

Mastère Professionnel

Développement des Systèmes
Informatiques et Réseaux (DSIR)



Micro services

01- Outil Maven

Mohamed ZAYANI

2023/2024

C'est quoi Maven?

- Maven est un outil de gestion de projet logiciel pour Java maintenu par « l'Apache Software Foundation ».
- Maven est un outil qui permet de gérer les dépendances d'un projet JAVA et automatiser sa construction :
 - ❖ compilation, test,
 - ❖ packaging,
 - ❖ déploiement,
 - ❖ production de livrable
 - ❖ gestion de sites web
- Maven permet aussi de générer des documentations (sous forme de rapports) concernant le projet.



Caractéristiques Maven

Principe de Convention plutôt que configuration

- Maven établit un certain nombre de conventions afin d'**automatiser** certaines tâches et rendre la procédure de configuration **plus facile**.
- Une de ces conventions est de **fixer l'arborescence** d'un projet. Ainsi, Maven permet de générer une squelette du code du projet en utilisant la notion des « **archetypes** ».

Approche déclarative

- Maven utilise une **approche déclarative** où la structure du projet et son contenu sont décrits dans un document XML nommé **POM.xml**
(**P**roject **O**bject **M**odel)

Aspect extensible

- Maven est **extensible** grâce à un mécanisme de plugins qui permettent d'ajouter des fonctionnalités.

Notion d'artéfact

- Un **artéfact** est un composant packagé possédant un identifiant unique composé de trois éléments : **un groupId**, **un artifactId** et **un numéro de version**.
 1. **groupId** : définit l'organisation ou groupe qui est à l'origine du projet. Il est formulé sous la forme d'un package Java
(Exemple : **org.jee.maven**)
 2. **artifactId** : définit le nom du projet (nom unique dans le groupe)
(Exemple: **premierProjet**)
 3. **version** : définit la version du projet. Les numéros de version sont souvent utilisés pour des comparaisons et des mises à jour.

NB: La gestion des versions est importante pour identifier quel artefact doit être utilisé: la version est utilisée comme une partie de l'identifiant d'un artéfact.

Exemple

- La déclaration d'une dépendance est spécifiée dans le fichier « **pom.xml** » (cœur de Maven)
- Exemple:

```
<dependency>  
  <groupId>org.hibernate</groupId>  
  <artifactId>hibernate</artifactId>  
  <version>3.3.2</version>  
  <scope>compile</scope>  
</dependency>
```

- La notion de «**scope**» définit la portée d'une dépendance: permet de préciser dans quel contexte une dépendance est utilisée
- La portée « **compile** » indique que la dépendance est utilisable par toutes les phases et à l'exécution. **C'est le scope par défaut**

Fichier « pom.xml »

- Le fichier **POM** (**P**roject **O**bject **M**odel) contient la description du projet Maven. Il contient les informations nécessaires à la génération du projet : (identification de l'artéfact, déclaration des dépendances, définition d'informations relatives au projet..). Voici un exemple:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jee.test</groupId>
  <artifactId>MaWebApp</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Mon application web</name>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.7</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Balises du fichier « pom.xml »

dossier	Description
<modelVersion>	Préciser la version du modèle de POM utilisée
<groupId>	Préciser le groupe ou l'organisation qui développe le projet. C'est une des clés utilisée pour identifier de manière unique le projet et ainsi éviter les conflits de noms
<artifactId>	Préciser la base du nom de l'artéfact du projet
<packaging>	Préciser le type d'artéfact généré par le projet (jar , war , ear , pom , ...). La valeur par défaut est jar
<version>	Préciser la version de l'artéfact généré par le projet. Le suffixe - SNAPSHOT indique une version en cours de développement
<name>	Préciser le nom du projet utilisé pour l'affichage

Balises du fichier « pom.xml »

dossier	Description
<description>	Préciser une description du projet
<url>	Préciser une url qui permet d'obtenir des informations sur le projet
<dependencies>	Définir l'ensemble des dépendances du projet
<dependency>	Déclarer une dépendance en utilisant plusieurs tags fils : <groupId> , <artifactId> , <version> et <scope>

- Le fichier POM doit être à la racine du répertoire du projet.
- La balise racine du fichier « pom.xml » est la balise **<project>**.

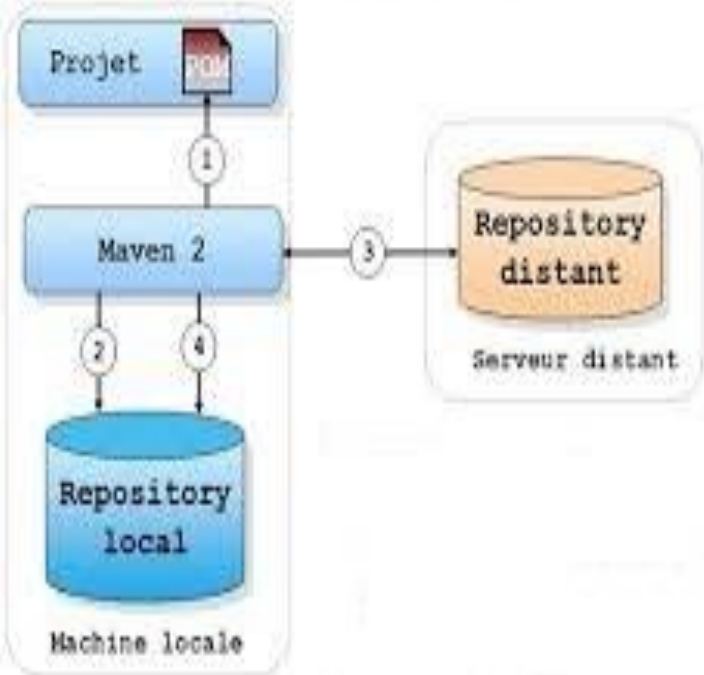
Gestion de dépendances

- Maven utilise la notion de référentiel ou dépôt (**repository**) pour stocker les **dépendances** et les **plugins** requis pour générer les projets.
- Un dépôt contient un ensemble d'artéfacts qui peuvent être des livrables, des dépendances, des plugins, ...
- Ceci permet de **centraliser** ces éléments qui sont généralement utilisés dans plusieurs projets : c'est notamment le cas pour les plugins et les dépendances.
- Maven distingue deux types de dépôts : **local** et **distant** (remote):
 1. **dépôt central (repository central)**: il stocke des dépendances et les plugins utilisables par tout le monde car disponible sur le web; ce sont généralement des artéfacts open source
 2. **dépôt local (repository local)** : il stocke une copie des dépendances et plugins requis par les projets à générer en local. Ces artéfacts sont téléchargés des dépôts centraux

Gestion de dépendances

- Maven utilise un ou plusieurs dépôts (**repository**) qui peuvent être locaux ou distants
- Si un élément n'est pas trouvé dans le répertoire local, il sera téléchargé dans ce dernier à partir d'un dépôt distant

Gestion des dépendances par Maven 2



1. Lecture du fichier « **pom.xml** » et la liste des dépendances
2. Vérification de l'existence des dépendances dans le **repository local** (dépôt local)
3. Téléchargement des dépendances non trouvées en accédant au **repository central** (via Internet)
4. Copies de dépendances dans le repository local

Repository Maven

- La première exécution d'une commande « Maven », un dossier nommé « **.m2** » est créé dans le répertoire « HOME » de l'ordinateur

(Exemple: **c:\Utilisateurs\Ma_Machine**)

- Le dossier « **.m2** », ainsi créé, comporte un sous-dossier « **repository** » constituant le dépôt local de Maven
- Il est possible de personnaliser l'emplacement du **repository local** en spécifiant le chemin dans un fichier « **settings.xml** » à placer dans le dossier « **.m2** ».



Archetypes Maven

- Afin de générer la squelette d'un projet, Maven s'appuie sur des archétypes (ou modèles).
- Un **Archetype** est un outil pour faire **des templates** de projet.
- Un projet généré via un Archetype est dit **projet Maven**.
- Un **Archetype** est un **modèle** de projet à partir duquel d'autres projets sont créés.
- L'utilisation d'archétypes a pour principal avantage de **normaliser le développement de projets** et de permettre aux développeurs de suivre facilement **les meilleures pratiques** tout en débutant leurs projets plus rapidement.
- **Maven** fournit, aux utilisateurs, une très grande liste de différents types de modèles de projet (**ENVIRON 614**) en utilisant le concept d'**Archetype**.

Exemples d'Archetypes Maven

archetype	Description
quickstart	Contient un exemple projet Maven standard
simple	Contient un simple projet Maven
webapp	Contient un exemple projet Maven d'une application web
j2ee-simple	Contient un exemple projet Maven d'une application JEE

- **Maven** permet de créer la structure d'un projet selon un modèle donné (archetype) en utilisant la commande suivante:

mvn archetype:generate

Structure d'un projet Maven

dossier	Description
/src	les sources du projet
/src/main	les fichiers sources principaux
/src/main/java	le code source (sera compilé dans /target/classes)
/src/main/resources	les fichiers de ressources (fichiers de configuration , images , ...). Le contenu de ce répertoire est copié dans target/classes pour être inclus dans l'artéfact généré
/src/main/webapp	les fichiers de l'application web
/src/test	les fichiers pour les tests
/src/test/java	le code source des tests (sera compilé dans /target/test-classes)

Structure d'un projet Maven

dossier	Description
/src/test/resources	les fichiers de ressources pour les tests
/target	les fichiers générés pour les artéfacts et les tests
/target/classes	les classes compilées
/target/test-classes	les classes compilées des tests unitaires
/target/site	site web contenant les rapports générés et des informations sur le projet
/pom.xml	le fichier POM de description du projet

NB: l'arborescence d'un projet Maven est par défaut imposée par l'outil Maven
(**par convention**)

Principe de fonctionnement du Maven 3

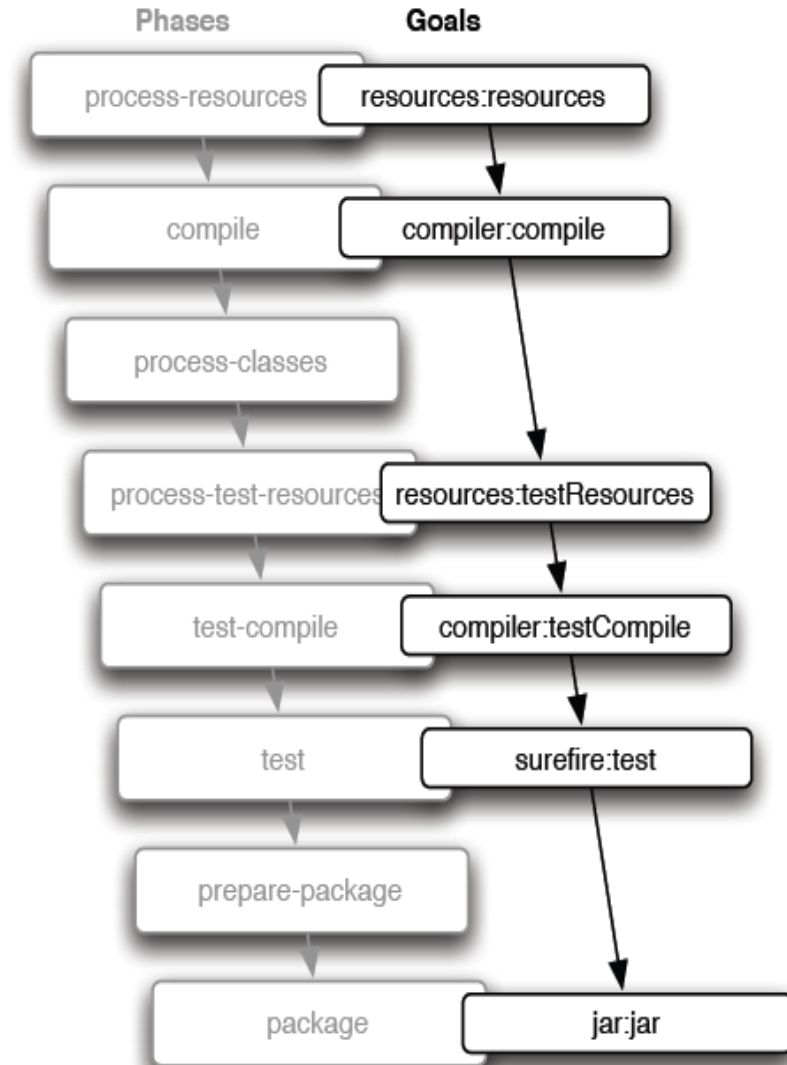
- Toutes les fonctionnalités décrites ici font partie de la version **3** de Maven.
- Une connexion à internet est nécessaire pour permettre le téléchargement des plugins requis et des dépendances.
- Pour installer Maven:
 - ❖ Télécharger l'archive sur le site:
<http://maven.apache.org/download.html>
 - ❖ Décompresser l'archive dans un répertoire du système
 - ❖ Créer la variable d'environnement **M2_HOME** qui pointe sur le répertoire contenant Maven
 - ❖ Ajouter le chemin **M2_HOME/bin** à la variable **PATH** du système
 - ❖ Pour vérifier l'installation, il suffit de lancer la commande:

mvn -version

Cycle de vie d'un projet Maven

▪ Dans le cycle de vie 'par défaut' d'un projet Maven, les phases les plus utilisées sont:

- ❖ **validate** : vérifie les prérequis d'un projet maven
 - ❖ **compile** : compilation du code source
 - ❖ **test** : lancement des tests unitaires
 - ❖ **package** : assemble le code compilé en un livrable
 - ❖ **install** : partage le livrable pour d'autres projets sur le même ordinateur
 - ❖ **deploy** : publie le livrable pour d'autres projets dans le « repository » distant
- Les phases s'exécutent de façon **séquentielle** de façon à ce qu'une phase dépende de la phase précédente.
- Par exemple, le lancement par l'utilisateur de la phase test (**mvn test**) impliquera le lancement préalable par maven des phases « **validate** » et « **compile** ».



Commandes Maven 3

- Toutes les fonctionnalités décrites font partie de la version **3** de Maven.
- Une commande Maven3 s'utilise en ligne de commande sous la forme suivante :

mvn **plugin:goal** ou **mvn** **plugin**

- Exemple:

mvn **archetype:generate**

- Il est possible d'utiliser des options précédées par « - »

- Exemple:

mvn **-version**

Exemples de commandes Maven 3

commande	Description
mvn package	Construire le projet pour générer l'artéfact
mvn site	Générer le site de documentation dans le répertoire target/site
mvn clean	Supprimer les fichiers générés par les précédentes générations
mvn install	Générer l'artéfact et le déployer dans le dépôt local
mvn eclipse:eclipse	Générer des fichiers de configuration Eclipse à partir du POM (notamment les dépendances)
mvn javadoc:javadoc	Générer la Javadoc
mvn test	Exécuter les tests unitaires



Mastère Professionnel

Développement des Systèmes
Informatiques et Réseaux (DSIR)

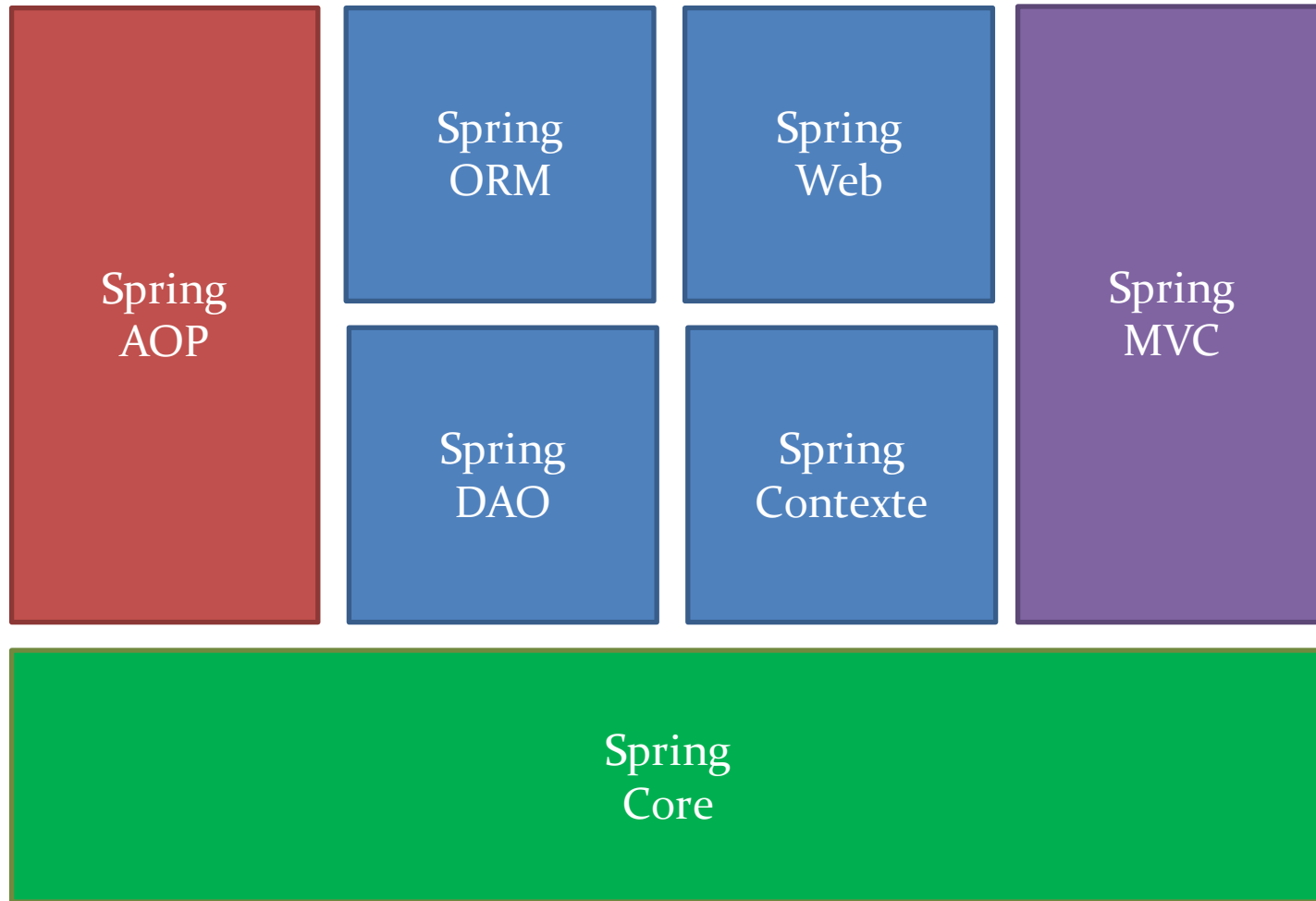
Micro services

02- JPA

Mohamed ZAYANI

2023/2024

Structure de Spring

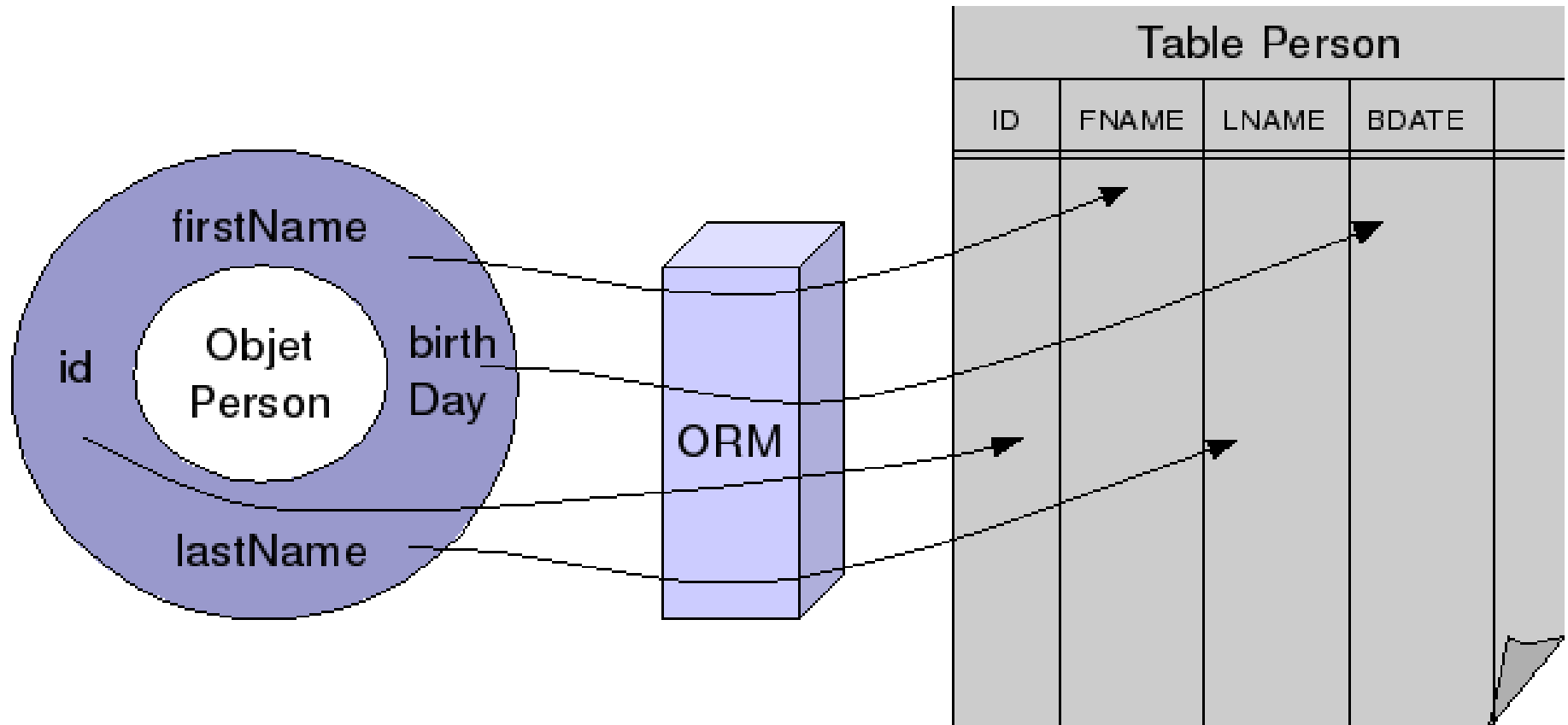


ORM

- **ORM** est l'acronyme de **O**bject/**R**elational **M**apping
- **ORM** est un mapping objet-relationnel: type de programme informatique qui se place en interface entre un programme applicatif et une base de données relationnelle pour simuler une base de données orientée objet.
- Le but de **l'ORM** est de faciliter la manipulation de données stockées dans un Système de Gestion de Base de Données Relationnelles (SGBDR) au sein des langages de programmation objet.
- **ORM** offre **une couche d'abstraction** pour traduire des données extraites de la base de données vers un objet propre au langage de programmation.

Le développeur travaille ainsi uniquement avec des objets sans se soucier du stockage sous-jacent des données.

schéma ORM



Avanatges ORM

- **Simplifier le code**

on ne fait que "parler objet" de la couche la plus haute à celle la plus basse

- **Augmenter la maintenabilité**

il n'y a plus de code SQL à maintenir au sein du code objet

- **Augmenter la portabilité**

la base de données étant masquée il est possible de passer d'un SGBD à un autre

JPA ?

- **JPA** est l'acronyme de **J**ava **P**ersistence **A**PI
- **JPA** est une solution JEE pour adopter le concept ORM
- **JPA** une **spécification JEE** qui définit un ensemble de règles permettant la gestion de la correspondance entre des objets Java et une base de données (**gestion de la persistance**).
- **JPA** décrit comment respecter le standard, mais c'est au développeur de choisir l'implémentation:
 - ❖ **Hibernate**
 - ❖ **EclipseLink**
 - ❖ **Spring DATA**

Entité JPA

- Plain Old Java Object (POJO)

Une simple classe JAVA

- Peut contenir des attributs persistants ou non

l'état **non persistant** est spécifié grâce à l'annotation:

@Transient

- Peut étendre d'autres entités ou des classes qui ne sont pas des entités

- Sérializable

pas besoin de s'occuper des transferts d'objets

- Déclarée avec le mot clé :

@Entity

Classe persistance

- Pour qu'une classe puisse être persistante, il est nécessaire:
 - ❖ qu'elle soit identifiée comme une entité (**entity**) en utilisant l'annotation **@java.persistence.Entity**
 - ❖ qu'elle possède un attribut identifiant en utilisant l'annotation **@javax.persistence.Id**
 - ❖ qu'elle ait un constructeur sans argument

```
@Entity
public class Client {
    @Id
    private Long id;
    private String prénom;
    private String nom;
    private String téléphone;
    private String email;
    private Integer age;
    private Date dateNaissance;
    // constructeurs/setter/getter
}
```

Entité JPA

- Une entité **JPA**, déclarée par l'annotation **@Entity** définit une classe Java comme étant persistante (associée à une table dans la base de données).
- Cette classe doit être implantée selon les normes **des beans**:
 - ❖ Les propriétés sont déclarées non publiques (de préférence de type « **private** »)
 - ❖ Chaque propriété possède deux accesseurs selon les conventions habituelles:
 - Un accesseur en lecture (**getter**) permet de lire la valeur d'un attribut.
 - Un accesseur en écriture (mutateur ou **setter**) permet de modifier la valeur d'un attribut.
 - ❖ Une entité doit comporter un constructeur sans argument

Les annotations

- Toutes les annotations sont définies dans le package «**javax.persistence**»
- Une annotation peut marquer:
 - soit le champ de la classe concernée
 - soit le getter de la propriété.

- **Exemple:**

```
@Column (name = "name")  
private String nom;
```

Ou bien :

```
@Column (name = "name")  
public String getNom()  
{  
    return nom;  
}
```

Définition d'une entité

```
import javax.persistence.*;

@Entity
public class Personne
{
    private String nom;
    private String prenom;
    public String getNom()
    {
        return nom;
    }
    public void setNom(String nom)
    {
        this.nom = nom;
    }
    public String getPrenom()
    {
        return prenom;
    }
    public void setPrenom(String prenom)
    {
        this.prenom = prenom;
    }
    public Personne () {}
}
```

- **Remrques:**
- Toute entité doit avoir une propriété déclarée comme étant l'identifiant de la ligne dans la table correspondante.
- L'identifiant est indiqué avec l'annotation **@Id**

```
@Id
private int matricule;
```

- La propriété annotée par « **@Id** » est traduite en une colonne constituant **la clé primaire** de la table correspondante dans la BD

Identifiant d'une entité

- L'identifiant d'une entité JPA est indiqué avec l'annotation **@Id**.
- Pour produire **automatiquement** les valeurs d'identifiant (en insérant dans le BD), on ajoute une annotation **@GeneratedValue** avec un paramètre **strategy**.
- Le paramètre **strategy** peut avoir plusieurs valeurs :
 - **strategy = GenerationType.AUTO**: La génération de la clé primaire est laissée à l'implémentation. C'est **hibernate** qui s'en charge et qui crée une séquence unique sur tout le schéma via la table **hibernate_sequence**
 - **strategy = GenerationType.IDENTITY** : La génération de la clé primaire se fera à partir d'une Identité propre au **SGBD**. Il utilise un type de colonne spéciale à la base de données. (Exemple pour **MySQL**, il s'agit d'un **AUTO_INCREMENT**)
 - **strategy = GenerationType.TABLE** : La génération de la clé primaire se fera en utilisant une table dédiée **hibernate_sequence** qui stocke les noms et les valeurs des séquences. Cette stratégie doit être utilisée avec une autre annotation qui est **@TableGenerator**.
 - **strategy = GenerationType.SEQUENCE** : La génération de la clé primaire se fera par une séquence définie dans le **SGBD**, auquel on ajoutera l'attribut **generator**. Cette stratégie doit être utilisée avec une autre annotation qui est **@SequenceGenerator**.
- Généralement, on utilise la première ou bien la deuxième stratégie. Exemple:

```
@Id
@GeneratedValue (strategy = GenerationType.AUTO)
private int matricule;
```

Annotation @Table

- Par défaut, une entité est associée à la table portant le même nom que la classe.
- Il est possible d'indiquer le nom de la table par une annotation **@Table**. (annotation optionnelle)
- **Exemple:**

```
@Entity
@Table(name="Person")
public class Personne
{
    . . . . .
}
```


Annotation @Column

- Par défaut, toutes les propriétés **non-statiques** et **non-finales** d'une classe-entité sont **persistantes** (à être stockées dans la BD)
- Pour indiquer des options à une colonne dans la BD, on utilise le plus souvent l'annotation **@Column**.
- L'annotation **@Column** présente les principaux attributs suivants
 - ❖ **name**: indique le nom de la colonne dans la table
 - ❖ **length**: indique la taille maximale de la valeur de la propriété
 - ❖ **nullable**: (avec les valeurs false ou true) indique si la colonne accepte ou non des valeurs à NULL
 - ❖ **unique**: indique que la valeur de la colonne est unique.
- **Exemple:**

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

Annotation @Transient

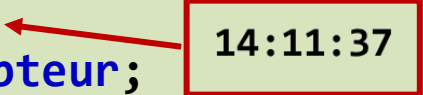
- Un objet métier peut avoir des propriétés que l'on ne souhaite pas rendre persistantes dans la BD. Il faut alors impérativement les marquer avec l'annotation **@Transient**.
- L'annotation **@Transient** permet d'indiquer au gestionnaire de persistance d'ignorer cette propriété.
- **Exemple:**

```
@Transient  
private String nom_prenom;
```

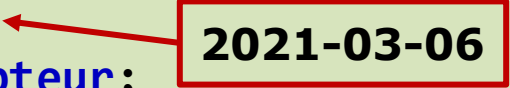
Annotation @Temporal

- L'annotation **@Temporal** permet de fournir des informations complémentaires sur la façon dont les propriétés encapsulant des données temporelles (**Date** et **Calendar**) sont associées aux colonnes dans la table (date, time ou timestamp).
- La valeur par défaut est timestamp.
- **Exemple:**


```
//prendre uniquement l'information du temps (heure:minute:seconde)  
@Temporal(TemporalType.TIME)  
private java.util.Date heureCapteur;
```



```
//prendre uniquement l'information du jour (année-mois-jour)  
@Temporal(TemporalType.DATE)  
private java.util.Date jourCapteur;
```



```
//prendre l'information totale de la date (année-mois-jour heure:minute:seconde)  
@Temporal(TemporalType.TIMESTAMP)  
private java.util.Date dateCapteur;
```



EntityManager

- Toutes les actions de persistance sur les entités JPA sont réalisées grâce à un objet dédié de l'API : Il s'agit de **EntityManager**.
- Une instance de « EntityManager » est réalisée par injection de dépendance en spécifiant l'annotation **@PersistenceContext**

- **Exemple:**

```
@Repository
@Transactional
public class PersonneDaoImpl implements IPersonneDao{
    //déclarer un objet « EntityManager »
    @PersistenceContext
    private EntityManager em;
    .....
}
```

- Un contexte de persistance (persistence context) est un ensemble d'entités géré par un EntityManager.
- **Exemple:**

```
// référencer le contexte
ApplicationContext contexte=SpringApplication.run(JpaSpringBootApplication.class, args);
// Récupérer une implémentation de l'interface "IProduitDao" par injection de dépendance
IProduitDao daoProduit = contexte.getBean(IProduitDao.class);
```

Fonctionnalités de « EntityManager »

- **EntityManager** est donc au cœur de toutes les actions de persistance.
- EntityManager permet de réaliser des opérations CRUD (**create**, **read**, **update**, **delete**) sur les données.
- Elle permet aussi de rechercher des données (**find**).
- La méthode **contains()** de l'EntityManager permet de savoir si une instance fournie en paramètre est gérée par le contexte. Dans ce cas, elle renvoie true, sinon elle renvoie false.
- La méthode **clear()** de l'EntityManager permet de détacher toutes les entités gérées par le contexte.
- L'appel des méthodes de mise à jour **persist()**, **merge()** et **remove()** ne réalise pas d'actions immédiates dans la base de données sous-jacente,
- Il est possible de forcer l'enregistrement des mises à jour dans la base de données en utilisant la méthode **flush()** de l'EntityManager

Utilisation de « EntityManager »

```
@PersistenceContext  
private EntityManager em;
```

```
public Personne save(Personne p)  
{  
    em.persist(p);  
    return p;  
}
```

insertion

```
public Personne findOne(Long id)  
{  
    Personne p = em.find(Personne.class, id);  
    return p;  
}
```

Recherche par clé primaire

```
public Personne update(Personne p)  
{  
    em.merge(p);  
    return p;  
}
```

Mise à jour

```
public void delete(Long id)  
{  
    Personne p = em.find(Personne.class, id);  
    em.remove(p);  
}
```

suppression

Recherche par requête

- La recherche par requête repose sur des méthodes dédiées de la classe EntityManager (Exemple : `createQuery`) et sur un langage de requête spécifique nommé **HQL** (implémenté par Hibernate)
- HQL est un langage d'interrogation extrêmement puissant qui ressemble au SQL.
- HQL est totalement orienté objet, cernant des notions comme l'héritage, le polymorphisme et les associations.
- Les requêtes sont insensibles à la casse, à l'exception des **noms de classes** Java et des **propriétés**.

- Exemple:**

Nom de l'attribut de la classe Java et non de la colonne de la table

```
public List<Personne> findAll()
{
    Query query= em.createQuery("select p from Personne p order by p.nom");
    return query.getResultList();
}
```

P est un alias pour référer à Personne dans la requête

Nom de la classe Java et non de la table

Recherche par requête paramétrée

- L'objet **Query** gère aussi des paramètres nommés dans la requête.
- Le nom de chaque paramètre est préfixé par « : » dans la requête.
- La méthode **setParameter()** permet de fournir une valeur à chaque paramètre.
- **Query** fournit une méthode **getResultList()** qui renvoie une collection contenant les éventuelles occurrences retournées par la requête.
- Il est possible d'utiliser la méthode **getSingleResult()** pour obtenir un objet unique retourné par la requête.

- **Exemple :**

```
public List<Produit> findByDesignation(String mc)
{
    Query query=
        em.createQuery("select p from Produit p where p.designation like :X");
        query.setParameter("X", "%" + mc + "%");

    return query.getResultList();
}
```

Paramètre nommé « x » préfixé par « : »

Affecter la valeur du paramètre « x »

Mettre à jour une entité JPA

- Pour modifier une entité existante dans la base de données, il faut :
 - ❖ Obtenir une instance de l'entité à modifier (ou bien à travers une recherche sur la clé primaire ou l'exécution d'une requête) (**find**)
 - ❖ Modifier les propriétés de l'entité
 - ❖ Utiliser la méthode **merge**

```
public Produit update(Produit p, String nom)
{
    p.setNom(nom);
    em.merge(p);
    return p;
}
```

Mettre à jour l'entité

Appeler la méthode « merge »

```
public Personne update(int id , String nom)
{
    Personne p =(Personne) em.find(Personne.class, id);
    p.setNom(nom);
    em.merge(p);
    return p;
}
```

Recherche par clé primaire

Supprimer une entité JPA

- Pour supprimer une entité existante dans la base de données, il faut :
 - ❖ Obtenir une instance de l'entité à modifier (ou bien à travers une recherche sur la clé primaire ou l'exécution d'une requête) (**find**)
 - ❖ Utiliser la méthode **remove**

```
public void delete(int id)
{
    Personne p = em.find(Personne.class, id);
    em.remove(p);
}
```

Recherche par clé primaire

Appeler la méthode « remove »

Rafraîchir une entité JPA

- Pour rafraîchir une entité existante dans la base de données, il faut :
 - ❖ Obtenir une instance de l'entité à modifier (ou bien à travers une recherche sur la clé primaire ou l'exécution d'une requête) (**find**)
 - ❖ Utiliser la méthode **refresh()**

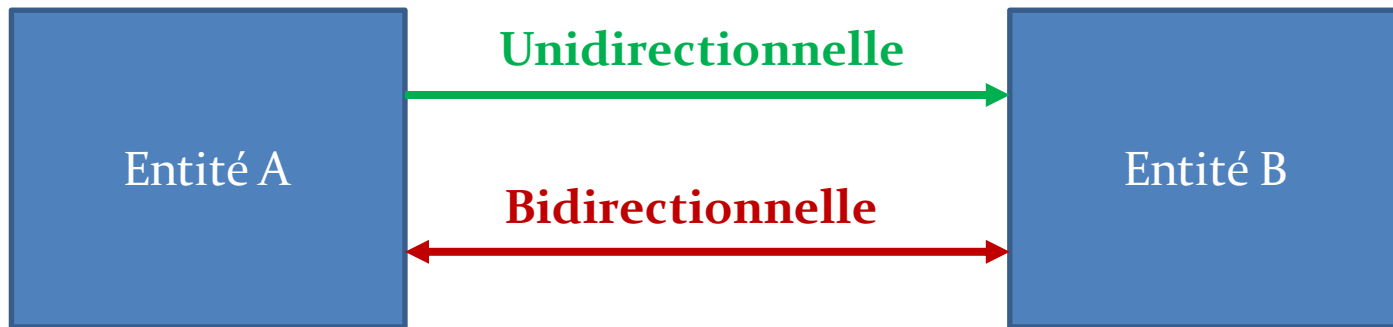
```
public void delete(int id)
{
    Personne p = em.find(Personne.class, id);
    em.refresh(p);
}
```

Recherche par clé primaire

Appeler la méthode « refresh »

Gestion des relations entre les tables

- Dans le modèle des bases de données relationnelles, les tables peuvent être liées entre elles grâce à des relations (ou associations)
- Les relations peuvent avoir différentes cardinalités :
 - 1-1 (one-to-one)
 - 1-N (one-to-many)
 - N-1 (many-to-one)
 - N-N (many-to-many)
- Chacune de ces relations peut être **unidirectionnelle** ou **bidirectionnelle** sauf **one-to-many** et **many-to-one** qui sont par définition bidirectionnelles.
- Dans le cas unidirectionnel, l'une des deux entités doit être **maître** et l'autre **esclave**,
- Dans les deux cas « 1-N » et « N-1 », l'entité du côté **1** est l'entité **esclave**.



Relation 1-1 unidirectionnelle

- Une « **Personne** » possède une seule « **Identite** » et une « **Identite** » ne peut être relative qu'à une seule « **Personne** ».
- Une « **Personne** » peut consulter son identité et le sens inverse n'est pas permis.
- Nous avons donc bien une relation **1-1 unidirectionnelle**



L'entité « **Identite** » ne peut pas connaître la personne y associée

- Dans ce cas, « **Personne** » est l'entité maître, donc elle maintient la relation:

```
@Entity
@Table(name="Person")
public class Personne
{
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int matricule;
    private String nom;
    private String prenom;

    @Transient
    private String nom_prenom;

    @OneToOne
    private Identite identite;
    .....
}
```

```
@Entity
public class Identite
{ @Id
  @GeneratedValue(strategy =
    GenerationType.AUTO)
  private int id;
  .....
}
```

#	Nom	Type	Interclassement
<input type="checkbox"/> 1	<u>matricule</u>	int(11)	Clé étrangère
<input type="checkbox"/> 2	nom	varchar(255)	latin1_swedish_ci
<input type="checkbox"/> 3	prenom	varchar(255)	latin1_swedish_ci
<input type="checkbox"/> 4	identite_id	int(11)	

L'annotation **@OneToOne** associé à l'attribut **identite** sera converti dans la table « **Produit** » en une clé étrangère « **identite_id** » référençant la colonne **id**(clé primaire) de la table « **Identite** »

Utilisation d'une relation 1-1 unidirectionnelle

```
public Personne save (String nom, String prenom, int cin, String adresse)
{
    Identite i = new Identite ( cin, adresse);
    em.persist(i);
    Personne p = new Personne(nom, prenom);
    p.setIdentite(i);
    em.persist(p);
    return p;
}
```

Insérer tout d'abord
l'entité « esclave »

Affecter la valeur de
l'entité « esclave »

```
public Personne updateAdressePersonne(int matriculePersonne, String adresse)
{
    Personne p = em.find(Personne.class, matriculePersonne);
    p.getIdentite().setAdresse(adresse);
    return p;
}
```

Référencer l'entité
« maître » par clé
primaire puis appeler
l'entité « esclave » pour
réaliser la modification

```
public Identite updateAdresseIdentite(int idIdentite, String adresse)
{
    Identite i= em.find(Identite.class, idIdentite);
    i.setAdresse(adresse);
    em.merge(i);
    return i;
}
```

Référencer l'entité
« esclave » par clé
primaire puis réaliser la
modification et appeler la
méthode « merge »

Relation 1-1 bidirectionnelle

- Chaque entité peut accéder à l'autre (à travers un accesseur getter)



- Toujours l'entité « maitre » est la propriétaire de la relation, elle présente un attribut transformé en une clé étrangère dans la BD
- L'entité esclave doit préciser un champ retour par une annotation `@OneToOne` et un attribut `mappedBy` qui doit référencer le champ qui porte la relation côté maître.
- Ce champ ne génère pas une clé étrangère, mais permet de une requête est lancée sur la base pour réaliser une jointure

```
@Entity
@Table(name="Person")
public class Personne
{
    @Id
    @GeneratedValue (strategy = GenerationType.AUTO)
    private int matricule;
    private String nom;
    private String prenom;

    @Transient
    private String nom_prenom;

    @OneToOne
    private Identite identite;
    .....
}
```

```
@Entity
public class Identite {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private int cin;

    private String adresse;

    @OneToOne (mappedBy = "identite")
    private Personne personne;
}
```

référer la relation dans la classe « Personne »

Relation 1-N et N-1

- Une « **Personne** » possède un ou plusieurs « **Compte** » et un « **Compte** » ne peut être relatif qu'à une seule « **Personne** ». Nous avons donc bien une relation 1-N



- Dans ce cas, « **Personne** » est l'entité « esclave » et présente l'annotation « **@OneToMany** ».
- De l'autre côté, la classe « **Compte** » est l'entité « maître » qui maintient la relation avec l'annotation « **@ManyToOne** » et qui contient un champ transformé en clé étrangère dans la BD.
- L'entité « **Personne** » présente une « **Collection** » de type « **Compte** »

```
@Entity
@Table(name="Person")
public class Personne
{
    @Id
    @GeneratedValue (strategy = GenerationType.AUTO)
    private int matricule;
    private String nom;
    private String prenom;

    @OneToMany (mappedBy = "personne")
    private Collection <Compte> comptes = new
    ArrayList<Compte>();
}
```

référer la relation dans la classe « **Compte** »

```
@Entity
public class Compte
{
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private int code;
    private double solde;
    @ManyToOne
    private Personne personne;
}
```

Clé étrangère

#	Nom	Type
<input type="checkbox"/> 1	<u>id</u>	int(11)
<input type="checkbox"/> 2	code	int(11)
<input type="checkbox"/> 3	solde	double
<input type="checkbox"/> 4	personne_matricule	int(11)

Relation N-N unidirectionnelle

- Une « **Personne** » réalise un ou plusieurs « **Vol** » et un « **Vol** » regroupe une ou plusieurs « **Personne** ». Nous avons donc bien une relation **N-N**
- Dans le cas d'une relation unidirectionnelle (**@ManyToMany**) maintenue par l'entité « **Personne** », Une « **Personne** » peut accéder à ses vols mais un « **Vol** » ne peut déterminer les personnes associées.



- La façon classique d'enregistrer ce modèle en base consiste à créer une table de jointure « **personne_vols** » qui comporte deux clés étrangères:

- o **personne_matricule**: référence la table « **Personne** »
- o **vols_id**: référence la table « **Vol** »

```
@Entity
public class Personne
{
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int matricule;
    private String nom;
    private String prenom;

    @ManyToMany
    private Collection<Vol> vols = new ArrayList<Vol>();
}
```

Etablir une relation (N-N)
dans la classe « Vol »

```
@Entity
public class Vol
{
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private int code;
}
```

<input type="checkbox"/>	personne	★
<input type="checkbox"/>	personne_vols	★
<input type="checkbox"/>	vol	★

#	Nom
1	personne_matricule 🔑
2	vols_id 🔑

Relation N-N bidirectionnelle

- Une « **Personne** » réalise un ou plusieurs « **Vol** » et un « **Vol** » regroupe une ou plusieurs « **Personne** ». Nous avons donc bien une relation **N-N**



- Dans le cas d'une relation bidirectionnelle (**@ManyToMany**) maintenue par l'entité « **Personne** », Une « **Personne** » peut accéder à ses vols et un « **Vol** » peut déterminer les personnes associées.
- On ajoute une autre annotation (**@ManyToMany**) dans l'entité « **Vol** » sur une collection de « **Personne** » et en utilisant l'attribut « **mappedBy** » pour référence la relation dans l'entité « **Personne** »

```
@Entity
public class Personne
{
    @Id
    @GeneratedValue (strategy = GenerationType.AUTO)
    private int matricule;
    private String nom;
    private String prenom;

    @ManyToMany
    private Collection<Vol> vols = new ArrayList<Vol>();
}
```

Référencer la relation dans la classe « Personne »

```
@Entity
public class Vol
{
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private int code;

    @ManyToMany (mappedBy = "vols")
    private Collection<Personne> presonnes = new ArrayList<Personne>();
}
```

Comportement en cascade

- Le comportement **cascade** consiste à spécifier ce qui se passe pour une entité en relation d'une entité mère lorsque cette entité mère subit une des opérations définies ci-dessus.
- Le comportement cascade est précisé par l'attribut **cascade**, disponible sur les annotations : **@OneToOne**, **@OneToMany** et **@ManyToMany** (et non pour **@ManyToOne**)
- La valeur de l'attribut est une énumération de type **CascadeType** ayant les principales valeurs suivantes:
 - MERGE** : cascade en cas de « merge »
 - PERSIST** : cascade en cas de « persist »
 - REMOVE**: cascade en cas de « remove »
 - ALL** : correspond à toutes les valeurs à la fois.

```
@Entity
public class Personne
{
    @Id
    @GeneratedValue (strategy = GenerationType.AUTO)
    private int matricule;
    private String nom;
    private String prenom;

    @OneToOne (cascade ={CascadeType.PERSIST,
        CascadeType.REMOVE, CascadeType.MERGE})
    private Identite identite;
```

```
@Entity
public class Personne
{
    @Id
    @GeneratedValue (strategy = GenerationType.AUTO)
    private int matricule;
    private String nom;
    private String prenom;

    @OneToOne (cascade =CascadeType.ALL)
    private Identite identite;
```


Mode de récupération d'une collection

- Pour récupérer les éléments d'une collection contenu dans une entité, nous disposons de deux modes:
 - **EAGER**: effectuer la récupération des éléments de la collection, dès que l'on récupère l'objet et donc on initialise la collection. C'est le Fetch Type "eager" (**fetch=FetchType.EAGER**).
 - **LAZY**: (par défaut) effectuer la récupération des éléments de la collection à la demande, c'est à dire dès que l'on aura besoin de la collection. C'est le Fetch Type "lazy" (**fetch=FetchType.LAZY**).
- Le mode « **LAZY** » est le mode recommandé pour ne pas faire des requêtes inutiles vers la base de données surtout en cas de non besoin d'utiliser la collection

```
@Entity
@Table(name="Person")
public class Personne
{
    @Id
    @GeneratedValue (strategy = GenerationType.AUTO)
    private int matricule;
    private String nom;
    private String prenom;

    @OneToMany (mappedBy = "personne", fetch=FetchType.EAGER)
    private Collection <Compte> comptes = new ArrayList<Compte>();
}
```

Charger la collection des comptes
au moment du chargement de l'entité
« Personne »



Héritage et JPA

- Les bases de données relationnelles n'ont pas une fonctionnalité permettant de traduire ou mapper une hiérarchie de classe en tables de bases de données
- L'héritage est un des concepts clefs en java en particulier et de la programmation orientée objet en général. **JPA** et ses différentes implémentations avec doivent donc fournir un moyen pour traduire ce concept clef en un concept compréhensible par les bases de données relationnelles.
- PA propose plusieurs stratégies :
 - ❖ **MappedSuperclass** : les classes parentes ne sont pas des entités.
 - ❖ **Single Table** : les entités de la hiérarchie de classe sont placées dans une seule table.
 - ❖ **Joined Table** : chaque classe a sa table et effectuer une requête sur une sous-classe de la hiérarchie implique de faire une jointure sur les tables.
 - ❖ **Table-Per-Class** : une table par classe.
- Chacune des stratégies implique une structure différente de la base de données.

MappedSuperclass

- Cette stratégie permet de partager les propriétés entre plusieurs entités.
- Elle permet de mapper chaque classe vers une table dédiée.
- La classe mère sur laquelle est définie la stratégie **MappedSuperclass** n'est pas une entité et aucune table ne sera créée dans la BD pour elle..

```
@MappedSuperclass  
  
public class Personne {  
  
    @Id  
    protected long Id;  
    protected String nom;  
    protected String prenom;  
  
    // constructeur, getters, setters  
}
```

```
@Entity  
  
public class Formateur extends Personne {  
  
    private String matiere;  
  
    // constructeur, getters, setters  
}
```

```
@Entity  
  
public class Etudiant extends Personne {  
  
    private double note;  
  
    // constructeur, getters, setters  
}
```

- Dans la BD, on aura une table **Etudiant** et une table **Formateur** qui en plus de leurs propriétés auront les propriétés de la classe mère comme champs.
- Avec la stratégie **MappedSuperclass**, les classes mères ne peuvent pas définir de relations avec d'autres entités.

Single Table

- C'est la **stratégie par défaut** utilisée par JPA lorsqu'aucune stratégie n'est implicitement définie et que la classe mère de la hiérarchie est une entité.
- Avec cette stratégie, **une seule table est créée** et partagée par toutes les classes de la hiérarchie.
- L'annotation **@Inheritance** est utilisée sur la classe mère pour préciser à JPA la stratégie d'héritage à utiliser.
- Dans cette stratégie, toutes les classes entités sont **mappées dans une unique table**.
- JPA a besoin de faire la différence entre les différences lignes de la table ainsi mappée afin de pouvoir convertir chaque enregistrement vers la classe entité correspondante.
- Pour ce faire, JPA utilise un mécanisme permettant de faire cette différence en créant une colonne appelée **discriminator** qui ne fait pas partie des attributs de l'entité mappée.

```
@Entity
@Inheritance(strategy =
InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name = "TYPE_PERSONNE")
public class Personne {

@Id
protected long id;
protected String nom;
protected String prenom;

// constructeur, getters, setters
}
```

Single Table

- Une colonne sera créée dans la table générée ayant pour valeur **TYPE_PERSONNE** pour différencier les enregistrements des entités Etudiant et Formateur.
- Par défaut, les valeurs de la colonne **TYPE_PERSONNE** seront les noms des classes filles.
- Pour préciser le nom à enregistrer, on ajoute l'annotation **@DiscriminatorValue** sur les classes filles.

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

    private double note;

    // constructeur, getters, setters
}
```

```
@Entity
@DiscriminatorValue("form")
public class Formateur extends Personne
{

    private String matiere;

    // constructeur, getters, setters
}
```


JOINED

- Cette stratégie consiste à enregistrer les champs de chaque entité dans une table propre à cette classe.
- On a donc autant de tables que de classes dans notre modèle, abstraites ou concrètes.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@DiscriminatorColumn(name = "TYPE_PERSONNE")
public class Personne {

    @Id
    protected long Id;

    protected String nom;

    protected String prenom;

    // constructeur, getters, setters
}
```

JOINED

```
@Entity
@DiscriminatorValue("etu")

public class Etudiant extends Personne {

    private double note;

    // constructeur, getters, setters

}
```

```
@Entity
@DiscriminatorValue("form")

public class Formateur extends Personne {

    private String matiere;

    // constructeur, getters, setters

}
```

- Enfin on remarque que chacune des deux tables **Etudiant** et **Formateur** comporte une clé primaire: id.
- Cette clé primaire est aussi une clé étrangère qui référence la clé primaire de la table **Personne**.
- Les trois tables partagent en fait la même clé primaire, ce qui est logique puisque toutes les lignes de la tables **Etudiant** ont une partie de leurs champs dans la table **Personne** et de même pour la table **Formateur**

TABLE_PER_CLASS

- Cette stratégie fonctionne à l'inverse de la stratégie SINGLE_TABLE. Plutôt que d'envoyer tous les champs de toutes les entités vers une table unique, on les envoie vers autant de tables qu'il y a de classes concrètes annotées @Entity dans la hiérarchie..

```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
@DiscriminatorColumn(name = "TYPE_PERSONNE")
public class Personne {

    @Id
    protected long Id;

    protected String nom;

    protected String prenom;

    // constructeur, getters, setters
}
```

TABLE_PER_CLASS

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

    private double note;

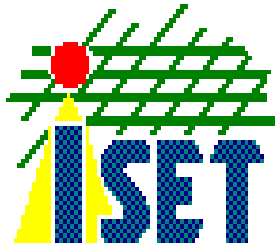
    // constructeur, getters, setters
}
```

```
@Entity
@DiscriminatorValue("form")
public class Formateur extends Personne {

    private String matiere;

    // constructeur, getters, setters
}
```

- On a donc autant de tables dans notre schéma que de classes concrètes annotées Entity dans la hiérarchie
- On remarque que les tables **Personne**, Etudiant et Formateur sont toujours présentes, c'est leur contenu qui change.
- Chacune des deux tables **Etudiant** et **Formateur** comporte maintenant deux colonnes nom et prenom, elle deviennent indépendantes de la table **Personne**. De fait, sa clé primaire n'est plus une clé étrangère.



Mastère Professionnel

Développement des Systèmes
Informatiques et Réseaux (DSIR)



Micro services

03- Notion de service web REST

Mohamed ZAYANI

2023/2024

Plan

4. Service web RESTful

1-Définition

2-Caractéristiques

3-Représentation

4-Bibliothèque
JAX-RS

Complexité de SOAP

- Le protocole SOAP est extrêmement sophistiqué **utilise beaucoup de ressources** puisqu'il est nécessaire de générer et de traduire des documents XML qui transitent au travers du protocole HTTP.
- Le protocole SOAP nécessite de construire, à la fois côté client et aussi côté serveur, un certain nombre de classes pour **réaliser ces différents décodages** pour retrouver les informations encapsulées dans le document XML.

Nouvelle approche: REST

- Une nouvelle approche et non un standard.
- Elle consiste à **utiliser directement le protocole HTTP sans couche supplémentaire**, à l'aide cette fois-ci d'un web service dénommé, service web REST.
- Cette approche est **plus simple à implémenter**.
- Par ailleurs, les services web, dans ce cas là, sont plutôt associés à **gérer des ressources distantes** avec toutes les phases classiques, de création, de récupération, de modification et de suppression (CRUD).

Définition

- REST est l'acronyme de **RE**presentational **S**tate Transfert défini dans la thèse de Roy Fielding en 2000.
- **REST** n'est pas un protocole ou un format, **contrairement à SOAP**, HTTP ou RCP, mais **un style d'architecture** inspiré de l'architecture du web fortement basé sur le protocole HTTP

 REST est:

- ❖ Un système d'architecture
- ❖ Une approche pour construire une application
- Les applications qui respectent l'architecture REST sont dites **RESTful**

Caractéristiques

- Les services REST sont **sans état** (Stateless):
 - ❖ Chaque requête envoyée au serveur doit contenir toutes les informations relatives à son état et est traitée **indépendamment** de toutes autres requêtes.
 - ❖ **Minimisation des ressources systèmes** (pas de gestion de session, ni d'état).
- Les architectures RESTful sont construites à partir de **ressources** uniquement identifiées par **des URI(s)** (Uniform Resource Identifiers).
- Chaque ressource peut subir **quatre** opérations qui correspondent aux principales **méthodes (ou verbes)** HTTP
 - **GET** (pour la lecture),
 - **POST** (pour la création),
 - **PUT** (pour la mise à jour),
 - **DELETE** (pour la suppression)

REST est orienté « ressources »

- Dans l'architecture REST, toute information est **une ressource**.
- Chaque ressource est désignée par une **URI** (généralement un lien sur le Web). Une URI est un identifiant unique formé d'un nom et d'une adresse indiquant où trouver la ressource
- Les ressources sont manipulées par un ensemble d'**opérations** simples et bien définies (GET, POST, DELETE, PUT)



*Ces principes encouragent la **simplicité**, la **légèreté** et l'**efficacité** des applications.*

Méthode GET

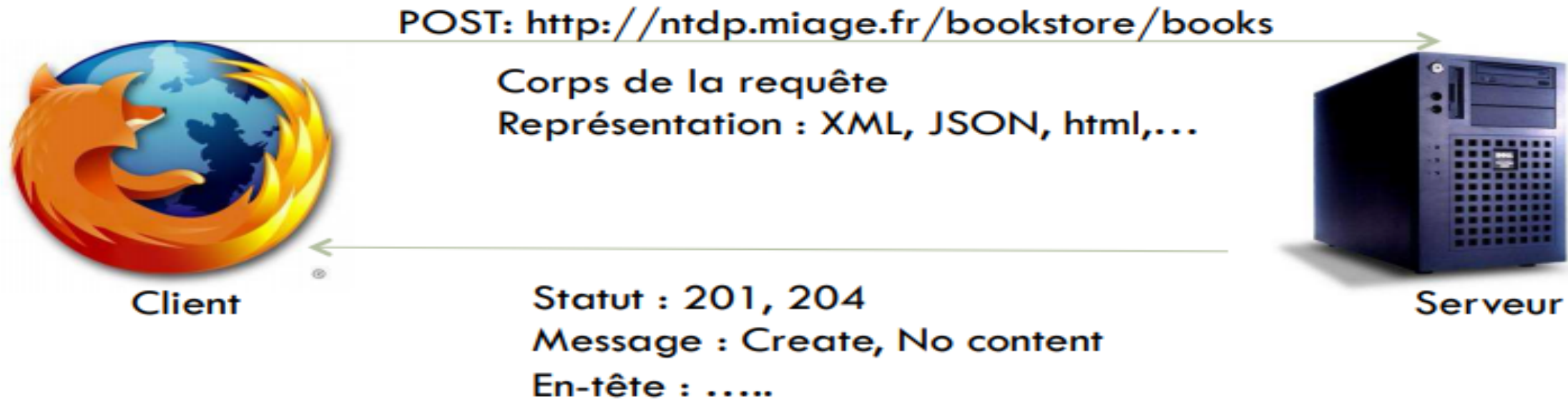
La méthode GET renvoie une représentation de la ressource tel qu'elle est sur le système



GET est une méthode de lecture demandant une représentation d'une ressource. GET doit être implémentée de sorte à ne pas modifier l'état de la ressource.

Méthode POST

La méthode POST crée une nouvelle ressource sur le système



POST crée une nouvelle ressource subordonnée à une ressource principale identifiée par l'URI demandée.
POST modifie donc l'état de la ressource.

Méthode DELETE

Supprime la ressource identifiée par l'URI sur le serveur



DELETE supprime une ressource. La réponse à DELETE peut être un message d'état dans le corps de la réponse ou aucun code du tout.

Méthode PUT

Mise à jour de la ressource sur le système



PUT modifie l'état de la ressource stockée à une certaine URI. Si l'URI de la requête fait référence à une ressource inexistante, celle-ci sera créée avec cette URI.

Représentation

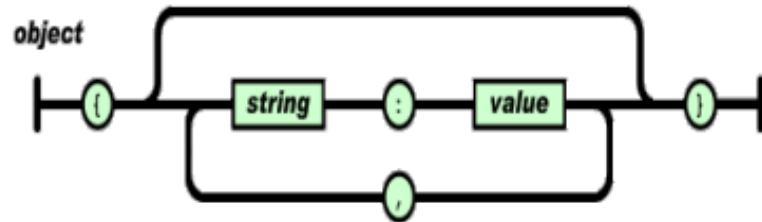
- Un client traite toujours une ressource au travers de sa représentation.
- Une représentation désigne **les données échangées** entre le client et le serveur pour une ressource.
- Cette représentation peut être sous différents formats:
 - ✓ JSON
 - ✓ XML
 - ✓ XHTML
 - ✓ CSV
 - ✓ Text/plain

Format JSON

- **JSON** « **J**ava **S**cript **O**bject **N**otation » est un format **léger** pour l'échange de données, **facile** à lire par un humain et interpréter par une machine.
- Basé sur JavaScript, il est complètement **indépendant des langages de programmation** mais utilise des conventions qui sont communes à toutes les langages de programmation (C, C++, Perl, Python, Java, C#, VB, JavaScript,....)
- Il existe **deux** structures :
 - ✓ Une collection de clefs/valeurs → **Object**
 - ✓ Une collection ordonnée d'objets → **Array**

Object – JSON

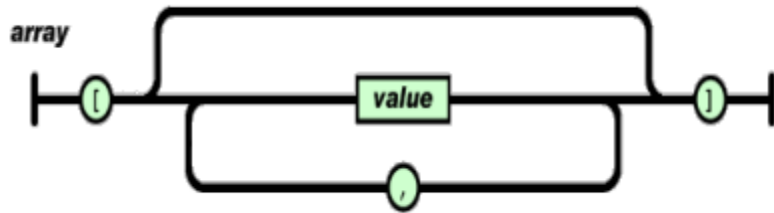
- Un objet JSON commence par un « { » et se termine par « } » et composé d'une liste non ordonnée de paire **clefs/ valeurs**. Une clef est suivie de « : » et les paires clef/ valeur sont séparés par « , »



```
{ "id": 51,  
  "nom": "Mathematiques 1", "resume":  
  "Resume of math ", "isbn": "123654",  
  "categorie":  
    {  
      "id": 2, "nom": "Mathematiques",  
      "description": "Description of  
      mathematiques "  
    },  
  "quantite": 42,  
  "photo": ""  
}
```

Array–JSON

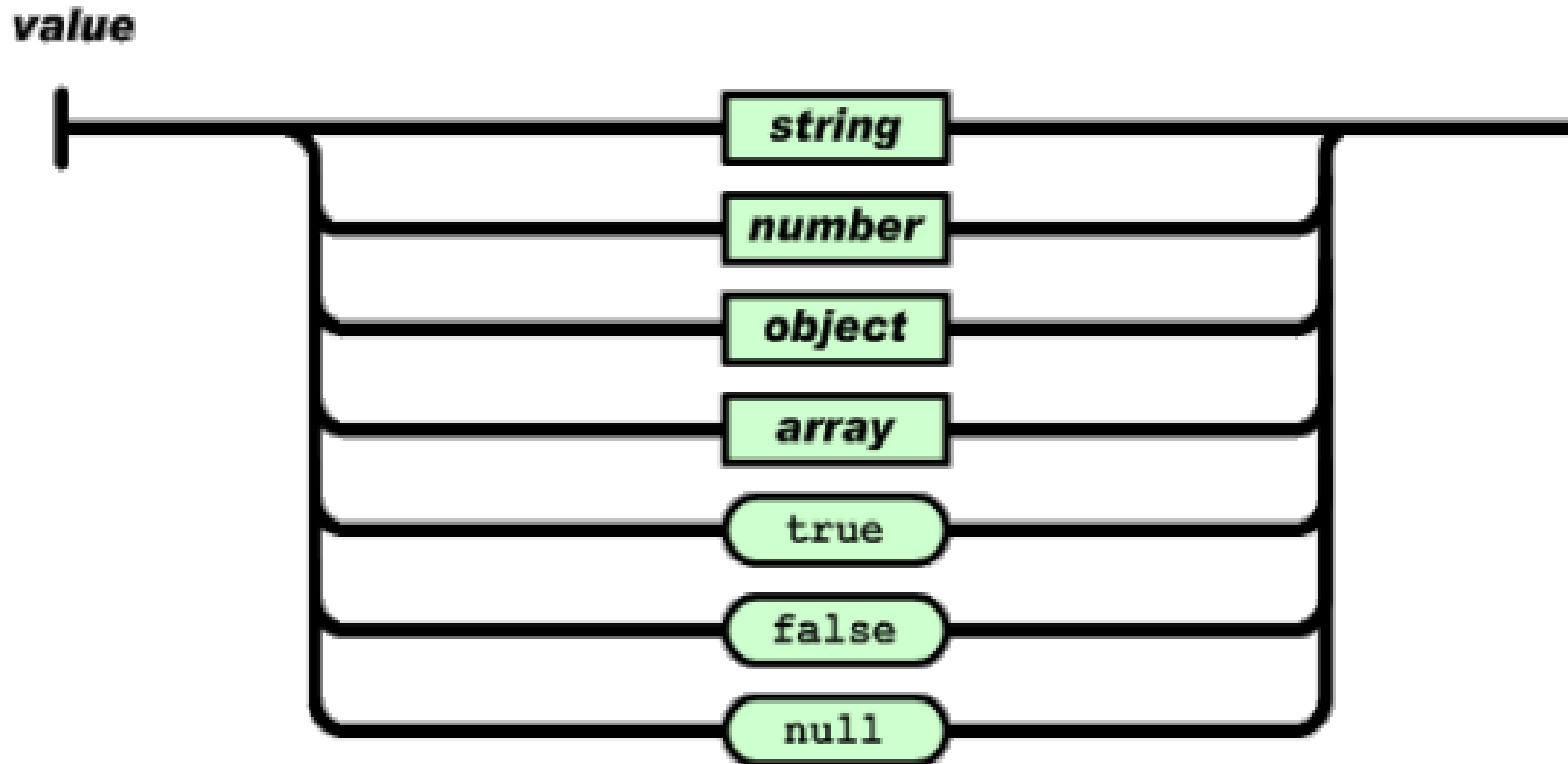
- Un array JSON est une liste ordonnée de valeurs commençant par « [» et se terminant par «] ».
- Les valeurs sont séparées l'une de l'autre par « , ».



```
[  
  { "id": 51,  
    "nom": "Mathematiques 1",  
    "resume": "Resume of math ",  
    "isbn": "123654",  
    "quantite": 42,  
    "photo": ""  
  },  
  { "id": 102,  
    "nom": "Mathematiques 1",  
    "resume": "Resume of math ",  
    "isbn": "12365444455",  
    "quantite": 42,  
    "photo": ""  
  }  
]
```

Value – JSON

- Une valeur peut être soit un string entre «`""`» ou un
- nombre (**entier**, **décimal**) ou un **boolean** (true, false) ou **null** ou un object.



WADL

- Par analogie à WSDL pour les services web SOAP.
- **WADL** est l'acronyme de **W**eb **A**pplication **D**escription **L**anguage.
- C'est un standard du W3C qui permet de décrire les éléments des services web REST:
 - **Resource,**
 - **Méthode,**
 - **Paramètre,**
 - **Réponse**
- **WADL** permet d'interagir de manière dynamique avec les applications REST

Exemple WADL

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<application xmlns="http://wadl.dev.java.net/2009/02">
  <doc xmlns:jersey="http://jersey.java.net/" jersey:generatedBy="Jersey: 2.0 2013-05-03 14:50:15"/>
  <grammars/>
  ▼<resources base="http://localhost:8080/Bibliotheque/webresources/">
    ▼<resource path="category">
      ▼<method id="test" name="GET">
        ▼<response>
          <representation mediaType="application/xml"/>
          <representation mediaType="application/json"/>
        </response>
      </method>
      ▼<method id="apply" name="OPTIONS">
        ▼<request>
          <representation mediaType="*/*/"/>
        </request>
        ▼<response>
          <representation mediaType="application/vnd.sun.wadl+xml"/>
        </response>
      </method>
      ▼<method id="apply" name="OPTIONS">
        ▼<request>
          <representation mediaType="*/*/"/>
        </request>
        ▼<response>
          <representation mediaType="text/plain"/>
        </response>
      </method>
      ▼<method id="apply" name="OPTIONS">
        ▼<request>
          <representation mediaType="*/*/"/>
        </request>
        ▼<response>
          <representation mediaType="*/*/"/>
        </response>
      </method>
    </resources>
  ▼<resource path="application.wadl">
    ▼<method id="getWadl" name="GET">
      ▼<response>
        <representation mediaType="application/vnd.sun.wadl+xml"/>
        <representation mediaType="application/xml"/>
      </response>
    </method>
    ▼<method id="apply" name="OPTIONS">
      ▼<request>
        <representation mediaType="*/*/"/>
      </request>
      ▼<response>
        <representation mediaType="text/plain"/>
      </response>
    </method>
    ▼<method id="apply" name="OPTIONS">
      ▼<request>
        <representation mediaType="*/*/"/>
      </request>
      ▼<response>
        <representation mediaType="*/*/"/>
      </response>
    </method>
  </resource>
</application>
```

JAX-RS : Java API pour les services web REST

- Le développeur JAVA n'a pas besoin d'écrire des requêtes HTTP ni de créer manuellement des réponses.
- C'est plutôt **JAX-RS** (API très élégante) qui permet d'écrire une ressource à l'aide de quelques annotations seulement.
- JAX-RS repose sur HTTP et dispose d'un ensemble de classes et d'**annotations** clairement définies pour gérer HTTP et les URI.
- Une ressource pouvant avoir plusieurs représentations, l'API permet de gérer un certain nombre de types de contenu et utilise **JAXB** pour sérialiser et désérialiser les représentations XML et JSON en objets.

Exemple avec JAX-RS

```
package rest;

import javax.ws.rs.*;

@Path("/")
@Produces("text/plain")
public class Conversion {
    private final double TAUX = 6.55957;

    @GET
    public String bienvenue() {
        return "Web service de conversion entre les euros et les francs";
    }

    @GET
    @Path("/{euro}€") // @Path("/{franc}/{euro}")
    public String euroFranc(@PathParam("euro") double euro) {
        return "" + (euro * TAUX);
    }

    @GET
    @Path("/{franc}F") // @Path("/{euro}/{franc}")
    public String francEuro(@PathParam("franc") double franc) {
        return "" + (franc / TAUX);
    }
}
```

Pour définir un service web à la racine de l'application web

Format de présentation

Pour permettre le traitement des requêtes HTTP GET

Pour définir le path de la méthode

Pour extraire le paramètre de la requête HTTP



- x **Conversion** étant une classe Java annotée par **@Path**, la ressource sera hébergée à l'URI « / », la racine de l'application web.
- x Les méthodes **bienvenue()**, **euroFranc()** et **francEuro()** sont elles-mêmes annotées par **@GET** afin d'indiquer qu'elles traiteront les requêtes HTTP GET.
- x Ces méthodes produisent du texte. Le contenu est identifié par le type MIME « **text/plain** » grâce à l'annotation **@Produces**.
- x Pour accéder aux ressources, il suffit d'un client HTTP, simple navigateur par exemple, pouvant envoyer une requête GET vers l'URL <http://localhost:8080/ConversionREST/>.

Extraction des paramètres: @PathParam

- **@PathParam** : Cette annotation permet d'extraire la valeur du paramètre d'une requête. Il s'agit d'intégrer dans syntaxe de l'URI **un nom de variable entouré d'accolades** : ces variables seront ensuite évaluées à l'exécution.
- Le code suivant permet d'extraire la valeur en euro présente dans l'URI afin d'effectuer la conversion et de fournir la valeur numérique de l'équivalent en franc sous forme de simple chaîne de caractères (non interprété).

```
@GET
@Path("/{euro}€") // @Path("/franc/{euro}")
public String euroFranc(@PathParam("euro") double euro) {
    return "" + (euro * TAUX);
}
```

The screenshot shows a Mozilla Firefox browser window. The address bar contains the URL `localhost:8080/ConversionREST/187000€`. A blue box labeled "Racine de l'application web" points to the `/ConversionREST` part of the URL. The page content displays the number `1226639.59`. A blue box labeled "Valeur du paramètre « euro »" points to the `187000` part of the URL. Another blue box labeled "Valeur du résultat" points to the `1226639.59` displayed on the page.

Extraction des paramètres: @QueryParam

- **@QueryParam** : Cette annotation permet d'extraire la valeur d'un paramètre modèle d'une URI.
- Il s'agit ici d'une **utilisation plus classique** de la récupération de paramètres au moyen de la syntaxe usuelle prévue par la méthode GET HTTP.

```
@GET
@Path("/franc")
public String euroFranc(@QueryParam("euro") double euro) {
    return "" + (euro * TAUX);
}
```

The screenshot shows a Mozilla Firefox browser window. The address bar displays the URL `localhost:8080/ConversionREST/franc?euro=25`. A blue box labeled "Path de la méthode" points to the `/franc` part of the URL. The page content shows the result `163.98925`, with a blue box labeled "Valeur du résultat" pointing to it. The parameter `euro=25` in the query string is highlighted, with a blue box labeled "Valeur du paramètre « euro »" pointing to it.

Extraction des paramètres: @DefaultValue

- **@DefaultValue**: Il est possible d'ajouter cette annotation à toutes celles que nous venons de découvrir pour définir une valeur par défaut pour le paramètre que nous attendons.
- Cette valeur sera utilisée si les métadonnées correspondantes sont absentes de la requête.

The image shows a Java code snippet for a REST API endpoint and a screenshot of a web browser displaying the result of a call to that endpoint.

Code Snippet:

```
@GET
@Path("franc")
public String euroFranc(@DefaultValue("15.24") @QueryParam("euro") double euro) {
    return "" + (euro * TAUX);
}
```

Annotations and Parameters:

- @GET**: HTTP method annotation.
- @Path("franc")**: Path annotation for the REST endpoint. A callout box points to this annotation with the text "Path de la méthode".
- @DefaultValue("15.24")**: Annotation for the `euro` parameter, providing a default value of 15.24.
- @QueryParam("euro")**: Annotation for the `euro` parameter, indicating it is a query parameter.
- double euro**: The parameter type and name.

Browser Screenshot:

The screenshot shows a Mozilla Firefox browser window. The address bar displays `localhost:8080/ConversionREST/franc`. A callout box points to the `/franc` part of the URL with the text "Path de la méthode". The page content displays the result of the calculation: `99.9678468`. A callout box points to this result with the text "Valeur du résultat".