Concevoir et développer des applications avec Spring

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

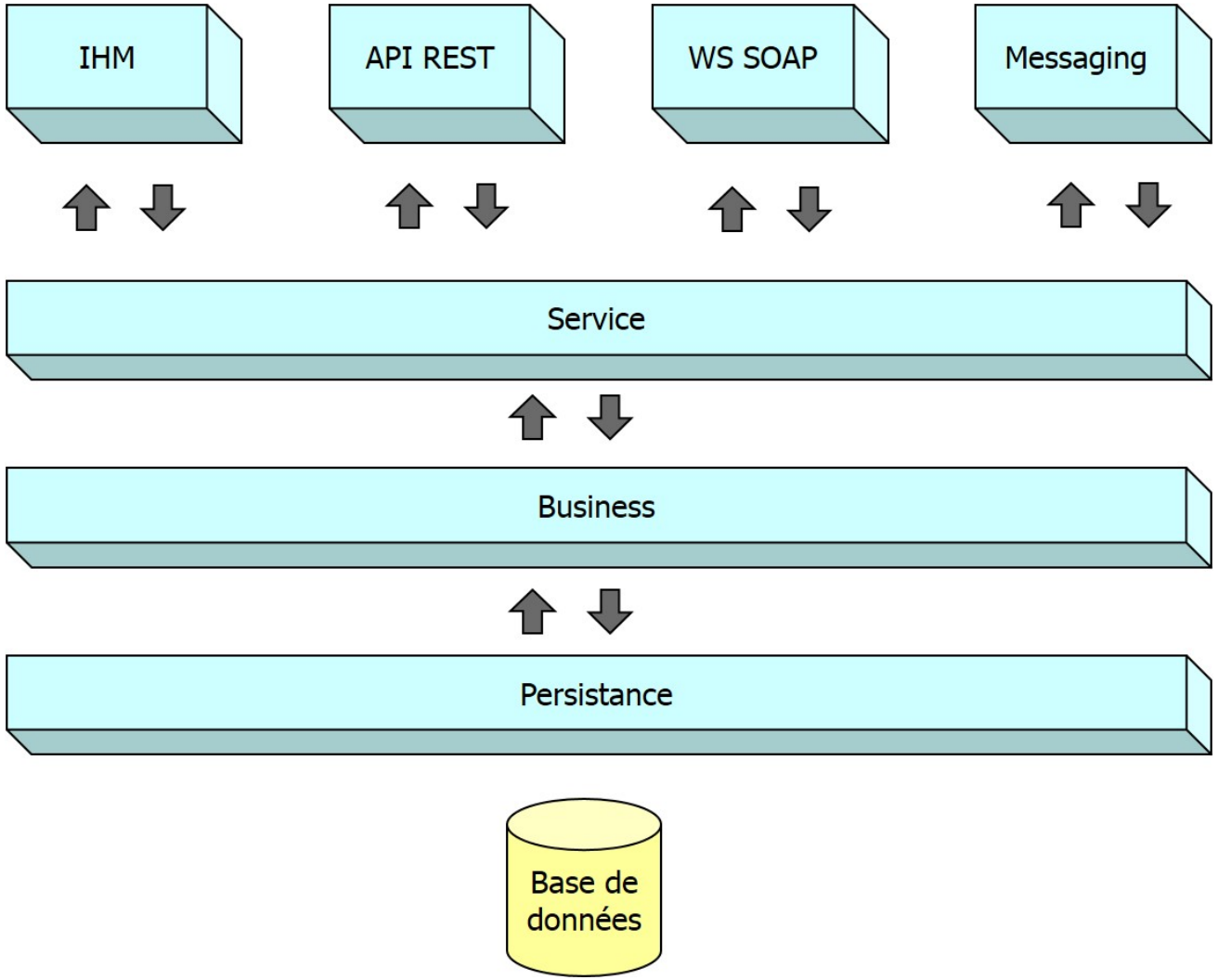
A ce stade l'application prend la forme d'un *jar*, elle est comme la partie immergée d'un iceberg.

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` entretien un lien fort avec `LdapUserDao`. Conséquence :

- 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 recoit 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", dao);
        objects.put("userService", userService);
    }

    public static Object lookup(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 *lookup* 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.lookup("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", dao);
        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`) et ne prennent pas de paramètres. Ce sont ces méthodes qui 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.

Distinguons deux types de tests :

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*.

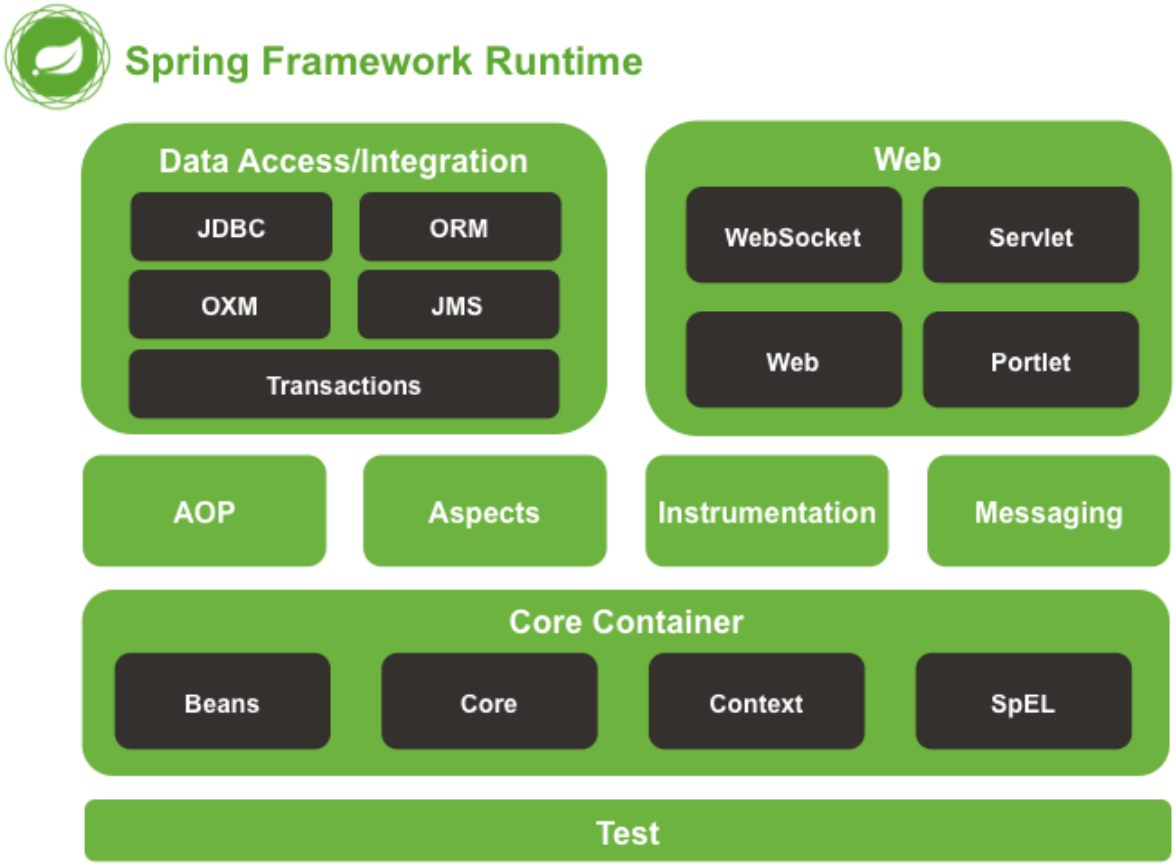
Spring : introduction

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, HttpInvoker, Hessian, Burlap…).

Ne nécessite aucune autre infrastructure qu'une machine virtuelle: la seule présence des bonnes libraires (spring-*.jar) suffit.

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

Data access

Conscient que de nombreux projets délèguent la gestion de la persistance à un framework (Hibernate par exemple) ou à l'API JPA, Spring propose un couplage avec ces technologies et un intercepteur transactionnel.

Ceci permet de gérer très finement et de manière non intrusive :

- la démarcation transactionnelle.
- le cycle de vie des objets impliqués dans l'accès aux sources de données dans le cadre d'une transaction (`EntityManager` JPA par exemple).

Web

Spring propose trois manière de développer des applications web :

- Spring MVC, 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 basées sur le protocole HTTP (requête/réponse).

IHM

Spring MVC et Spring WebFlux adressent tous les besoins courants associés au développement d'une couche IHM web.

Nos besoins :

- Routing
- Controleurs POJO
- 2 ways binding (modèle < - > vue)
- Validation (support l'API bean validation)
- Couplages avec des technologies de présentation (jsp, Freemarker, Thymleaf…)
- Internationalisation
- Gestion des exceptions

API REST

Spring MVC et Spring WebFlux permettent aussi de développer des API REST.

Nos besoins :

- Routing
- Contrôleurs POJO
- Validation (support l'API bean validation)
- Abstraction par rapport à la requête et à la réponse HTTP.
- Négociation de contenu (couplage avec *Jackson* ou *GSON* pour la sérialisation).
- Gestion des exceptions.

WebSocket

Spring facilite la mise en place de communication bidirectionnelle entre un navigateur et un serveur web.

Nos besoins :

- Abstraction vis à vis du protocole grâce à *SockJS*
- Logique de messaging pub/sub avec *Spring messaging* et *STOMP*.
- Externalisation du stockage des sessions grâce à *Spring sessions*. Ceci est particulièrement utile lors du déploiement de l'application sur un *cluster*.

Integration

L'intégration a d'autres technologies prend plusieurs forme :

- Faciliter l'utilisation d'une technologie particulière en prenant en charge le code répétitif. Spring fournit alors des classes utilitaires qui ciblent une technologie précise : `JdbcTemplate`, `HibernateTemplate`, `JmsTemplate`, `JavaMailSenderImpl`.
- Intégrer des objets distants (rmi, webservice…) dans notre projet en les voyant comme des implémentations d'interfaces locales.
- Couplage avec d'autres *frameworks* pour disposer des beans Spring dans des objets non gérées par Spring (exemple : Junit, GWT, JSF, etc…)

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 @Scope("singleton") ①
public class ShopServiceImpl implements ShopService {
    // couplage avec JPA : injection du contexte de persistence.
    @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 @Scope("singleton") ①
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();
        // La servlet Spring récuperera ce que renvoie notre méthode et fabriquera une
        // réponse dont le statut sera 201 et dont l'en-tête location aura comme valeur
        // l'URI de la ressource nouvellement créée.
    }
}
```

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.

Environnement de développement

- JDK >= 1.6 pour Spring 4, JDK >= 1.8 pour Spring 5.
- Eclipse ou IntelliJ..
- Framework Spring et frameworks tiers (Hibernate, Junit, Logback, etc...), idéalement récupérés via Maven.

Pour les dépendances Spring il faut de préférence utiliser le bom 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>5.0.7.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Environnement d’exécution

Plusieurs solutions sont possibles :

1. Utiliser 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. Utiliser un **servlet container**, Tomcat, Jetty ou Undertow par exemple. 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. Utiliser **Spring Boot**. Dans ce cas l’application porte avec elle non seulement les librairies dont elle a besoin (Spring et autres librairies tiers) mais aussi son environnement d’exécution (un *servlet container* : Tomcat, Jetty ou Undertow). Ainsi l’application est un jar avec une méthode `main` comme point d’entrée et n’a besoin que d’une JVM pour fonctionner.

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

La seconde et la troisième fonctionnent particulièrement bien avec les environnements cloud PaaS (Amazon Beanstalk par exemple) car le déploiment consiste alors à seulement uploader et déployer un war ou un jar sur un environnement managé (Java 8 et Tomcat 8 dans le cas d'Amazon Beanstalk).

La troisième solution (Spring Boot) facilite même la *dockerisation* notre application, le DockerFile ressemblerait alors à :

```
FROM openjdk:8u111-jdk-alpine
ADD /target/myArtifact-myVersion.jar app.jar
RUN sh -c 'touch /app.jar'
ENTRYPOINT ["java", "-jar","/app.jar"]
```

Et l'image construite suite au docker `build` pourrait être déployée sur un *Docker host* (Kubernetes par exemple).

Spring core container

- Principes
- L'ApplicationContext
- Déclaration d'un bean Spring
- Cycle de vie d'un bean Spring
- Injection de dépendances
- Tests d'un bean Spring
- Services techniques
- Synthèse

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

Principes

Une architecture n-tiers est composée de plusieurs couches, typiquement :

- Une couche service
- Une couche business
- Une couche d’accès au données.

Les composants de des différentes couches ont une responsabilité precise et doivent être liés entre eux :

- Chaque composant de la couche service utilise un ou plusieurs composants de la couche business.
- Chaque composant de la couche business utilise un ou plusieurs composants de la couche d’accès aux données..

DONT : couplage fort

```
public class ShopServiceImpl implements ShopService {
    private ShoppingCartManager shoppingCartManager;
    private ProductManager productManager;
    private List<ShippingManager> shippingManagers;
    private ClientManager clientManager;
    private PaymentManager paymentManager;

    public ShopServiceImpl(){ // TROP DE RESPONSABILITE CONFIEES AU SERVICE
        this.shoppingCartManager = new ShoppingCartManagerImpl();
        this.productManager = new ProductManagerImpl ();
        this.shippingManager = new ArrayList<ShoppingCart>();//add impls
        this.clientManager = new ClientManagerImpl();
        this.paymentManager = new PaymentManagerImpl();
    }
    public void validateShoppingCart(int id){
        // use dependencies
    }
}
```

DO : couplage faible par application du pattern strategy

```
public class ShopServiceImpl implements ShopService {
    private ShoppingCartManager shoppingCartManager;
    private ProductManager productManager;
    private List<ShippingManager> shippingManagers;
    private ClientManager clientManager;
    private PaymentManager paymentManager;

    public ShopServiceImpl(ShoppingCartManager shoppingCartManager, ProductManager productManager, List<ShippingManager>
shippingManagers, ClientManager clientManager, PaymentManager paymentManager) {
        this.shoppingCartManager = shoppingCartManager;
        this.productManager = productManager;
        this.shoppingManagers = shippingManagers;
        this.clientManager = clientManager;
        this.paymentManager = paymentManager;
    }

    public void validateShoppingCart(int id){
        // use dependencies
    }
}
```

Ce que nous voyons ici est l'inversion du contrôle : la classe ne vas chercher les objets dont elle a besoin, elle les reçoit.

Reste à mettre en place un mécanisme qui instanciera les objets (et gérera ainsi leur cycle de vie) en leur transmettant leurs dépendances.

Le premier « conteneur IOC » est apparu en Java : Spring.

Ainsi les objets sont vus comme des 'ingrédients' et la 'recette' pour les lier les uns aux autres est donnée à la factory. Celle-ci est ainsi capable d'« assembler » le système et de rendre les objets disponibles à ceux qui en ont besoin (applications web, applications console…, qui ne sont/peuvent pas être gérées par le container).

Le succès de Spring a conduit la JCP a établir deux specifications pour standardiser ces principes : CDI (Context and Dependency Injection).

Un conteneur est capable de 'prendre en charge' les objets qui constituent l'ossature de notre application :

- objets de la couche service,
- objets de la couche business,
- composants de la couche DAO,
- composants de la couche web.

Le conteneur joue le rôle de factory, il peut :

- Gérer le cycle de vie (*patterns singleton* et *prototype*).
- Injecter les dépendances (*pattern strategy*)
- Intercepter l'invocation des méthodes et appliquer des traitements avant/après l'invocation (*pattern proxy*)
- Gérer l'exposition (en remoting, en SOAP, etc...)

Nos exigences :

- Pouvoir confier au conteneur n'importe quel objet POJO
- Pouvoir injecter par constructeur ou par propriété.
- Pouvoir injecter aussi bien des valeurs (types primitifs, String) que d'autres objets
- Pouvoir injecter des `List`, `Set`, `Map` et `Array` (de valeurs ou de références vers d'autres objets), des `Properties`.

Pour utiliser Spring :

- Ajouter au projet une dépendance à Spring (via Maven, artifactId : spring-context).
- Déclarer les objets que l'on souhaite lui "confier". Cela se fait aujourd'hui majoritairement par annotations mais historiquement seule une configuration xml était possible (à l'époque où les annotations n'existaient pas en Java).

Spring est notre factory mais pas seulement, il propose aussi des services techniques prêts à l'emploi. C'est en cela qu'il est un conteneur.

Conteneur = factory + services techniques.

L'ApplicationContext

L'`ApplicationContext` se met en place sur la base d'une classe de configuration :

```
@Configuration
public class MyApplication {
}
```

La principales annotations de la classe de configuration :

`@ComponentScan`
: pour activer le *scan* des classes annotées par `@Component` (ou les dérivés de `@Component`). Spring construira les objets correspondants et les inscrira dans le contexte. Par défaut le *scan* s'effectue dans le package en cours, l'attribut `basePackages` permet d'étendre le scan à d'autres *packages*.

`@Import`
: pour inclure des beans d'après leur `.class`, permet aussi d'inclure des classes de configuration.

`@PropertySource`
: pour inscrire dans le contexte les propriétés lues dans un fichier .properties. Celles-ci sont ensuite injectables par l'annotation `@Value`

Le chargement du contexte se fait au démarrage de l'application :

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

L'application est alors mise en place.

Nous pouvons considérer le chargement du contexte applicatif comme faisant parti de la phase de déploiement de l'application

C'est une opération atomique : la moindre erreur de configuration conduira à une exception *Failed to load application context*

Une fois l'application mise en place, les objets construits par Spring sont accessibles via la méthode `getBean`.

Déclaration d'un bean Spring

Déclaration d'un bean auprès du conteneur :

`@Component` (annotation Spring) ou un de ses dérivés :

- `@Service` pour les services métiers
- `@Repository` pour la couche d'accès aux données
- `@Controller` pour les contrôleurs MVC
- `@RestController` (pour les endpoints REST)

Il est aussi possible de créer un dérivé de `@Component`, en créant une annotation elle-même annotée par `@Component` (comme le sont `@Service`, `@Repository`, `@Controller` et `@RestController`)

L'attribut `value` de ces annotations permet d'inscrire le bean dans le contexte Spring sous un autre identifiant que celui déterminé par défaut (qui est le nom de la classe avec une minuscule pour la première lettre).

```
@Component // ou @Service puisqu'il s'agit d'un service
public class ShopServiceImpl implements ShopService {
    // champs (points d'injection)
    // méthodes
}
```

L'annotation des classes par `@Component` (ou dérivés de `@Component`) a des limites :

- Impossible d'annoter des objets dont nous n'avons pas le code source.
- Impossible de disposer de plusieurs instances d'une même classe mais construites de manière différente.

Dans ce cas : utiliser des **méthodes productrices**, annotées par `@Bean` dans une classe de configuration.

Les méthodes productrices se définissent dans la classe de configuration.

Les objets qu'elles renvoient seront inscrits par Spring dans le contexte applicatif.

Ces méthodes peuvent recevoir par argument tous les objets que Spring connait par ailleurs.

```
@Configuration @ComponentScan
@PropertySource("classpath:datasource.properties")
public class MyApplication {

    @Bean
    public DataSource ds(Environment env){
        BasicDataSource ds = new BasicDataSource();
        // utilisation de env.getProperty() pour récupérer le
        // username, le password, l'url, le driver class name....
        return ds;
    }
}
```

N'importe quelle classe annotée par `@Component` (ou dérivés) ou n'importe quelle méthode productrice peut ensuite recevoir la `DataSource` par injection.

Par défaut, les beans *singleton* sont construits au démarrage de l'application lors de la création de l'`ApplicationContext`.

Nous pouvons redéfinir ce comportement avec l'annotation `@Lazy`

Elle peut se poser à côté de l'annotation `@Component` (et ses dérivés) et à côté de l'annotation `@Bean`

Dans ce cas le bean sera créé au moment d'être injecté ou au moment du premier appel à la méthode `getBean` de l'`ApplicationContext`.

Pour définir que ce comportement est le comportement par défaut des beans découvert par Spring, il suffit d'utiliser l'attribut `lazyInit` de l'annotation `@ComponentScan`

L'inscription conditionnelle d'un bean consiste à n'inscrire un bean dans le contexte que si une certaine condition est remplie.

Une première manière de mettre en application cette flexibilité est d'utiliser les profiles :

D'abord nous annotons nos beans par l'annotation `@Profile`.

Exemple : `@Profile("test")`.

Peuvent donc être annotés :

- Les classes annotées par `@Component` ou dérivés (`@Service`, etc...)
- Les méthodes productrices annotées par `@Bean`

Le bean ainsi annoté ne sera inscrit dans le contexte applicatif que si le profil (ici : test) est actif.

L'annotation peut aussi être posée sur une classe de configuration, auquel cas elle s'applique de fait à tout ce qui est déclarée dans celle-ci.

Ensuite nous lançons le programme en spécifiant le(s) profil(s) actifs, il y a plusieurs manière de le faire :

- `@ActiveProfiles` sur les tests unitaires.
- avec la variable d'environnement `SPRING_PROFILES_ACTIVE`
- En fixant une valeur au paramètre `spring.profiles.active` au lancement de la jvm pour une application Spring boot
- en tant que context-param ou init-param d'une application web (nom du paramètre : `spring.profiles.active`)

Pour aller au-delà des profils il est possible d'utiliser l'annotation `@Conditional`.

Celle-ci doit être présente sur les beans (`@Component` et dérivés, méthodes productrices `@Bean`) et sur les classes de configuration.

Elle reçoit en argument le *.class* d'une implémentation de l'interface `Condition`. Cette dernière définit une méthode `matches` qui retourne `true` ou `false` :

```
public interface Condition {
    boolean matches(ConditionContext ctx, AnnotatedTypeMetadata metadata);
}
```

Ainsi un *bean* peut être inscrit ou non dans le contexte d'après une condition qui peut-être déterminée dynamiquement.

L'annotation `@Scope` permet de préciser le cycle de vie de l'objet : singleton (par défaut) ou prototype.

singleton
: l'objet ne maintient pas d'état, la même instance peut être partagée par toutes les classes qui en ont besoin.

prototype
: l'objet maintient un état, une instance dédiée doit reconstruite pour chaque classe qui en a besoin.

```
@Component // ou @Service puisqu'il s'agit d'un service
@Scope("singleton") // optional car singleton est le scope par défaut
public class ShopServiceImpl implements ShopService {
    // champs (points d'injection)
    // méthodes
}
```

`@Scope` peut aussi se poser sur une méthode productrice, s'il est valorisé à *prototype* il est alors possible de récupérer la référence vers le point d'injection :

```
@Bean
@Scope("prototype")
public Logger logger(InjectionPoint ip){
    return LoggerFactory.getLogger(ip.getMember().getDeclaringClass());
}
```

Enfin il est possible de déclarer dans nos beans des *callbacks*, c'est-à-dire des méthodes qui seront invoquées par Spring à un moment particulier du cycle de vie de l'objet.

- `@PostConstruct` : une méthode annotée ainsi sera invoquée après l'instanciation de la classe et après que les injections aient été réalisées sur le *bean*.
- `@PreDestroy` : une méthode annotée ainsi sera invoquée lorsque le *bean* est sorti du contexte applicatif (c'est-à-dire à l'arrêt de l'application pour les singletons).

Il ne s'agit pas d'annotations Spring mais d'annotation standard Java SE.

Les mêmes mécanismes peuvent s'appliquer sur les beans produits par les méthodes productrices, via les attributs `init-method` et `destroy-method` de l'annotation `@Bean`.

L’injection de dépendances

Spring est capable de procéder à l’injection de dépendances, permettant ainsi de mettre en place l’inversion du contrôle (IOC).

Cela s’applique bien sûr à tous les beans Spring (`@Component` et dérivés de `@Component`), mais pas seulement.

En effet, même un objet qui n’est pas une bean Spring (une entité par exemple) peut se profiter de l’injection de dépendances.

3 prérequis :

- Annoter la classe de configuration par `@EnableLoadTimeWeaving` et par `@SpringConfigured` (cette dernière est fournie avec `spring-aspect`)
- Annoter la classe qui doit bénéficier de l’injection de dépendance par `@Configurable`
- Ajouter au lancement de la JVM l’argument `-javaagent=/path/to/spring-instrument.jar`

Ainsi Spring pourra intervenir même si la classe est instanciée par l’opérateur `new`

L'injection de dépendances: références

Pour les références vers d'autres beans : `@Autowired`

Peut être posé sur :

- un champ
- un constructeur
- un setter

Le rapprochement se fait par type.

Un attribut `required` permet de préciser si l'injection est obligatoire ou non. La valeur par défaut de cet attribut est `true`, si bien que Spring lève une exception si l'injection ne peut être réalisé (faute de candidat à l'injection disponible ou en cas d'ambiguité).

Ne sont injectables par `@Autowired` que des références vers d'autres beans que Spring connait par ailleurs.

Echec de l'injection ⇒ *Failed to load application context*.

Sont aussi injectables dans n'importe quel bean Spring :

- l'`ApplicationContext`
- l'`Environment` (essentiellement utile pour accéder aux propriétés référencée par l'annotation `@PropertySource` sur la classe de configuration).

```
@Component @Scope("singleton")
public class ShopServiceImpl implements ShopService {

    @Autowired
    private ShoppingCartManager shoppingCartManager;

    @Autowired
    private ProductManager productManager;

    @Autowired
    private List<ShippingManager> shippingManagers;

    @Autowired
    private ClientManager clientManager;

    @Autowired
    private PaymentManager paymentManager;
    // méthodes
}
```

Si plusieurs beans sont candidats à la même injection (pour un point d’injection donné) et que le point d’injection n’est pas un agrégat (`List`, `Set`, `Map`, `Array`) cela conduit à un conflit.

Pour lever l’ambigüité il est possible d’utiliser l’annotation `@Primary` sur un des candidats à l’injection pour indiquer qu’il doit être choisi au détriment des autres candidats à l’injection.

Enfin il est possible d'utiliser des `@Qualifier` :

Ainsi l'injection ne sera résolu non plus seulement d'après le type du point d'injection mais aussi d'après le *qualifier* qui le décore.

Ainsi si dans l'exemple le point d'injection

```
@Autowired
private List<ShippingManager> shippingManagers;
```

Avait été :

```
@Autowired @Qualifer("GARANTEED_DELIVERY")
private List<ShippingManager> shippingManagers;
```

Alors seulement les beans de type `ShippingManager` annotés par `@Qualifer("GARANTEED_DELIVERY")` aurait été injectés.

Ainsi l'injection peut être résolu non plus seulement d'après une correspondance basée sur le types mais aussi d'après une méta-information métier (ici l'idée qu'un transporteur garantit le délai de livraison).

L’injection de dépendances : valeurs

Pour les valeurs : `@Value("${propertyName}")`

Cela suppose qu’un fichier .properties soient inscrit dans le contexte, car Spring va rapprocher le nom de la propriété (ici `propertyName`) d’une valeur réelle lue dans un fichier *properties*.

Les fichiers de properties peuvent se référencer avec l’annotation : `@PropertySource` sur la classe de configuration.

Le type de la valeur n’est pas contraint : il peut s’agir du type String ou de n’importe quel primitif ou d’un type qui se construit d’après une String : `URL`, `File`, `Class`.

Test d'un bean Spring

Pour tester un objet géré par Spring il suffit de coupler Junit à Spring en utilisant `spring-test` (artifact Maven : spring-test).

Ensuite :

1. Grâce à l'annotation `@Runwith` (Junit 4) ou `@ExtendWith` (Junit 5) : l'exécution du test est déléguée à Spring.
2. Grâce à l'annotation `@ContextConfiguration` : nous précisons sur quelle(s) classe(s) de configuration doit s'appuyer le contexte applicatif mis en place pour le test

```
@RunWith(SpringRunner.class) // si Junit 4 et Spring >=4
@ExtendWith(SpringExtension.class) // si Junit 5 et Spring >=5
@ContextConfiguration(classes={MyApplication.class})
public class MyTest {
    private @Autowired ShopServiceImpl service;

    // méthode de tests
}
```

Enfin, l'annotation `@TestPropertySource` permet de référencer un fichier de propriétés, ces dernières viendront compléter ou remplacer celles mentionnées dans le fichier référencé par l'annotation `@PropertySource` sur la classe de configuration

Par défaut Spring test gardera le contexte disponible jusqu'à la fin des tests afin que d'autres tests ayant la même annotation `@ContextConfiguration` puisse réutiliser le contexte.

Si nous souhaitons limiter l'utilisation du contexte au méthodes de la classe de tests, nous pouvons utiliser l'annotation `@DirtiesContext`

Avec `@DirtiesContext`, chaque classe de test disposera d'un `applicationContext` dédié, créé avant le premier test de la classe et fermé après le dernier test de la classe.

Nous gagnons alors en isolation mais perdons en performance si plusieurs classes de tests ont la même `@ContextConfiguration`

Services techniques

Spring propose un ensemble de services techniques prêts à l'emploi pour les besoins courants de nos applications :

1. Pub / sub (fourni par spring-context)
2. Planification (fourni par spring-context).
3. Supervision via JMX (fourni par spring-context)
4. Intercepteurs
    - Cache (fourni par spring-context)
    - Transaction (fourni par spring-tx)
    - AOP (fourni par spring-aspect, qui utilise AspectJ)

Pub / Sub

L'`ApplicationContext` propose une méthode `publishEvent` permettant de publier une instance d'un objet en tant qu'évenement dans l'application.

Ainsi nous disposons d'un moyen simple d'implémenter un mécanisme de publication / souscription sans avoir recours à un *broker* externe tel que JMS.

Bien sûr ce mécanisme est interne à l'application : ne peuvent recevoir l'évènement publié que des méthodes des *beans* Spring de l'application.

L'évènement peut-être de n'importe quel type.

Bien sûr il s'agit de publier un objet qui contient des informations à traiter (*value object*), et pas un composant métier.

```
public class OrderEvent {
    int shoppingCartId; // valorisé avant la publication
    Invoice invoice; // valorisé suite au traitement par un listener
    List<Item> suggestions; // valorisé suite au traitement par un listener
    // getters & setters
}
```

Recevrons l'évènement toutes les méthodes annotées par `@EventListener`.

Ces méthodes peuvent recevoir l'évènement en paramètre en vue de faire un traitement sur la base des informations qu'il contient.

```
@EventListener
public void doSomethingWithEvent(OrderEvent event)
// traitement de l'évènement
}
```

Ou, si l'on souhaite être explicite

```
@EventListener(classes=OrderEvent.class)
public void doSomethingWithEvent(OrderEvent event)
// traitement de l'évènement
}
```

Bien sûr nous souhaitons pouvoir choisir si les `EventListener` sont appelés de manière synchrone (et donc bloquante mais éventuellement ordonnée) ou en parallèle (auquel cas la question de l'ordre n'a pas de sens).

Par défaut l'invocation se fait de manière bloquante (les *listeners* sont appelés les uns après les autres) et l'ordre dans lequel intervient un `@EventListener` peut être précisé avec l'annotation `@Order`.

L'utilisation de cette annotation est toutefois délicate car les *event listeners* ne se connaissent pas forcément et il n'est pas évident pour un *event listener* de choisir sa position dans la liste des *event listeners* qui recevront l'évènement.

Si un `@EventListener` doit être appelé de manière non bloquante, il faut l'annoter par `@Async`. La publication de l'évènement est alors de type 'fire and forget', il n'est pas possible d'attendre que les listener ait fini de traiter l'évènement.

Si l'on utilise `@Async` il faut activer le support des invocations asynchrones avec l'annotation `@EnableAsync` sur la classe de configuration.

Planification

Puisque Spring « s'occupe de nos composants », il est capable de déclencher l'invocation des méthodes suivant une planification.

Cela se fait en décorant les méthodes concernées par `@Scheduled`.

Celle-ci propose au développeur de définir les arguments suivants :

- `cron`
- `fixedDelay`
- `fixedRate`

Si l'on a choisit une configuration par annotation, l'activation des invocations planifiées se fait en décorant la classe de configuration par `@EnableScheduling`

Supervision

JMX permet de superviser une application et d'invoquer les méthodes de certaines classes (les MBeans) depuis une console (la Jconsole par exemple).

Cela permet de superviser les objets qui s'y prêtent (activation ou désactivation d'un logger, réglage d'un timeout, etc...).

L'enjeu est d'exposer un composant Spring en tant que MBean

Il suffit pour ça d'annoter les composants que l'on veut exposer avec les annotations

- `@ManagedResource` (sur la classe)
- `@ManagedAttribute` (sur les getter et setters exposés)
- `@ManagedOperation` (sur les méthodes exposées)

et d'annoter la classe de configuration par `@EnableMBeanExport`

Interceptions

Les interceteurs (transactions, cache, AOP) correspondent à l'application du pattern *proxy* : Spring inscrit dans le contexte un *proxy* vers notre objet afin de contrôler l'invocation des méthodes et apporter une valeur ajoutée technique au code métier.

Il est possible de préciser si les *proxies* doivent se faire par association avec le *bean* (la cible de l'interception) ou par héritage du *bean*.

Dans le premier cas le *bean* doit implémenter une ou plusieurs interfaces que le *proxy* pourra implémenter à son tour : le *proxy* « se fait passer » pour le *bean* en en implémentant les mêmes interfaces.

Dans le second cas le *proxy* sera une classe fille du *bean*.

Fonctionnement par défaut : *proxy* par association si le *bean* implémente une interface, *proxy* par héritage sinon.

Il est toutefois possible de forcer l'application de *proxy* par héritage en valorisant à `true` l'attribut `proxyTargetClass` des annotations `@Enable*` (voir pages suivantes)

Cache

Spring propose un mécanisme de mise en cache de ce que retourne les méthodes de nos *beans*.

Pour une méthode donnée, nous observons alors le comportement suivant :

- À la première invocation : la méthode est appelée et ce qu'elle retourne est mise en cache par Spring (plus précisément : par le *proxy* qui se trouve *devant* notre objet).
- Lors des invocations suivantes, le *proxy* retourne le résultat gardé en cache.

Pour chaque entrée dans le cache :

- La clé est constituée des arguments reçus lors de l'invocation.
- La valeur est le résultat retourné suite à l'invocation de la méthode.

L'activation du cache se fait par l'annotation `@EnableCaching` sur la classe de configuration.

Il y a un prérequis : la présence d'un bean de type `CacheManager` dans le contexte applicatif.

`CacheManager` est une interface, Spring propose une implémentation : `ConcurrentMapCacheManager`.

Cette implémentation est limitée dans le sens où elle ne permet pas une réplication du cache entre les différents nœuds d'un *cluster*.

Pour disposer d'un cache répliqué il faudra coupler Spring avec une librairie dédiée, Hazelcast par exemple.

Les caches doivent être nommés, par exemple : *books*, *authors*, *publishers* pour une application de gestion d'une librairie.

Chaque cache doit être prise en charge par un `CacheManager`, mais bien sûr un même `CacheManager` peut gérer plusieurs caches.

```
@Bean
public CacheManager cacheManager()
    return new ConcurrentMapCacheManager("books", "authors", "publishers");
}
```

Les annotations à connaitre :

- `@Cacheable` : pour indiquer à Spring que ce que retourne une méthode peut être mise en cache.
- `@CacheEvict` : pour indiquer à Spring que l'invocation d'une méthode doit invalider le cache.
- `@CachePut` : pour indiquer à Spring que ce que retourne une méthode doit être mis dans le cache.

Les principaux attributs de ces annotations :

- `cacheNames` : le(s) caches où se trouve éventuellement l'entrée associée à la méthode appelée.
- `key` : la clé de l'entrée associée à la méthode appelée (par défaut : la valeur de chacun des paramètres de la méthode).

```
@Component
public class BookService{
    // soit Item une classe dont les propriétés sont id et name
    @Cacheable(cacheNames= "books", key="#id")
    public Book getById(int id){
        /* code */
    }

    @CacheEvict(cacheNames= "books", key="#id")
    public void delete(int id){
        /* code */
    }

    @CachePut(cacheNames= "books", key="#book.id")
    public Book update(Book book){
        /* code */
    }
}
```

Ici nous indiquons explicitement pour les deux premières méthodes que la clé est la valeur passée en argument pour le paramètre *id*.

Le comportement par défaut (clé déterminée à partir de la valeur des paramètres) aurait convenu et le comportement aurait été le même qu'en spécifiant `key="#id"` à chaque fois.

Transactions

Spring peut détecter la présence de l'annotation `@Transactional` sur nos méthodes et appliquer une gestion transactionnelle autour d'elles lorsque celles-ci sont invoquées.

Pour cela Spring inscrira dans le contexte un *proxy* vers notre bean plutôt que notre bean, c'est le *proxy* qui recevra les invocations.

Le fonctionnement par défaut :

- Si une transaction est déjà en cours, le *proxy* invoque la méthode appelée dans la transaction en cours.
- Si une transaction n'est pas déjà en cours, le *proxy* :
  1. ouvre une transaction,
  2. invoque la méthode appelée dans un bloc `try/catch`
  3. *commit* de la transaction si la méthode ne lève pas d'exception, *rollback* sinon (si l'exception levée est une `RuntimeException`).

Cela correspond au niveau de propagation *REQUIRED*, nous verrons que d'autres niveaux de propagation existent.

L'activation se fait l'ajout de l'annotation `@EnableTransactionManagement` sur la classe de configuration. L'attribut `proxyTargetClass` permet de préciser si le *proxy* doit se faire par association ou par héritage (`proxyTargetClass = true` : *proxy* par héritage).

Prérequis : la présence dans l'`ApplicationContext` d'un *bean* de type `org.springframework.transaction.PlatformTransactionManager`

AOP

La programmation par aspect conduit à

1. identifier les traitements techniques transverses (log, mesure de performances, règles de sécurité, transaction). Nous parlons de *cross cuting concerns*.
2. sortir de notre code métier le code de ces traitements techniques transverses.
3. confier à Spring le soin d'appliquer ces traitements techniques transverses avant/après/autour des méthodes de nos *beans*.

Les mots clés :

- **advice** : le traitement technique proprement dit (typiquement une méthode dans une classe) à appliquer avant/après/autour des joinpoints correspondant à un pointcut.
- **joinpoint** : un point identifiable dans l'exécution d'un programme. Exemple une méthode, un constructeur.
- **pointcut** : la règle qui définit sur quels *joinpoints* doit s'appliquer l'*advice*.
- **target** : cible de l'interception. Toutes les classes dont une méthode (un *joinpoint*) correspond au pointcut seront cibles d'interception.
- **proxy** : ce que Spring fabrique et inscrit dans l'applicationContext en mémoire en lieu et place du *target*.

Ainsi l'appelant de manipule pas une instance de l'objet appelé, mais un *proxy* chargé d'appliquer en plus les mécanismes d'interception.

C'est transparent pour l'appelant car le *proxy* est du même type que le *target*, soit parcequ'il en hérite (*proxy* par héritage), soit parcequ'il implémente les mêmes interfaces (*proxy* par association).

En sortant ces mécanismes techniques et transverses du code métier il est facile d'étendre/réduire leur champ d'application (modification du *pointcut*), de les désactiver ou de les modifier.

Exemple de traitement techniques:

- mesurer le temps d'exécution des méthodes,
- entourer l'invocation des méthodes d'une transaction,
- restreindre l'invocation des méthodes à un certain groupe d'utilisateurs…

Certains de ces mécanismes sont déjà prévus par Spring (sécurité, cache, transaction), cf. pages précédentes.

Pour les autres, nous sommes responsable de la création de l'*advice*.

Attention, cela induit une dissémination du code puisqu'il faut savoir que ce mécanisme d'interception est appliqué pour avoir les idées claires sur la chaine d'exécution du programme.

Dans un premier temps nous définissons l'*advice*, c'est-à-dire le mécanisme technique transverse qui doit s'exécuter avant et/où après notre code métier :

```
@Component
public class MyAdvice {
    // injections éventuelles, comme pour tous beans spring

    public Object intercept(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("before invoke");
        // pjp permet d'identifier l'appelant, la méthode appelée...
        Object ret = pjp.proceed();
        System.out.println("after invoke");
        return ret ;
    }
}
```

`pjp` correspond au *joinpoint* en cours d'exécution (typiquement l'invocation d'une méthode), il est ainsi possible d'avoir le contrôle de l'invocation.

Il s'agit maintenant de déclarer `MyAdvice` comme un aspect et de préciser avec un *pointcut* les *jointpoints* qui doivent faire l'objet de l'interception par la méthode `intercept`

```
@Component
@Aspect // annotation aspectj
public class MyAdvice {
    // injections éventuelles, comme pour tous beans spring
    @Around("execution(* com.acme.dao..*.*(..))") ① ② ③ ④ ⑤ ⑥
    public Object intercept(ProceedingJoinPoint pjp) throws Throwable {
        System.out.printlln("before invoke");
        // pjp permet d'identifier l'appelant, la méthode appelée...
        Object ret = pjp.proceed();
        System.out.println("after invoke");
        return ret ;
    }
}
```

① `*` : peu importe le type de retour des méthodes

② `com.acme.dao` : les méthodes doivent se trouver dans des classes qui se trouvent dans le *package* `com.acme.dao`

③ `..` : les sous-packages sont aussi concernés

④ `*` : peu importe le nom de la classe

⑤ `*` : peu importe le nom de la méthode

⑥ `(..)` : peu importe les paramètres

L'activation des aspects se fait en décorant la classe de configuration par `@EnableAspectJAutoProxy`.

A nouveau l'attribut `proxyTargetClass` permet de préciser si le *proxy* doit se faire par association ou par héritage (proxyTargetClass=true : *proxy* par héritage)

Synthèse

Si l'on reprend l'anatomie d'une classe (déclaration de la classe, déclaration des champs, déclaration des méthodes).

- Nos besoin sur la classe : déclarer son cycle de vie. Réponse de Spring : les annotations `@Component` (et dérivés : `@Service`, `@Repository`, etc…) et l'annotation `@Scope` (`@Scope("prototype")` ou `@Scope("singleton")`).
- Nos besoins sur les champs : déclarer les injections à réaliser. Réponse de Spring : les annotations `@Autowired` et `@Value`
- Nos besoins sur les méthodes : des *callbacks* et des interceptions, de la supervision, des invocations planifiées. Réponses de Spring :
  - les annotations `@PostConstruct` et `@PreDestroy` pour les callbacks
  - les annotations `@Aspect` et `@Around` pour les interceptions personnalisées.
  - l'annotation `@Transactional` pour les transactions.
  - les annotations `@Cacheable`, `@CachePut`, `@CacheEvict` pour le cache.
  - les annotations `@ManagedOperation` et `@ManagedAttribute` pour la supervision.
  - l'annotation `@Schedule` pour les invocations planifiées.

Spring propose en plus un bus évènementiel (`applicationContext.publishEvent`, `@EventListener`).

Nous avons donc une standardisation des designs patterns suivants :

- *prototype* et singleton pour le cycle de vie
- *strategy* pour l'injection de dépendances
- *proxy* pour les interceptions
- *observer* pour la logique évènementielle
- *factory*: Spring est la *factory* des composants de notre application

En tant que conteneur, Spring va, pour chaque classe que nous souhaitons lui confier :

1. Trouver le constructeur à appeler
2. Invoquer le constructeur, en procédant à l'injection par constructeur si le constructeur attend des arguments
3. Procéder aux injections sur les champs et les *setters* annotés par `@Autowired` ou `@Value`
4. Invoquer la/les méthodes annotée(s) par `@PostConstruct`
5. Examiner si des méthodes doivent voir leurs invocations accompagnées d'un traitement technique (transaction, cache, aop).

Si oui : création d'un *proxy* et inscription de de celui-ci dans l'`ApplicationContext`

Si non : inscription de l'instance telle quelle dans l'`ApplicationContext`

Le *bean* inscrit dans l'`ApplicationContext` sera ensuite :

- éventuellement exposé en JMX si des méthodes sont annotées par `@ManagedOperation` (ou `@ManagedAttribute` pour les accesseurs) et si la classe est annotée par `@ManagedResource`.
- L'objet d'invocations planifiées si des méthodes sont annotées par `@Scheduled`.

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 stockée 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 forcément 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.

Deux stratégies possibles :

- fetch="**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= "**lazy**" : la relation est matérialisée quand on y accède.
  - avantage : requêtes sql plus légères.
  - inconvénient : graph d'objet moins riche.

Le *lazy loading* peut être précisé pour toutes les relations (`@ManyToOne`, `@ElementCollection`, etc...) via l'attribut `fetch`. Les valeurs par défaut :

- EAGER pour les `@ManyToOne` et les `@Embedded`.
- 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 `@Cacheable`

favoriser le mode EAGER pour les relations vers des classes dont on peut mettre les instances en cache niveau 2.

toujours utiliser LAZY si la relation porte vers une classe dont les instances ne sont pas en cache niveau 2

Dans notre mapping, il aurait été utile de choisir `fetch=FetchType.LAZY` pour la relation `@ManyToOne` de `Auction` vers `Member`

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
        <properties> ④
        <!-
            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 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("age", 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` :

```
@Component
public class PersonManager {

    @javax.persistence.PersistenceContext
    private javax.persistence.EntityManager em;

    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 la notion d'objet attaché / détaché.
- Comprendre l'alignement entre le cycle de vie de l'EntityManager et celui de la transaction.
- Comprendre ce qu'implique la méthode getResultList() :
  - une ou plusieurs requête dans la base de données,
  - boucle sur un resultset
  - chargement en mémoire de plusieurs instances (quid de la profondeur du graph d'objet !?)
  - Persistance et mapping O/R : JPA

Couplage Spring et JPA

- Principes
- Configuration
- Gestion des transactions
- Introduction à Spring Data

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

Principes

Spring peut jouer son rôle de conteneur en

- gérant le cycle de vie de l'`EntityManager`
- appliquant des transactions autours des méthodes de nos *beans*.

Objectif :

Disposer de l'`EntityManager` de la transaction en cours dans n'importe quel bean Spring :

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

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
    }
}
```

Configuration

Pour atteindre cet objectif, 2 *beans* doivent être présents 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.

L'EntityManagerFactory

Si un fichier `META-INF/persistence.xml` existe dans le *classpath* :

```
@Configuration()
public class MyApplication {
    @Bean
    public EntityManagerFactory emf(){
        return Persistence.createEntitytManagerFactory("pu1");
    }
}
```

Sans fichier `META-INF/persistence.xml` :

```
@Configuration
public class MyApplication {
    @Bean
    public FactoryBean<EntityManagerFactory> emf(){
        LocalContainerEntityManagerFactoryBean _emf = new LocalContainerEntityManagerFactoryBean();
        _emf.setPersistenceProviderClass(HibernatePersistenceProvider.class);
        Properties jpaProperties = new Properties();
        // ajout des propriétés qui auraient autrement été définies dans le fichier persistence.xml
        _emf.setJpaProperties(props);
        return _emf;
    }
}
```

Le PlatformTransactionManager

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

Quelques implémentations fournies par Spring :

- `org.springframework.transaction.jpa.JpaTransactionManager`
- `org.springframework.transaction.jta.JtaTransactionManager`
- `org.springframework.transaction.orm.hibernate5.HibernateTransactionManager`

Exemple, avec `JpaTransactionManager` :

```
@Configuration
public class MyApplication {

    @Bean
    public EntityManagerFactory entityManagerFactory() {
        return Persistence.createEntitytManagerFactory("pu1");
    }

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

C'est parceque l'`EntityManagerFactory` et le `PlatformTransactionManager` sont liés que Spring pourra ouvrir un `EntityManager` au début de chaque transaction et le fermer au moment du *commit* ou du *rollback*.

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);
}
```

Gestion des transactions

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 évoquer 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'autre : 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.

Application des transactions par AOP

L'application des transactions via des aspects peut se faire ainsi :

```
@Component @Aspect
public class TransactionalAspect{

    @Autowired
    private PlatformtransactionManager txManager;

    @Around("execution(* eg.domain.service.*.*(..))")
    public Object txRequiresNew(ProceedingJoinpoint pjp) throws Throwable{
        DefaultTransactionDefinition txDef = new DefaultTransactionDefinition();
        txDef.setPropagationBehavior(Propagation.REQUIRES_NEW);
        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 intrusion dans le code (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
- nécessite de savoir lire et écrire les *pointcuts*
- risque qu'aucune transaction ne soit appliquée si le *pointcut* est mal écrit

Application des transactions par annotations

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 `org.springframework.transaction.annotation.Transactional`

Cette annotation reprend sous forme d'attribut les propriété vues précédemment pour le la classe `DefaultTransactionDefinition` :

- niveau de propagation (*REQUIRED*, *MANDATORY*, etc...)
- niveau d'isolation
- transaction en lecture seule.
- classes des exceptions qui doivent donner lieu à un *rollback* si elles sont levées
- classes des exceptions qui ne doivent pas donner lieu à un *rollback* si elles sont levées

Avantage : possibilité de préciser le niveau d'isolation et d'activer un mode `readOnly` pour les transactions en lecture seule.

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

L'annotation `javax.transaction.Transactional`

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

Les attributs de cette annotation :

- niveau de propagation (*REQUIRED*, *MANDATORY*, etc...)
- classes des exceptions qui doivent donner lieu à un *rollback* si elles sont levées
- 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.

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 TransactionSynchronizationAdapter(){
    @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`

Introduction à Spring Data

Principes

Spring Data est un ensemble de librairies qui facilitent la persistance de nos objets dans une base de données (SQL ou NoSQL).

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

- spring-data-jpa
- spring-data-cassandra
- spring-data-redis
- spring-data-mongo
- spring-data-elasticsearch

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>2.0.8.RELEASE</version>
</dependency>
```

Dans certains cas (spring-data-cassandra par exemple), Spring data propose

- des annotations pour définir un *mapping* entre les entités et la base de données.
- des *repositories* prêt à l'emploi pour réaliser les opérations CRUD.

Notons que pour certaines bases de données (Cassandra, Redis, Mongo), Spring Data propose des *repositories* réactifs car les *drivers* de connexion le permettent. Ce n'est pas le cas de spring-data-jpa puisque JDBC (sur lequel s'appuie JPA) ne propose pas de connexions réactives.

Dans le cas de JPA, Spring s'appuie sur un mapping JPA existant et ne propose donc *que* les *repositories* prêts à l'emploi. En cela, Spring n'implémente pas JPA mais facilite son utilisation.

Spring Data JPA

L'activation de Spring Data se fait avec l'annotation `@EnableJpaRepositories` sur la classe de configuration :

```
@Configuration
@EnableJpaRepositories
public class ApplicationConfig{
    // code
}
```

Prérequis : la présence dans l'`ApplicationContext` de deux beans :

- 1 `EntityManagerFactory`
- 1 `PlatformTransactionManager`

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

```
@Configuration @EnableJpaRepositories
public class ApplicationConfig{
    @Bean
    public EntityManagerFactory entityManagerFactory(){
        return Persistence.createEntitytManagerFactory("pu1");
    }

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

Si nos *beans* sont nommées différemment nous pouvons écrire :

```
@Configuration
@EnableJpaRepositories(
    entityManagerFactoryRef= "emf",
    transactionManagerRef ="txManager"
)
public class ApplicationConfig{
    @Bean
    public EntityManagerFactory emf(){
        return Persistence.createEntitytManagerFactory("pu1");
    }

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

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 extends JpaRepository<Person, Integer>
{
}
```

Nous pouvons ensuite en disposer dans nos beans Spring par injection :

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

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

Quelques unes des méthodes dont nous disposons compte tenu de l'héritage de l'interface `JpaRepository<T, ID>` :

```
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);
```

Bien sûr il s'agit de *raccourcis* vers les méthodes de l'`EntityManager` : `find`, `remove`, `persist`, `merge`

Définitions de requêtes personnalisées

Nous aurons probablement besoin de requêtes spécifiques, exemple : rechercher toutes les personnes dont le nom contient 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 :

- celle 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 extends JpaRepository<Person, Integer>
{
    List<Person> findByLastnameContaining(String lastname);

    Optional<Person> findByUsername(String username);
}
```

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 que le filtre sur la propriété `lastname` utilise l'opérateur `like`.

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

Notons que le type `CompletableFuture` est accepté, dans ce cas la méthode doit être annotée par `@Async` :

```
@Async
CompletableFuture<List<Person>> findByLastnameContaining(String lastname);
```

Deux autres types de paramètres sont supportés :

- `PageRequest`, pour des requêtes paginées.
- `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 résulats.
- `Page<Person>` si nous souhaitons obtenir une partie des résultats (principe des requêtes paginées).

Exemple d'utilisation de `Page` :

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

    Optional<Person> findByUsername(String username);
}
```

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 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);
}
```

Nous disposons des mêmes mécanismes que ceux vus précédemment, mais 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();
```

En réalité *spring-data-jpa* fait tout cela par reflexion.

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>`).

Mais il faut alors que la requête JPA-QL soit adaptée à la possible absence de valeur pour ce paramètre : 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 filtrer 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 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, PageRequest pr);
}
```

Spring Web MVC

- Principes
- Installation
- Spring MVC pour concevoir des API REST
- Spring MVC pour concevoir des IHM

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

Principes

Spring MVC n'est pas seulement un framework orienté *présentation*.

Lorsqu'il recoit une requête HTTP, la réponse qu'il renvoie à l'appelant peut être :

- une vue (lorsqu'il s'agit de présenter une IHM à un utilisateur)
- la représentation d'une ressource (lorsqu'il s'agit d'exposer une API REST).

L'idée centrale est que chaque méthode répond à un couple *path*/*verb*.

Si nous développons une API REST, cette méthode retournera un objet qui sera serialisé (sous une forme xml ou json par exemple)

Si nous développons un IHM, cette méthode retournera une vue à rendre.

Lorsque le *servlet container* reçoit une requête HTTP :

1. Le *servlet container* rapproche l'url la requête entrante de de l'*url-pattern* de la servlet Spring (`DispatcherServlet`) et invoque la méthode `service` de cette dernière (cf. https://en.wikipedia.org/wiki/Java_servlet#Life_cycle_of_a_servlet , points 4 et 5)
2. Cette méthode `service` trouve la méthode à invoquer en rapprochant l'URI et le verbe HTTP des annotations `@RequestMapping` posées sur les méthodes des beans gérés par Spring.
3. La *servlet* Spring invoque la méthode trouvée en 2. Cette méthode dispose de tout ce qui figure dans la reqête (paramètres, *body*, en-têtes…) et sollicite éventuellement la couche métier.

C'est donc l'annotation `@RequestMapping` qui nous permet d'indiquer à Spring quelle méthode doit être invoquée pour chaque requête entrante.

Par exemple :

- `GET` sur `/people` doit invoquer une méthode
- `GET` sur `/people/1` doit invoquer une autre méthode.
- `POST` sur `/people` doit invoquer une troisème méthode
- Etc…

Les deux attributs essentiels de l'annotation `@RequestMapping` : `path` et `method`

Exemple :

- `@RequestMapping(path="people", method=RequestMethod.GET)`
- `@RequestMapping(path="people/{id}", method=RequestMethod.GET)`
- `@RequestMapping(path="people", method=RequestMethod.POST)`

Il existe des annotations *raccourcis* (aussi appelées annotations composites) : `@GetMapping`, `@PostMapping`, `@PutMapping`, etc...

Ainsi :

- `@RequestMapping(path="people", method=RequestMethod.GET)` peut s'écrire `@GetMapping(path="people")`
- `@RequestMapping(path="people/{id}", method=RequestMethod.GET)` peut s'écrire `@GetMapping(path="people/{id}")`
- `@RequestMapping(path="people", method=RequestMethod.POST)` peut s'écrire `@PostMapping(path="people")`

Et, puisque l'attribut `path` et l'attribut `value` sont alias l'un de l'autre :

- `@GetMapping(path="people")` peut s'écrire `@GetMapping("people")`
- `@GetMapping(path="people/{id}")` peut s'écrire `@GetMapping("people/{id}")`
- `@PostMapping(path="people")` peut s'écrire `@PostMapping("people")`

Spring peut transmettre de nombreux paramètres à la méthode responsable du traitement de la requête :

- Tous les paramètres disponibles dans la requête grâce à l'annotation `@RequestParam`
- Tous les en-têtes de la requête grâce à l'annotation `@RequestHeader`
- Tous les attributs de la session grâce à l'annotation `@SessionAttribute`
- Tous les valeurs des cookies grâce à l'annotation `@CookieValue`
- Tous les éléments variables du *path* grâce à l'annotation `@PathVariable`
- Un objet reconstruit d'après les paramètres de la requête

Dans le cas d'une API REST nous pouvons aussi disposer du `body` de la requête déserialisé en objet grâce à l'annotation `@RequestBody`

Dans le cas d'une IHM, nous pouvons aussi obtenir un *objet de laison* pour transmettre le modèle à la vue. Plusieurs types sont supportés : `Map`, `Model`, `ModelMap`

Bien sûr nous pouvons aussi disposer de la requête et de la réponse, via des paramètres de type `HttpServletRequest` et `HttpServletResponse`

Spring analyse la signature de la méthode et l'invoque en lui transmettant ce dont elle à besoin.

Tous ces paramètres sont facultatifs, les méthodes annotées par `@RequestMapping` peuvent donc avoir des signatures adaptés à nos besoins.

La méthode responsable du traitement de la requête peut retourner :

Pour les API REST :

- `void`
- un objet (qui sera sérialisé et renvoyé à l'appelant)
- une `ResponseEntity` (qui précise le *body*, le statut et les en-têtes de la réponse à retourner à l'appelant).

Pour une IHM :

- Une `String`, pour déclencher le rendue d'une vue ou une redirection.
- un `ModelAndView`, contenant les éléments nécessaires au rendu de la vue :
  - le nom de la vue d'une part
  - les éléments dynamique qu'elle est supposée afficher d'autre part.

Installation

Une application web basée sur Spring MVC doit avoir sa configuration spécifique.

Cela peut prendre la forme d'une classe de configuration.

Celle-ci doit implémenter `WebMvcConfigurer` et être annotée par `@EnableWebMvc`

```
@Configuration @EnableWebMvc @ComponentScan
public class WebConfiguration implements WebMvcConfigurer
{
    // code
}
```

Spring va reconnaitre que cette classe de configuration implémente `WebMvcConfigurer` et en invoquera les méthodes.

Ainsi nous sommes guidées dans la configuration de Spring MVC : il suffit d'implémenter les méthodes définies dans l'interface `WebMvcConfigurer`

Spring 4 fournit une classe abstraite qui propose une implémentation par défaut de toutes ces méthodes : `WebMvcConfigurerAdapter`

```
@Configuration @EnableWebMvc @ComponentScan
public class WebConfiguration extends WebMvcConfigurerAdapter
{
    // code
}
```

Cette classe n'a plus de raison d'être utilisé à partir de Spring 5 puisque les implémentations par défaut figurent directement dans l'interface `WebMvcConfigurer`

Il faut ensuite déclarer :

- Un `ServletContextListener` qui va charger notre contexte applicatif. Ce `listener` est le `ContextLoaderListener` fourni par Spring.
- Une `HttpServlet` qui va accueillir les requêtes entrantes et faire le lien avec nos controleurs. Cette servlet est la `DispatcherServlet` fournie par Spring.

Ces deux déclaration peuvent se faire en Java, en implémentant l'interface `WebApplicationInitialzer` et en redéfinissant la méthode `onStartup`.

La méthode `onStartup` reçoit en paramètre le `servletContext` (`javax.servlet.ServletContext`) et peut ainsi profiter des méthodes :

- `addListener`
- `addServlet`

```
public class WebAppInitializer implements WebApplicationInitializer{
    public void onStartup(javax.servlet.ServletContext servletContext) throws ServletException {
        AnnotationConfigWebApplicationContext appCtx = new AnnotationConfigWebApplicationContext(); ①
        appCtx.register(ApplicationConfig.class);
        servletContext.addListener(new ContextLoaderListener(appCtx));②

        AnnotationConfigWebApplicationContext webCtx = new AnnotationConfigWebApplicationContext();③
        webCtx.register(WebConfiguration.class);
        HttpServlet springServlet = new DispatcherServlet(webCtx); ④
        javax.servlet.ServletRegistration.Dynamic d = servletContext.addServlet("springServlet", springServlet);⑤
        d.addMapping("/");
    }
}
```

① Définition du contexte applicatif

② Activation du *listener* pour charger le contexte applicatif au démarrage de l'application web

③ Définition du contexte web

④ Instanciation de la *servlet* Spring

⑤ Inscription de la servlet Spring auprès du *servlet container*

Spring MVC pour concevoir des API REST

REST : principes généraux

REST fait suite à la thèse de doctorat de Roy Fielding en 2000. https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

Le point de vue : les applications ont avantage à dialoguer entre elles à travers les ressources qu'elles exposent.

Quelques exemples de ressources : une formation, la fiche de paie d'un salarié, un livre dans une bibliothèque, un passager dans un avion, etc…

Une organisation doit faire un travail de conception :

- De quelles ressources est elle responsable ?
- Comment les représenter ?
- Comment permettre à des applications appelantes d'interagir avec elles ?

Répondre aux deux premières questions conduise à travailler sur les **URI** et sur les **mediaType**.

Uniform Resource Identifier (URI)

Les identifiants des ressources dont notre organisation est responsable. Exemple : `users/1`, `users/8`, `users/8/preferences`...

MediaType

La manière dont une ressource est représentée : ses propriétés et les liens avec d'autres ressources. Le *mediaType* designe au minimum un format d'échange mais peut aussi préciser les propriétés qui qualifient la ressource.

Ainsi un utilisateur pourrait avoir comme `URI` *users/{id}* et être représenté ainsi :

```
{
    "id":8,
    "firstname":"John",
    "lastname":"Doe",
    "mailAddress":"jdoe@acme.com",
    "links":[
        {"rel":"friends", "href":"/users/8/friends"},
        {"rel":"preferences", "href":"/users/8/preferences"}
    ]
}
```

Cette idée de liens conduit à appliquer les bases du web à des ressources.

On appelle cela HATEOAS (*hypermedia as the engine of application state*).

Cela rend l'API découvrable : un client ne maintiendra pas dans son code l'uri `users/{id}/friends` mais suivra la relation `friends`.

REST et HTTP

REST n’est pas lié à priori à HTTP, mais HTTP va permettre à des applications appelantes d’interagir concrètement avec nos ressources.

Cela suppose une bonne compréhension des points suivants :

- Verbes
- En-têtes
- Statuts

Les verbes

GET

une requête `GET` porte sur une ressource existante et permet d'en obtenir sa représentation sans la modifier. Exemple : `GET` sur `/users`, GET sur `/users/1`.

PUT

une requête `PUT` porte sur une ressource (qui peut ne pas exister), le *body* de la requête contient la représentation complète de la ressource. Ainsi un `PUT` sur `/users/1` doit contenir dans le *body* de la requête la représentation complète de l'utilisateur n°1.

PATCH

une requête `PATCH` porte sur une ressource existante, le *body* de la requête contient des informations permettant de mettre à jour la ressource.

DELETE

une requête `DELETE` porte sur une ressource existante et conduit à la supprimer. Exemple : `DELETE` sur `/users/1`

POST

une requête `POST` contient des informations qui permettent de faire un traitement sur une ressource. Exemple : `POST` sur `/users` pour créer un utilisateur.

Les statuts

Les principaux status 2XX

Les statuts 2XX correspondent à un succès :

OK (200)
: signifie que la requête a été traité avec succès. Essentiellement utilisée pour les requêtes GET

CREATED (201)
: la requête a été traitée avec succès et une nouvelle ressource a été créée (typiquement suite à un `POST`). L'`URI` de la ressource créée doit figurer dans l'en-tête `location` de la réponse.

ACCEPTED (202)
: la requête a été acceptée mais il n'est pas possible de confirmer qu'elle a été traitée. Ce statut est utilisé si l'acceptation de la requête conduit à lancer un traitement asynchrone (*batch* par exemple).

NO CONTENT (204)
: la requête a été traitée avec succès mais aucune information n'est retournée dans le *body* dela réponse. Un statut 204 est alors préférable à un statut 200.

PARTIAL CONTENT (206)
: seule une partie du contenu demandée est retournée (utile pour une requête paginée). L'en-tête *range* doit alors préciser à quelle "plage" correspond le *body* de la réponse.

Les principaux statuts 4XX

Les statuts 4XX correspondent à une erreur dans la requête :

BAD REQUEST (400)
: la requête entrante est invalide (problème sur le *body* par exemple).

NOT_AUTHORIZED (401) et FORBIDDEN (403)
: pour signaler l'échec de l'authentification (401) ou de l'autorisation (403).

NOT FOUND (404)
: la resource n'a pas été trouvée.

CONFLICT (409)
: la requête rentre en conflit avec l'état de la ressource.

Les en-têtes

Les en-têtes à connaître côté requête :

Accept

les *media type* acceptés par le client. C'est cet en-tête que le serveur va lire pour choisir le format de représentation de la ressource, l'*accept* de la requête va donc conditionner le *content-type* du *body* de la réponse.

Content-type

le *media type* du *body* de la requête.

Range

la plage de résultats demandée (pour la pagination par exemple).

Les en-têtes à connaître côté réponse :

Content-type

le *media type* du *body* de la réponse.

Location

l'`URI` de la ressource nouvellement créée (dans le cas d'un statut CREATED) ou vers lequel on veut rediriger l'appelant (dans le cas d'un *redirect*).

La bonne compréhension des verbes, des statuts et des en-têtes permettra par exemple d'implémenter correctement la création d'un nouvelle ressource :

- Le verbe : `PUT` (si l'URI de la ressource que l'on créée est connu) ou `POST` sinon.
- Les informations fournies dans le *body* de la requêtes sont représentées suivant un *media type* précisé dans l'en-tête *content-type* de la requête entrante. L'en-tête *accept* servira à préciser quels sont les *mediaType* qu'il sait interpréter.
- Le serveur retourne une réponse dans le statut est 201 (CREATED) et précise dans l'en-tête location l'URI de la ressource nouvellement créée. Si la représentation de cette dernière figure dans la réponse alors le format utilisé doit être précisé dans l'en-tête *content-type* de la réponse.

REST peut être mise en œuvre avec de simples *servlets*, mais le développeur aurait alors la responsabilité

- d'analyser chaque requête (verbe, URI, en-tête *accept*) pour comprendre à quelle ressource l'appelant veut accéder.
- de déserialiser le corps de la requête (en cas de `POST` ou `PUT`).
- de serialiser le corps de la réponse

Exemple :

```
@WebServlet(urlPatterns="/users")
public class UserServlet extends HttpServlet {

    private UserService myService; // valorisé dans la méthode init

    private ObjectMapper jackson = new ObjectMapper();

    @Override
    public void doGet(HttpServletRequest req, HttpServletResponse resp){
        String name= req.getParameter("name");
        List<User> users;
        if(req.getHeader("accept").equals("application/json")){
            users = myService.find(name);
            resp.setHeader("content-type" , "application/json");
            resp.setStatus(200);
            jackson.writeValue(resp.getOutputStream(), users);
        } else{
            resp.setStatus(415); // 'media type' not supported
        }
    }
}
```

L'apport de Spring MVC

Spring MVC fournit une abstraction par rapport au traitement technique de la requête et de la réponse.

Ainsi le développeur

- crée son API à base de *beans* Spring
- décore les méthodes de ces *beans* avec l'annotation `@RequestMapping` pour associer chaque couple URI / verbe à une méthode.

La responsabilité de Spring MVC, pour chaque requête http :

- identifier la méthode à invoquer (grâce à l'annotation `@RequestMapping`).
- l'invoquer en lui transmettant ce dont elle a besoin.
- fabriquer une réponse HTTP sur la base de ce qu'elle a retourné.

Exemple :

```
@RestController
public class UserEndpoint{

    @Autowired
    private UserService myService;

    // invoquée suite à un GET sur http://[host:port]/[webapp]/users
    @RequestMapping(path="users", method=RequestMethod.GET) ①
    public List<User> users(@RequestParam String name) {
        return myService.find(name);
    }

    // invoquée suite à un POST sur http://[host:port]/[webapp]/users
    @RequestMapping(path="users", method=RequestMethod.POST) ②
    public ResponseEntity<User> addUser(@RequestBody User user){
        this.myService.save(user);
        URI uri = URI.create("/users/"+user.getId());
        return ResponseEntity.created(uri).body(user);
        // created = http 201
    }

    // suite sur la page suivante
```

① ou `@GetMapping("users")`

② ou `@PostMapping("users")`

```
    // invoquée suite à un GET sur http://[host:port]/[webapp]/users/{id}
    @RequestMapping(path="users/{id}", method=RequestMethod.GET) ①
    public ResponseEntity<User> user(@PathVariable int id) {
        Optional<User> _user = myService.getUser(id);
        if(!_user.isPresent()){
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(_user.get()):
    }

    // invoquée suite à un PUT sur http://[host:port]/[webapp]/users/{id}
    @RequestMapping(path="users/{id}", method=RequestMethod.PUT) ②
    public ResponseEntity<Void> user(@PathVariable int id, @RequestBody User user){
        if( !user.getId().equals(id))
            return ResponseEntity.badRequest().build() ; // HTTP 400
        }
        this.myService.save(user);
        return ResponseEntity.noContent().build(); // HTTP 204
    }
}
```

① ou `@GetMapping("users/{id}")`

② ou `@PutMapping("users/{id}")`

La gestion des exceptions

La gestion des exceptions peut-être centralisée dans une classe annotée par `@RestControllerAdvice` (un dérivé de `@Component`).

Un `@RestControllerAdvice` est un *assistant* des contrôleurs et implémente des mécanismes communs aux `@RestController`:

```
@RestControllerAdvice
public class MyAdvice
{
    @ExceptionHandler
    public ResponseEntity<?> handleException(Exception e) {
        // code
    }
}
```

Validation

Spring se couple avec l'API Bean validation, ainsi il est possible de poser l'annotation `@Valid` sur

- le paramètre correspondant à l'objet reconstruit d'après les paramètres de la requête.
- le paramètre correspondent à l'objet reconstruit d'après le body de la requête.

```
@RequestMapping(path="users", method=RequestMethod.POST)
public ResponseEntity<User> addUser(@RequestBody @Valid User user){
    this.myService.save(user);
    URI uri = URI.create("/users/"+user.getId());
    return ResponseEntity.created(uri).body(user);
    // created = http 201
}
```

Ici Spring vérifiera si les contraintes de validité sur l'objet `user` sont respectées, si ce n'est pas le cas il n'invoquera pas la méthode et renverra une réponse HTTP 400 (*bad request*).

En réalité Spring délègue la validation à une implémentation de *bean validation*, hibernate-validator par exemple.

Pour contrôler le body des réponses HTTP 400 émises suite à un échec de la validation : utilisation d'un `ExceptionHandler` dédié aux erreurs de validation.

En effet, lorsque Spring détecte qu'une instance d'un objet annoté par `@Valid` n'est pas valide, il lève :

- Une `MethodArgumentNotValidException` si l'instance a été reconstruite sur la base du *body* de la requête
- Une `BindException` si l'instance a été reconstruire sur la base des paramètres de la requête.

Pour gérer ces deux cas nous donc devons écrire 2 `@ExceptionHandler`

```
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<?> handler1(MethodArgumentNotValidException exception){
    BindingResult br = exception.getBindingResult();
    List<ValidationError> errors = onValidationFailure(br.getFieldErrors());
    return ResponseEntity.badRequest().body(errors).build();
}

@ExceptionHandler(BindException.class)
public ResponseEntity<?> handler2(BindException exception){
    BindingResult br = exception.getBindingResult();
    List<ValidationError> errors = onValidationFailure(br.getFieldErrors());
    return ResponseEntity.badRequest().body(errors).build();
}

private List<ValidationError> onValidationFailure(List<FieldErrors> fieldErrors){
    return fieldErrors.stream().map(ValiationError::new).collect(Collectors.toList()); ①
}
```

① `ValiationError` serait un type à nous dont les propriétés correspondrait aux informations utiles pour l'appelant : propriété en erreur, valeur rejeté, message d'erreur… Autant d'informations disponible dans le type Spring `FieldError`.

Spring MVC pour concevoir des IHM

Dans le cas d'une application web IHM, il s'agira de présenter une page à un utilisateur.

Le type de retour des méthodes annotées par `@RequestMapping` sera généralement une `String` correspondant au nom de la vue à la vue dont Spring doit déclencher le rendu.

```
return "users/list";
```

Conduit à déclencher le rendu de la vue `users/list`

Ou, si cette chaine de caractère commence par redirect, à la route vers laquelle il faut rediriger l'utilisateur.

```
return "redirect:/users";
```

Conduit à une réponse HTTP dont le statut est 302 dont l'en tête `location` sera http://[host:port]/webapp/users

Le ViewResolver

Dans le premier exemple, *users/list* correspond à une vue.

Le rapprochement entre un nom de vue et un fichier physique (jsp par exemple) se fait par la déclaration d'un `ViewResolver` dans la classe de configuration.

```
@Configuration @EnableWebMvc @ComponentScan
public class MvcConfiguration implements WebMvcConfigurer {①
    @Override
    public void configureViewResolvers(ViewResolverRegistry registry){
        registry.jsp().prefix("/WEB-INF/views/").suffix(".jsp");
        // autres méthodes possibles : groovy(), freemarker()
    }
}
```

① ou `extends WebMvcConfigurerAdapter` avec Spring 4.

Dès lors, `return users/list` revient à déclencher le rendu de la vue `WEB-INF/views/users/list.jsp`

Le 2 ways binding

Le développement d'IHM pose une problématique spécifique: l'échange d'informations avec l'utilisateur.

- Les informations issues de l'application métier doivent être transmises à la vue.
- Les informations qui viennent de la vue (saisie dans un formulaire par exemple) doivent être récupérables dans le contrôleur.

Ce principe s'appelle le **2 ways binding**.

Contrôleur ⇒ vue

L'envoi d'informations à la vue depuis le contrôleur se fait avec une `Map<String, Object>`.

Cela représente le modèle (le M de MVC).

Ce modèle figure en paramètre de méthode de nos contrôleurs, s'y trouve tout ce qui sera accessible pour la vue. A nous de l'alimenter, Spring transmettra ensuite à la vue tout ce que nous y aurons mis.

Spring propose aussi des types qui peuvent jouer le rôle de modèle : `Model` et `ModelMap`. Ces derniers proposent des facilités pour ajouter des éléments au modèle (nom de l'attribut déterminé automatiquement, ajout d'éléments en bloc…).

Exemple :

```
@Controller
public class UserController{
    @Autowired
    private UserService myService;

    @RequestMapping(path="users", method = RequestMethod.GET)
    public String users(@RequestParam String name, Map<String, Object> model) {
        List<User> usersList = myService.find(name);
        model.put("results", usersList);
        return "users/list";
    }
}
```

Les attributs du modèle (c'est-à-dire les entrées de la `Map`) sont disponibles dans la vue `users/list`.

Si nous utilisons des vues jsp (Spring peut aussi se coupler avec d'autres technologies), nous disposerons de ces éléments en utilisant la notation `${...}`.

```
<ul>
    <c:forEach var="u" items="${results}">
        <li>${u.firstname} - ${u.lastname}</li>
    </c:forEach>
</ul>
```

`firstname` et `lastname` sont des propriétés de la classe `User`.

Le moteur de rendu fabriquera le HTML d'après les blocs html statiques (ici `<ul>` et `<li>`) et les taglibs (ici `c:forEach`)

Parfois les informations transmises par le contrôleur doivent s'afficher dans un formulaire.

Exemple, si la route `/users/{id}` permet à un utilisateur d'accéder à son profil :

```
@Controller
public class UserController{

    @Autowired
    private UserService myService;

    @RequestMapping(path="users/{id}", method = RequestMethod.GET)
    public String user(@PathVariable int id, Map<String, Object> model) {
        User user = this.userService.getUserById(id);
        model.put("user", user);
        return "users/frm";
    }
}
```

La vue dispose ainsi de l'objet `user`, les champs du formulaire peuvent être associées à des propriétés de cet objet.

Nous utiliserons alors la taglib Spring plutôt que les tags html.

```
<form:form modelAttribute="user">
    Firstname :<form:input path="firstname" /><br/>
    Lastname :<form:input path="lastname" /><br/>
    Mail address :<form:input path="mailAddress" /><br/>
    Password : <form:password path="password" /><br/>
    <input type="submit" />
</form>
```

Bien sûr il faut que l'objet `user` du modèle transmis par le contrôleur ait les propriétés `firstname`, `lastname`, `mailAddress` et `password`.

Les relations sont supportées (exemple : `path="address.city"` si l'objet `user` a une propriété `address` qui a elle-même une propriété `city`)

Vue ⇒ contrôleur

La récupération d'informations depuis la vue suppose d'avoir identifié quel objet pouvait être rapprocher de la saisie utilisateur.

Rappel : les informations venant de la vue sont présentes dans les paramètres de la requête. Ceux-ci peuvent provenir

- du *form data* dans le cas d'un formulaire soumis en POST
- de la *query string* (?key=value&key2=value2 , etc...) dans le cas d'un formulaire soumis en GET et/ou si l'URL contient des paramètres.

Cela est uniifié dans l'API servlet : ce sont les méthodes `request.getParameter` et `request.getParametersMap` de l'interface `HttpServletRequest` qui permettent d'accéder aux paramètres de la requête.

A la soumission d'un formulaire nous ne souhaitons pas récupérer les paramètres de la requête un à un et reconstruire un objet métier (ici : `User`) et valoriser une à une les propriétés `firstname`, `lastname`, `mailAddress` et `password` de cet objet.

DON'T

```
@RequestMapping(path = "users/{id}" , method=RequestMethod.POST)
public String update(@PathVariable int id, @RequestParam String firstname, @RequestParam String lastname, @RequestParam
String mailAddress, @RequestParam String password){
    User user = new User(id, firstname, lastname, mailAddress, password);
    // sauvegarde de l'objet user
    return "redirect:/users/update-success";
}
```

DO

```
@RequestMapping(path = "users/{id}" , method=RequestMethod.POST)
public String update(@PathVariable int id, User user){
    // sauvegarde de l'objet user
    return "redirect:/users/update-success";
}
```

En effet, nous savons que Spring peut reconstruire un objet d'après les paramètres de la requête et le passer en argument à notre méthode.

Spring va créer (par réflexion) une instance de la classe `User` et parcourir les paramètres de la requêtes (`request.getParametersMap()`) pour valoriser les propriétés correspondantes sur l'instance nouvellement créée.

Si notre méthode déclenche le rendu d'une vue (par opposition à une redirection), l'objet reconstruit d'après les paramètres de la requête sera retransmis automatiquement à la vue sous un nom égal au nom de classe mais avec une minuscule pour la première lettre.

Si nous souhaitons contrôler ou rendre explicite ce nommage, nous utiliserons l'annotation `@ModelAttribute`

```
@RequestMapping(path = "users/{id}" , method=RequestMethod.POST)
public String update(@PathVariable int id, @ModelAttribute("user") User user)
{
    // sauvegarde de l'objet user
    return "redirect:/users/update-success";
}
```

Dans l'exemple précédent cela ne s'appliquera pas car notre méthode déclenche une redirection et non pas le rendu d'une vue.

L'intérêt est surtout visible :

- Si la soumission d'un formulaire conduit à rendre une vue. Il est souhaitable que la saisie utilisateur puisse être rappelée; ainsi pour un moteur de recherche la soumission du formulaire doit conduire à obtenir des résultats mais sans pour autant perdre les valeurs saisies dans les champs de recherche.
- Si nous avions eu un mécanisme de validation de saisie. Dans l'exemple précédent, un échec de la validation devra conduire à représenter la vue `users/frm` et il est important que l'objet `user` reçu en argument (et qui contient toute la saisie utilisateur) soit bien transmis à la vue afin de représenter un formulaire non vide à l'utilisateur.

La validation

Comme pour une API REST nous pouvons demander à ce que l'objet reconstruit d'après les paramètres de la requête (ici : une instance de `User`) soit valide compte tenu des contraintes de validation posées sur ses propriétés (`@NotNull`, `@Size`, etc...).

Il suffit alors, comme dans le cas d'une API REST, d'annoter le paramètre par `@Valid`

```
@RequestMapping(path = "users/{id}" , method=RequestMethod.POST)
public String update(@PathVariable int id, @Valid @ModelAttribute("user") User user)
{
    // sauvegarde de l'objet user
    return "redirect:/users/update-success";
}
```

Ici Spring vérifiera si les contraintes de validité sur l'objet `User` sont respectées.

Si ce n'est pas le cas il n'invoquera pas la méthode et retournera une erreur 400.... ce qui nous conduit dans une impasse : nous ne pouvons pas présenter une erreur 400 à l'utilisateur.

Nous devons pouvoir décider quelle vue présenter à l'utilisateur si la saisie n'est pas valide. Il faut alors ajouter un argument de type `BindingResult` immédiatement après le paramètre annoté par `@Valid` :

```
@RequestMapping(path = "users/{id}" , method=RequestMethod.POST)
public String update(@PathVariable int id, @Valid @ModelAttribute("user") User user, BindingResult binding)
{
    if(binding.hasErrors()){
        return " users/frm";
    }
    // sauvegarde de l'objet user
    return "redirect:/users/update-success";
}
```

Si l'objet `user` n'est pas valide nous déclenchons le rendu de la vue `users/frm`.

L'objet user reçu en argument sera automatiquement transmis à la vue (sous le nom `user`), ce qui permettra de représenter le formulaire avec la dernière saisie utilisateur.

La présentation des messages d'erreurs dans la vue se fera avec la taglib Spring `form:errors`

Exemple : `<form:errors path="firstname"/>` à côté du tag `<form:input path="firstname"/>`

Les controllerAdvice

Comme pour une API REST, nous pouvons déclarer des `@ControllerAdvice`.

Un `@ControllerAdvice` est un *assistant* des contrôleurs et implémente des mécanismes communs aux `@Controller`:

Il s'agit essentiellement

- des modèles : méthodes annotées par `@ModelAttribute`
- des traitement d'exceptions : méthodes annotées par `@ExceptionHandler`

Exemple de `@ControllerAdvice` :

```
@ControllerAdvice
public class MyControllerAdvice{

    private List<Country> countries;// valorisé dans une méthode @PostConstruct

    @ModelAttribute("countriesList")
    public List<Country> countries(){
        return this.countries;
    }

    // gestionnaires d'exception
}
```

Ainsi toutes les vues peuvent profiter d'un attribut `${countriesList}`.

Attention à ne pas réaliser de traitement lourd dans ces méthodes (requêtes SQL par exemple) car elles seront invoquées à chaque rendu de chaque vue.

Elle ne doivent être que des accesseurs vers des objets valorisés par ailleurs, dans une méthode d'initialisation annotées par `@PostConstruct` par exemple.

Cette méthode peut aussi être appelée suivant une planification ou être exposée en *JMX*.

Exemple de `@ExceptionHandler` :

```
@ExceptionHandler(ApplicationException.class)
public String handleException(ApplicationException e, Map<String, Object> model) {
    model.put("exception",e);
    return "application-exception";
}

@ExceptionHandler(Throwable.class)
public String handleException(Throwable t) {
    return "default-exception";
}
```

Les view controllers

Déclarer un *view controller* revient à associer une route à une vue, sans passer par un contrôleur.

Cela est utile pour des pages 100% statiques (page d'aide, page de contacts, page de mentions légales…).

La déclaration d'un *view controller* se fait dans la classe de configuration, en redéfinissant la méthode `addViewControllers`.

Exemple :

```
@Override
public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/help").setViewName("static/help");
    registry.addViewController("/contacts").setViewName("static/contacts");
}
```

`static/help` et `static/contacts` sont des noms de vues, le rapprochement avec les fichiers physiques correspondant se fera par le `ViewResolver`

L'internationalisation

La problématique est d'externaliser les messages des vues afin de pouvoir gérer un site multilingue.

La solution consiste à identifier tous ces messages et à les stocker dans des fichiers properties spécifiques à chaque langue (exemple : `uiMessages_en.properties`, `uiMessages_fr.properties`).

L'inscription d'un jeu de fichier properties (aussi appelé `ResourceBundle`) se fait par la déclaration d'un bean nommé `messageSource` dans la classe de configuration :

```
@Configuration @EnableWebMvc @ComponentScan
public class WebConfiguration implements WebMvcConfigurer { ①

    @Bean(name="messageSource")
    public ResourceBundleMessageSource messageSource() {
        ResourceBundleMessageSource source = new ResourceBundleMessageSource();
        source.setBasename("uiMessages");
        return source;
    }
}
```

① ou `extends WebMvcConfigurerAdapter` avec Spring 4.

Dans les vues jsp il est ensuite possible d'utiliser la taglib jstl `fmt` :

```
<fmt:message key="... (clé du message)" />
```

Le choix de la langue est effectué par un `LocaleResolver`

Par défaut Spring MVC inscrit dans le contexte une instance de l'implementation `AcceptHeaderLocaleResolver`.

Cette implementation définit la `Locale` de l'utilisateur en examinant l'en-tête `accept-language` de la requête HTTP

D'autres implémentations sont disponibles :

- `CookieLocaleResolver` (avec les propriétés `cookieName`, `cookieAge` et `cookiePath`)
- `SessionLocaleResolver`

Il suffit de les déclarer (l'une ou l'autre…) dans le contexte Spring sous forme de `beans`

Spring Web Flux

- Principes
- `Mono<T>` et `Flux<T>`
- Push, pull, back pressure
- Le `WebClient`
- L'accès aux données
- Spring Web Flux et API REST.
- Le serveur web
- Synthèse

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

Principes

Souvent, les applications de gestions doivent accéder à des ressources externes :

- bases de données
- webservices

Cela introduit une latence : un délai entre le moment où l'application sollicite la ressource et le moment où la ressource retourne un résultat.

Traditionnellement, en Java, le *thread* qui sollicite la ressource **attend** au lieu d'être occupé à autre chose. Ainsi la latence est subie au lieu d'être contournée.

Nous parlons alors de *blocking I/O*.

Conséquence : le nombre de threads augmente à mesure que la charge augmente, et ceux-ci passent une grande partie de leur temps de vie à attendre.

Si ce fonctionnement est très courant en Java, d'autres languages gèrent ces situations différement.

Par exemple, le moteur JavaScript du navigateur ne dispose que d'un seul *thread*.

Et l'accès à une ressource externe se fait de manière non bloquante et asynchrone.

```
fetch("http://api.acme.com/books/1").then(x => x.json()).then(x => console.log(x));
```

La méthode `fetch` retourne une `Promise`, qui représente un résultat pas encore disponible.

Mais nous n'avons pas besoin que le résultat soit disponible pour définir les traitements que nous voulons appliquer dessus quand il le sera (ici la transformation en *json* et l'affichage dans la console).

Ce fonctionnement s'applique aussi côté serveur avec NodeJS : il y a un seul *thread*, celui-ci ne peut **jamais** être bloqué.

La manière de programmer devient différente : appels asynchrones et programmation fonctionnelle (les *callbacks*) plutôt que programmation impérative synchrone.

Mono<T> et Flux<T>

Pour proposer les mêmes mécanismes, Spring s'appuie sur un projet nommé **Reactor**.

Ainsi, la dépendance `spring-webflux` a comme dépendance transitive :

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <!--version-->
</dependency>
```

Ce projet propose deux classes essentielles : `reactor.core.publisher.Mono<T>` et `reactor.core.publisher.Flux<T>`.

- `Mono<T>` : représente un (0..1) élément pas encore disponible.
- `Flux<T>` : représente des (0..n) éléments pas encore disponibles.

Dans une applications réactive, si une méthode qui accède à une source de données externe alors elle doit retourner qu'un `Mono<T>` ou un `Flux<T>`.

Ainsi,

- si nous soumettons une requête `select` à une base de données, nous obtenons immédiatement un `Mono<T>` ou un `Flux<T>` correspondant au(x) résultat(s) prochainement disponible(s).
- si nous accédons à une ressource HTTP, nous obtenons immédiatement un `Mono<T>` ou un `Flux<T>` correspondant au(x) résultat(s) prochainement disponible(s).

L'objectif n'est pas tant de gagner en performance que de pouvoir accepter une augmentation de la charge.

Pour qu'une application soit réactive, il faut que les ressources auxquelles elle accède acceptent les appels non bloquants.

`Mono<T>` et `Flux<T>` implémentent `org.reactivestreams.Publisher<T>`. En effet, ce sont des publicateurs d'éléments : les éléments que nous avons demandés.

Extrait de la JavaDoc : *A Publisher is a provider of a potentially unbounded number of sequenced elements, publishing them according to the demand received from its Subscriber(s)*

Nous remarquons que l'interface `Publisher<T>` vient de *org.reactivestreams*, une spécification qui standardise les princpes réactifs en Java.

Reactor est une implémentation de cette spécification mais Spring peut aussi utiliser d'autres implémentations, RxJava par exemple. Les équivalent de `Mono<T>` et `Flux<T>` sont alors `Single<T>` et `Observable<T>`

Ce mécanisme est couvert par Java SE depuis la version 9 avec le type `java.util.concurrent.Flow`, un effort de convergence entre la spécification *reactive streams* et Java SE est en cours.

Push, pull, back pressure

Les `Mono<T>` et `Flux<T>` peuvent fonctionner en mode *push* ou *pull*.

Soit *l'appelant* celui qui obtient le `Mono<T>` ou le `Flux<T>` :

- *pull* : l'appelant exige d'obtenir le résultat.
- *push* : l'appelant définit une fonction qui sera appelé lorsque le résultat sera disponible.

La congestion peut donc se produire de deux côté :

- côté appelant en cas de *push* si l'appelant n'est pas en position de traiter le résultat qui lui est donné.
- côté appelé en cas de *pull* si l'appelant exige d'obtenir le résultat.

En privilégiant le *push*, nous évitons de surcharger l'appelé : c'est lui qui décide quand il traite les demandes qui lui sont adressés. Ce fonctionnement porte un nom : *back pressure*.

Quant à l'appelant, il dispose des `delayElement`, `delaySubscription` et `delayUntil` pour ralentir la cadance laquelle il reçoit les résultats.

Le WebClient

Pour les requêtes HTTP, Spring propose un `WebClient` :

Exemple d'utilisation avec `Mono<T>` :

```
// soit BookInfo une classe représentant les informations d'un livre.

WebClient webClient = WebClient.builder().baseUrl("http://api.acme.com").build();
Mono<BookInfo> mono = webClient.get().uri("books/1")
    .retrieve()
    .bodyToMono(BookInfo.class);
mono.subscribe(bookInfo -> System.out.println(bookInfo.getTitle()); ①
```

① Nous reconnaissons ici le *push*. En *pull* nous aurions écrit : `BookInfo b = mono.block()`

Exemple avec `Flux<T>`

```
// soit FlighInfo une classe représentant un vol proposé par une compagnie aérienne.

Map<String, Object> args= Map.of("from", "CDG", "to", "UIO", "departure", LocalDate.now(), "return", LocalDate.now
().plusDays(30));

WebClient webClient = WebClient.builder().build();
Flux<FlightInfo> results1 = webClient.get().uri("http://api.foo-airways.com/flights", args)
    .retrieve()
    .bodyToFlux(FlightInfo.class));

Flux<FlightInfo> results2 = webClient.get().uri("http://api.bar-airways.com/flights", args)
    .retrieve()
    .bodyToFlux(FlightInfo.class));

Flux<FlightInfo> allResults = results1.concatWith(results2)
    .sort((x, y) -> x.getPrice().compareTo(y.getPrice()));
```

L’accès aux données

Pour les base de données les choses sont plus compliquées, car **JDBC est bloquant** et n’implémente donc pas les *reactive streams*.

Cela rend impossible toute appel réactif : nous ne pouvons pas soumettre une requête SQL à une base de données et obtenir tout de suite un résultat qui sera disponible plus tard.

En revanche d’autres bases de données proposent des *drivers* reactifs (c’est à dire implémentant `org.reactivestreams.Publisher<T>`).

Liste non exhaustive :

- MongoDB
- Cassandra
- Redis

Et Spring Data propose une interface `ReactiveCrudRepository` dont nos *repositories* peuvent hériter :

```
public interface BookRepository extends ReactiveCassandraRepository<Book, UUID> {

    // méthodes
}
```

Nous disposons alors des méthodes suivantes (liste non exhaustive) :

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

<S extends T> Flux<S> saveAll(Iterable<S> entities);

<S extends T> Flux<S> saveAll(Publisher<S> entityStream);

Mono<T> findById(ID id);

Mono<T> findById(Publisher<ID> id);

Mono<Boolean> existsById(ID id);

Mono<Boolean> existsById(Publisher<ID> id);

Flux<T> findAll();

Mono<Long> count();

Mono<Void> deleteById(ID id);

Mono<Void> delete(T entity);

Mono<Void> deleteAll(Iterable<? extends T> entities);

Mono<Void> deleteAll(Publisher<? extends T> entityStream);

Mono<Void> deleteAll();
```

Spring Web Flux et API REST

Si les objets utilisés pour échanger avec des sources de données sont des `Mono<T>` et des `Flux<T>`, notre API REST a avantage a exposer ces mêmes objets :

```
@RestController
public class MyEndpoint{

    private WebClient webClient = WebClient.builder().build();

    @GetMapping("flights")
    public Flux<FlightInfo> flightInfos(@RequestParam String from, @RequestParam String to, @RequestParam LocalDate
departureDate, @RequestParam LocalDate returnDate){

        Map<String, Object> args= Map.of("from", from, "to", to, "departure", departureDate, "return", returnDate);

        String uri1 = "http://api.foo-airways.com/flights";
        Flux<FlightInfo> results1 = webClient.get().uri(uri1, args)
            .retrieve().bodyToFlux(FlightInfo.class));

        String uri2 = "http://api.bar-airways.com/flights";
        Flux<FlightInfo> results2 = webClient.get().uri(uri2, args)
            .retrieve().bodyToFlux(FlightInfo.class));

        Flux<FlightInfo> allResults = results1.concatWith(results2)
            .sort((x, y) -> x.getPrice().compareTo(y.getPrice()));

        return allResults;
    }
}
```

Le serveur web

Une API REST réactive s'intègre assez mal avec à l'API Servlet, car celle-ci prévoit un traitement bloquant des requêtes HTTP.

La surcouche asynchrone proposée par Servlet 3.1 ne rend pas le traitement des requêtes réellement non bloquant : le seul fait que la méthode `service(HttpServletRequest request, HttpServletResponse response)` retourne un `void` montre que l'appel d'une *servlet* est bloquant.

```
public abstract class HttpServlet{
    public void service(HttpServletRequest request,HttpServletResponse response){
        // code
    }
}
```

Un traitement non bloquant des requêtes tel que Spring le propose permet de bien saisir la différence :

`org.springframework.http.server.reactive.HttpHandler` :

```
package org.springframework.http.server.reactive.HttpHandler

// imports
public interface HttpHandler {
    Mono<Void> handle(ServerHttpRequest request, ServerHttpResponse response);
}
```

Toutefois les *servlet containers* (Tomcat, Jetty, Undertow) ont évolué et implémentent des mécanismes asynchrones.

Ainsi il est possible d'utiliser Spring WebFlux avec Tomcat, Jetty ou Undertow : Spring propose alors un *bridge* entre ces serveurs et le type `Publisher<T>`.

Toutefois Spring encourage à utiliser un serveur web nativement prévu pour cela : Netty.

Ce n'est pas un *servlet container* mais un framework bas niveau capable de gérer les échanges réseau de manière asynchrone.

Notre code ne dépend pas de Netty, c'est Spring qui gère le lien entre les `Mono<T>` et `Flux<T>` que renvoient notre application et la gestion asynchrone et non bloquante des requêtes HTTP par Netty.

Synthèse

- Spring facilite la mise en place d'applications réactives en s'appuyant sur des les *reactive streams* (dont *Reactor* est une implémentation).
- Le gain ne concerne pas tant la performance que l'économie de ressource : une application réactive peut accepter une charge supérieure sans mobiliser plus de *thread*.
- Le gain est d'autant plus important que nos applications accèdent à des données externes car chaque appel à celles-ci occasionne une **latence**.
- Mais une application n'est réactive que si elle peut solliciter les sources de données externes de manière réactive. Et JDBC n'est pas réactif.
- Cela conduit à programmer de manière différente : programmation fonctionnelle plutôt que programmation impérative.
- Spring **ne nous encourage pas** à utiliser `spring-webflux` plutôt que `spring-webmvc`.
- Spring nous laisse le choix, aussi bien pour l'implémentation de la spécification *reactive streams* que pour le serveur web.

A voir : https://www.youtube.com/watch?v=rdgJ8fOxJhc

Spring Security

- Principes
- Configuration
- Définition d'un référentiel d'utilisateur
- Sécurisation des routes
- Sécurisation des méthodes
- Sécurisation des pages
- Obtention du contexte de sécurité
- Les tests

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</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 : `@Configuration` et `@EnableWebSecurity`

A nouveau, le fait d'étendre une classe abstraite permet d'être guidée 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, dans la méthode `onStartup` de la classe qui implémente `WebApplicationInitializer`

Elle doit être chargé avec le contexte applicatif (celui chargé par le `ContextLoaderListener`).

Ensuite, il faut inscrire le filtre de sécurité (dans la méthode `onStartup`) :

```
DelegatingFilterProxy filter = new DelegatingFilterProxy("springSecurityFilterChain");
Dynamic filterRegistration = sc.addFilter("springSecurityFilterChain", filter);
filterRegistration.addMappingForUrlPatterns(null, false, "/*");
```

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

La classe `AuthenticationManagerBuilder` propose aussi des methodes `ldapAuthentication` et `jdbcAuthentication`, chacun se configurant de manière spécifique.

Sécurisation des routes

La sécurisation des routes d'après des *patterns* se fait en redéfinissant une autre méthode `configure` de la classe abstraite `WebSecurityConfigurerAdapter`

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.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')`

Cette seconde méthode permet aussi de redéfinir, si on le souhaite, les routes correspondant au login et au logout ainsi que le nom des champs où l'utilisateur saisit son login et son mot de passe.

Ces informations seront bien sûr utilisées par le filtre qui analyse les requêtes et autorise ou non l'accès à la route demandée.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/books/**").hasRole("BOOK_ADMIN")
        .antMatchers("/publishers/**").access("hasRole('PUBLISHER_ADMIN')")
        .anyRequest().authenticated()
    .and()
        .formLogin()
        .usernameParameter("username")
        .passwordParameter("password")
        .loginPage("/login").permitAll()
    .and()
        .logout().logoutUrl("/logout")
        .logoutSuccessUrl("/logout-success").permitAll();
}
```

Les routes `/login` et `/logout-success` ont intérêt à être définies avec des `ViewController`.

Sécurisation des méthodes

La sécurisation des méthodes consiste 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 pages

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

Cela suppose d'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>
```

Obtention du contexte de sécurité

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 `javax.security.Principal` ou `org.springframework.security.core.Authentication` dans une méthode annotée par `@RequestMapping`
- Dans les vues, avec la *taglib* spring-security. 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 web socket

- Principes
- Activation
- L'échange de messages
- L'externalisation des sessions

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

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

Principes

Les WebSockets sont un moyen d'établir un canal TCP bi-directionnel entre un navigateur et un serveur.

Les WebSockets sont supportées par les navigateurs récents (Internet Explorer depuis la version 10).

Des mécanismes de *fallback* (« mode dégragé » existent) pour les navigateurs plus anciens.

Les cas d'utilisation sont nombreux :

- Mise à jour de positions sur une carte,
- Mise à jour de stock pour un magasin
- Chat
- Jeux en ligne

C’est toujours le client navigateur qui initie la connexion.

Une fois celle-ci établit il peut :

- envoyer des messages
- recevoir des messages
- fermer la connexion

Le premier point est moins fréquent car si le dialogue entre le client et le serveur se fait par WebSocket dans les deux sens cela signifie que WebSocket s’est substitué au protocole http.

Ceci n’est pas forcément souhaitable compte tenu des avantages du protocole http (cache, non bloqué par les firewall, verbes, codes de retour, compression gzip, etc…).

A lire : http://blog.arungupta.me/2014/02/rest-vs-WebSocket-comparison-benchmarks/Enjeux

Les enjeux côté client :

- ouvrir une connexion
- définir un callback qui recevra les messages destinés à ce client
- fermer la connexion

Les enjeux côté serveur :

- accueillir les connexions clientes
- envoyer des messages aux clients
- maintenir à jour la liste des clients

Activation

Le couplage de Spring MVC à Spring WebSocket et Spring Messaging se fait via une classe de configuration.

Celle-ci est bien sûr annotée par `@Configuration` et par `@EnableWebSocketMessageBroker`

Et implémente l'interface `WebSocketMessageBrokerConfigurer`

```
@Configuration @EnableWebSocketMessageBroker
public class WsConfig implements WebSocketMessageBrokerConfigurer  { ①
    @Override
    public void configureStompEndpoints(StompEndpointRegistry registry) {
        registry.addEndpoint("/hello").withSockJS(); ②
    }
}
```

① Avec Spring 4, la classe abstraite `AbstractWebSocketMessageBrokerConfigurer` fournit une implémentation par défaut des méthodes de cette interface.

② Nous avons ici ajouter un point de connexion `/hello`

Côté client (avec SockJS) :

```
const socket = new SockJS("//api.acme.com/hello");
```

Si le protocole WebSocket n'est pas supporté par le navigateur un *mode dégradé* s'appliquera. C'est SockJS qui fournit cette abstraction.

L’échange de messages

Spring propose aussi, via Spring messaging, un moyen d’organiser un acheminement des messages vers les clients WebSocket et d’implémenter ainsi une logique applicative.

Cela passe par l’identification préalable des destinations auxquelles les clients javascript pourront s’abonner.

Il sera ainsi possible côté serveur de recevoir les messages envoyés sur des destinations (*topics* ou *queues*) et de les acheminer auprès des clients qui ont souscrits à ces destinations.

Côté serveur nous utiliserons `spring-messaging`, d'où l'importance de l'inscrire comme dépendance Maven en plus de `spring-WebSocket`.

Côté client (JavaScript) nous utiliserons STOMP :

```
const socket = new SockJS("//api.acme.com/hello");
const stompClient = Stomp.over(socket);
/* le deuxième argument passé à la fonction connect est un callback qui sera appelé une fois la connexion établie.*/
stompClient.connect({}, frame => {
    var topic = "/games/123";

    const subscription = stompClient.subscribe(topic, (msg) => {
        const body = msg.body;
        // si le body est un objet
        const obj = JSON.parse(msg.body);
        // mise à jour de la vue html compte tenu de l'information reçue
    });
    // désinscription : subscription.unsubscribe();
    }
)
```

Côté Java il est possible de définir un *callback* qui sera appelé lors des souscriptions :

```
@RestController
public class GameController{
    // sera appelé à chaque souscriptions sur /games/{gameId}
    @SubscribeMapping("/games/{gameId}")
    public void joinGame(@DestinationVariable int gameId){
        // code (historisation des souscriptions par exemple).
    }
}
```

Côté client : envoi d'un message sur une destination :

```
const stompClient = Stomp.over(socket);
const topic = "games/123";
stompClient.connect({}, frame => {
    const subscription = stompClient.subscribe(topic, (msg) => {
    const body = msg.body;
    // si le body est un objet
    const obj = JSON.parse(msg.body);
    });
});

// appelé lors d'un clic sur un bouton par l'utilisateur
function play(suite, rank) // exemple : play('diamond', "8");
    const msg = {"suite": suite , "rank":rank, player: "John Doe"}:
    const headers = {"accessKey": "1a2b3c"}
    stompClient.send(topic, headers, JSON.stringify(msg));
}
```

Côté Java il est aussi possible de s’abonner aux destinations et de recevoir les messages qui y sont postés :

```
@RestController
public class GameController{
    public static class GameAction{
        public String suite, rank, player;
    }

    // sera appelé à chaque message envoyé sur /games/{gameId}
    @MessageMapping("/games/{gameId}")
    public void onMessage(@DestinationVariable int gameId, @Payload GameAction action, @Header String accessKey)
    {
        // code (historisation des actions par exemple).
    }
}
```

Dans l'exemple précédent il n'est pas possible de contrôler le message envoyé et d'effectuer un contrôle ou un traitement métier dessus.

Pour cela il faut utiliser deux destinations : une pour les messages postés par les clients et une où le serveur dépose les messages qui leurs sont destinés

```
@RestController
public class GameController{
    public static class GameAction{
        public String suite, rank, player;
    }

    @SendTo("/games/{gameId}/actions") ①
    @MessageMapping("/games/{gameId}/action-requests")
    public GameAction onMessage(@DestinationVariable int gameId, @Payload GameAction action, @Header String userId) {
    /*
        traitement(s) métier(s), par exemple : vérifiation que l'action est correcte
        compte tenu de l'état du jeu et levée d'une exception si ce n'est pas le cas
    */
        return action;
    }
}
```

① Ce que retourne la méthode sera déposé sur la destionation `/games/{gameId}/action-requests`

Si nous utilisons deux destinations (ici : `action-requests` et `actions`), le code JavaScript doit être :

```
var stompClient = Stomp.over(socket);
stompClient.connect({}, frame => {
    const subscription = stompClient.subscribe("games/123/actions", (msg) => {
    const body = msg.body;
    // si le body est un objet
    var obj = JSON.parse(msg.body);
    });
});

// appelé lors d'un clic sur un bouton par l'utilisateur
function play(suite, rank) // exemple : play('diamond', "8");
    const msg = {suite: suite ,rank:rank, player:"John Doe"}:
    const headers = {"accessKey": "1a2b3c"}
    stompClient.send("games/123/action-requests", headers, JSON.stringify(msg));
}
```

L'utilisation conjointe de REST et WebSocket fait sens lorsque WebSocket n'est utilisé que pour diffuser des informations vers les clients navigateur.

Dans ce cas :

- Les appels du client navigateur vers le serveur se font en REST
- WebSocket n'est utilisé que pour diffuser des informations vers les clients.

L'objet `SimpMessagingTemplate` peut alors être utilisé.

Un *bean* de ce type est automatiquement inscrit dans le contexte compte tenu de l'annotation `@EnableWebSocketMessageBroker` , et peut donc être injecté dans nos beans avec l'annotation `@Autowired`.

Cet objet permet entre autre l'envoi de messages sur une destination, avec la méthode `convertAndSend`.

```
@RestController
public class GameController{
    public static class GameAction{
        public String suite, rank, player;
    }
    @Autowired
    protected SimpMessagingTemplate broker;

    @RequestMapping(path="games/{gameId}", method=RequestMethod.POST)
    public void card(@PathVariable int gameId, @RequestBody GameAction action, @RequestHeader String accesKey){
        // traitement métier sur action
        String destination = "games/"+gameId+"/actions";
        this.broker.convertAndSend(destination, action);
    }
}
```

L’externalisation des sessions

Dans un environnement Cloud (ou sur n’importe environnement composé d’un *load balancer* et d’un ensemble de serveurs), chaque serveur ne connait que les souscriptions qui lui ont été adressées.

Celles-ci devraient être centralisées, dans une base de données par exemple.

Cela peut être mis en œuvre avec `spring-session` (récupérable par Maven).

La classe de configuration ne doit alors pas étendre `AbstractWebSocketMessageBrokerConfigurer` mais `AbstractSessionWebSocketMessageBrokerConfigurer`

Et le contexte chargé au démarrage doit contenir un *bean* (déclaré via une méthode annotée par `@Bean`) qui représente une connexion vers la base de données où seront stockées les informations sur les sessions en cours (qui est abonné à quelles destinations).