

# Applications « Client /Serveur »



## Points faibles:

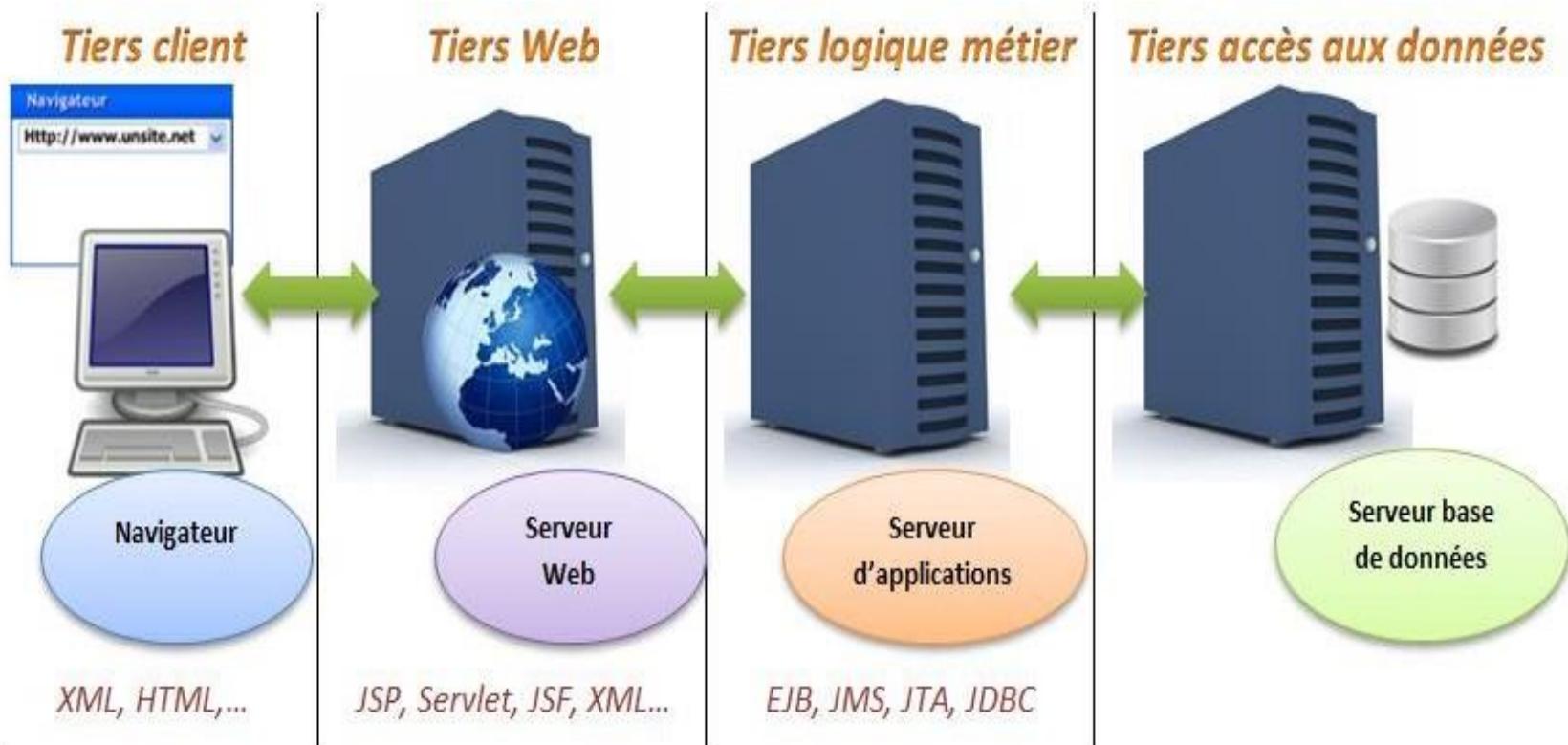
- Elle possède une sécurité limitée (gérée au niveau du SGBDR).
- Son déploiement demeure long et laborieux.
- Mise à jour couteuse.

# Applications 3-tiers



- **Point fort** : La séparation entre le client, l'application et le stockage, est le principal atout de ce modèle.
- **Point fiable** : Aucune séparation n'est faite au sein même de l'application, qui gère aussi bien la logique métier que la logique fonctionnelle ainsi que l'accès aux données

# Applications N-tiers



# Applications N-tiers

## ■ Avantage :

Dans la pratique, on travaille généralement avec un tiers permettant de regrouper la logique métier de l'entreprise. L'avantage de ce système, c'est que ce tiers peut être appelé par différentes applications clientes, et même par des applications classiques, de type fenêtrées, qui ne passent donc pas par le serveur Web

# C'est quoi Java EE?

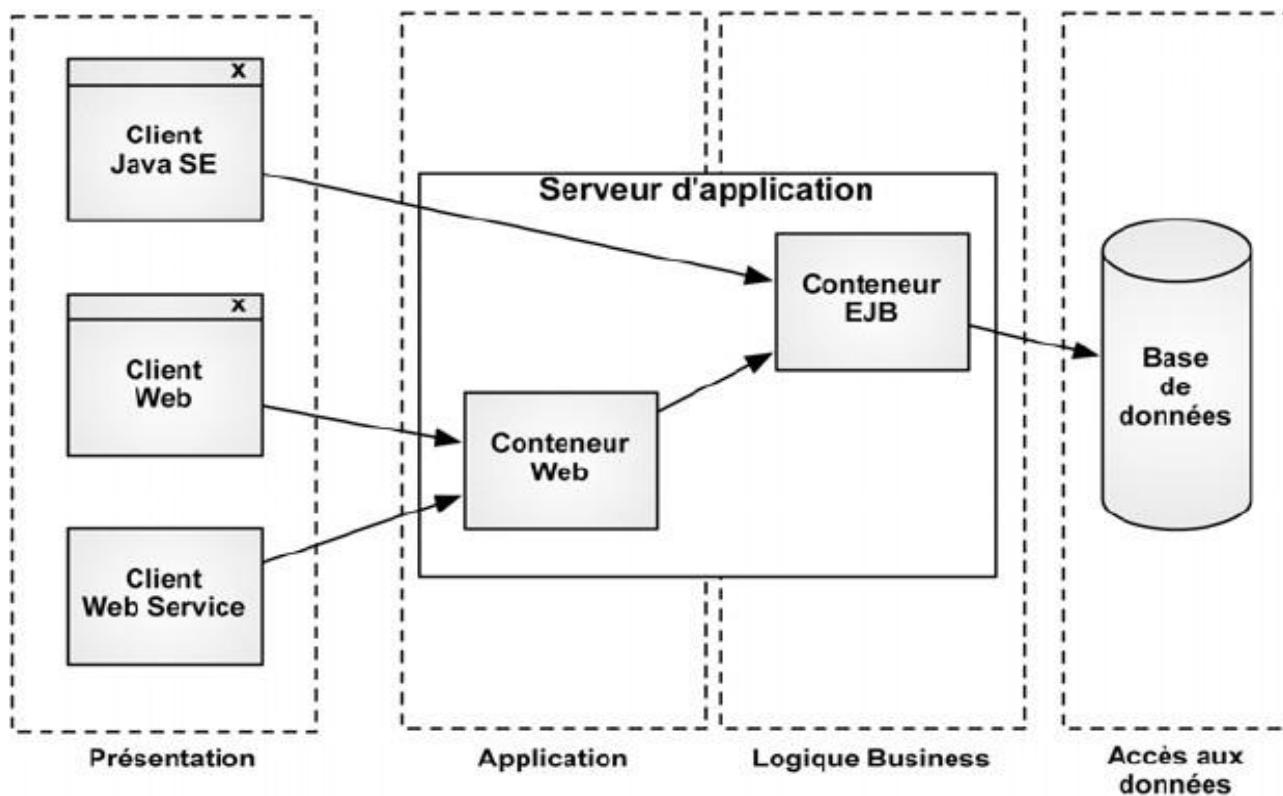
Java EE signifie Java Enterprise Edition, auparavant connu sous le nom de J2EE et actuellement connu sous le nom de Jakarta EE. Elle constitue une spécification pour la plate-forme Java destinée aux applications d'entreprise. Cette plate-forme est construite en se basant sur la plate-forme Java SE. La plate-forme Java EE fournit des APIs et un environnement d'exécution pour développer et exécuter des applications distribuées de grande échelle, multi-tiers, évolutives, fiables et sécurisées.

# C'est quoi Java EE?

Java EE est composée de deux parties essentielles:

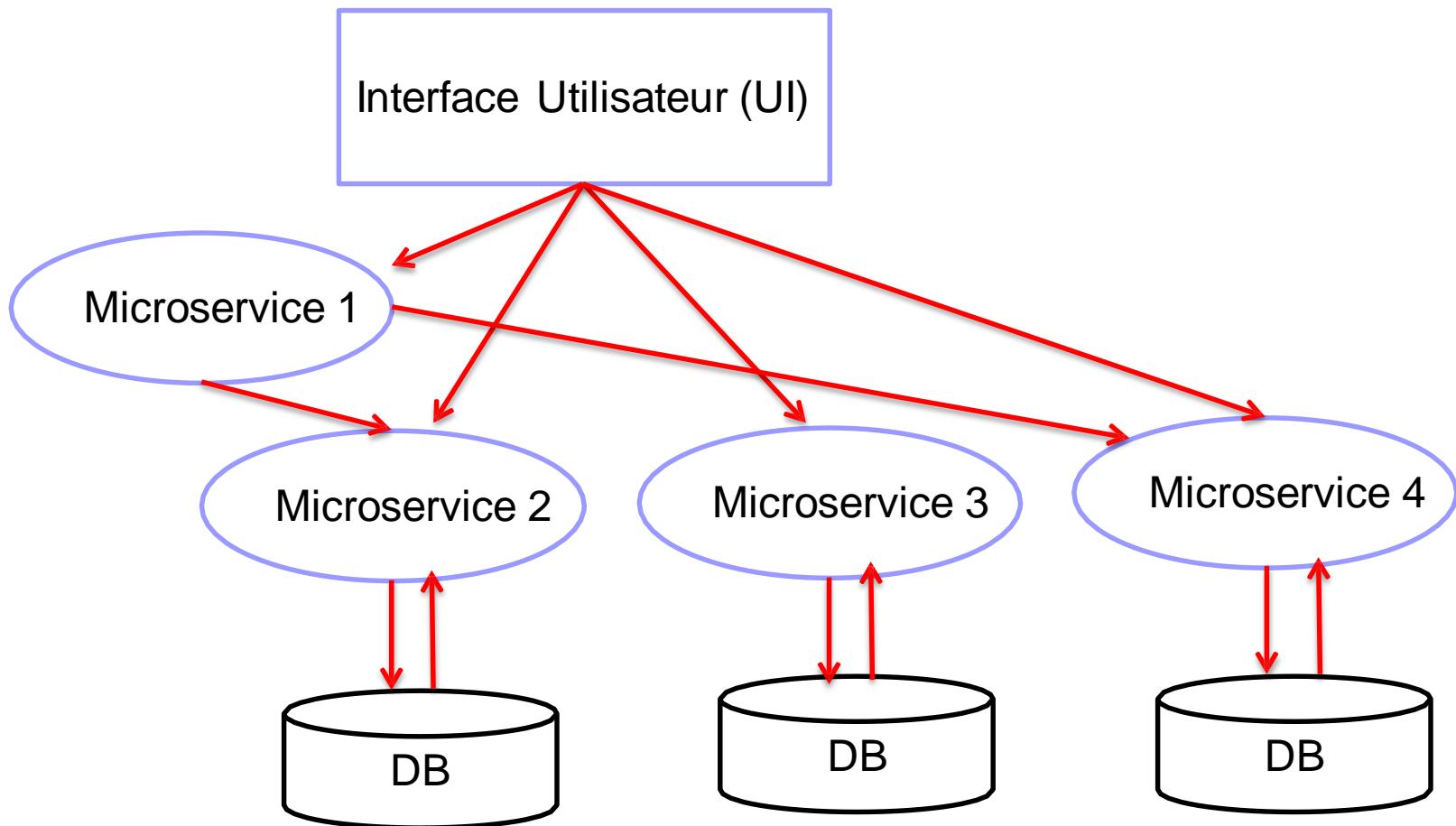
- un ensemble de spécifications pour une infrastructure dans laquelle s'exécutent les composants écrits en Java : un tel environnement se nomme serveur d'applications.
- un ensemble d'API qui peuvent être obtenues et utilisées séparément. Pour être utilisées, certaines nécessitent une implémentation de la part d'un fournisseur tiers

# Architecture d'une application Java EE standard

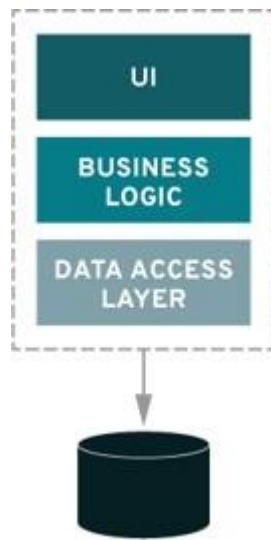


Architecture générale d'une application en couches (Java EE)

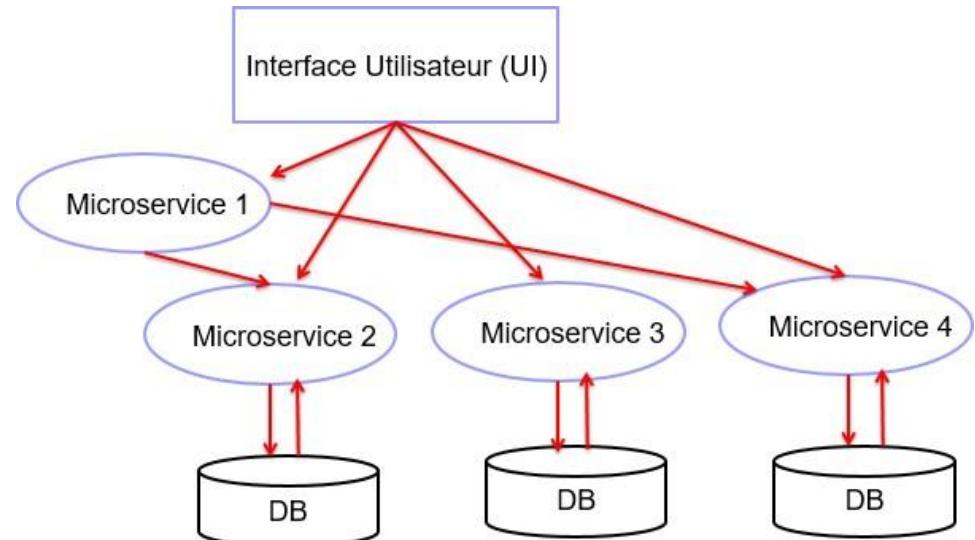
# Architecture microservices



# Architecture microservices contre architecture monolithique



Application monolithique

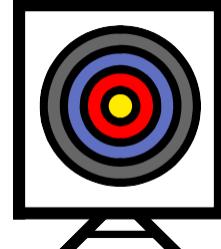


Application microservices



# **Servlets, Filtres, JSP et modèle MVC**

# Objectifs



- Comprendre le fonctionnement des applications JEE basées sur les Servlets
- Apprendre à écrire des Servlets
- Comprendre la gestion de la session
- Comprendre la gestion des cookies et leur utilisation
- Comprendre la notion de Filtre

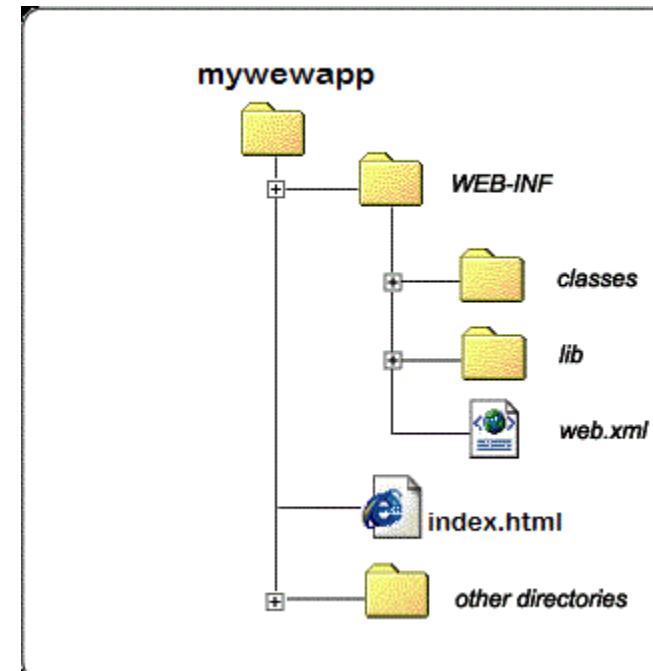
# Introduction

- Les servlets sont des programmes exécutés sur un serveur Web qui analysent les requêtes HTTP en provenance d'un navigateur Web, traitent la demande du client, puis fournissent une réponse adaptée au format HTML.
- Les servlets sont la base de la programmation Web Java EE. Toute la conception d'une application Web en Java repose sur des servlets.

# Descripteur de déploiement

## ■ Structure des dossiers d'une application Web J2EE

Le descripteur de déploiement d'une application web est un fichier nommé `web.xml` et il est situé dans le répertoire `WEB-INF` du document root (répertoire racine) de l'application web. Il contient les caractéristiques et les paramètres de l'application. Cela inclut la description des servlets utilisées, ou les différents paramètres d'initialisation.



# Invocation d'une servlet

## ■ Invocation d'une servlet depuis un navigateur

On peut appeler la Servlet avec son mapping url :

*http://serveur:port/ensah/HelloEnsaH*

## ■ Invocation d'une servlet depuis une page html

❖ Sur une balise **<a>** :

**<a href="http://serveur:port/ensah/HelloEnsaH">Cliquez ici</a>**

❖ Sur une balise **<form>** :

**<form action="http://serveur:port/ensah/HelloEnsaH"  
method="post">**

.....

**</form>**

# Paramètres d'initialisation

```
<servlet>

<servlet-name>TestConnexion</servlet-name>
    <servlet-class>com.ensah.tp.ihm.TestConnexion</servlet-class>

    <init-param>
        <param-name>jdbc.Drvier</param-name>
        <param-value>com.mysql.jdbc.Driver</param-value>
    </init-param>

    <init-param>
        <param-name>localisation</param-name>
        <param-value>jdbc:mysql://localhost/gestion</param-
value>
    </init-param>

</servlet>
```

# Exemple sur machine

- Ecrire une Servlet simple
- Ecrire un descripteur de déploiement
- Invoyer la servlet



*Le code sources des exemples de cours est disponible sur  
[https://github.com/boudaa/code\\_ensah/tree/master/ExemplesCoursJavaWeb](https://github.com/boudaa/code_ensah/tree/master/ExemplesCoursJavaWeb)*

# Les classes principales mises en jeu

- **Cookie** : Permet de gérer les cookies.
- **HttpServlet**: Fournit une classe abstraite à dériver pour créer une servlet http.
- **HttpServletRequest** : Dérive de ServletRequest pour fournir les informations des demandes pour les servlet HTTP.
- **HttpServletResponse** : Dérive de ServletResponse pour fournir les fonctionnalités spécifiques HTTP dans l'envoi d'une réponse.
- **HttpSession** : Fournit le moyen d'identifier un utilisateur, d'une demande de page à l'autre et de stocker des informations sur cet utilisateur.

# Cycle de vie d'une servlet

- Le conteneur de servlet gère le cycle de vie d'une servlet. L'instance est créée automatiquement par le conteneur Web.
- Le conteneur peut détruire la servlet à tout moment.
- Les méthodes des servlets sont *multiThreadées*.

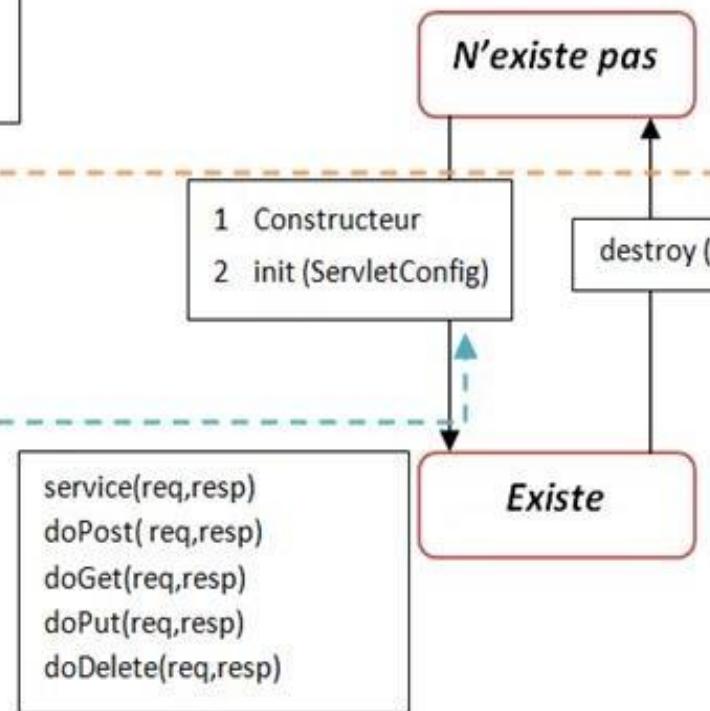
# Cycle de vie d'une servlet

## ■ Initialisation d'une Servlet :

Lors de l'appel au constructeur, le conteneur va aussi créer un objet *ServletConfig* contenant les paramètres d'initialisation de la Servlet. Cet objet contiendra aussi l'objet *ServletContext*.



1. Le conteneur lit et parse le fichier web.xml
2. Le conteneur crée un objet *ServletContext*



# Cycle de vie d'une servlet

- **La méthode init**

La méthode **init(ServletConfig)** est appelée une seule fois après la création de la servlet. Elle reçoit un paramètre de type **ServletConfig** (récupérable par la suite grâce à **getServletConfig()**) contenant les paramètres de configuration de la servlet.

La méthode **init(ServletConfig config)** qu'utilise le conteneur ne doit pas être redéfini, il est plus astucieux de redéfinir **init( )**, car cette méthode est appelée automatiquement par **init(ServletConfig)** et, en standard, ne fait rien. On peut y récupérer les paramètres de la servlet.

# Cycle de vie d'une servlet

- **La méthode service**

Chaque fois que le serveur reçoit une requête pour une servlet il crée un thread et y appelle la méthode service. Celle-ci vérifie le type de la requête http (GET, POST, PUT, etc ...) et appelle la méthode correspondante (doGet, doPost, doPut, etc.). Cette méthode n'est pas à redéfinir.

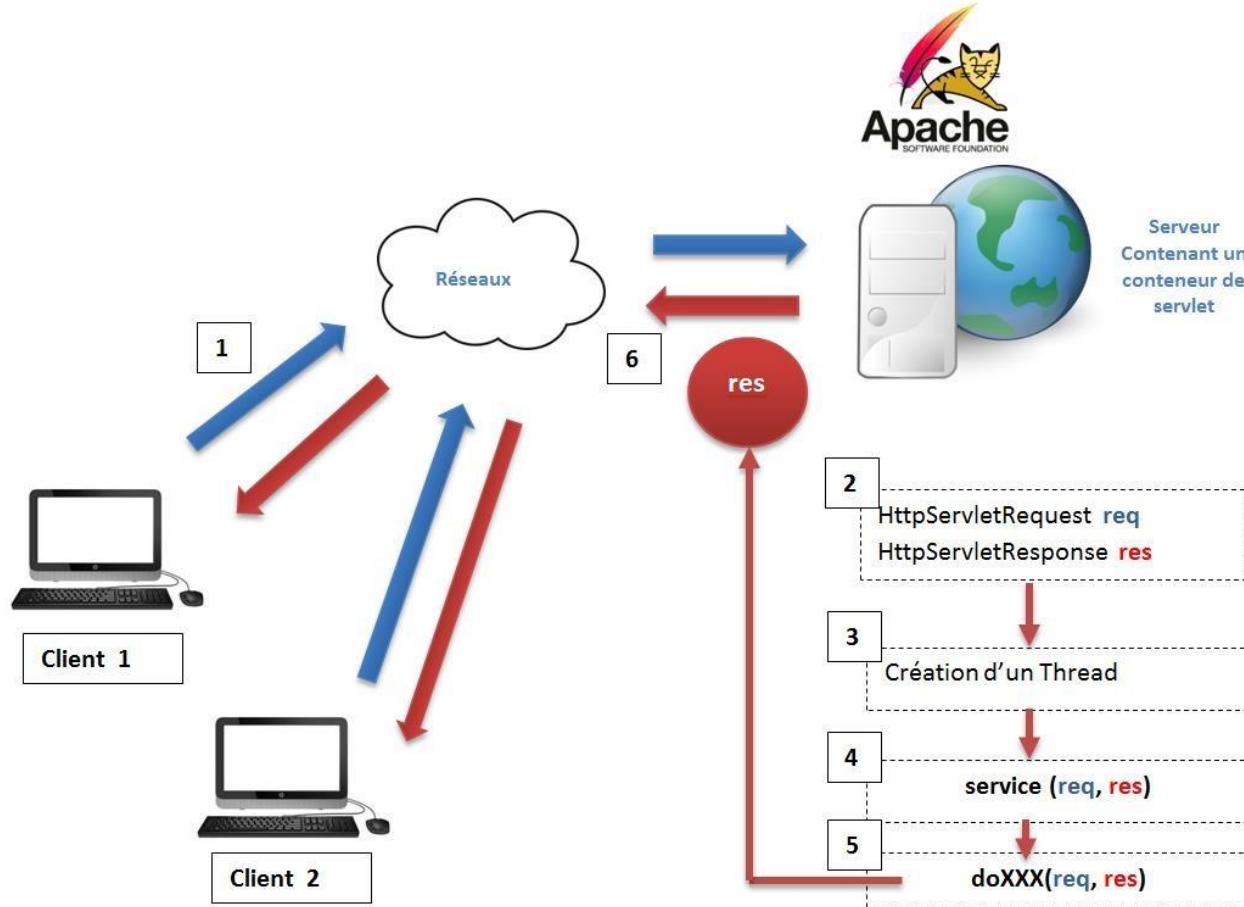
# Exemple sur machine

- Récupérer les paramètres d'initialisation d'une servlet dans la méthode `init()`



*Le code sources des exemples de cours est disponible sur  
[https://github.com/boudaa/code\\_ensah/tree/master/ExemplesCoursJavaWeb](https://github.com/boudaa/code_ensah/tree/master/ExemplesCoursJavaWeb)*

# Utilisation de la servlet



**Etape 1 :** Le client envoie une requête

**Etape 2 :** Le conteneur Web Crée les deux objets req et res de types HttpServletRequest et HttpServletResponse respectivement

**Etape 3 :** Création d'un Thread pour traiter la requête

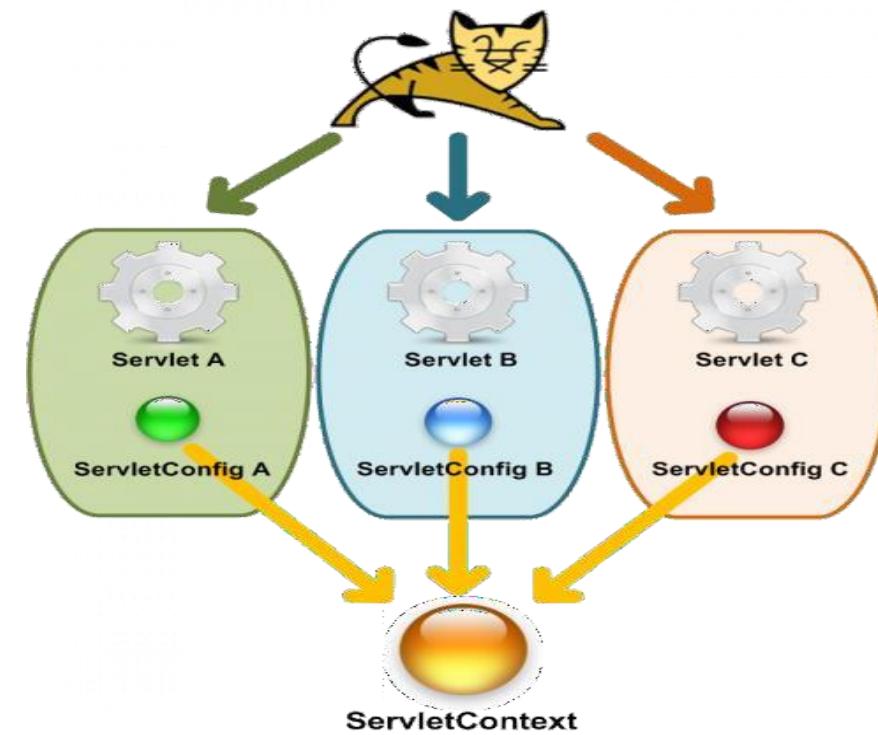
**Etape 4 :** Appel de la méthode service avec en paramètres les objets req et res

**Etape 5 :** appel de la méthode doGet, doPost, ou doXXX selon le type de la requête

**Etape 6 :** Le conteneur envoit la réponse au client

# contexte d'une application Web

- Un contexte constitue pour chaque servlet d'une même application une vue sur le fonctionnement de cette application.
- Le contexte est représenté par un objet ServletContext
- Il existe une seule instance de ServletContext par application



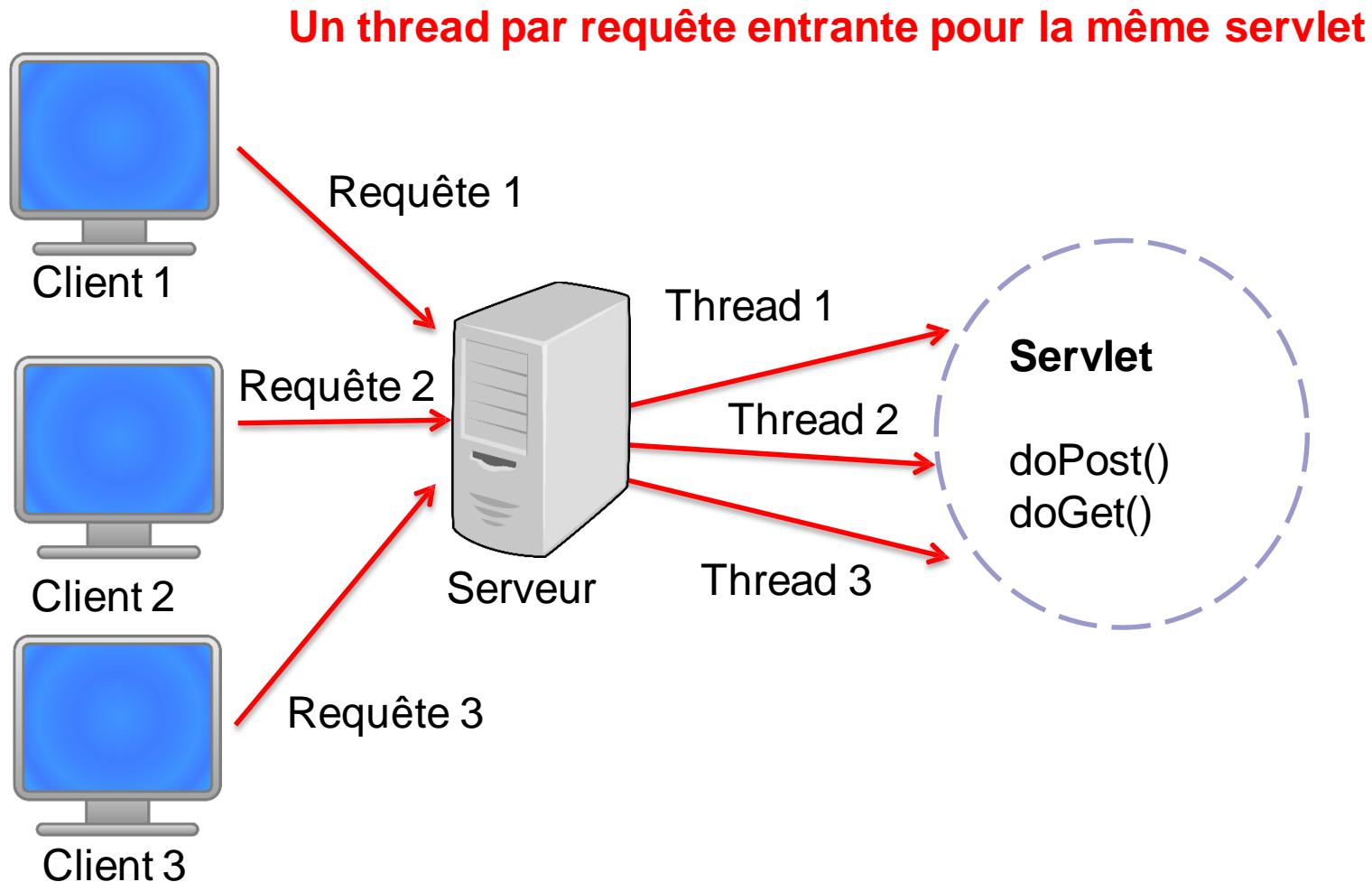
# Exemple sur machine

- Récupérer les paramètres d'initialisation du context dans une Servlet



*Le code sources des exemples de cours est disponible sur  
[https://github.com/boudaa/code\\_ensah/tree/master/ExemplesCoursJavaWeb](https://github.com/boudaa/code_ensah/tree/master/ExemplesCoursJavaWeb)*

# Précautions à prendre avec le multi-Threading



# Précautions à prendre avec le multi-Threading

1. Votre méthode doXX ne doit accéder à aucune variable membre, sauf si ces variables membres sont elles-mêmes thread-safe.
2. Votre méthode doXX ne doit pas réaffecter les variables membres, car cela peut affecter d'autres threads associés à d'autres requêtes. Si vous avez vraiment besoin de réaffecter une variable membre, assurez-vous que cela est fait à l'intérieur d'un bloc synchronisé.
3. Les règles 1 et 2 comptent également pour les variables statiques.
4. Les variables locales sont toujours thread-safe. Gardez cependant à l'esprit que l'objet vers lequel pointe une variable locale peut ne pas l'être. Si l'objet a été instancié à l'intérieur de la méthode, il n'y aura pas de problème. En revanche, une variable locale pointant vers un objet partagé peut toujours poser des problèmes. Ce n'est pas parce que vous affectez un objet partagé à une référence locale que cet objet devient automatiquement thread-safe.

# Récupération des paramètres de la requête

## ■ Traitement d'un GET

- ❖ Envoi des paramètres avec la méthode GET dans un lien HTML :

```
<a href =  
"http://localhost:8080/TP2/EnsaServlet?param1=ENSAH  
&param2=Alhoceima"> Cliquer ICI  
</a>
```

- ❖ Récupération des paramètres dans la Servlet :

```
String valparam1 = request.getParameter("param1");  
String valparam2 = request.getParameter("param2");
```

# Récupération des paramètres de la requête

## ■ Traitement d'un POST

On utilise directement les méthodes `getParameter(nameParameter)`. Avec `nameParameter` est le nom du champs du formulaire.

```
<form ... >
    <input name ="nom" type="text" />
    <input name ="prenom" type="text" />
</form >
```

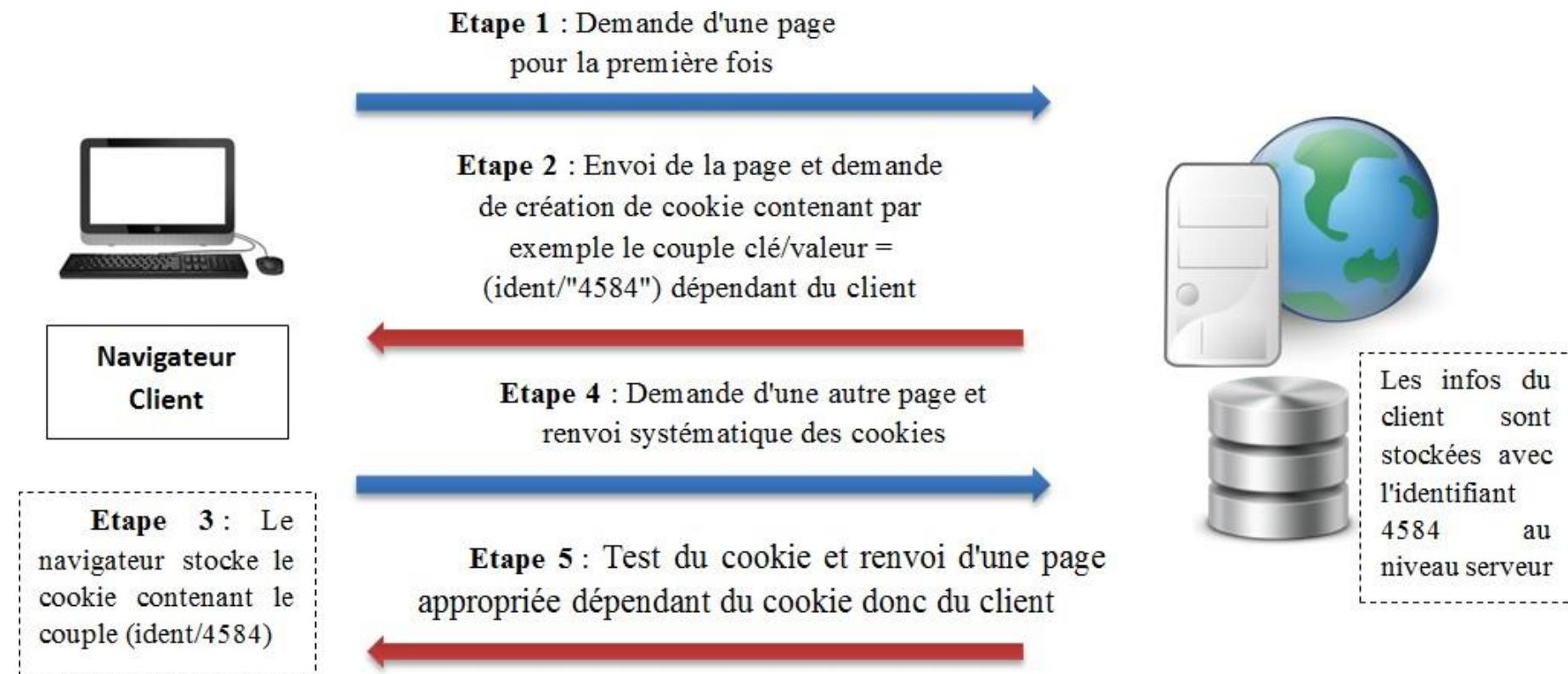
### ❖ Récupération des paramètres dans la Servlet :

```
String nom= request.getParameter("nom");
```

```
String prenom = request.getParameter(" prenom ");
```

# Gestion des cookies

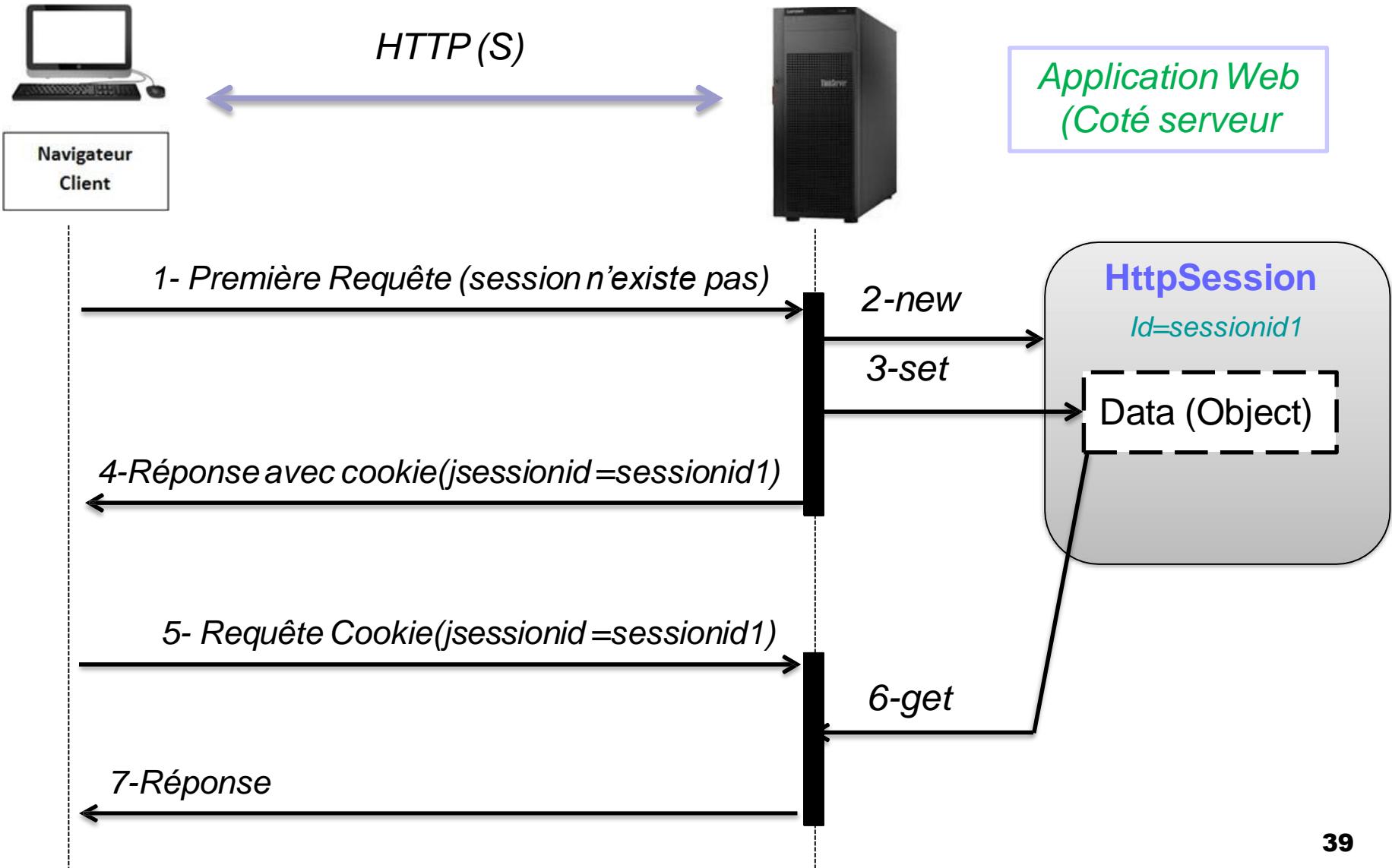
- ❖ C'est une information (couple nom/valeur) généralement écrite par un serveur sur le poste du client le consultant



# Gestion de la session

- ❖ Une session permet de suivre un utilisateur lors de sa navigation sur une application web, d'une ressource à une autre (d'une page en page par exemple). Elle peut être maintenue pendant un temps assez long, même de plusieurs jours, afin d'identifier un « utilisateur » qui reviendrait sur l'application web.
- ❖ La notion de session crée donc une notion de persistance, durant une certaine période fixée à l'avance. Au-delà d'une certaine durée d'activité, une session HTTP expire, et toutes les informations qui y sont attachées expirent.

# Gestion de la session

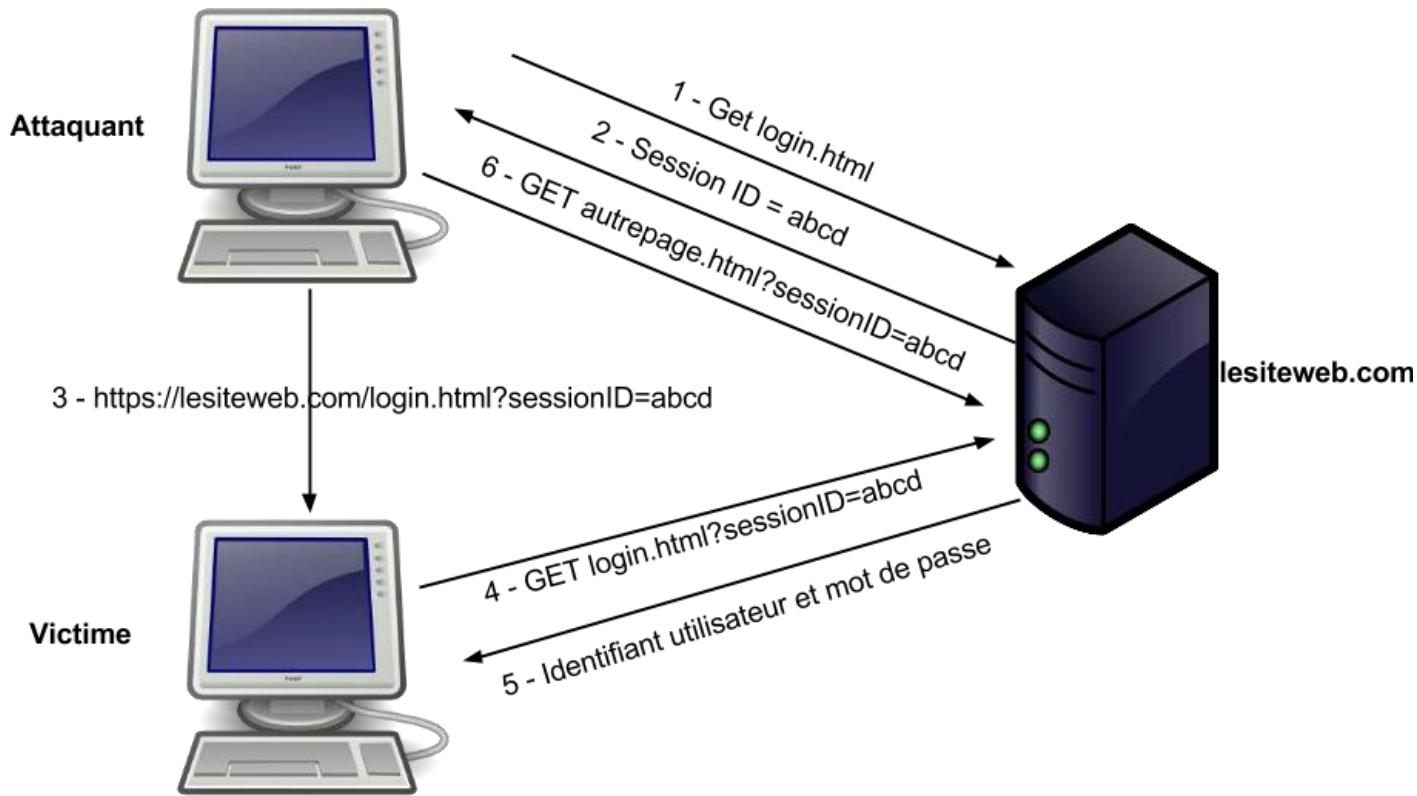


# Gestion de la session

- ❖ L'interface HttpSession permet de définir la notion de session, qui n'existe pas en HTTP.
- ❖ La gestion d'une session dans le standard Servlet peut se faire de deux manières. La manière la plus simple est de créer un cookie de session, envoyé au client, qu'il retournera dans l'en-tête HTTP à chaque requête. La valeur de ce cookie est un code de hachage, qui identifie un internaute de façon unique. Le serveur conserve la trace de ces cookies, et il les associe aux bons utilisateurs.
- ❖ Si le navigateur client refuse les cookies, alors ce code de hachage est ajouté à l'URL de requête dans le cas de requête par la méthode GET, et aux paramètres internes pour les requêtes de type POST.

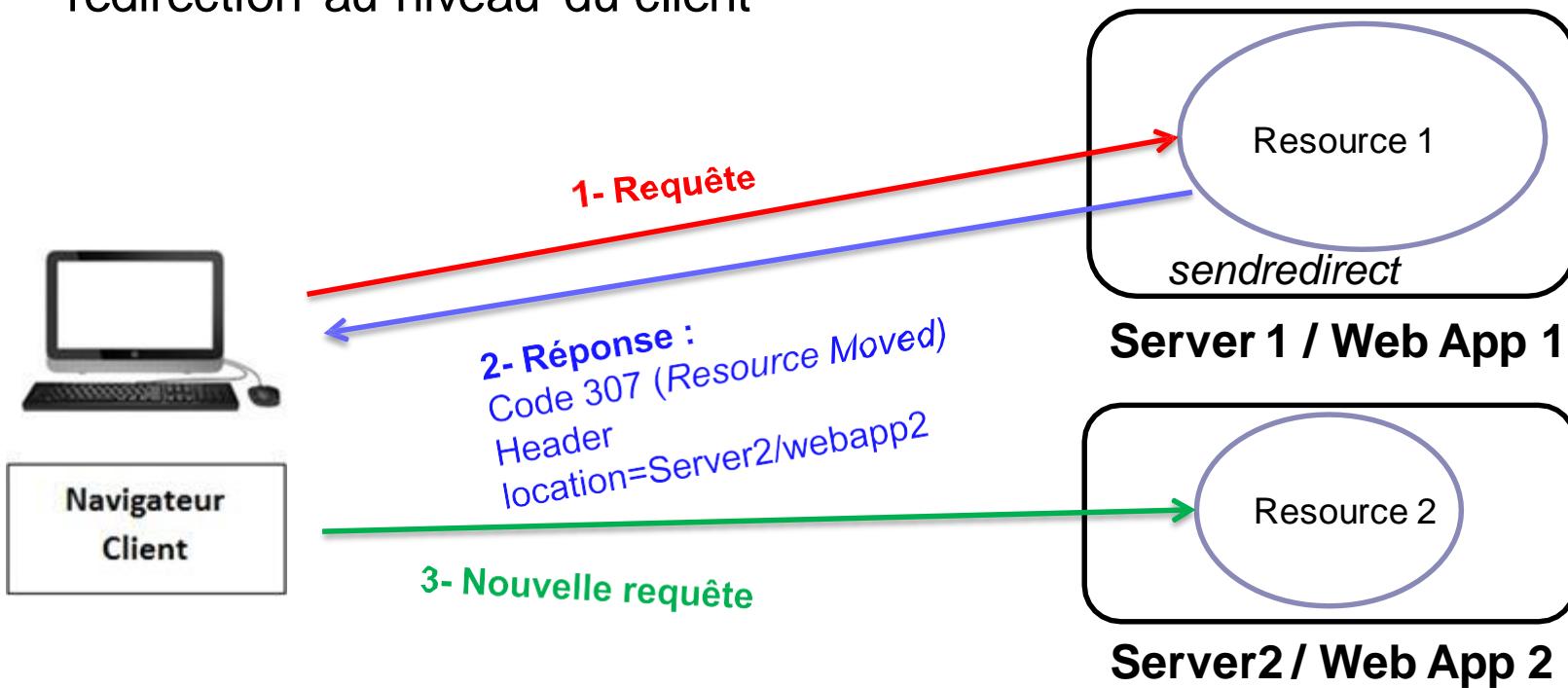
# Gestion de la session

## Exemple d'attaque des applications par fixation de session



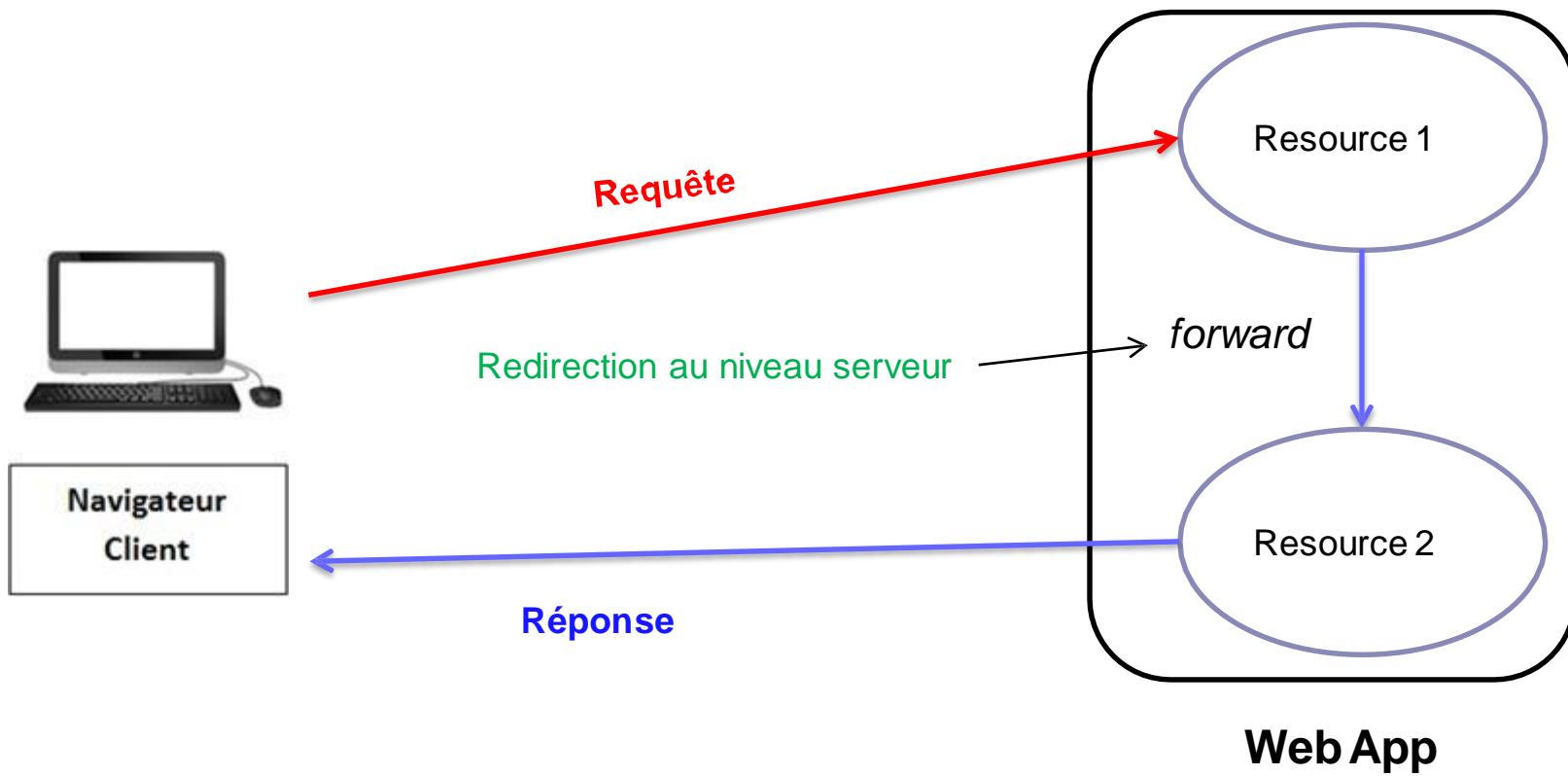
# Redirection au niveau client

- la méthode `sendRedirect(String urlCible)` (méthode de l'objet `HttpServletResponse`): est à utiliser pour rediriger vers une adresse située sur un autre serveur ou une autre webapp. On parle de redirection au niveau du client



# Redirection au niveau Serveur

- **les méthodes forward / include:** ces méthodes sont à utiliser pour faire une redirection au niveau du serveur



# Redirection au niveau serveur

- La servlet se contente de jouer le rôle de contrôleur et va donc chaîner vers d'autres servlets et/ou JSP
- Il faut commencer par récupérer un **RequestDispatcher** auprès du contexte de la servlet puis confier à cet objet le soin de chaîner (*en appelant forward*) ou d'inclure (*en appelant include*) le résultat d'une autre servlet ou JSP en fournissant son url. Les objets requête et réponse doivent, bien sûr, être transmis.

**Exemple :**

```
String urlCible = "/urlPattern";
RequestDispatcher dispatcher;
dispatcher= getServletContext( ).getRequestDispatcher(urlCible);
dispatcher.forward(req,resp);
```

# Redirection au niveau serveur

## ■ *Differences essentielles :*

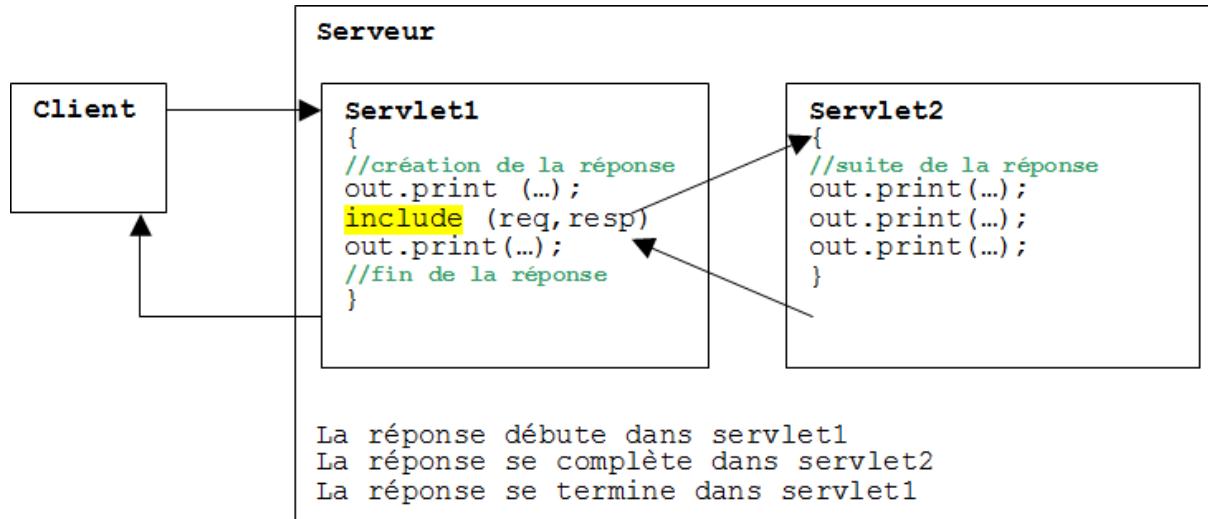
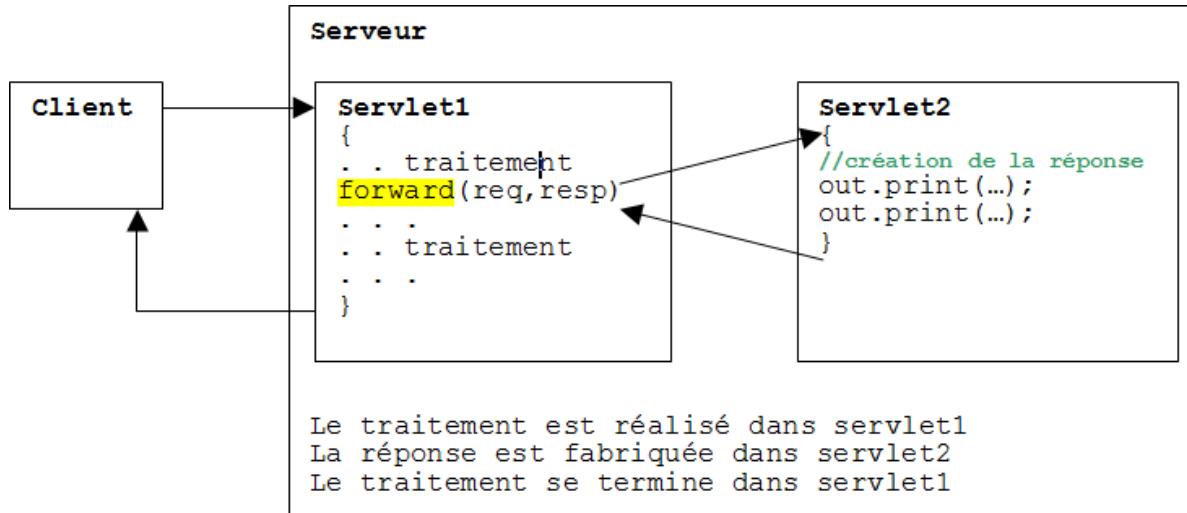
### ❖ ***Forward***

- ✓ Séparer le traitement de la requête du client (effectué sur la servlet appelante) de la génération de la réponse (effectuée sur la servlet appelée).
- ✓ Répartir le traitement de la requête sur plusieurs servlets.

### ❖ ***Include***

- ✓ Répartir la génération de la réponse sur plusieurs servlets.

# Redirection au niveau serveur



# Forward Vs SendRedirect

<b>Forward()</b>	<b>SendRedirect()</b>
La requête est transférée vers une autre ressource du même serveur	La requête est transférée vers une autre ressource vers un domaine ou un serveur différent
Le conteneur de servlet gère le process des redirections en interne, le client ne sera pas au courant des redirections effectuées au niveau serveur.  On ne peut pas voir la nouvelle cible au niveau de la barre d'adresse du navigateur par exemple.	Le conteneur redirige le client vers une autre ressource en lui indiquant cette ressource dans la réponse, le client est donc au courant de la redirection et c'est lui qui envoi la nouvelle requête vers la nouvelle cible.  On peut voir la nouvelle adresse au niveau de la barre d'adresse
On passe les objets request et response à la méthode forward de telle sorte que la nouvelle cible aura accès à ces mêmes objets.  La servlet source peut insérer des données dans l'objet request avec la méthode setAttribut, et la servlet cible peut les récupérer avec la méthode getAttribute	Dans ce cas la nouvelle cible n'a pas accès au anciens objets request et response

# Encodage des URL

- En cas d'utilisation de la réécriture d'URL (ce qui est le cas si le client ne supporte pas les cookies) il faut que, dès l'envoie au client d'une URL faisant référence à une application, ajouter explicitement les données de la session. Ceci se fait par les méthodes `encodeURL` et `encodeRedirectURL` de `HttpServletResponse`. Ces méthodes détectent si la réécriture d'url est employée et, dans ce cas, ajoutent les informations de session ; sinon, elles laissent l'url inchangé.
- Si on ne peut pas imposer l'usage des cookies de session aux clients, alors il faut utiliser systématiquement l'encodage d'URL.

# Encodage des URL

- Lorsque l'URL est intégrée à la page Web générée par la servlet

```
String originalURL ="/gestionsession";
String encodeURL= response.encodeURL(originalUrl);
out.println("<form action=\""+encodeURL+ "\" method=\"post\">");
```

- Lorsque on veut imposer une redirection au niveau du client par un *sendRedirect* et que la page destination participe à la session (ce qui en principe n'est pas le cas).

```
String originalURL ="/identification";
String encodeURL= response.encodeRedirectURL(originalUrl);
response.sendRedirect(encodeURL);
```

# Les pages JSP

Les pages JSP (Java Server Pages) sont une autre façon d'écrire des applications serveurs web. En fait ces pages JSP sont traduites en servlets avant d'être exécutées et on retrouve alors la technologie des servlets. Les pages JSP permettent de mieux mettre en relief la structure des pages HTML générées.

# Portée des objets

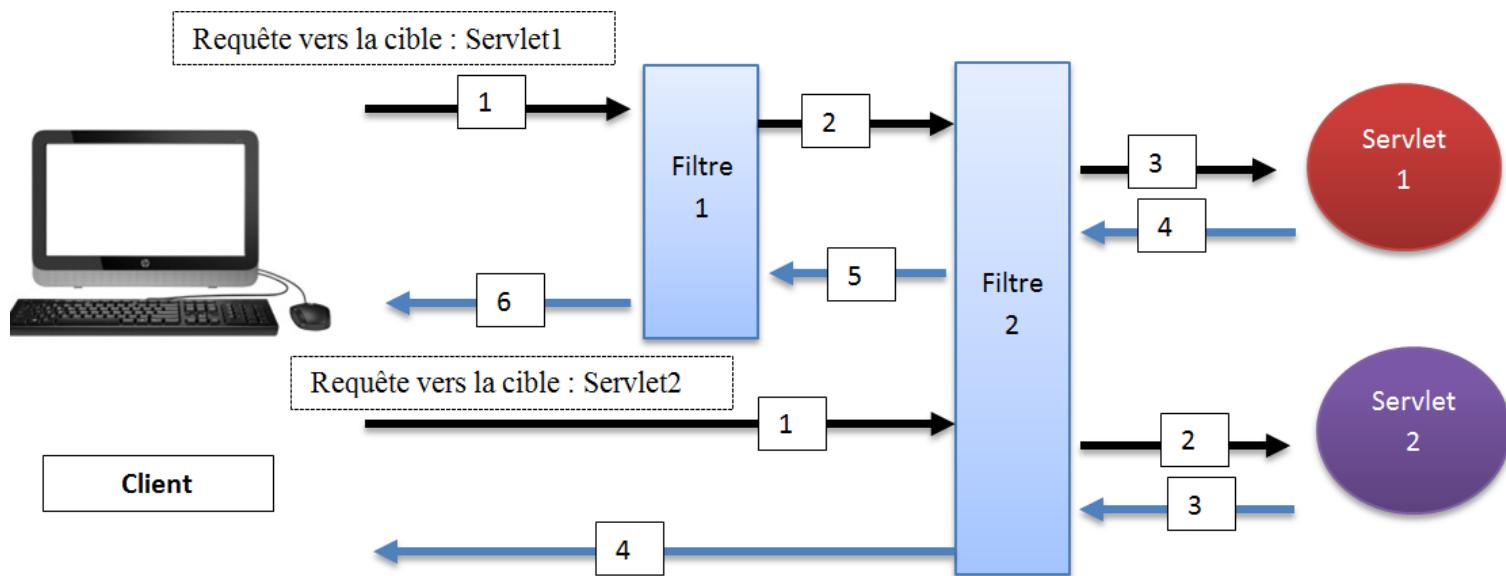
- **Page (JSP seulement)** : les objets dans cette portée sont uniquement accessibles dans la page JSP en question ;
- **Requête (Request)**: les objets dans cette portée sont uniquement accessibles durant l'existence de la requête en cours ;
- **Session** : les objets dans cette portée sont accessibles durant l'existence de la session en cours ;
- **Application** : les objets dans cette portée sont accessibles durant toute l'existence de l'application.

# Filtre dans l'API Servlet

**Un filtre peut faire les choses suivantes :**

- Intercepter une invocation de servlet avant qu'elle ne soit appelée
- Examiner une requête avant l'appel d'une servlet ;
- Intercepter l'invocation d'une servlet après qu'elle a été appelée
- Modifier les en-têtes et les données de la requête en fournissant une version personnalisée de l'objet de requête (l'objet de type `HttpServletRequest`) qui enveloppe la requête réelle.
- Modifier les en-têtes et les données de réponse en fournissant une version personnalisée de l'objet de réponse (l'objet de type `HttpServletResponse`) qui enveloppe la réponse réelle.

# Filtre dans l'API Servlet



**En Anglais :** Filters are components that can be used to pre process the request and post process the response. Based on the Interceptor Design Pattern, filters are useful in separating the **cross cutting concerns** like Security, logging, ...

You can map a filter to one or more Web resources, and you can map more than one filter to a Web resource.

# Filtre dans l'API Servlet

Dans la pratique les filtre sont utilisés généralement pour traiter les préoccupations transversales (*the cross cutting concerns*), ainsi parmi leurs utilisations classiques on trouve :

- l'authentification,
- la journalisation (en anglais logging),
- La conversion d'image,
- Upload des fichiers,
- La compression et le chiffrement de données,
- La conversion de données,
- Ajout systématique d'en-têtes et de pieds de pages
- etc.

# Modèle MVC

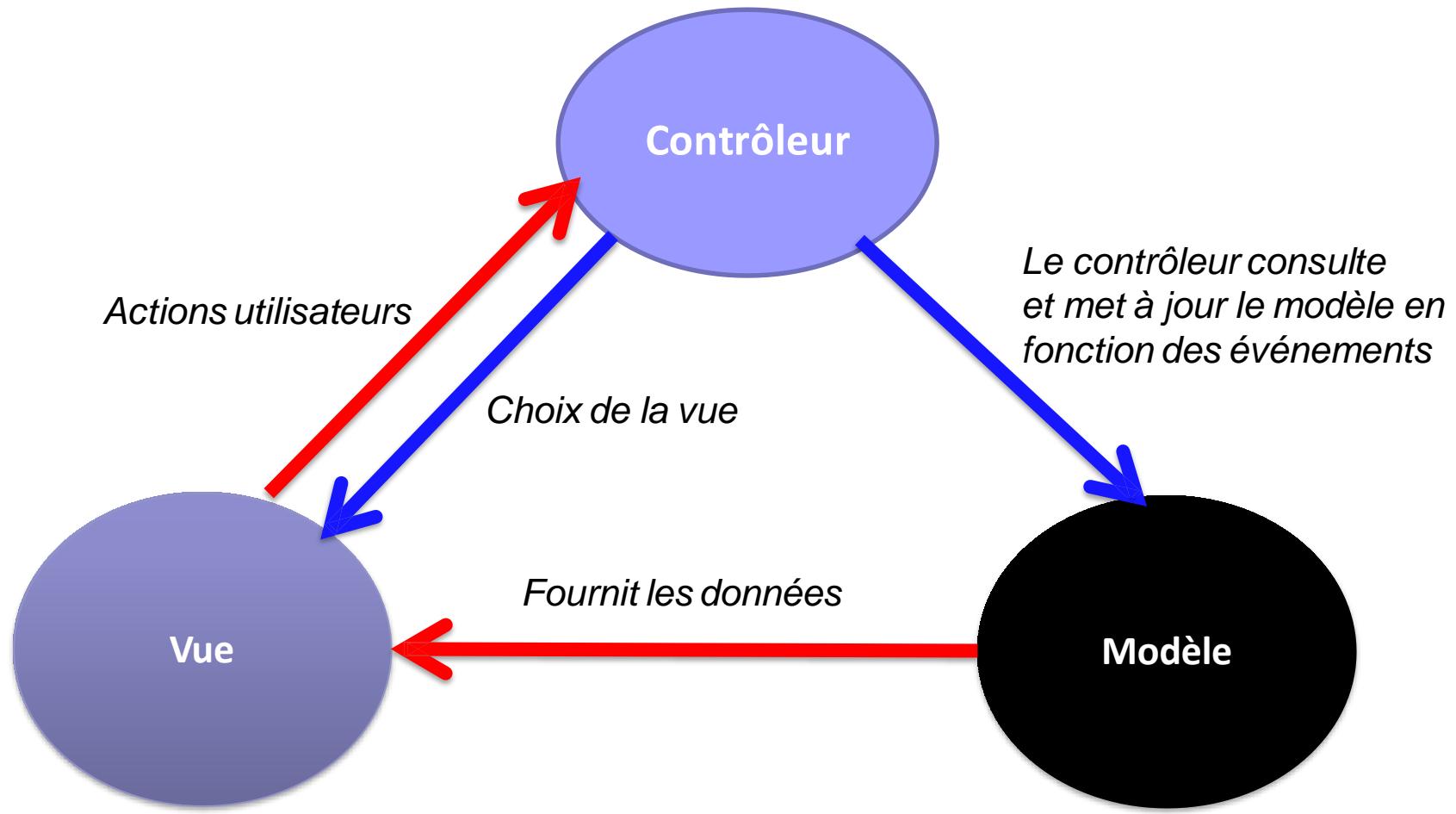
L'architecture (**Modèle ou paradigme**) **MVC** (**Model-View-Controller**) est un modèle de conception (Design Pattern) classique qui a été introduit avec le langage Smalltalk; son principe est relativement simple, on divise le système interactif en trois parties distinctes :

**le modèle (*Model*)** qui offre l'accès et permet la gestion des données (état du système) : il conserve toutes les données relatives à l'application (sous quelque forme que ce soit : base de données, fichiers...) et contient la logique métier de l'application.

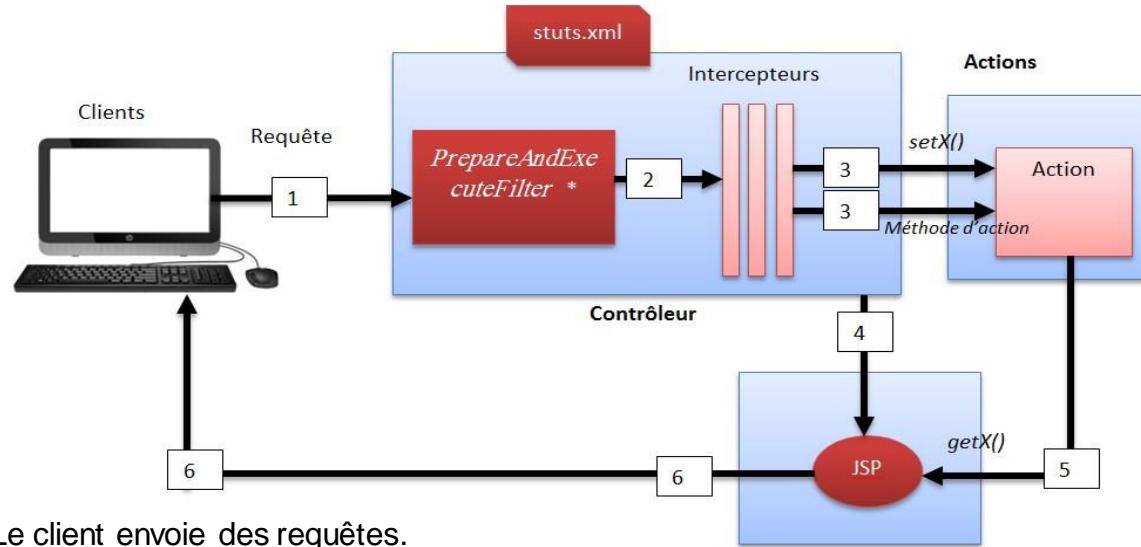
**La vue (*View*)** qui a pour tâche l'affichage des informations (visualisation) et qui participe à la détection de certaines actions de l'utilisateur

**Le contrôleur (*Controller*)** qui est chargé de réagir aux actions de l'utilisateur (clavier, souris) et à d'autres événements internes et externes

# Interactions du modèle MVC

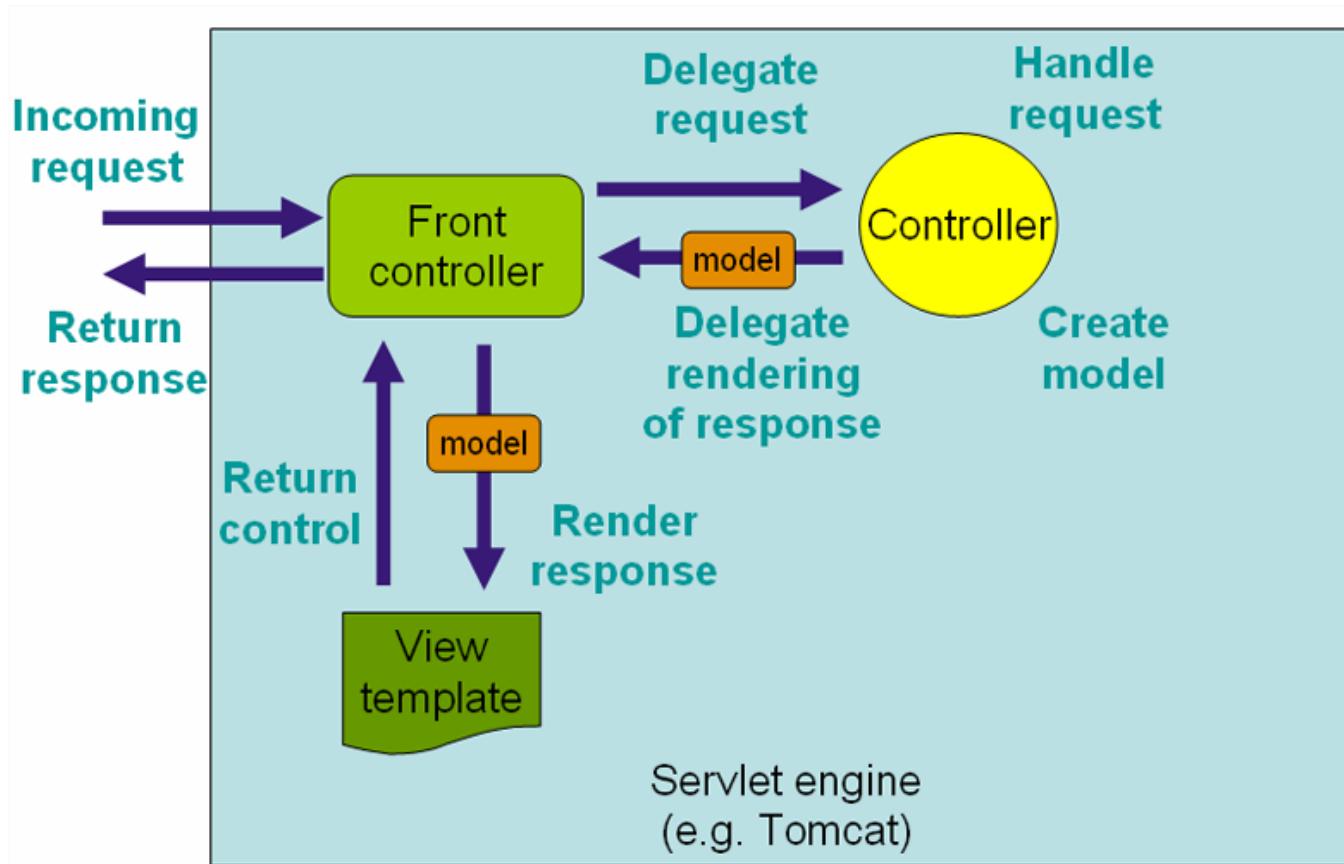


# Struts 2 et Modèle MVC



- 1) Le client envoie des requêtes.
- 2) Le conteneur de servlet reçoit sa requête.
- 3) Le conteneur de servlets invoque le filtre *FilterDispatcher* (ou *PrepareAndExecuteFilter* dans les versions récentes du Framework), ce filtre détermine l'action associée à la requête du client (en consultant la configuration des associations url/action dans le fichier struts.xml).
- 4) Chaque intercepteur associé à l'action est déclenché. L'un de ces intercepteurs est chargé d'assigner automatiquement les valeurs reçues dans la requête aux propriétés de la classe d'action, en fonction des noms des variables. Les intercepteurs (qui sont similaires conceptuellement aux filtres) sont utilisés généralement pour traiter les problématiques transversales comme la journalisation (Logging), validation des données, upload de fichiers, remédier au problème de soumission double des formulaires
- 5) L'action est exécutée et le résultat est générée par l'action. La vue à afficher est sélectionnée en accord avec le fichier de configuration struts.xml. La classe d'action transmet les données nécessaires à la vue.
- 6) La sortie d'action est rendue dans la vue (JSP, Velocity, etc) et le résultat est renvoyé à l'utilisateur.

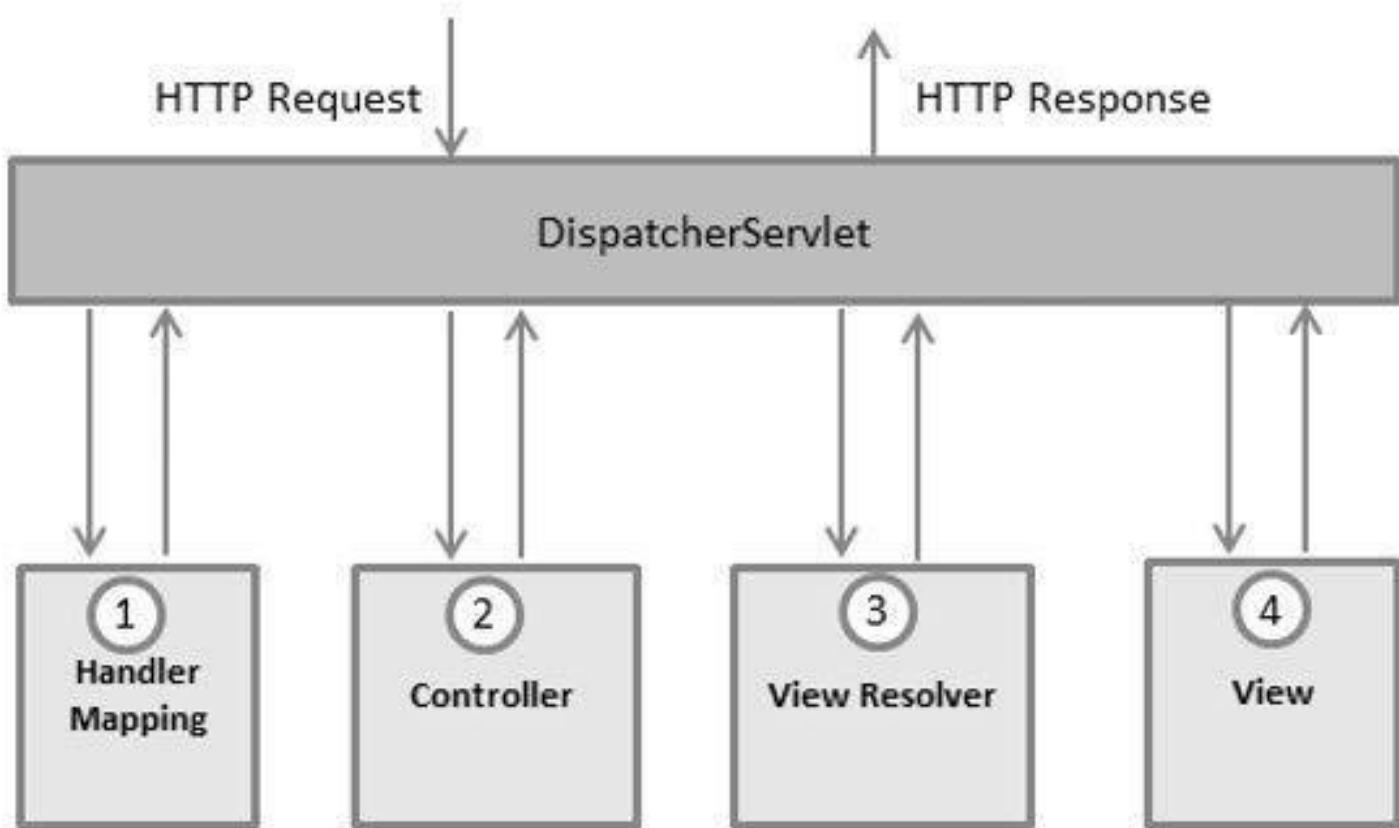
# Le modèle MVC de Spring MVC



Référence :

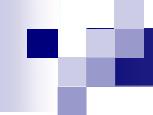
<https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html>

# Le modèle MVC de Spring MVC



Référence :

<https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html>



# **Injection de dépendances et Framework Spring**

# Framework Spring

## C'est quoi Spring ?

- Un Framework très populaire pour la construction des applications Java
- Initialement a été proposé comme une solution légère et alternative aux EJBs de J2EE.
- Il fournit le socle technique d'une application Java et un ensemble important de composants logiciels pour faciliter le développement des applications complexes et assurer des services techniques transverses (IoC, Security, logging, ...)
- Spring suit un développement léger (*Lightweight development*) avec Java POJOs (*Plain Old Java Objects*).

# Framework Spring

## Pourquoi Spring ?

- Les premières versions de EJB était très compliquées à utiliser et difficiles à déployer.
- Rod Johnson (le créateur de Spring) a donc écrit le livre "*J2EE Development without EJB* ».
- Avis personnel : «*Actuellement, les deux technologies sont très proches de point de vue fonctionnalités et simplicité mais les EJBs ont une mauvaise réputation à cause de leur historique. Les deux technologies existent toujours sur le marché. Maîtriser les deux c'est l'idéal mais si il faut choisir Spring est le choix prioritaire* ». 66

# POJO, JavaBean et Spring Bean

- **POJO est un objet Java ordinaire** càd il n'est contraint par aucune restriction autres que les spécification du langage Java i.e. a POJO ne doit pas étendre des classes ou implémenter des interfaces spécifiques à une technologie ou à framework .
- Un JavaBean est un POJO implémentant Serializable, ayant un constructeur public sans arguments et offre l'accès aux propriétés avec des getters/setters suivant une certaine convention de nommage.
- Spring Bean est tout objet géré par Spring. Plus spécifiquement, il s'agit d'un objet qui est instancié, configuré et géré par un conteneur Spring Framework.

# Couplage faible/ Couplage fort

## Couplage fort

- Quand une classe A dépend directement d'une classe B, on dit que la classe A est fortement couplée à la classe B.
- La classe A ne peut être compilée qu'en présence du code de la classe B.
- Si une nouvelle version de la classe B (soit B2), est créée et que A doit dépendre de B2, on est obligé de modifier dans la classe A.

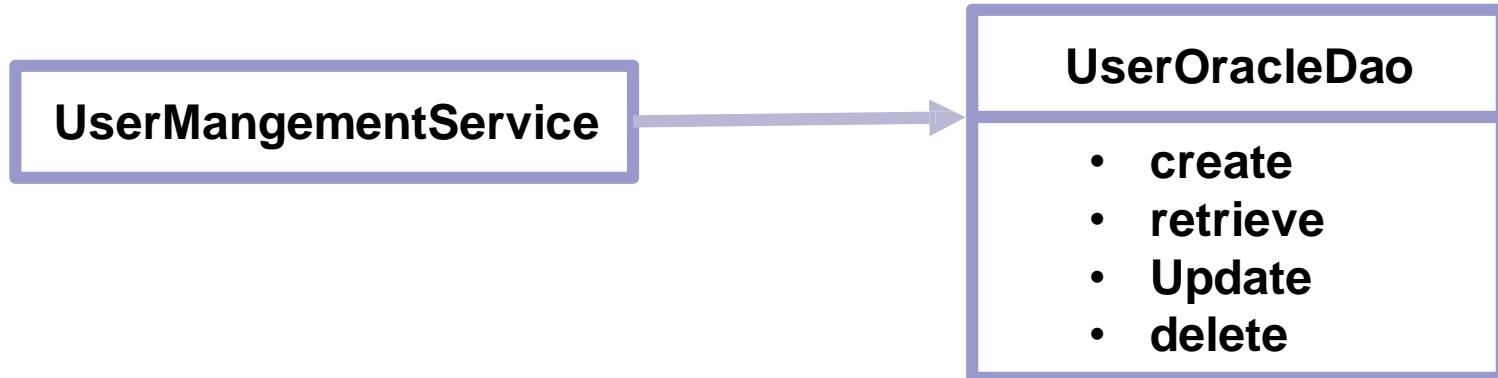


- En général le couplage fort réduit la flexibilité et la réutilisation du code

# Couplage faible/ Couplage fort

## Couplage fort

### ■ Exemple

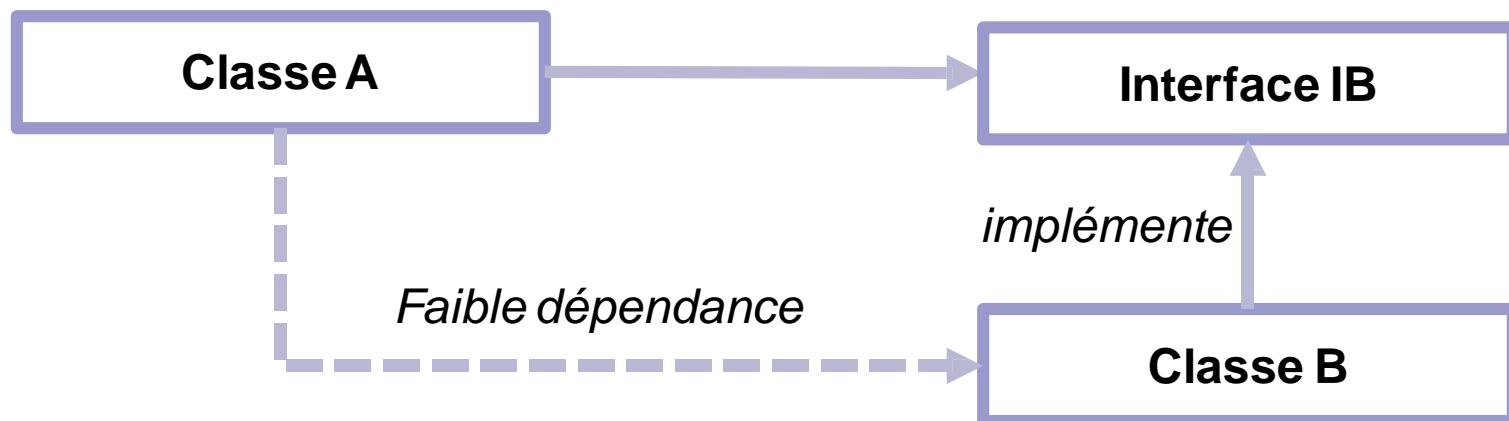


```
class UserMangementService {  
    ....  
    UserOracleDao dao = new UserOracleDao();  
}
```

# Couplage faible/ Couplage fort

## Couplage faible

- Le couplage faible permet de réduire les interdépendances entre les composants d'un système dans le but de réduire le risque que les changements dans un composant nécessitent des changements dans tout autre composant.
- Le couplage faible est un concept destiné à augmenter la flexibilité du système, à le rendre plus maintenable et à rendre l'ensemble du système plus stable.



# Couplage faible/ Couplage fort

## Couplage faible

### ■ Exemple

```
class UserMangementService {  
    ...  
    Dao dao ;  
    UserMangementService (Dao pDao) {  
        this.dao = pDao;  
    }  
}
```

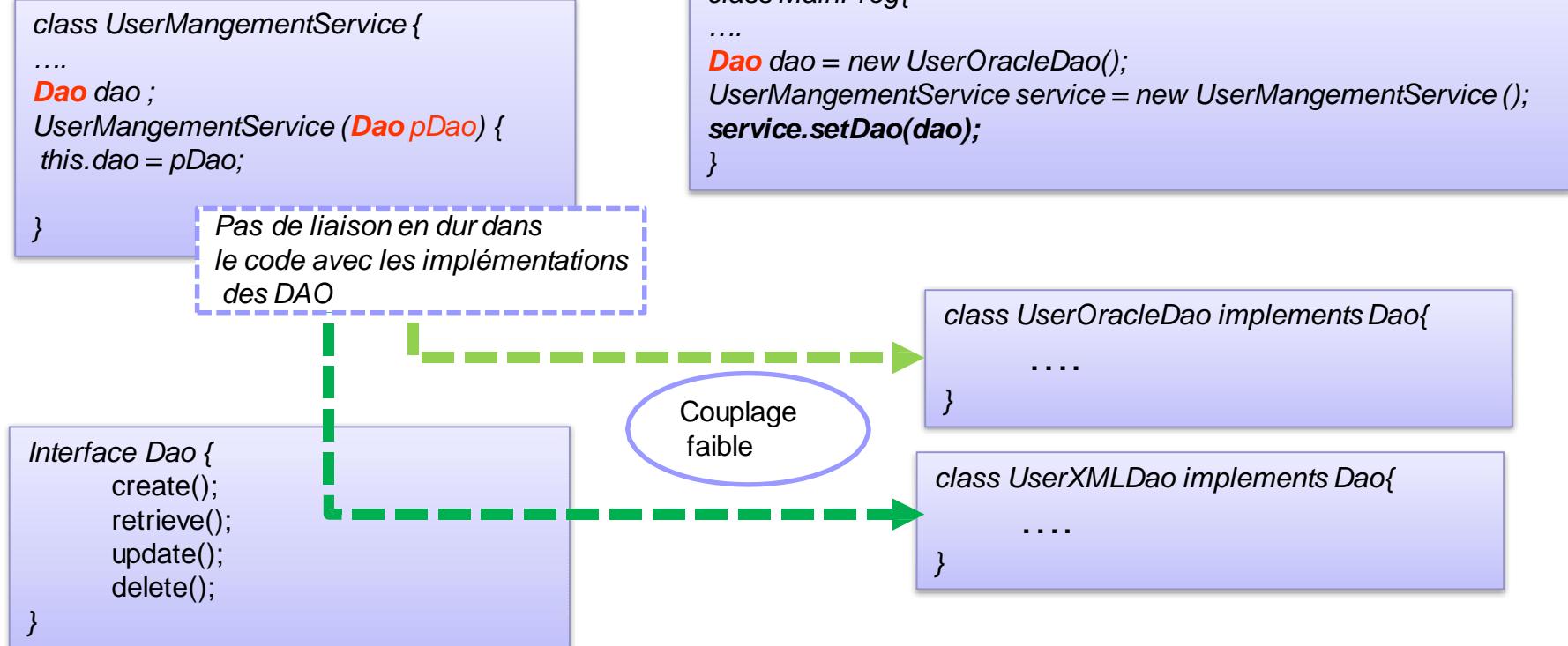
```
Interface Dao {  
    create();  
    retrieve();  
    update();  
    delete();  
}
```

```
class UserOracleDao implements Dao{  
    ...  
}
```

```
class UserXMLDao implements Dao{  
    ...  
}
```

# Couplage faible/ Couplage fort

## Couplage faible et injection de dépendances



# Couplage faible/ Couplage fort

## Injection de dépendance avec une Factory

### ■ Exemple

```
class UserMangementService {  
    ....  
    Dao dao;  
    UserMangementService (Dao pDao) {  
        this.dao = pDao;  
    }  
}
```

*3- Injecter la dépendance via le constructeur*

*Pas de liaison en dur dans le code avec les implémentations des DAO*

```
Interface Dao {  
    create();  
    retrieve();  
    update();  
    delete();  
}
```

### *Fichier de configuration*

UserMangementService *Dépend de* UserOracleDao

Factory

1- Lecture du fichier de configuration des dépendances

2- instancier

```
class UserOracleDao implements Dao{  
    ....  
}
```

```
class UserXMLDao implements Dao{  
    ....  
}
```

**Couplage faible**

# Injection de dépendance

- « L'injection de dépendances (dependency injection en anglais) est un mécanisme qui permet d'implémenter le principe de l'inversion de contrôle. Il consiste à créer **dynamiquement (injecter)** les dépendances entre les différents objets en s'appuyant sur une description (fichier de configuration ou métadonnées) ou de manière programmatique. Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement à l'exécution. »

([https://fr.wikipedia.org/wiki/Injection\\_de\\_d%C3%A9pendances](https://fr.wikipedia.org/wiki/Injection_de_d%C3%A9pendances))

# Injection de dépendance

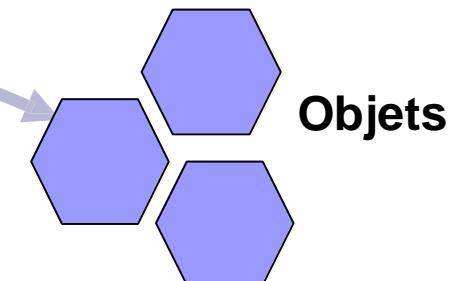
- L'injection des dépendances, ou l'inversion de contrôle est un concept qui intervient généralement au début de l'exécution de l'application.
- Spring commence par lire un fichier de configuration XML (ou les métadonnées) qui déclare quelles sont différentes classes à instancier et d'assurer les dépendances entre les différentes instances.
- Quand on a besoin d'intégrer une nouvelle implémentation à une application, il suffirait de la déclarer dans la configuration.

1- Lire les dépendances depuis le fichier de configuration ou Metadata



Spring IoC  
(Conteneur Spring)

2- Création des objets et injection des dépendances



Les fonctions principales du conteneur Spring :

- Création et gestion du cycle de vie des objets
- Injection des dépendances entre les objets

# Injection de dépendance

## Méthodes de configuration des bean avec Spring

- Dans la version actuelle de Spring (Version 5) il existe trois méthodes de configuration:
  - **Configuration XML** (*Legacy, mais plusieurs applications legacy utilisent XML, donc plusieurs projets de maintenance nécessitent des connaissances sur ce mode Legacy*)
  - **Annotations Java** (moderne)
  - **Configuration par un code Java** (moderne)

# Injection de dépendance

## Configuration XML avec le fichier applicationContext.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.xsd">

    <!-- bean definitions here -->

    <bean id="utilisateurDao" class="com.dao.impl.UtilisateurDaoImpl">
        <property name="sessionFactory" ref="sessionFactory"></property>
    </bean>
    <bean id="roleDao" class="com.dao.impl.RoleDaoImpl">
        <property name="sessionFactory" ref="sessionFactory"></property>
    </bean>
    <bean id="utilisateurService" class="com.services.impl.UtilisateurServiceImpl">
        <property name="userDao" ref="utilisateurDao"></property>
        <property name="roleDao" ref="roleDao"></property>
    </bean>
</beans>
```

## Récupérer une instance d'un bean depuis le conteneur de Spring

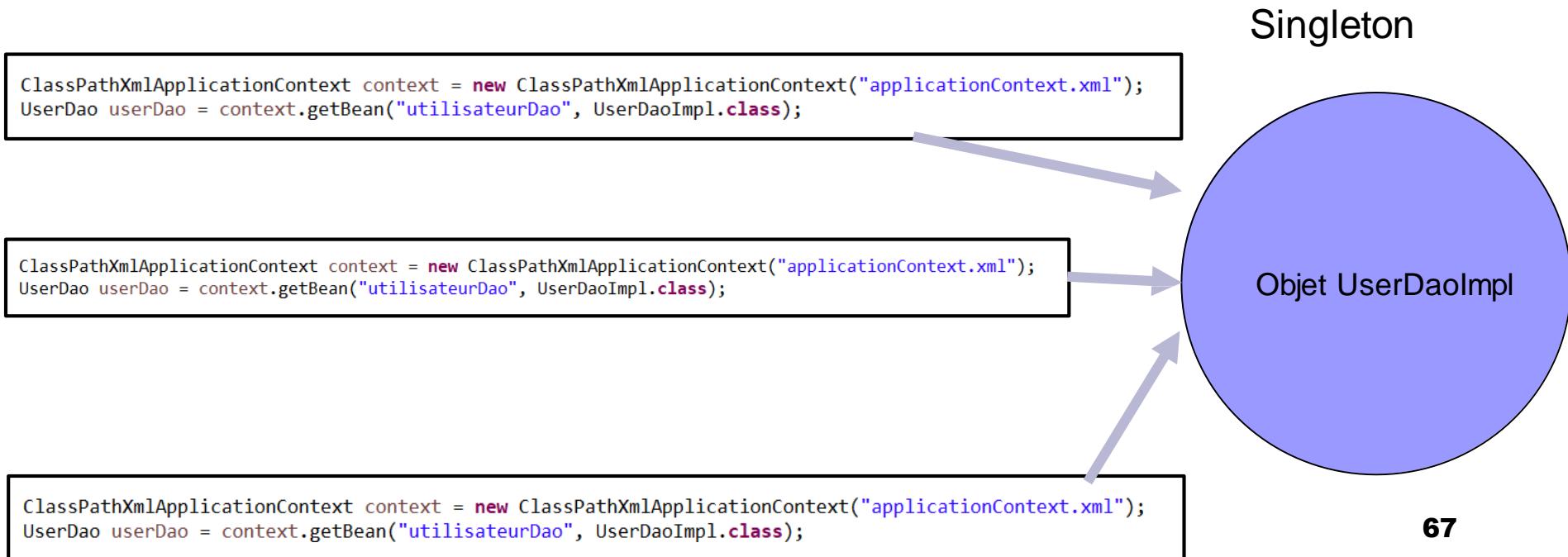
```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
UserDao userDao = context.getBean("utilisateurDao", UserDaoImpl.class);
```

# Bean scope (portée des beans)

## Le Scope Singleton:

- Le score fait référence au cycle de vie d'un bean
  - Combien de temps le bean reste existant?
  - Combien d'instances seront créées?

Le Scope par défaut est « Singleton »



# Bean scope (portée des beans)

## Le Scope prototype :

```
<bean id="utilisateurService" class="com.services.impl.UtilisateurServiceImpl" scope="prototype">
    <property name="userDao" ref="utilisateurDao"></property>
    <property name="roleDao" ref="roleDao"></property>
</bean>
```

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
UserDao userDao = context.getBean("utilisateurDao", UserDaoImpl.class);
```

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
UserDao userDao = context.getBean("utilisateurDao", UserDaoImpl.class);
```

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");
UserDao userDao = context.getBean("utilisateurDao", UserDaoImpl.class);
```

Object UserDaoImpl

Object UserDaoImpl

Object UserDaoImpl

# Bean scope (portée des beans)

## Les autres scopes :

Scope	Description
singleton	(Default) Scopes a single bean definition to a single object instance for each Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request. That is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring <code>ApplicationContext</code> .
session	Scopes a single bean definition to the lifecycle of an HTTP <code>session</code> . Only valid in the context of a web-aware Spring <code>ApplicationContext</code> .
application	Scopes a single bean definition to the lifecycle of a <code>ServletContext</code> . Only valid in the context of a web-aware Spring <code>ApplicationContext</code> .
websocket	Scopes a single bean definition to the lifecycle of a <code>WebSocket</code> . Only valid in the context of a web-aware Spring <code>ApplicationContext</code> .

*Extrait de la documentation officielle*

Pour plus d'informations :

<https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-scopes>

# Bean scope (portée des beans)

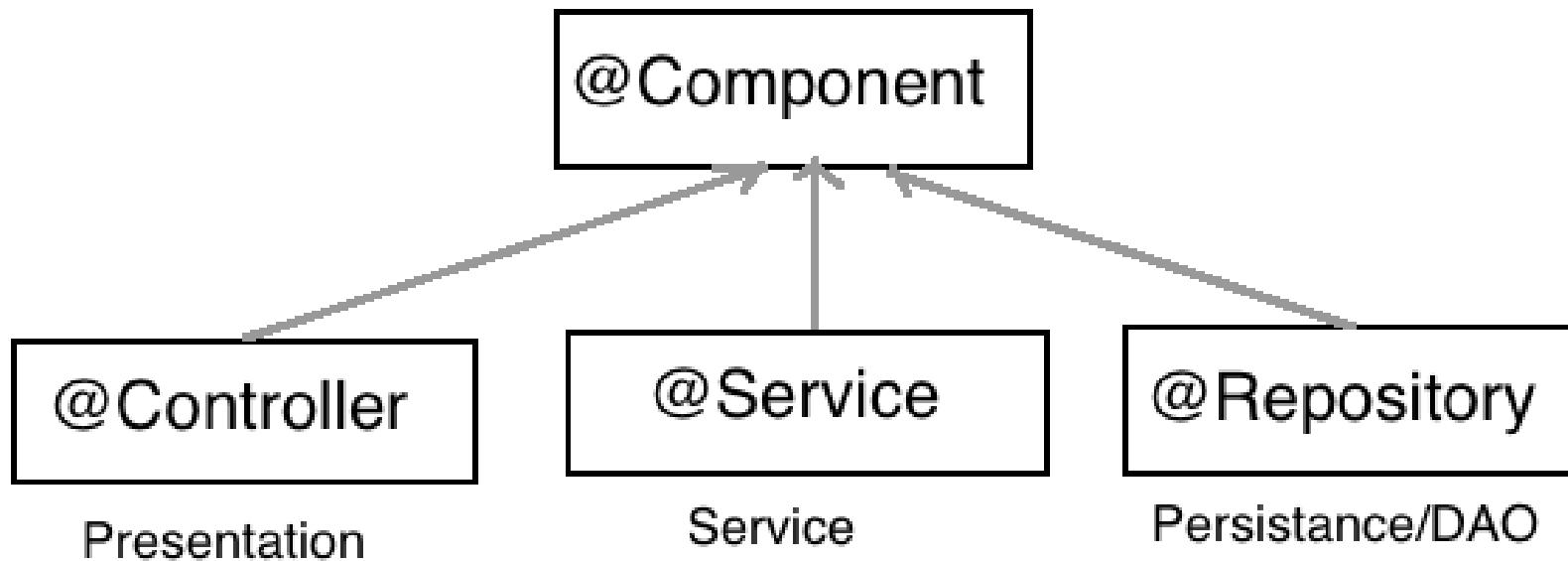
## Cycle de vie d'un bean / méthodes d'initialisation et de destruction

- Le cycle de vie d'un bean est géré par le conteneur Spring. Lorsque nous exécutons le programme, tout d'abord, le conteneur démarre. Après cela, le conteneur crée l'instance d'un bean selon la demande, puis les dépendances sont injectées, la méthode d'initialisation s'exécute en suite. Et enfin, le bean est détruit lorsque le conteneur est fermé. La méthode de destruction s'exécute juste avant la destruction de l'objet.

```
<bean id="utilisateurService"
      class="com.services.impl.UtilisateurServiceImpl"
      init-method="doInitializations"
      destroy-method="doCleanUp"
    >
    <property name="userDao" ref="utilisateurDao"></property>
    <property name="roleDao" ref="roleDao"></property>
</bean>
```

# Configuration par annotations

## Les annotations des beans



# Configuration par annotations

## Processus d'utilisation des annotations

- Activer le scan à la recherche des beans Spring (annotés par `@Component` ou ses descendantes)
- Ajouter les annotations sur les classes Java
- Retrouver les beans à partir du conteneur Spring

# Configuration par annotations

## Activer le scan

La configuration ci-dessous indique à Spring de chercher les classes annotées dans le package ***org.example***

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

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

</beans>
```

# Configuration par annotations

## Ajouter les annotations sur les beans

```
1 package com.ensah;  
2  
3 import org.springframework.stereotype.Component;  
4  
5 @Component("userDao")  
6 public class UserDaoImpl implements UserDao{  
7 //Your code here  
8 }
```

# Configuration par annotations

## Id par défaut d'un bean

Les beans Spring possèdent un id par défaut

- Si le nom de la classe est **XyyyZzzz** → l'id serait :  
**xyyyZzzz**

```
1 package com.ensah;
2
3 import org.springframework.stereotype.Component;
4
5 @Component
6 public class UserDaoImpl implements UserDao{
7 //Your code here
8 }
```

- Dans ce cas l'id par défaut est userDaoImpl

# Configuration par annotations

## Retrouver un bean depuis le context Spring

```
//Création de l'objet Standalone XML application  
context  
ClassPathXmlApplicationContext context = new  
ClassPathXmlApplicationContext("applicationContext.  
xml");  
  
//Retrouver le bean depuis le conteneur  
UserDao userDao = context.getBean("userDaoImpl",  
UserDaoImpl.class);
```

# Configuration par annotations

## Spring autowiring

- Le framework Spring permet l'injection automatique de dépendances (*autowiring*) . Spring peut gérer automatiquement les relations entre les beans collaborant. Il y a trois types d'injection par *autowiring* :
  - Injection par constructeur
  - Injection par setter
  - injection par attribut

# Configuration par annotations

## Spring autowiring - Injection par constructeur

```
1 package com.ensah;
2
3 import org.springframework.stereotype.Component;
4
5 @Component
6 public class UserDaoImpl implements UserDao{
7 //Your code here
8 }
9
```

```
1 package com.ensah;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Service;
5
6 @Service
7 public class UserServiceImpl implements UserService {
8
9     private UserDao userDao;
10
11     @Autowired
12     public UserServiceImpl(UserDao dao) {
13         this.userDao = dao;
14     }
15 }
```

# Configuration par annotations

## Spring autowiring - Injection par setter

```
1 package com.ensah;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Service;
5
6 @Service
7 public class UserServiceImpl implements UserService {
8
9     private UserDao userDao;
10
11
12     @Autowired
13     public void setUserDao(UserDao userDao) {
14         this.userDao = userDao;
15     }
16 }
```

# Configuration par annotations

Spring autowiring - Injection par une méthode quelconque

```
1 package com.ensah;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Service;
5
6 @Service
7 public class UserServiceImpl implements UserService {
8
9     private UserDao userDao;
10
11     @Autowired
12     public void doSomeStuff(UserDao userDao) {
13         this.userDao = userDao;
14     }
15 }
```

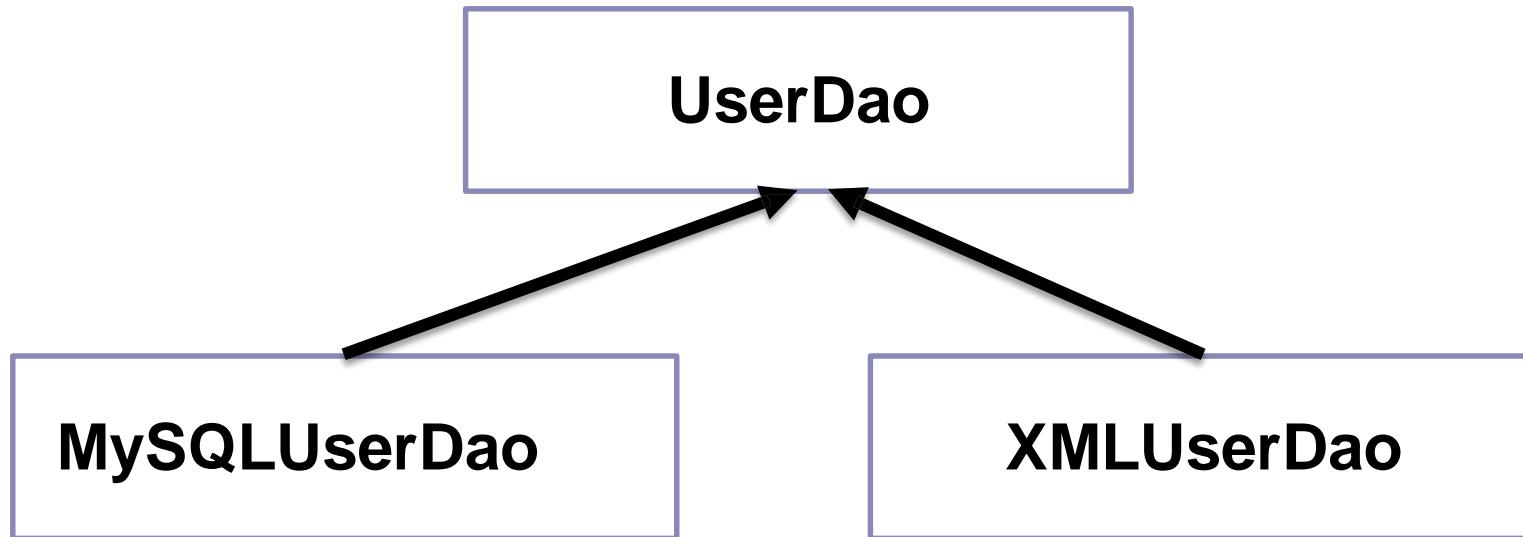
# Configuration par annotations

## Spring autowiring - Injection par attribut

```
1 package com.ensah;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Service;
5
6 @Service
7 public class UserServiceImpl implements UserService {
8
9     @Autowired
10    private UserDao userDao;
11
12    //No need to setter method
13 }
```

# Configuration par annotations

Que faire en cas de plusieurs implémentations ?



- En cas d'ambiguïté on utilise une autre annotation  
**@Qualifier**

# Configuration par annotations

## L'annotation @Qualifier

```
@Service
public class UserServiceImpl implements UserService {
    @Autowired
    @Qualifier("xmlDaoImpl")
    private UserDao userDAO;
    //No need to setter method
}
```

# Configuration par annotations

## Le scope d'un bean avec @scope

```
| @Service  
| @Scope("prototype")  
| public class UserServiceImpl implements UserService {  
|     @Autowired  
|     private UserDao userDao;  
|     //No need to setter method  
| }
```

# Configuration par annotations

## Les annotations `@PostConstruct` et `@PreDestroy`

```
3 public class UserServiceImpl implements UserService {  
4  
5     @Autowired  
6     private UserDao userDao;  
7  
8     @PostConstruct  
9     public void doSomethingAtStarting() {  
10         //code to execute at initialization  
11     }  
12  
13     @PreDestroy  
14     public void doSomethingAtEnd() {  
15         //code to execute at initialization  
16     }  
17 }  
18 }
```

# Configuration via code Java

- La configuration se fait via une classe annotée par **@Configuration**

```
@Configuration
@ComponentScan("com.ensah.app.services")
public class AppConfig {
    // définitions des bean...
}
```

- Pour plus d'informations voir :

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/Configuration.html>

# Configuration via code Java

## ■ Définition des bean dans la classe de configuration

```
1 package com.ensah;
2
3 import org.springframework.context.annotation.Bean;
4 import org.springframework.context.annotation.ComponentScan;
5 import org.springframework.context.annotation.Configuration;
6
7 @Configuration
8 @ComponentScan("com.ensah") // Optionnel
9 public class AppConfig {
10
11     @Bean
12     public UserService getUserService() {
13         return new UserServiceImpl(getUserDao());
14     }
15
16     @Bean
17     public UserDao getUserDao() {
18         return new XMLUserDaoImpl();
19     }
20
21 }
```

*Injection de dépendance  
Ici l'id du bean est le nom de la méthode*

# Configuration via code Java

- Définition des bean dans la classe de configuration

```
public static void main( String[] args )
{
    AnnotationConfigApplicationContext context =
        new AnnotationConfigApplicationContext(AppConfig.class);

    UserService bean = context.getBean("getUserService", UserServiceImpl.class);

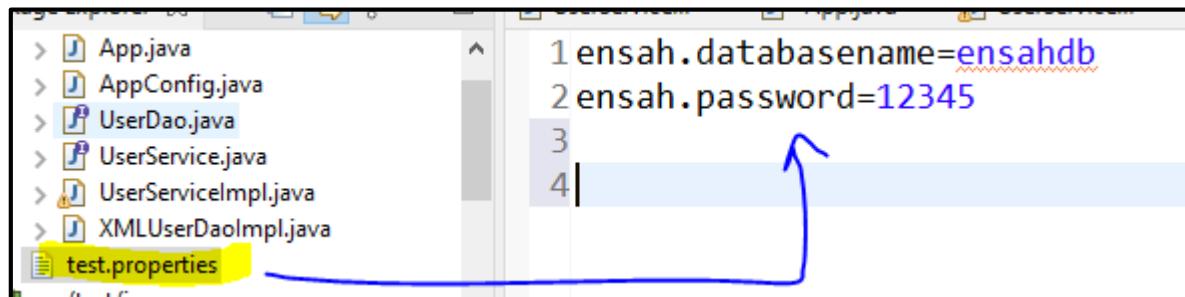
    context.close();
}
```

*L'id du bean est le nom de la méthode*

# Configuration via code Java

## ■ Injection des valeurs depuis un fichier properties

*Fichier test.properties*



*Annotation @PropertySource*

```
8 @Configuration
9 @PropertySource("classpath:test.properties")
10 public class AppConfig {
11
12
13
14 }
```

# Configuration via code Java

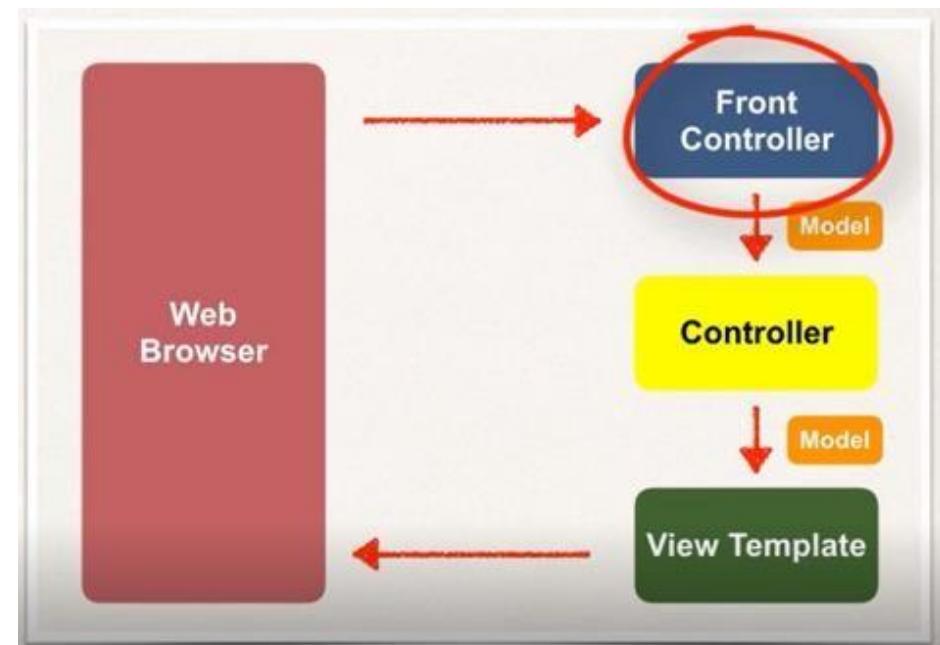
## ■ Injection des valeurs depuis un fichier properties

```
1 package com.ensah;
2
3 import org.springframework.beans.factory.annotation.Value;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class XMLUserDaoImpl implements UserDao{
8
9
10    @Value("${ensah.databasename}")
11    private String dbname;
12
13
14    public void create() {
15
16        System.out.println("DbName = "+dbname);
17    }
18}
19
```

# Spring MVC

# ■ Modèle MVC de Spring MVC

- **Contrôleur frontal (Front Controller)** : le contrôleur *DispatcherServlet*, offert par le Framework
- Les utilisateurs du Framework ont à développer :
  - **Les modèles**
  - **Les contrôleurs**
  - **Les vues**



# Modèle MVC de Spring MVC

## ■ Contrôleur

- Son code est à écrire par le développeur (l'utilisateur du Framework)
- Implémente la logique business :
  - Traitement de la requête
  - Sauvegarder/retrouver les données (en appelant des services métiers, des web services, ou en accédant à des bases de données,...)
  - Mise à jour des données du modèle
- Rediriger vers les vues appropriées

## ■ Modèle

- Dans le contexte de Spring MVC, un modèle représente généralement les données qui seront transmises vers la vue depuis une opération (définie dans un contrôleur Web)

## ■ Vue

- Spring MVC offre une grande flexibilité pour le choix du View Template: JSP + JSTL, Thymeleaf, Groovy, Freemarker, Velocity,...

# Configuration de Spring MVC

## ■ Etape 01 : Déclarer le contrôleur dans le fichier web.xml

```
<!-- Spring MVC Configs -->

<!-- Step 1: Configure Spring MVC Dispatcher Servlet -->
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/applicationContext.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>

<!-- Step 2: Set up URL mapping for Spring MVC Dispatcher Servlet -->
<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

# Configuration de Spring MVC

## ■ Etape 02 : fichier applicationContext.xml

- On Ajoute le code nécessaire pour activer le scan à la recherche des classes annotées par @component ou ses décadentes.
- On configure « View Resolver »

```
<!-- Step 3: Add support for component scanning -->
<context:component-scan base-package="com.ensah" />
```



```
<!-- Step 4: Define Spring MVC view resolver -->
<bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/view/" />
    <property name="suffix" value=".jsp" />
</bean>
```



# Configuration de Spring MVC

## ■ Etape 03 : Création d'un contrôleur

```
@Controller  
@RequestMapping("/students")  
public class StudentController {  
  
    @RequestMapping("/showForm")  
    public String showForm(Model model) {  
        Student s = new Student();  
        s.setFirstName("Boudaa");  
        model.addAttribute("student",s);  
        return "form";  
    }  
  
    @RequestMapping("/addStudent")  
    public String process(@ModelAttribute("student") Student pStudent) {  
        System.out.println(pStudent);  
        return "test";  
    }  
}
```

# Configuration de Spring MVC

## ■ Etape 04 : La vue

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

```
<f:form method="POST" action="addStudent" modelAttribute="student">

    <label>First name</label>    <br>
    <f:input path="firstName" /><br>
    <label>Last name</label><br>
    <f:input path="lastName" /><br>
    <label>Last name</label><br>
    <f:input path="lastName" /><br>
    <label>Email</label><br>
    <f:input path="email" /><br>
    <label>Birth date</label><br>
    <f:input type="date" path="birthDate" /><br>
    <label>Contact Number</label><br>
    <f:input path="contactNumber" /><br>
    <input type="submit" value="Submit" /><br>

</f:form>
```

Form.jsp

```
9<body>
10
11 Thank you ${student.firstName}
12
13 </body>
```

test.jsp

# Configuration de Spring MVC

## ■ Configuration sans fichier web.xml

Le fichier web.xml peut être remplacé par une classe Java

```
package com.ensah.config;
import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;

public class DispatcherServletInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        // TODO Auto-generated method stub
        return null;
    }
    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { AppConfig.class };
    }
    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

Voir également la solution du TP 2 :

[https://github.com/boudaa/ensah\\_javaee\\_2021/tree/master/tp2\\_no\\_xml](https://github.com/boudaa/ensah_javaee_2021/tree/master/tp2_no_xml)

# Validation des données avec Hibernate Validator

- Validation des données avec les annotations de Hibernate validator:  
@NotNull, @NotEmpty, @NotBlank, @Size, @Min, @Max, @Pattern,  
@Email,...

```
@NotBlank(message = "This field is required")
@Pattern(regexp = "[A-Z]{2}[0-9]{8}", message= "The National ID must be 2 upper
    + " letters followed by 8 digits")
private String nationalIdNumber;
@NotBlank(message = "This field is required")
private String firstName;
@NotBlank(message = "This field is required")
private String lastName;
@Min(value = 20, message = "Age must be > 19")
@Max(value = 90, message = "Age must be < 91")
@NotNull(message = "This field is required")
private int age;
@email(message = "Enter a valid email")
@NotBlank(message = "This field is required")
private String email;

@NotBlank(message = "This field is required")
@Size(min = 10, message = "The password is too short")
@Size(max = 20, message = "The password is too big")
private String password;
```

Pour plus d'informations sur  
Hibernate Validator :

[https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html\\_single/](https://docs.jboss.org/hibernate/stable/validator/reference/en-US/html_single/)

# Spring MVC : Accès à la requête, le contexte et la session

## ■ Accès à la requête et la session

Il y a plusieurs méthodes possibles, ci-dessous des exemples:

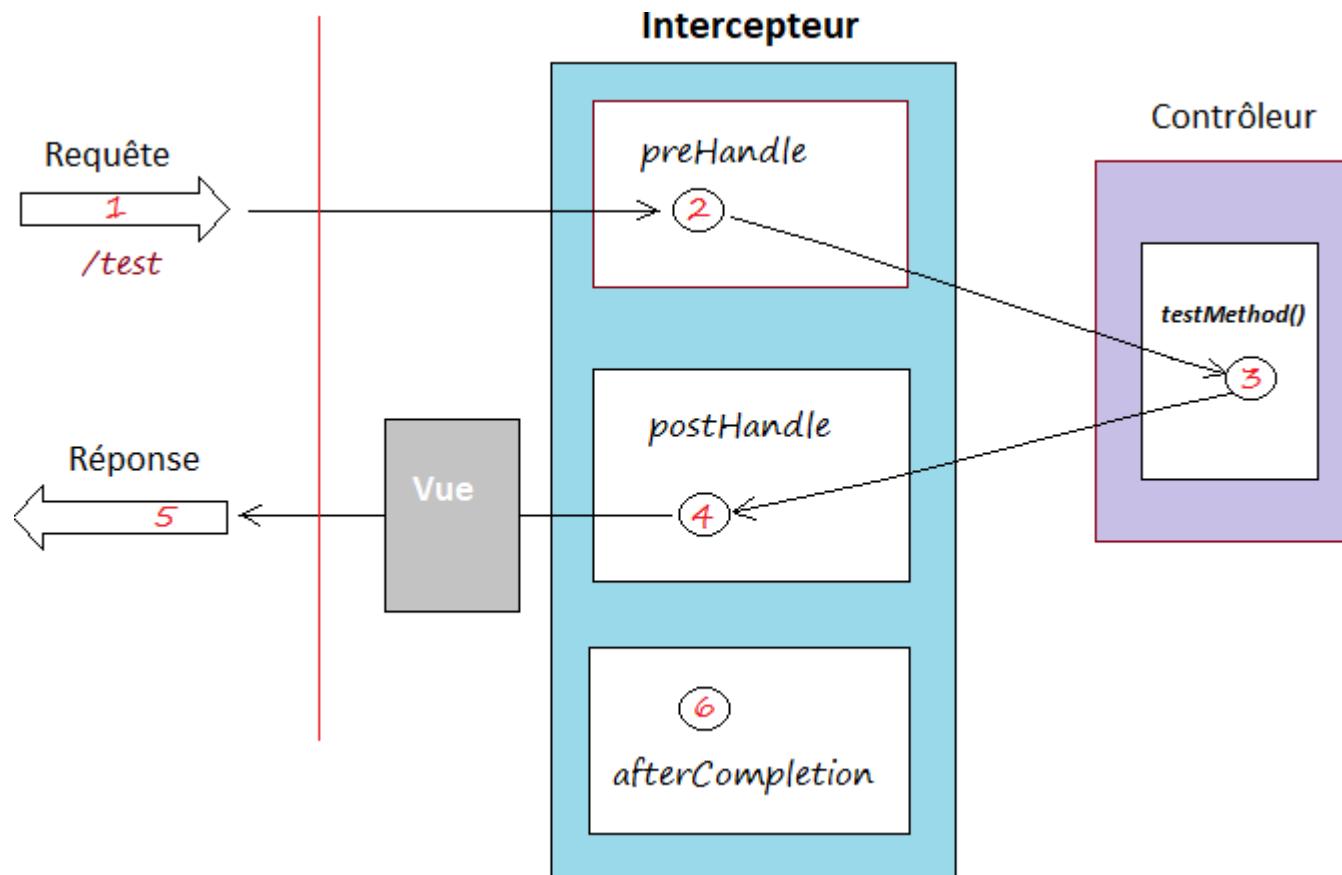
```
@RequestMapping("/test")
public String showForm(Model model, HttpServletRequest request) {
    //Accès à la session via HttpServletRequest
    HttpSession session = request.getSession();
    session.setAttribute("TEST", new Date());
    //Accès au contexte de servlets
    ServletContext context = request.getServletContext();
    session.setAttribute("TEST", new Date());
    return "test";
}
```

## ■ Autre méthode pour l'accès au ServletContext

```
@Autowired
private ServletContext
appContext;
```

# Spring MVC : Intercepteurs

## ■ Cycle de vie de l'intercepteur



# Spring MVC : Intercepteurs

## ■ Configurer l'intercepteur

```
@EnableWebMvc  
 @Configuration  
 @ComponentScan(basePackages = { "com.ensah" })  
 @EnableTransactionManagement  
 public class AppConfig implements WebMvcConfigurer {
```

```
.....
```

```
 @Bean  
 MyInterceptor myInterceptor () {  
     return new MyInterceptor ();  
 }
```

```
 @Override  
 public void addInterceptors(InterceptorRegistry registry) {  
     registry.addInterceptor(myInterceptor());  
 }
```

```
}
```

# Spring MVC : Intercepteurs

## ■ Définir l'intercepteur

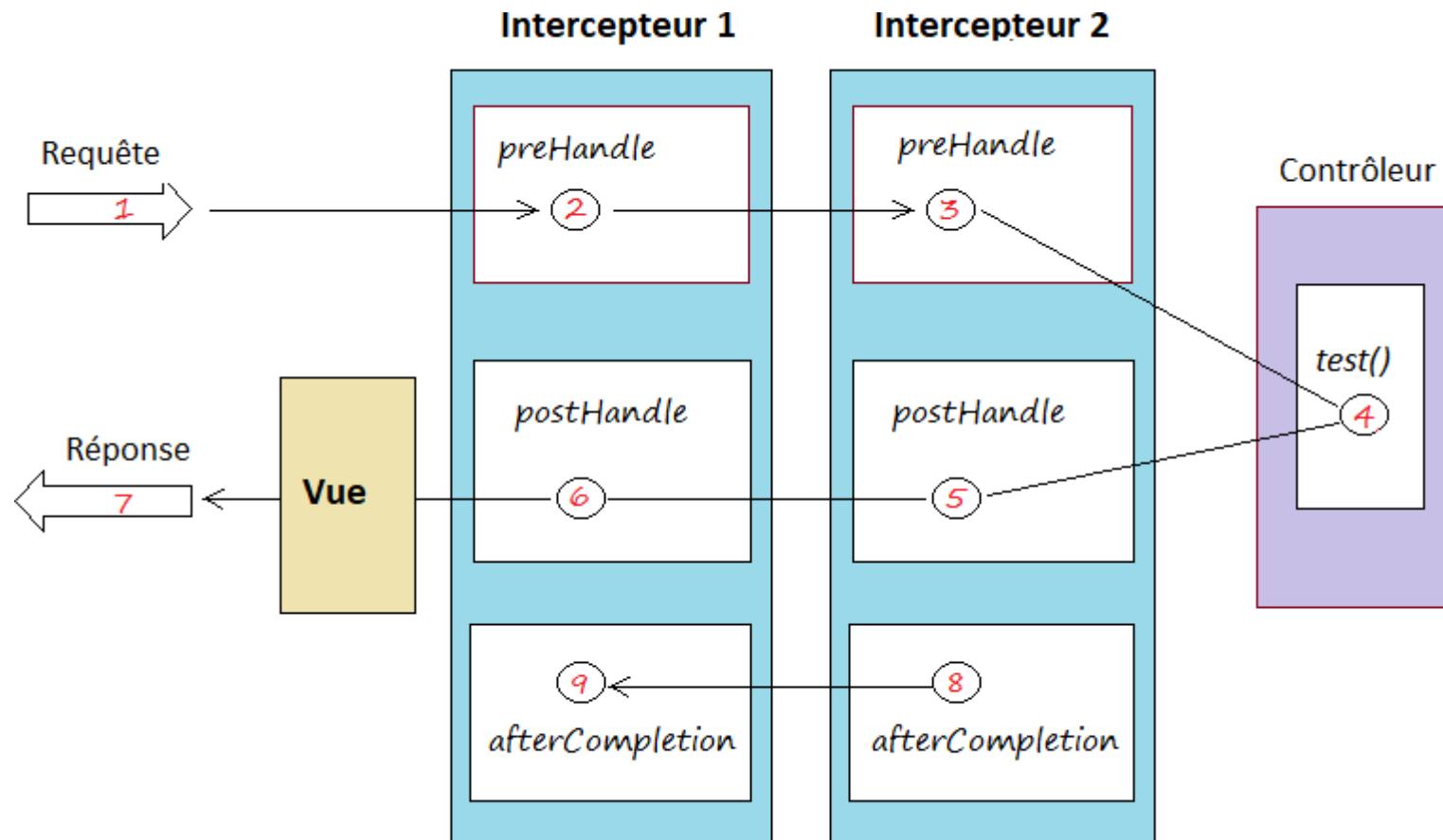
```
public class MyInterceptor implements HandlerInterceptor {
    @Override
    public boolean preHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler) throws Exception {
        ...
    }

    @Override
    public void postHandle(HttpServletRequest request,
        HttpServletResponse response, Object handler,
        ModelAndView modelAndView) throws Exception {
        ...
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
        HttpServletResponse response, Object handler, Exception exception)
        throws Exception {
        ...
    }
}
```

# Spring MVC : Intercepteurs

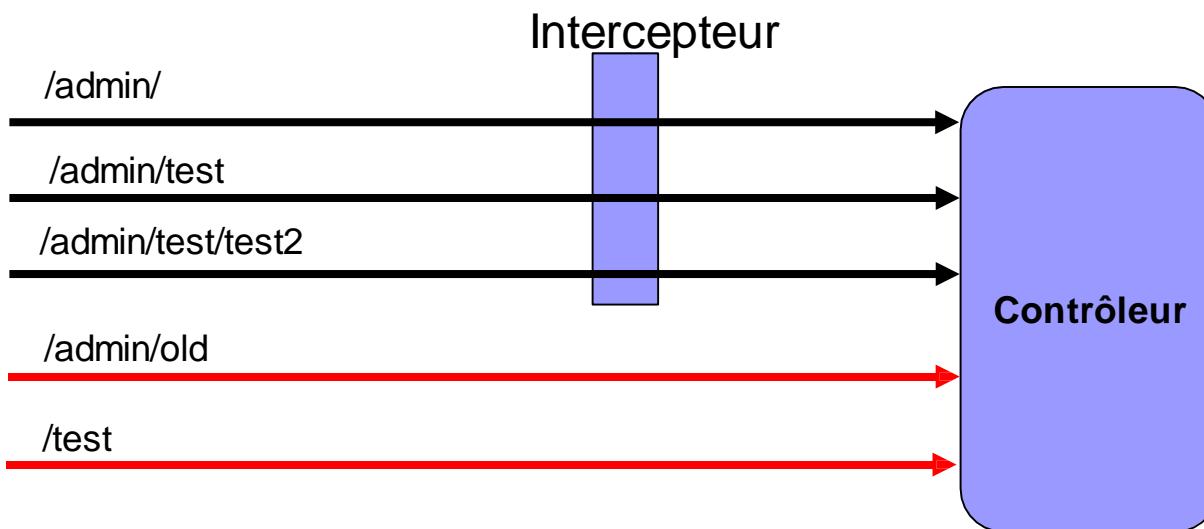
## ■ Cas de plusieurs intercepteurs

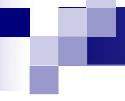


# Spring MVC : Intercepteurs

## ■ Déterminer les requêtes à intercepter et à exclure

```
@Bean  
MyInterceptor myInterceptor() {  
    return new MyInterceptor();  
}  
  
@Override  
public void addInterceptors(InterceptorRegistry registry) {  
    registry.addInterceptor(myInterceptor())  
        .addPathPatterns("/admin/*") // Intercept toutes les requêtes  
                                // qui commencent par /admin/  
        .excludePathPatterns("/admin/old"); //sauf /admin/old  
}
```





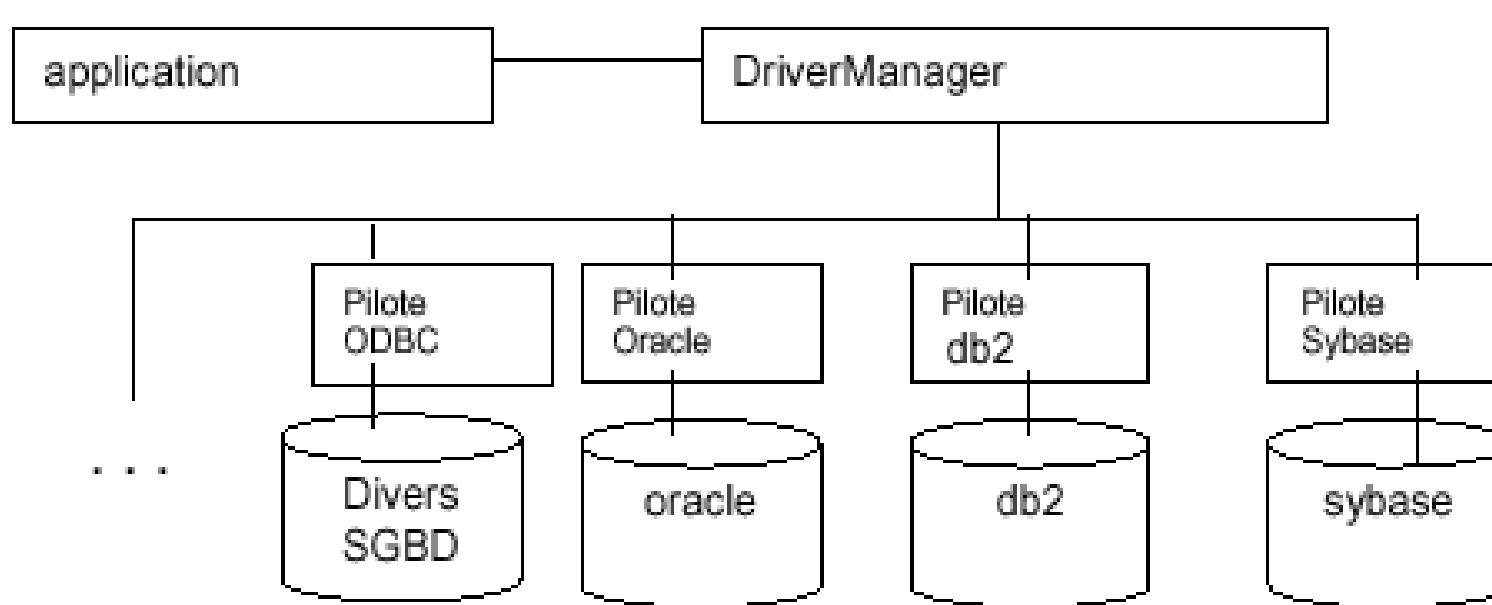
# **Techniques de gestion de persistance**

# Introduction

- Il existe plusieurs moyen pour persister l'état des objets :
  - ❖ Sérialisation (insuffisante pour les applications complexes)
  - ❖ JDBC (plusieurs inconvénients pour les grands projets)
  - ❖ **ORM (Mapping Objet Relationnel)**

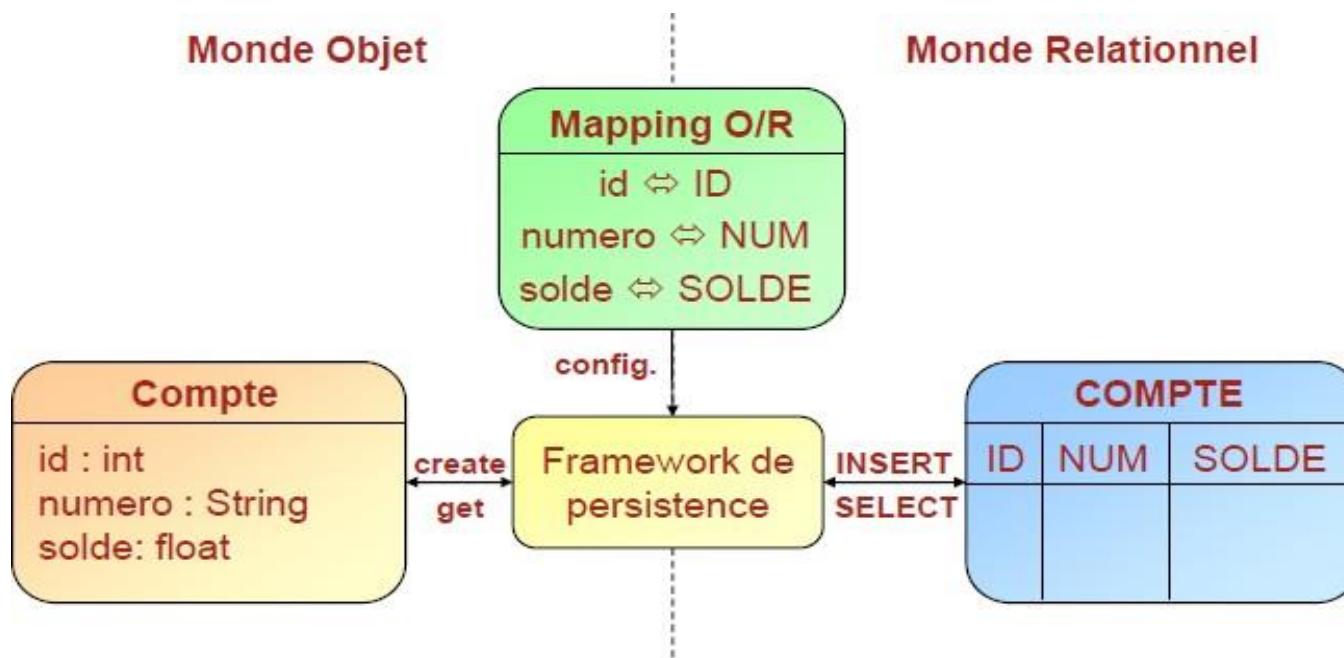
# JDBC

JDBC est l'acronyme de Java DataBase Connectivity et désigne l'API Java standard qui permet d'accéder, à partir de programmes Java, à un SGBD relationnel.



# ORM

Le mapping objet-relationnel (en anglais object-relational mapping ou ORM) est une technique de programmation informatique qui crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé. On pourrait le désigner par « correspondance entre monde objet et monde relationnel »



# Mapping objet relationnel

Un outil de mapping O/R doit cependant proposer un certain nombre de fonctionnalités parmi lesquelles :

- Assurer le mapping des tables avec les classes, des champs avec les attributs, des relations et des cardinalités
- Proposer une interface qui permette de facilement mettre en œuvre des actions de type CRUD
- Proposer un langage de requêtes indépendant de la base de données cible et assurer une traduction en SQL natif selon la base utilisée
- Proposer un support des transactions
- Assurer une gestion des accès concurrents (verrou, deadlock, ...)
- Fournir des fonctionnalités pour améliorer les performances (cache, lazy loading, ...)
- ...

# Un Framework c'est quoi?

- Un framework, désigne un ensemble cohérent de composants logiciels **structurels**, qui sert à créer les fondations ainsi que les grandes lignes de tout ou d'une partie d'un logiciel (**architecture**). Un framework se distingue d'une simple bibliothèque logicielle principalement par :
  - **son caractère générique, faiblement spécialisé**, contrairement à certaines bibliothèques ; un framework peut à ce titre être constitué de plusieurs bibliothèques chacune spécialisée dans un domaine. Un framework peut néanmoins être spécialisé, sur un langage particulier, une plateforme spécifique, un domaine particulier : communication de données, data mapping, etc. ;
  - le cadre de travail (traduction littérale de l'anglais : framework) qu'il impose de par sa construction même, **guidant l'architecture logicielle voire conduisant le développeur à respecter certains patrons de conception** ; les bibliothèques le constituant sont alors organisées selon le même paradigme.

# Une API c'est quoi?

- Une interface de programmation applicative (**API** pour *Application Programming Interface*) est un ensemble normalisé d'interfaces, de classes, de méthodes ou de fonctions qui sert de façade par laquelle un logiciel offre des services à d'autres logiciels. Elle est offerte par une bibliothèque logicielle ou un service web, le plus souvent accompagnée d'une description qui spécifie comment des programmes consommateurs peuvent se servir des fonctionnalités du programme fournisseur.

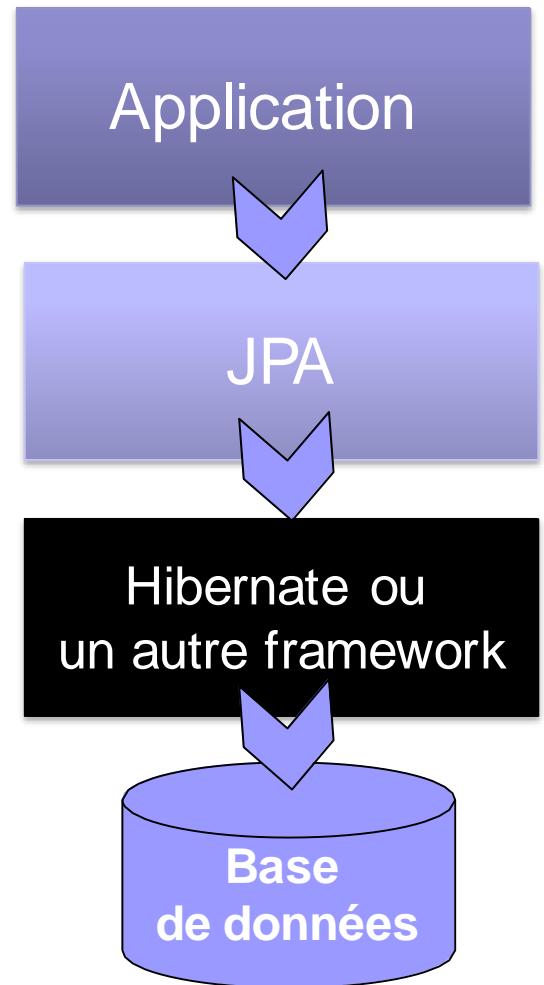
# API et Framework ORM

## ■ Framework (Implémentations)

- *Hibernate*
- *iBatis*
- *EclipseLink*
- *TopLink*
- *Java Data Objects (JDO)*
- ...

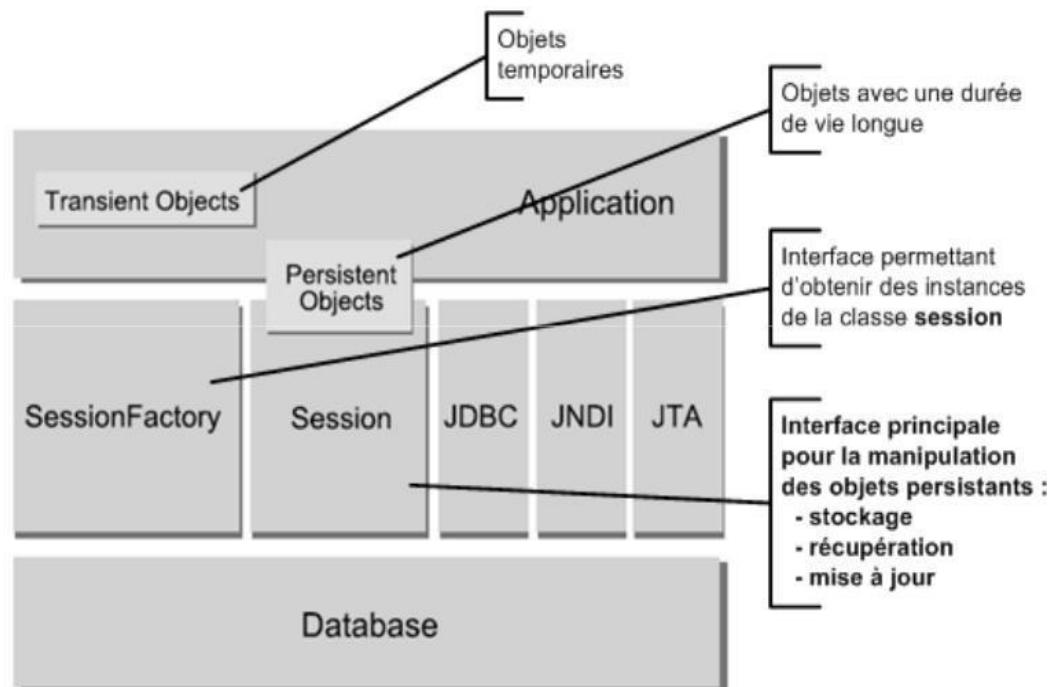
## ■ API (Interfaces / Specifications) :

- JPA



# Le framework Hibernate

- Hibernate est un outil de mapping O/R qui permet la persistance transparente pour des objets Java dans des bases de données relationnelles. Il regroupe un ensemble de librairies assurant la tâche de persistance.



# Le framework Hibernate

- Hibernate génère le code SQL
- Avec Hibernate, il n'y a pas d'objet ResultSet à gérer : Cycle de récupération manuelle des ResultSet + Casting de chaque ligne du resultset (type Object) vers un type d'objet métier.
- Persistance transparente : Le développeur peut faire de ses classes métiers des classes persistantes sans ajout de code tiers.
- Les objets métiers sont plus faciles à manipuler.
- Peu de dépendance envers une base de données précise.

# Le fichier hibernate.cfg

Parce qu'Hibernate est conçu pour fonctionner dans différents environnements, il existe beaucoup de paramètres de configuration à fixer (*paramètres d'accès à la base de données : login, mot de passe, le dialecte SQL à utiliser, taille du pool de connexion, configuration des transactions, ...*)

Ces configurations s'effectuent dans le fichier **hibernate.cfg.xml** (ou **hibernate.properties**)

Les dernières versions du Framework permettent également de faire **une configuration par une classe Java.**

# Le fichier hibernate.cfg

## ■ Exemple d'un fichier hibernate.cfg.xml

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- Database host -->
        <property name="hibernate.connection.url">jdbc:mysql://localhost/tpj2ee2017</property>
        <!-- JDBC Driver for MySQL -->
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <!-- Database user login -->
        <property name="hibernate.connection.username">root</property>
        <!-- Database user password -->
        <property name="hibernate.connection.password">boudaa</property>
        <!-- SQL Dialect, we will use MySQL Dialect -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <!-- we cant to show sql command in console (just for debug purpose) -->
        <property name="hibernate.show_sql">true</property>
        <!-- we generate the database schema once and we update it if necessary -->
        <property name="hibernate.hbm2ddl.auto">update</property>
        <!-- transaction manager -->
        <property name="hibernate.transaction.factory_class">org.hibernate.transaction.JDBCTransactionFactory</property>
        <property name="hibernate.current_session_context_class">thread</property>
        <!-- here we specify the mapping classes -->
        <mapping class="com.ensah.Etudiant" />
    </session-factory>
</hibernate-configuration>
```

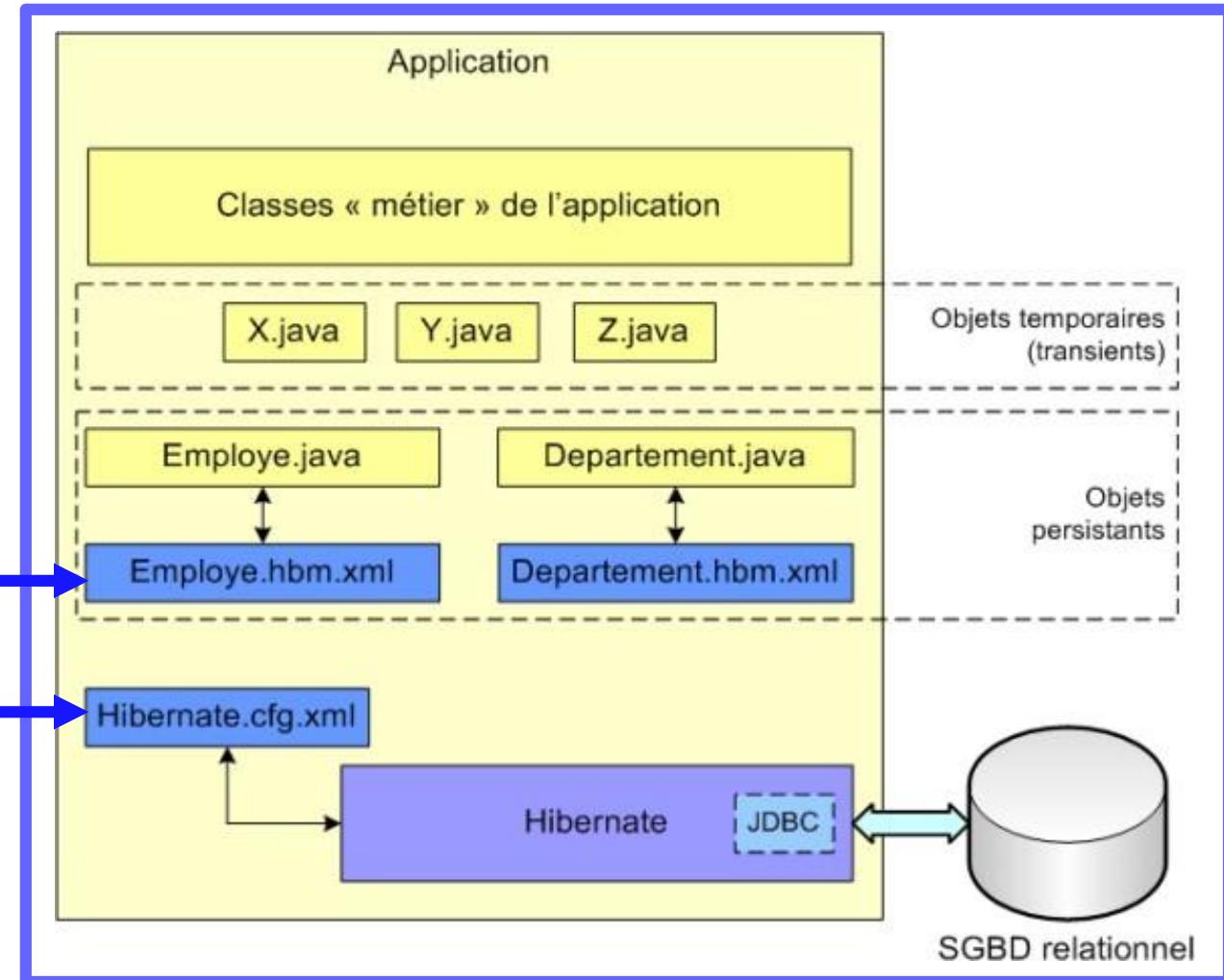
Voir également le code des exemples de cours sur :

[https://github.com/boudaa/ensah\\_javaee\\_2021/tree/master/Exemple\\_Cours\\_Hibernate\\_01](https://github.com/boudaa/ensah_javaee_2021/tree/master/Exemple_Cours_Hibernate_01)

# Contenu d'une (simple) application Hibernate

Ou annotations

Ou classe Java de configuration



# Session Hibernate

- Un objet *mono-threadé*, à durée de vie courte, qui représente une conversation entre l'application et l'entrepôt de persistance. Encapsule une connexion JDBC. Fabrique des objets Transaction. La Session contient un cache (de premier niveau) des objets persistants, qui sont utilisés lors de la navigation dans le graphe d'objets ou lors de la récupération d'objets par leur identifiant.

# SessionFactory

- Un cache *threadsafe* de mappages compilés pour une base de données. Elle permet de fabriquer les Sessions
- Elle est Client du *ConnectionProvider* (*une fabrique de connexions JDBC*)
- *SessionFactory* peut contenir un cache optionnel de données (*de second niveau*)

# SessionFactory

## ■ Récupérer la session factory

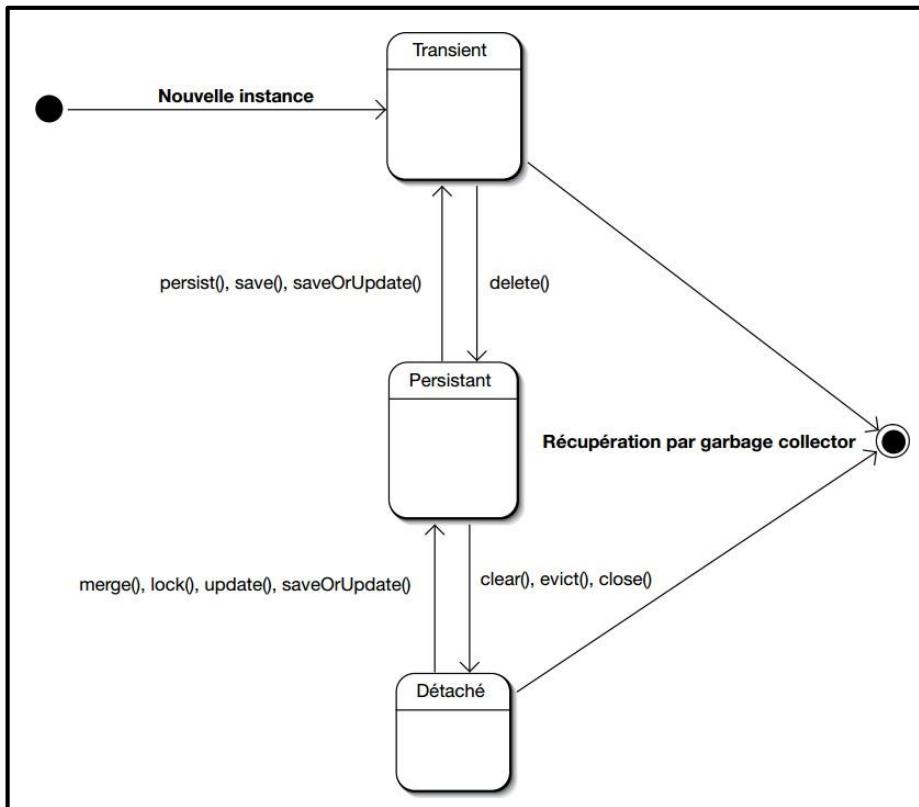
```
public static SessionFactory getSessionFactory() {  
    SessionFactory sessionFactory = null;  
    // A SessionFactory is set up once for an application!  
    final StandardServiceRegistry registry = new StandardServiceRegistryBuilder()  
        .configure() // configures settings from hibernate.cfg.xml  
        .build();  
    try {  
        sessionFactory = new MetadataSources(registry).buildMetadata()  
            .buildSessionFactory();  
    } catch (Exception e) {  
        // The registry would be destroyed by the SessionFactory, but we had  
        // trouble building the SessionFactory  
        // so destroy it manually.  
        StandardServiceRegistryBuilder.destroy(registry);  
    }  
  
    return sessionFactory;  
}
```

# Le cycle de vie d'un objet manipulé avec Hibernate

Il y a trois états possibles pour les instances d'objets :

- **Transient (temporaire, éphémère)** : Objet n'ayant pas d'image dans la base de donnée ( ne survivent pas à l'arrêt de l'application).
- **Persistant** : Objet stocké dans la base de données, liés à un contexte de persistance (objet session). Il y a garantie par Hibernate de l'équivalence entre les données stockées en base et l'objet Java persistant
- **Détaché** : ancien objet persistant en dehors d'un contexte de persistance. Cet objet n'est pas surveillé par la session.

# Le cycle de vie d'un objet manipulé avec Hibernate et méthodes de la session



Pour plus d'informations voir la documentation :

<https://docs.jboss.org/hibernate/orm/5.5/javadocs/org/hibernate/Session.html>

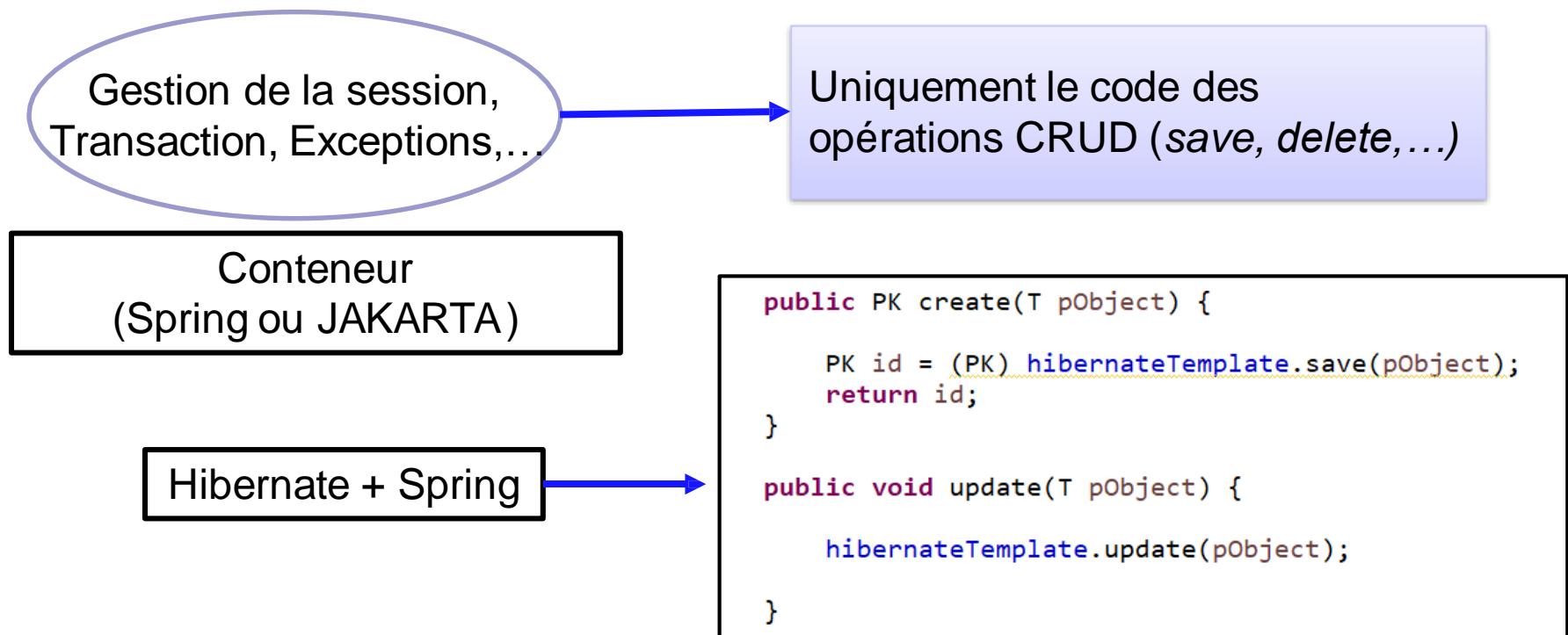
# Manipulation des données avec Hibernate

## ■ Méthode d'exécution d'une opération sur la base de données

```
Session session = null;
Transaction tx = null;
try {
    SessionFactory sf = SessionFactoryBuilder.getSessionFactory();
    // on obtient une session
    session = sf.getCurrentSession();
    // On commence une transaction
    tx = session.beginTransaction();
    // on execute les opérations bases de données (save, delete,...)
    // ...
    // On valide la transaction, ceci ferme également la session
    tx.commit();
} catch (HibernateException ex) {
    // Si il y a des problèmes et une transaction a été déjà créée on l'annule
    if (tx != null) {
        // Annulation d'une transaction
        tx.rollback();
    }
    // On n'oublie pas de remonter l'erreur originale
    throw ex;
} finally {
    // Si la session n'est pas encore fermée par commit if
    (session != null && session.isOpen()) {
        session.close();
    }
}
```

# Utilisation de Hibernate au sein d'un conteneur

- Le code précédent devient simple lorsqu'on utilise Hibernate dans un conteneur comme Spring.

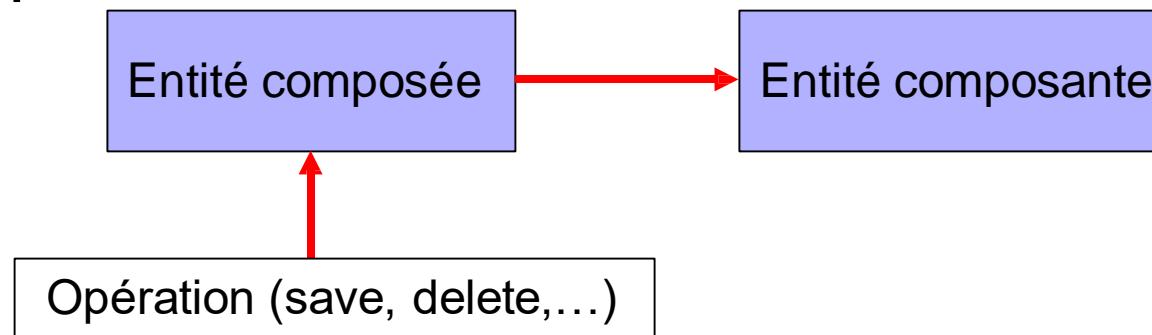


# Propagation des opérations (Cascade)

- La propagation est à spécifier par l'attribut cascade de l'annotation `@OneToOne`
- JPA Cascade Type

*Propagation de l'opération ?*

- ✓ `ALL`
- ✓ `PERSIST`
- ✓ `MERGE`
- ✓ `REMOVE`
- ✓ `REFRESH`
- ✓ `DETACH`



- Hibernate Cascade Type

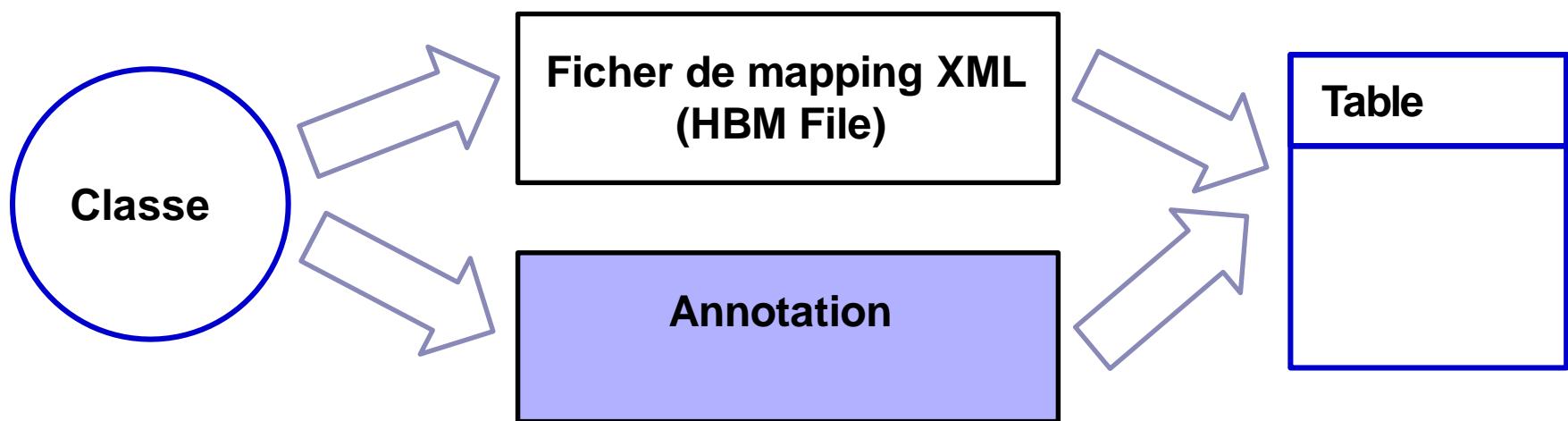
En plus des types précédents Hibernate possède d'autres types de propagations:

# Méthodes de requêtage des données

- Hibernate offre plusieurs moyens pour la manipulation et le requêtage des données :
  - ❖ Langage HQL
  - ❖ Langage JPQL (Via API JPA)
  - ❖ API Criteria
  - ❖ Requêtes SQL natives

# ORM avec Hibernate

- Deux moyens pour faire le mapping avec Hibernate :
  - Avec les annotations (**méthode moderne**)
  - Avec des configurations XML via les fichiers *hbm* (**méthode anicienne**)



# ORM avec Hibernate

## ■ Mapping avec annotations

```
@Entity  
@Table( name = "EVENTS" )  
public class Event {  
    ...  
}
```

```
@Id  
@GeneratedValue(generator="increment")  
@GenericGenerator(name="increment", strategy = "increment")  
public Long getId() {  
    return id;  
}
```

# ORM avec Hibernate

## ■ Mapping avec annotations

```
@Entity(name = "AUTEUR_TAB")
public class Etudiant {
    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    private Long id;

    @Column(name = "nom_etudiant", length = 50, nullable = false)
    private String nom;

    @Column(name = "nom_etudiant", length = 50, nullable = false, unique = true)
    private String cin;

    // Cette propriété est persistante, elle reçoit une configuration par défaut
    // (JPA utilise le principe de configuration par exception )
    private String prenom;
    // Un objet peut avoir des propriétés que l'on ne souhaite pas
    // rendre persistantes dans la base. Il faut alors impérativement les marquer
    // avec l'annotation avec @Transient.
    @Transient
    private int valeurCalculée;

    //getters & setters
}
```

# ORM avec Hibernate

## ■ Mapping avec XML

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD//EN"
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="Employee" table="EMPLOYEE">
        <meta attribute="class-description">
            This class contains the employee detail.
        </meta>
        <id name="id" type="int" column="id">
            <generator class="native"/>
        </id>
        <property name="firstName" column="first_name" type="string"/>
        <property name="lastName" column="last_name" type="string"/>
        <property name="salary" column="salary" type="int"/>
    </class>
</hibernate-mapping>
```

# Relation unidirectionnelle de type One-to-One

```
package com.bo;
import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
@SuppressWarnings("unused")

@Entity(name = "ETUDIANT_TAB")
public class Etudiant {
    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    private Long id;
    private String nom;
    private String cin;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name="id_adresse_etudiant")
    private Adresse adresse;

    private String prenom;
    //getters & setters
}

@Entity
public class Adresse {

    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    @Column(name = "id")
    private Long id;

    private String ville;
    // getters/setters
}
```

dbtestcours adresse	
!	id : bigint(20)
!	ville : varchar(255)

dbtestcours etudiant_tab	
!	id : bigint(20)
!	cin : varchar(255)
!	nom : varchar(255)
!	prenom : varchar(255)
#	id_adresse_etudiant : bigint(20)

# Relation bidirectionnelle de type One-to-One

```
@Entity
public class Adresse {

    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    @Column(name = "id")
    private Long id;
    @OneToOne(mappedBy = "adresse")
    private Etudiant etudiant;

    private String ville;

    public Long getId() {
        return id;
    }
}
```

adresse est le nom de l'attribut dans la classe Etudiant

```
@OneToOne(cascade = CascadeType.ALL)
@JoinColumn(name="id_adresse_etudiant")
private Adresse adresse;
```

# Relation unidirectionnelle de type One-to-many

```
@Entity(name = "ETUDIANT_TAB")
public class Etudiant {
    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    private Long id;
    private String nom;
    private String cin;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name="id_etudiant")
    private Set<Adresse> adresses;

    public Set<Adresse> getAdresses() {
        return adresses;
    }
}
```

```
@Entity
public class Adresse {

    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    @Column(name = "id")
    private Long id;

    private String ville;
```



# Relation bidirectionnelle de type One-to-many

```
@Entity(name = "ETUDIANT_TAB")
public class Etudiant {
    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    private Long id;
    private String nom;
    private String cin;

    @OneToMany(mappedBy = "etudiant", cascade = CascadeType.ALL)
    private Set<Adresse> adresses;

    @Entity
    public class Adresse {

        @Id
        @GeneratedValue(generator = "increment")
        @GenericGenerator(name = "increment", strategy = "increment")
        @Column(name = "id")
        private Long id;

        @ManyToOne
        @JoinColumn(name= "id_etudiant")
        private Etudiant etudiant;

        private String ville;
    }
}
```

# Relation many-to-many

## ■ Classe Etudiant

```
@ManyToMany(cascade = CascadeType.ALL)
@JoinTable(
    name="adresse_etudiant",
    joinColumns=@JoinColumn(name="id_etudiant"),
    inverseJoinColumns =@JoinColumn(name="id_adresse")
)
private Set<Adresse> adresses;
```

## ■ Classe Adresse

```
@ManyToMany
@JoinTable(name = "adresse_etudiant",
joinColumns = @JoinColumn(name = "id_adresse"),
inverseJoinColumns = @JoinColumn(name = "id_etudiant"))
private Set<Etudiant> etudiants;
```



# Mapping de l'héritage

## ■ Différentes stratégies pour le mapping:

Les bases de données relationnelles ne disposent pas de la notion de l'héritage: pour résoudre ce problème, la spécification JPA propose plusieurs stratégies pour faire la correspondance entre une hiérarchie d'héritage et les tables de la base de données:

- **Avec l'annotation `@MappedSuperclass`**
- **Stratégie à Table unique**: les entités de différentes classes avec une classe mère commune seront placées dans une seule table.
- **Table jointe** : chaque classe a sa table et l'interrogation d'une entité de sous-classe nécessite de joindre les tables.
- **Table par classe** : Chaque classe est associée à sa propre table. Toutes les propriétés d'une classe sont dans sa table, donc aucune jointure n'est requise.

# Héritage avec @MappedSuperclass

```
@MappedSuperclass  
public class Personne {
```

Notez que cette classe n'a plus d'annotation `@Entity`, car elle ne aura pas une table en base de données

Si nous utilisons cette stratégie, les classes mères ne peuvent pas contenir d'associations avec d'autres entités.

```
@Id  
@GeneratedValue(generator = "increment")  
@GenericGenerator(name = "increment", strategy = "increment")  
private Long id;  
private String nom;  
private String cin;  
private String prenom;
```

```
@Entity  
public class Etudiant extends Personne {  
  
    private String cne;  
  
    public String getCne() {  
        return cne;  
    }  
  
    public void setCne(String cne) {  
        this.cne = cne;  
    }  
  
}
```

Résultat dans MySQL:

v dbtestcours etudiant	
id	: bigint(20)
cin	: varchar(255)
nom	: varchar(255)
prenom	: varchar(255)
cne	: varchar(255)

# Stratégie Table unique (*Single Table*)

```
@Entity  
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)  
public class Personne {  
  
    @Id  
    @GeneratedValue(generator = "increment")  
    @GenericGenerator(name = "increment", strategy = "i  
    private Long id;  
    private String nom;  
    private String prenom;
```

L'identifiant des entités est également défini dans la super-classe.

```
@Entity  
public class Prof extends Personne {  
    private String cin;  
  
    public String getCin() {  
        return cin;  
    }  
  
    public void setCin(String cin) {  
        this.cin = cin;  
    }  
}
```

```
@Entity  
public class Etudiant extends Personne {  
  
    private String cne;  
  
    public String getCne() {  
        return cne;  
    }  
  
    public void setCne(String cne) {  
        this.cne = cne;  
    }  
}
```

# Stratégie Table unique (*Single Table*)

Étant donné que les enregistrements de toutes les entités seront dans la même table, Hibernate a besoin d'un moyen de les différencier. Par défaut, cela se fait via une colonne discriminante appelée DTYPE qui a le nom de l'entité comme valeur.

The screenshot shows the MySQL Workbench interface with the 'personne' table selected in the database browser on the left. The table structure is displayed with columns: DTYPE, id, nom, prenom, cne, and cin. Two rows of data are shown in the data grid, both belonging to the 'Etudiant' entity type (DTYPE = Etudiant). The first row has id=1, nom=boudaa, prenom=mohamed, cne=A11112, and cin=NULL. The second row has id=2, nom=boudaa, prenom=mohamed, cne=NULL, and cin=A11111.

DTYPE	id	nom	prenom	cne	cin
Etudiant	1	boudaa	mohamed	A11112	NULL
Etudiant	2	boudaa	mohamed	NULL	A11111

# Stratégie Table unique (*Single Table*)

Pour personnaliser la colonne discriminateur, nous pouvons utiliser l'annotation `@DiscriminatorColumn`:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="type_personne", discriminatorType = DiscriminatorType.INTEGER)
public class Personne {

    @Id
    @GeneratedValue(generator = "increment")
    @GenericGenerator(name = "increment", strategy = "increment")
    private Long id;
    private String nom;
    private String prenom;
```

```
@Entity
@DiscriminatorValue("2")
public class Prof extends Personne {
    private String cin;

    public String get Cin() {
        return cin;
    }

    public void set Cin(String cin) {
        this.cin = cin;
    }
}
```

```
@Entity
@DiscriminatorValue("1")
public class Etudiant extends Personne {

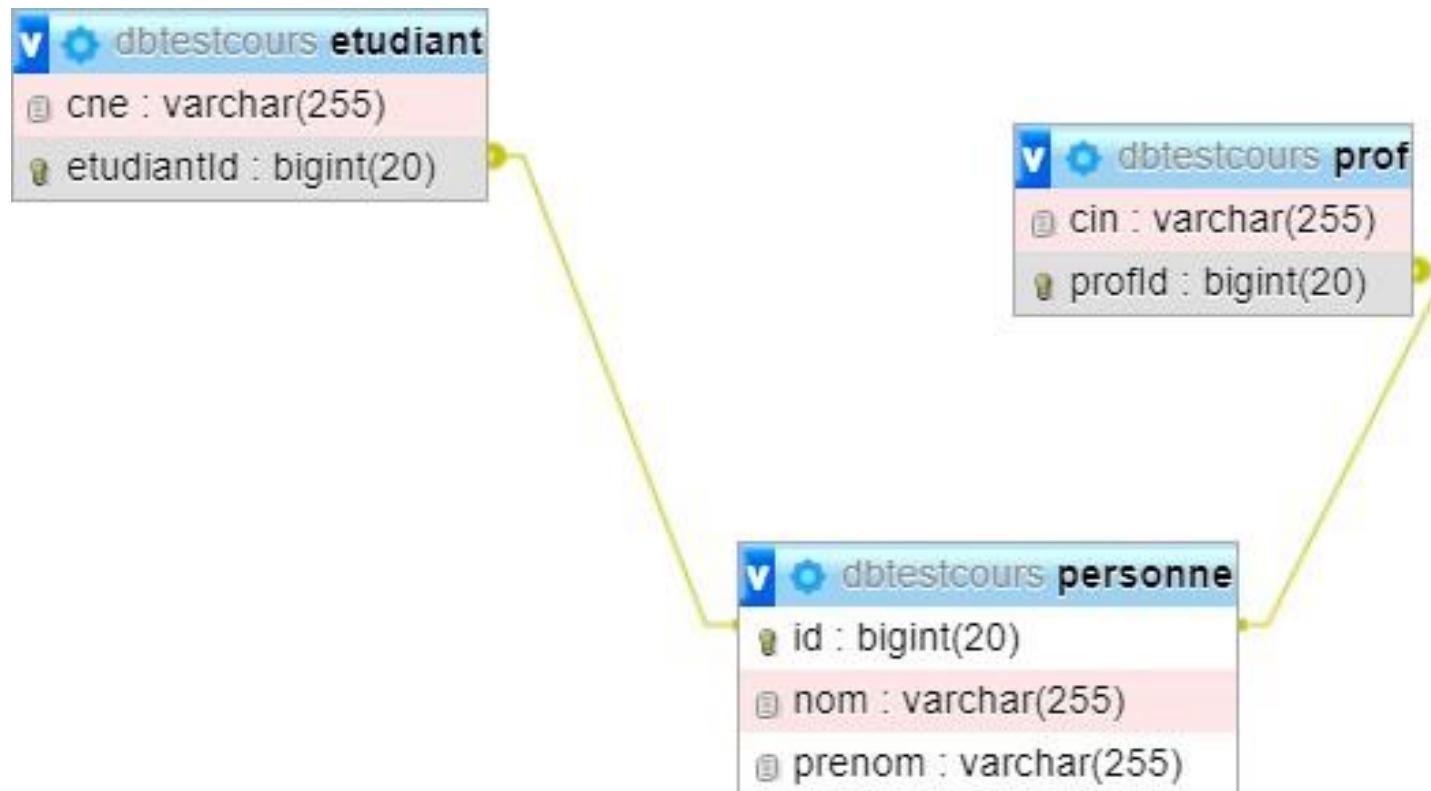
    private String cne;

    public String get Cne() {
        return cne;
    }

    public void set Cne(String cne) {
        this.cne = cne;
    }
}
```

type_personne	id	nom	prenom	cne	cin
1	1	boudaa	Mohamed	A11112	NULL
2	2	boudaa	Mohamed	NULL	A11111

# Stratégie avec table de jointure (*Joined Table*)



# Stratégie avec table de jointure (Joined Table)

```
@Entity  
@Inheritance(strategy = InheritanceType.JOINED)  
public class Personne {  
  
    @Id  
    @GeneratedValue(generator = "increment")  
    @GenericGenerator(name = "increment", strategy = "increment")  
    private Long id;  
    private String nom;  
    private String prenom;
```

```
@Entity  
@PrimaryKeyJoinColumn(name = "etudiantId")  
public class Etudiant extends Personne {  
  
    private String cne;
```

```
@Entity  
@PrimaryKeyJoinColumn(name = "profId")  
public class Prof extends Personne {  
    private String cin;
```

En utilisant cette stratégie, chaque classe de la hiérarchie est mappée à sa table. La seule colonne qui apparaît à plusieurs reprises dans toutes les tables est l'identifiant, qui sera utilisé pour les joindre en cas de besoin

# Stratégie table par classe (Table Per Class)

v dbtestcours personne	
!	id : bigint(20)
!	nom : varchar(255)
!	prenom : varchar(255)

v dbtestcours etudiant	
!	id : bigint(20)
!	nom : varchar(255)
!	prenom : varchar(255)
!	cne : varchar(255)

v dbtestcours prof	
!	id : bigint(20)
!	nom : varchar(255)
!	prenom : varchar(255)
!	cin : varchar(255)

# Stratégie table par classe (Table Per Class)

```
@Entity  
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)  
public class Personne {  
  
    @Id  
    @GeneratedValue(generator = "increment")  
    @GenericGenerator(name = "increment", strategy = "increment")  
    private Long id;  
    private String nom;  
    private String prenom;
```

```
@Entity  
public class Etudiant extends Personne {  
  
    private String cne;
```

```
@Entity  
public class Prof extends Personne {  
    private String cin;
```

Ce n'est pas très différent du simple mappage de chaque entité sans héritage. La différence est lors d'exécution d'une sélection sur la classe de base, elle retournera également tous les enregistrements des sous-classes en utilisant une instruction UNION en arrière-plan.

(Requêtes polymorphiques - *Polymorphic Queries*)

# Requêtes polymorphiques

- l'interrogation d'une classe de base récupérera également toutes les entités de sous-classes.
- Cela fonctionne également pour n'importe quelle super-classe ou interface, qu'il s'agisse d'une @MappedSuperclass ou non.

```
Session session = SessionFactoryBuilder.getSessionFactory().getCurrentSession();

Transaction tx = session.beginTransaction();
// Requete sur la super-classe Personne
String hqlQuery = "from Personne ";

Query<Etudiant> query = session.createQuery(hqlQuery);

List<Etudiant> list = query.getResultList();

tx.commit();

for(Personne it :list) {
    System.out.println(it);
}
```

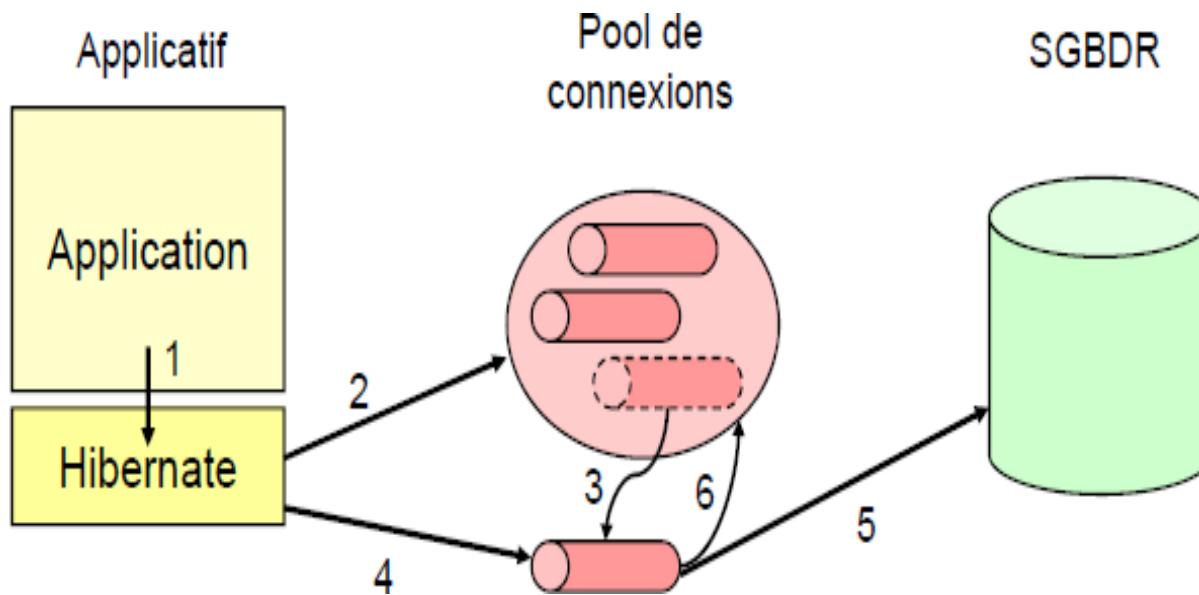
# Hibernate et Performance

Hibernate implémente et intègre des techniques et des outils pour garantir une bonne performance :

- Pool de connexion
- Lazy loading (chargement tardif)
- Systèmes de cache

# Pool de connexions JDBC

- Un pool de connexion est un cache de connexions de base de données gérée de telle sorte que les connexions peuvent être réutilisées ceci permet d'améliorer les performances d'exécution des commandes sur une base de données.

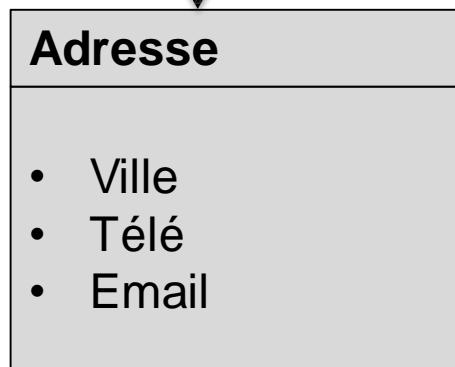
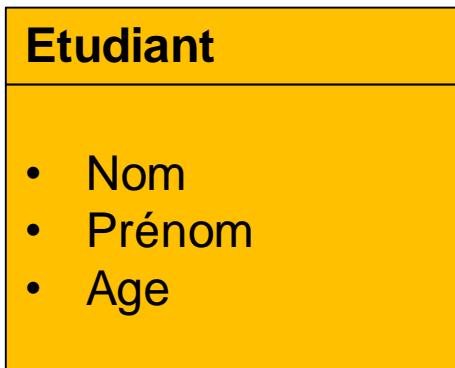


# Lazy loading

- La chargement à la demande (ou "*lazy loading*") est la stratégie native mise en œuvre par Hibernate pour le chargement optimal de données. Elle consiste à ne charger que le minimum de données, puis de générer une nouvelle requête SQL pour récupérer les données supplémentaires lorsque celles-ci seront demandées par le programme.

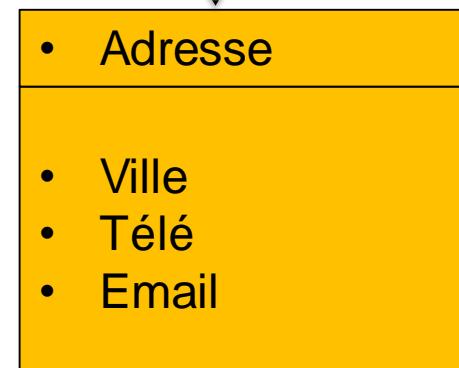
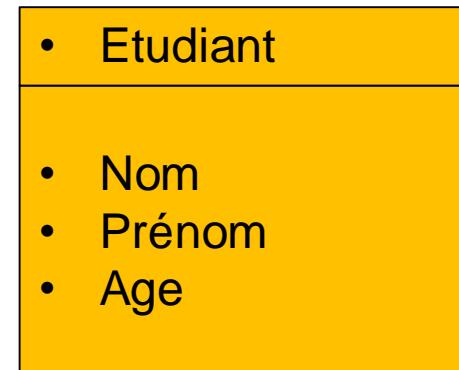
# Lazy loading

*Chargement juste de l'objet étudiant sans sa dépendance avec l'adresse*



***etudiant.getAdresse()***

*Hibernate effectue un SQL Select pour récupérer (tardivement) la dépendance « adresse »*



# Lazy loading

- Dans la pratique, Hibernate charge tous champs de la table principale, et les clés étrangères sont stockées sous forme simplifiée (seul l'ID est renseigné), ce que l'on nomme un **proxy**. Lorsque le programme essaiera d'accéder aux membres de ce proxy, Hibernate executera une requête SQL et récupérera les données nécessaires afin de le remplir...

# Lazy loading

- Précaution à prendre en compte en cas d'utilisation du chargement Lazy

1. Charger un objet X



*LazyInitializationException*

2. Fermer la session

3. Charger les dépendances de l'objet X

# Lazy loading

- Précaution à prendre en compte en cas d'utilisation du chargement Lazy

1. Charger un objet X

2. Charger les dépendances de X

3. Fermer la session

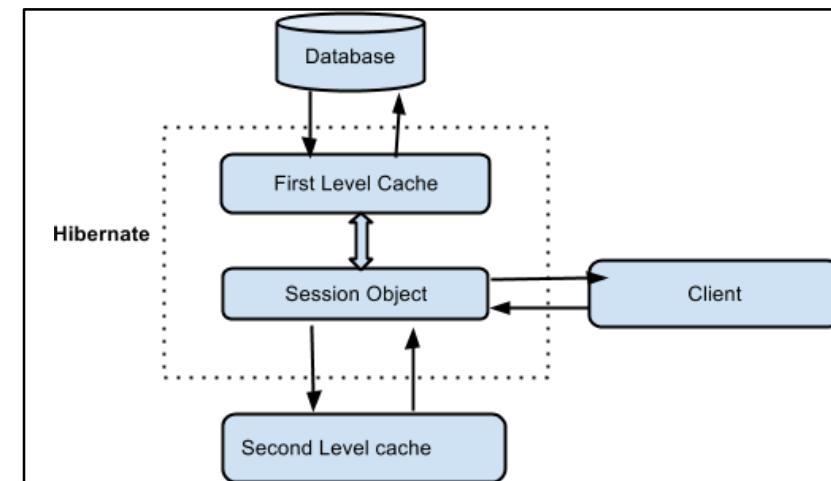
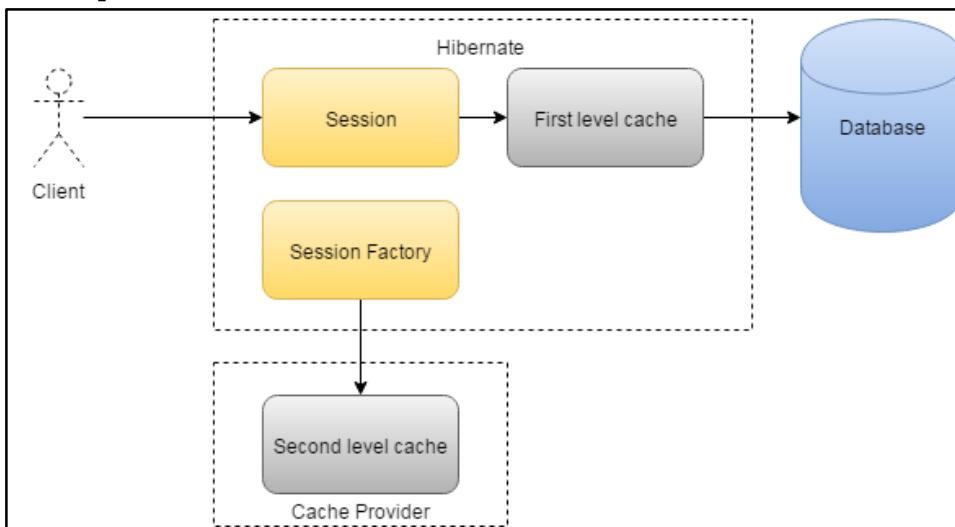
4. Accéder aux dépendances de l'objet X



# Les différents caches d'Hibernate

Hibernate fonctionne avec deux niveaux de cache. **Le cache de premier niveau (obligatoire)** est lié à la session Hibernate. Quand la session est fermée, le cache n'a plus d'existence. **Le cache de second niveau (optionnel)** est quant à lui lié à la session factory d'Hibernate. La portée de ce cache est la JVM. Les objets cachés sont donc visibles depuis l'ensemble des transactions.

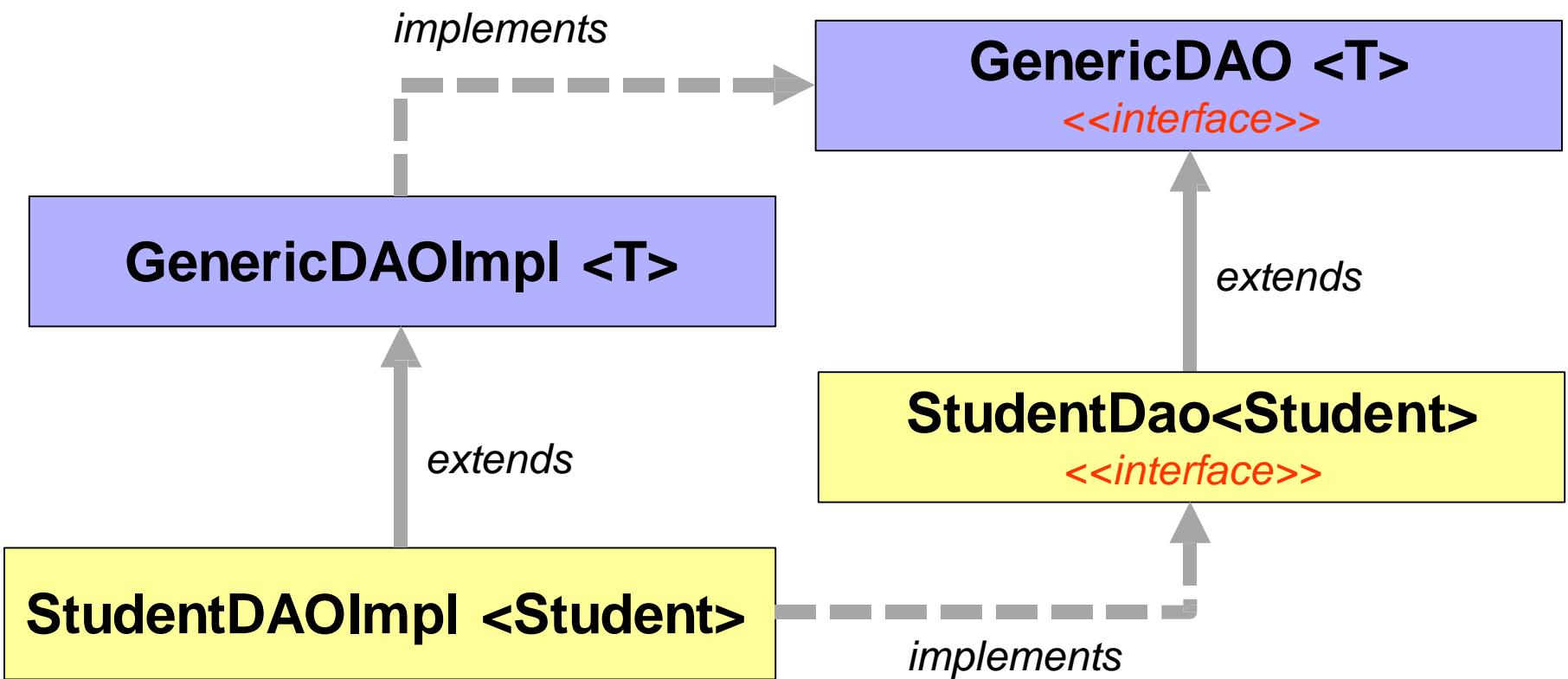
Il existe enfin un cache de requête qui permet de **cacher les résultats des requêtes exécutées**.



# Le pattern DAO

Le pattern DAO (Data Access Object) est un pattern structurel qui permet d'isoler la couche application/métier de la couche de persistance (généralement une base de données relationnelle, mais il peut s'agir de tout autre mécanisme de persistance) à l'aide d'une API abstraite. La fonctionnalité de cette API est de cacher à l'application toutes les complexités impliquées dans l'exécution des opérations CRUD dans le mécanisme de stockage sous-jacent. Cela permet aux deux couches d'évoluer séparément sans rien savoir l'une de l'autre.

# Le pattern DAO et généricité



# Persistance avec JPA

## Fichier persistence.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence_2_0.xsd"
              version="2.0">
    <persistence-unit name="com.ensah.gs_etudiants" transaction-type="RESOURCE_LOCAL">
        <description>Hibernate EntityManager Demo</description>
        <class>com.bo.Etudiant</class>
        <exclude-unlisted-classes>true</exclude-unlisted-classes>

        <properties>
            <property name="hibernate.dialect"
value="org.hibernate.dialect.MariaDB103Dialect" />
            <property name="hibernate.hbm2ddl.auto" value="update"/>
            <property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver" />
            <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost/dbTestCours" />
            <property name="javax.persistence.jdbc.user" value="root" />
            <property name="javax.persistence.jdbc.password" value="" />
        </properties>
    </persistence-unit>
</persistence>
```

# Persistance avec JPA

Hibernate	JPA
Hibernate est un Framework implémentant JPA et possède d'autres fonctionnalités qui lui sont propres	JPA est une spécification qui définit la gestion des données relationnelles dans les applications Java.
Session (n'est pas thread-safe)	EntityManager (n'est pas thread-safe)
Transaction	EntityTransaction
SessionFactory (thread-safe)	EntityManagerFactory (thread-safe)
Hibernate Query Language (HQL)	Java Persistence Query Language (JPQL)

# Persistance avec JPA

- Quelques méthodes de manipulation des entités via EntityManager
  - ✓ persist : Permet de persister l'état d'une entité en base de données
  - ✓ merge : Fusionner l'état de l'entité donnée dans le contexte de persistance actuel.
  - ✓ remove : Supprime une instance d'une entité
  - ✓ find : Rechercher par clé primaire.
  - ✓ close : ferme l'entityManager.
  - ✓ getTransaction : retourne l'objet EntityTransaction permettant la gestion des transactions.

***Pour plus d'informations voir la documentation:***

<https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html>

# Spring Data JPA

*Spring Data JPA, part of the larger Spring Data family, makes it easy to easily implement JPA based repositories. This module deals with enhanced support for JPA based data access layers. It makes it easier to build Spring-powered applications that use data access technologies.*

Source : <https://spring.io/projects/spring-data-jpa>

# Spring Data JPA

En plus des dependences essentielles du Framework Spring il faut ajouter la dépendance spring data jpa:

```
<dependency>
<groupId>org.springframework.data</groupId>
<artifactId>spring-data-jpa</artifactId>
<version>2.5.0</version>
</dependency>
```

# Spring Data JPA

- L'écriture d'un DAO offrant les opérations CRUD avec Spring data jpa est réduite à l'écriture d'une interface qui hérite de l'interface **JpaRepository** de Spring data jpa:

```
package com.ensah.core.dao;

import org.springframework.data.jpa.repository.JpaRepository;

import com.ensah.core.bo.Person;

public interface IPersonDao extends JpaRepository<Person, Long>, IPersonDaoCustom {

}
```

# Spring Data JPA

- Pour personnaliser le DAO il suffit d'écrire une autre interface en respectant une convention de nommage et puis implémenter les méthodes personnalisées dans une classe implementant cette interface. Ces règles de nommage dépendent des versions de Spring data. Pour plus d'informations :  
<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.custom-implementations>

```
package com.ensah.core.dao;
import java.util.List;
import com.ensah.core.bo.Person;
public interface IPersonDaoCustom {
    List<Person> getPersonsByFirstName(String firstName);
    //d'autres méthodes
}
```

La convention de nommage utilisée est : \${Original Repository name}Custom.

```
package com.ensah.core.dao;
import java.util.List;
public class PersonDaoImpl implements IPersonDaoCustom {
    @Autowired
    private EntityManager entityManager;
    @Override
    public List<Person> getPersonsByFirstName(String firstName) {
        Query query = entityManager.createNativeQuery(
            "SELECT em.* FROM Person WHERE firstname LIKE ?", Person.class);
        query.setParameter(1, firstName + "%");
        return query.getResultList();
    }
    //d'autres méthodes
}
```

# Autres techniques de persistance

- Intégration de Spring & Hibernate et utilisation de HibernateTemplate
- Utilisation de Hibernate avec Spring Boot
- Utilisation de Hibernate avec Spring data JPA
- Utilisation de JPA avec Spring data JPA

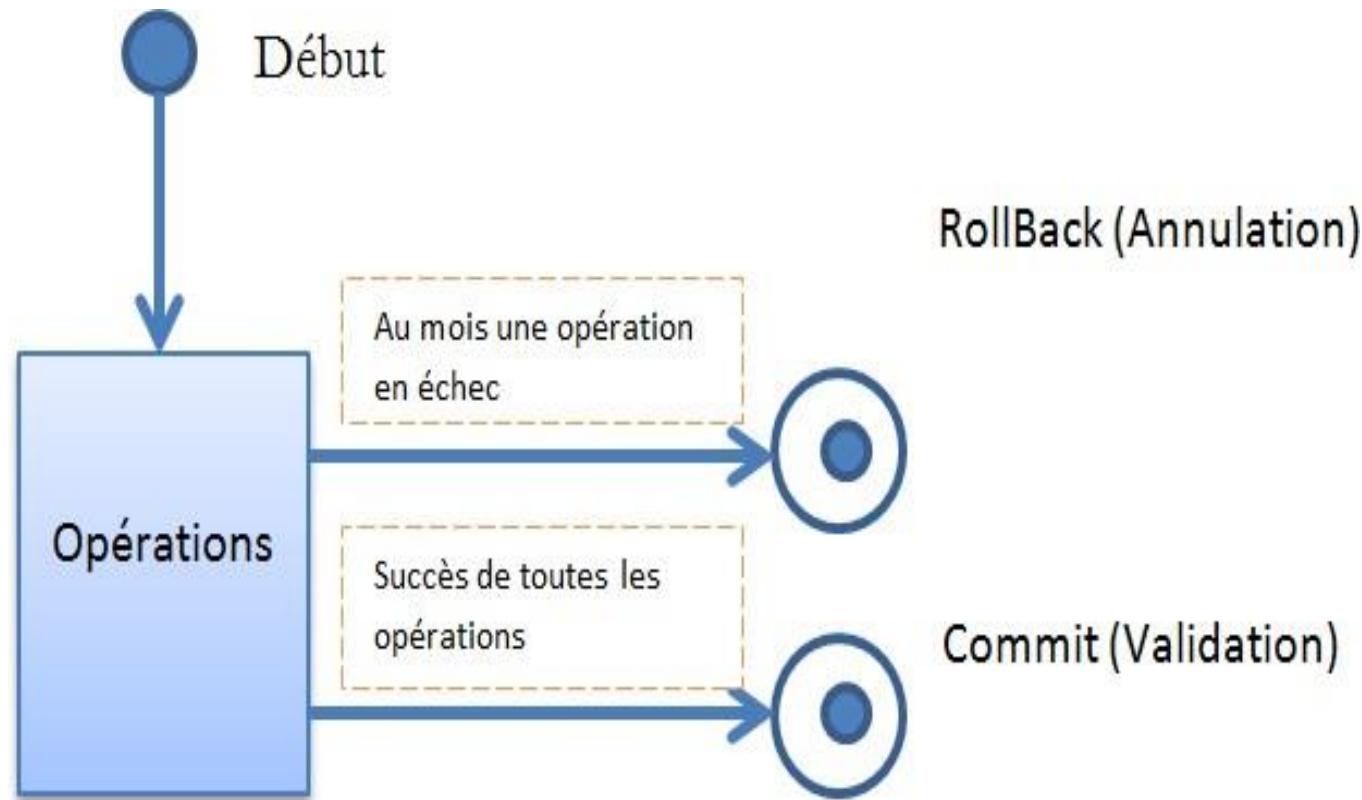
*Ces techniques seront étudiées  
en Travaux Pratiques*

# Gestion des transactions

- Une transaction est un ensemble indivisible d'opérations dans lequel tout aboutit ou rien n'aboutit.
- La gestion des transactions est un élément essentiel dans les applications d'entreprises car elle permet de garantir l'intégrité et la cohérence des données.
- Sans les transactions les données et les ressources pourraient être endommagées et laissés dans un état incohérent.
- Oublier de gérer les transactions dans une application n'engendre pas nécessairement de dysfonctionnement visible. Cependant, cet oubli ne permettant pas de garantir la consistance des données manipulées, des erreurs délicates à détecter peuvent survenir.
- La notion de transaction est donc primordiale et doit être mise en œuvre dans toute application réalisant des mises à jour dans des sources de données.

# Gestion des transactions

- Une transaction doit vérifier les propriétés fondamentales suivantes, communément désignées sous l'acronyme **ACID** (**atomicité, consistance, isolation, durabilité**) :



# Gestion des transactions

## ■ Atomicité :

Une transaction est une opération atomique constituée d'une suite d'opérations. Cette propriété garantit que toutes les actions sont entièrement exécutées ou qu'elles n'ont aucun effet.

# Gestion des transactions

## ■ Cohérence :

la transaction laisse toujours les données dans un état cohérent.

Cette propriété assure que chaque transaction amènera le système d'un état valide à un autre état valide. Tout changement à la base de données doit être valide selon toutes les règles définies.

# Gestion des transactions

## ■ Isolation :

Puisque plusieurs transactions peuvent manipuler le même jeu de données au même moment, chaque transaction doit être isolée des autres afin d'éviter la corruption des données

La propriété d'isolation **assure que l'exécution simultanée de transactions produit le même état que celui qui serait obtenu par l'exécution en série des transactions.** Chaque transaction doit s'exécuter en isolation totale : si T1 et T2 s'exécutent simultanément, alors chacune doit demeurer indépendante de l'autre.

# Gestion des transactions

## ■ Durabilité :

Dès lors qu'une transaction est terminée, les résultats doivent être durables dans le temps. C'est-à-dire les modifications sont définitives et entièrement visibles par le reste des applications qui utilisent la même source de données.

# Types de transactions :

Il existe deux grands types de transactions, les transactions locales et les transactions globales.

- **Transactions locales.** Adaptées lorsqu'une seule ressource transactionnelle est utilisée.
- **Transactions globales.** Utilisables si une ou plusieurs ressources sont présentes. À partir de deux ressources, nous sommes toujours en présence de transactions globales. En cas d'utilisation des transactions globales, le gestionnaire de transactions est externalisé par rapport aux ressources et est capable de dialoguer avec elles grâce à des interfaces normalisées.
- Les transactions globales sont cependant beaucoup plus difficiles à mettre en œuvre. En effet, ce type de transaction nécessite la mise en œuvre d'un gestionnaire de transactions externe. Ce dernier est souvent fourni par le serveur d'applications.
- Spring masque la complexité derrière les différents framework de gestion de transactions avec une API générique.

# Problèmes liés aux accès concourants :

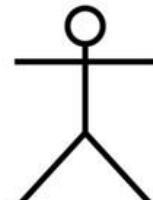
- Lecture sale
- Lecture non reproductible
- Lecture fantôme

# Problèmes liés aux accès concourants :

- **Lecture sale:** Pour deux transactions T1 et T2, T1 lit un champ qui a été mis à jour par T2 mais pas encore validé. Plus tard, si T2 est annulée, la valeur du champ lu par T1 était en réalité une valeur temporaire désormais invalide.

# Lecture sale

Etape 1

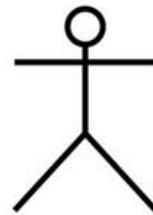


Acteur 1

Début de la transaction 1  
Récupération de l'entité E  
Modification de E

Entité  
E

Etape 2

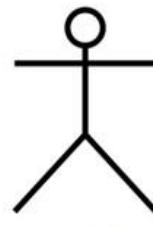


Acteur 2

Début de la transaction 2  
Récupération de l'entité E

Entité  
E

Etape 3



Acteur 1

Annulation de la  
transaction 1

Entité  
E

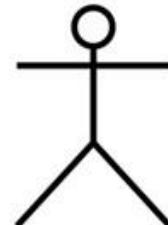
- L'acteur 2 travaille sur des données fausses non validés par l'acteur 1 (données sales)

# Lecture non reproductible

- **Lecture non reproductible:** Pour deux transactions T1 et T2, T1 lit un champ et T2 met ensuite ce champ à jour. Plus tard, si T1 lit à nouveau le même champ, la valeur est différente.

# Lecture non reproductible

Etape 1

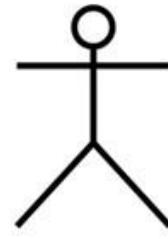


Acteur 1

Début de la transaction 1  
Récupération de l'entité E

Entité  
E

Etape 2

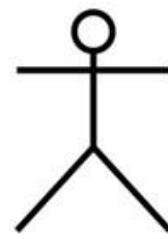


Acteur 2

Début de la transaction 2  
Récupération et  
modification de l'entité E  
commit de la transaction 2

Entité  
E

Etape 3



Acteur 1

Toujours dans la  
transaction 1 on récupère  
la même entité E

Entité  
E

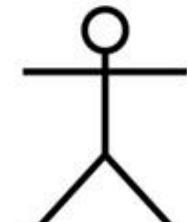
- Dans ce cas au sein d'une même transaction la lecture successive d'une même entité donne deux résultats différents.

# Lecture fantôme

- **Lecture fantôme:** Pour deux transactions T1 et T2, T1 lit quelques lignes d'une table, puis T2 insère de nouvelles lignes dans la table. Plus tard, si T1 lit à nouveau la même table, il existe des lignes supplémentaires.

# Lecture fantôme

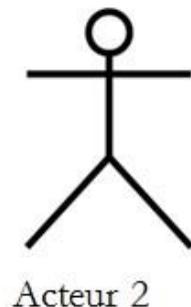
Etape 1



Début de la transaction 1  
Récupération des entités E  
répondant à un jeu de  
critères X

Entité  
E

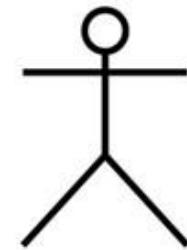
Etape 2



Début de la transaction 2  
Insertion de nouvelles  
entités E répondant au jeu  
de critères X  
commit de la transaction2

Entité  
E

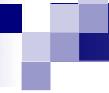
Etape 3



Réexécution de la  
recherche selon le même  
jeu de critères X

Entité  
E

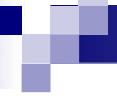
- Dans ce cas une même recherche fait apparaître ou disparaître des entités.



# Niveaux d'isolation transactionnelle

- En fonction du type de l'application, il existe plusieurs façons de gérer les collisions. Au niveau de la connexion JDBC, le niveau d'isolation transactionnelle permet de spécifier le comportement de la transaction selon la base de données utilisée.
- Il existe en standard quatre niveaux d'isolation transactionnelle

Niveau d'isolation	Lecture sale	Lecture non répétable	Lecture fantôme
TRANSACTION_READ_UNCOMMITTED	OUI	OUI	OUI
TRANSACTION_READ_COMMITTED	NON	OUI	OUI
TRANSACTION_REPEATABLE_READ	NON	NON	OUI
TRANSACTION_SERIALIZABLE	NON	NON	NON



# Niveaux d'isolation transactionnelle

- En théorie, les transactions doivent être totalement isolées les unes des autres (c'est-à dire sérialisables) pour éviter les problèmes précédents. Cependant, un tel niveau d'isolation a un impact important sur les performances car les transactions s'exécutent alors l'une après l'autre. Dans la pratique, les transactions peuvent s'exécuter à des niveaux d'isolation plus faibles de manière à améliorer les performances.

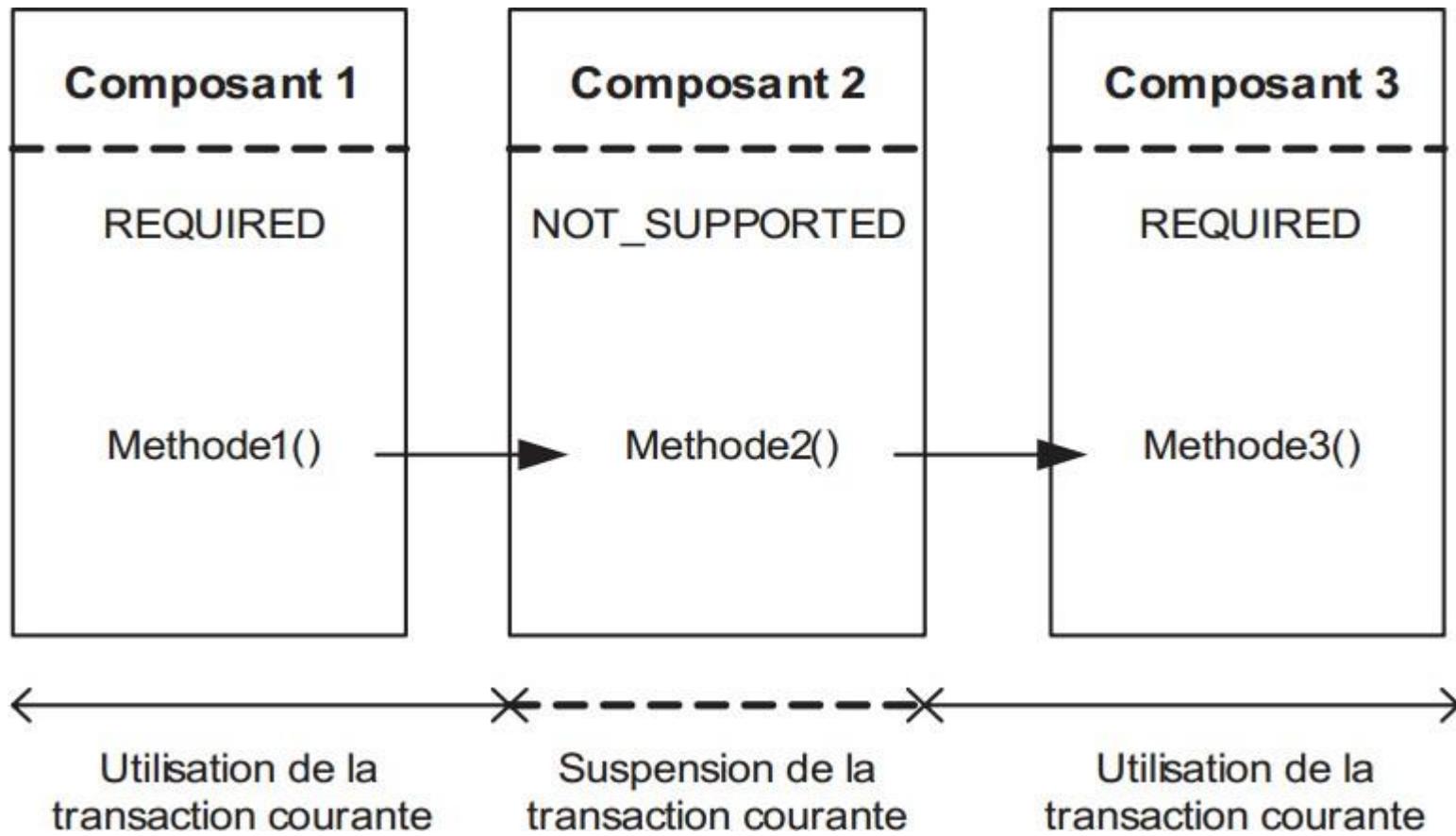
# Comportements transactionnels: propagation

Mot-clé de comportement transactionnel	Description
REQUIRED	La méthode doit forcément être exécutée dans un contexte transactionnel s'il existe. S'il n'existe pas lors de l'appel, il est créé.
SUPPORTS	La méthode peut être exécutée dans un contexte transactionnel s'il existe. Dans le cas contraire, la méthode est tout de même exécutée, mais hors d'un contexte transactionnel.
MANDATORY	La méthode doit forcément être exécutée dans un contexte transactionnel. Si tel n'est pas le cas, une exception est levée.
REQUIRES_NEW	La méthode impose la création d'une nouvelle transaction pour la méthode.
NOT_SUPPORTED	La méthode ne supporte pas les transactions. Si un contexte transactionnel existe lors de son appel, celui-ci est suspendu.
NEVER	La méthode ne doit pas être exécutée dans un contexte transactionnel. Si tel est le cas, une exception est levée.
NESTED	La méthode est exécutée dans une transaction imbriquée si un contexte transactionnel existe lors de son appel.

# Comportements transactionnels: propagation

Type de comportement transactionnel	Transaction initiale	Transaction utilisée
REQUIRED	Aucune T1	T1 T1
SUPPORTS	Aucune T1	Aucune T1
MANDATORY	Aucune T1	Erreur T1
REQUIRES_NEW	Aucune T1	T1 T2
NOT_SUPPORT	Aucune T1	Aucune Aucune
NEVER	Aucune T1	Aucune Erreur
NESTED	Aucune T1	T1 T1 (imbriquée)

# Comportements transactionnels: propagation



# La notion de verrou

- La notion de verrou est implémentée pour fournir des solutions aux différents types de collisions. Il existe deux possibilités, le **verrouillage pessimiste** et le **verrouillage optimiste**.

# Verrouillage pessimiste

- Ce mécanisme de verrouillage fort est géré directement par le système de stockage des données. Pendant toute la durée du verrou, aucune autre application ou fil d'exécution ne peut accéder à la donnée. Pour les bases de données relationnelles, cela se gère directement au niveau du langage SQL. À l'image d'Hibernate, plusieurs frameworks facilitent l'utilisation de ce type de verrou. Une requête SQL de type `select for update` est alors utilisée. Ce verrouillage particulièrement restrictif peut impacter les performances des applications. En effet, ces dernières ne peuvent accéder à l'enregistrement tant que le verrou n'est pas levé.

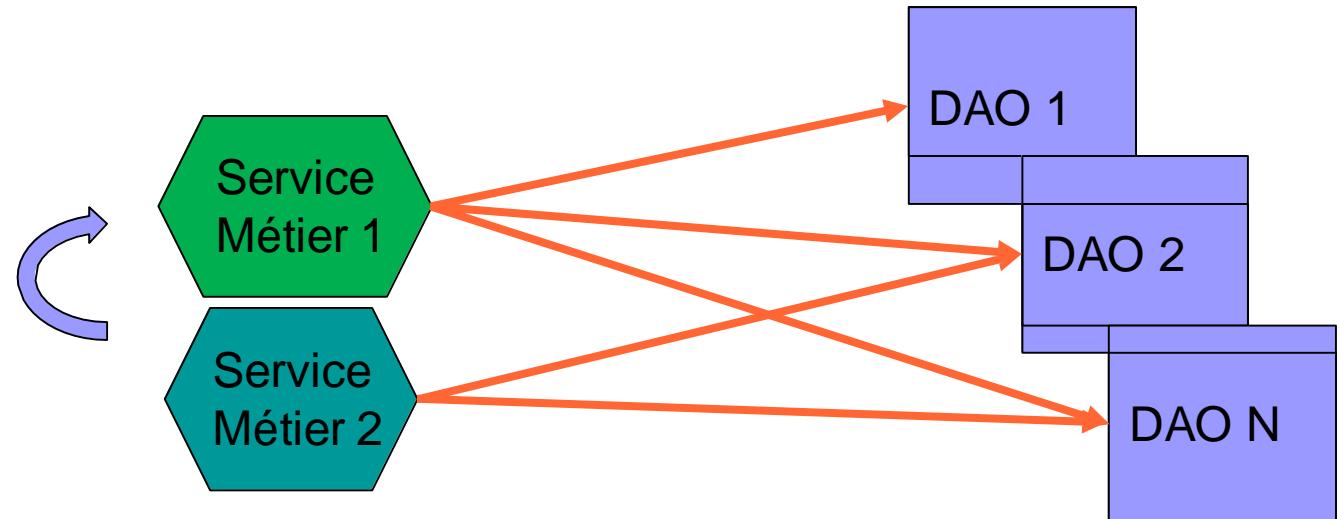
# Verrouillage optimiste :

- Ce type de verrouillage adopte la logique de détection de collision. Son principe est qu'il peut être acceptable qu'une collision survienne, à condition de pouvoir la détecter et la résoudre.
- Ce type de verrouillage doit être implémenté dans l'application elle-même. Il ne nécessite pas de verrou dans le système de stockage des données. Pour les bases de données relationnelles, il est généralement implémenté en ajoutant une colonne aux différentes tables impactées. Cette colonne représente une version ou un indicateur de temps indiquant l'état de l'enregistrement lorsqu'il est lu. De ce fait, si cet état change entre la lecture et la modification, nous nous trouvons dans le cas d'un accès concourant.

# Gestion de la démarcation

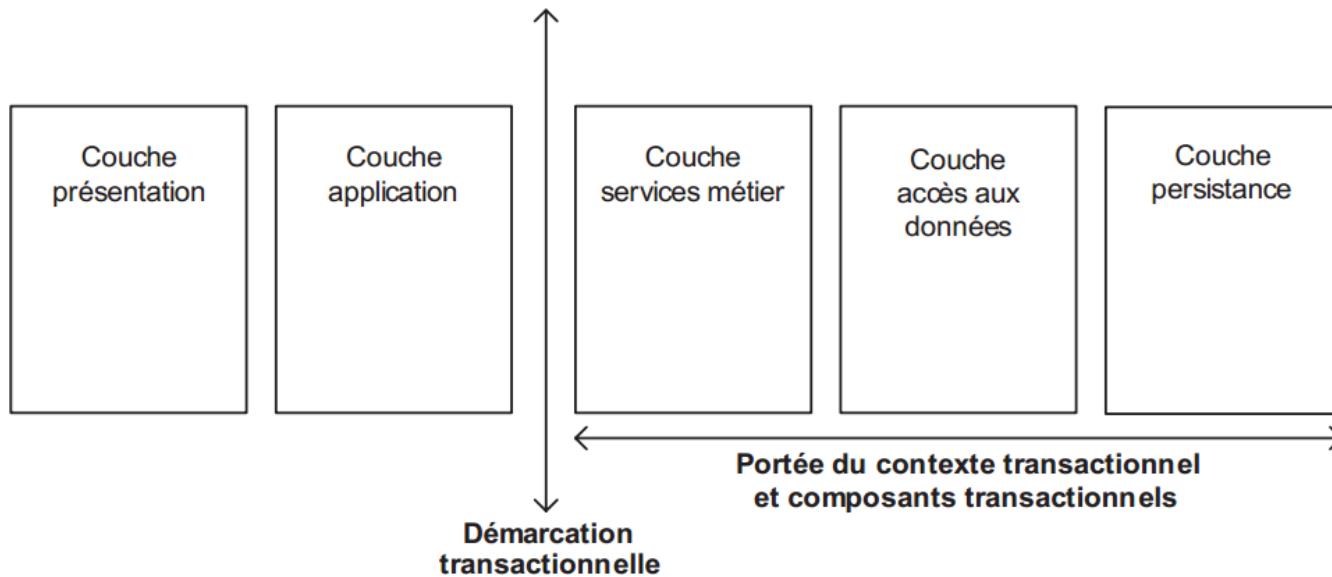
- Généralement les transactions doivent être gérées dans la couche services métier et non pas dans la couche DAO. En effet:
  - Dans un service on pourra avoir besoin d'exécuter plusieurs méthodes DAO au sein d'une même transaction.
  - Les DAOs ne seront pas réutilisables si nous gérions la transaction au niveau DAO

Le support des transactions doit de plus être suffisamment flexible pour permettre de gérer les appels entre services.



La démarcation transactionnelle doit être réalisée au niveau des services métier. Ces derniers peuvent s'appuyer sur plusieurs composants d'accès aux données. Plusieurs appels à des méthodes de ces composants sous-jacents peuvent donc être réalisés dans une même transaction.

# Gestion de la démarcation



- La démarcation consiste à spécifier le commencement et la fin de la transaction.
- La définition de types de comportement transactionnel rend ce support flexible et déclaratif. Spring implémente ces stratégies dans sa gestion des transactions.
- Avec l'API générique de démarcation de Spring on peut éviter d'utiliser une connexion JDBC ou une session d'un outil d'ORM dans la couche service métier (c'est un anti-pattern).

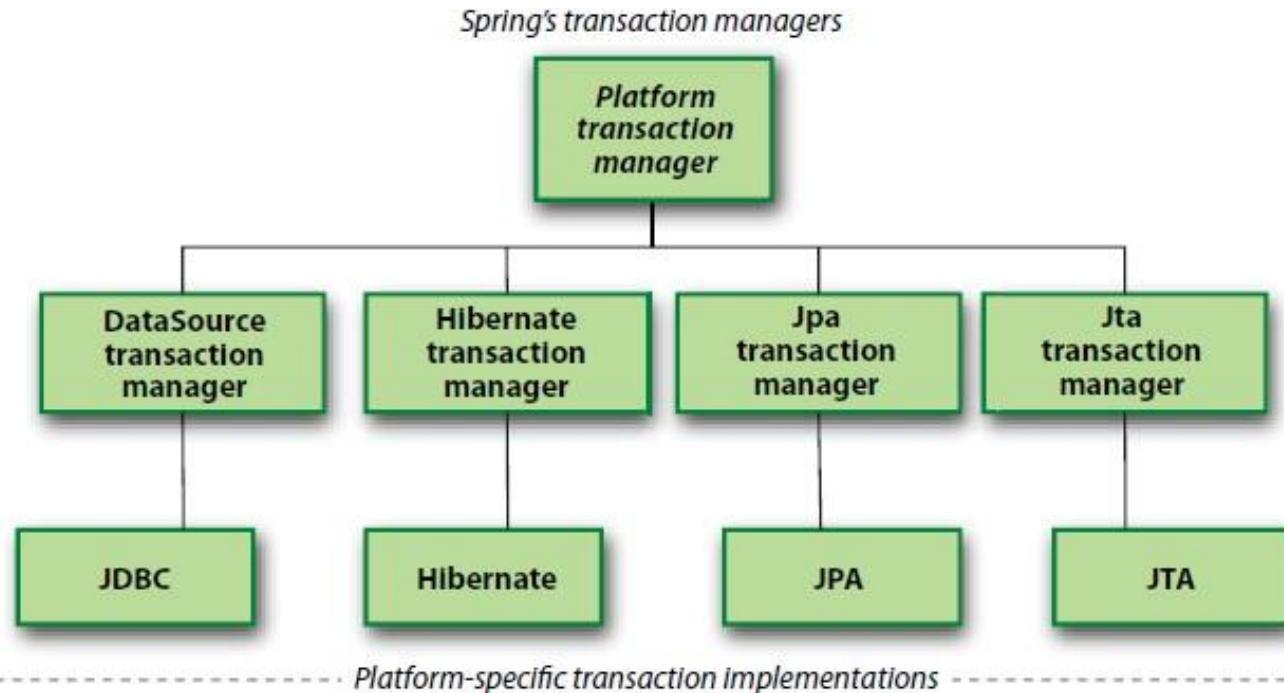
# Configuration de la gestion des Transactions avec Spring

En utilisant une configuration par code Java il faut ajouter l'annotation **@EnableTransactionManagement** pour activer la prise en charge transactionnelle:

```
@Configuration  
@EnableTransactionManagement  
public class AppConfig {  
    ...  
}
```

# Configuration de la gestion des Transactions avec Spring

Le module Spring Transaction essaie de simplifier cette situation en utilisant une interface unique pour la gestion des transactions : **PlatformTransactionManager**. Le module fournit ensuite plusieurs implémentations selon la technologie sous-jacente utilisée.



# Configuration de la gestion des Transactions avec Spring

## ■ Exemple de configuration du Gestionnaire de transaction pour Hibernate

```
@Configuration  
@EnableTransactionManagement  
public class AppConfig {  
    @Bean  
    @Autowired  
    public PlatformTransactionManager transactionManager(final  
        SessionFactory sessionFactory) {  
        final HibernateTransactionManager txManager = new  
            HibernateTransactionManager();  
        txManager.setSessionFactory(sessionFactory);  
        return txManager;  
    }  
}
```

# Configuration de la gestion des Transactions avec Spring

## ■ Exemple de configuration du Gestionnaire de transaction pour JPA

```
@Configuration  
@EnableTransactionManagement  
public class AppConfig {  
    @Bean  
    @Autowired  
    public PlatformTransactionManager transactionManager(final  
        EntityManagerFactory emf){  
        JpaTransactionManager tm = new JpaTransactionManager();  
        tm.setEntityManagerFactory(emf);  
        return tm;  
    }  
}
```

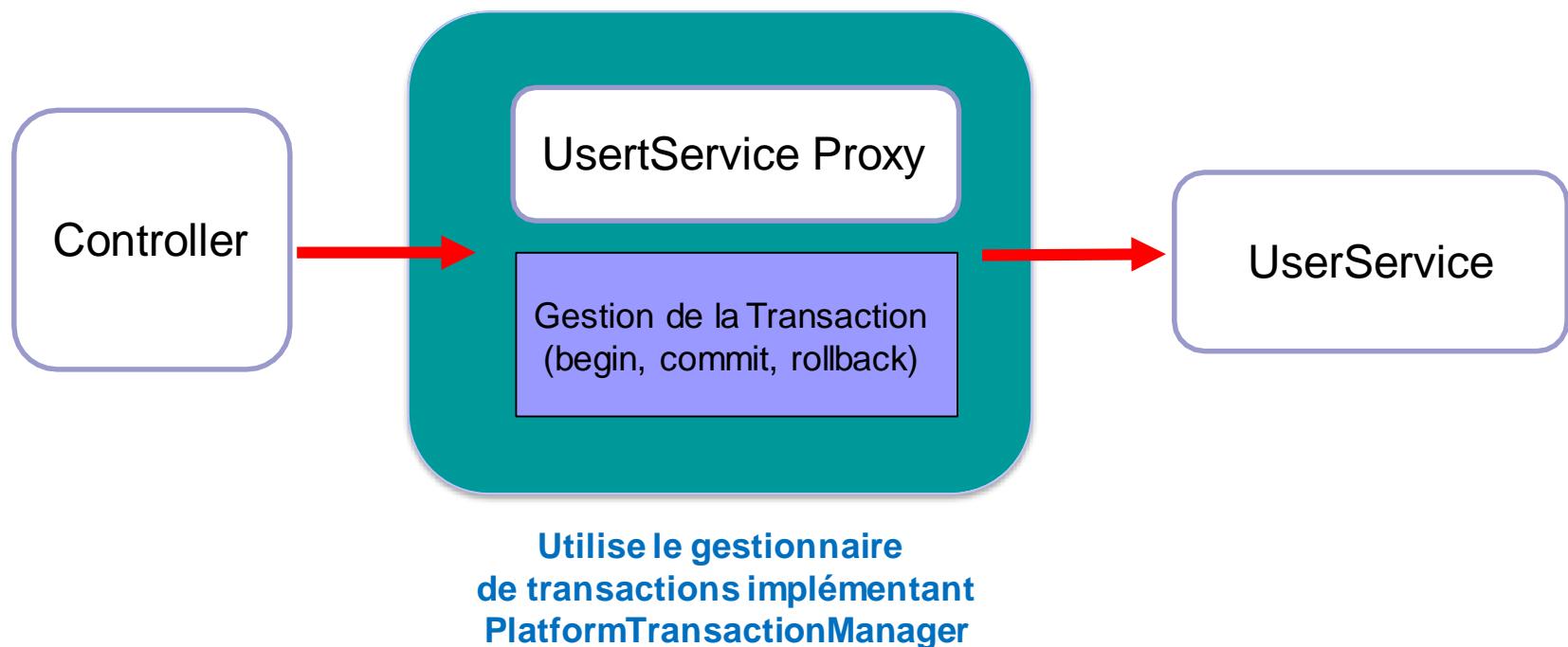
# Configuration de la gestion des Transactions avec Spring

- Annoter les classes services avec `@transactional`

```
public class UserService {  
  
    @Transactional  
    public Long registerUser(User user) {  
        //Code de la méthode service  
        // Généralement, les méthodes services peuvent  
        //exécuter plusieurs méthodes de différents DAOs  
    }  
}
```

# Configuration de la gestion des Transactions avec Spring

- Spring utilise des proxies en background



# Configuration de la gestion des Transactions avec Spring

- Configurer le niveau de propagation avec @transactional

@Transactional(propagation = Propagation.REQUIRED) (**Valeur par défaut**)

@Transactional(propagation = Propagation.REQUIRES\_NEW)

@Transactional(propagation = Propagation.SUPPORTS)

@Transactional(propagation = Propagation.MANDATORY)

@Transactional(propagation = Propagation. NOT\_SUPPORTED)

@Transactional(propagation = Propagation. NEVER)

@Transactional(propagation = Propagation. NESTED)

# Configuration de la gestion des Transactions avec Spring

## ■ Configurer le niveau d'isolation avec `@Transactional`

//Utiliser le niveau d'isolement par défaut du SGBD sous-jacent..

`@Transactional(isolation = Isolation.DEFAULT) (Valeur par défaut)`

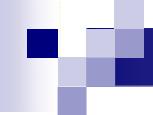
//Les autres niveaux correspondent aux niveaux d'isolement vus précédemment

`@Transactional(isolation = Isolation.READ_COMMITTED)`

`@Transactional(isolation = Isolation.READ_UNCOMMITTED)`

`@Transactional(isolation = Isolation.REPEATABLE_READ)`

`@Transactional(isolation = Isolation.SERIALIZABLE)`



# Développement des applications REST avec Spring

# Qu'est ce qu'un Web Service?

## ■ Les Web Services

- ❑ Un service Web est une « unité logique applicative» accessible en utilisant les protocoles standard d'Internet
- ❑ Il est offerte à une application cliente (consommateur) via une interface de service.
- ❑ Il existe généralement deux types de Web Services:
  - ✓ Web Services REST
  - ✓ Web Services SOAP

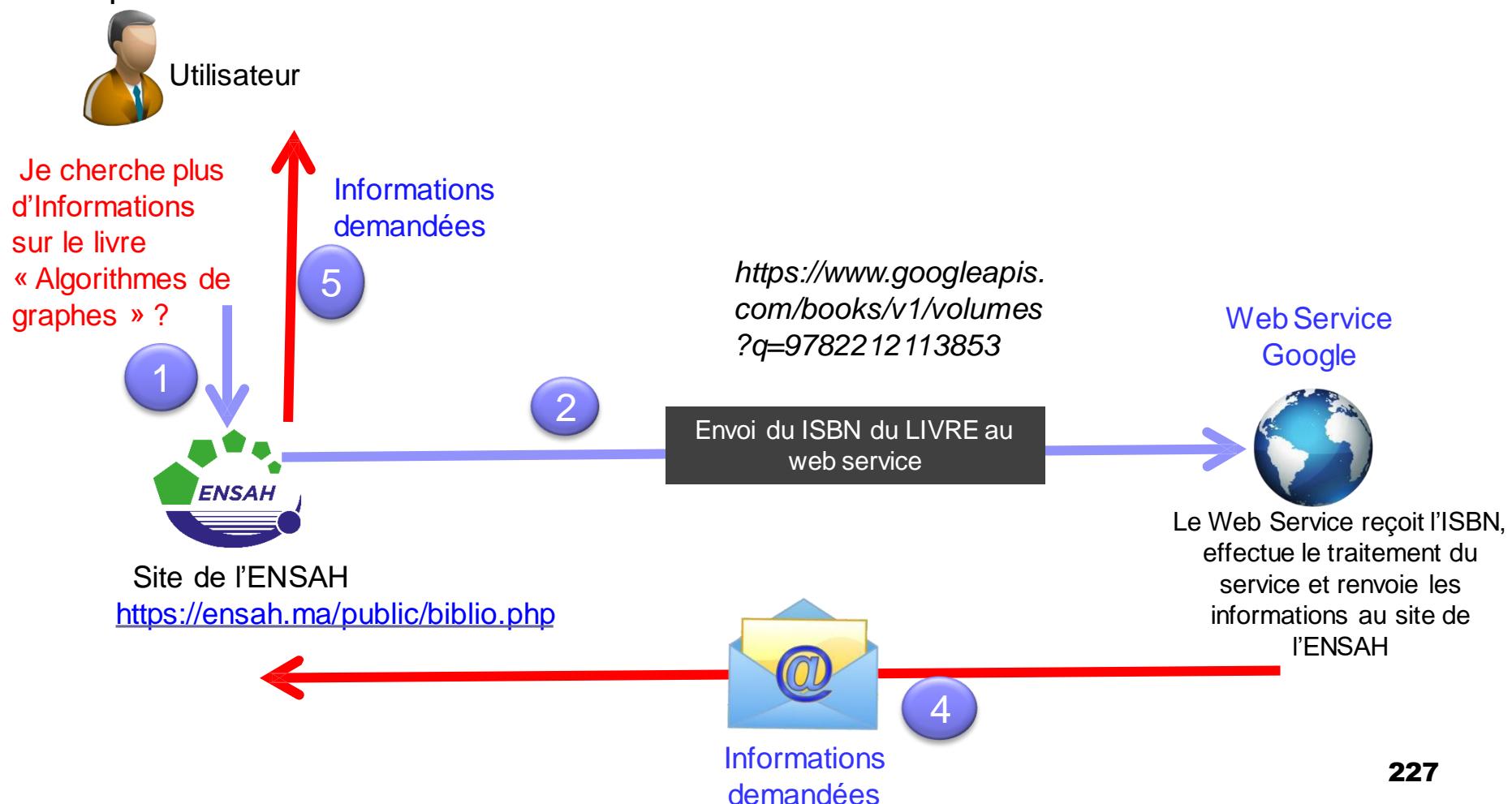
## ■ SOA (Service Oriented Architecture)

- ❑ Bien qu'elle soit souvent confondue avec les services web, SOA est une forme architecture reposant principalement sur des applications orientées services qui peuvent être implémentées à l'aide de services web, mais également avec d'autres technologies

# Qu'est ce qu'un Web Service?

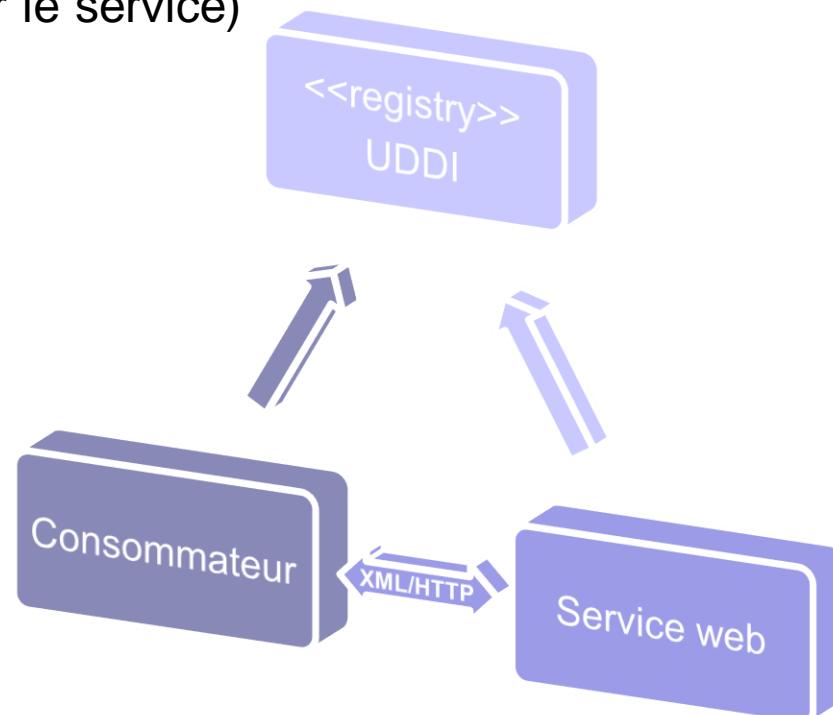
## ■ Exemple d'un Web Service REST

Le catalogue en ligne de la bibliothèque disponible sur le site de l'ENSAH <https://ensah.ma/public/biblio.php> consomme le web le web service de google Books pour récupérer des informations sur des livres.

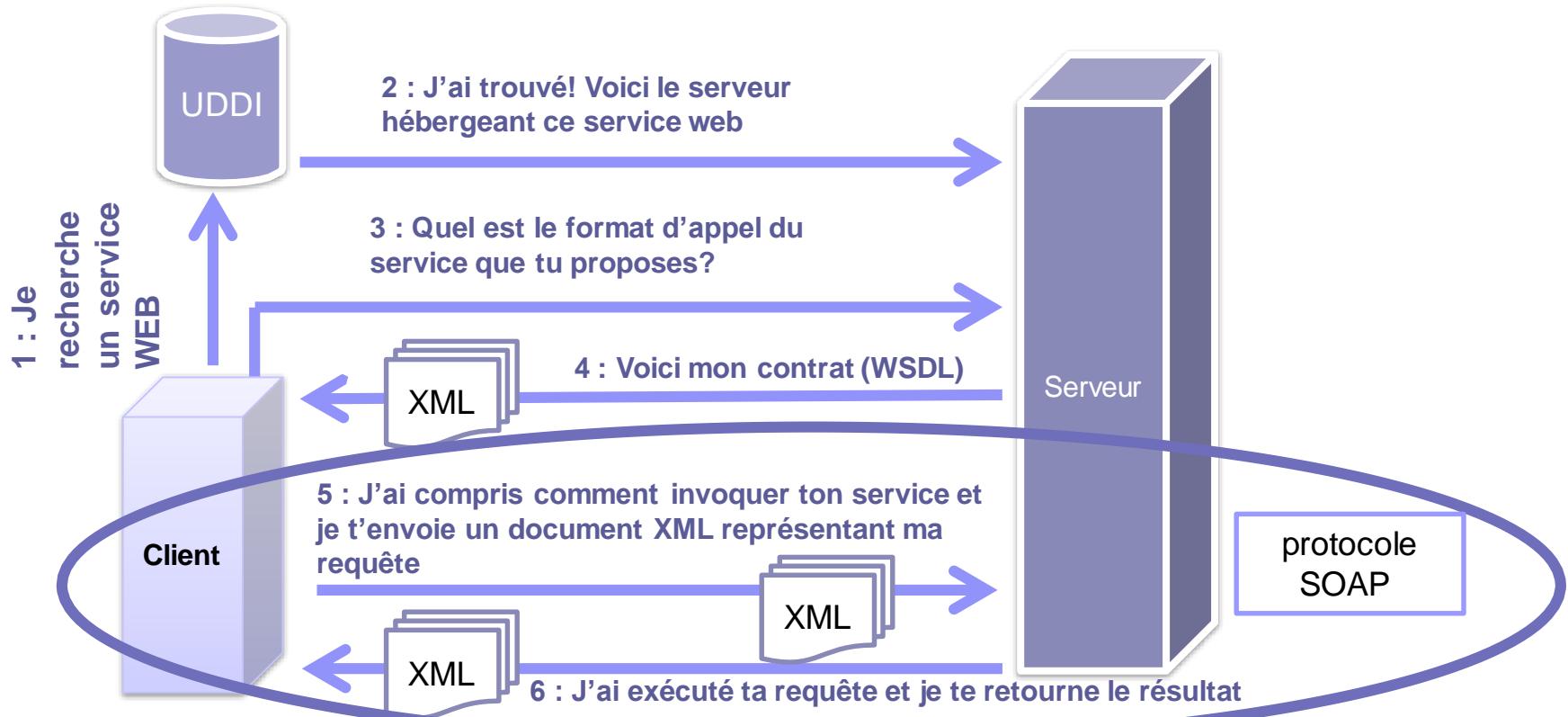


# Trouver et consommer un Web Service SOAP

Le service peut éventuellement enregistrer son interface dans une base de registres (**Annuaire UDDI**) afin qu'un consommateur puisse le trouver (ceci ressemble aux pages jaunes) . Une fois que le consommateur connaît **l'interface du service et le format du message (WSDL)**, il peut envoyer une requête (Demander un service) et recevoir une réponse (consommer le service)



# Trouver et consommer un Web Service SOAP



# Technologies et protocoles de transport des services web SOAP

**UDDI (Universal Description Discovery and Integration)** : est une base de registres et un mécanisme de découverte qui ressemble aux pages jaunes. Il sert à stocker et à classer les interfaces des services web.

**WSDL (Web Services Description Language)** : définit l'interface du service web, les données et les types des messages, les interactions et les protocoles.

**SOAP (Simple Object Access Protocol)** : est un protocole d'encodage des messages reposant sur les technologies XML. Il définit une enveloppe pour la communication des services web.

**Protocole de transport** : Les messages sont échangés à l'aide d'un protocole de transport. Bien que HTTP (Hypertext Transfer Protocol) soit le plus utilisé, d'autres comme SMTP ou JMS sont également possibles.

**XML (Extensible Markup Language)** : est la base sur laquelle sont construits et définis les services web (SOAP, WSDL et UDDI).

# Présentation des web services REST

- **REST (Representational State Transfer)** ou RESTful est un style d'architecture permettant de construire des applications (Web, Intranet, Web Service). Il s'agit d'un ensemble de conventions et de bonnes pratiques à respecter et non pas une technologie à part entière. L'architecture REST utilise les spécifications originelles du protocole HTTP, plutôt que de réinventer une surcouche (comme le font SOAP ou XML-RPC par exemple).
- **Principes définis dans la thèse de Roy FIELDING en 2000**
  - Parmis les principaux auteurs de la specification HTTP
  - Membre fondateur de la fondation Apache
  - Développeur du serveur Web Apache
- **REST est un style d'architecture inspiré de l'architecture du Web**
  - REST est un style d'architecture qui repose sur le protocole HTTP : On accède à une ressource (par son URI unique) pour procéder à diverses opérations (GET lecture / POST écriture / PUT modification / DELETE suppression), opérations supportées nativement par HTTP.
  - REST est une approche pour construire une application

## REST n'est pas

- un format
- un protocole
- un standard

# Eléments de base du REST

- **Ressources** : Une ressource est tout ce que peut designer ou manipuler un client, toute information pouvant être référencée dans un lien hypertexte
- **URI** : Une ressource web est identifiée par une URI, qui est un identifiant unique formé d'un nom et d'une adresse indiquant où trouver la ressource  
exemples d'URI :
  - <http://www.pearson.fr/catalogue/programmation/>
  - <http://www.pearson.fr/Resources/titles/27440100325790/Images>
- **Représentations** : On peut vouloir obtenir la représentation d'un objet sous forme de texte, de XML, de PDF ou sous un autre format. Un client traite toujours une ressource au travers de sa représentation
- **WADL** : Alors que les services web SOAP utilisent WSDL pour décrire le format des requêtes possibles, WADL (*Web Application Description Language*) sert à indiquer les interactions possibles avec un service web REST (**rarement utilisé**)
- **Protocole HTTP**

# Caractéristiques des web services REST

- Les services Web REST sont sans états (**Stateless**)
  - ✓ **Sans état** : chaque requête d'un client vers un serveur doit contenir toute l'information nécessaire pour permettre au serveur de comprendre la requête, sans avoir à dépendre d'un contexte conservé sur le serveur. Cela libère de nombreuses interactions entre le client et le serveur.
- Les services web REST fournissent une interface uniforme basée sur les méthodes HTTP ( GET, POST, PUT et DELETE ) et les URI, dès que l'on connaît l'emplacement d'une ressource (son URI), il suffit d'invoquer les verbes HTTP (GET, POST, etc.)
- Les architectures orientées REST sont construites à partir de ressources qui sont uniquement identifiées par des URIs

# Caractéristiques des web services REST

## ■ Simplicité de REST face à SOAP

Les Principes de REST encouragent la simplicité, la légèreté et l'efficacité des applications.



**SOAP**

**REST**

# Méthodes (Verbes) HTTP

Une ressource quelconque peut subir quatre opérations de base désignées par CRUD :

- ❖ Create (Créer)
- ❖ Retrieve (Lire)
- ❖ Update ( Mettre à jour)
- ❖ Delete (Supprimer)

REST s'appuie sur le protocole HTTP pour exprimer les opérations via les méthodes HTTP:

- ❖ Create      POST
- ❖ Retrieve    GET
- ❖ Update     PUT
- ❖ Delete    DELETE

# Web Services avec Java/JEE et Spring



# Rappel : JSON

- JSON = JavaScript Object Notation
- Un format léger en texte brut (*plain text*) pour échanger les données
- Format indépendant du langage: On peu l'utiliser avec Java, JS, Python, PHP,...

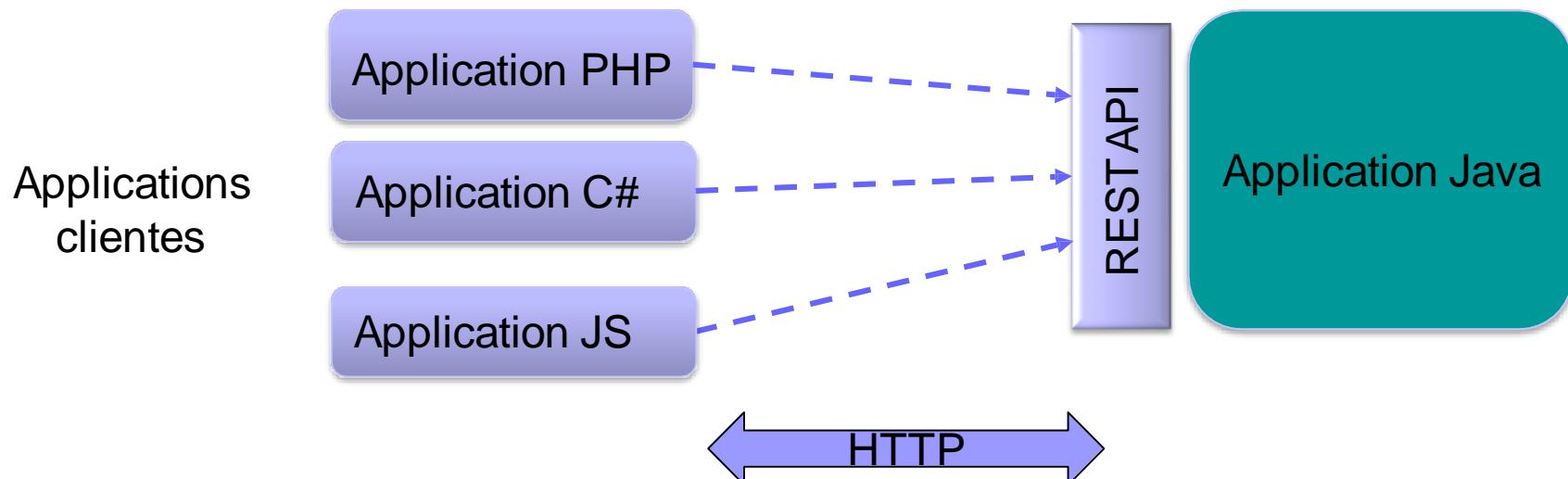
```
{ nom      valeur      Chaînes de caractères
      "id": 122,           en guillemets
      "firstName": "BOUDAA", Valeur
      "lastName" : "Tarik",  booléenne
      "hasResearchPublications": true,
      "researchField" : {
          "title": "Natural Language Processing",
          "subject" : "Textual Entailment "
      },
      "languages" : ["Tamazight", "English", "French", "Arabic"],
      "siteWeb" : null
}
```

Annotations:

- nom
- valeur
- Chaînes de caractères en guillemets
- Valeur booléenne
- Valeur composée
- Tableau

# REST API

- REST est une approche légère (*lightweight*) pour communiquer entre applications
- REST ne dépend pas d'un langage de programmation:
  - ✓ L'application cliente pourra utiliser n'importe quel langage de programmation
  - ✓ le serveur pourra utiliser n'importe quel langage de programmation
- Les applications REST peuvent utiliser n'importe quel format de données.
- Les formats utilisés généralement sont XML et JSON
- **JSON** est plus populaire et plus moderne



*Appels de l'API REST à travers HTTP*

# REST API

## ■ Bonnes pratiques

Il faudrait utiliser les verbes HTTP (GET, POST, PUT, DELETE) pour les actions à faire sur les données.

`/api/createStudent`

`/api/getStudentById/2`

`/api/deleteStudent/1`

...

**Anti-pattern**

`POST /api/students`

`GET /api/students/2`

`DELETE /api/students`

...

**Best practice**

# REST API

## ■ Tester l'API REST

- Pour les applications simples nous pouvons utiliser un navigateur Web
- Pour des applications avancées on pourra utiliser un logiciel client REST comme Postman
- Postman permet de construire des requêtes de n'importe quelle type et éditer ou analyser JSON.

# REST API

## ■ Tester l'API REST avec POSTMAN



The screenshot shows the Postman application interface. A red box highlights the URL field containing "http://localhost:8080/Exemple\_Cours\_rest\_01/api/persons/1". A red arrow labeled "1" points from this box to the "Send" button. Another red arrow labeled "2" points from the "Send" button to the status bar at the bottom, which displays "Status: 200 OK". A large red box highlights the JSON response body, which contains the following data:

```
1
2   "idPersonne": 1,
3   "nationalIdNumber": "AA12345678",
4   "firstName": "TARIK",
5   "lastName": "BOUDAA",
6   "age": 33,
7   "email": "tarikboudaa@yahoo.fr",
8   "password": "fsdfsdfsdfsdf",
9   "address": "Ecole Nationale des Sciences Appliquées d'Al Hoceima BP 03, Ajdir Al-Hoceima",
10  "state": "Morocco",
11  "community": [
12    "Spring",
13    "JAKARTA EE"
14  ],
15  "gender": "Male "
```

A red arrow labeled "3" points from the "Send" button to the JSON response body. A callout box with the text "Réponse du serveur" has a red arrow pointing to the JSON response body.

# REST API

## ■ Tester l'API REST avec POSTMAN

POST  Send

Params Authorization Headers (9) Body  Pre-request Script Tests Settings Cookies Beautify

none  form-data  x-www-form-urlencoded  raw  binary  GraphQL

1  
2  
3     "nationalIdNumber": "AA12345678",  
4     "firstName": "TARIK",  
5     "lastName": "BOUDAA",  
6     "age": 33,  
7     "email": "tarikboudaa@yahoo.fr",  
8     "password": "fsdfsdfsdgsdf",  
9     "address": "Ecole Nationale des Sciences Appliquées d'Al Hoceima BP 03, Ajdir Al-Hoceima",  
10    "state": "Morocco",  
11    "community": [

Body Cookies Headers (5) Test Results Status: 200 OK Time: 476 ms Size: 483 B Save Response

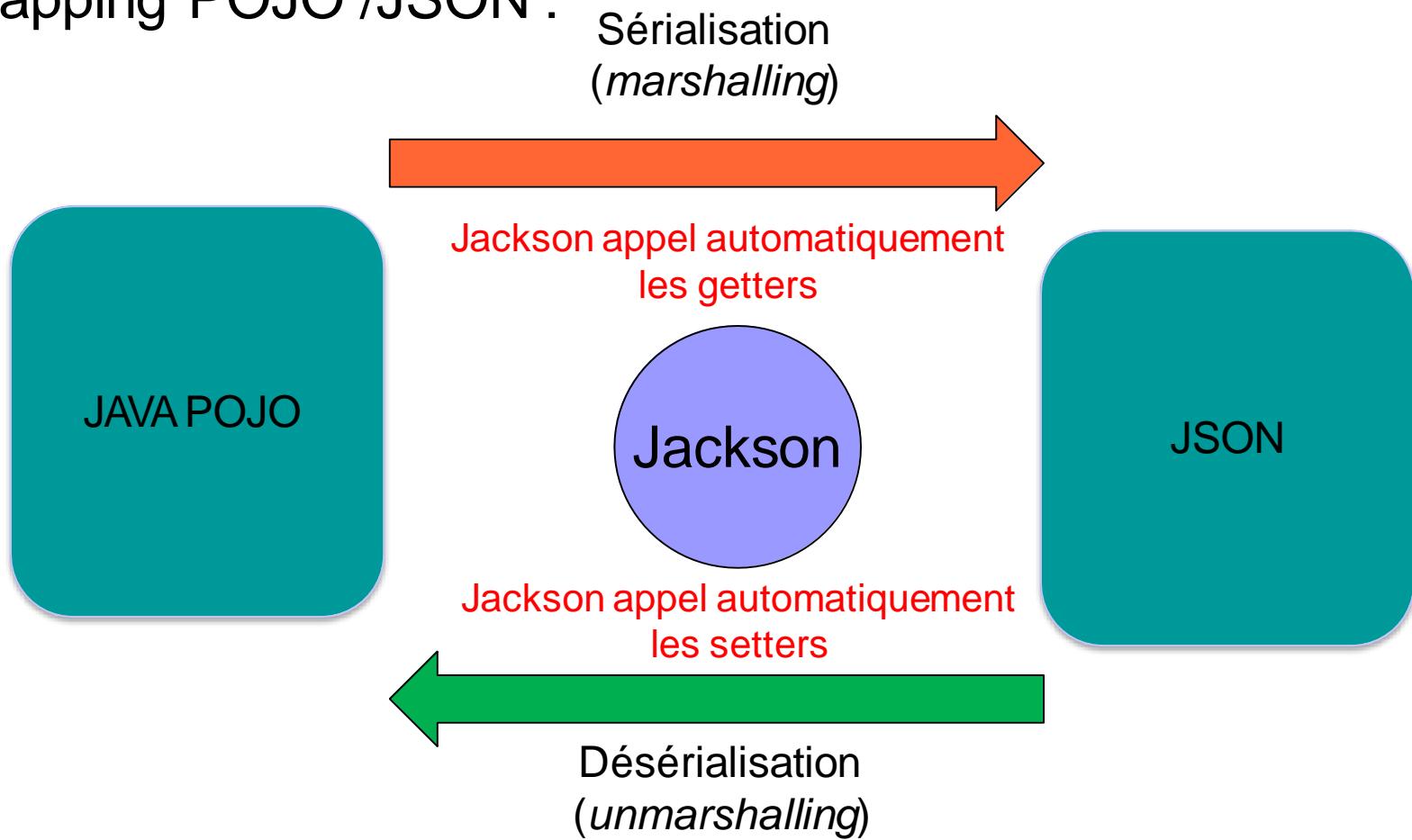
Pretty Raw Preview Visualize JSON

1  
2     "idPersonne": 4,  
3     "nationalIdNumber": "AA12345678",  
4     "firstName": "TARIK",  
5     "lastName": "BOUDAA",

Réponse du serveur

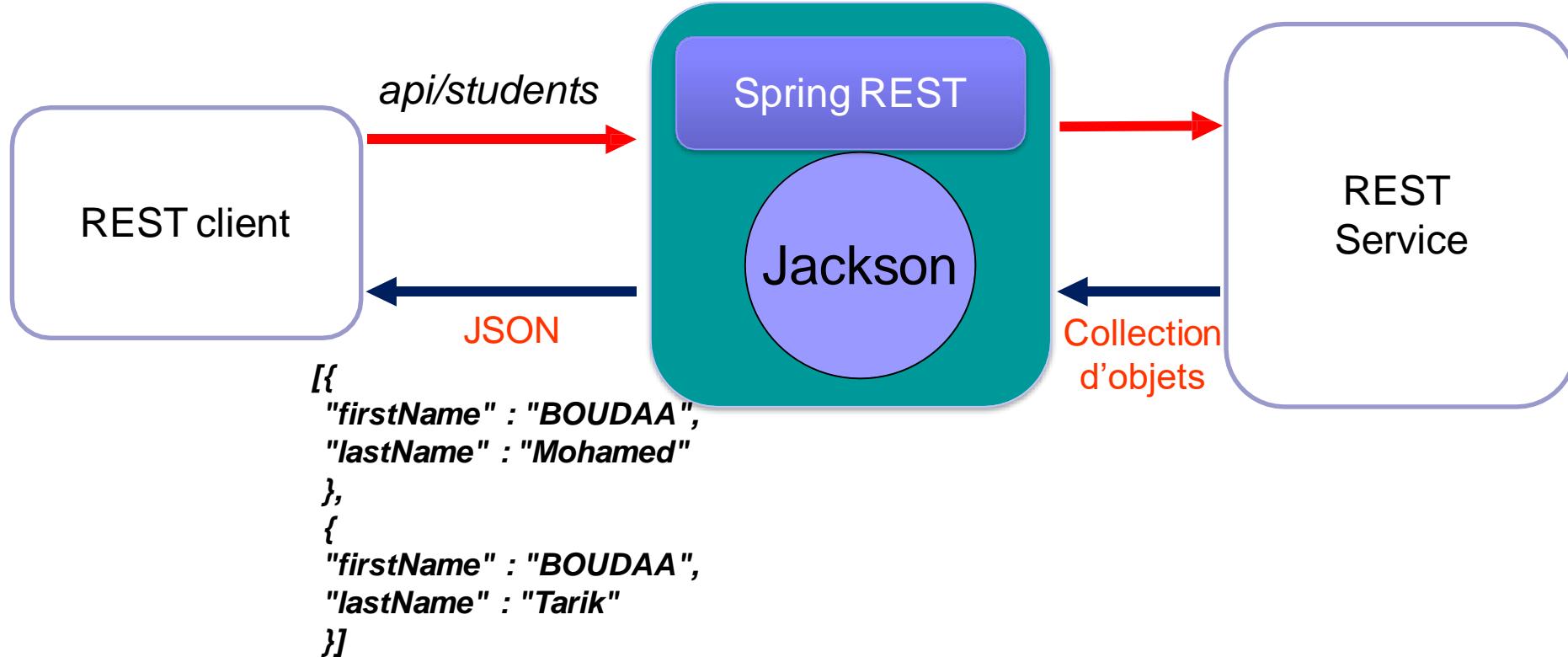
# Sérialisation et désérialisation JSON (Data binding)

## ■ Mapping POJO / JSON :



# REST avec Spring

- Spring REST utilise Jackson en background



- Les données JSON passées à un contrôleur Spring REST seront automatiquement converties en POJO
- Les données retournées sous forme de POJOs seront automatiquement converties en JSON

# REST avec Spring

## ■ Méthodes HTTP CRUD et les annotations Spring

HTTP Method	Opérations CRUD	Annotations Spring
GET	Lecture d'entités	@GetMapping
PUT	Mise à jour d'une entité	@PutMapping
POST	Création d'une nouvelle entité	@PostMapping
DELETE	Supprimer une entité	@DeleteMapping

**@methodXMapping** est un raccourci de **@RequestMapping(method= RequestMethod.methodX)**

Par exemple:

**@GetMapping** est un raccourci de **@RequestMapping(method= RequestMethod.GET)**

# REST avec Spring

## ■ Codes des statuts HTTP

Intervalles des codes	Signification
100-199	Information
200-299	Succès
300-399	Redirection
400-499	Erreur du client <b>(Exemple:</b> <i>404: File Not Found, 401 : Authentification Required</i> )
500-599	Erreur du serveur <b>(Exemple :</b> <i>500 : internal server error</i> )

Les codes des statuts sont définies dans Spring dans l'énumération :  
[org.springframework.http.HttpStatus](#)

**Exemple** Le statut 400 :

`org.springframework.http.HttpStatus.BAD_REQUEST.value()`

# Contrôleur REST de Spring MVC

- Définir un contrôleur REST avec **@RestController**
- Spring MVC fournit un support pour REST
- Il suffit d'utiliser l'annotation **@RestController** qu'est une extension de **@Controller**.
- Un contrôleur REST gère la communication HTTP
- Le databinding est assuré en background par Jackson

# Contrôleur REST de Spring MVC

## ■ Définir un contrôleur REST avec `@RestController`

```
@RestController
@RequestMapping("/test")
public class TestRestController{
    @GetMapping("/helloRest")
    public String sayHelloEnsaah(){
        return "Hello ENSAH";
    }
}
```

On ajoute le support de REST

Le service « REST endpoint » est accessible par `/test/helloRest`

Les données à envoyer au client (texte brute)

The diagram illustrates the annotations in the Java code and their meanings:

- `@RestController`: A red annotation pointing to a callout box stating "On ajoute le support de REST".
- `@RequestMapping("/test")`: A blue annotation pointing to a callout box stating "Le service « REST endpoint » est accessible par /test/helloRest".
- `@GetMapping("/helloRest")`: A green annotation pointing to a callout box stating "Les données à envoyer au client (texte brute)".

# Contrôleur REST de Spring MVC

## ■ Cas des services REST recevant des données

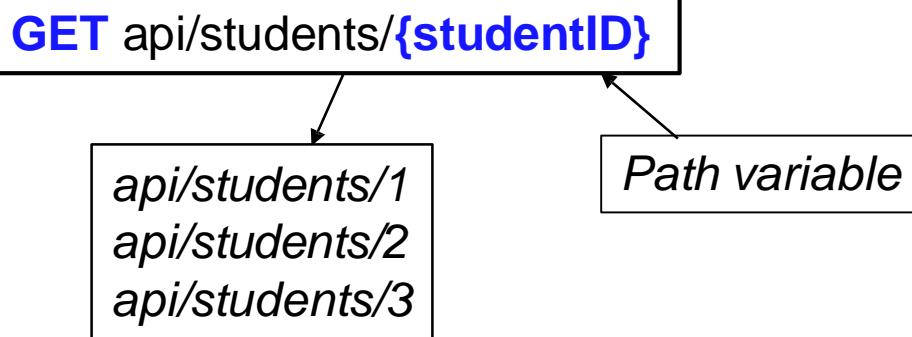
```
@PostMapping("/persons")
public Person addPerson(@RequestBody Person person) {
    ...
    personService.addPerson(person);
    ...
    return person;
}
```

```
@PutMapping("/persons")
public Person updatePerson(@RequestBody Person person)
{
    ...
    personService.updatePerson(person);
    ...
    return person;
}
```

# Contrôleur REST de Spring MVC

## ■ Paramètres de la requête (*path variables*)

Retrouver un étudiant par ID:



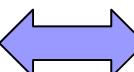
```
@RestController  
@RequestMapping("/api")  
public class StudentRestController{  
    @GetMapping("/students/{studentID}")  
    public Student getStudent(@PathVariable studentID){  
        ....  
        return student;  
    }  
}
```

# Contrôleur REST de Spring MVC

## ■ Gestion des erreurs

- Pour informer le consommateur de l'API REST des erreurs, il faut lui envoyer les informations de celle-ci dans la réponse sous le format d'échange utilisé (JSON).
- Supposons que nous voulons gérer les exceptions **StudentNotFoundException** dans la méthode *getStudent* du contrôleur REST.
- Pour ce faire on définit l'erreur sous forme d'un POJO avec les attributs nécessaires et Spring va convertir ce POJO en JSON lors de l'envoi de la réponse au client.

```
public class StudentError {  
    int codeStatut;  
    String description;  
    //Getters/setters  
}
```



```
{  
    "codeStatut": 404,  
    "description": "étudiant introuvable"  
}
```

# Contrôleur REST de Spring MVC

## ■ Gestion des erreurs

Supposons que nous voulons gérer les exceptions de type

**StudentNotFoundException** dans la méthode *getStudent* du contrôleur REST

```
@RestController  
@RequestMapping("/api")  
public class StudentRestController{  
  
    @GetMapping("/students/{studentID}")  
    public Student getStudent(@PathVariable studentID){  
  
        ....  
        throws new StudentNotFoundException();  
        return student;  
    }  
}
```

### **@ExceptionHandler**

```
public ResponseEntity<StudentError> handleException(StudentNotFoundException ex){  
    //Créer et initialiser StudentError
```

```
    StudentError err =new StudentError();  
    err.setCodeStatut(HttpStatus.NOT_FOUND.value());  
    err.setDescription("Etudiant introuvable");  
    return new ResponseEntity<StudentError>(err, HttpStatus.NOT_FOUND);
```

Gestionnaire d'exception  
pour le service REST

```
}
```

# Contrôleur REST de Spring MVC

## ■ Gestion de plusieurs erreurs

- ✓ On peut gérer plusieurs exceptions en écrivant plusieurs gestionnaires.
- ✓ On peut également gérer les exceptions d'une façon générique en interceptant les exceptions mères comme Exception

```
@RestController  
@RequestMapping("/api")  
public class StudentRestController{  
    ....  
    @ExceptionHandler  
    public ResponseEntity<StudentError> handleException(StudentNotFoundException ex){  
        ....  
    }  
    @ExceptionHandler  
    public ResponseEntity<StudentError> handleException(StudentBadDataException ex){  
        ....  
    }  
    ....  
    @ExceptionHandler  
    public ResponseEntity<StudentError> handleException(Exception ex){  
        ....  
    }  
}
```

The code snippet illustrates the use of multiple `@ExceptionHandler` annotations within a single controller class. It defines three nested exception handlers:

- Gestionnaire 1** (green box): Handles `StudentNotFoundException`.
- Gestionnaire 2** (blue box): Handles `StudentBadDataException`.
- Gestionnaire générique** (red box): Handles all other exceptions.

# Contrôleur REST: Gestion des erreurs

## ■ Gestion des erreurs d'une manière globale

La gestion des exceptions au niveau des contrôleurs pourra rapidement devenir une tâche fastidieuse surtout dans le cas d'une grande application. Dans la pratique on pourra externaliser la gestion des exceptions en adoptant une gestion globale des exceptions. Ceci pourra se faire avec `@ControllerAdvice`.

Il peut être considéré comme un intercepteur d'exceptions levées par des méthodes annotées avec `@RequestMapping` (ou ses raccourcis)



# Contrôleur REST: Gestion des erreurs

## ■ Gestion des erreurs d'une manière globale

### **@ControllerAdvice**

```
public class StudentRestExceptionHandler{
```

```
//Même code qu'avant
```

### **@ExceptionHandler**

```
public ResponseEntity<StudentError>
```

```
handleException(StudentNotFoundException ex){
```

```
....
```

```
}
```

```
....
```

### **@ExceptionHandler**

```
public ResponseEntity<StudentError> handleException(Exception ex){
```

```
....
```

```
}
```

```
}
```