Spring Avancé

IPPON
Enterprise Java Delivery

Bienvenue

- **Présentation**
- **Organisation**
- **Détails pratiques**

Pré requis

- Pour suivre le cours
  - Avoir de bonnes bases en Java : les JavaBeans, les Collections, JDBC…
  - Avoir des bases en Java EE : Servlets, JSPs…
  - Connaître XML
- Pour faire les exercices
  - Un ordinateur connecté à Internet
  - L'envie de coder :-)

Présentation

- Qui suis-je ?

Présentation

- Qui êtes-vous ?

Présentation du cours

- 3 jours pour devenir ceinture noire en Spring
  - Spring Core en détail
  - Les accès aux bases de données
  - La couche Web
- Objectif : savoir réaliser une application Spring de bout en bout
  - En comprenant correctement le fonctionnement du framework
  - Spring étant très homogène, cela vous permettra également de comprendre rapidement les parties de Spring que nous ne verrons (volontairement) pas
- Cours théorique (slides) et pratique (exercices)

Versions étudiées

- Ce cours est basé sur les versions suivantes
  - Spring Framework 3.2
  - Spring Web Flow 2.3
  - Spring Security 3.1
  - JPA 2.0
- Spring a toujours eu une excellente compatibilité ascendante, et des APIs très stables

IPPON Enterprise Java Delivery

Plan du cours

- Introduction à Spring
- Spring IoC
- Configuration Spring avancée
- Architecture d'applications Spring
- Spring AOP
- Spring JDBC
- Les transactions
- Les tests avec Spring
- Hibernate & JPA
- Bean Validation
- Spring JMS
- Spring JMX
- Spring MVC
- Spring MVC REST
- Spring Web Flow
- Spring Security

IPPON Enterprise Java Delivery

Les bases de Spring

- Introduction à Spring
- Spring IoC
- Configuration Spring avancée
- Architecture d'applications Spring

Introduction à Spring

IPPON Enterprise Java Delivery

Agenda

- Histoire (brève) de Spring
- Qu’est-ce que Spring ?
- Que peut-on faire avec Spring ?

IPPON Enterprise Java Delivery

Agenda

- **Histoire (brève) de Spring**
- Qu’est-ce que Spring ?
- Que peut-on faire avec Spring ?

Historique de Spring

- **Octobre 2002** Rod Johnson publie son livre «Expert One-on-One J2EE Design and Development», dans lequel il propose du code, qui va devenir plus tard le framework Spring
- **Mars 2004** Spring 1.0 sort sous licence Apache 2.0
- **2005** Spring devient populaire, en particulier en réaction par rapport aux EJBs 2.x
- **2006** Spring gagne un «Jolt Productivity Award», ce qui assoit sa notoriété
- **2007** Sortie de Spring 2.5, avec support des annotations
- **2009** Achat de SpringSource par VMWare (420 M$)
- **2013** Création de Pivotal, une joint venture entre VMWare et EMC Corporation

IPPON Enterprise Java Delivery

Agenda

- Histoire (brève) de Spring
- **Qu'est-ce que Spring ?**
- Que peut-on faire avec Spring ?

IPPON Enterprise Java Delivery

Qu'est-ce que Spring ?

- Un framework Java
- Un conteneur IoC
- Un ensemble de projets
- Une communauté et une société
- Open Source : licence Apache 2.0

Un framework Java

- Framework : un cadre de développement
  - Contient des «bonnes pratiques»
  - Permet d'éviter de recoder des classes utilitaires
  - Permet de se focaliser sur le métier
    - Spring fournit la «plomberie» : le socle technique
    - Les développeurs peuvent se concentrer sur le code métier (le vrai travail)
- A l'origine orienté Java et Java EE
  - Aujourd'hui d'autres implémentations existent : .NET et Python

Un container IoC

- IoC == Inversion of Control
  - Le «principe d'Hollywood» :
    Ne nous appelez pas, nous vous rappellerons
  - Permet d'avoir des composants «faiblement couplés»
    - Améliore la qualité du code
    - Facilite les tests
- La principale forme d'IoC : «l'injection de dépendances» (ou DI)

Exemple d'injection de dépendances

```
public class TrainingService {

    @Inject
    private TrainingRepository repository;

    // Code...

}
```

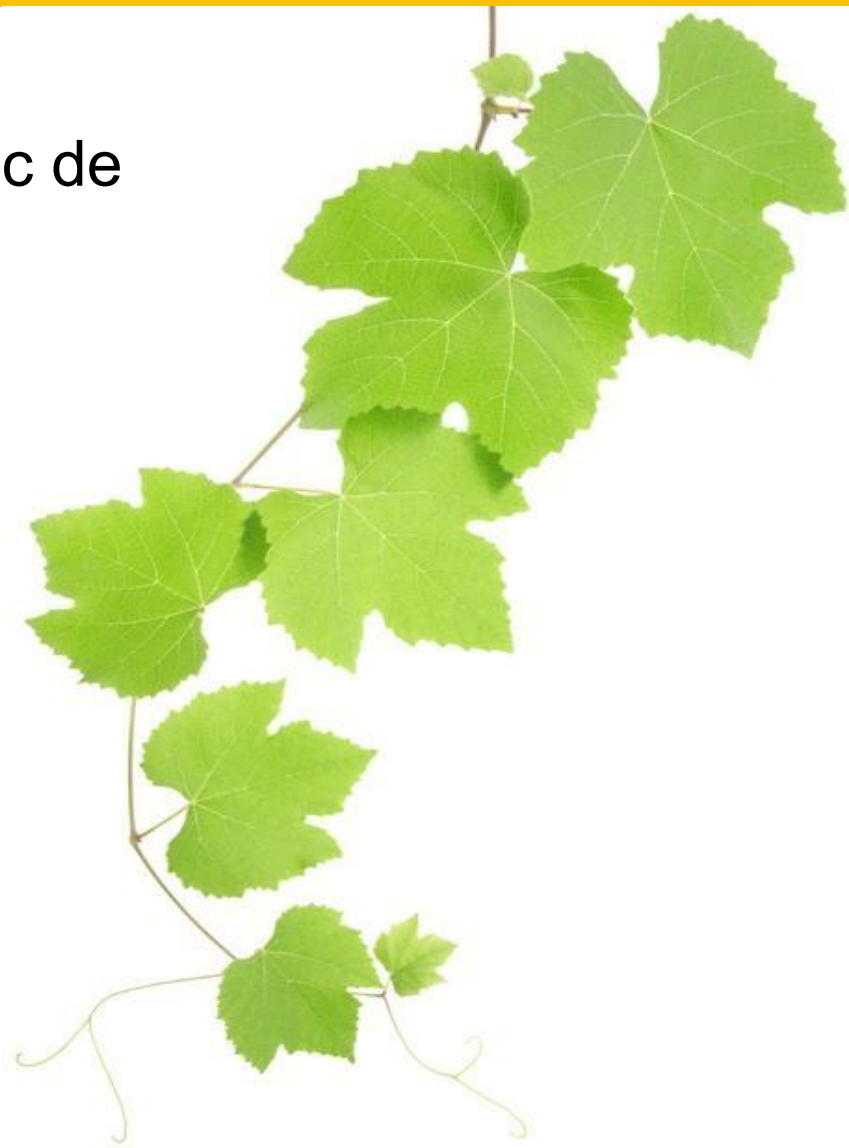
- A l'exécution, une instance de «*TrainingRepository*» est injectée dans l'instance de «*TrainingService*»
- Spring se charge de cette injection

Quel intérêt ?

- Cette injection permet de «découpler» les classes
  - Donne une architecture logicielle plus souple, plus simple à faire évoluer
  - Facilite les tests
- Permet à Spring «d'enrichir» les instances d'objets injectés
  - Les objets injectés peuvent être modifiés par Spring
  - Par exemple, les rendre transactionnels ou sécurisés
  - Ceci est permis grâce à la Programmation Orientée Aspect
  - C'est ainsi que Spring peut fournir la «plomberie technique», sans que le développeur n'ait à faire quoi que ce soit

Un écosystème complet

- Spring est un projet principal avec de nombreux sous-projets
  - On parle de «Spring Core» pour le projet principal
  - Spring MVC
  - Spring Web Flow
  - Spring Security
  - Spring Batch
  - Spring Integration
  - Spring Web Services
  - Spring Roo
  - ...



La communauté & VMWare

- Spring est à la base un projet communautaire
- Mais le projet est «sponsorisé» par VMWare
  - Les développeurs principaux travaillent pour VMWare
  - La propriété intellectuelle appartient à VMWare

- Certains sous-projets ont leurs propres particularités (pas de «sponsorisation» par VMWare, ou «sponsorisation» en commun avec d'autres sociétés)
- Vous pouvez participer, et le code sera toujours Open Source, mais la gestion du projet reste maîtrisée par VMWare

Open Source et licence

- Spring et ses sous-projets sont sous licence «Apache 2.0»
  - Licence certifiée Open Source par l'OSI
  - Non virale, et donc «business friendly», par opposition à la GNU GPL
  - Vous pouvez donc utiliser Spring, l'étendre et le modifier sans aucun souci
- Par contre ce n'est pas un projet de la fondation Apache, et il n'a pas son modèle de gouvernance et de développement

Show me the code !

- Le code de Spring est disponible gratuitement, à plusieurs endroits
- Officiellement : le site de Spring (http://spring.io/)
  - Vous n'êtes pas forcés de vous inscrire
  - Garantie d'avoir le «bon» code (sans modification par des tiers malveillants)
- Maven : les versions stables sont sur le repository central, les autres sur le repository de Spring
- GitHub
  - Spring Core
    - https://github.com/spring-projects/spring-framework
  - Sous-projets Spring
    - GitHub: https://github.com/spring-projects

IPPON Enterprise Java Delivery

Agenda

- Histoire (brève) de Spring
- Qu’est-ce que Spring ?
- **Que peut-on faire avec Spring ?**

IPPON Enterprise Java Delivery

Que peut-on faire avec Spring ?

- Spring fournit un «cadre de travail»
  - Une aide, essentiellement pour simplifier les aspects techniques des projets
  - Des patterns d'architecture prédéfinis
  - Spring fait «la plomberie», à vous de coder la partie métier du projet
- Les sous-projets **traitent** de problématiques techniques plus spécifiques
  - Réaliser un batch, un Web Service, sécuriser son application...
  - Ils fonctionnent tous sur le même principe, et sont par conséquent très faciles à apprendre une fois qu'on maîtrise le «Core»
- Rien n'est obligatoire, tout est configurable
- Vous ne prenez que ce qui vous plaît
  - Très souple et sans contrainte

Spring et les serveurs d'applications

- Spring n'est pas un serveur d'applications
  - Il peut fonctionner sans serveur d'applications (application «stand alone», par exemple un batch)
  - Il peut fonctionner à l'intérieur d'un serveur d'applications
    - Il peut alors utiliser les fonctionnalités du serveur, pour en simplifier ou en abstraire le fonctionnement
    - Il peut également remplacer certaines fonctionnalités du serveur, si on juge que Spring propose une meilleure alternative
- Généralement Spring est utilisé conjointement à un serveur d'applications léger : Tomcat ou Jetty
  - Peu d'intérêt à utiliser Spring dans un serveur Java EE «complet»
  - Mais Spring fonctionne également parfaitement avec Weblogic, WebSphere, Glassfish, etc

Spring dans la vraie vie

- Spring est aujourd'hui le framework Java n°1 en entreprise
- Dans tous les secteurs d'activité
- Dans tous les pays
- Tous les frameworks concurrents ont plus ou moins disparu (Avalon, HiveMind, PicoContainer...)
- Les EJB 3.x s'inspirent beaucoup de Spring
  - Java EE revient progressivement dans la course contre Spring

Conclusion

- Spring est un framework Open Source
- Propose des «bonnes pratiques», des patterns d'architecture, du code utilitaire
- Très souple et très polyvalent
- Avec de nombreux sous-projets
- Fournit «la plomberie», et permet aux développeurs de se focaliser sur le code métier

Spring IoC

IPPON Enterprise Java Delivery

Agenda

- L'injection de dépendances
- IoC dans Spring
- Les scopes de beans
- Le cycle de vie des beans
- Démarrer et arrêter Spring

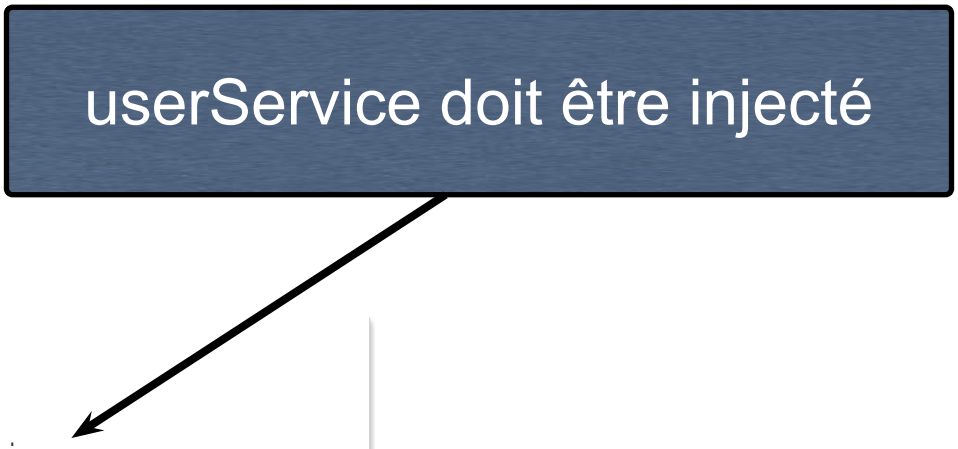
IPPON Enterprise Java Delivery

Agenda

- **L'injection de dépendances**
- IoC dans Spring
- Les scopes de beans
- Le cycle de vie des beans
- Démarrer et arrêter Spring

L'Injection de Dépendances

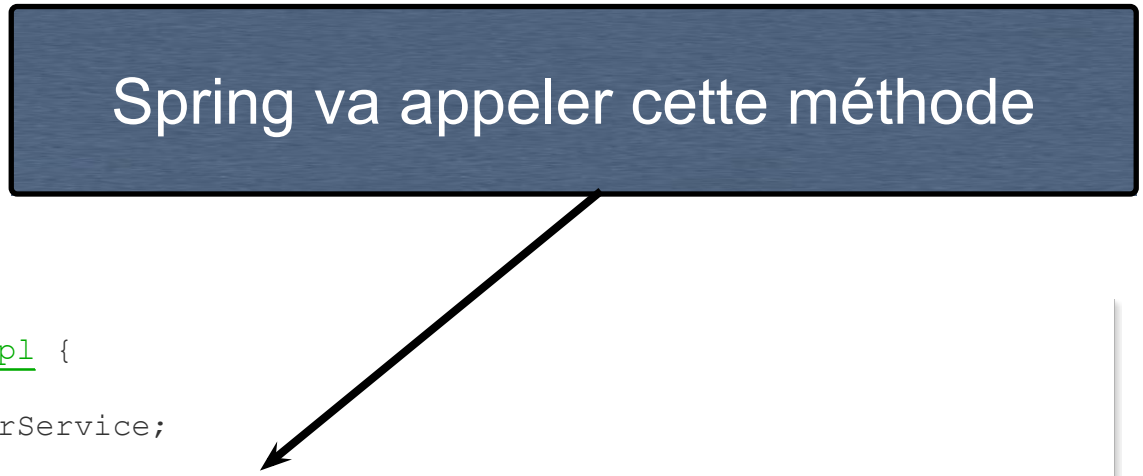
- A la base, il s'agit simplement d'injecter un objet dans un autre




```
public class TodosServiceImpl {

    private UserService userService;

}
```

L'injection par setter

- 1ère méthode, «classique» : l'injection par setter
  - Utilise la convention Java Bean

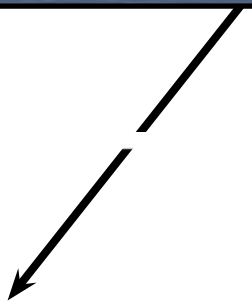



```
public class TodosServiceImpl {

    private UserService userService;

    public void setUserService(UserService userService) {
        this.userService = userService;
    }

}
```

L'injection par constructeur

- 2ème méthode, relativement populaire : utiliser le constructeur de l'objet

Spring va construire l'objet correctement



```
public class TodosServiceImpl {

    private UserService userService;

    public TodosServiceImpl(UserService userService) {
        this.userService = userService;
    }

}
```

L'injection dans un champ

- 3ème méthode : Spring injecte directement dans le champ
  - Méthode «magique» : en fait les champs «private» en Java peuvent être modifiés (si vous venez d'avoir un cours sur Java, on vous a menti)
  - De plus en plus populaire car la méthode la plus simple


```
public class TodosServiceImpl {

    private UserService userService;

}
```

Quelle méthode pour quelle utilisation ?

- Injection par setter
  - Respecte la convention JavaBeans (sans grand intérêt)
  - Héritage automatique
  - Plus clair que par constructeur
  - Permet d'avoir des dépendances optionnelles
- Injection par constructeur
  - Permet d'avoir des objets immutables
  - Oblige à avoir toutes les dépendances correctement définies
  - Plus concise que par setter
- Injection par champ
  - Mêmes qualités que par constructeur
  - Encore plus concise
  - Mais gênant pour les tests unitaires

IPPON Enterprise Java Delivery

Conseils sur l'injection

- Vous pouvez mélanger les 3 types d'injection
  - Utilisez le plus simple en fonction de votre existant
- L'injection par champ est la plus efficace pour le développement
  - Utilisez l'injection par setter pour les dépendances optionnelles
- Le plus important est d'être homogène
  - Si vous injectez votre data source de 3 manières différentes, personne ne va rien y comprendre !
  - Il est important de mettre en place des règles à ce sujet dès le début du projet

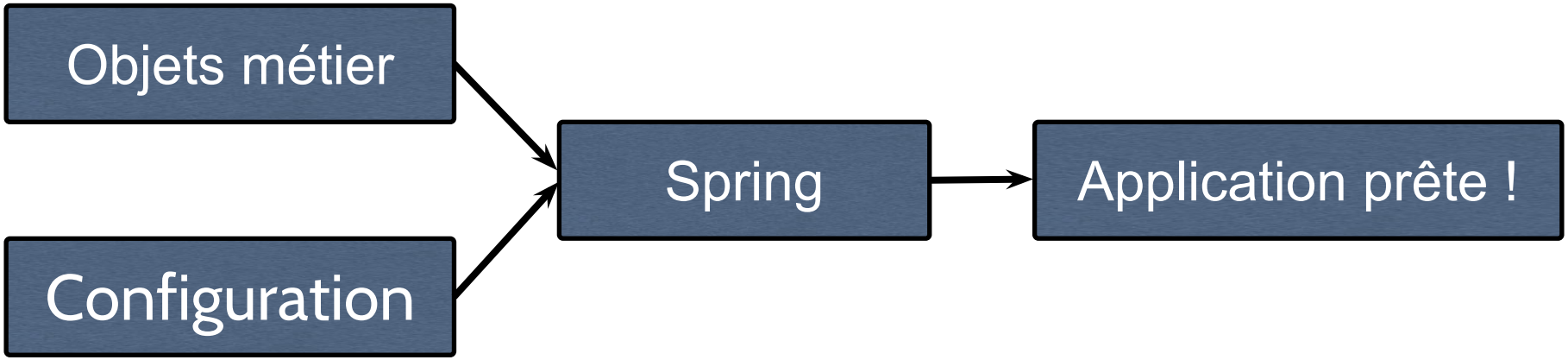
IPPON Enterprise Java Delivery

Agenda

- L'injection de dépendances
- **IoC dans Spring**
- Les scopes de beans
- Le cycle de vie des beans
- Démarrer et arrêter Spring

Les composants de Spring

- Un Application Context (une des implémentations de l'interface *org.springframework.context.ApplicationContext*) représente le conteneur Spring : il est chargé de démarrer les beans, de les injecter, de les gérer, de les détruire
- Il en existe plusieurs sortes : *WebApplicationContext* pour les applications Web par exemple
- Le rôle de cette classe est de prendre vos objets et votre configuration, et de faire fonctionner l'ensemble



IPPON Enterprise Java Delivery

Les objets métier

- Ces objets métier sont des objets Java simples
  - «POJO» pour Plain Old Java Object, un «bon vieil objet Java» en français
  - Pas forcément des JavaBeans, mais c'était le cas à l'origine, c'est pourquoi on parle de «Beans» par habitude
- Généralement ils implémentent une interface métier
  - Cela permet de découpler les objets (on injecte une interface, pas une implémentation), et donc d'avoir une architecture plus souple
  - Cela simplifie l'écriture des tests unitaires (c'est en fait une conséquence du point précédent)
  - C'est également intéressant au niveau technique, car Spring utilise cette particularité pour injecter une implémentation «enrichie» : c'est ainsi que fonctionne la Programmation Orientée Aspect, dans sa version la plus simple

Exemple d'objet métier

```
public class TodosServiceImpl implements TodosService {

    private UserService userService;

    public Collection<Todo> findAssignedTodos() {
        User user = userService.getCurrentUser();
        Set<Todo> assignedTodos = new TreeSet<Todo>();
        for (TodoList todoList : user.getTodoLists()) {
            for (Todo todo : todoList.getTodos()) {
                if (todo.getAssignedUser() != null
                        && todo.getAssignedUser().equals(user)
                        && !todo.isCompleted()) {

                    assignedTodos.add(todo);
                }
            }
        }
        return assignedTodos;
    }
}
```

Exemple de configuration Spring

- Configuration classique : via un fichier XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.1.xsd" >

  <!-- injection par setter -->
  <bean id="todoService" class="example.TodoServiceImpl" >
    <property name="userService" ref="userService" />
  </bean>

  <!-- injection par constructeur -->
  <bean id="userService" class="example.UserServiceImpl" >
    <constructor-arg ref="userRepository" />
  </bean>

</beans>
```

Configuration XML

- Utilise les «namespaces» XML
  - Spring fournit une dizaine de namespaces spécialisés : beans, transactions, sécurité, Programmation Orientée Aspect, etc...
  - L'import de ces namespaces permet de considérablement simplifier cette configuration
    - Propose l'auto-complétion et fournit la documentation
    - Fonctionne avec tout éditeur XML
    - Crée automatiquement des ensembles de Beans Spring
- Un Bean a un ID (unique) et une classe (son implémentation)
- Les Beans sont injectés
  - Par Setter avec *<property name="" ref=""/>*
  - Par constructeur avec *<constructor-arg ref=""/>*

Exemple de configuration par annotations

- Configuration «moderne» : par annotations

```
@Component
public class TodosServiceImpl implements TodosService {

    @Inject
    private UserService userService;

    public Collection<Todo> findAssignedTodos() {
        User user = userService.getCurrentUser();
        ...
        return assignedTodos;
    }
}
```

Configuration par annotations

- Il faut préciser dans le fichier XML que l'on veut utiliser les annotations

```
<beans xmlns="...">

    <context:component-scan base-package="example.test" />

</beans>
```

- Les Beans annotés @*Component* sont automatiquement créés
- Les Setters, constructeurs et champs annotés avec @*Inject* sont automatiquement injectés

Comment Spring trouve-t-il les dépendances ?

- Le plus évident : par nom
  - Pour injecter un Bean nommé «userService», Spring recherche le Bean qui a cet ID
  - C'était notre exemple de configuration XML
- Le plus concis : par type
  - On ne nomme pas le Bean à injecter : Spring recherche alors son type
  - Comprendre type au sens Java : Spring recherche alors quel Bean est de ce type là (même classe, ou bien implémentant cette interface)
  - Si Spring en trouve un, il l'injecte, et tout se passe bien
    - S'il n'en trouve pas on a alors une *Exception*, et Spring ne peut pas se lancer
    - S'il en trouve plusieurs, on a également une Exception
  - C'était notre exemple de configuration par annotations

Annotations, XML, Java ?

- Il existe en fait 3 manières de configurer Spring
  - **XML** : méthode «classique», très souple et très puissante
    - Essentielle à connaître
    - Convient très bien à la configuration dite «d'infrastructure»
  - **Annotations** : depuis Spring 2.5
    - Plus rapide à utiliser
    - Plus simple : ne convient qu'à de la configuration «métier»
  - **Java** : depuis Spring 3.0
    - Permet de coder en Java quelque chose de similaire à la configuration XML
    - Plus puissant (c'est du code, on peut faire ce qu'on veut)
    - Moins simple à modifier, en particulier pour de la configuration «d' infrastructure»
    - Moins répandu

Métier vs. Infrastructure

- Configuration «métier»
  - Les Beans codés par les développeurs du projet
  - Représentent des objets et des méthodes métier
  - Changent peu souvent
  - Ne sont pas sensibles aux environnements : ce sont les mêmes objets en développement, test et production
  - Exemple : Beans «transfert d'argent» ou «gestion des utilisateurs»
- Configuration «d'infrastructure»
  - Typiquement fournie par Spring ou un framework complémentaire
  - Représente des objets et des méthodes techniques
  - Change souvent
  - Est sensible aux environnements : objets et configuration différents en fonction de l'environnement (développement, test, production)
  - Exemple : une data source, un gestionnaire de transaction

IPPON Enterprise Java Delivery

Une configuration typique

- Des fichiers «d'infrastructure»
  - Plusieurs fichiers, découpés en fonction de leur périmètre fonctionnel
  - Exemple : un fichier pour configurer la sécurité, un fichier pour configurer la base de données
- Des annotations dans les Beans de la couche «métier»
  - Les développeurs gagnent ainsi en temps de développement
  - Facilite le refactoring
  - Recommandation : avoir un IDE qui «comprend» cette configuration

IPPON Enterprise Java Delivery

Les noms des Beans

- Tout Bean est référencé dans le conteneur Spring avec un nom unique
  - Si deux Beans ont le même nom, vous aurez une *Exception* au démarrage
- Ce nom est libre, mais par convention on utilise généralement le nom de la classe (ou de l'interface implémentée), en CamelCase, en commençant par une minuscule
  - «dataSource», «monServiceMetier», «entityManager»
  - L'homogénéité du nommage dans l'application est important
  - Bien nommer les Beans aide pour la Programmation Orientée Aspect.
    Par exemple, pour sélectionner tous les Beans dont le nom finit par «Metier».
  - A l'origine, en configuration XML, ce nom était l'ID XML du Bean
    - Garantit l'unicité dans un fichier XML (norme XML)
    - Mais cela interdisait certains caractères («/») ainsi que de donner deux noms différents au même Bean (il fallait utiliser un alias)

Nommer son Bean en XML

- Configuration «classique», avec ID XML :

```
<bean id="todoService" class="example.TodoServiceImpl" >
    <property name="userService" ref="userService" />
</bean>
```

- Alternatives :

```
<bean name="todoService" class="example.TodoServiceImpl" >
    <property name="userService" ref="userService" />
</bean>
```

```
<bean name="todoService/A, mainService" class="example.TodoServiceImpl" >
    <property name="userService" ref="userService" />
</bean>
```

Nommer son Bean par annotation

- Configuration par défaut : «todosServiceImpl»

```
@Component
public class TodosServiceImpl {

    private UserService userService;

}
```

- En nommant explicitement le Bean

```
@Component("todosService")
public class TodosServiceImpl {

    private UserService userService;

}
```

IPPON Enterprise Java Delivery

Agenda

- L'injection de dépendances
- IoC dans Spring
- **Les scopes de beans**
- Le cycle de vie des beans
- Démarrer et arrêter Spring

IPPON Enterprise Java Delivery

Les scopes de Beans

- Par défaut, les Beans Spring sont dits être des «singletons»
  - Ils ne sont instanciés qu'une seule fois par contexte Spring
  - Ils ne sont pas de «vrais» singletons : on peut lancer deux fois la même classe (deux Beans ayant la même implémentation)
- Les Beans sont instanciés et configurés au démarrage du contexte Spring
  - Permet de valider que la configuration est correcte dès le démarrage de l'application
  - Permet de gagner en performance : on ne crée pas un grand nombre d'instances d'objets pour rien

Les Beans et le multi-threading

- Le fait d'avoir des singletons a un impact en environnement multi-threadé
  - Les variables de classe sont partagées entre les threads
  - Les beans doivent donc être thread-safe

```
@Service
@Transactional
public class TodosServiceImpl implements TodosService {

    @Inject
    private UserService userService;

}
```

Singletons et données partagées

- Que faire si vous avez besoin de plusieurs instances du même Bean ?
  - Exemple très fréquent : il contient des données de l'utilisateur
- On peut donner un «scope» à un Bean
  - singleton, session, flow, request, prototype
  - prototype : une nouvelle instance à chaque fois qu'on injecte ce Bean

```
<bean id="todoService" class="example.TodoServiceImpl" scope="prototype" >
  <property name="userService" ref="userService"/>
</bean>
```

Attention !

- Que se passe-t-il si on injecte un Bean avec un scope «session» dans un Bean avec un scope «singleton» ?
  - Votre Bean «session» se retrouve injecté une seule fois dans le Bean «singleton»
  - Ce sera une variable partagée !

- Pour éviter cela, il faut configurer ce Bean spécialement :

```
<bean id="userService" class="example.UserServiceImpl" scope="session" >
  <aop:scoped-proxy/>
</bean>
```

IPPON Enterprise Java Delivery

Agenda

- L'injection de dépendances
- IoC dans Spring
- Les scopes de beans
- **Le cycle de vie des beans**
- Démarrer et arrêter Spring

Le cycle de vie des Beans

- La vie des Beans est gérée par Spring
  - C'est Spring qui crée les Beans
  - C'est Spring qui les «enrichit» avec la Programmation Orientée Aspect
  - C'est Spring qui les injecte
  - C'est Spring qui les détruit (parfois)
- Ce cycle de vie est défini par Spring
  - Spring propose également des options de configuration pour agir sur les Beans, au moment de leur création ou de leur destruction

Etape 1 : Lecture de la configuration par Spring

- Au démarrage, Spring lit sa configuration
  - Dans un fichier XML
  - Dans les annotations
  - Dans la configuration Java de Spring
- Spring possède alors un modèle mémoire de la configuration qu'on lui a fournie
  - A partir de ce moment, Spring ne différencie plus d'où provient la configuration
  - Il est donc **impossible** d'avoir un comportement différent entre une configuration XML et une configuration par annotation (= en cas de problème, c'est votre configuration qui est mauvaise)

Le BeanFactoryPostProcessor

- Spring propose à ce moment un premier point d'extension : le ***BeanFactoryPostProcessor***
  - Il permet de modifier la configuration des Beans
- Exemple typique : le ***PropertyPlaceholderConfigurer***
  - Permet de remplacer des variables (de type *${}*) par des valeurs externes à l'application, par exemple en provenance d'un fichier .properties
  - Très souvent utilisé, le fichier plat pouvant être modifié facilement par des administrateurs ou des scripts

Exemple de PropertyPlaceholderConfigurer

```
<bean class="org.springframework.beans.factory.config.
PropertyPlaceholderConfigurer" >
  <property name="locations" value="classpath:com/foo/jdbc.properties" />
</bean>
<bean id="dataSource" destroy-method="close"
      class="org.apache.commons.dbcp.BasicDataSource" >

  <property name="driverClassName" value="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
```

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```

Etape 2 : instanciation et injection des Beans

- Une fois la configuration traitée par Spring, celui-ci va créer les Beans (scope Singleton) qui ont été définis
  - Il crée les Beans en utilisant l'API Reflection de Java
  - Il crée les Beans dans le bon ordre
    - Il fait normalement l'injection de dépendance après l'instantiation des Beans
    - Sauf pour l'injection de dépendance par constructeur
    - Attention aux dépendances cycliques dans ce cas (mais elles sont généralement la preuve d'une mauvaise architecture)
- C'est au moment de l'injection que Spring crée des «proxy» sur les objets
  - Si nécessaire, Spring génère une implémentation «enrichie» des objets
  - Ajout des transactions, de la sécurité, etc...
    - C'est ainsi que fonctionne la Programmation Orientée Aspect
- Tant que l'injection de dépendance n'a pas eu lieu, les objets ne sont donc pas prêts à être utilisés

IPPON Enterprise Java Delivery

Etape 3 : initialisation des Beans

- Une fois tous les Beans créés et injectés par Spring, un nouveau point d'extension est disponible
  - Il ne faut pas utiliser les constructeurs des objets pour faire des choses complexes, car les Beans ne sont pas encore prêts (l'injection de dépendance n'est pas encore faite)
- On peut alors initialiser les Beans, avec 3 méthodes
  - Une annotation *@PostConstruct*
  - Une configuration XML (attribut «init-method»)
  - Une interface à implémenter (*InitializingBean*)
- La manière recommandée est d'utiliser l'annotation
- Elle est standardisée (JSR 250) : aucune dépendance sur Spring !
  - Elle est simple et peu intrusive

Exemple de @PostConstruct

```
@Service
@Transactional
public class TodosServiceImpl {

    @PostConstruct
    public void init() {
        System.out.println("Bonjour de TodosServiceImpl");
    }
}
```

- Astuce : c'est un bon moyen pour vérifier si vos Beans sont bien démarrés

Les BeanPostProcessor

- Les phases d'injection de dépendance et d'initialisation (étapes 2 & 3) peuvent être personnalisées
- Une fois un Bean créé, Spring laisse en fait une chaîne de *BeanPostProcessor* modifier l'instance de ce Bean
  - Ainsi, le *@PostConstruct* que nous venons de voir est traité par un *BeanPostProcessor*
  - Attention, nous avons vu auparavant les *BeanFactoryPostProcessor*
    - *BeanFactoryPostProcessor* : manipule les configurations XML des Beans
    - *BeanPostProcessor* : manipule les instances des Beans

Etape 4 : Run !

- Une fois les Beans instanciés, injectés et initialisés le conteneur Spring est prêt à l'emploi
  - Il devient accessible (on peut lui demander un Bean via l'API)
  - L'application est prête et est démarrée
- C'est dans cette phase que votre application se trouve 99% du temps
  - Sauf si elle crashe souvent :-)

Etape 5 : Destruction

- On peut demander à Spring de s'arrêter
  - Manuellement via l'API
  - Lorsque votre conteneur Web s'arrête
- Lors de cette phase d'arrêt, tous les Beans Spring vont s'arrêter
- Cette phase est **optionnelle**
  - L'application peut tout simplement crasher : dans ce cas cette phase n'est évidemment pas prise en compte
  - Cela ne fonctionne pas pareil pour tous les Beans
    - Aucun souci pour les Singletons
    - Les Prototypes, par nature, ne sont pas arrêtés par Spring («fire and forget»)

Destruction des Beans

- Même principe que pour l'initialisation, avec 3 méthodes
  - Une annotation @*PreDestroy*
  - Une configuration XML (attribut attribut «destroy-method»)
  - Une interface à implémenter (*DisposableBean*)
- L'annotation sera aussi privilégiée

Exemple de @PreDestroy

```
@Service
@Transactional
public class TodosServiceImpl {

    @PreDestroy
    public void destroy() {
        System.out.println("Au revoir de TodosServiceImpl");
    }
}
```

- ATTENTION : pour un Prototype cette méthode ne sera pas appelée

Un cas particulier : le lazy-loading

- Que faire si vos beans sont lents au démarrage ?
  - Un exemple possible : l'initialisation d'un cache
- On ne va pas vouloir suivre le cycle de vie classique : on veut avoir des Beans uniquement instanciés à la demande : c'est le «lazy loading»
- C'est généralement une fausse bonne idée :
  - Le premier utilisateur à utiliser cet objet va alors subir le chargement
  - Il y a toujours un risque de mauvaise configuration : mieux vaut être sûr du bon lancement de ses Beans au démarrage de l'application
  - Une solution : avoir une configuration d'infrastructure différente suivant votre environnement, et ne faire le lazy loading qu'en développement

```
<bean id="todoService" class="example.TodoServiceImpl" lazy-init="true" >
  <property name="userService" ref="userService"/>
</bean>
```

IPPON Enterprise Java Delivery

Résumé sur le cycle de vie des Beans

- Ce cycle de vie est toujours respecté par Spring
- Il est prévu à l'origine pour des Singletons, qui sont donc tous démarrés, injectés et initialisés au démarrage de l' application
  - Cela vous assure d'avoir une configuration fonctionnelle une fois l' application démarrée
  - Cela vous assure aussi d'excellentes performances : on travaille uniquement avec des Singletons, qui sont déjà prêts à être utilisés
- Mais dans certains cas particuliers, ce cycle n'a pas lieu au démarrage de l'application : les beans qui ne sont pas des Singletons, et les beans utilisant le lazy loading

Agenda

- L'injection de dépendances
- IoC dans Spring
- Les scopes de beans
- Le cycle de vie des beans
- **Démarrer et arrêter Spring**

Démarrer Spring

- Pour démarrer Spring, il faut créer une instance de l'interface *ApplicationContext*
  - Plusieurs implémentations existent
  - Elles sont spécialisées pour certains environnements : application Web, test unitaire, etc...
  - Elles sont toutes sur le même principe : il faut charger la configuration Spring (habituellement, charger le fichier de configuration XML)
- La manière la plus simple :

```
ApplicationContext ctx =
        new ClassPathXmlApplicationContext( "application-context.
xml");
```

Astuces de configuration

- Pour lire le fichier dans le système de fichier :

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext( "/home/application/config.xml" );
```

- Pour lire plusieurs fichiers (varargs) :

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext( "conf-1.xml", "conf-2.xml");
```

- Pour lire tous les fichiers d'un répertoire (pattern Ant) :

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext( "/home/application/*.xml" );
```

Démarrage dans une application Web

- Spring est lancé via un listener dans le fichier web.xml

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:META-INF/spring/application-context.xml</param-value>
</context-param>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

- Note : cette configuration lance uniquement Spring IoC, sans Spring MVC (que nous verrons plus tard)

Arrêter Spring

- L'application peut parfaitement être arrêtée «normalement» ou crasher...
- Si vous voulez explicitement arrêter Spring :

```
applicationContext.close();
```

- Les Beans sont informés qu'ils doivent s'arrêter (*@PreDestroy* est appelé)
- Les Beans sont détruits
- Le context Spring n'est alors plus utilisable

Conclusion sur Spring IoC

- Nous avons vu les bases de la configuration de Spring
  - La configuration, en XML et par annotation
  - L'injection de dépendances
  - Le cycle de vie des Beans
- Ces concepts vont nous servir tout au long de ces 4 jours de cours
  - Il est donc essentiel de bien les maîtriser
  - Nous allons les revoir à plusieurs reprises

Préparation aux exercices

Installation de l'environnement de travail

- Dézippez le fichier fourni avec ce cours, contenant les TPs
- Il s'agit d'un projet Maven
  - Maven est disponible dans le répertoire «00_build»
  - Configurez le fichier «conf/settings.xml» de Maven si nécessaire
  - Un «*mvn package*» à la racine doit fonctionner
- A chaque exercice correspondent ensuite plusieurs sous-projets Maven : l'exercice lui-même et sa (ou ses) correction(s)
  - Vous pouvez développer avec l'IDE de votre choix
  - Le sujet de l'exercice se trouve dans le projet «exercice», et se nomme EXERCICE.txt

Exercice 1
Configuration Spring simple

Configuration Spring avancée

IPPON Enterprise Java Delivery

Agenda

- L'injection d'objets complexes
- Le ProperyPlaceholderConfigurer
- Les inner beans
- L'héritage de beans
- Bien découper ses fichiers de configuration
- Les profils
- Les application context hiérarchiques

IPPON Enterprise Java Delivery

Agenda

- **L'injection d'objets complexes**
- Le ProperyPlaceholderConfigurer
- Les inner beans
- L'héritage de beans
- Bien découper ses fichiers de configuration
- Les profils
- Les application context hiérarchiques

IPPON Enterprise Java Delivery

Injecter autre chose que des Beans

- Spring ne se limite pas à l'injection de Beans
- Vous pouvez injecter :
  - Des objets simples : int, String ...
  - Des objets complexes : properties, fichiers ...
  - Des ensembles d'objets : List, Set, Map ...
- Cela permet d'avoir des configurations d'infrastructure très complètes et très souples
- De par la complexité de ces configurations, elles sont essentiellement réalisées via XML ou en Java (pas en annotations)

Injection de types primaires

- Vous pouvez injecter n'importe quel type primaire dans un Bean Spring
  - Spring va automatiquement convertir la chaîne de texte dans le bon type

```
<bean id="dataSource"
          class="com.mchange.v2.c3p0.ComboPooledDataSource"
          destroy-method="close">

  <property name="driverClass" value="com.mysql.jdbc.Driver" />
  <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/tudu" />
  <property name="user" value="system"/>
  <property name="password" value="manager"/>
  <property name="initialPoolSize" value="10"/>
  <property name="minPoolSize" value="10"/>
  <property name="maxPoolSize" value="30"/>
</bean>
```

Injection d'objets complexes

- Spring est également capable d'injecter des objets plus complexes
  - Fichiers, properties, patterns (expressions régulières), URL...
  - Ce mécanisme peut également être étendu à vos propres objets

```
<bean id="fileBean" class="example.FileBean">
  <property name="fileToInject" value="classpath:test/monfichier.txt" />
</bean>
```

Injection de Collections

- On peut également injecter des collections Java
  - List, Set, Map…

```
<property name="emails">
  <list>
    <value>formation@ippon.fr</value>
    <value>contact@ippon.fr</value>
    <value>recrutement@ippon.fr</value>
  </list>
</property>
```

```
<property name="emails">
  <map>
    <entry key="formation" value="formation@ippon.fr" />
    <entry key="contact" value="contact@ippon.fr" />
    <entry key="recrutement" value="recrutement@ippon.fr" />
  </map>
</property>
```

Le namespace «util»

- Le namespace «util» permet de simplifier cette configuration

```
<util:list id="emails">
  <value>formation@ippon.fr</value>
  <value>contact@ippon.fr</value>
  <value>recrutement@ippon.fr</value>
</util:list>
```

```
<util:map id="emails" map-class="java.util.TreeMap">
  <entry key="formation" value="formation@ippon.fr" />
  <entry key="contact" value="contact@ippon.fr" />
  <entry key="recrutement" value="recrutement@ippon.fr" />
</util:map>
```

IPPON Enterprise Java Delivery

Agenda

- L'injection d'objets complexes
- **Le ProperyPlaceholderConfigurer**
- Les inner beans
- L'héritage de beans
- Bien découper ses fichiers de configuration
- Les profils
- Les application context hiérarchiques

Le PropertyPlaceholderConfigurer

- Généralement, ces variables injectées sont des paramètres, qui dépendent de l'environnement
  - Elles se trouvent dans la configuration «d'infrastructure»
  - Par exemple : l'URL, le login et le mot de passe de la base de données
- Il est plus pratique de les externaliser dans un fichier de properties

```
<context:property-placeholder
     location="classpath*:META-INF/infrastructure/database.properties"/>

<bean id="dataSource"
     class="com.mchange.v2.c3p0.ComboPooledDataSource">

  <property name="user" value="${dataSource.username}"/>
  <property name="password" value="${dataSource.password}"/>
    ...
</bean>
```

```
dataSource.username=system
dataSource.password=manager
```

IPPON Enterprise Java Delivery

Agenda

- L'injection d'objets complexes
- Le ProperyPlaceholderConfigurer
- **Les inner beans**
- L'héritage de beans
- Bien découper ses fichiers de configuration
- Les profils
- Les application context hiérarchiques

Les inner Beans

- Même principe que les inner class en Java
- Un inner Bean est déclaré à l'intérieur d'un autre Bean
  - Il est injecté dans ce deuxième Bean
  - Seul ce deuxième Bean peut le «voir»
  - Clarifie la configuration
  - Il n'y a pas de limite : on peut faire des inner Beans de inner Beans…

```
<bean id="beanA" class="example.BeanA">
  <property name="beanB">
    <bean class="example.BeanB">
      <property name="prop1" value="ABC"/>
      <property name="prop2" value="123"/>
    </bean>
  </property>
</bean>
```

Combinaison de ces dernières techniques

- Un exemple concret de **configuration Hibernate**

```
<bean id="entityManagerFactory"
   class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

  <property name="dataSource" ref="dataSource"/>
  <property name="jpaVendorAdapter">
    <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
      <property name="database" value="${jpavendoradapter.database}"/>
      <property name="databasePlatform" value="${jpavendoradapter.databaseplatform}"/>
      <property name="showSql" value="${jpavendoradapter.showsql}"/>
      <property name="generateDdl" value="${jpavendoradapter.generateddl}"/>
    </bean>
  </property>
  <property name="persistenceXmlLocation" value="classpath:META-INF/persistence.xml"/>
</bean>
```

Combinaison de ces dernières techniques

- Un exemple concret de **configuration Spring MVC**

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
  <property name="mediaTypes">
    <map>
      <entry key="html" value="text/html"/>
      <entry key="json" value="application/json"/>
    </map>
  </property>
  <property name="viewResolvers">
    <list>
      <bean class="org.springframework.web.servlet.view.UrlBasedViewResolver">
        <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
        <property name="prefix" value="/WEB-INF/views/"/>
        <property name="suffix" value=".jsp"/>
      </bean>
    </list>
  </property>
  <property name="defaultViews">
    <list>
      <bean class="org.springframework.web.servlet.view.json.MappingJacksonJsonView"/>
    </list>
  </property>
 </bean>
```

IPPON Enterprise Java Delivery

Agenda

- L'injection d'objets complexes
- Le ProperyPlaceholderConfigurer
- Les inner beans
- **L'héritage de beans**
- Bien découper ses fichiers de configuration
- Les profils
- Les application context hiérarchiques

L'héritage de Beans

- Même principe que l'héritage Java
  - On peut surcharger des classes ou des properties
  - On peut même faire des Beans abstraits

```
<bean id="parentBeanDefinition" abstract="true">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</bean>

<bean id="childBean" class="example.childBean"
  parent="parentBeanDefinition" init-method="initialize">

  <property name="name" value="override"/>
  <!-- l'âge est hérité, il est donc de «1» -->
</bean>
```

IPPON Enterprise Java Delivery

Agenda

- L’injection d’objets complexes
- Le ProperyPlaceholderConfigurer
- Les inner beans
- L’héritage de beans
- **Bien découper ses fichiers de configuration**
- Les profils
- Les application context hiérarchiques

Bien découper les fichiers de configuration

- Nous avons vu qu'il y a deux types de fichiers de configuration
  - Les fichiers «d'infrastructure»
  - Les fichiers «métier»
- Nous avons également vu que les Beans métier étaient généralement configurés par annotation
  - Ils changent peu
  - L'objectif est de les coder le plus efficacement possible
- Les fichiers «d'infrastructure» changent souvent, et sont différents d'un environnement à l'autre
  - Il faut donc les découper en fonction de l'environnement

Exemple de découpage (1/2)

- La configuration métier est réalisée à l'aide d'annotations
  - Elle ne change pas, quel que soit l'environnement
  - Elle est ainsi plus rapide à réaliser pour les développeurs
  - On utilise l'auto-wiring par type : on ne nomme même pas les Beans

Exemple de découpage (2/2)

- La configuration d'infrastructure est stockée dans plusieurs fichiers XML :
  - Tous sont dans le même répertoire : *classpath:META-INF/spring/*.xml*
  - Ces fichiers sont spécialisés : *applicationContext-database.xml, applicationContext-security.xml*, etc...
  - Ces fichiers sont paramétrables via des fichiers de properties (utilisation du *PropertyPlaceholderConfigurer*)
  - Il est également possible d'avoir des fichiers de configuration spécifiques par environnement, si les fichiers de properties ne sont pas suffisants

IPPON Enterprise Java Delivery

Agenda

- L'injection d'objets complexes
- Le ProperyPlaceholderConfigurer
- Les inner beans
- L'héritage de beans
- Bien découper ses fichiers de configuration
- **Les profils**
- Les application context hiérarchiques

Les profils

- Les profils sont une nouveauté Spring 3.1
  - Ils permettent de simplifier le découpage en fonction de l'environnement
  - Ils fonctionnent en configuration XML et Java (annotation *@Profile* à placer sur le Bean)
  - Ils simplifient encore plus la configuration d'infrastructure

```
<beans profile="production">
  <jee:jndi-lookup id="dataSource"
      jndi-name="java:comp/env/jdbc/datasource"/>
</beans>
```

Choisir un profil

- Pour choisir un profil au démarrage
  - Utiliser une variable d'environnement :

```
-Dspring.profiles.active=production
```

  - Le configurer au démarrage via l'API :

```
GenericXmlApplicationContext context =  new
GenericXmlApplicationContext();
context.getEnvironment().setActiveProfiles("prod");
context.load("classpath:META-INF/spring/applicationContext-*.xml" );
context.refresh();
```

IPPON Enterprise Java Delivery

Agenda

- L'injection d'objets complexes
- Le ProperyPlaceholderConfigurer
- Les inner beans
- L'héritage de beans
- Bien découper ses fichiers de configuration
- Les profils
- **Les application context hiérarchiques**

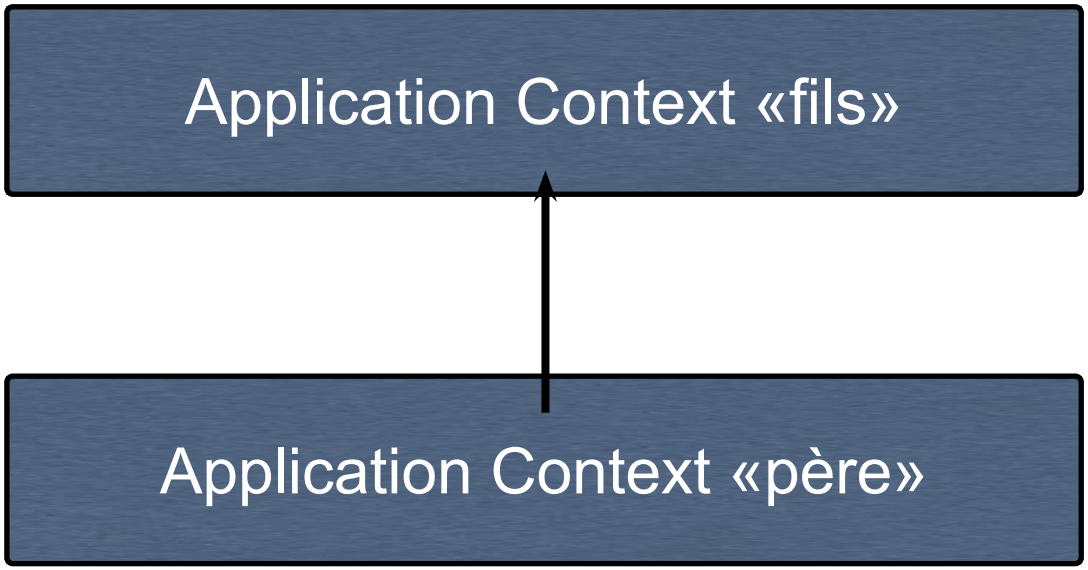
IPPON Enterprise Java Delivery

Les «application context» hiérarchiques

- Pour l'instant, nous avons vu que Spring permettait de créer un «application context», dans lequel étaient configurés un certain nombre de Beans
- En fait on peut créer plusieurs «application context» et les mettre dans une hiérarchie
- Cette notion est importante, car elle est très fréquemment utilisée : Spring MVC et Spring Web Services fonctionnent de cette manière

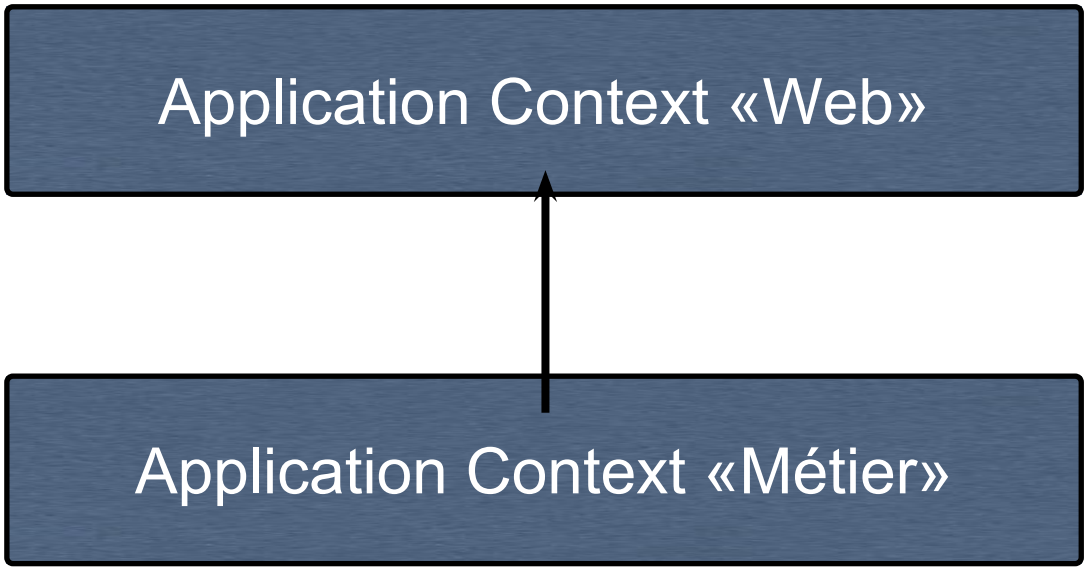
Fonctionnement de la hiérarchie des «application contexts»

- Cette hiérarchie «d'application contexts» permet de donner une visibilité aux Beans
  - Les Beans enfants «voient» les Beans parents
  - L'inverse n'est pas vrai



Exemple avec Spring MVC

- On peut injecter un Bean «métier» dans Spring MVC
  - C'est ainsi qu'on appelle la couche métier depuis la couche Web
- On ne peut pas faire l'inverse
  - Ce serait une erreur en termes d'architecture
  - Cela permet aussi de simplifier la configuration

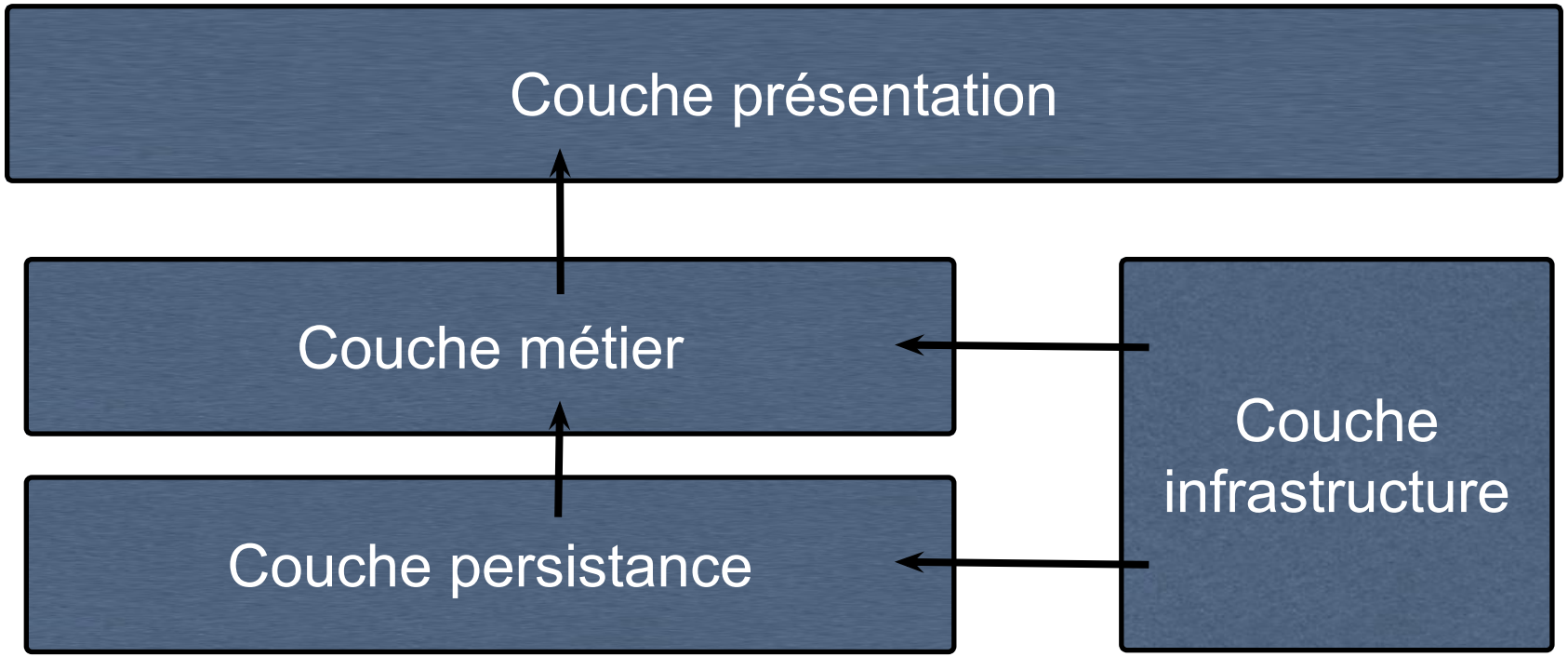


Exercice 2
Configuration Spring avancée

Architecture d'applications Spring

Architecture «classique»

- Les applications Spring «classiques» sont découpées en 4 couches



Avantages de cette architecture

- Cette architecture est tellement bien intégrée à Spring qu'il y a même des annotations spéciales : *@Controller*, *@Service* et *@Repository*
  - Permettent de facilement configurer son application
  - Facilitent l'utilisation de l'AOP : il est très simple de faire un aspect sur l'ensemble de la couche "repository"
- Cette architecture est très performante et a fait ses preuves
  - L'ensemble des Beans Spring sont des Singletons
  - Les entités Hibernate sont simples et traversent les différentes couches

Améliorations de cette architecture

- La couche «persistance» (ou repository, ou DAO) est de moins en moins pertinente : c'est Hibernate qui remplit cette fonction
  - On peut la supprimer facilement
- Les entités Hibernate, qui contiennent les données métier, traversent ainsi toutes les couches et n'ont aucune intelligence
  - C'est un modèle de données «anémique»
  - On peut facilement leur donner plus d'intelligence avec des outils comme Bean Validation

IPPON Enterprise Java Delivery

Design Pattern «Active Record»

- Popularisé par Ruby on Rails
- Adopté en particulier par Spring Roo
- Des Beans Spring peuvent être injectés dans les entités Hibernate
  - Les entités Hibernate contiennent des méthodes CRUD, voire même des méthodes métier
  - C'est la suppression de la couche «Persistance» et la fusion de la couche «Métier» avec les entités Hibernate
  - La couche «métier» n'est utile que si plusieurs entités différentes sont utilisées dans la même fonction
- Problème : pour que cela fonctionne il faut utiliser AspectJ
  - Plus grande complexité

IPPON Enterprise Java Delivery

Spring avancé

- Spring AOP
- Spring JDBC
- Les transactions
- Les tests avec Spring

Spring AOP

IPPON Enterprise Java Delivery

Introduction à l'AOP

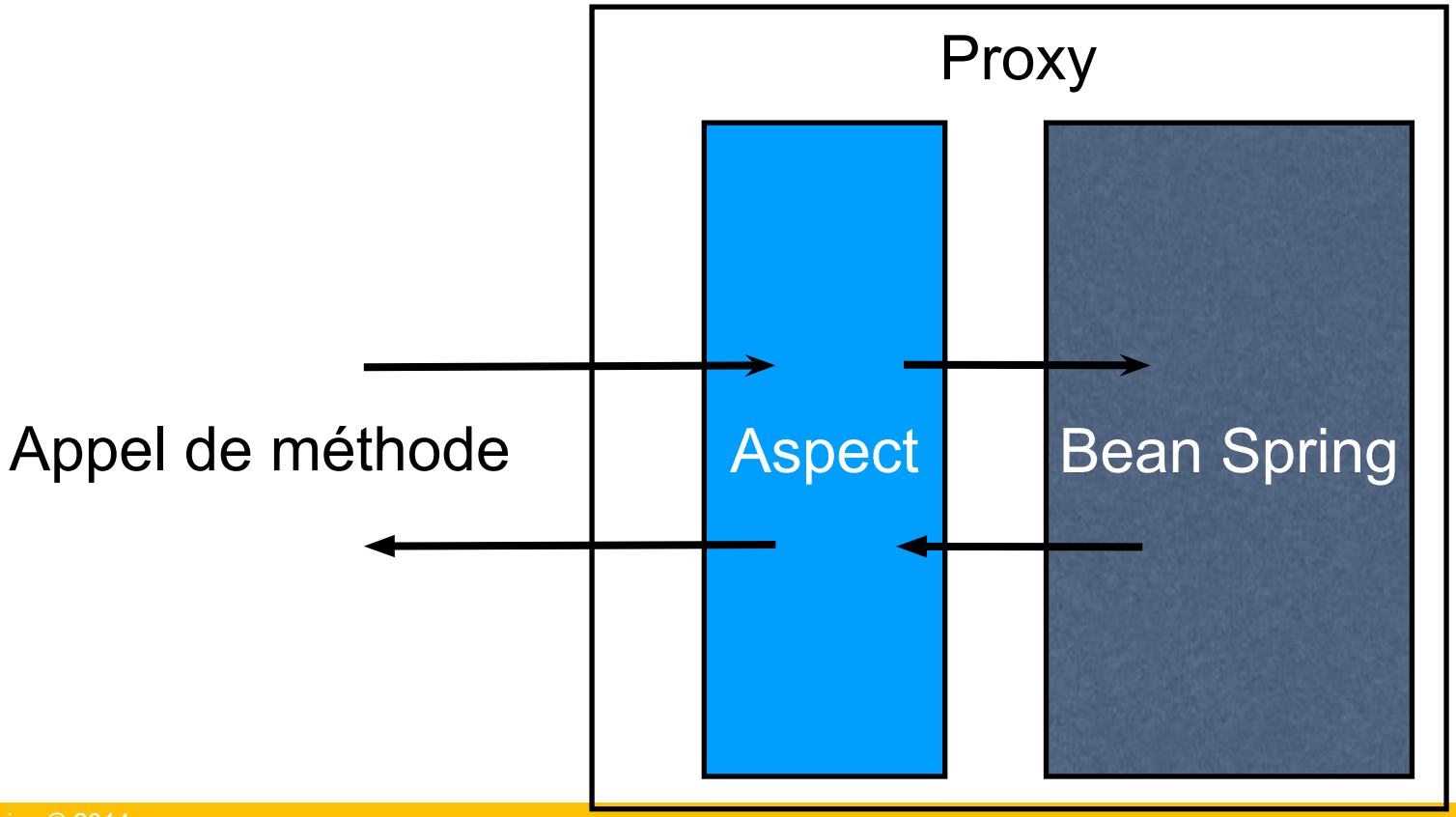
- **AOP** : **A**spect **O**riented **P**rogramming, ou Programmation Orientée Aspect en Français
- L'un des deux concepts principaux de Spring (avec l' Inversion de Contrôle)
- Permet de rajouter des comportements à des classes ou des méthodes existantes
  - Ajouter de la sécurité
  - Ajouter la gestion des transactions
  - Ajouter du monitoring
- Il s'agit de problématiques transverses
  - Elles sont généralement techniques (infrastructure)
  - Elles sont plus rarement métier

Spring AOP ou AspectJ

- Spring AOP utilise un mécanisme de proxy
  - Ne fonctionne que sur des Beans Spring, et son utilisation est très simple
  - Est largement suffisant pour des besoins "normaux", ce qui fait que la très grande majorité des utilisateurs de Spring utilise Spring AOP
- AspectJ est une technologie nettement plus complète et complexe
  - Repose sur une modification du bytecode des classes Java
  - Permet de faire des choses nettement plus compliquées : injecter des Beans Spring dans des classes standards, par exemple
  - Est plus performant en production (mais cela a un impact mineur dans la réalité : à comparer avec un accès base de données)
- Spring AOP utilise le même "langage" que AspectJ, ce qui fait que l'on peut facilement migrer de Spring AOP vers AspectJ
- Nous allons donc nous concentrer sur Spring AOP

Fonctionnement de Spring AOP

- Un proxy «enrobe» le Bean Spring
  - Il implémente la même interface, et peut ainsi le remplacer



Fonctionnement des proxys

- Ce sont normalement des proxys Java
  - Technologie standard Java (introduite dans le JDK 1.3)
  - Aujourd'hui suffisamment performants (impact mineur)
  - Nécessitent l'utilisation d'une interface
  - L'utilisation d'interfaces est recommandée de toute manière (pour les tests et la souplesse de l'application)
- Si vos classes n'implémentent pas d'interface
  - Spring AOP va utiliser CGLIB pour générer le proxy
  - CGLIB est une librairie Open Source qui permet de générer des classes à la volée
  - Complique de débuggage et les stack traces
  - Aujourd'hui cette technologie est fiable, mais les proxys Java restent à privilégier

Exemple : les transactions

- Les transactions sont l'un des Aspects techniques fournis en standard par Spring

```
@Service
@Transactional
public class TodosServiceImpl implements TodosService {

    @PersistenceContext
    private EntityManager em;

    @Transactional(readOnly = true)
    public Todo findTodo(final String todoId) {
        return em.find(Todo.class, todoId);
    }
}
```

Que se passe-t-il si on a plusieurs Aspects sur le même Bean ?

- Exemple : une méthode est transactionnelle et sécurisée
- Spring ne génère qu'un seul proxy
- Spring va enchaîner ces Aspects
- L'ordre de ces Aspects peut être paramétré avec l'interface *org.springframework.core.Ordered* ou l'annotation *@Order*

IPPON Enterprise Java Delivery

Les concepts de l'AOP

- **Join point** : l'endroit où l'on veut qu'un aspect s'applique. Avec Spring AOP, il s'agit toujours d'une méthode (du fait de l'utilisation de proxy)
- **Pointcut** : une expression, utilisant la syntaxe AspectJ, qui permet de sélectionner plusieurs Join points. Par exemple, «toutes les méthodes qui se nomment find()».
- **Advice** : le code que l'on veut rajouter. On peut ajouter ce code avant, après, autour de la méthode...
- **Aspect** : Pointcut + Advice

Les types d'advices

- Il est possible de définir 5 types d'advices
  - **Before advice** : s'exécute avant le Join point. S'il lance une *Exception*, le Join point ne sera pas appelé
  - **After returning advice** : s'exécute après le Join point, si celui-ci s'est bien exécuté (s'il n'y a pas eu d'*Exception*)
  - **After throwing advice** : s'exécute si une *Exception* a été lancée pendant l'exécution du Join point
  - **After advice** : s'exécute après le Join point, qu'il y ait eu une Exception ou non
  - **Around advice** : s'exécute autour du Join point. C'est l'advice le plus puissant.

Configuration des aspects

- Plusieurs configurations sont possibles
  - En XML
  - En utilisant des annotations, dite méthode «@AspectJ»
- La méthode @AspectJ est à privilégier
  - Plus simple à utiliser
  - Permet d'avoir des Pointcut et les Advice au même endroit
- Pour pouvoir utiliser la méthode @AspectJ, ajouter dans votre configuration Spring :

```
<aop:aspectj-autoproxy/>
```

Définition d'un aspect

- Un Aspect est également un Bean Spring
  - Mais il ne peut pas y avoir d'Aspect sur un autre Aspect
  - On peut séparer le Pointcut de l'Advice, mais c'est plus lisible de tout mettre dans le même fichier

```
@Aspect
@Component
public class Monitor {

    @Before("execution(* find(*))")
    public void monitorFinders() {
        System.out.println("A Finder is fired!");
    }
}
```

Le langage AspectJ

- La complexité vient en particulier de l'utilisation du langage AspectJ
  - Les bases du langage sont simples, mais il peut vite devenir très complexe
  - Il peut être difficile de sélectionner l'ensemble des méthodes voulues
  - Il faut faire attention à ne pas sélectionner d'autres méthodes par erreur
  - L'outillage, en particulier d'Eclipse (plug-in AspectJ) permet de considérablement simplifier cette tâche

Introduction au langage AspectJ

- Spring AOP supporte un sous-ensemble du langage AspectJ
- Une expression Spring AOP/AspectJ est de la forme suivante :

```
execution(modifiers-pattern? ret-type-pattern
declaring-type-pattern? name-pattern(param-pattern)
throws-pattern?)
```

- C'est une sorte d'expression régulière ou de requête qui permet de sélectionner les méthodes à instrumenter

Exemples de Pointcut (1/3)

- Exécution de toutes les méthodes publiques

```
execution(public * *(..))
```

- Exécution de tous les getters

```
execution(* get*(..))
```

- Exécution de tous les getters qui retournent des String

```
execution(* get*(..))
```

Exemples de Pointcut (2/3)

- Exécution de toutes les méthodes de l'interface *ExampleService*

```
execution(* example.ExampleService.*(..))
```

- Exécution de toutes les méthodes du package *example.test*

```
execution(* example.test.*.*(..))
```

- Exécution de toutes les méthodes du package *example.test* et de ses sous-packages

```
execution(* example.test..*.*(..))
```

Exemples de Pointcut (3/3)

- Toutes les méthodes annotées avec @*Transactional*

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

- Toutes les méthodes du Bean nommé «testService»

```
bean(testService)
```

- Toutes les méthodes de tous les Beans dont le nom se termine par «Service»

```
bean(*Service)
```

Les PointCuts peuvent être combinés

- On peut nommer des PointCuts afin de les réutiliser
- On peut les combiner avec les opérateurs logiques «&&», «||» et «!»

```
@Pointcut("execution(public * get*(..))")
private void tousLesGetters() {}

@Pointcut("execution(example.metier..*)")
private void toutesLesMethodesMetier() {}

@Before("tousLesGetters() && toutesLesMethodesMetier()")
private void tousLesGettersMetier() {
    // Code métier...
}
```

Accès aux arguments des méthodes

- Il est intéressant d'avoir accès aux arguments des méthodes sur lesquelles les Aspects s'appliquent
- On peut les «binder» sur l'Advice

```
@Before("bean(*Service) && args(account,..)")
public void invoiceAccount(Account account) {
    // Facture le compte pour tous les appels à une méthode "métier"
}
```

Accès au Join Point

- Pour accéder au Join Point, il suffit de passer l'objet en premier paramètre de l'Advice
- Vous avez alors accès à :
  - *Args* : les arguments passés à la méthode
  - *Signature* : la signature de la méthode
  - *Target* : le Bean sur lequel s'applique l'Aspect
  - *This* : l'objet en cours (le proxy entourant le Bean)

```
@Before("* example.test..*.*(..)")
public void testMethod(JoinPoint joinPoint) {
    System.out.println(joinPoint.getSignature());
}
```

Exemples d'Aspects

- Un Advice «Before», qui s'exécute avant l'exécution des méthodes métier
  - Si cet Advice lance une *Exception*, le code métier ne sera pas appelé
  - C'est le fonctionnement de Spring Security

```
@Aspect
public class BeforeExample {

    @Before("execution(* example.metier.*.*(..))" )
    public void controlAcces() {
        // Vérification accès métier
    }
}
```

Exemples d'Aspects

- Un Advice «After Throwing», qui s'exécute si la méthode a lancé une *Exception*
  - C'est ainsi que fonctionne la couche «Repository» en Spring : elle peut ainsi traiter automatiquement les *Exceptions* provenant de la base de données
  - Il existe aussi «After returning» et «After» (finally)

```
@Aspect
public class AfterThrowingExample {

    @AfterThrowing(
       pointcut="@annotation(org.springframework.stereotype.Repository)" ,
       throwing="e")
    public void exceptionHandler(e) {
        mailService.sendEmailAlert(e);
    }
}
```

Exemples d'Aspects

- Un Aspect «Around»
  - Doit avoir en premier argument un *ProceedingJoinPoint*, qui est une sous-classe de *JoinPoint*
  - Permet de se passer totalement de l'appel à la méthode sur laquelle s'applique l'Aspect (on peut remplacer la méthode par une autre)

```
@Aspect
public class AroundExample {

    @Around("* example.test.*.*(..)")
    public Object profileMethods(ProceedingJoinPoint pjp) {
        long start = Calendar.getInstance().getTimeInMillis();
        Object result = pjp.proceed();
        System.out.println(Calendar.getInstance().getTimeInMillis() - start);
        return result;
    }
}
```

Utilisation concrète des Aspects

- Spring fournit de nombreux Aspects techniques
  - Sécurité
  - Transaction
  - Gestion des *Exceptions*
- Vous pouvez faire vos propres Aspects
  - Logging
  - Monitoring
  - Fonctionnalité métier particulière

Astuce de configuration 1

- Les annotations facilitent considérablement l'écriture des Point Cut
  - On matche sur l'annotation et non sur le package ou le nom de la méthode
  - Exemple : l'annotation @*Transactionnal* plutôt que «toutes les méthodes qui commencent par tx»

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

Astuce de configuration 2

- Nous avons déjà vu qu'il fallait découper les fichiers de configuration Spring en fonction des fonctionnalités
  - Vous pouvez avoir une configuration Spring lisant tous les fichiers suivants : WEB-INF/spring/aop/*.xml
  - Il vous suffit ensuite de copier/coller dans ce répertoire la configuration des Aspects que vous voulez appliquer sur votre projet
  - Par exemple : ajouter ou enlever un Aspect monitorant les méthodes métier
  - Il est ainsi possible d'ajouter ou de modifier des fonctionnalités de l'application, sans modifier son code source

Tester unitairement un Aspect

- Les Aspects que nous avons vus sont des Beans Spring normaux, avec des annotations
  - Pas d'interface ou d'héritage particulier
- Ils sont donc testables unitairement comme n'importe quelle classe Java simple
  - Ces tests sont d'autant plus importants qu'un Aspect est censé renfermer une fonctionnalité particulière que l'on veut appliquer à de nombreux autres objets

Limitations de Spring AOP

- Ces limitations concernent Spring AOP, mais pas AspectJ
  - Ne fonctionne qu'avec les Beans Spring
  - Ne s'applique qu'aux méthodes publiques
  - Ne fonctionne pas à l'intérieur d'un même Bean
- Cela est dû à l'utilisation des Proxys (on «passe» par une interface Java)
- Le 3ème point (une méthode d'un Bean qui appelle directement une autre méthode du même Bean) est l'une des plus grandes sources de bugs et d'incompréhension sur les projets Spring

IPPON Enterprise Java Delivery

Conclusion sur les Aspects

- C’est une technologie largement utilisée sur les projets Spring
  - Fiable
  - Très puissante
- Elle permet de rajouter dynamiquement du code à des méthodes, de manière transverse et non intrusive
- Bien comprendre son fonctionnement est essentiel
  - Pour comprendre comment fonctionne Spring
  - Pour pouvoir faire soi-même des Aspects si besoin

Exercice 3
Spring AOP

Spring JDBC

IPPON Enterprise Java Delivery

Une abstraction ?

- Nous allons voir la 3ème (et dernière) grande fonctionnalité de Spring
  - Les deux premières : Inversion de Contrôle et Programmation Orientée Aspect
  - La 3ème : la mise à disposition d'abstractions, qui simplifient des problématiques techniques courantes
- Ces abstractions ne peuvent fonctionner que grâce à l'AOP, qui lui-même est permis grâce à l'IoC

Introduction à Spring JDBC

- Spring JDBC est la première **abstraction** que nous allons étudier
  - C'est l'une des plus simples
  - Nous allons vite voir qu'elles fonctionnent toutes de la même manière, et qu'une fois le principe compris il est très aisé d'en apprendre une nouvelle
- Spring JDBC va nous aider à faire des requêtes en base de données
  - Gestion des accès à la base de données
  - Aide à l'exécution du SQL
  - Gestion des Exceptions

IPPON Enterprise Java Delivery

Spring JDBC : pour quoi faire ?

- Spring JDBC simplifie l'utilisation de JDBC
  - Mais il est nettement plus basique qu'Hibernate
- Il n'est donc à utiliser que dans des cas assez particuliers
  - Un projet très simple sur lequel on ne veut pas utiliser Hibernate
  - Un point précis d'un projet, pour lequel on ne veut pas utiliser Hibernate afin d'être plus proche de la base de données

Configuration de *JdbcTemplate*

- *JdbcTemplate* est la classe principale de Spring JDBC
  - Elle est thread-safe
  - Elle est parfois configurée en tant que Bean Spring
- Voici la configuration la plus classique
  - *@Repository* hérite de *@Component* : cette annotation marque un Bean Spring de la couche d'accès aux données

```
@Repository
public class JdbcExampleDao implements ExampleDao {

    private JdbcTemplate jdbcTemplate;

    @Inject
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

}
```

Configuration de la data source

- La data source utilisée est un pool de connexion à la base de données
- Voici une configuration Spring typique :

```
<context:component-scan base-package="example.test" />

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close">

  <property name="driverClassName" value="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>

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

IPPON Enterprise Java Delivery

Gestion de la Data Source

- La Data Source est gérée par Spring
- C'est Spring JDBC qui va gérer le fait d'ouvrir une connexion JDBC et de la refermer
  - Il n'y a plus à coder ces parties techniques
  - Cela élimine les risques d'erreurs : en cas d'*Exception*, c'est Spring qui se charge de correctement fermer la connexion
- Cela reprend ce que nous avons vu au début
  - Spring se charge de la plomberie : ouvrir/fermer les connexions, gérer les *Exceptions*
  - Au développeur de coder le métier : la requête SQL

Utilisation de *JdbcTemplate*

- *JdbcTemplate* permet de faire des requêtes JDBC en une seule ligne
  - Beaucoup plus pratique que la méthode «manuelle»


```
int nbAccounts =
    jdbcTemplate.queryForInt("select count(*) from account");

int nbGoodCustomers = jdbcTemplate.queryForInt(
        "select count(*) from account where balance > ?", 10000);
```

Utilisation de *JdbcTemplate*

- *JdbcTemplate* gère également les create/update/delete
  - On peut donc faire un CRUD assez simplement

```
jdbcTemplate.update(
        "insert into account (user_id, creation_date, balance) "  +
        "values (?, ?, ?)",
        account.getUserId(),
        account.getCreationDate(),
        user.getBalance());

jdbcTemplate.update(
             "update account set balance=? where id=?" ,
             account.getBalance(),
             account.getId());
```

Utilisation d'un *RowMapper*

- Un *RowMapper* permet de mapper un objet Java avec une ligne retournée par une requête SQL
  - Cela permet de faire un mapping objet/relationnel très simple


```
jdbcTemplate.query(
    "select id, user_id, balance from account" ,
    new ParameterizedRowMapper<Account>() {
       public Account mapRow(ResultSet rs,  int i) throws SQLException {
          return new Account(rs.getLong("id"),
                             rs.getInt("user_id"),
                             rs. getInt("balance"));
       }
    }
);
```

Utilisation des *Callbacks*

- Le *RowMapper* que nous venons de voir est ce que l'on appelle un *Callback*
- Il s'agit d'une classe anonyme, implémentant une interface fournie par Spring
- Cela permet de recevoir dans notre code métier un appel provenant de Spring
  - Dans ce cas, Spring nous donne le *ResultSet* et le numéro de la ligne
- Ce mécanisme de *Templates* et de *Callbacks* est très largement utilisé dans les différentes abstractions fournies par Spring

IPPON Enterprise Java Delivery

Gestion des *Exceptions*

- Le *JdbcTemplate* ne lance pas de *SQLException*
  - Relancer ces *Exceptions* jusqu'à la couche de présentation ne sert à rien
  - Cela casse l'architecture logicielle : si une *SQLException* est remontée, on sait que dans les couches basses on utilise JDBC

- Spring JDBC enrobe les SQLException par des DataAccessException
  - Ce sont des exceptions de type Runtime (il n'y a pas à les catcher)
  - Elles sont communes à toutes les technologies d'accès aux données (=utilisé par Hibernate et JPA également)
  - Elles permettent toujours d'avoir accès à l'*Exception* d'origine (pas de perte de connaissance)

IPPON Enterprise Java Delivery

Conclusion sur Spring JDBC

- Abstraction relativement simple au-dessus de JDBC
  - Gestion automatique de la *DataSource*
  - Gestion automatique des *Exceptions*
  - Classes utilitaires simples, utilisant des *Templates* et des *Callbacks*
- Spring JDBC est pratique pour
  - Faire des CRUD simples
  - Avoir un accès «bas niveau» à la base de données
- Spring JDBC peut être utilisé conjointement à Hibernate/JPA, qui fournit une solution bien plus complète d'accès aux bases de données relationnelles

Les transactions

Introduction aux transactions

- Les transactions sont typiquement gérées par une base de données relationnelles
- Une transaction est normalement **ACID**
  - **A**tomique
  - **C**ohérente (**C**onsistant)
  - **I**solée
  - **D**urable
- En Java, nous pouvons les gérer via des APIs simples, par exemple en JDBC ou avec JTA

IPPON Enterprise Java Delivery

Utilité des transactions

- Il est primordial de gérer les transactions si on veut avoir des données de qualité
  - Les transactions permettent de traiter un ensemble d'opérations comme une seule opération
  - Pas de données incohérentes dans la base
- Les transactions permettent également d'avoir de meilleures performances
  - Il vaut mieux faire 10 requêtes dans une transaction que de faire 10 requêtes dans 10 transactions
  - C'est d'ailleurs une des raisons de leur utilisation dans les batchs

Exemple d'une transaction

- 3 requêtes : 1 lecture, 2 écritures
- 1 seule transaction (matérialisée par la flèche)
- Soit les 2 modifications sont appliquées, soit aucune n'est appliquée

R

```
SELECT *
FROM account
WHERE balance > 0;
```

W

```
INSERT INTO account
VALUES (1, 15);
```

W

```
UPDATE account
SET balance = 0
WHERE id = 1;
```

Les transactions en Java

- En JDBC

```
conn = dataSource.getConnection();
conn.setAutoCommit(false);
// requêtes SQL
conn.commit();
```

- Avec JTA (**J**ava **T**ransaction **A**PI)

```
UserTransaction utx = ctx.getUserTransaction();
// Démarrage de la Transaction
utx.begin();
// Exécution des requêtes
utx.commit();
```

Spring Transactions

- Spring propose une couche d'abstraction
  - Gère les transactions JDBC, Hibernate, JTA etc... de manière homogène
  - Permet de simplement configurer ses transactions : utilisation d' annotations ou d'XML, sans utilisation obligatoire de code
- Cela permet d'avoir une meilleure architecture
  - Les transactions sont déclarées dans la couche métier (service), et non dans la couche d'accès aux données (repository / DAO)
  - Les transactions ne dépendent pas d'une technologie particulière d' accès aux données (JDBC)

Utilisation simple avec Spring

- Configurer un gestionnaire de transaction

```
<bean id="transactionManager"
 class="org.springframework.jdbc.datasource.
DataSourceTransactionManager" >
  <property name="dataSource"  ref="dataSource"/>
</bean>
```

- Dire à Spring que l'on veut utiliser les annotations

```
<tx:annotation-driven/>
```

- Utiliser les annotations

```
@Transactional
public void uneMethodeMetier() {
   // Unité de travail atomique
}
```

Fonctionnement dans Spring

- Spring fournit un Aspect spécialisé
  - Le Point Cut est sur les méthodes annotées @*Transactional*
  - L'Advice est de type Around, et ajoute la gestion des transactions autour des méthodes annotées
- C'est le fonctionnement que nous avons vu dans le chapitre sur Spring AOP, avec la génération d'un proxy
  - Ne fonctionne que sur les Beans Spring
  - Ne fonctionne que sur les méthodes publiques
  - Ne fonctionne pas à l'intérieur d'un même Bean

Configuration d'un gestionnaire de transaction

- Le gestionnaire de transaction est une classe fournie par Spring
  - Il fait partie de l'infrastructure
  - Il est spécifique à la technologie utilisée
  - Hors JTA, il a besoin d'une Data Source pour être configuré
  - Par convention, il possède l'id «transactionManager»
- Si vous êtes dans un serveur d'applications (Websphere, Weblogic...), Spring peut retrouver automatiquement le gestionnaire de transactions de ce serveur (utilisant l'API JTA) :

```
<tx:jta-transaction-manager/>
```

Utilisation des annotations

- L'annotation *@Transactional* peut être mise sur une classe (toutes les méthodes publiques sont transactionnelles) ou sur une méthode
- Cette annotation a un certain nombre de paramètres : Isolation, Propagation, Timeout, readOnly…

```
@Service
@Transactional
public class TodoListsServiceImpl implements TodoListsService {

    @Transactional(readOnly = true)
    public TodoList findTodoList(String listId) {
        // Code métier, utilisant Hibernate par exemple
    }
}
```

Variante : utilisation du XML

- On peut également utiliser une configuration XML
  - On code alors un Point Cut spécifique : par exemple toutes les méthodes des classes d'un package spécifique
  - On privilégiera la configuration par annotations : elle est plus simple et plus lisible

```
<aop:config>
  <aop:pointcut id="serviceBeans"
      expression="execution(public * test.service.*(..))" />
  <aop:advisor pointcut-ref="serviceBeans" advice-ref="txAdvice"/>
</aop:config>

<tx:advice id="txAdvice">
  <tx:attributes>
    <tx:method name="find*" read-only="true"/>
    <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
```

Le *TransactionTemplate*

- Si la configuration par annotations ou XML n'est pas suffisante, Spring propose une API
  - Elle utilise le système des *Templates* et des *Callbacks* que nous avons déjà vus pour Spring JDBC

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {
    protected void doInTransactionWithoutResult(TransactionStatus status) {
        try {
            insertionEnBase();
            miseAJourDeLaBase();
        } catch (ExceptionMetier ex) {
            status.setRollbackOnly();
        }
    }
});
```

Transactions et isolation

- Dans la pratique, les transactions ne sont en réalité pas toujours bien isolées
- Il y a **quatre niveaux** d'isolation, du plus sécurisé au moins sécurisé :
  - SERIALIZABLE
  - REPEATABLE READS
  - READ COMMITTED
  - READ UNCOMMITTED

- Plus le niveau d'isolation est élevé, plus la base doit **locker** des ressources, et moins les performances sont bonnes
- Le niveau d'isolation par défaut est configurable dans la base de données, sous Oracle il est à «READ COMMITTED»

Exemple 1 : non repeatable read

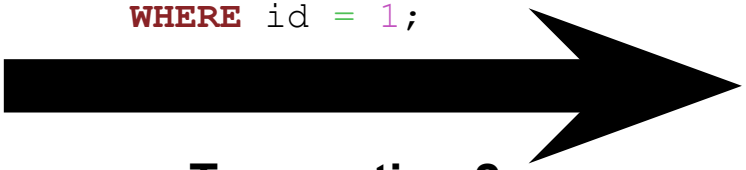
- Ce problème arrive lorsque l'on est en READ COMMITTED ou READ UNCOMMITTED

```
SELECT *
FROM account
WHERE id = 1;
```

```
SELECT *
FROM account
WHERE id = 1;
```

Transaction 1

```
UPDATE account
SET balance = 0
WHERE id = 1;
```



Transaction 2

La transaction n°1 va lire deux données différentes !

Exemple 2 : phantom read

```
SELECT *
FROM account
WHERE balance
BETWEEN 10 AND 30;
```

```
SELECT *
FROM account
WHERE balance
BETWEEN 10 AND 30;
```

Transaction 1

- Ce problème arrive lorsque l'on est en REPEATABLE READ, READ COMMITTED ou READ UNCOMMITTED

```
INSERT INTO account
VALUES (1234, 15);
```

Transaction 2

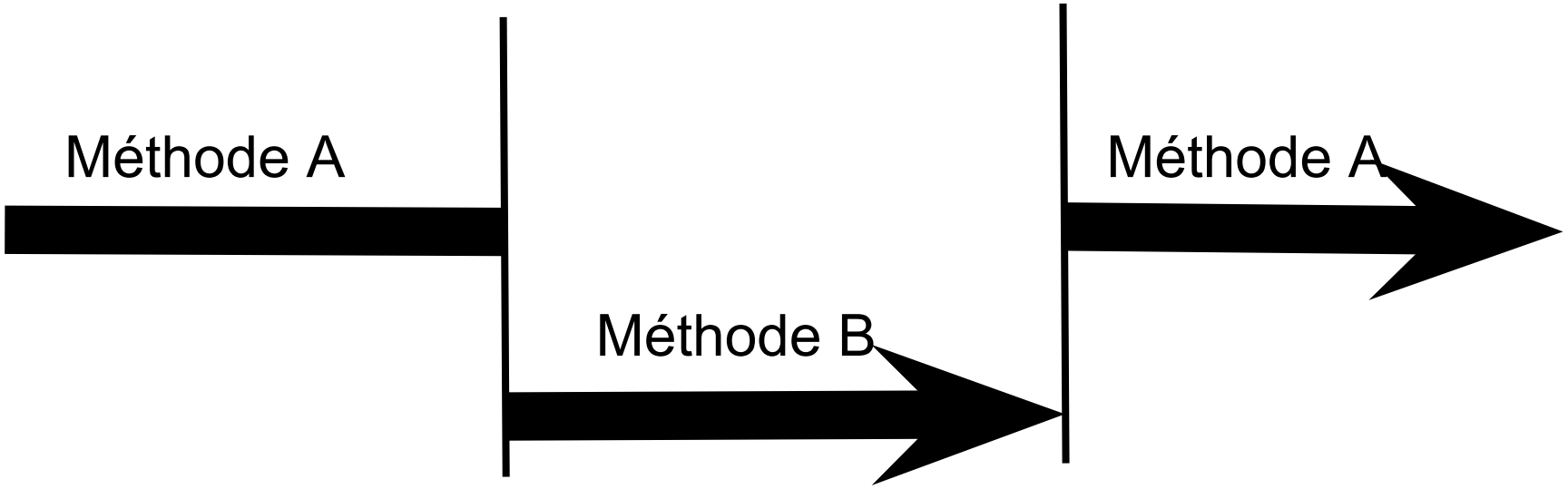
La transaction n°1 n'aura pas deux fois la même liste !

Les transactions read-only

- On peut marquer une transaction comme étant «read-only»
  - On indique qu'elle ne va pas modifier de données en base
  - En lecture, il est toujours important d'utiliser des transactions, ne serait-ce que pour les performances
- Cet attribut est important
  - Hibernate va le comprendre, il ne va alors plus vérifier s'il doit impacter des modifications sur les objets en base : meilleures performances
  - Certains drivers JDBC vont le comprendre (Oracle ne vous autorisera plus qu'à faire des «SELECT») : meilleure qualité

La propagation des transactions 1

- Que se passe-t-il quand une méthode transactionnelle en appelle une autre ?



La propagation des transactions 2

- On peut configurer si l'on veut deux transactions différentes ou une seule transaction englobant les deux méthodes
  - **REQUIRED** : S'il y a déjà une transaction, l'utiliser. Sinon, en créer une nouvelle. C'est le mode par défaut.
  - **REQUIRES_NEW** : Crée toujours une nouvelle transaction. S'il y en a déjà une, la suspend.
  - **NESTED** : Peut faire une transaction imbriquée, si cela est supporté par le gestionnaire de transaction.
  - **MANDATORY** : Une transaction doit déjà exister. Sinon, lance une *Exception*.
  - **SUPPORTS** : Si une transaction existe déjà, l'utilise. Sinon, n'utilise pas de transaction.
  - **NOT_SUPPORTED** : N'utilise pas de transaction. Si une transaction existe déjà, la suspend.
  - **NEVER** : N'utilise pas de transaction. Si une transaction existe déjà, lance une *Exception*.

Exemple

- On peut configurer la transaction via l'annotation *@Transactional* : si elle est «read-only», spécifier un timeout ou un mode de propagation particulier

```
@Transactional(readOnly = true, timeout = 30,
               propagation = Propagation. REQUIRES_NEW)
public void maMethodeMetier() {
    //
}
```

IPPON Enterprise Java Delivery

Les transactions XA

- Les transactions XA permettent d'avoir une seule transaction en utilisant des ressources différentes
  - Deux bases de données
  - Une base de données et un serveur JMS
- Pour fonctionner, il faut que ces ressources et le gestionnaire de transaction supportent les transactions XA
  - WebSphere, Weblogic proposent des gestionnaires de transaction XA
  - Avec Spring, vous pouvez configurer un gestionnaire de transaction XA externe : Atomikos, Bitronix
- Avec Spring, c'est juste une configuration différente de l' infrastructure
  - Aucun changement dans le code !
  - Nous vous déconseillons d'utiliser cette technologie
  - On peut généralement obtenir le même résultat de manière plus simple (sans faire de transaction distribuée)
  - De par son fonctionnement, elle est peu performante

IPPON Enterprise Java Delivery

Pattern «Open Transaction in View»

- C'est un pattern très répandu dans les applications Web, aussi appelé «Open Session In View» (la session étant une session Hibernate)
  - Spring propose un listener de Servlet qui implémente ce pattern
- Ce pattern est très pratique
  - Permet d'avoir une transaction ouverte tout le temps, y compris dans les pages Web
  - Règle les problèmes de «lazy loading» avec Hibernate
- Nous le déconseillons parfois pour des raisons de performances
  - Il a tendance à laisser les transactions ouvertes trop longtemps
  - On arrive à terme à une requête HTTP == une transaction, et donc à une connexion en base de données. Difficile alors de monter en charge !

IPPON Enterprise Java Delivery

Conclusion sur les transactions

- L'utilisation des transactions est **essentielle**, elle vous garantie la qualité des données et une bonne performance
- La configuration des transactions avec Spring est très simple
  - Utilisation des annotations
  - Une configuration avancée reste un simple paramétrage
  - En termes d'architecture, cela permet de gérer les transactions au niveau de la couche métier (service), et plus dans les couches basses (repository/DAO)

Exercice 4
les transactions

Les tests avec Spring

Introduction sur les tests

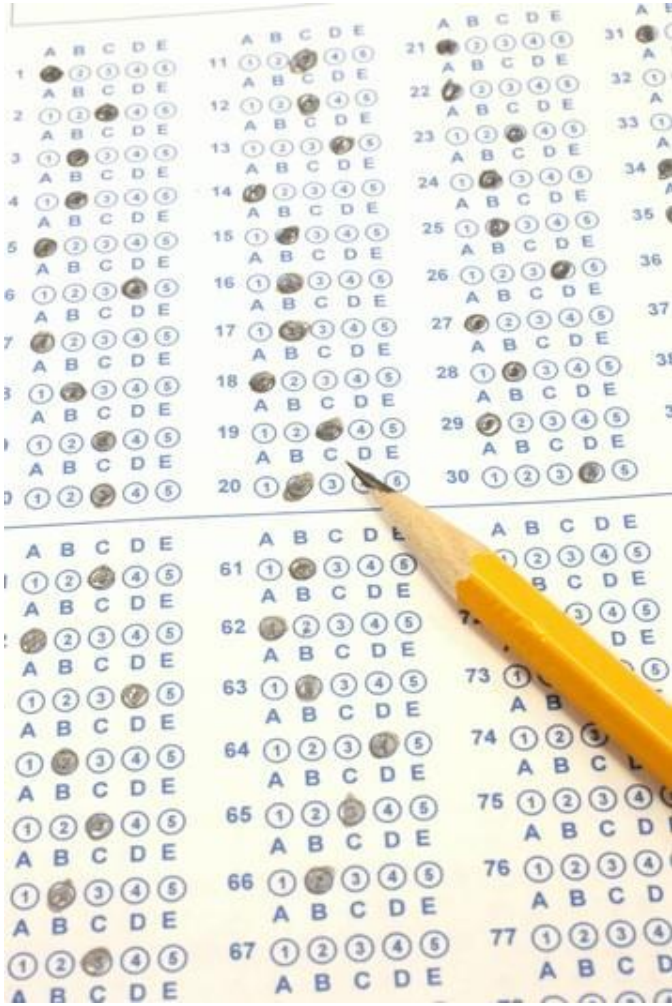
- Les tests automatisés permettent d'améliorer la qualité du code
  - Garantissent le bon respect des règles métier
  - Facilitent le refactoring
- Ils permettent également de coder plus efficacement
  - L'automatisation permet un feedback rapide
    - Permet de corriger un bug juste après l'avoir causé
    - Evite de polluer l'application et d'impacter les autres développeurs
- Spring permet d'améliorer la conception des tests unitaires et propose un excellent support des tests d'intégration

Tests unitaires contre tests d'intégration

- Il y a deux types de tests
  - Test d'un composant unique (métier ou technique), en isolation du reste des autres composants : ce sont les tests unitaires
    - Cela exclut l'utilisation de Spring
  - Test d'un ensemble de composants dans un environnement comparable à la production : ce sont les tests d'intégration
    - Cela inclut l'utilisation de Spring, sans doute avec une configuration d'infrastructure spécifique
- Les tests unitaires et les tests d'intégration ne sont pas exclusifs : il faut utiliser les deux conjointement pour bien tester

Objectifs des tests

- Les tests doivent couvrir un maximum de lignes de code de l'application
  - Il ne s'agit pas de tester toutes les lignes de code, mais de bien tester les lignes de code importantes
  - Si une méthode utilise des branchements conditionnels, il faut valider tous les cas possibles
- Ces tests doivent être rapides et automatisés
  - Un jeu de test qui dure longtemps ne sera jamais exécuté par les développeurs : son utilité est donc nulle



L'intégration continue

- L'utilisation d'un serveur d'intégration continue est **essentielle** pour exécuter ces tests
  - Aujourd'hui il n'y a plus de question à se poser : utilisez Jenkins ! http://jenkins-ci.org/
- Ce serveur vous alerte si les tests ne fonctionnent pas
  - Vous évite d'updater votre projet si un de vos collègues a commité des bugs
- Bonne pratique : **toujours faire passer les tests unitaires avant de commiter**
  - Renforce l'argument que ces tests doivent être rapides
  - Certains IDE proposent cette fonctionnalité (ils refusent de commiter s'il y a des erreurs)
  - Certaines équipes vont plus loin : elles commitent dans un repository intermédiaire, et leur code n'est ensuite poussé dans le repository principal que si les tests passent

Tests unitaires

- Les tests unitaires permettent de tester une méthode en isolation du reste de l'application
  - Cela exclut Spring
- Cependant, grâce à Spring, vous avez des Beans faciles à tester
  - L'injection de dépendance fait que les Beans Spring sont faciles à tester unitairement : il suffit de remplacer ces dépendances par des Stubs ou des Mocks
  - L'AOP permet de n'avoir qu'une seule fonctionnalité métier par méthode
    - Pas de code technique gérant les transactions ou la sécurité mélangé au code métier

Exemple de test «simple»

- JUnit permet de lancer facilement toutes les méthodes marquées *@Test*
  - Maven les lance ensuite dans sa phase «test»
  - Astuce : les imports statiques permettent d'utiliser directement les méthodes JUnit dans la classe à tester

```
@Test
public void testEquals() {
    Todo todo1 = new Todo();
    todo1.setTodoId("001");
    Todo todo2 = new Todo();
    todo2.setTodoId("001");
    assertEquals(todo1, todo2);

    Todo todo3 = new Todo();
    todo3.setTodoId("003");
    assertNotSame(todo1, todo3);
}
```

IPPON Enterprise Java Delivery

Des Stubs ou des Mocks ?

- En Français on confond les deux termes sous le nom de «bouchon»
- Ils sont nécessaires pour tester les dépendances, en particulier celles injectées par Spring
- Un **Stub** : une implémentation «vide» d'une dépendance
  - Exemple : pour un DAO, faire une implémentation qui n'accède pas en base de données mais renvoie toujours les mêmes valeurs
- Un **Mock** : une implémentation générée par une librairie spécialisée, qui la crée à la volée en fonction de l'interface à respecter

Astuce: comment injecter dans un champ ?

- Pour tester unitairement un Bean Spring ayant des dépendances
  - Il ne faut pas utiliser Spring (sinon ce ne serait plus un tes
  - Il faut donc injecter manuellement ses dépendances
- Cette injection est évidente si l'on utilise l'injection par Setter ou par Constructeur
- Pour l'injection par Champ, qui est de plus en plus populaire, Spring propose cet utilitaire :

```
ReflectionTestUtils.setField(todosService,
                             "todoListsService",
                             todoListsService);
```

  - Il injecte une variable «todoListsService» dans le champ nommé «todoListsService» du Bean «todosService»

Exemple de Stub

- Le Stub implémente la même interface que la dépendance injectée
  - Le Stub peut être une classe anonyme (exemple ci-dessous), pour éviter de créer trop de fichiers
  - Cela peut être également une vraie classe, afin de pouvoir le réutiliser sur plusieurs tests

```
AccountService accountService =  new AccountService();
accountService.setUserService(new UserService() {
    public User getCurrentUser() {
        User user = new User();
        user.setLogin("test");
        user.setFirstName("John");
        return user;
    }
});
assertEquals(accountService. getCurrentUser().getFirstName(),
"John");
```

Utilisation de EasyMock

- EasyMock permet de générer automatiquement une implémentation à partir d'une interface
- Cette implémentation fonctionne ensuite comme un magnétoscope :
  - On liste les méthodes qui vont être appelées : ce sont les méthodes «expect()»
  - On dit ensuite à EasyMock que l'on va jouer ce scénario : c'est la méthode «replay()»
  - En fin de test, on demande à EasyMock de valider que le scénario s'est déroulé comme prévu : c'est la méthode «verify()»

IPPON Enterprise Java Delivery

Exemple de test avec EasyMock

```
@Test
public void testCreateTodo() {
    TodosService todosService =EasyMock.createMock(TodosService.class);
    TodoListsService todoListsService =EasyMock.createMock(TodoListsService.class);
    EntityManager em =EasyMock.createMock(EntityManager.class);
    ReflectionTestUtils.setField(todosService, "em", em);
    ReflectionTestUtils.setField(todosService, "todoListsService', todoListsService);

    TodoList todoList =new TodoList();
    todoList.setListId("001");
    todoList.setName("Test Todo List');
    Todo todo =new Todo();
    todo.setTodoId("0001");
    todo.setDescription("Test description');

    EasyMock.expect(todoListsService.findTodoList("001")).andReturn(todoList);
    todoListsService.updateTodoList(todoList);
    em.persist(todo);

    EasyMock.replay(em);
    EasyMock.replay(todoListsService);

    todosService.createTodo("001", todo);

    assertNotNull(todo.getCreationDate());
    assertEquals(todoList, todo.getTodoList());
    assertTrue(todoList.getTodos().contains(todo));

    EasyMock.verify(em);
    EasyMock.verify(todoListsService);
}
```

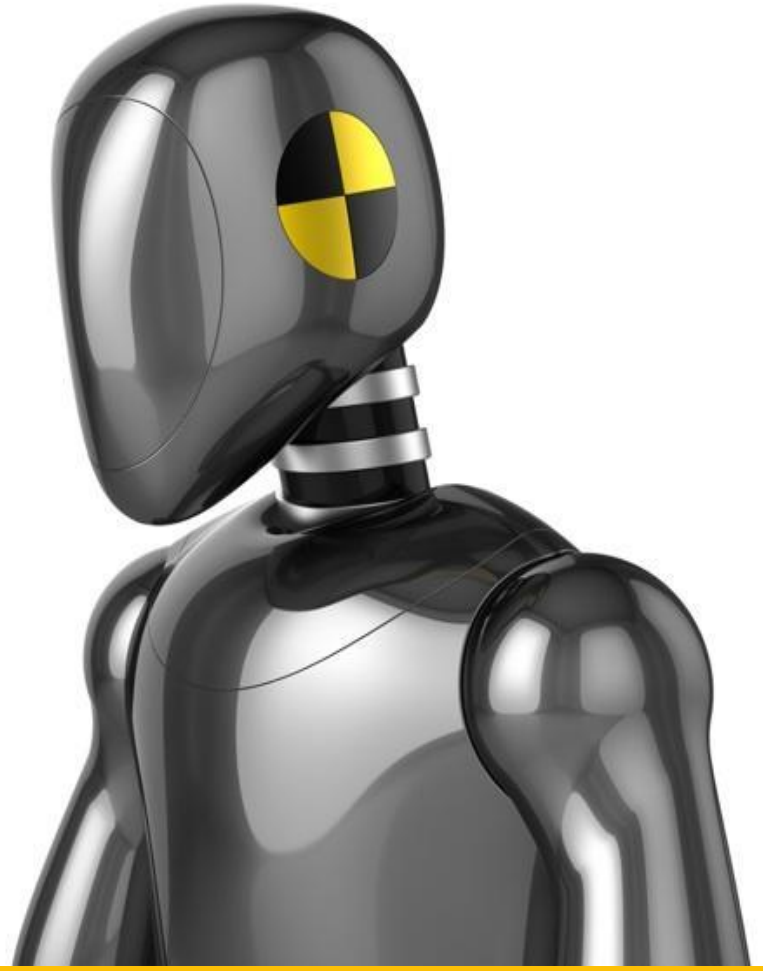
IPPON
Enterprise Java Delivery

Pourquoi utiliser des Mocks ?

- Les Mocks sont aujourd'hui très populaires
  - Ils évitent d'avoir à coder des Stubs
  - Ils sont plus rapides à coder et à maintenir que des Stubs : on ne code que les méthodes nécessaires au test
  - Il est plus simple de les faire changer de comportement
  - Ils permettent des tests plus puissants : on vérifie que les Mocks ont bien été appelés comme on l'a défini
- Ils restent plus complexes à coder
- Il faut utiliser une librairie spécialisée
  - Mockito : http://code.google.com/p/mockito/
  - EasyMock : http://easymock.org/

Tests d'intégration

- Les tests d'intégration incluent Spring
  - Normalement avec un application context réduit : uniquement un ensemble de classes que l'on veut tester
  - Avec une configuration d'infrastructure spécifique : une base de données en mémoire ou une instance spécifique de la base de données cible
- Spring propose une intégration à JUnit qui simplifie grandement ce type de configuration



Support des tests d'intégration dans Spring

- *SpringJUnit4ClassRunner* permet d'intégrer Spring dans un test JUnit
- L'annotation *@ContextConfiguration* permet alors de localiser la configuration de Spring et de lancer l'Application Context
- On peut ainsi tester son application Spring avec JUnit, sans serveur d'applications
  - Toutes les fonctionnalités gérées par Spring fonctionnent de manière identique : connexion à la base de données, transactions...
  - On peut activer le debugging et le profiling, qui peuvent être faits directement dans l'IDE
  - C'est évidemment beaucoup plus rapide à exécuter que de déployer l'application sur un serveur d'applications

IPPON Enterprise Java Delivery

Exemple de test d'intégration

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath*:/META-INF/spring/application-context-test.xml"})
public class IntegrationTest {

   @Inject
   private UserService userService;

   @Test
   public void createUser() {
      try {
         userService.findUser("test_user");
         fail("User already exists in the database.");
      } catch (ObjectRetrievalFailureException orfe) {
         // User should not already exist in the database.
      }

      User user = new User();
      user.setLogin("test_user");
      user.setFirstName("First name");
      userService.createUser(user);
      User userFoundInDatabase = userServicefindUser("test_user");
      assertEquals("First name", userFoundInDatabase.getFirstName());
   }
}
```

Astuce 1 : lancer l'application context une seule fois

- En réalité, cette astuce est déjà utilisée dans les exemples précédents
- Spring ne lance qu'un seul application context par classe
  - Toutes les méthodes de test d'une classe donnée utilisent la même instance
- Cela permet d'accélérer les tests : sinon on lancerait beaucoup plus d'application contexts
- Cela ne doit pas avoir d'autre impact
  - En effet, vos Beans sont censés être thread safe

Astuce 2 : rollback des transactions dans les tests d'intégration

- Par défaut, toute méthode de test annotée *@Transactional* va être rollbackée à la fin du test
  - Inutile de nettoyer la base de données après un test
  - Le test sera également plus performant
  - Le rollback n'est possible qu'à la condition que personne ne commite explicitement pendant le test !

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath*:/META-INF/spring/application-context-test.xml"})
public class IntegrationTest {

   @Inject
   private UserService userService;

   @Test
   @Transactional
   public void createUser() {
      // Même code que précédemment
   }
}
```

IPPON Enterprise Java Delivery

Conclusion sur les tests

- Les tests sont essentiels pour réaliser des applications de qualité
- Spring nous permet d'avoir une architecture qui facilite grandement l'écriture de tests unitaires
- Spring fournit un excellent support des tests d'intégration

Exercice 5
les tests

«APIs externes»

- Hibernate et JPA
- Bean Validation
- Spring JMS
- Spring JMX

Hibernate & JPA

IPPON Enterprise Java Delivery

Introduction à l'ORM

- ORM == **O**bject-**R**elational **M**apping
- Cette technologie permet de mapper automatiquement des objets sur des tables
  - Cela facilite le développement
  - Cela donne une architecture logicielle de meilleure qualité
- Il existe de nombreuses solutions d'ORM
  - JPA n'est qu'une API
  - EclipseLink en est l'implémentation officielle
  - Hibernate en est l'implémentation (de loin) la plus populaire

Qu'apporte une solution d'ORM ?

- Une solution d'ORM permet de se concentrer sur des objets Java (souvent appelés «objets de domaine», pour le domaine métier)
  - Le code SQL pour créer/sélectionner/mettre à jour/effacer ces données est automatiquement généré
  - Il n'y a plus de dépendance avec une base de données spécifique (différences dans le langage SQL)
  - Elle fournit généralement des mécanismes avancés de cache et de chargement des données qui font qu'elle est au final plus performante que du SQL codé «à la main»

Les problèmes soulevés par l'ORM

- Une technologie de mapping est nécessaire car il y a des différences fondamentales entre une table et un objet
  - Il n'y a pas d'héritage ou de polymorphisme en base de données
  - La gestion des associations est différente (one-to-many et many-to-many)
  - De nombreux types de données sont différents (par exemple il y a de nombreuses façons de stocker une String en base de données)
  - Les contraintes de validation ne sont pas gérées de la même manière...
- Une solution d'ORM a pour but d'alléger ou de simplifier ces problèmes

JPA, EclipseLink, Hibernate

- **1994** : TopLink, premier ORM au monde, en SmallTalk
- **1996** : TopLink propose une version Java
- **1998** : EJB 1.0
- **2000** : EJB 2.0
- **2001** : lancement d'Hibernate, afin de proposer une alternative aux EJB 2
- **2003** : Gavin King, créateur d'Hibernate, rejoint JBoss
- **2006** : EJB 3.0 et JPA 1.0
- **2007** : TopLink devient EclipseLink
- **2009** : JPA 2.0
- **2013** : JPA 2.1

Quelle implémentation choisir ?

- Hibernate est de très loin l'implémentation la plus répandue
  - Hibernate a largement fait ses preuves en termes de qualité et de performance
  - Il faut avoir une très bonne raison pour ne pas prendre Hibernate
- Privilégier l'API JPA
  - C'est ce que recommande également l'équipe Hibernate !
  - Lorsque JPA n'est pas suffisant, on peut toujours compléter avec l'API spécifique Hibernate

Mapping d'une entité simple (1/2)

```java
@Entity
@Table(name = "t_todo")
public class Todo implements Serializable {

    @Id
    @Column(name = "id")
    private String todoId;

    @Column(name = "creation_date")
    private Date creationDate;

    private String description;

    private int priority;

    // getters et setters
}
```

IPPON Enterprise Java Delivery

Mapping d'une entité simple (2/2)

- Ce mapping utilise uniquement des annotations JPA
- Les annotations sont :
  - Au niveau de la classe, pour la mapper sur une table donnée
  - Au niveau des champs, qui correspondent aux colonnes de la table
- La configuration est implicite
  - Les champs qui n'ont pas d'annotations sont par défaut mappés sur des colonnes ayant le même nom qu'eux
  - Si le nom de la colonne est différent, on peut le paramétrer avec l'annotation *@Column*
  - Si le champ ne doit pas être mappé, il faut le marquer avec *@Transient*
- Dans le cas le plus simple, il suffit d'annoter la classe avec *@Entity* et définir le champ contenant la clef primaire avec *@Id*

Utilisation de ce mapping (1/2)

```
@Service
@Transactional
public class TodosServiceImpl implements TodosService {

    @PersistenceContext
    private EntityManager em;

    public void createTodo(Todo todo) {
        Date now = Calendar.getInstance().getTime();
        todo.setCreationDate(now);
        em.persist(todo);
    }

    public Todo findTodo(String todoId) {
        return em.find(Todo.class, todoId);
    }

    public Todo updateTodo(String todoId, String description) {
        Todo todo = em.find(Todo.class, todoId);
        todo.setDescription(description);
    }

    public void deleteTodo(String todoId) {
        Todo todo = em.find(Todo.class, todoId);
        em.remove(todo);
    }
}
```

Utilisation de ce mapping (2/2)

- Le *PersistenceContext* est la classe principale, qui permet de requêter, créer ou supprimer des objets en base de données
- La persistance est transparente : dès qu'un objet est géré par le *PersistenceContext*, ses modifications seront automatiquement répercutées en base de données à la fin de la transaction
- Pour mettre à jour un objet, il suffit donc d'appeler ses setters, et d'attendre la fin de la transaction

Le cache de premier niveau

- Hibernate stocke en fait tout objet lu depuis la base dans un cache de premier niveau
  - Ce cache correspond à la transaction en cours
- Si vous faites deux fois un «find» sur la même instance, seul le premier appel lancera une requête SQL
  - Le deuxième appel le lira dans le cache de premier niveau
  - Cela limite considérablement les risques de «NON REPEATABLE READS» que nous avons abordés dans le chapitre sur les transactions
- A la fin de la transaction, Hibernate va «flusher» ce cache
  - Il va calculer toutes les modifications de l'objet qu'il doit répercuter en base
  - Il va alors exécuter toutes les requêtes à la suite

Mapping many-to-one et one-to-many

- Nous avons vu un premier exemple simple
  - Il correspond à peu près à ce que nous avons fait avec Spring JDBC
- JPA permet également de gérer les relations entre les objets
  - Un objet qui contient une collection d'autres objets
  - Cette relation est mappée comme une relation one-to-many en base de données, utilisant une foreign key

Exemple de mapping many-to-one

```
@Entity
public class Todo implements Serializable {

    @Id
    private String todoId;

    @ManyToOne
    private TodoList todoList;

    private Date creationDate;

    private String description;

    // getters et setters

}
```

Utilisation du mapping many-to-one

- Ce mapping se configure également avec une annotation sur un champ
- Nous avons deux objets mappés sur deux tables, et une annotation qui correspond à la foreign key joignant ces deux tables
- Il suffit de modifier l'objet lié (todoList dans l'exemple précédent) pour modifier la foreign key

```
public void createTodo(String listId, Todo todo) {
    Date now = Calendar.getInstance().getTime();
    todo.setCreationDate(now);
    TodoList todoList = todoListsService.findTodoList(listId);
    todo.setTodoList(todoList);
    em.persist(todo);
}
```

IPPON Enterprise Java Delivery

Les associations bi-directionnelles

- En fonction du métier, certaines associations sont en fait bi-directionnelles
  - L'objet parent a accès à une collection d'enfants
  - Chaque enfant a accès à l'objet parent
- Il faut prévenir JPA que le mapping est bi-directionnel
- Nous travaillons toujours en Java : il faut donc traiter cela également côté Java
- Faire tous les mappings bi-directionnels est une mauvaise pratique
  - Cela a un impact sur les performances
  - Cela signifie également que vous avez mal modélisé votre métier

Exemple de mapping bi-directionnel


```
@Entity
public class Todo implements Serializable {

    @Id
    private String todoId;

    @ManyToOne
    private TodoList todoList;
}
```

```
@Entity
public class TodoList implements Serializable {

    @Id
    private String listId;

    @OneToMany(mappedBy = "todoList")
    private Set<Todo> todos = new
HashSet<Todo>();
}
```

Utilisation du mapping bi-directionnel

```
public void createTodo(String listId, Todo todo) {
    TodoList todoList = todoListsService. findTodoList
(listId);
    todo.setTodoList(todoList);
    em.persist(todo);
    todoList.getTodos().add(todo);
}

public void deleteTodo(Todo todo) {
    TodoList todoList = todo. getTodoList();
    Set<Todo> todos = todoList. getTodos();
    todos.remove(todo);
    em.remove(todo);
}
```

Les méthodes equals() et hashCode()

- L'utilisation de Set renforce la nécessité de bien implémenter les méthodes *equals()* et *hashCode()*
- En effet, un Set utilise le hashCode pour savoir si un objet est déjà présent ou non dans la Collection
  - Si le hashCode est mauvais, on peut remplacer par erreur un autre objet de la Collection
  - Si le hashCode est inexistant, Java va alors utiliser l'adresse mémoire : il sera alors possible d'insérer deux instances du même objet, ce qui sera très certainement problématique en base de données (violation de la Clef Primaire)

Mapping many-to-many

- JPA permet également de modéliser des relations many-to-many
  - Exemple : un utilisateur possède plusieurs listes, une liste peut être possédée par plusieurs utilisateurs
  - C'est un cas métier relativement classique lorsqu'on code en Java
  - Ce type de relation n'existe pas en base de données : il impose d' utiliser une table de jointure
- La table de jointure est automatiquement gérée par JPA
  - Elle possède deux foreign keys : une vers chaque table contenant les entités

Exemple de mapping many-to-many

```java
@Entity
public class User implements Serializable {

    @Id
    private String login;

    @ManyToMany
    private Set<TodoList> todoLists = new HashSet<TodoList>();
}
```

```java
@Entity
public class TodoList implements Serializable {

    @Id
    private String listId;

    @ManyToMany(mappedBy = "todoLists")
    private Set<User> users = new HashSet<User>();

}
```

Utilisation du mapping many-to-many

```
public void createTodoList(TodoList todoList) {
    User user = userService.getCurrentUser();
    todoList.getUsers().add(user);
    em.persist(todoList);
    user.getTodoLists().add(todoList);
}

public void deleteTodoList(String listId) {
    TodoList todoList = em.find(TodoList.class, listId);
    for (User user : todoList.getUsers()) {
        user.getTodoLists().remove(todoList);
    }
    em.remove(todoList);
}
```

Programmation défensive (1/2)

- En Java, quand nous avons une relation bi-directionnelle, nous devons traiter les deux côtés de la relation
- Dans les exemples précédents, c'est la couche service qui gère cela
  - Mais un développeur peut la contourner, en utilisant directement les objets de domaine
  - Il va faire un bug involontairement : l'API lui permet de le faire, il n'a donc pas à se méfier
  - Il est donc recommandé de gérer les bi-directions au niveau des entités

Programmation défensive (2/2)

```java
@Entity
public class User implements Serializable {

// ...

protected void setTodoLists(Set todoLists) {
    this.todoLists = todoLists;
}

public void addTodoList(TodoList todoList) {
    this.getTodoLists().add(todoList);
    todoList.getUsers().add(this);
}

public void removeTodoList(TodoList todoList) {
     this.getTodoLists().remove(todoList);
     todoList.getUsers().remove(this);
}
```

JPQL

- Java Persistence Query Language (JPQL) est un langage de requêtage spécifique à JPA, qui est indépendant de la base de données
  - Il ressemble à du SQL
  - Mais il s'exécute sur les objets Java (et non les tables), et a accès à leurs propriétés

```
SELECT user FROM User user where user.login LIKE :login
```

```
SELECT COUNT(user) FROM User user
```

API Criteria

- L'API Criteria permet également de requêter la base via JPA
  - Elle évite de faire de la manipulation de chaînes de caractères pour avoir la bonne requête
  - Elle est très adaptée aux écrans de recherche, où l'on peut sélectionner de nombreux critères différents

```
CriteriaBuilder qb = em.getCriteriaBuilder();
CriteriaQuery<Todo> query = qb.createQuery(Todo.class);
Root from = query.from(Todo.class);
Predicate condition = qb.gt(from.get("priority"), 20);
query.where(condition);
TypedQuery<Todo> typedQuery = em.createQuery(query);
List<Todo> todos = typedQuery.getResultList();
```

Configuration avec Spring (1/3)

- Il faut configurer un *EntityManagerFactory*, qui sera capable de fournir les *EntityManager* utilisés précédemment

```
<bean id="entityManagerFactory"
          class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

  <property name="dataSource" ref="dataSource"/>
  <property name="jpaVendorAdapter">
    <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
      <property name="database" value="MYSQL"/>
      <property name="databasePlatform" value="org.hibernate.dialect.MySQLDialect"/>
    </bean>
  </property>
  <property name="persistenceXmlLocation" value="classpath:META-INF/persistence.xml"/>
</bean>
```

Configuration avec Spring (2/3)

- La configuration Spring référence un fichier persistence.xml
  - C'est le fichier standard de configuration de JPA
  - Depuis Spring 3.1, ce fichier est optionnel

```
<persistence xmlns="...">
  <persistence-unit name="default"
                    transaction-type="RESOURCE_LOCAL">
    <class>tudu.domain.Todo</class>
    <class>tudu.domain.TodoList</class>
    <class>tudu.domain.User</class>
    <properties>
      <property name="hibernate.cache.region.factory_class"
        value="net.sf.ehcache.hibernate.SingletonEhCacheRegionFactory"/>
    </properties>
  </persistence-unit>
</persistence>
```

Configuration avec Spring (3/3)

- Pour que Spring puisse injecter un *EntityManager*, il faut lui ajouter le BeanPostProcessor suivant :

```
<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"  />
```

IPPON Enterprise Java Delivery

Gestion des Exceptions (1)

- De même que pour Spring JDBC, on peut ajouter un *BeanPostProcessor* qui va gérer les *Exceptions*
  - Il va également transformer les *Exceptions* JPA et Hibernate en *DataAccessException* de Spring
  - Il permet donc d'avoir la même hiérarchie d'*Exceptions*, quelle que soit la technologie d'accès aux données utilisée : JDBC, Hibernate ou JPA
  - Au niveau de la couche «Service», l'implémentation de la couche «Repository» n'est donc pas exposée

Gestion des Exceptions (2)

- Pour configurer cette gestion des *Exceptions*, il faut utiliser un Aspect fourni par Spring
  - Cet Aspect catche les *Exceptions* lancées par tous les Beans annotés *@Repository*
  - La configuration de cet Aspect passe par l'utilisation d'un *BeanPostProcessor*
    - Rappel : les *BeanPostProcessor* permettent de modifier les instances des Beans Spring, après leur instanciation

```
@Repository
public class ExampleDaoImpl implements ExampleDao {
    // méthodes
}
```

```
<!-- Aspect à ajouter dans la configuration des DAOs -->
<bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />
```

HibernateTemplate et JpaTemplate

- Il existe des classes *HibernateTemplate* et *JpaTemplate* pour faciliter l'utilisation d'Hibernate et de JPA
  - Exactement le même mécanisme que pour Spring JDBC et que pour les transactions
- Cependant ces classes ne sont plus utiles avec les versions récentes de Spring
  - Spring utilise les APIs internes d'Hibernate pour gérer la session Hibernate et les transactions
  - Spring utilise des *BeanPostProcessor* spécifiques pour injecter les *EntityManager* et gérer les *Exceptions*
- Il n'est donc pas conseillé d'utiliser ces Templates
  - Mieux vaut coder des classes JPA «normales», sans dépendance sur Spring

Exemple de Bean Spring «Repository»

```java
@Repository
public class UserServiceImpl implements UserService {

    @PersistenceContext
    private EntityManager em;

    public User findUser(String login) {
        return em.find(User.class, login);
    }
}
```

Pourquoi la couche «Repository» est-elle superflue ?

- Nous avons vu au chapitre 4 (Architecture d'applications Spring) que cette couche «Repository» devenait superflue
- Effectivement, cette couche ne sert plus à rien car c'est Hibernate qui gère l'ensemble de la persistance
  - Cette couche est régulièrement contournée avec Hibernate : ajouter une entité dans une collection peut revenir à faire un update en base de données, sans passer par la couche Repository

Gestion des transactions

- Avec JPA, comme en Spring JDBC, la gestion des transactions est essentielle
  - C'est d'autant plus important qu'Hibernate flushe ses données à la fin de la transaction
  - JPA est généralement utilisé dans un serveur d'applications, et donc avec un gestionnaire de transactions JTA, mais il a également une implémentation dédiée si nécessaire :

```
<bean id="transactionManager"
    class="org.springframework.orm.jpa.JpaTransactionManager" >
  <property name="entityManagerFactory"  ref="entityManagerFactory"
/>
</bean>
```

Utilisation avec Spring JDBC

- Avec un gestionnaire de transactions JTA, on peut faire des requêtes JDBC et JPA dans la même transaction
  - Il suffit d'annoter sa couche service avec *@Transactional*, et toutes les requêtes sont dans la même transaction
- **Attention** avec le cache de 1er niveau : en cas de modification d'une entité JPA, elle ne sera visible en base de données qu'après avoir été «flushée»

  - Une requête JDBC exécutée après du code JPA ne verra donc pas les données modifiées
  - Pour résoudre ce problème, il faut «flusher» manuellement la session JPA, ce qui exécutera alors immédiatement les requêtes, sans pour autant commiter la transaction

Le lazy-loading

- Le lazy-loading permet de ne charger des entités JPA que lorsqu'elles sont demandées
  - C'est le comportement par défaut des collections
  - On peut aussi le paramétrer sur des champs
- Cela évite, lorsque l'on requête une entité, de charger toutes les entités qui lui sont liées
  - En règle générale, c'est donc une excellente option pour la performance
- Cependant, cela peut multiplier les requêtes
  - Il ne faut pas faire de lazy-loading sur des associations qui sont toujours nécessaires lorsque l'on charge une entité
- Cela va également exécuter des requêtes SQL dans d'autres couches de l'application
  - Si on demande cette association dans une JSP, la requête sera faite à ce niveau

Les stratégies de fetch

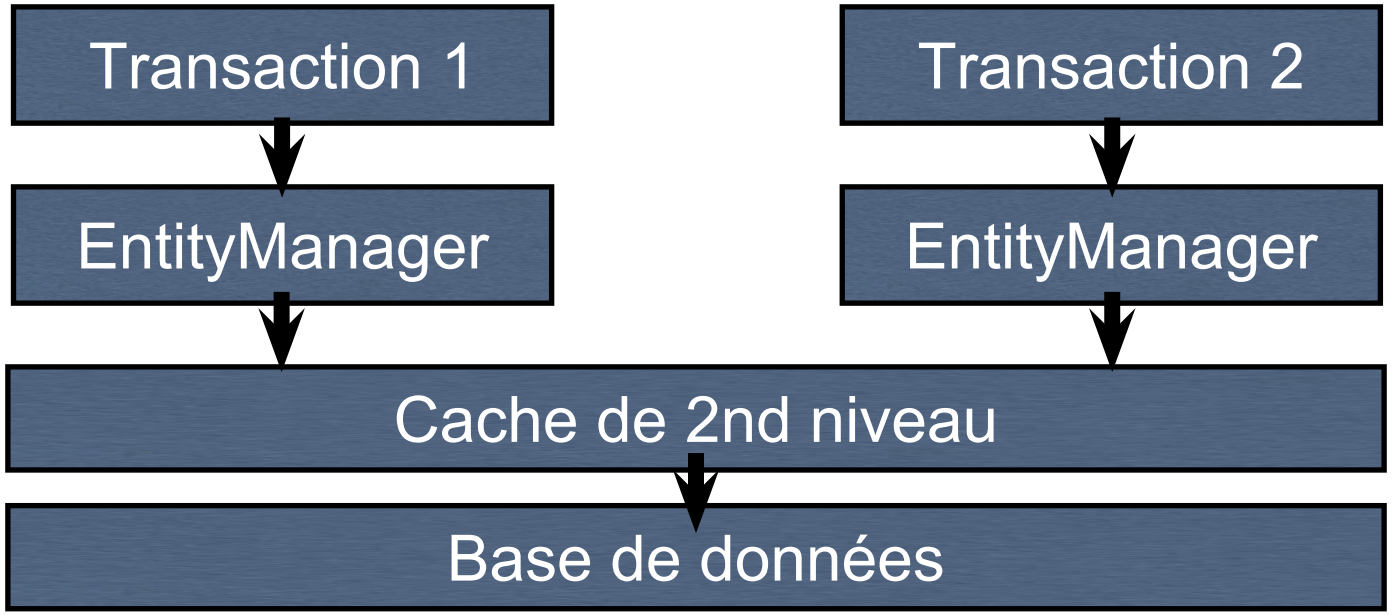
- On peut configurer JPA pour ne pas faire de lazy-loading sur une association :

```
@OneToMany(fetch=FetchType.EAGER)
private Set<Todo> todos;
```

- Hibernate va alors faire un outer-join : il ne fera donc qu'une seule requête qui va chercher l'ensemble des entités (l'entité principale et toutes les entités associées)
- **Attention**, si l'on utilise cette technique sur de trop nombreuses associations, la requête Hibernate va devenir très complexe

Le cache de 2nd niveau

- Le cache de 2nd niveau permet de stocker les entités fréquemment utilisées
  - Il est commun à tous les *EntityManager*
  - Il permet de considérablement améliorer les performances
  - Il faut utiliser une solution tierce : EHCache est la plus populaire



IPPON Enterprise Java Delivery

Conclusion sur JPA et Hibernate

- Une solution d'ORM permet de considérablement améliorer l'architecture d'une application Spring
  - Elimination du code SQL
  - Utilisation d'objets «du domaine» persistants, qui permettent de correctement modéliser le métier du client
  - Gain de temps important avec la fin de la couche Repository
- Et ce sans avoir à faire de compromis
  - JDBC est toujours accessible et utilisable conjointement

Exercice 6
Hibernate & JPA

Bean Validation

Qu'est-ce que Bean Validation ?

- Bean Validation est une spécification (JSR-303)
- L'implémentation de référence est Hibernate Validator, un sous-projet Hibernate
- Bean Validation permet de valider des Java Beans par l'ajout d'annotations
  - Cela permet de «renforcer» les entités Hibernate
    - Ces entités ne se contentent plus de stocker simplement des données, elles peuvent les valider
    - A rapprocher du chapitre précédent avec la «programmation défensive» et l'architecture DDD

  - Bean Validation est également utilisable en dehors d'Hibernate

Exemple d'entité avec Bean Validation

```java
@Entity
public class User implements Serializable {

    @Id
    @NotNull
    private int id;

    @Size(min = 0, max = 150)
    @Email
    private String email;

    @Past
    private Date creationDate;

    @Min(0)
    @NotNull
    private int balance;

    // getters et setters
}
```

IPPON Enterprise Java Delivery

Exemples d'annotations fournies

@Size(min=2, max=240)
@AssertTrue / @AssertFalse
@Null / @NotNull
@Max / @Min
@Future / @Past
@Digits(integer=6, fraction=2)
@Pattern(regexp="\\(\\d{3}\\)\\d{3}-\\d{4}")

Hibernate Validator

- Hibernate Validator est l'implémentation de référence
- Hibernate Validator propose un certain nombre d'annotations qui ne sont pas présentes dans la spécification

@Email
@URL
@CreditCardNumber
@Range

Faire ses propres annotations de validation

- Il est possible de créer des annotations spécifiques, qui correspondent à un besoin ou à un métier spécifique
  - Exemple : valider un numéro ISBN (code utilisé pour identifier les livres)

```
public class Book {

    @NotNull
    @Isbn
    private String isbn;

    @Size(min = 1, max = 1024)
    private String title;

    @NotNull
    private Author author;

    // getters et setters
}
```

Faire ses propres annotations de validation

```
@Constraint(validatedBy = IsbnValidator.class)
@Target(value = ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Isbn {
      String message() default "Mauvais numéro ISBN";
      Class<?>[] groups() default {};
      Class<? extends Payload>[] payload() default { };
}
```

```
import org.apache.commons.validator.routines.ISBNValidator;

public class IsbnValidator implements ConstraintValidator<Isbn, String> {

      public void initialize(Isbn constraintAnnotation) {
      }
      public boolean isValid(String value, ConstraintValidatorContext
context) {
            return ISBNValidator.isValidISBN13(value);
      }
}
```

Les groupes de validation

- On ne veut pas toujours valider une entité dans son ensemble
  - C'est en particulier utile dans la couche de présentation : il faut peut-être deux étapes pour avoir une entité complète
- On utilise alors des groupes de validations
  - Il peut y avoir plusieurs groupes, et on peut valider plusieurs groupes en une fois
  - Le groupe par défaut s'appelle «Default»
  - Il faut pour cela créer une interface qui correspond à ce groupe

Exemple de groupe de validation

```java
public interface MinimalBook {}
```

```java
public class Book {

    @NotNull(groups = {MinimalBook.class, Default.class})
    @Isbn
    private String isbn;

    @Size(min = 1, max = 1024)
    @NotNull(groups = MinimalBook.class)
    private String title;

    @NotNull
    private Author author;

    // getters et setters
}
```

Configuration et utilisation avec JPA

- Avec JPA 2, l'intégration de Bean Validation est automatique ! Il n'y a donc rien à faire
  - Avec JPA 1 il fallait rajouter un *listener* sur les entités
- Lorsqu'une entité est créée ou modifiée, Bean Validation va automatiquement la valider
  - En cas d'erreur, Bean Validation lancera une *Exception* de type *ConstraintViolationException*
  - Cette *Exception* fournit un *Set* de *ConstraintViolation*, lesquelles fournissent tous les détails sur les contraintes enfreintes (message d'erreur, nom de la contrainte, champ, etc...)
  - A priori, cette *Exception* va être traitée par la couche de présentation : il existe des tag libs ou des composants JSF spécialisés dans l'affichage de ces *Exceptions*

Utilisation programmatique de Bean Validation

- On peut également choisir de valider un Bean via l'API de Bean Validation
  - Si ce Bean n'est pas une entité Hibernate
  - Parce que l'on se trouve dans une autre couche de l'application : c'est la technique qui sera utilisée dans le chapitre traitant de Spring MVC

```
ValidatorFactory factory =
    Validation.buildDefaultValidatorFactory();
Validator validator = factory.getValidator();
Set<ConstraintViolation<Book>> constraintViolations =
    validator.validate(book);

for (ConstraintViolation<Book> constraintViolation : constraintViolations) {
     System.out.println(constraintViolation.getPropertyPath() + " - " +
     constraintViolation.getMessage());
}
```

Conclusion sur Bean Validation

- Bean Validation permet de valider des Java Beans, et tout particulièrement des entités JPA/Hibernate
- Bean Validation fonctionne grâce à un système d'annotations
  - Ces annotations peuvent être étendues
  - Ces annotations peuvent être séparées en groupes
- Bean Validation permet d'éviter d'avoir une architecture «anémique», et offre ainsi la possibilité d'avoir des entités capables de se protéger

Exercice 7
Bean Validation

Spring JMS

IPPON Enterprise Java Delivery

Introduction à JMS

- JMS (Java Message Service) est une API standard Java, permettant d'envoyer et de recevoir des messages de manière asynchrone
- Spring JMS propose une couche d'abstraction simplifiant l'utilisation de JMS dans un contexte Spring

Pourquoi utiliser un serveur de messages ?

- Ce système est **asynchrone**
  - Un client peut envoyer un message et reprendre immédiatement son travail, sans attendre de réponse
  - Un client peut recevoir un message, ce qui va déclencher un traitement
- Il permet une architecture **faiblement couplée**
  - Les systèmes clients sont connectés via la messagerie, et ne se connaissent pas entre eux. Ils n'utilisent pas forcément la même technologie
  - Les messages peuvent être routés d'un système à l'autre
  - Le serveur de messages sert de buffer : si l'un des systèmes clients est indisponible ou surchargé c'est le serveur de messages qui garantit la livraison des messages

Pourquoi utiliser JMS ?

- JMS est juste une API
  - API standard en Java, comme JDBC pour les bases de données
  - Il existe de nombreuses implémentations, dans de nombreuses technologies différentes : JMS permet donc de s'abstraire de ces implémentations propriétaires

Les implémentations de serveurs JMS

- Il existe de nombreuses implémentations de serveurs JMS
  - Apache ActiveMQ
  - HornetQ, de JBoss
  - Oracle AQ, d'Oracle
  - Websphere MQ, d'IBM (anciennement MQ Series)
  - RabbitMQ, de VMWare
  - SonicMQ, de Progress Software
  - etc etc etc...

Concepts de base

- JMS propose deux modèles
  - Point-to-Point
    - Un client envoie un message dans une «Queue»
    - Un autre client vient lire le message dans cette Queue
    - C'est le système de la boîte aux lettres
  - Publish/Subscribe
    - Un client envoie un message dans un «Topic»
    - Tous les clients abonnés à ce «Topic» reçoivent ce message
    - C'est le système de la liste de diffusion

ActiveMQ

- ActiveMQ est utilisé comme implémentation JMS dans ce cours
  - Open Source (Fondation Apache)
  - Très populaire
  - Robuste et fiable, fréquemment utilisé en production

- Il est écrit en Java et est très simple à lancer depuis Spring
  - Il est donc particulièrement adapté à des tests d'intégration
  - Le serveur JMS, au même titre que la base de données, fait partie de la configuration d'infrastructure

Exemple d'utilisation de l'API JMS standard

```
QueueConnectionFactory  queueConnectionFactory = (QueueConnectionFactory) jndiContext.
lookup("QueueConnectionFactory");

Queue queue = (Queue) jndiContext.lookup(queueName);
QueueConnection queueConnection = queueConnectionFactory.createQueueConnection();

QueueSession queueSession = queueConnection.createQueueSession(false, Session.
AUTO_ACKNOWLEDGE);

QueueSender queueSender = queueSession.createSender(queue);
TextMessage message = queueSession.createTextMessage();
message.setText("Voici un message");
queueSender.send(message);
```

IPPON Enterprise Java Delivery

Limitations de l'API standard

- Elle est complexe à utiliser : une dizaine de lignes pour un message simple !
- Elle nécessite de gérer des *Exceptions* (ignorées dans l'exemple précédent)
- Si vous n'utilisez pas un serveur Java EE, elle est synchrone en réception
  - Si vous êtes un consommateur de messages, vous ne pouvez pas les recevoir automatiquement : il faut poller la Queue ou le Topic
- Si vous utilisez un serveur Java EE, vous êtes dépendant de JNDI

Les types de messages

- TextMessage
  - Fournit une chaîne de caractères
  - Astuce : on peut envoyer ainsi un fichier XML
- BytesMessage
  - Un tableau de bytes
- StreamMessage
- MapMessage
  - Des paires clef/valeur
- ObjectMessage
  - Un objet Java sérialisé

JmsTemplate

- *JmsTemplate* propose un système de *Template*/*Callback*
  - Très similaire à *JdbcTemplate*, *TransactionTemplate*, etc...
  - Simplifie l'utilisation de l'API JMS
  - Abstrait le code de JNDI, et permet de ne plus en dépendre
  - Est capable de transformer automatiquement un Message JMS (TextMessage, etc...) en objet Java
- Comme JdbcTemplate, JmsTemplate est thread-safe
  - On crée un Template et on le réutilise ensuite
  - Il est même parfois configuré en tant que Bean Spring (Singleton)

Envoi d'un message avec JmsTemplate

```java
@Component
public class JmsQueueSender {

  private JmsTemplate jmsTemplate;

  @Inject
  private Queue queue;

  @Inject
  public void setConnectionFactory(ConnectionFactory cf) {
    this.jmsTemplate = new JmsTemplate(cf);
  }

  public void simpleSend() {
    this.jmsTemplate.send(this.queue, new MessageCreator() {
      public Message createMessage(Session session)  throws JMSException {
        return session.createTextMessage("Voici un message");
      }
    });
  }
}
```

Réception d'un message avec JmsTemplate

```
@Component
public class JmsQueueReceiver {

   private JmsTemplate jmsTemplate;

   @Inject
   private Queue queue;

   @Inject
   public void setConnectionFactory(ConnectionFactory cf) {
      this.jmsTemplate = new JmsTemplate(cf);
   }

   public void simpleReceive() {
      TextMessage message = (TextMessage) this.jmsTemplate.receive(queue);
      System.out.println(textMessage.getText());
    }
}
```

Accès à un serveur JMS avec JNDI

- Le serveur JMS est ici configuré dans un serveur d'applications Java EE
  - On trouve donc le serveur JMS, les Topics et les Queues via l'annuaire JNDI du serveur Java EE
  - Spring propose un namespace «jee» pour faciliter ces opérations de recherche dans JNDI
  - Les connexions au serveur JMS sont censées être mises dans un pool de connexions par le serveur d'applications (même principe que pour les connexions JDBC)

```
<jee:jndi-lookup id="connectionFactory" jndi-name="jms/ConnectionFactory"/>
<jee:jndi-lookup id="queue" jndi-name="jms/TestQueue"/>
```

Accès à un serveur ActiveMQ sans JNDI

```
<bean id="amqConnectionFactory"
    class="org.apache.activemq.ActiveMQConnectionFactory"    >
    <property name="brokerURL" value="tcp://localhost:61616" />
</bean>

<bean id="cachedConnectionFactory"
    class="org.springframework.jms.connection.CachingConnectionFactory" >
    <property name="targetConnectionFactory" ref="amqConnectionFactory" />
    <property name="sessionCacheSize" value="10"/>
</bean>

<bean id="queue" class="org.apache.activemq.command.ActiveMQQueue" >
     <constructor-arg value="Queue.TEST" />
</bean>

<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate" >
   <property name="connectionFactory" ref="cachedConnectionFactory" />
   <property name="defaultDestination" ref="queue"/>
</bean>
```

Lancement d'un serveur ActiveMQ depuis Spring

- Il est possible «d'embarquer» ActiveMQ directement dans une application Spring

```
<bean id="broker" class="org.apache.activemq.xbean.BrokerFactoryBean" >
    <property name="config" value="classpath:org/apache/activemq/xbean/activemq.xml"  />
    <property name="start" value="true" />
</bean>
```

- ActiveMQ étant lui-même basé sur Spring, il propose même un namespace «amq» pour faciliter sa propre configuration

```
<amq:broker useJmx="false" persistent="false">
   <amq:transportConnectors>
      <amq:transportConnector uri="tcp://localhost:0" />
   </amq:transportConnectors>
</amq:broker>
```

Réception automatique de messages

- La configuration que nous avons vu jusqu'à présent simplifiait juste l'utilisation et la configuration de l'API JMS
- Le problème, pour la réception de messages, c'est que nous sommes synchrones
  - Il faut appeler la méthode «receive()» pour voir si un message est arrivé
  - Cela force à faire du polling : une thread qui va régulièrement voir si il y a des messages en attente
- Une solution existe : ce sont les EJB «Message Driven», ou «MDB» (Message Driven Beans)
  - Ces objets sont automatiquement notifiés quand un message leur est destiné
  - Cela permet de faire de la programmation événementielle

Les «Message Driven POJOs»

- Spring propose des «Message Driven POJOs», c'est-à-dire des objets Java simples qui reçoivent automatiquement des message JMS
  - Spring fonctionne avec l'interface *MessageListener* de JMS, mais l'utilisation de cette interface n'est pas obligatoire (on a alors de «vrais» POJOs, qui ne connaissent même pas JMS)
  - Spring utilise pour cela un pool de threads, que l'on peut configurer si nécessaire

Exemple de MessageListener

```java
@Component
public class SimpleMessageListener implements MessageListener {

   public void onMessage(Message message) {
      try {
         TextMessage testMessage = (TextMessage) message;
         System.out.println(textMessage.getText());
      } catch (JMSException e) {
         e.printStackTrace();
      }
   }
}
```

```xml
<jms:listener-container
      container-type="default"
      connection-factory="connectionFactory"
      acknowledge="auto">

    <jms:listener destination="test.Queue" ref="simpleMessageListener" />
</jms:listener-container>
```

La gestion des transactions

- Pour qu'un message soit considéré comme traité, un accusé de réception doit être renvoyé
  - Selon la configuration, cet accusé de réception peut être renvoyé par le client ayant lu le message, ou être envoyé automatiquement
- Si l'on utilise des sessions JMS transactionnelles, les messages vont pouvoir être commités ou rollbackés
  - Fonctionnement similaire à une transaction en base de données
  - Peut être géré via JTA
  - Attention : à moins que vous n'utilisiez la technologie XA, vos transactions JMS et vos transactions JDBC seront deux choses séparées (voir le chapitre sur les transactions à ce sujet)

IPPON Enterprise Java Delivery

Conclusion sur JMS

- JMS est une technologie essentielle pour concevoir des architectures hautement disponibles, capables de monter en charge, et regroupant des sous-systèmes parfois hétérogènes
- JMS peut être transactionnel, et garantit la bonne distribution des messages
- Spring JMS permet de simplifier l'envoi et la réception de messages
- Spring JMS permet également de recevoir des messages de manière asynchrone, et donc d'être notifié à la réception d'un message

Exercice 8
Spring JMS

Spring JMX

Introduction à JMX

- Java Management Extensions (JMX) est une technologie standard Java permettant l'étude (monitoring) et la gestion (management) d'objets Java
  - Ces objets sont appelés des «managed beans», ou MBeans
- JMX est intégré dans la JVM
- JMX est accessible à distance
  - Il existe de nombreux outils permettant de gérer des MBeans, le plus simple étant JConsole, qui est fourni avec la JVM
  - Sur un poste où Java est installé, lancer «jconsole» pour faire apparaître la console

IPPON Enterprise Java Delivery

Pourquoi utiliser JMX ?

- En production, il est essentiel de pouvoir monitorer/manager les services Java
- JMX n'a pas vraiment de concurrent en Java, et c'est ce qu'utilisent de nombreux projets Open Source
- Tomcat, Hibernate, ActiveMQ, ...
- Tous les outils de monitoring/management du marché supportent JMX

IPPON Enterprise Java Delivery

Exemple avec ActiveMQ

- ActiveMQ est accessible via JMX
  - Lancer un serveur ActiveMQ (voir le chapitre précédent)
  - Lancer une console JMX
- JConsole propose un arbre avec les MBeans exposés
  - La JVM et ActiveMQ exposent un certain nombre de services
  - Les MBeans permettent de monitorer le système
    - Exemple : le nombre de messages en attente dans une Queue JMS
  - Les MBeans permettent de manager le système
  - Exemple : vider une Queue JMS

Demo
Monitoring JMX d'ActiveMQ

Spring et JMX

- Exposer un Bean Spring en JMX est une simple configuration Spring
  - Pas besoin de coder quoi que ce soit
  - Une configuration plus précise (méthodes exposées, etc) peut être réalisée en XML ou via des annotations

- Astuce : stocker cette configuration dans un fichier séparé, de manière à pouvoir facilement enlever/modifier/reconfigurer le monitoring
  - C'est encore un bon exemple de configuration «d'infrastructure»

Exemple d'export de Beans Spring en JMX

```
<bean id="mbeanServer"
      class="org.springframework.jmx.support.MBeanServerFactoryBean"
      lazy-init="false"/>

<bean id="exporter" class="org.springframework.jmx.export.
MBeanExporter">
    <property name="beans">
      <map>
        <entry key="spring:name=todoService" value-ref="todoService"/>
      </map>
    </property>
    <property name="server" ref="mbeanServer"/>
</bean>

<bean id="todoService" class="test.TodoService">
    <property name="exemple" value="TEST"/>
</bean>
```

Utilisation des annotations (1/2)

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="namingStrategy" ref="namingStrategy"></property>
  <property name="assembler" ref="assembler"></property>
</bean>

<bean id="attributeSource" class="org.springframework.jmx.export.annotation.
AnnotationJmxAttributeSource"/>

<bean id="assembler" class="org.springframework.jmx.export.assembler.
MetadataMBeanInfoAssembler">
  <property name="attributeSource" ref="attributeSource"/>
</bean>

<bean id="namingStrategy" class="org.springframework.jmx.export.naming.
MetadataNamingStrategy">
  <property name="attributeSource" ref="attributeSource"/>
</bean>
```

Utilisation des annotations (2/2)

```
@Service
@ManagedResource(objectName = "spring:name=accountService")
public class AccountService {

    public int balance;

    @ManagedAttribute
    public int getBalance() {
        return balance;
    }

    public void setBalance(int balance) {
        this.balance = balance;
    }
}
```

Monitoring d'Hibernate avec JMX

- Spring peut également exposer des MBeans existants : on peut ainsi exposer le service de statistiques d'Hibernate
  - C'est un excellent moyen d'étudier le comportement et les performances d'Hibernate

```
<bean id="jmxExporter" class="org.springframework.jmx.export.MBeanExporter" >
  <property name="beans">
    <map>
      <entry key="Hibernate:type=statistics" value-ref="statisticsBean" />
    </map>
  </property>
</bean>

<bean id="statisticsBean" class="org.hibernate.jmx.StatisticsService" >
  <property name="statisticsEnabled" value="true" />
  <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Conclusion sur JMX

- JMX est une technologie essentielle pour pouvoir gérer une application en production
  - Elle permet de monitorer et de manager l'application
  - C'est une technologie standard, largement utilisée en Java
- JMX peut être relativement compliqué à mettre en place
  - Spring permet d'utiliser JMX par configuration uniquement
  - Spring permet d'exposer facilement tout Bean Spring
  - Spring permet également d'exposer des MBeans existants

IPPON Enterprise Java Delivery

«Le web»

- Spring MVC
- Spring MVC pour les API Web
- Spring Web Flow
- Spring Security

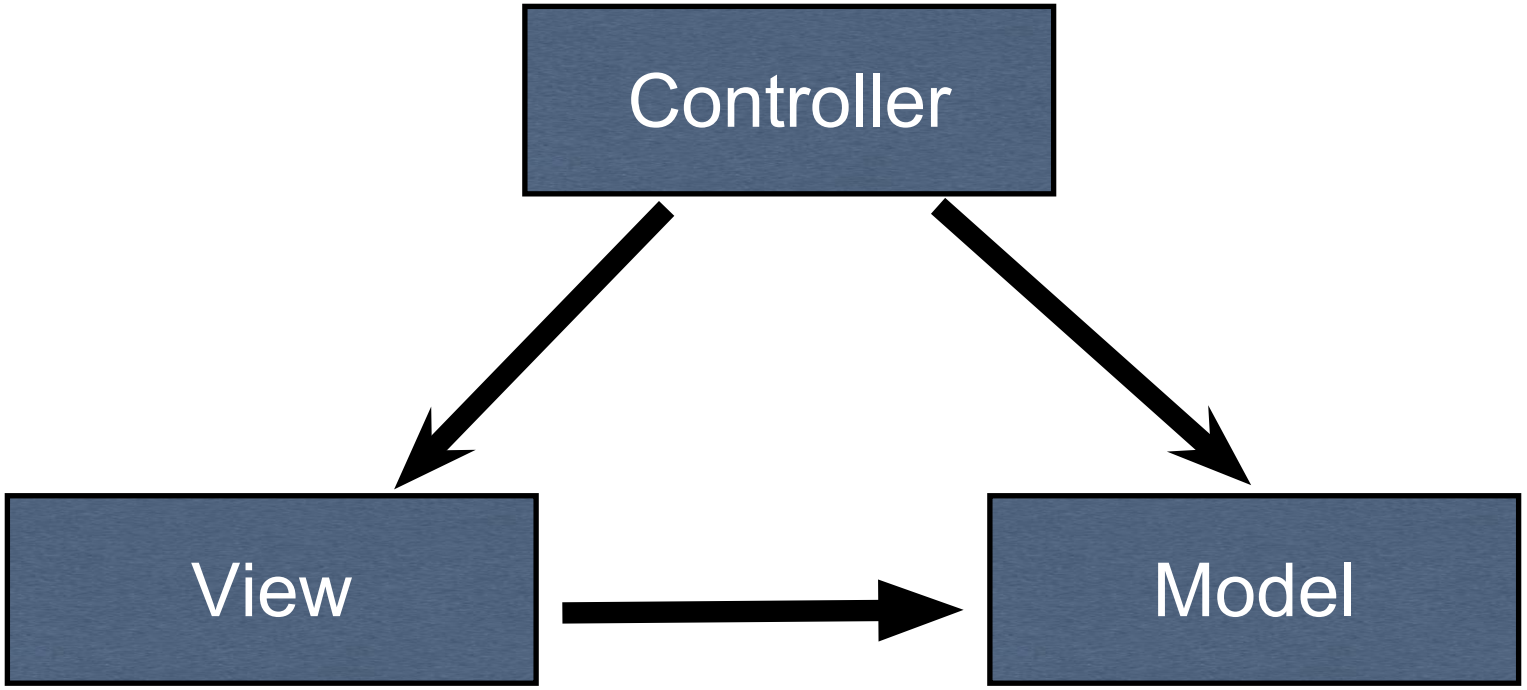
Spring MVC

Introduction à Spring MVC

- Spring MVC est un framework Model-View-Controller
  - Il permet de faire des applications Web
  - Il est relativement simple à apprendre
  - Il est performant et robuste
  - Il est très configurable
- Spring MVC est intégré à Spring Core (ce n'est pas un sous-projet)

Le modèle MVC

- Le modèle MVC a été popularisé en Java par Struts
  - Spring MVC est un concurrent direct de Struts



Les composants de Spring MVC

- Le Controller
  - Un Bean Spring, annoté *@Controller*
  - Supporte donc l'injection de dépendance : c'est ainsi qu'il accède à la couche Service
  - C'est lui qui valide la requête, remplit le modèle, et redirige vers la bonne vue
- La vue
  - C'est typiquement une JSP qui va être chargée d'afficher le modèle
  - Spring MVC n'est pas limité à la technologie JSP, mais c'est elle qui est le plus couramment utilisée
- Le modèle
  - Un Java Bean ou un ensemble de Java Beans (Collection), provenant de la couche service et étant affiché par la JSP
  - Dans l'architecture que nous avons étudiée, ce Java Bean est généralement une entité Hibernate, ou un ensemble d'entités Hibernate
  - Il peut être validé par Bean Validation (cf. chapitre sur Bean Validation)

Pré-requis : lancer Spring

- Avant de configurer et lancer Spring MVC, il faut configurer et lancer Spring
  - Il s'agit de configurer un listener de Servlet, dans le fichier web.xml
  - Ce listener va lire le fichier de configuration Spring passé en paramètre, et se charger de lancer le contexte Spring
  - Comme nous l'avons vu dans les chapitres précédents, Spring va alors trouver la Data Source, lancer JPA, gérer les transactions... Nous aurons donc ainsi une application pleinement fonctionnelle

```
<context-param>
  <param-name>contextConfigLocation </param-name>
  <param-value>classpath:META-INF/spring/application-context.xml </param-value>
</context-param>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener </listener-class>
</listener>
```

Configuration de Spring MVC

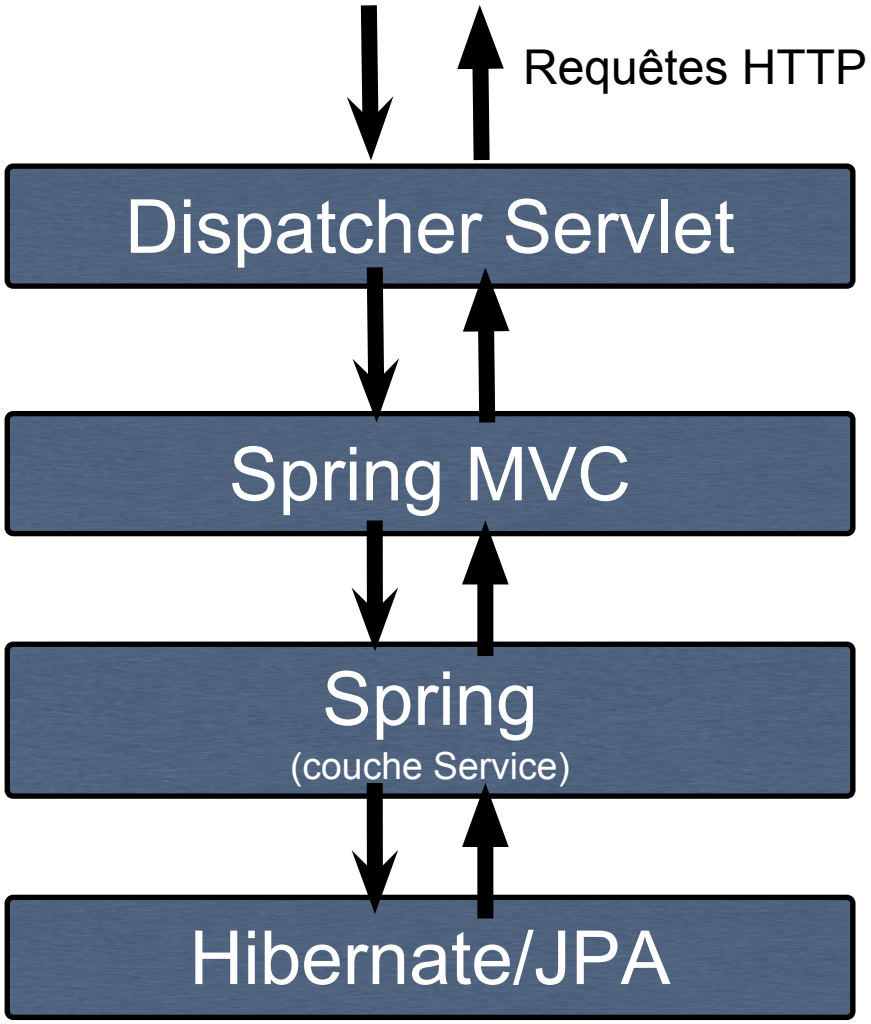
- La classe principale est la *Servlet* nommée *DispatcherServlet*
  - Elle lance un Application Context Spring
  - Elle lui transmet ensuite les requêtes HTTP

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/app/*</url-pattern>
</servlet-mapping>
```

Lancement du context Spring MVC

- Cette *Servlet* va rechercher un fichier de configuration Spring
  - Par défaut, elle lit le fichier
    WEB-INF/${NOM_DE_LA_SERVLET}-servlet.xml
  - Dans notre exemple, il s'agit donc de
    WEB-INF/dispatcher-servlet.xml
- Elle va alors instancier un contexte applicatif Spring, qui sera un enfant du contexte applicatif principal
  - Voir les Application Contexts hiérarchiques, dans le chapitre sur la configuration Spring avancée

Fonctionnement de Spring MVC avec Spring



Fichier de configuration de Spring MVC

```
<beans xmlns="...">

    <context:component-scan base-package="exemple.test.web"/>
    <mvc:annotation-driven/>

</beans>
```

Choix de la JSP en fonction de la vue demandée

- En fonction du nom de la vue demandée, Spring MVC va retrouver la JSP correspondante
- Généralement, il s'agit juste de mettre un préfixe et un suffixe à la vue demandée
  - La vue «account» correspond ainsi à la JSP «*WEB-INF/jsp/account.jsp*»

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.UrlBasedViewResolver" >
    <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
</bean>
```

IPPON Enterprise Java Delivery

Configuration d'un *Controller*

- Un *Controller* Spring MVC est un Bean Spring
  - On peut lui injecter des Beans Spring de la couche service, qui sont dans l'Application Context Spring parent
- Il se configure uniquement par annotations
  - Historiquement en XML
  - Généralement, il n'a pas d'interface et on lui laisse son nom par défaut : il n'est pas destiné à être injecté dans un autre Bean

IPPON Enterprise Java Delivery

Exemple de *Controller*

```
@Controller
@RequestMapping("/account")
public class AccountController {

    @Inject
    private UserService userService;

    @RequestMapping(method = RequestMethod.GET)
    public ModelAndView display() {
        User user = userService.getCurrentUser();
        ModelAndView mv = new ModelAndView("account");
        mv.addObject("user", user);
        return mv;
    }

    @RequestMapping(method = RequestMethod.POST)
    public String update(@RequestParam String name) {
        userService.updateName(name);
        return "account_ok";
    }
}
```

Mapping des requêtes

- Les requêtes peuvent être mappées au niveau de la classe ou des méthodes
- Le mapping s'effectue sur un chemin, mais peut aussi être fait sur d'autres critères comme en fonction du verbe HTTP utilisé
  - GET pour lire une donnée
  - POST pour la mettre à jour

Gestion des paramètres passés à la requête

- La manière la plus simple : ajouter à la méthode un argument annoté avec *@RequestParam*
- Par défaut, ce paramètre est obligatoire et a le même nom que le nom de la variable

```
public String update(@RequestParam String name) {...}
```

- Cela est bien entendu paramétrable :

```
public String update(@RequestParam(value="userName", required="false") String name) {...}
```

@ModelAttribute sur une méthode (1/2)

- La première utilisation de *@ModelAttribute* est de mettre dans le modèle un objet métier
  - Cet objet peut être une entité JPA
  - Il peut être également un objet spécifiquement développé pour la vue
- Lorsqu'une méthode est annotée avec *@ModelAttribute*, cette méthode est appelée avant chaque mapping (méthodes annotées *@RequestMapping*)
  - Ainsi, on est certain d'avoir les bonnes données dès le début
  - Attention : cela peut avoir un impact de performances, surtout si cet objet n'est pas utilisé dans toutes les méthodes annotées *@RequestMapping*

@ModelAttribute sur une méthode (2/2)

- Dans cet exemple, un objet de type *User* sera stocké dans le modèle, avec la clef «user» :

```
@ModelAttribute("user")
public User formBackingObject() {
   return userService.getCurrentUser();
}
```

@ModelAttribute sur un paramètre de méthode

- Sur un paramètre d'une méthode annotée *@RequestMapping*, l'annotation *@ModelAttribute* va permettre de «binder» les paramètres de la requête HTTP sur cet objet

```
@RequestMapping(method = RequestMethod.POST)
public String register(@ModelAttribute("user") User user,
                       BindingResult result) {

    if (result.hasErrors()) {
        return "register";
    }
    return "register_ok";
}
```

L'objet *ModelAndView*

- L'objet *ModelAndView* est retourné par les méthodes du *Controller*
  - Comme son nom l'indique, il représente le modèle et la vue du pattern MVC
- Le modèle est un objet *ModelMap*, qui est une *Map* améliorée
  - Les objets mis dans cette *Map* se retrouvent en attributs dans la requête HTTP
- Si le *Controller* ne retourne qu'une String, il s'agit de la vue à renvoyer, avec un modèle vide

```
ModelAndView mv = new ModelAndView();
mv.setViewName("account");
mv.addObject("user", user);
```

```
ModelAndView mv = new ModelAndView("account");
mv.addObject("user", user);
```

La Tag Library de Spring MVC

- Côté vue, Spring MVC propose une Tag Library pour gérer les formulaires
  - Cela permet de binder les données du formulaire HTML avec les méthodes et les paramètres annotés *@ModelAttribute*

```
<form:form modelAttribute="user">
  <form:input path="firstName" size="15" maxlength="60"/><br/>
  <form:input path="lastName" size="15" maxlength="60"/><br/>
  <form:select path="sex">
    <form:option value="M">M</form:option>
    <form:option value="F">F</form:option>
  </form:select><br/>
  <input type="submit" value="Save Changes" />
</form:form>
```

Spring MVC et Bean Validation (1/2)

- Dans la configuration de Spring MVC, la ligne suivante permet de prendre en compte Bean Validation :

```
<mvc:annotation-driven/>
```

- Ensuite, tout paramètre annoté @*Valid* sera validé par Bean Validation avant binding

```
@RequestMapping(method = RequestMethod.POST)
public String register(@Valid User user, Errors
errors) {
    if (errors.hasErrors()) {
        return "register";
    }
    return "register_ok";
}
```

Spring MVC et Bean Validation (2/2)

- Cette configuration permet d'utiliser Bean Validation, qui est un standard, pour valider les formulaires HTML
  - Dans ce cadre, on aura certainement des Java Beans spécifiques pour gérer les formulaires, comme les FormBeans de Struts
- Elle permet également de valider automatiquement les objets de domaines que l'on afficherait directement dans la page
  - Cela reprend les principes du «DDD» et d'objets de domaine «forts» : on peut maintenant directement interagir avec ces objets dans la page HTML
  - Attention, les entités JPA sont ainsi validées deux fois : une fois au niveau du Controller Spring MVC, et une fois au niveau d'Hibernate

IPPON Enterprise Java Delivery

Conclusion sur Spring MVC

- Spring MVC est un framework MVC simple et puissant
  - Il est facile d'accès et très performant
  - Il se configure entièrement grâce à Spring
  - Il utilise son propre Application Context
- Il est très souple et très paramètrable
  - Cela peut être déroutant, et fait que l'on a parfois des configurations très complexes
- Il s'intègre avec Bean Validation pour valider les formulaires

Exercice 9
Spring MVC

Spring MVC REST

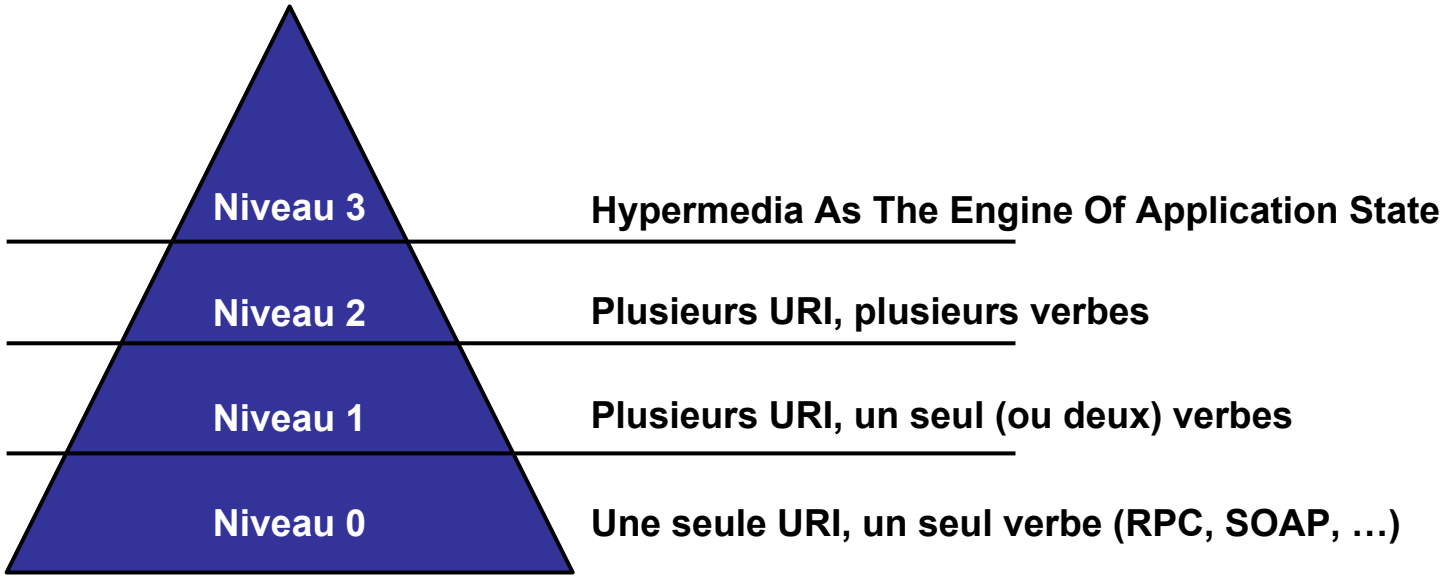
IPPON Enterprise Java Delivery

Qu'est-ce que REST ?

- REST est un style d'architecture distribuée
  - De nombreux frameworks facilitent sa mise en oeuvre
  - Spring MVC est l'un d'entre eux, son principal intérêt étant justement de ne pas être limité à ce style d'architecture
- REST propose de travailler avec des représentations de ressources, adressables (URL), sur lesquelles il est possible d'agir à travers l'interface qu'est HTTP (via ses méthodes GET, POST, PUT, DELETE, …)
- Cette architecture est très bien adaptée au Web
  - Elle prône une approche stateless
  - Elle tire pleinement partie de HTTP (permet d'utiliser le cache des navigateurs correctement par exemple)
- Cette architecture pousse le client à gérer l'état
  - Une page Web avec beaucoup d'Ajax
  - Une application mobile

Qu'est-ce que les API Web ?

- Le terme API Web est moins contraignant qu'API REST
  - Désigne une API pouvant s'inspirer des principes de REST
  - N'oblige pas à honorer l'ensemble des contraintes de REST
- Une API Web pourra donc se conformer **plus ou moins** au style d'architeture REST
  - On parlera de degré de maturité ou modèle de maturité de Richardson



Introduction aux API Web avec Spring MVC

- Spring MVC permet la réalisation d'API Web via un support partiel de REST
  - Capacité à créer des représentations de ressources (typiquement les entités JPA)
  - Capacité à créer des URLs unique vers ces ressources et de les manipuler avec les verbes HTTP
    - GET /accounts/123 renverra l'objet Account avec la clef 123
    - POST /accounts/123 modifiera l'objet Account avec la clef 123
    - PUT /accounts créera un nouvel objet Account
  - Capacité à utiliser différents media types pour représenter la ressource en fonction d'un header HTTP ou du suffixe de l'URL
    - «account.json» renverra une représentation au format JSON, «account.xml» au format XML par exemple

Exemple de *Controller* pour API

- L'annotation *@PathVariable* permet de binder un paramètre de l'URL sur un paramètre de la méthode Java
  - Le nom entre {} dans l'URL est par défaut celui du paramètre Java

```
@Controller
public class TodoListsRest {

    @Inject
    private TodoListsService todoListsService;

    @RequestMapping(value = "/lists/{listId}/todos.json", method =
RequestMethod.GET)
    public Collection<Todo> todos(@PathVariable String listId) {
        TodoList todoList = todoListsService.findTodoList(listId);
        return todoList.getTodos();
    }
}
```

IPPON Enterprise Java Delivery

Utilisation des verbes HTTP

- Une API Web doit tirer profit du protocole HTTP et en utiliser les verbes pour agir sur les ressources
  - GET pour accéder à une ressource
  - POST pour mettre à jour une ressource
  - PUT pour créer une ressource
  - DELETE pour effacer une ressource
- En HTML, nous n'avons accès qu'aux verbes GET et POST
  - Spring MVC simule PUT et DELETE en utilisant le verbe POST avec un paramètre caché dans les formulaires (grâce au filtre *HiddenHttpMethodFilter*)

```
<form:form method="delete">
  <input type="submit" value="Delete Account"/>
</form:form>
```

Représentation JSON d'une entité

- JSON est un format très populaire
  - Plus compact que XML, tout en étant aussi lisible
  - Issu de JavaScript, naturellement compris par les navigateurs
- Spring MVC propose une intégration avec Jackson (http://jackson.codehaus.org/), qui permet de transformer des objets Java en JSON (et inversement)
  - Jackson transformera automatiquement au format JSON les entités ou collections d'entités retournées par les Controllers

```
@RequestMapping(value = "/api/lists.json", method = RequestMethod.GET)
@ResponseBody
public Collection<TodoList> lists() {
    User user = userService.getCurrentUser();
    return user.getTodoLists();
}
```

Utilisation du Content-Type

- Spring MVC peut utiliser le Content-Type (header HTTP) de la requête pour comprendre qu'un objet envoyé est au format JSON
  - L'annotation *@RequestBody* permet de binder le corps de requête HTTP sur un objet Java
  - Jackson va se charger de transformer le corps de la requête en objet Java

```
@RequestMapping(value = "/api/todos",
                method = RequestMethod.POST,
                consumes="application/json")
public void addTodo(@RequestBody Todo todo, Model model) {
    // ajout du Todo
}
```

Configuration de Jackson via Spring

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver" >
  <property name="mediaTypes">
    <map>
      <entry key="html" value="text/html" />
      <entry key="json" value="application/json" />
    </map>
  </property>
  <property name="viewResolvers">
    <list>
      <bean class="org.springframework.web.servlet.view.BeanNameViewResolver" />
      <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver" >
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
      </bean>
    </list>
  </property>
  <property name="defaultViews">
    <list>
      <bean class="org.springframework.web.servlet.view.json.MappingJackson2JsonView" />
    </list>
  </property>
</bean>
```

Utilisation des ETags

- Un ETag (Entity Tag) est un header HTTP 1.1
  - Identifiant unique donné par un serveur à une ressource Web
  - Le navigateur peut ainsi mettre en cache la ressource : quand il la demandera de nouveau, il renverra le ETag, et si elle n'a pas changé le serveur renverra une réponse *HTTP 304 Not Modified*
  - Cela permet donc de gagner en trafic réseau
- Spring propose le filtre *ShallowEtagHeaderFilter* pour automatiquement gérer les ETags des ressources REST
  - Attention : pour fonctionner, la requête doit quand même s'exécuter (nous sommes dans un environnement dynamique, impossible de savoir si la ressource a été modifiée ou pas). Ce filtre permet donc de gagner du trafic réseau, mais rien d'autre (l'utilisation du serveur reste identique sinon)

Configuration du ShallowEtagHeaderFilter

```
<filter>
  <filter-name>etagFilter</filter-name>
  <filter-class>org.springframework.web.filter.ShallowEtagHeaderFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>etagFilter</filter-name>
  <servlet-name>dispatcher</servlet-name>
</filter-mapping>
```

Utilisation avec JQuery

- Côté client, le moteur JavaScript comprend le format JSON
- Avec une librairie comme JQuery, il est très simple de faire des appels Ajax vers une API Spring MVC
  - Les objets échangés sont au format JSON
  - Les URLs “propres” sont faciles à utiliser avec JQuery
  - L’état est bien géré côté client (nous sommes stateless côté serveur)

```
$.get("/api/accounts/123", { } ,
    function(account) {
        $('#container').append('<div>' + account.login + ' - <b>' +
        account.firstName + ' ' + account.userLastName + '</b></div>');
    },
    "json");
```

Conclusion sur Spring MVC pour les API Web

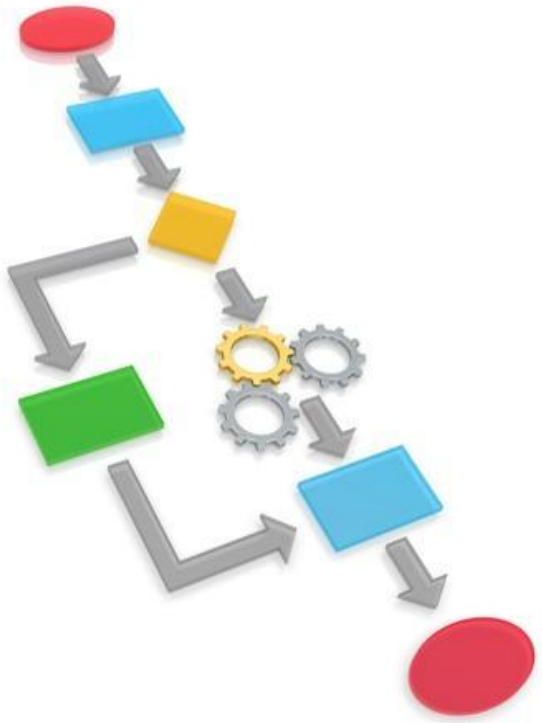
- Spring MVC permet de simplifier la réalisation d'API Web
  - Support (partiel) de REST : URLs, verbes HTTP, gestion du *Content-Type*, gestion des ETags...
  - Possibilité de conserver Spring MVC pour réaliser des pages Web plus classiques, en particulier avec un état géré coté serveur
- C'est une excellente manière de réaliser des applications Web «riches»
  - Le support de JSON permet à du code JavaScript d'interagir facilement avec une API Spring MVC
  - En particulier, on peut réaliser un client JQuery qui communique via des requêtes Ajax avec une API codée avec Spring MVC

Exercice 10
Spring MVC REST

Spring Web Flow

Introduction à Spring Web Flow

- Spring Web Flow est une extension de Spring MVC, qui permet de gérer les enchaînements de pages
  - Cet enchaînement est un workflow, dans lequel un état complexe va être manipulé sur plusieurs pages
  - Par exemple : une réservation de chambre d'hôtel, le paiement d'un ensemble de produits



- Spring Web Flow est en fait une machine à état
  - Un état de départ
  - Plusieurs états intermédiaires, avec des transitions entre ces états qui sont définies
  - Un ou plusieurs états de fin
  - Une fois un état de fin atteint, il n'est plus possible de rejouer cette instance de workflow

IPPON Enterprise Java Delivery

La notion de «flow»

- Un flow est un concept apporté par Spring Web Flow, qui est intermédiaire entre une requête et une session HTTP
  - Un flow dure sur plusieurs requêtes HTTP
  - Il peut y avoir plusieurs flows dans une session HTTP
- Un flow définit également un scope spécifique pour les Beans Spring
  - Voir la section sur les «scopes de Beans» dans le chapitre «Spring IoC»

IPPON Enterprise Java Delivery

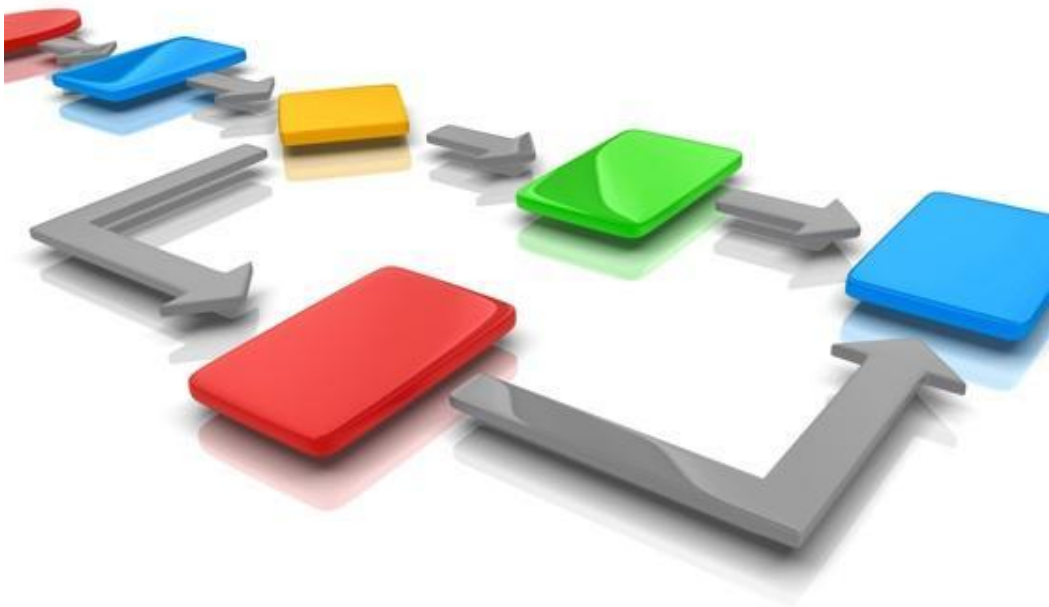
Fonctionnalités de Spring Web Flow

- On peut lancer plusieurs fois le même flow en parallèle (support des onglets du navigateur)
- Le bouton «back» est supporté : Web Flow stocke l'historique des états de chaque flow
- On peut créer des sous-flows et les réutiliser dans plusieurs flows parents
- On peut utiliser un «persistence context étendu», c'est-à-dire un persistence context JPA qui «vit» sur plusieurs pages HTML
- Spring Web Flow permet de gérer la navigation dans des pages JSF
- Les flows peuvent être sécurisés avec Spring Security
- En développement, chaque flow peut être reconfiguré à la volée (pas de redéploiement nécessaire)

Demo
Spring Web Flow

La définition d'un Flow

- Un Flow est défini par un DSL (domain-specific langage) : un schéma XML spécifique qui permet de définir les états et leurs transitions
  - Un fichier de définition de Flow décrit ainsi toutes les étapes d'un flow, et fait partie du travail de développement
  - Certains IDE (SpringSource Tool Suite et Intellij IDEA) permettent de construire ces flows graphiquement, mais dans la pratique ils sont souvent écrits directement en XML



Intégration de Spring Web Flow avec Spring MVC

```
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
  <property name="flowExecutor" ref="flowExecutor"/>
</bean>

<webflow:flow-executor id="flowExecutor"/>

<webflow:flow-registry id="flowRegistry"
  flow-builder-services="flowBuilderServices" base-path="/WEB-INF">
  <webflow:flow-location-pattern value="flows/**/*-flow.xml" />
</webflow:flow-registry>

<webflow:flow-builder-services id="flowBuilderServices" development="true"/>
```

- Pour une URL de type «example/test», on utilisera une instance d'un flow dont la configuration est stockée dans «WEB-INF/flows/example/test-flow.xml»

Exemple de flow

```
<flow xmlns="...">
  <view-state id="newAccount">
    <transition on="submit" to="validateNewAccount" />
  </view-state >
  <view-state id="validateNewAccount">
    <transition on="confirm" to="accountConfirmed" />
    <transition on="revise" to="newAccount" />
    <transition on="cancel" to="newAccountCancelled" />
  </view-state >
  <end-state id="accountConfirmed" />
  <end-state id="newAccountCancelled" />
</flow>
```

Les éléments principaux

- **start-state** : état de départ
- **view-state** : état qui affiche une vue (page JSP, JSF, etc...)
- **transition** : permet de passer d'un état à un autre
- **decision-state** : permet de faire un choix en fonction d'une expression (if-then-else)
- **end-state** : état de fin (une fois cet état atteint, le flow est détruit)

Expression Language (EL)

- Spring Web Flow peut utiliser un Expression Language (EL) afin de pouvoir exécuter des méthodes Java dans des transitions ou des états
  - Il y a 3 EL supportés : Spring EL, Unified EL et OGNL
  - Les 3 sont très proches mais Spring EL est recommandé. Nous allons donc nous concentrer sur lui.
- SpEL est utilisable dans de nombreux projets Spring (Spring Batch, par exemple)
  - Il permet d'accéder à des Beans Spring et de les invoquer
  - Il peut accéder à des instances de Java Beans, par exemple pour lire leurs propriétés
  - Il peut effectuer des branchements conditionnels

Utilisation de SpEL avec Spring Web Flow (1/2)

```
<view-state id="accountSelect">
  <on-entry>
    <evaluate expression="accountService.find()" result="viewScope.accounts"/>
  </on-entry>
</view-state>
```

- A l'entrée dans l'état «accountSelect», et donc avant l'affichage de la page
  - Exécute la méthode «*find*» sur le Bean Spring nommé «*accountService*»
  - Stocke le résultat de cette méthode dans le scope «*view*», de manière à ce qu'il soit accessible par la vue

Utilisation de SpEL avec Spring Web Flow (2/2)

```
<transition on="deleteAndClose" to="end">
  <evaluate expression="accountService.delete(account)" />
</transition>
```

- La transition sur l'événement «*deleteAndClose*» va renvoyer vers un état «*end*» (à priori un end-state)
- Le passage par cette transition déclenche l'exécution d'un SpEL
  - Sur le Bean Spring nommé «*accountService*»
  - La méthode «*delete*» est appelée avec en paramètre l'objet nommé «*account*» qui est stocké dans le scope flow

IPPON Enterprise Java Delivery

Gestion du bouton «back»

- Spring Web Flow stocke chaque état d'un flow en cours
  - Par défaut 30 états par flow, avec 5 flows ouverts
- Toutes les données mises dans le scope «flow» sont alors «snapshotées»
  - Elles sont sérialisées et stockées dans le flow en cours
  - Si un état précédent est demandé (bouton «back»), ces données sont désérialisées et l'état est ainsi reconstruit
- Ce mécanisme est très robuste
  - On peut ainsi revenir à un état antérieur sans problème
  - Cela fonctionne même en cluster ! (les flows sont stockés en Session, et tout cela est sérialisable)
- Par contre il faut être attentif
  - Cela ne marche qu'avec les valeurs en scope «flow»
  - Cela peut utiliser beaucoup de mémoire : par exemple si on stocke un tableau de résultats dans le scope «flow»

IPPON Enterprise Java Delivery

Le persistence context étendu

- Spring Web Flow permet d'utiliser un «persistence context étendu»
- Il s'agit d'une fonctionnalité avancée de JPA
  - Le persistence context JPA (= la session Hibernate) va survivre pendant la durée du flow
  - A chaque page affichée, une nouvelle transaction est créée, et JPA va y rattacher son persistence context
  - Ce n'est que lorsqu'on passe à un état en écriture que ce persistence context (= cache de premier niveau Hibernate) va «flusher» et donc être envoyé et commité en base de données
- Ce mécanisme est particulièrement adapté à Web Flow
  - Pendant plusieurs états, plusieurs entités JPA vont être modifiées
  - A la fin du flow, soit tout est annulé (pas de flush), soit tout est commité en base de données (flush + commit)

IPPON Enterprise Java Delivery

Conclusion sur Spring Web Flow

- Spring Web Flow est un complément à Spring MVC
  - Il n'est utile que dans un certain cas : les enchaînements d'écrans complexes
- Spring Web Flow permet de maintenir un état entre ces enchaînements
  - Cela peut être relativement coûteux en termes de performances
  - Cela va considérablement simplifier le développement de ce type d' applications
- Spring Web Flow s'intègre avec JSF, Bean Validation et Spring Security, ce qui permet de faire des applications «riches»

Spring Security

Introduction à Spring Security

- Spring Security permet de sécuriser des applications Spring, en particulier dans un environnement Java EE
- Il fonctionne dans tous les serveurs d'applications
  - Il vient alors remplacer la sécurité standard Java EE
  - Il est très répandu en production, car il propose des fonctionnalités particulièrement avancées

- Il se configure via un fichier de configuration Spring
- Il peut sécuriser des URLs mais aussi des méthodes Java

IPPON Enterprise Java Delivery

Pourquoi utiliser Spring Security ?

- Spring Security est une alternative à la sécurité fournie par les serveurs d'applications Java EE
  - Il n'y a pas vraiment d'autre concurrent actuellement (on citera tout de même Apache Shiro et PicketLink)
- Il se configure via une configuration Spring, et bénéficie de toute la richesse de celle-ci
  - Par exemple, le fait d'avoir facilement plusieurs fichiers d'infrastructure en fonction de l'environnement est très utile pour un *framework* de sécurité
- Il est portable d'un serveur à un autre
- Il propose une large palette de *plugins* : utilisation de solutions de SSO (CAS), de la sécurité Windows (NTLM), de OpenID...
- Il fournit de nombreux services non disponibles avec Java EE : gestion des *cookies* «remember me», sécurité des instances d'objets, etc...

IPPON Enterprise Java Delivery

Authentification et Autorisation

- Il y a deux concepts principaux en sécurité
  - Authentification
    - Vérification de l'identité de la personne
    - En entreprise, on utilise généralement un serveur LDAP
  - Autorisations
    - Les droits sur l'application possédés par la personne
    - Généralement, ces droits sont stockés dans un serveur LDAP ou une base de données
- Spring Security permet de traiter ces deux concepts de manière indépendante : par exemple l'authentification en LDAP et les autorisations en base de données

Installation du filtre de Servlet Spring Security

- Spring Security est un filtre de Servlet, qui se configure donc via le fichier web.xml

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

- Il peut ainsi sécuriser toutes les requêtes vers l'application

Fonctionnement interne

- Spring Security va en fait déléguer la gestion des URLs demandées à une chaîne de filtres spécialisés
  - Ils vont valider si un utilisateur est authentifié ou non
  - Ils vont valider si un utilisateur a le droit d'accéder à la ressource ou non
  - Ils vont gérer l'authentification et la déconnexion de l'utilisateur
  - Ils vont gérer les cas d'erreurs et rediriger l'utilisateur vers les pages d'erreurs
- Nous utilisons ici la configuration fournie par défaut
  - Elle fonctionne parfaitement pour un site Web normal
  - Elle est entièrement paramétrable si besoin

Configuration minimale

- Spring Security se configure ensuite dans un fichier de configuration Spring classique, aidé par un namespace spécialisé

```
<http>
  <intercept-url pattern="/app/admin" access="ROLE_ADMIN"/>
  <intercept-url pattern="/app/**" access="ROLE_USER"/>
  <form-login login-processing-url="/login"
        login-page="/welcome"
        authentication-failure-url="/welcome?authentication=failure"
        default-target-url="/app/index"/>

  <logout logout-url="/logout"/>
</http>

<authentication-manager alias="authenticationManager">
   <authentication-provider user-service-ref="userDetailsService"/>
</authentication-manager>
```

La règles sur les URLs

- Les tags <intercept-url/> associent un ou plusieurs rôles nécessaires pour accéder à une URL donnée
- Ils se configurent avec des patterns «Ant»
- Ils sont ordonnés : le premier filtre à intercepter la requête est celui qui va la traiter
- Il faut donc les ordonner du plus spécialisé au moins spécialisé
  - Par exemple : un filtre sur «/**» sera mis à la fin
- Ils peuvent lister un ou plusieurs rôles de sécurité
  - Dans la configuration par défaut, il faut avoir l'un de ces rôles pour avoir accès à l'URL

La page de login

- Le tag <form-login/> détermine la page de login
  - Il détermine également l'URL qui va traiter l'authentification, la page d'erreur, etc...
- La page de login va alors pointer vers l'URL de traitement de l'authentification :

```
<form action="${context}/login" method="post">
  <label for="login">Login</label>
  <input type="text" name="j_username" id="login" />
  <label for="password">Passord</label>
  <input type="password" name="j_password" id="password" />
  <input type="submit" value="Login" />
</form>
```

Configuration avec un serveur LDAP

- Cette configuration utilise Spring LDAP, un autre sous-projet Spring qui facilite l'utilisation d'un serveur LDAP
  - Le namespace Spring Security permet de simplifier cette configuration

```
<ldap-server url="ldap://myldapserver:389/dc=exemple,dc=org" />

<authentication-manager>
  <ldap-authentication-provider
          user-dn-pattern="uid={0},ou=people"
          group-search-base="ou=groups" />
</authentication-manager>
```

Configuration avec une base de données relationnelle

- Voici la configuration, les requêtes SQL étant paramétrables :

```
<authentication-manager>
    <authentication-provider>
      <jdbc-user-service data-source-ref="dataSource"
                  users-by-username-query="..."
                  authorities-by-username-query="..."/>
    </authentication-provider>
</authentication-manager>
```

- Les requêtes par défaut sont :

```
SELECT username, password, enabled FROM users WHERE username = ?
```

```
SELECT username, authority FROM authorities WHERE username = ?
```

Configuration de test, directement dans la configuration Spring

- Cette configuration est uniquement utile en test, pour pouvoir facilement ajouter ou modifier des utilisateurs avec des droits spécifiques

```
<authentication-manager>
  <authentication-provider>
    <user-service>
      <user name="admin" password="password1" authorities="ROLE_USER, ROLE_ADMIN" />
      <user name="user" password="password1" authorities="ROLE_USER" />
    </user-service>
  </authentication-provider>
</authentication-manage>
```

Utilisation dans une page Web

- Spring Security s'intègre avec la sécurité Java EE standard : les méthodes «isUserInRole» et «getPrincipal» de l'API Servlet fonctionnent donc
  - Les Tag Lib ou frameworks (Struts) utilisant ces méthodes fonctionnent sans modification
- Spring Security propose également sa propre Tag Lib, avec une API plus élaborée :

```
<sec:authorize access="hasRole('ROLE_ADMIN')">
Seul un administrateur peut voir ce texte !
</sec:authorize>

<sec:authorize url="/admin">
Seule une personne pouvant voir l'URL "/admin" peut voir ce texte !
</sec:authorize>
```

Sécurisation de méthodes Java

- Spring Security peut également sécuriser l'accès à des méthodes Java
  - Il s'agit d'un nouvel Aspect (cf. le chapitre sur Spring AOP pour en voir les limitations)
- Le PointCut peut s'appliquer sur
  - Une annotation *@Secured*, spécifique à Spring Security
  - Une annotation *@RolesAllowed*, standardisée dans la JSR 250


```
<global-method-security secured-annotations="enabled" />
```

```
@Secured("ROLE_ADMIN")
public void uneMethodeSecurisee() {
   // code ne pouvant être exécuté que par un admin
}
```

Mise en place du «remember me»

- Spring Security permet de mettre en place un cookie dit «remember me»
  - C'est une fonctionnalité courante des sites Web, qui permet à un utilisateur de ne plus renseigner son mot de passe pendant une période donnée
- Il y a deux implémentations principales
  - Une basée sur un hash, simple à mettre en place, mais qui pose un problème de sécurité car le mot de passe est alors utilisé dans l'empreinte MD5 stockée dans le cookie (avec un salt)
  - Une basée sur une table en base de données, que nous recommandons car elle est plus sécurisée

```
<http>
    ...
    <remember-me data-source-ref="dataSource"/>
</http>
```

IPPON Enterprise Java Delivery

Conclusion sur Spring Security

- Spring Security est un framework de sécurité robuste et très répandu en entreprise
  - Il permet de sécuriser des applications Web, en particulier en donnant des règles d'accès aux URLs
  - Il peut également sécuriser des Beans Spring grâce à un Aspect
- Sa configuration de base est relativement simple, avec l'utilisation d'un namespace spécifique
- Il s'intègre avec de nombreuses solutions existantes : serveur LDAP, base de données, CAS, NTLM, OpenID...

Exercice 11
Spring Security

Félicitations, vous pouvez maintenant coder votre première application Spring !

ippon.fr blog.ippon.fr atomes.com

@ippontech contact@ippon.fr