<Employee> reader(){
        return new JdbcCursorItemReaderBuilder<Employee>()
            .name("employeesItemReader")
            .dataSource(new DriverManagerDataSource("..."))
            .sql("select id, firstname, lastname, iban, bic from Employee")
            .beanRowMapper(Employee.class)
            .build();
    }
    // méthode productrice processor() retournant un ItemProcessor<Employee, WireTransfer>()
    // méthode productrice writer() retournant un ItemWriter<WireTransfer>()

    @Autowired
    private StepBuilderFactory stepBuilderFactory;

    @Bean
    public Step prepareWireTransfers() {
        StepBuilder builder = stepBuilderFactory.get("prepareWireTransfers");
        return builder.<Employee, WireTransfer> chunk(10)
            .reader(reader()).processor(processor()).writer(writer())
            .build();
    }

    // méthode productrice payrollJob retournant un Job
}
```

Attention, si nous utilisons `@StepScope`, Spring inscrira un *proxy* vers notre *reader* (idem pour le *processor* et le *writer*) afin de contrôler le cycle de vie de l'objet (créer un nouveau *bean* à chaque exécution du *step*, l'enlever du contexte à la fin du *step*).

Par défaut, c'est le type de retour de la méthode productrice qui sera utilisé par Spring pour établir:

- un *proxy* par association si le type de retour est une interface.
- un *proxy* par héritage si le type de retour est une classe concrète.

Dans le premier cas, le *proxy* n'exposera que les méthodes de l'interface :

- `read` si le type de retour est `ItemReader<T>`
- `open`, `read`, `update` et `close` si si le type de retour est `ItemStreamReader<T>`

Hors si le *reader* est un `ItemStreamReader<T>` il est essentiel que le *proxy* en soit aussi un, sinon seule la méthode `read` sera exposée, hors une source de donnée ne peut pas être lue si elle n'a pas été ouverte avant. Une exception de type `ReaderNotOpenException` sera alors levée (message : *Reader must be open before it can be read*).

Deux solutions pour que Spring voit le *proxy* comme un `ItemStreamReader<T>`

- choisir `ItemStreamReader<T>` comme type de retour de méthode.
- annoter la méthode par `@Scope(proxyMode=ScopedProxyMode.TARGET_CLASS)` en plus de `@§tepScope` afin de forcer l'application d'un *proxy* par héritage.

Le problème ne se pose pas si le type de retour de la méthode est l'implémentation concrète de `ItemReader<T>`.

La déclaration du *bean* devient :

```
@Bean
@StepScope @Scope(proxyMode=ScopedProxyMode.TARGET_CLASS)
public ItemReader<Employee> reader(){
    return new JdbcCursorItemReaderBuilder<Employee>()
        .name("employeesItemReader")
        .dataSource(new DriverManagerDataSource("..."))
        .sql("select id, firstname, lastname, iban, bic from Employee")
        .beanRowMapper(Employee.class)
        .build();
}
```

ou

```
@Bean
@StepScope
public ItemStreamReader<Employee> reader(){
    return new JdbcCursorItemReaderBuilder<Employee>()
        .name("employeesItemReader")
        .dataSource(new DriverManagerDataSource("..."))
        .sql("select id, firstname, lastname, iban, bic from Employee")
        .beanRowMapper(Employee.class)
        .build();
}
```

Déclaration d'un `ItemProcessor`

C'est dans l'`ItemProcessor` que se trouve le code métier.

Rappel de l'interface :

```
public interface ItemProcessor<I,O> {
    O process(I item) throws Exception;
}
```

Exemple de traitements :

- créer un fichier video mp4 HD et un fichier vidéo mp4 SD à partir d'un fichier source (transmis par l'`ItemReader`).
- créer, pour un employé, un ordre de virement pour la paie du mois en cours.
- créer, pour chaque client d'une banque, un relevé bancaire à partir des opérations du mois précédent.

Dans chaque cas :

- le type paramétré `I` correspond à ce que le *reader* a retourné.
- le type paramétré `O` correspond à ce que le *processor* retourne (et qui sera donc transmis au *writer* à chaque *commit interval*).

Si nous reprenons l'exemple de création d'ordres de virement pour la paie des employés, nous pouvons envisager une classe `WireTransfer` :

```
public class WireTransfer {

    private final int employeeId;
    private final String iban,bic;
    private final double amount;

    public WireTransfer(int employeeId, String iban, String bic, double amount){
        this.employeeId = employeeId;
        this.iban = iban;
        this.bic = bic;
        this.amount = amount;
    }

    // getters
}
```

et une classe `SalaryItemProcessor` :

```
public class SalaryItemProcessor implements ItemProcessor<Employee, WireTransfer> {

    public WireTransfer process(Employee employee){
        // code
    }
}
```

Mais la démarcation des responsabilité entre le *reader* et le *processor* pose question.

En effet, une règle métier précisera sans doute que le montant versé à un salarié pour un mois donné dépend

- de son temps de présence dans l'entreprise pendant cette période. A-t-il quitté l'entreprise en cours de mois ? Est il arrivé dans l'entreprise en cours de mois ?
- des arrêts maladie déclarés pendant cette période.
- des congés sans solde pris par cet employé pendant cette période.

Les informations sur la date d'arrivée et la date de départ de l'entreprise pourraient légitimement figurer dans l'objet fournir par le *reader* (une instance de `Employee` dans notre cas).

Mais est-ce au *reader* de fournir une instance de la classe `Employee` ayant en plus des propriétés `List<SeakLeave>` et `List<Vacation>` ?

Si oui : le *processor* disposera de tout ce dont il a besoin pour retourner un ordre de virement.

Si non : le *processor* doit accéder aux données connexes : arrêts maladie et congés. Dans ce cas il est judicieux de mettre en cache ces éléments afin de ne pas solliciter la base de données à chaque appel à la méthode `process`.

Si l'Employee reçu en argument contient les informations sur les arrêts maladie et les congés :

```
public class SalaryItemProcessor implements ItemProcessor<Employee, WireTransfer> {

    public WireTransfer process(Employee employee){
        // code
    }
}
```

Sinon

```
public class SalaryItemProcessor implements ItemProcessor<Employee, WireTransfer> {

    private Map<Integer, List<SeakLeveave>> seakLeavesByEmployeeId;
    private Map<Integer, List<Vacation>> vacationsByEmployeeId;

    @PostConstruct
    public void initRelatedData(){
        // valorisation de this.seakLeavesByEmployeeId
        // valorisation de this.vacationsByEmployeeId
    }

    public WireTransfer process(Employee employee){
        List<SeakLeveave> seakLeaves = this.seakLeavesByEmployeeId.get(employee.getId());
        List<Vacation> vacations = this.vacationsByEmployeeId.get(employee.getId());
        // code
    }
}
```

Enfin, l'inscription dans le contexte applicatif :

```
@Configuration @EnableBatchProcessing
public class PayrollJobConfiguration {

    // méthode productrice reader() retournant un ItemReader<Employee>()

    @Bean @StepScope
    public ItemProcessor<Employee, WireTransfer> processor(){
        return new SalaryItemProcessor();
    }

    // méthode productrice writer() retournant un ItemWriter<WireTransfer>()

    private @Autowired StepBuilderFactory stepBuilderFactory;

    @Bean
    public Step prepareWireTransfers() {
        StepBuilder builder = stepBuilderFactory.get("wireTransfers");
        return builder.<Employee, EmployeePay> chunk(10) ②
            .reader(reader()).processor(processor()).writer(writer())
            .build();
    }

    // méthode productrice payrollJob retournant un Job
}
```

Déclaration d'un `ItemWriter`

L'`ItemWriter` est celui qui va écrire, à interval régulier (cf. le *commit interval*) les *items* traités par le *processor*.

Rappel de l'interface :

```
public interface ItemWriter<T> {
    void write(List<? extends T> items) throws Exception;
}
```

Il s'agit donc d'écrire, à chaque *commit interval*, les *items* du *chunk* quelque part (fichier, base de données, etc...).

Dès lors un WireTransferItemWriter pourra s'écrire ainsi :

```
public class WireTransferItemWriter implements ItemWriter<WireTransfer> {

    private final Path path;

    public WireTransferItemWriter(Path path){
        this.path = path;
    }

    public void write(List<WireTransfer> items) throws Exception{
        List<String> allLines = items.stream()
            .map(x -> {
                int empId = x.getEmployeeId();
                String iban = x.getIban();
                String bic = x.getBic();
                double amount = x.getAmount();
                return String.format("%s;%s;%s;%s", empId, iban, bic, amount);
            })
            .collec(Collectors.toList());
        Files.write(this.path, allLines, StandardOpenOption.CREATE, StandardOpenOption.APPEND);
    }
}
```

Enfin, l'inscription dans le contexte applicatif :

```
@Configuration @EnableBatchProcessing
public class PayrollJobConfiguration {

    // méthode productrice reader() retournant un ItemReader<Employee>()
    // méthode productrice processor() retournant un ItemProcessor<Employee, WireTransfer>()

    @Bean @StepScope
    public ItemWriter<WireTransfer> writer(){
        return new WireTransferItemWriter(Paths.get("/path/to/file"));
    }

    private @Autowired StepBuilderFactory stepBuilderFactory;

    @Bean
    public Step prepareWireTransfers() {
        StepBuilder builder = stepBuilderFactory.get("prepareWireTransfers");
        return builder.<Employee, WireTransfer> chunk(10)
            .reader(reader()).processor(processor()).writer(writer())
            .build();
    }

    // méthode productrice payrollJob retournant un Job
}
```

Là encore, Spring propose des `ItemWriter<T>` configurables. Quelques exemples :

- `FlatFileItemWriter` si les `Item` doivent être écrits dans un fichier (1 item → 1 ligne)
- `JdbcItemWriter` si les `Item` doivent être écrits dans une base de données (1 item → 1 ligne)
- `StaxEventItemWriter` si les `Item` doivent être écrits dans un fichier XML (1 item → 1 évement xml)
- `JmsItemWriter` si les `Item` doivent être écrits dans une destination JMS (1 item → 1 message JMS).

Liste exhaustive : https://docs.spring.io/spring-batch/4.1.x/reference/html/appendix.html#itemWritersAppendix

Chacun de ces ItemWriter<T> dispose d'un *builder*.

Ainsi nous écrire :

```
FlatFileItemWriterBuilder<WireTransfer> builder = new FlatFileItemWriterBuilder<>();

ItemWriter<WireTransfer> writer = builder
        .name("wireTransfersWriter")
        .resource(new FileSystemResource("/path/to/file"))
        .delimited().delimiter("|")
        .names(new String[]{"employeeId", "firstname", "lastname", "iban", "bic"}) ①
        .build();
```

① pour indiquer que les valeurs de chque colonne (pour une ligne) donnée provient des propriétés de l'*item* (ici une instance de WireTransfer).

La classe de configuration devient alors :

```
@Configuration @EnableBatchProcessing
public class PayrollJobConfiguration {

    // méthode productrice reader() retournant un ItemReader<Employee>()
    // méthode productrice processor() retournant un ItemProcessor<Employee, WireTransfer>()

    @Bean @StepScope @Scope(proxyMode=ScopedProxyMode.TARGET_CLASS)
    public ItemWriter<WireTransfer> writer(){
        return new FlatFileItemWriterBuilder<WireTransfer>()
            .name("wireTransfersWriter")
            .resource(new FileSystemResource("/path/to/file"))
            .delimited().delimiter("|")
            .names(new String[]{"employeeId", "firstname", "lastname", "iban", "bic"})
            .build();
    }

    private @Autowired StepBuilderFactory stepBuilderFactory;

    @Bean
    public Step prepareWireTransfers() {
        StepBuilder builder = stepBuilderFactory.get("prepareWireTransfers");
        return builder.<Employee, WireTransfer> chunk(10)
            .reader(reader()).processor(processor()).writer(writer())
            .build();
    }

    // méthode productrice payrollJob retournant un Job
}
```

Cette suite d'écritures a avantage à s'exécuter dans une transaction, si la ressource accédée le permet (base de données par exemple).

Cela suppose la présence d'un `PlatformTransactionManager` dans le contexte applicatif.

Extrait de la documentation de l'annotation `@EnableBatchProcessing` :

The transaction manager provided by this annotation will be of type:

- `org.springframework.batch.support.transaction.ResourcelessTransactionManager` *if no* `javax.sql.DataSource` *is provided within the context*
- `org.springframework.jdbc.datasource.DataSourceTransactionManager` *if a* `javax.sql.DataSource` *is provided within the context*

Dans notre exemple c'est donc l'implémentation `ResourcelessTransactionManager` qui sera utilisée. Appliquée à l'ecriture d'un fichier, cela signifie que toutes les lignes du *chunk* sont écrits en une seule fois dans le fichier.

Les *tasklet steps*

Une tasklet est une classe qui implémente l'interface `Tasklet`

```
public interface Tasklet {
    @Nullable
    RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) throws Exception;
}
```

implémenter la méthode `execute` revient à réaliser un traitement et retourner un statut correspondant à l'issue du traitement.

Exemple de Tasklet :

```
public class TransferRequestSender implements Tasklet{

    public RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) throws Exception {
        // code
        return RepeatStatus.FINISHED;
    }
}
```

A propos des paramètres dont la méthode dispose :

StepContribution permet de renseigner des statistiques sur le traitement réalisé. Les méthodes disponibles :

- incrementFilterCount
- incrementReadCount
- incrementReadSkipCount
- incrementWriteSkipCount
- incrementWriteCount

ChunkContext permet d'obtenir les informations liées au contexte d'exécution du *step* (*job parameters* par exemple).

La déclaration d'un *tasklet step* se fait à nouveau avec le `StepBuilderFactory` :

```
@Configuration
@EnableBatchProcessing
public class PayrollJobConfiguration{

    // méthodes productrice ds()
    // injection du bean stepBuilderFactory
    // méthodes productrices reader(), processor(), writer() et prepareWireTransfers()

    @Bean
    public Tasklet transferRequestSender(){
        return new TransferRequestSender();
    }

    @Bean
    public Step sendWireTransferToBank(){
        StepBuilder builder = stepBuilderFactory.get("sendRequestsToBank");
        return builder.get("sendRequestsToBank")
            .tasklet(transferRequestSender()) ①
            .build();
    }
}
```

① C'est ici la méthode `tasklet` que nous appelons, et non pas la méthode `chunk` (qui invite à renseigner un *reader*, un *writer* et un *processor*).

Dès lors le *job* peut désormais comporter deux étapes :

```
@Configuration
@EnableBatchProcessing
public class PayrollJobConfiguration{

    // méthodes productrice ds()
    // injection du bean stepBuilderFactory
    // méthodes productrices reader(), processor(), writer() et prepareWireTransfers()
    // méthodes productrices transferRequestSender() et sendWireTransferToBank()


    @Bean
    public Job payrollJob(JobBuilderFactory jobBuilderFactory) {
        JobBuilder builder = jobBuilderFactory.get("payrollJob");
        return builder
            .start(prepareWireTransfers())
            .next(sendWireTransferToBank())
            .build();
    }
}
```

Les stratégies de réessai

La question du réessai se pose lorsqu'une exception survient pendant l'exécution d'un *step*.

Toutefois les réessais se concoivent différemment selon que le *step* soit un *chunk-orient step* ou un *tasklet step*.

chunk oriented step

Pour un *chunk oriented step*, le problème (levée d'une exception) se posera au niveau de l'*item* (lecture par le *reader* ou traitement par le *processor*).

Si une exception est levée, il faut pouvoir décider si

- elle doit donner lieu à un ou plusieurs réessais.
  - Si oui :
    - quel est le nombre maximum de réessai ? Au delà de ce seuil le *step* sera considéré comme en échec (`FAILED`).
    - faut il attendre un certain délai avant d'invoquer à nouveau la méthode qui a levé l'exception.
  - Si non :
- l'élément en échec peut être ignoré (*skipped*).
  - Si oui : combien de *skip* sont tolérés avant que le *step* ne soit considéré comme en échec (`FAILED`).

En règle générale :

- le *skip* concerne la donnée (l'`item`)
- le *retry* concerne la ressource à laquelle nous accédons. Si elle n'est pas accessible à un instant t cela peut avoir du sens de réessayer plus tard.

C'est à niveau de la définition du *step* que nous précisons la stratégie de réessai :

```
private @Autowired StepBuilderFactory stepBuilderFactory;

@Bean
public Step prepareWireTransfers() {
    StepBuilder builder = stepBuilderFactory.get("prepareWireTransfers");
    return builder.<Employee, WireTransfer> chunk(10)
        .reader(reader()).processor(processor()).writer(writer())
        .faultTolerant()
        .skip(NumberFormatException.class) ①
        .skipLimit(3) ②
        .retry(TimeoutException.class) ③
        .retryLimit(3) ④
        .backOffPolicy(new FixedBackOffPolicy() {{setBackOffPeriod(5000);}}) ⑤
        .build();
}
```

① si le *reader* ou le *processor* lève une `NumberFormatException`, l'*item* est ignoré.

② si plus de 3 éléments sont ignorés, le *step* est mis en échec (`exitStatus` : `FAILED`).

③ si le *reader*, le *writer* ou le *processor* lève une `TimeoutException`, la méthode est réinvoquée.

④ au delà de 3 réessais, le *step* est mis en échec.

⑤ Spring doit attendre 5 secondes avant de procéder à un réessai.

tasklet step

Pour un *tasklet step*, le problème (levée d'une exception) se posera au niveau de la méthode `execute`.

Dès lors c'est une annotation `@Retry` qui permettra d'indiquer la stratégie de réessai.

Nous retrouvons dans cette annotation les même paramétrage que pour les *chunk-oriented steps*, à l'exception du *skip* qui n'a pas lieu d'être puisque Spring n'a pas de visibilité sur les *items* mais seulement sur la méthode `execute`

```
public class TransferRequestSender implements Tasklet{

    @Retryable(
        include = TimeoutException.class, ①
        maxAttempts= 3, ②
        backoff= @Backoff(delay=5000) ③
    )
    public RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) throws Exception {
        // code
        return RepeatStatus.FINISHED;
    }
}
```

① si la méthode `execute` lève une `TimeoutException`, la méthode est réinvoquée.

② au delà de 3 réessais, le *step* est mis en échec (`exitStatus` : `FAILED`).

③ Spring doit attendre 5 secondes avant de procéder à un réessai.

Exécution d'un Job

Le lancement d'un *job* répond généralement à une planification.

Exemple : lancement du *job payrollJob* 3 jours ouvrés avant la fin du mois.

Le lancement du job peut être réalisé

- en ligne de commande, avec la commande `java` (il faut alors une méthode `main`)
- via une méthode annotée par `@Scheduled` dans une application déjà déployée.
- depuis une colonne d'administration, dans ce cas c'est une méthode annotée par `@RequestMapping` qui sera reponsable du lancement du *job*.

Dans tous les cas :

- c'est le *bean* de type `JobLauncher` (inscrit automatiquement dans le contexte compte tenu de l'annotation `@EnableBatchProcessing`) qui va nous permettre de lancer le *job*.
- le passage des paramètres se fait via une instance de `JobParameters`.

Exemple :

```
LocalDate now = LocalDate.now();
JobParameters params = new JobParametersBuilder()
    .addLong("month", now.getLong(ChronoField.MONTH_OF_YEAR))
    .addLong("year", now.getLong(ChronoField.YEAR))
    .toJobParameters();
```

Depuis une méthode main

```
@Configuration @EnableBatchProcessing
public class PayrollJobConfiguration{

    // méthodes productrices reader(), processor(), writer() et prepareWireTransfers()
    // méthodes productrices transferRequestSender() et sendWireTransferToBank()

    @Bean(name="payrollJob")
    public Job payrollJob(JobBuilderFactory jobBuilderFactory) {
        JobBuilder builder = jobBuilderFactory.get("payrollJob");
        return builder.start(wireTransfers()).then(sendWireTransferToBank()).build();
    }

    public static void main(String[] args){
        try(var ctx = new AnnotationConfigApplicationContext(PayrollJobConfiguration.class)) {
            Job job = ctx.getBean("payrollJob", Job.class); ①
            JobLauncher jobLauncher = ctx.getBean(JobLauncher.class);

            LocalDate now = LocalDate.now();
            JobParameters params = new JobParametersBuilder()
                .addLong("month", now.getLong(ChronoField.MONTH_OF_YEAR))
                .addLong("year", now.getLong(ChronoField.YEAR))
                .toJobParameters();
            JobExecution execution = jobLauncher.run(job, params);
            System.out.println(execution.getStatus());
        }
    }
}
```

① ou `ctx.getBean(Job.class)` s'il n'y a qu'un seul `Job` dans le projet. Dans ce cas le nom du *bean* n'a pas d'importance.

Si les paramètres ont passés via la ligne de commande :

```
$ > java -jar payroll-1.0.0.jar year=2019 month=3
```

```
public static void main(String[] args){

    Map<String, String> argsMap = Stream.of(args)
        .filter(x -> x.contains("=") && !x.startsWith("-"))
        .map(x -> x.split("="))
        .collect(Collectors.toMap(x -> x[0], x -> x[1]));

    try(var ctx = new AnnotationConfigApplicationContext(PayrollJobConfiguration.class)) {
        Job job = ctx.getBean("payrollJob", Job.class); ①
        JobLauncher jobLauncher = ctx.getBean(JobLauncher.class);

        LocalDate now = LocalDate.now();
        JobParameters params = new JobParametersBuilder()
            .addLong("month", Long.parse(argsMap.get("month"))
            .addLong("year", Long.parse(argsMap.get("year"))
            .toJobParameters();
        JobExecution execution = jobLauncher.run(job, params);
        System.out.println(execution.getStatus());
    }
}
```

Dans le cas d'une application Spring boot, l'invocation de la `SpringApplication.run` suffit à exécuter tous les *jobs* du projet :

```
@SpringBootApplication @EnableBatchProcessing
public class PayrollJobConfiguration {

    // méthodes productrices reader(), processor(), writer() et prepareWireTransfers()
    // méthodes productrices transferRequestSender() et sendWireTransferToBank()

    @Bean(name="payrollJob")
    public Job payrollJob(JobBuilderFactory jobBuilderFactory) {
        JobBuilder builder = jobBuilderFactory.get("payrollJob");
        return builder.start(wireTransfers()).then(sendWireTransferToBank()).build();
    }

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

Nous pouvons changer ce comportement en valorisant à `false` la propriété `spring.batch.job.enabled`.

Si nous le conservons, les paramètres transmis lors du programme Java (`year=2019 month=3`) sont automatiquement ajoutés en tant que *job parameters*

Depuis une méthode annotée par `@Scheduled` :

```
@Component
public class JobScheduler{

    @Autowired
    private JobLauncher;

    @Autowired
    @Qualifier("payrollJob") ①
    private PayrollJob payrollJob;

    @Scheduled(cron="0 0 9 * * *") ②
    public void payrollJob(){
        LocalDate now = LocalDate.now();
        if(now.lengthOfMonth() - now.getDayOfMonth() != 2) {
            return; ③
        }
        JobParameters params = new JobParametersBuilder()
            .addLong("month", now.getLong(ChronoField.MONTH_OF_YEAR))
            .addLong("year", now.getLong(ChronoField.YEAR))
            .toJobParameters();
        JobExecution execution = jobLauncher.run(job, params);
    }
}
```

① Pour obtenir le *bean* dont le nom est *payrollJob*. Inutile s'il n'y a qu'un seul `Job` dans le projet.

② La méthode doit être invoquée chaque matin à 9h00.

③ La paie ne doit se déclencher que l'antépénultième jour du mois.

Notons que les paramètres sont accessibles dans les *beans* Spring (*reader*, *writer* et *processor* notamment) via l'annotation `@Value`.

Exemple :

```
@Value("#{jobParameters['year']}")
private int year;
```

Cela pose plusieurs questions :

1. où sont stockées les informations concernant l'exécution du *job* ?
2. à paramètres identiques, le *job* peut il être lancé plusieurs fois ?

Les types `JobParameters` et l'interface `JobRepository` apportent des réponses à ces questions.

Exemple :

```
public static void main(String[] args){
    try(var ctx = new AnnotationConfigApplicationContext(PayrollJobConfiguration.class)) {
        Job job = ctx.getBean("payrollJob", Job.class);
        JobLauncher jobLauncher = ctx.getBean(JobLauncher.class);
        JobParameters params = new JobParametersBuilder()
            .addLong("month", now.getLong(ChronoField.MONTH_OF_YEAR))
            .addLong("year", now.getLong(ChronoField.YEAR))
            .toJobParameters();
        JobExecution execution = jobLauncher.run(job, params);
        System.out.println(execution.getStatus());
    }
}
```

La méthode `jobLauncher.run` va

1. Créer une `JobInstance` (couple unique `Job`;`JobParameters`) s'il n'y en a pas déjà une.
2. Créer une `JobExecution` pour cette instance (existance ou nouvellement créée) sauf si
   - une exécution a déjà été réalisée avec succès (statut : `COMPLETED`).
   - le nombre de réessais a déjà été atteint.

Nous en déduisons que si un *job* a été lancé avec succès, il ne peut pas être relancé avec les mêmes paramètres.

Par défaut, Spring batch stocke en mémoire les instances de *jobs* et l'historique des exécutions.

Toutefois les informations sont perdues lorsque le process Java s'arrête, nous préferons donc un stockage en base de données.

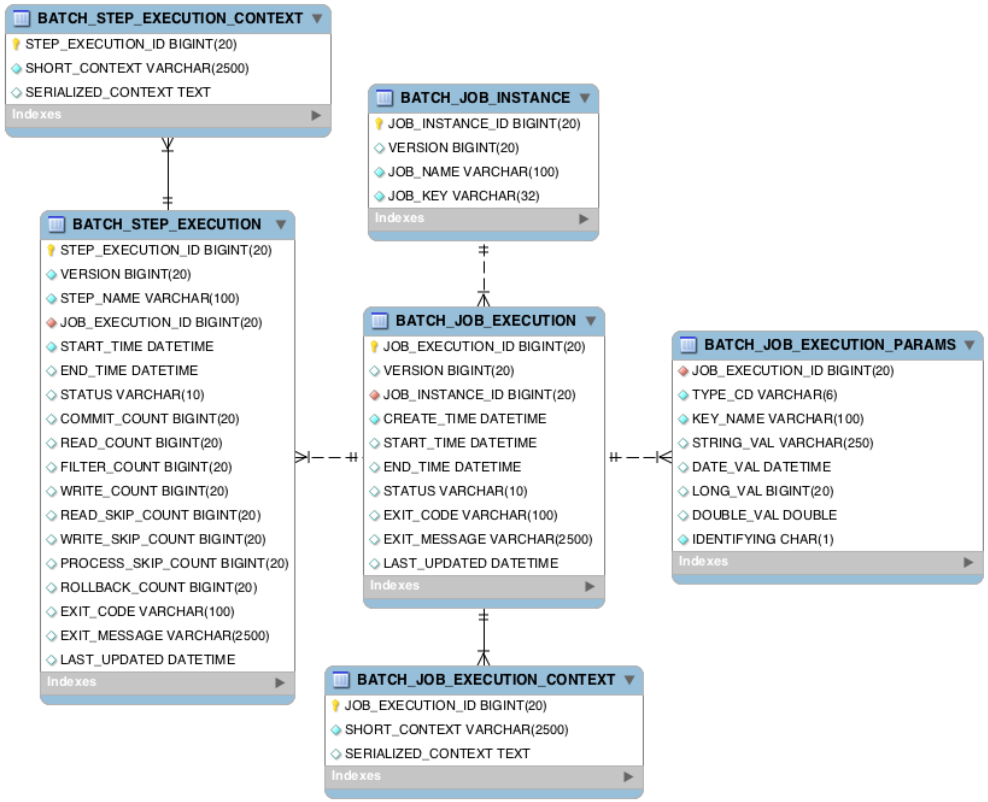
Il suffit pour cela de déclarer un *bean* de type `DataSource` :

```
@Bean @Primary
public DataSource ds(){
    return new DriverManagerDataSource("...");
}
```

Si plusieurs `DataSource` sont déclarées (exemple : une pour un *reader*, une pour un *writer*), il faut annoter par `@Primary` celle que nous souhaitons que Spring batch utilise.

Cela peut-être une bonne habitude d'utiliser l'annotation `@Primary` systématiquement, afin d'éviter toute regression en cas d'ajout d'un autre *bean* de type `DataSource`.

Cette base de données doit contenir les tables suivantes :



Des scripts de créations sont proposés dans le *package* `org.springframework.batch.core`.

Spring boot peut éventuellement créer ces tables si elle n'existent pas (propriété : `spring.batch.initialize-schema`, valeurs possibles : `always`, `never`).

Plus de détails sur les tables utilisées par Spring batch : https://docs.spring.io/spring-batch/4.1.x/reference/html/schema-appendix.html#metaDataSchema

Techniquement c'est l'interface `JobRepository` qui fournit une abstraction par rapport à la `DataSource`.

C'est l'annotation `@EnableBatchProcessing` qui permet l'inscription automatique d'un `JobRepository`.

Tests

Tester un batch consiste à vérifier la bonne exécution d'un *job* ou d'un *step*.

Spring fournit une librairie pour cela :

```
<dependency>
    <groupId>org.springframework.batch</groupId>
    <artifactId>spring-batch-test</artifactId>
</dependency>
```

Cette dépendance fournit notamment

- l'annotation `@SpringBatchTest`
- le *bean* `JobLauncherTestUtils`

Exemple :

```
@SpringBatchTest
@SpringJUnitConfig(classes=PayrollJobConfiguration.class)
class JobConfigTest {

    @Autowired
    private JobLauncherTestUtils launcher;

    @Test
    void testJob() throws Exception {
        JobParameters params = new JobParametersBuilder()
            .addLong("year", 2019L)
            .addLong("month", 2L)
            .toJobParameters();

        JobExecution exec = launcher.launchJob(params);
        assertEquals(BatchStatus.COMPLETED, exec.getStatus());
    }
}
```

Nous pouvons aussi tester un *step* :

```
@SpringBatchTest
@SpringJUnitConfig(classes=PayrollJobConfiguration.class)
class JobConfigTest {

    @Autowired
    private JobLauncherTestUtils launcher;

    @Test
    void testStep() throws Exception {
        JobParameters params = new JobParametersBuilder()
            .addLong("year", 2019L)
            .addLong("month", 2L)
            .toJobParameters();

        JobExecution exec = launcher.launchStep("preparepreprepareWireTransfers", params);
        assertEquals(ExitStatus.COMPLETED, exec.getStatus());
    }
}
```

Généralement les tests utilisent des source de données différentes de cette utilisée en production..

Pour cela il faut externaliser dans *properties* les chaines de connexions et le chemin des fichiers.

Ainsi la définition de l'`ItemReader` deviendrait :

```
@SpringBootApplication
@EnableBatchProcessing
@PropertySource("classpath:payrollJob.properties")
public class PayrollJobConfiguration {

    @Autowired
    private Environment env;

    @Bean @StepScope @StepScope @Scope(proxyMode=ScopedProxyMode.TARGET_CLASS)
    public ItemReader<Employee> reader(){
        String dbUrl = env.getProperty("employeesDb.url");
        String dbUsername = env.getProperty("employeesDb.username");
        String dbPassword = env.getProperty("employeesDb.password");
        return new JdbcCursorItemReaderBuilder<Employee>()
            .name("employeesItemReader")
            .dataSource(new DriverManagerDataSource(dbUrl, dbUsername, dbPassword))
            .sql("select id, firstname, lastname, iban, bic from Employee")
            .beanRowMapper(Employee.class)
            .build();
    }

    // autres méthodes
}
```

Dans la classe de test il faut déclarer un fichier de configuration dédié (via `@TestPropertySource`) qui propose des valeurs alternatives pour ces propriétés.

```
@SpringBatchTest
@SpringJUnitConfig(classes=PayrollJobConfiguration.class)
@TestPropertySource("classpath:payrollJob-tests.properties")
class JobConfigTest {

    @Autowired
    private JobLauncherTestUtils launcher;

    @Test
    void testJob() throws Exception {
        JobParameters params = new JobParametersBuilder()
            .addLong("year", 2019L)
            .addLong("month", 2L)
            .toJobParameters();

        JobExecution exec = launcher.launchJob(params);
        assertEquals(BatchStatus.COMPLETED, exec.getStatus());
    }
}
```

Cela pourrait s'appliquerait aussi

- au chemin du fichier écrit par le *writer*
- à l'URI du *endpoint* où la *tasklet* envoie le fichier généré par le *writer*.

Si nous avons besoin d'intialiser les données d'une base de données utilisée par un *reader* ou un *writer* nous pouvons :

- disposer d'un script sql d'initialisation dans le *source folder* `src/test/resources`.
- exécuter ce script dans une méthode annotée par `@BeforeEach` dans la classe de test.

Soit `init.sql` un script d'initialisation présent dans `src/test/resources`.

Nous pourrions écrire :

```
@SpringBatchTest
@SpringJUnitConfig(classes=PayrollJobConfiguration.class)
@TestPropertySource("classpath:payrollJob-tests.properties")
class JobConfigTest {

    @Environment
    private Environment env;

    @BeforeEach
    public void initData() throws Exception {
        String dbUrl = env.getProperty("employeesDb.url");
        String dbUsername = env.getProperty("employeesDb.username");
        String dbPassword = env.getProperty("employeesDb.password");
        try(Connection conn = DriverManager.getConnection(dbUrl, dbUsername, dbPassword)){
            ScriptUtils.executeSqlScript(conn, new ClassPathResource("init.sql"));
        }
    }

    // méthode de test
}
```

Synthèse

Spring batch fournit une infrastructure complète pour les besoins *batch* d'une entreprise :

- workflow
- différents type de *steps* : ETL ou tâche.
- règles de réessais
- transaction
- supervision
- tests

De nombreux types structurent une application *batch*, cela offre un cadre de travail qui favorise la maintenabilité.

- `Job`
- `Step`
- `ItemWriter<T>`, `ItemProcessor<I, O>`, `ItemWriter<T>`
- `Tasklet`

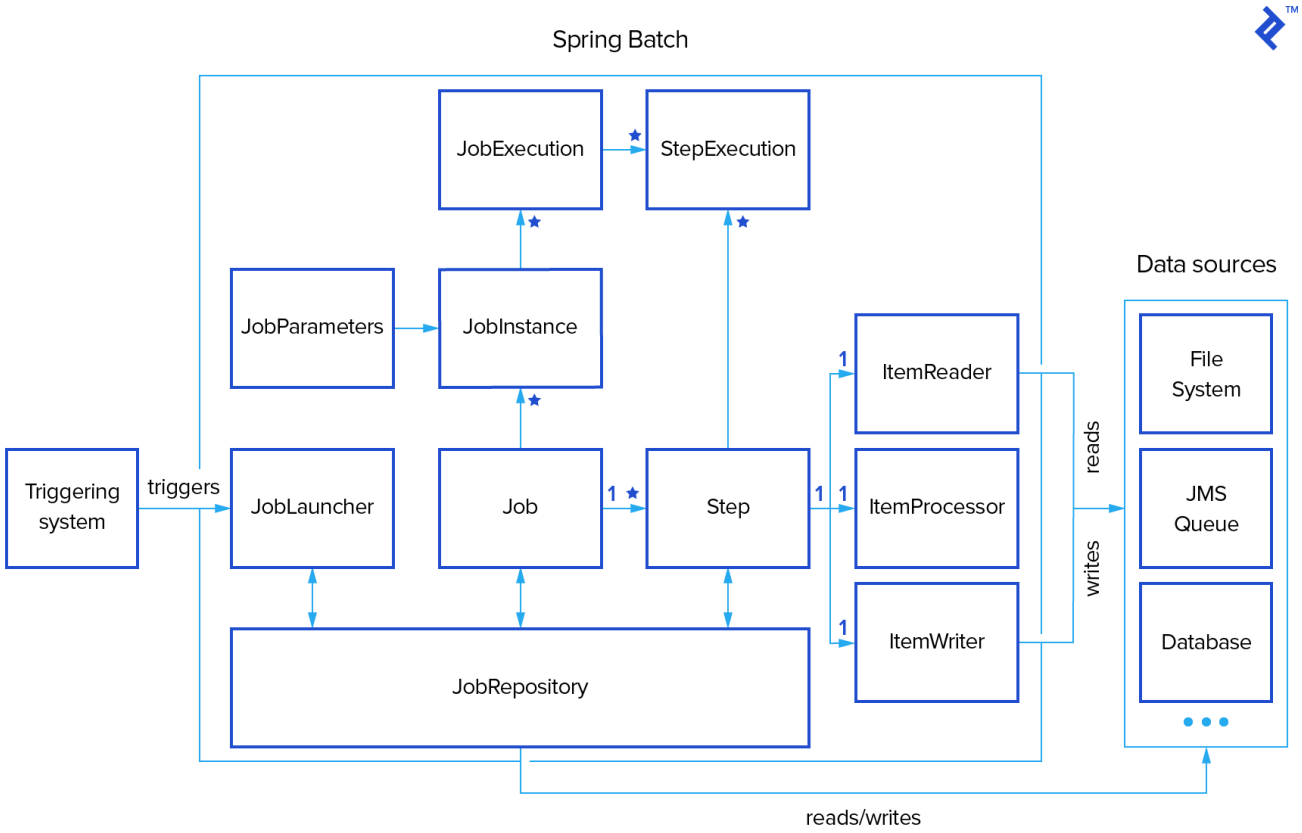
Des services techniques sont mis à disposition :

- `JobLauncher`
- `JobRepository`

Enfin citons les *facilitateurs* que sont :

- les implémentations de `ItemReader<T>` et `ItemWriter<T>`
- `JobBuilderFactory` et `StepBuilderFactory`

Nous remarquons par ailleurs l'utilisation intensive du pattern *builder* et des *inter-beans references*.



Crédits : Alexey Saenko, Toptal. https://www.toptal.com/spring/spring-batch-tutorial

Voir aussi :

- la validation des paramètres.
- Les commit interval dynamiques, via les `CompletionPolicy`.
- les *listeners* (`JobListener`, `StepListener`).
- https://docs.spring.io/spring-batch/4.1.x/reference/html/common-patterns.html#commonPatterns
- le bean `JobExplorer`
- le couplage avec Spring cloud data flow pour la supervision des batchs.

Annexes

- JPA : initiation / rappels

JPA : initiation / rappels

- Principes
- Le mapping
- Le lazy loading
- Manipulation de l'API
- Précautions

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
</dependency>
```

Principes

La persistance vise à assurer une liaison transparente entre le modèle relationnel et le modèle objet et à rendre notre application totalement indépendante de la source de données.

Dans les cas les plus simples, une instance correspond à une ligne dans la base de données.

Très vite des différences fondamentales apparaissent entre la modélisation objet et la modélisation relationnelle :

- Relations (association, agrégation et composition en OOP / clé(s) étrangères dans un SGBDR)
- Héritage (natif en OOP / clé(s) étrangères dans un SGBDR).

L'écriture du code nécessaire à la translation d'un modèle à l'autre est une tache répétitive et sans valeur ajoutée.

Des frameworks tels qu'Hibernate répondent à ce besoin, devant leur succès le JCP a normalisé la persistence à travers l'API JPA.

Utiliser JPA revient à :

- Écrire le mapping par annotations ou par xml. Le mapping décrit la translation entre le modèle objet (nos entités) et le modèle relationnel.
- Manipuler l'API, via l'entityManager (`javax.persistence.EntityManager`) pour effectuer les opérations *CRUD* sur nos entités persistantes.

JPA prend en charge le dialogue avec la base de données :

- Notre application transmet à JPA les instances à sauvegarder, modifier ou supprimer en invoquant les méthodes adéquates. Ceci conduit à des requêtes SQL de type `INSERT`, `UPDATE` ou `DELETE`.
- JPA retourne à notre application des instances (ou une liste d'instances au sens `java.util.List`) préalablement sauvegardées. Ceci donne lieu à des requêtes de type `SELECT`

Le mapping

Entités fortes et entités faibles

Pour les entités fortes : `@Entity`

Pour les entités faibles : `@Embeddable`

Par defaut toute les propriétés d'une classe sont *mappées* vers une colonne de la table à laquelle est associée la classe.

Si un champ ne doit pas être *mappé* nous l'annotons par `@Transient`.

Une **entité forte** peut se concevoir indépendamment des autres classes, elle est autonome dans sa définition. Exemple : un immeuble, un livre. Une entité forte a un identifiant unique.

Une **entité faible** n'existe que comme composant d'une entité forte. Exemple : un étage (car un étage appartient forcément à un immeuble), une page (car une page fait forcément partie d'un livre). Une entité faible n'a pas d'identifiant.

Si une entité étend une classe qui n'est pas elle même une entité, nous devons annoter cette dernière par `@MappedSuperClass` si nous souhaitons que ses propriétés soient *mappées*.

Valeurs

	Côté classe Java	Côté base de données
Identifiant(s) :	`@Id`	clé(s) primaire(s) dans la table associée à l'entité forte où se trouvent le champ identifiant.
Autres champs :	`@Basic`	colonnes dans la table associée à l'entité où se trouvent les champs

Par défaut tous les champs sont *mappés* (`@Basic` est donc facultatif).

L'annotation `@Column` permet de préciser le nom de la colonne, au cas où celui-ci ne soit pas le même que celui du champ de la classe Java.

Associations

Une relation d'association se fait toujours vers une entité forte.

	Côté classe Java	Côté base de données
Association n-1	`@ManyToOne`	clé(s) étrangère(s)
Association 1-n	`@OneToMany`	N/A
Association n-n	`@ManyToMany`	clé(s) étrangère(s) + table de jointure
Association 1-1	`@OneToOne`	clé(s) étrangère(s)

Compositions

Une relation de composition se fait toujours à partir d'une entité forte et vers une entité faible.

	Côté classe Java	Côté base de données
Composition 1-1	`@Embedded`	colonnes dans la table associée à l'entité forte propriétaire de la relation
Composition 1-n	`@ElementCollection`	table supplémentaire avec clé étrangère vers la table associée à l'entité forte propriétaire de la relation

Héritage

Si une entité forte (`@Entity`) a des classes filles :

Cas 1

toutes les classes filles voient leurs instances stockées dans la même table. La classe parente doit alors être annotée par `@Inheritance(strategy=InheritanceType.SINGLE_TABLE)`

Il faut alors une colonne discriminante qui précisera pour chaque ligne dans la table de quel sous-type il s'agit. Par défaut le nom de la table est celui de la classe parente.

Cas 2

les instances des classes filles sont dans des tables dédiées, et une table « parente » factorise les informations communes.

- La classe parente doit alors être annotée par `@Inheritance(strategy=InheritanceType.JOINED)`.
- Les tables auxquelles sont associées les classes filles ont une colonne qui est en même temps la clé primaire et la clé étrangère vers la clé primaire de la table associée à la classe parente.

Le lazy loading

La matérialisation systématique d'une relation lors de la récupération d'une entité n'est pas forcemment judicieux puisque cela conduit à des jointures entre les tables ou à des requêtes `select` supplémentaires.

Le principe du *lazy loading* permet justement de déclarer, pour chaque relation, si celle-ci doit systématiquement être matérialisée* lorsque l'on récupère une instance d'une entité. Le choix se fait par l'attribut `fetch` :

Deux stratégies possibles :

- `fetch = FetchType.EAGER` : la relation est matérialisée immédiatement (au moment de la fabrication de l'objet par lecture en base de données).
  - avantage : graph d'objet plus riche.
  - inconvénient : risque de requêtes sql plus complexes (join) ou plus nombreuses (!).
- `fetch = FetchType.LAZY` : la relation est matérialisée quand on y accède.
  - avantage : requêtes sql plus légères.
  - inconvénient : graph d'objet moins riche.

Les valeurs par défaut del'attribut `fetch`:

- `FetchType.EAGER` pour les `@ManyToOne`, `@OneToOne` et `@Embedded`.
- `FetchType.LAZY` pour les `@OneToMany`, `@ManyToMany` et `@ElementCollection`.

matérialiser une relation signifie :

1. recherche de l'élément dans le cache niveau 1 (celui de l'`EntityManager`)
2. s'il n'est pas dans le cache niveau 1 : recherche de l'élément dans le cache niveau 2 (celui de l'`EntityManagerFactory`);
3. s'il n'est pas dans le cache niveau 2 : recherche de l'élément dans la base de données.

Le cache niveau 1 est toujours actif, le cache niveau 2 s'active en annotant la classe concernée par `@javax.persistence.Cacheable`

favoriser `fetch = FetchType.EAGER` pour les relations vers des classes dont on peut mettre les instances en cache niveau 2.

toujours utiliser `fetch = FetchType.LAZY` si la relation porte vers une classe dont les instances ne sont pas en cache niveau 2

Le langage JPA-QL propose un opérateur *join fetch* pour anticiper le chargement d'une relation *LAZY* (cela conduit à une jointure dans la requête SQL).

Paramétrage

C'est dans le fichier `META-INF/persistence.xml` (dans un *source folder*) que l'on indique quelle implémentation de JPA nous voulons utiliser et éventuellement d'autres paramètres

```
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
    version="2.1">

    <persistence-unit name="pu1" transaction-type="RESOURCE_LOCAL"> ① ②
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider> ③
        <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode> ④
        <properties> ⑤
        <!-
            propriété de connexion : javax.persistence.jdbc.url, etc..
            properties additionnelles spécifiques au provider
            (pour hibernate : hibernate.dialect, hibernate.cache.provider_class)
        -->
        </properties>
    </persistence-unit>
</persistence>
```

① `name` (ici *pu1*) est arbitraire, il sert à différencier les 'persistence-unit' s'il y en a plusieurs.

② `transaction-type` indique que le serveur d'application s'occupe de gérer les transactions.

③ `provider` : le choix de l'implémentation JPA

④ `shared-cache-mode` : politique d'activation du cache niveau 2

⑤ `properties` : liste de propriétés spécifiques à l'implémentation JPA.

Manipulation de l'api

```
// ouverture de l'entityManagerFactory (une fois au démarrage de l'application)
EntityManagerFactory emf = Persistence.createEntityManagerFactory("pu1");
// ouverture de l'entityManager (à chaque transaction)
EntityManager em = emf.createEntityManager();
// close : fermeture de l'entityManager (à la fin de la transaction)
em.close();
// close : fermeture de l'entityManagerFactory (à l'arrêt de l'application)
emf.close();
```

C'est l'`EntityManager` qui propose les méthodes correspondant aux opérations de lecture, écriture et suppression de nos entités fortes dans la base de données.

Citons quelques méthodes essentielles de l'interface `EntityManager` :

```
public void persist(Object entity);

public <T> T merge(T entity);

public <T> T find(Class<T> entityClass, Object primaryKey);

public void remove(Object entity);

public <T> TypedQuery<T> createQuery(Class<T> entityClass, String qlString);
```

L'implémentation de ces méthodes est de la responsabilité de l'implémentation JPA mentionnée dans le fichier `persistence.xml` (voir élément `provider`).

Soit une entité `Person` (entité forte) ayant 4 propriétés : `id` (`@Id`), `firstname`, `lastname`, `age` :

```
// find : récupération par l'id (dans le cache d'abord, en base en dernier recours)
Person p = em.find(Person.class, 1);
// persist : requête(s) SQL de type INSERT pour stocker en base une nouvelle instance
Person p = new Person("John", "Doe");
em.persist(p);
// remove (sur un objet attaché) : requête(s) SQL de type DELETE lors du flush
Person p = em.find(Person.class, 1);
em.remove(p);
```

```
String s = "select p from Person p where p.lastname like :n and p.age >:a order by p.age";

TypedQuery<Person> query = em.createQuery(Person.class, s)
.setParameter("n", "John")
.setParameter("a", 18);

List<Person> persons = query.getResultsList(); // conséquence : requête SELECT

String query = "update Person p set p.lastname=upper(p.lastname)";
em.createQuery(query).executeUpdate();
// consequence : requête UPDATE
```

Relation avec Spring

Anti pattern : ouvrir et fermer à l'`EntityManager` à chaque fois qu'une méthode en a besoin.

Exemple : l'invocation du service conduit à utiliser 10 méthodes de la couche *business*, chacune créé un `EntityManager` et le ferme après utilisation.

Pattern : un `entityManager` par tansaction. Un `entityManager` est ouvert au même moment que la transaction, il est fermé à la fin de la transaction. C'est à Spring qu'il faut confier la gestion (création, injection, fermeture) des `entityManager`

Les transactions entourent quant à elles l'exécution des méthodes annotées par `@Transactional` avec un niveau de propagation fixé à `REQUIRED` ou `REQUIRES_NEW`

Ces méthodes se trouvent traditionnellement dans la couche service.

Les classes de notre applications utilisent un `EntityManager` mais ne s'occupent pas sa création. C'est à Spring de le faire.

Nous retrouvons ici le principe d'*inversion du contrôle* (IOC).

L'`EntityManager` créé pour la transaction en cours est disponible avec l'annotation `@javax.persistence.PersistenceContext` :

```
@Service
public class MyService {

    @javax.persistence.PersistenceContext
    private javax.persistence.EntityManager em;

    @Transactional(propagation = Propagation.REQUIRED)
    public void marry(int husbandId, int wifeId) {
        Person p1 = this.em.find(Person.class, husbandId);
        Person p2 = this.em.find(Person.class, wifeId);
        p1.setMarried(true);
        p2.setMarried(true);
        // les modifications sur p1 et p2 sont observées par l'entityManager et
        // donneront lieu aux requêtes SQL correspondantes au moment
        // du commit de la transaction auquel l'entityManager est associé
        Wedding w = new Weeding(p1, p2, LocalDate.now());
        this.em.persist(w);
    }
    // d'autres méthodes à implémenter...
}
```

Précautions

- Définir avec précaution le fetch type (lazy vs eager).
- Prendre garde au n+1 select
- Activer le debug SQL (trace des requêtes SQL)
- Comprendre les différents états d'une entité : transiente, attachée, détachée.
- Comprendre l'alignement entre le cycle de vie de l'`EntityManager` et celui de la transaction.