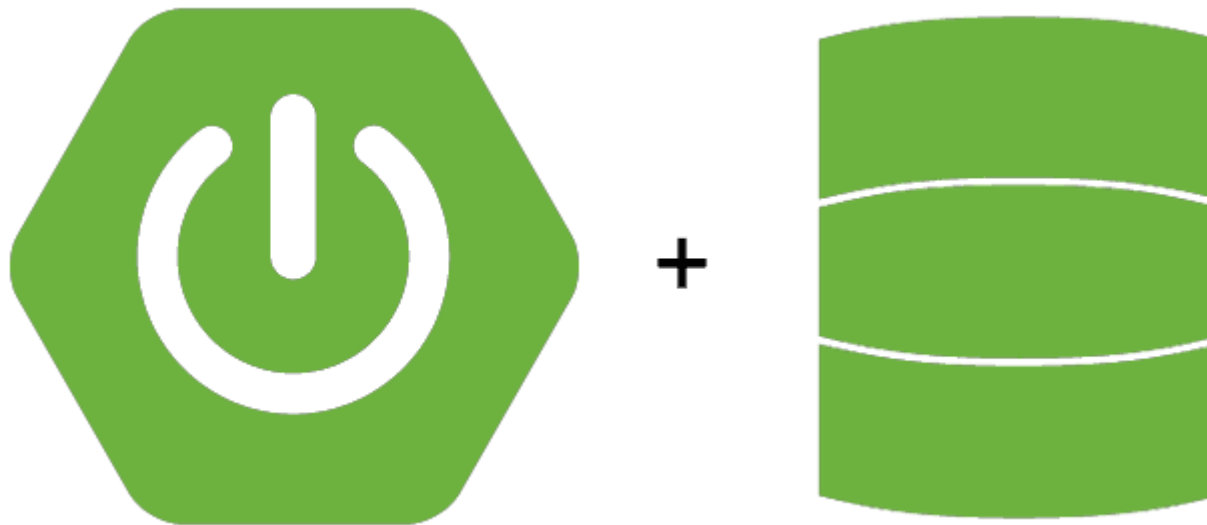


# SPRING DATA JPA – PREMIÈRE ENTITÉ



**UP ASI**  
**Bureau E204**

# Plan du Cours

- Persistance
  - JDBC
  - ORM
  - JPA
  - Hibernate
  - Spring Data
  - Spring Data JPA
  - Entité
  - Annotations
- 
- Configuration d'une DataSource avec Spring Boot
  - TP Spring Boot + Spring Data JPA : 1<sup>ère</sup> Entité

# PERSISTANCE

- **Le Modèle Relationnel** : les systèmes de gestion de bases de données relationnels (**SGBDR**) sont devenus un pilier incontournable dans le développement d'applications.
- **Le modèle Object** : se base sur la programmation orientée objet. Celle-ci permet aux applications d'atteindre un excellent niveau de qualité et de flexibilité.
- Comment stocker les objets modélisés de la mémoire vers les SGBDR ?
- On parle ainsi de **persistance d'objets métiers**.

# PERSISTANCE

- Pour persister les données :
  - Développer manuellement le code de projection des objets sur le support relationnel (Utilisation de JDBC natif).

Ou :

- Utiliser un Framework de projection objet-relationnel : EclipseLink, Hibernate , **Spring Data JPA** ... Ces Frameworks implémentent la spécification **JakartaEE JPA** (anciennement JavaEE).

# JDBC

- **JDBC** (Java DataBase Connectivity) est une interface de programmation créée par Sun Microsystems (racheté par Oracle Corporation).
- C'est une **API** (Application Programming Interface) pour les programmes utilisant la plateforme Java.
- Elle permet aux applications Java d'accéder à des bases de données, en utilisant des pilotes JDBC (Drivers).
- Des pilotes JDBC sont disponibles pour tous les systèmes connus de bases de données relationnelles.

# JDBC

- L'utilisation de l'**API JDBC** était une solution pour les développeurs java pour manipuler les données dans une base de donnée SQL (CRUD).
- **Inconvénients :**
  - Pas de séparation entre le code technique et le code métier.
  - utilisation du langage SQL rend la couche d'accès aux données difficilement maintenable.
- Exemple :

# JDBC

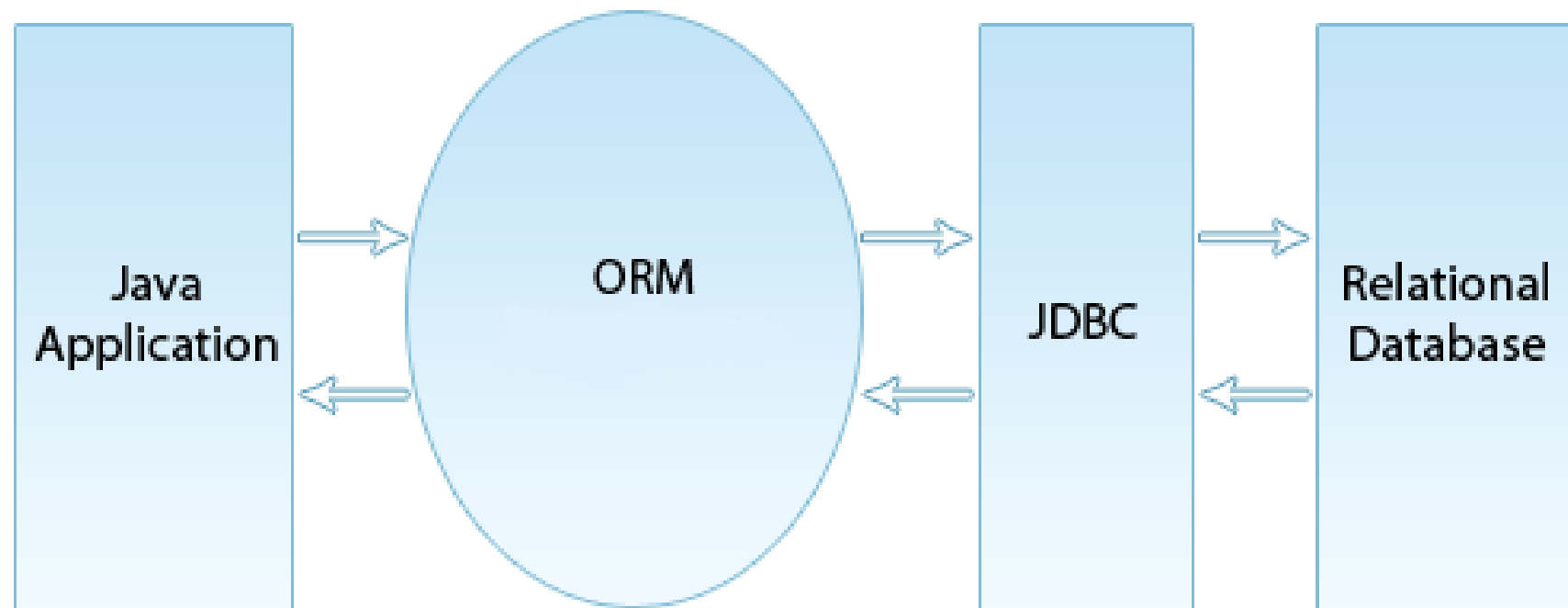
```
public void addEtudiant(Etudiant etudiant) {  
    Connection conn = null; PreparedStatement preparedStmt = null;  
  
    try {  
        conn = DriverManager.getConnection(DB_URL, USER, PASSWORD);  
  
        String insertSQL = "INSERT INTO Etudiant (idEtudiant, prenomE, nomE, option) VALUES  
            (?, ?, ?, ?)";  
        preparedStmt = conn.prepareStatement(insertSQL);  
        preparedStmt.setInt(1, etudiant.getIdEtudiant());  
        preparedStmt.setString(2, etudiant.getPrenomE());  
        preparedStmt.setString(3, etudiant.getNomE());  
        preparedStmt.setString(4, etudiant.getOption());  
        preparedStmt.execute();  
    }  
    catch (SQLException e) { e.printStackTrace(); }  
    finally { try { preparedStmt.close(); } catch (SQLException e) { e.printStackTrace(); }  
}
```

# ORM

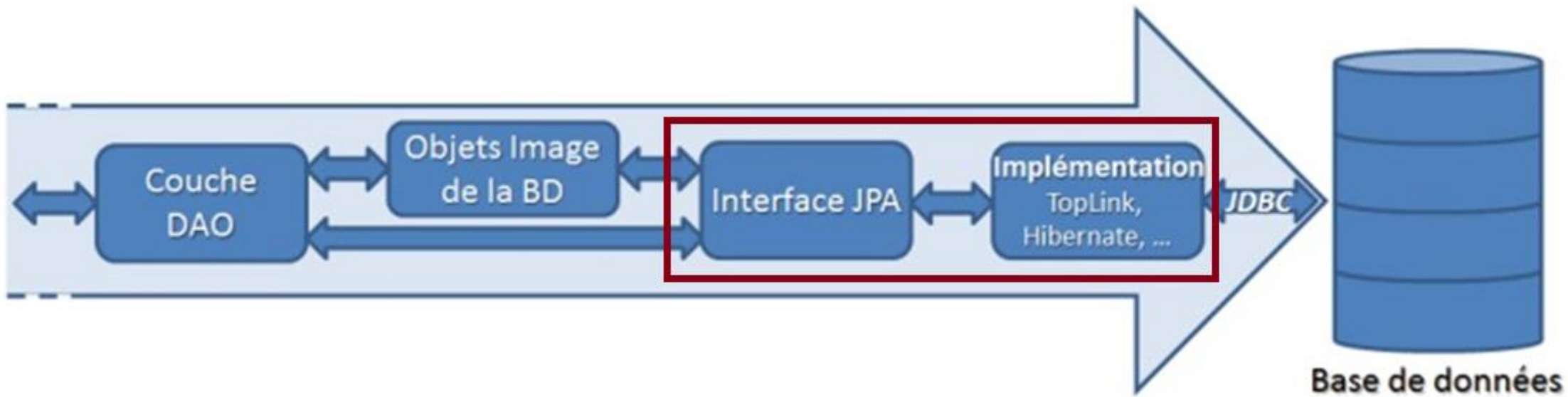
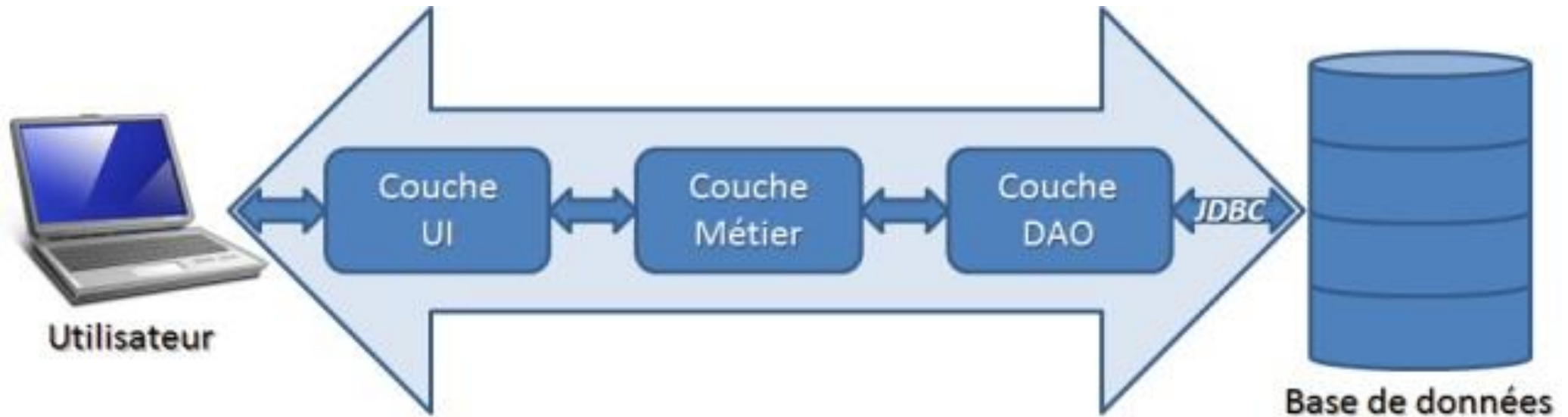
- Pour éviter les inconvénients liés à l'utilisation de l'API JDBC nativement, le concept d'**ORM** a vu le jour.
- Le **M**apping **O**bjets  $\leftrightarrow$  **R**elationnel (ORM) est une technique de programmation, qui permet d'associer une ou plusieurs classes avec une table, et chaque attribut de la classe avec un champ de la table.
- Elle vise à réduire la quantité de code produit par l'API JDBC (les opérations sont les mêmes : Connexion + CRUD + Déconnexion).



# ORM - JDBC



# ORM - JDBC



# JPA

- Pour normaliser le fonctionnement des ORM, **JPA** a été mis en place.
- La **Jakarta Persistence API** (abrégée en JPA), est une interface de programmation Java permettant de normaliser l'utilisation et la communication avec la couche de données, d'une application Java.

# JPA

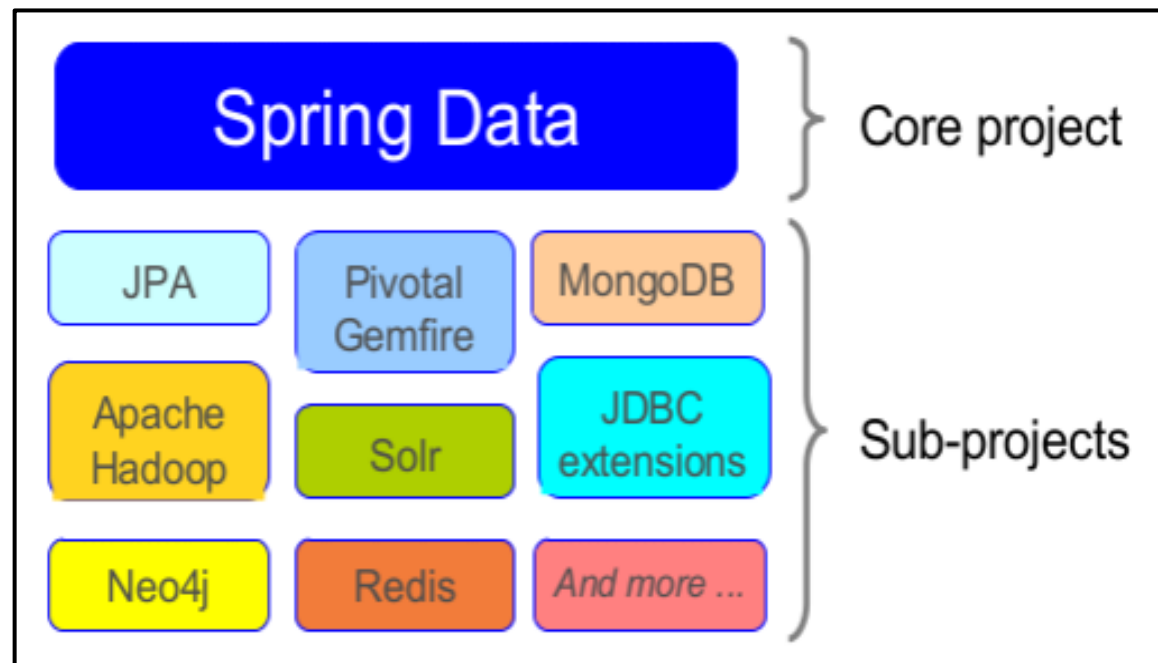
- l'utilisation de **JPA** nécessite **un fournisseur de persistance (ORM)** qui implémente les spécifications JPA (Hibernate, Toplink, ...)
- Dans ce cours nous allons utiliser **Spring Data JPA** qui se base sur **Hibernate** comme implémentation de la spécification **JPA**.
- **JPA** est une **spécification** (normalisation et standardisation de la communication avec la DB).
- **Hibernate** est un produit (**Implémentation** de cette spécification).

# HIBERNATE

- **Hibernate** est un Framework open source gérant la persistance des objets en base de données relationnelle.
- Il gère le Mapping entre les objets de l'application et la base de données.
- C'est un projet maintenu par l'entreprise JBoss, appartenant à RedHat.
- L'utilisation de JPA / HIBERNATE nous permet de gagner du temps en développement :
  - Génération automatique du code SQL (CRUD).
  - Génération automatique des tables à partir des entités Java.

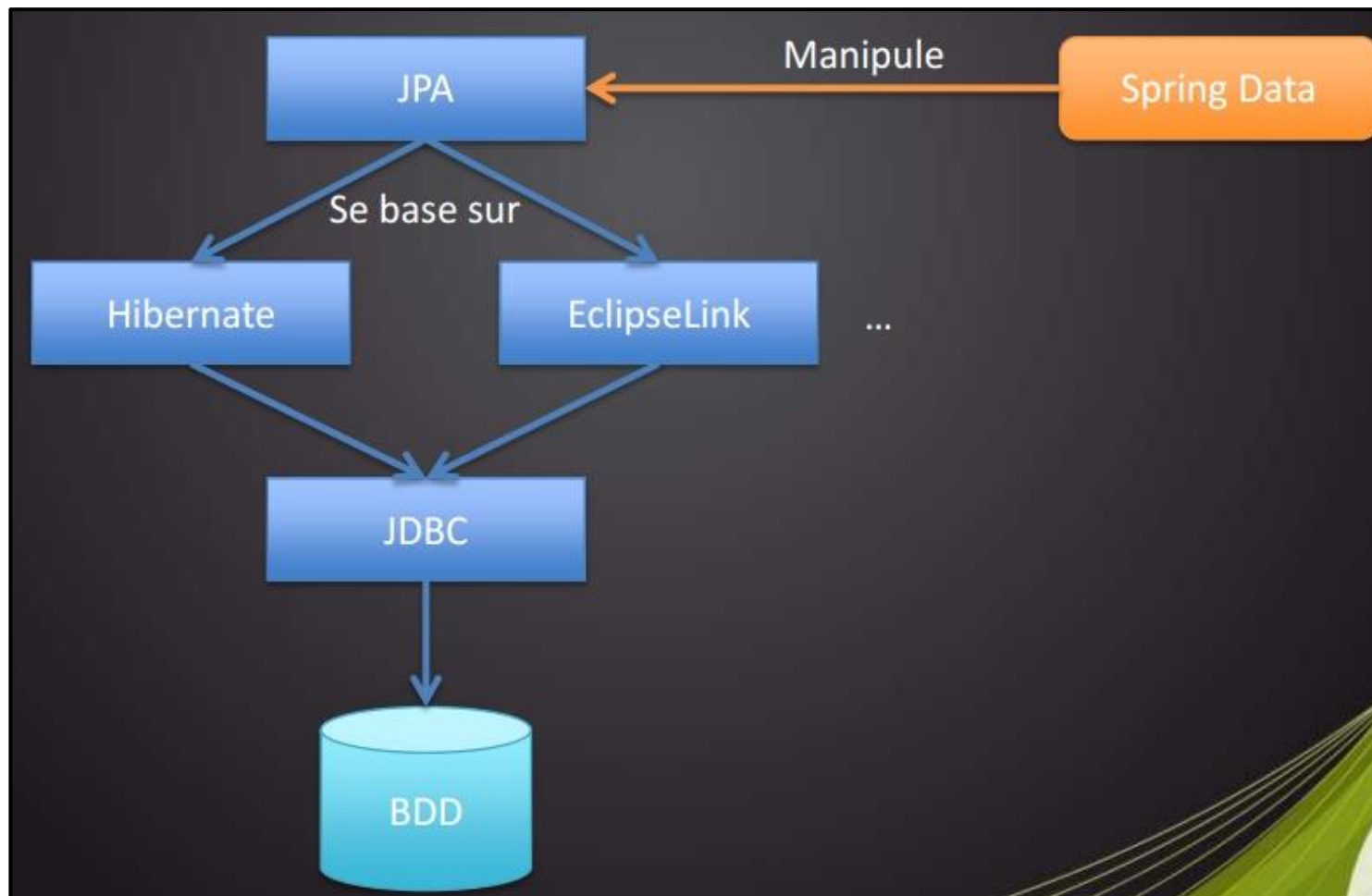
# SPRING DATA

- C'est un module Spring (Pivotal) qui a pour but de :
  - Faciliter l'écriture des couches d'accès aux données.
  - Offrir une abstraction commune pour l'accès aux données quelle que soit la source de données (SQL ou NoSQL).



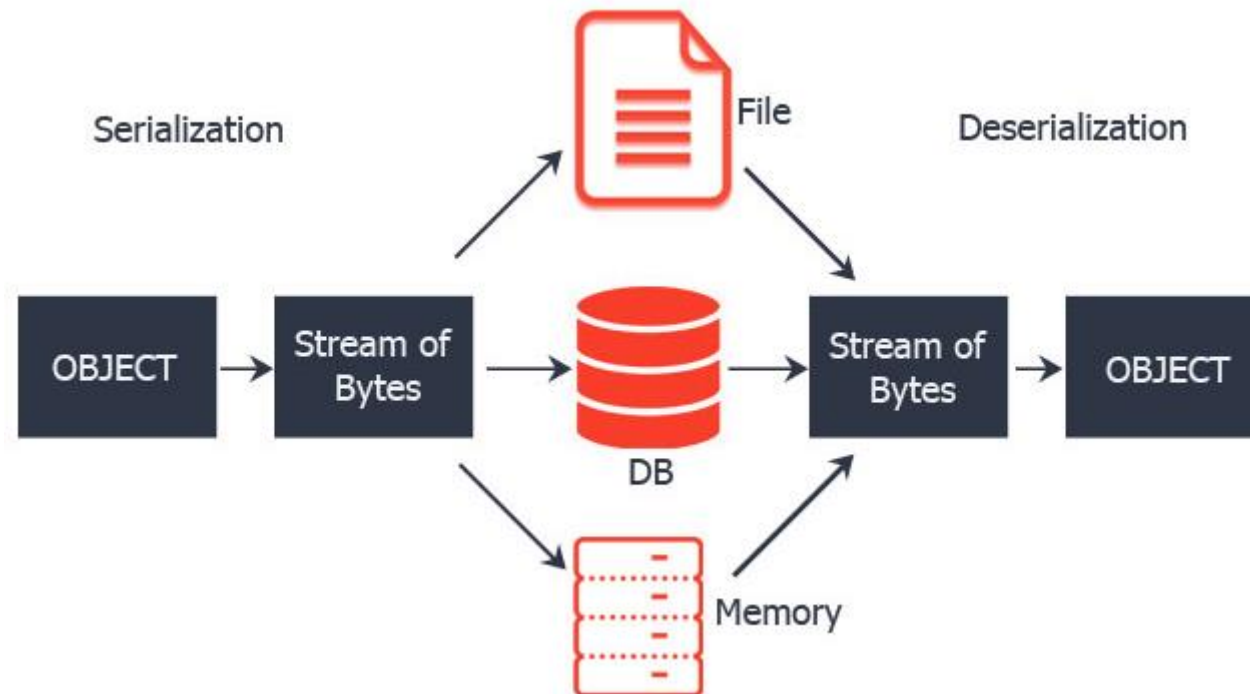
# SPRING DATA JPA

- Spring Data JPA est un sous projet du projet Spring Data.



# SÉRIALISATION / DÉSÉRIALISATION

- La **sérialisation** est le processus de conversion d'un **objet** en un **flux d'octets** pour stocker l'objet ou le transmettre à la mémoire, à une base de données, ou dans un fichier.
- Son principal objectif est d'enregistrer l'état d'un objet afin de pouvoir le recréer si nécessaire, par mécanisme de **désérialisation**.





# ENTITÉ JPA

- Une entité est une classe dont les instances peuvent être persistantes.
- Elle est déclarée avec l'annotation **@Entity**.
- Elle possède au moins une propriété déclarée comme identité de l'entité avec l'annotation **@Id**.
- Elle implémente l'interface **java.io.Serializable**.
- Utilisation d'annotations
  - Sur la classe : correspondance avec la table associée.
  - Sur les attributs ou sur les propriétés : correspondance avec les colonnes de la table.
- La classe est un JavaBean (attributs, accesseurs, mutateurs)
- Exemple :

# ENTITE JPA

```
package tn.esprit.entity;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;
.....
@Entity
@Table( name = "Etudiant")
public class Etudiant implements Serializable {
    @Id
    @GeneratedValue (strategy = GenerationType.IDENTITY)
    @Column(name="idEtudiant")
    private Integer idEtudiant; // Clé primaire
    private String prenomE;
    private String nomE;
    @Enumerated(EnumType.STRING)
    private Option op;
    // Constructeur et accesseurs (getters) et mutateurs (setters)
}
```

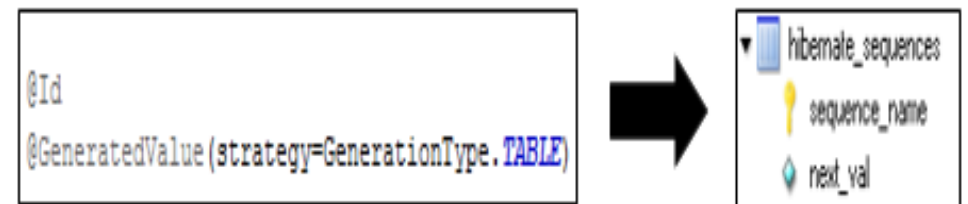
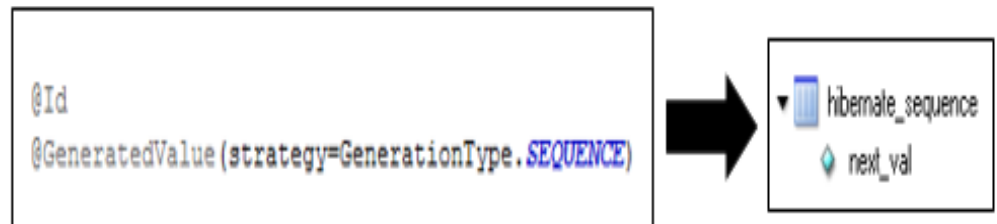
# ANNOTATIONS

- **@Entity** : Obligatoire, sur la classe.
- Une entité, déclarée par l'annotation **@Entity** définit une classe Java comme étant persistante et donc associée à une table dans la base de données.
- 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**.
- **@Table (name="nomTable")** : Facultatif, sur la classe. Cette annotation permet de mapper les objets de la classe avec la table dont le nom est redéfini. Si omis, la table prend le nom de la classe.

# ANNOTATIONS

- **@Id** : La déclaration d'une clé primaire est obligatoire. Sur un attribut ou sur le getter
- **@GeneratedValue** : Facultatif, sur l'attribut ou sur le getter annoté avec @Id. Définit la manière dont la base gère la génération de la clé primaire. L'attribut "**strategy**" obligatoire pouvant avoir comme valeur : AUTO / IDENTITY / SEQUENCE / TABLE :

```
@Id
@GeneratedValue (strategy = GenerationType.IDENTITY)
@Column(name="idEtudiant")
private Long idEtudiant; // Clé primaire
```



# ANNOTATIONS

- **Strategy = GenerationType.AUTO** : la génération de la clé primaire est garantie par le fournisseur de persistance (hibernate), une séquence unique de clé primaire **pour tout le schéma de base de données** défini dans une table **hibernate\_sequence**.
- **Strategy = GenerationType.TABLE** : la génération des clés primaires est garantie par le fournisseur de persistance (hibernate), une séquence de clés primaires **par table** définie dans une table **hibernate\_sequences**, cette table contient deux colonnes : une pour le nom de la séquence et l'autre pour la prochaine valeur.
- **Strategy = GenerationType.IDENTITY**. Hibernate s'appuie alors sur le mécanisme propre au SGBD pour la production de l'identifiant.  
Dans le cas de MySQL, c'est l'option AUTO-INCREMENT, dans le cas de Postgres ou Oracle, c'est une séquence. **On recommande l'IDENTITY.**
- **Strategy = GenerationType.SEQUENCE** = GenerationType.AUTO, mais c'est nous qui définissons les détails de la séquence.

# Exercice

1-

```
@Id
@GeneratedValue (strategy = GenerationType.IDENTITY)
@Column(name="idEtudiant")
private Long idEtudiant; // Clé primaire
```

```
@Id
@Column(name="idEtudiant")
private Long idEtudiant; // Clé primaire
```

- Comment le champ idEtudiant de la table Etudiant sera alimenté dans chacun des deux cas?

2- Exécuter la requête SQL : **show table status;**

# ANNOTATIONS

- **@Column** est une annotation utile pour indiquer le nom de la colonne dans la table, quand cette dernière est différente du nom de la propriété en java. Les principaux attributs pour **@Column**.
  - **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.
- **@Transient**
  - Facultatif, sur un attribut
  - Indique que l'attribut ne sera pas mappé (et donc non persisté) dans la table

# ANNOTATIONS

- **@Temporal** : gère l'attribut en tant que date.
- Pour déclarer une date :

```
// TIME : 30-09-19 10:50:56.780000000 AM
// DATE : 30-09-19
// TIMESTAMP : 1569840656 (nbre de secondes entre 01/01/1970 et la date voulue)
@Temporal (TemporalType.DATE)
private Date dateDebut;
```
- Pour insérer une date :

```
SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
Date dateDebut = dateFormat.parse("30/09/2019");
```



# ANNOTATIONS

- **@Enumerated :**

```
public enum Domaine {  
  
    mathematiques, informatique, biologie  
  
}
```

```
@Enumerated(EnumType.STRING)  
public Domaine getDomaine() {  
    return domaine;  
}
```

```
@Enumerated(EnumType.ORDINAL)  
public Domaine getDomaine() {  
    return domaine;  
}
```

Column Name	Datatype
• salaire	• FLOAT
• specialite	• VARCHAR(255)
• domaine	• VARCHAR(255)
• grade	• VARCHAR(255)

Column Name	Datatype	NOT NULL
• salaire	• FLOAT	✓
• specialite	• VARCHAR(255)	
• domaine	• INTEGER	
• grade	• VARCHAR(255)	

# Héritage

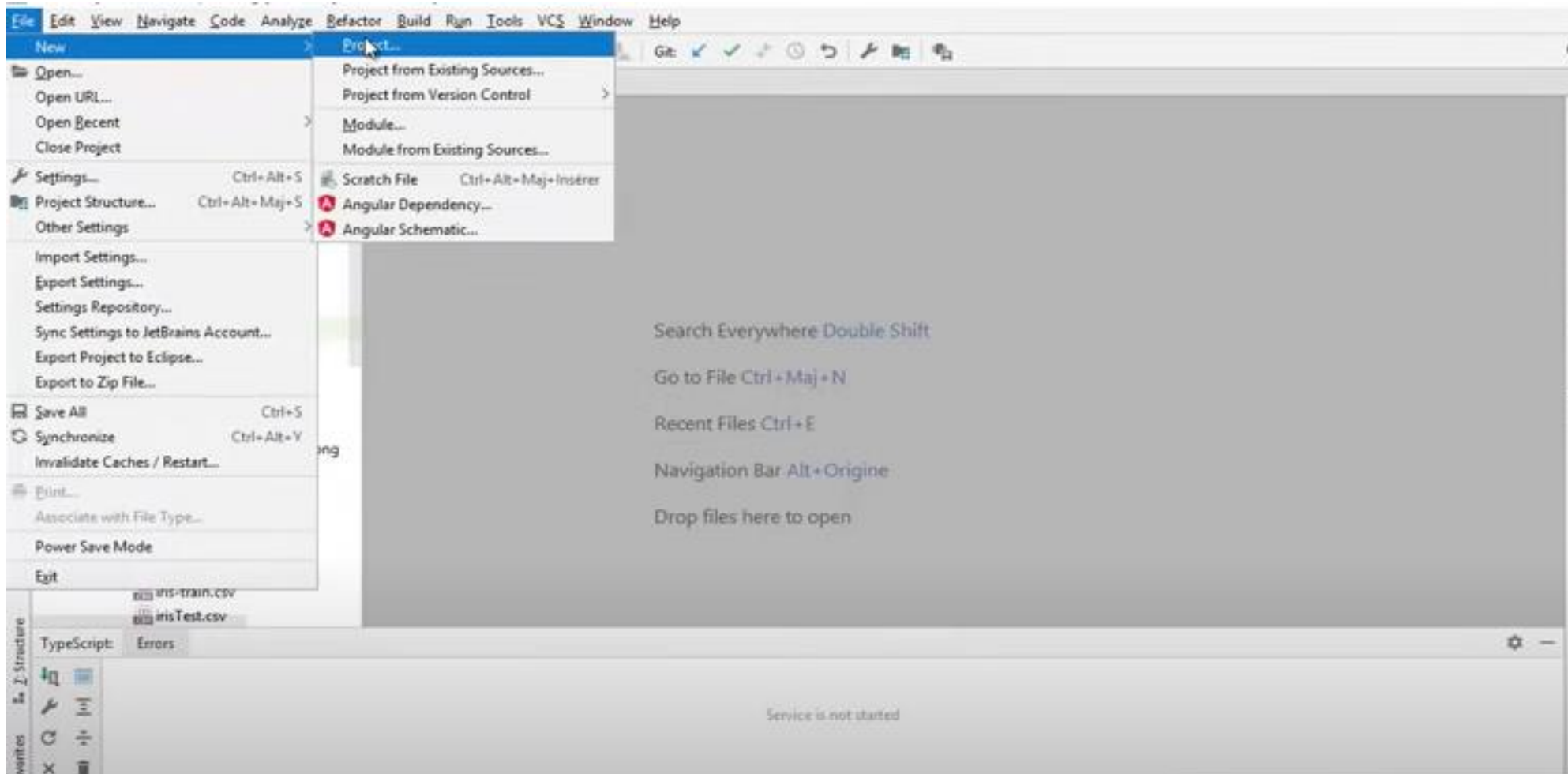
- Pour représenter un modèle hiérarchique dans un modèle relationnel, JPA propose alors trois stratégies possibles :
- 1. **Une seule table unique** pour l'ensemble de la hiérarchie des classes. L'ensemble des attributs de toute la hiérarchie des entités est mis à plat et regroupé dans une seule table (il s'agit d'ailleurs de la stratégie par défaut).
- 2. **Une table pour chaque classe concrète**. Chaque entité concrète de la hiérarchie est associée à une table.
- 3. **Jointure entre sous-classes**. Dans cette approche, chaque entité de la hiérarchie, concrète ou abstraite, est associée à sa propre table. Ainsi, nous obtenons dans ce cas là une séparation des attributs spécifiques de la classe fille par rapport à ceux de la classe parente. Il existe alors, une table pour chaque classe fille, plus une table pour la classe parente. Une jonction est alors nécessaire pour instancier la classe fille.

# TP – Spring Data JPA – 1<sup>ère</sup> Entité

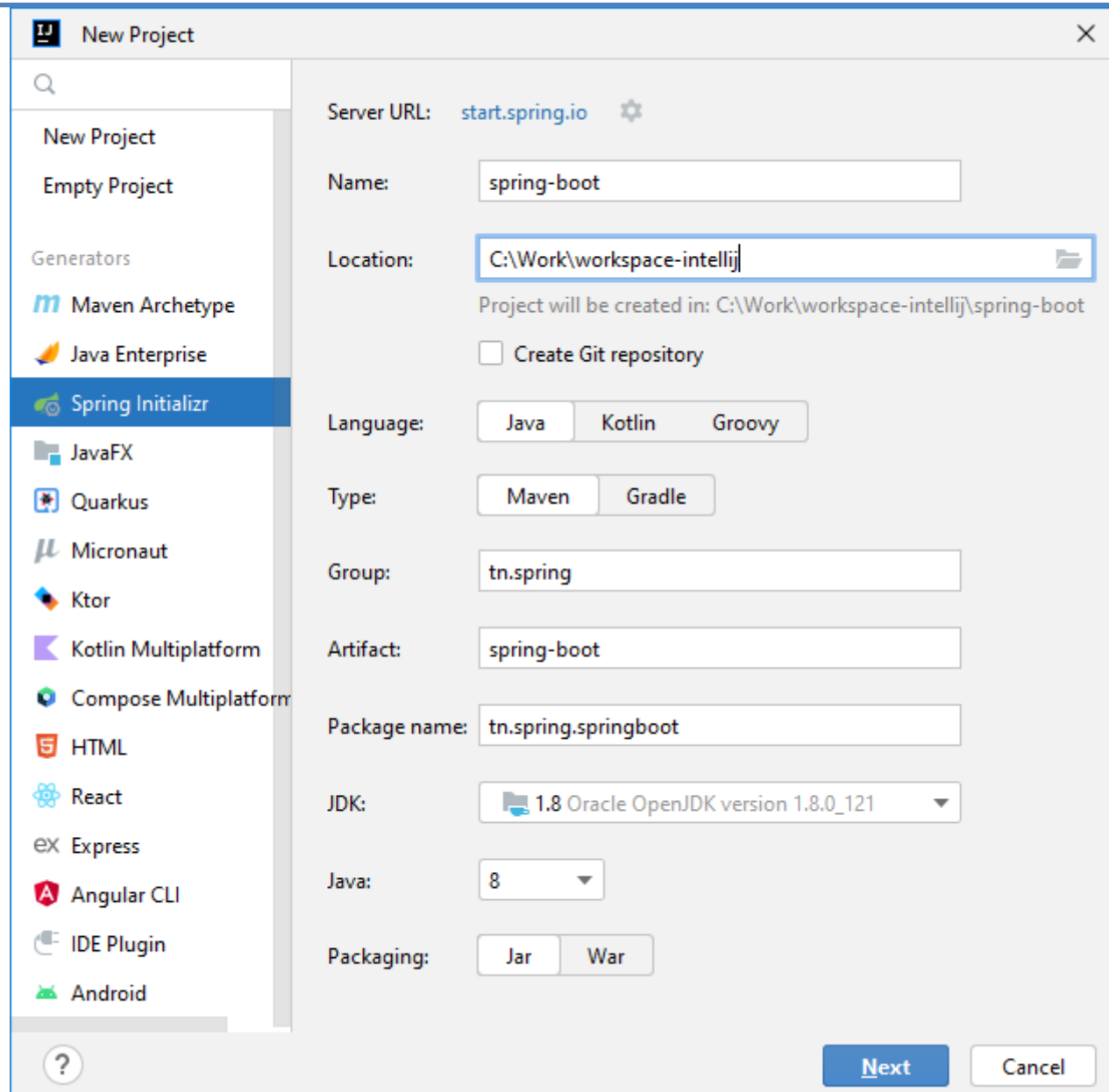
- Nous allons dans ce TP manipuler des données en base de données en utilisant Spring Data JPA.
- Pour faciliter l'implémentation de ce TP, nous allons utiliser également Spring Boot.
- Les étapes seront décrites dans les slides suivants :

# TP – Spring Data JPA – 1<sup>ère</sup> Entité

- Création d'un projet spring Boot



# TP – Spring Data JPA – 1<sup>ère</sup> Entité

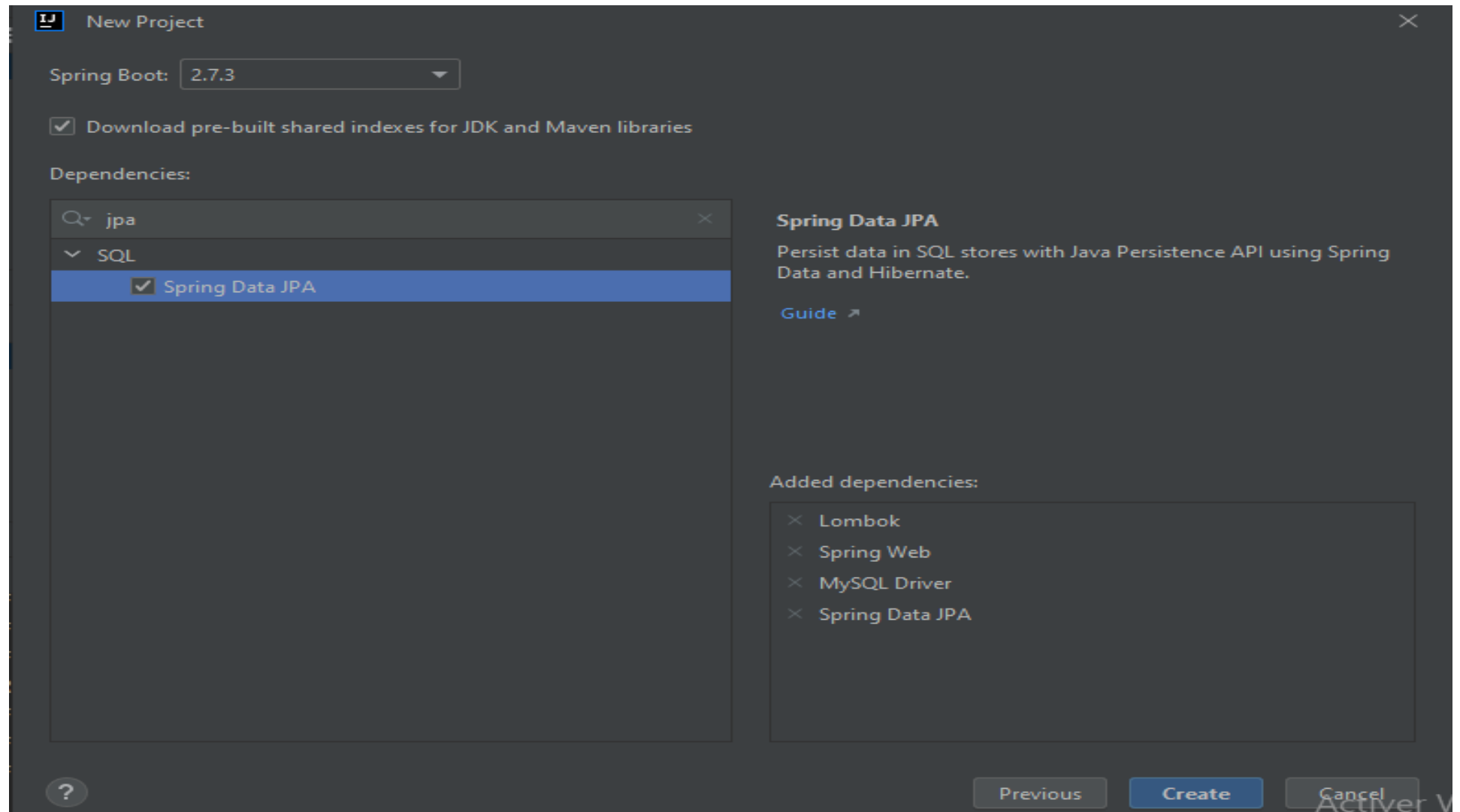


The image shows the 'New Project' dialog in IntelliJ IDEA. The 'Spring Initializr' option is selected in the left sidebar. The main configuration area on the right contains the following fields and options:

- Server URL:** `start.spring.io` (with a gear icon for settings)
- Name:** `spring-boot`
- Location:** `C:\Work\workspace-intellij` (with a folder icon for selection)
- Project will be created in:** `C:\Work\workspace-intellij\spring-boot`
- ☐ **Create Git repository**
- Language:** `Java` (selected), `Kotlin`, `Groovy`
- Type:** `Maven` (selected), `Gradle`
- Group:** `tn.spring`
- Artifact:** `spring-boot`
- Package name:** `tn.spring.springboot`
- JDK:** `1.8 Oracle OpenJDK version 1.8.0_121` (dropdown menu)
- Java:** `8` (dropdown menu)
- Packaging:** `Jar` (selected), `War`

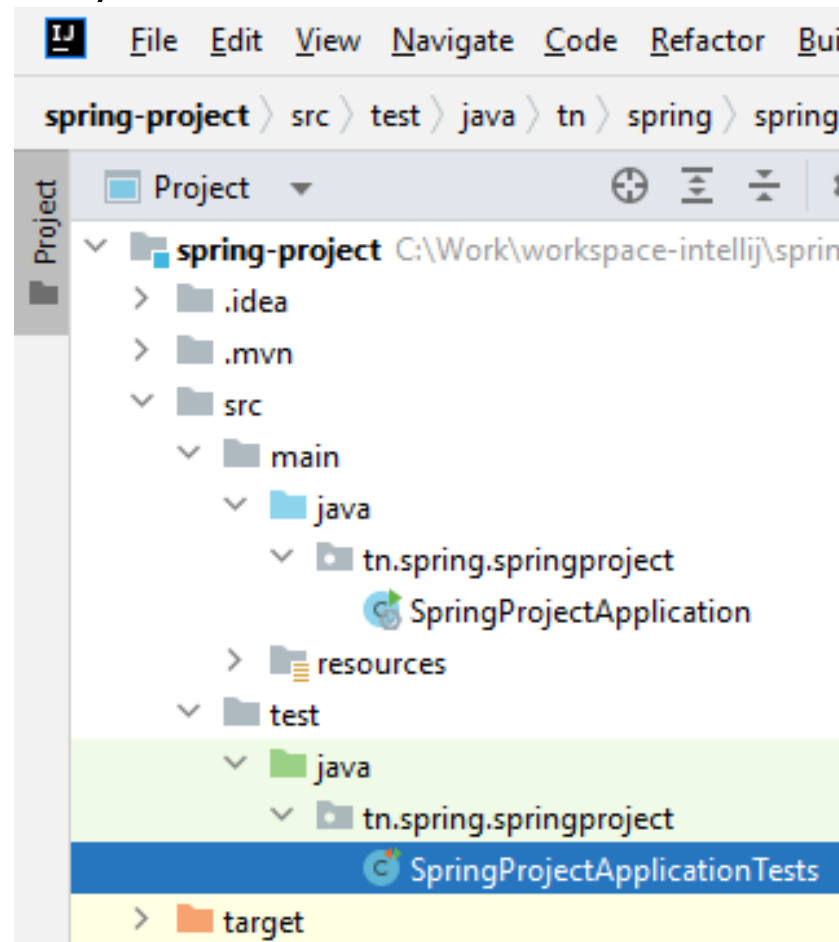
At the bottom right, there are **Next** and **Cancel** buttons. A help icon (?) is located at the bottom left of the dialog.

# TP – Spring Data JPA – 1<sup>ère</sup> Entité



# TP – Spring Data JPA – 1<sup>ère</sup> Entité

- Supprimer **la classe de test** pour éviter les erreurs lors de l'appel des commandes Maven (car « Maven install » par exemple essaiera de lancer les tests unitaires) :



# TP – Spring Data JPA – 1<sup>ère</sup> Entité

- Pour pouvoir se connecter à votre base de données, quel fichier faut-il mettre à jour?
- (Regarder l'arborescence du projet créé).



# TP – Spring Data JPA – 1<sup>ère</sup> Entité

- Dans ce fichier de properties ajouter les lignes suivantes :

### DATABASE ###

```
spring.datasource.url=jdbc:mysql://localhost:3306/springdb?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC
```

```
spring.datasource.username=root
```

```
spring.datasource.password=
```

### JPA / HIBERNATE ###

```
spring.jpa.show-sql=true
```

```
spring.jpa.hibernate.ddl-auto=update
```

```
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
```

**NB : Créer manuellement dans MySQL la base de donnée springdb ( phpMyAdmin => Nouvelle base de données).**

# Erreur (si ancien MySQL)

- Vérifier que la table a bien été créée dans la base de données :
- Si Erreur lors de l'exécution :  
`java.sql.SQLException: The server time zone value 'Paris, Madrid' is unrecognized or represents more than one time zone. You must configure either the server or JDBC driver (via the serverTimezone configuration property) to use a more specific time zone value if you want to utilize time zone support.`
- Solution : dans `application.properties` :  
`spring.datasource.url=jdbc:mysql://localhost:3306/springdb?useUnicode=true&useJDBCCompliantTimezoneShift=true&useLegacyDatetimeCode=false&serverTimezone=UTC`

# Exercice

Tp étude de cas Kaddem

## **Partie 1 Spring Data JPA – Première entité**

Créer les entités se trouvant dans le diagramme des classes (sans les associations) et vérifier qu'ils ont été ajoutés avec succès dans la base de données.

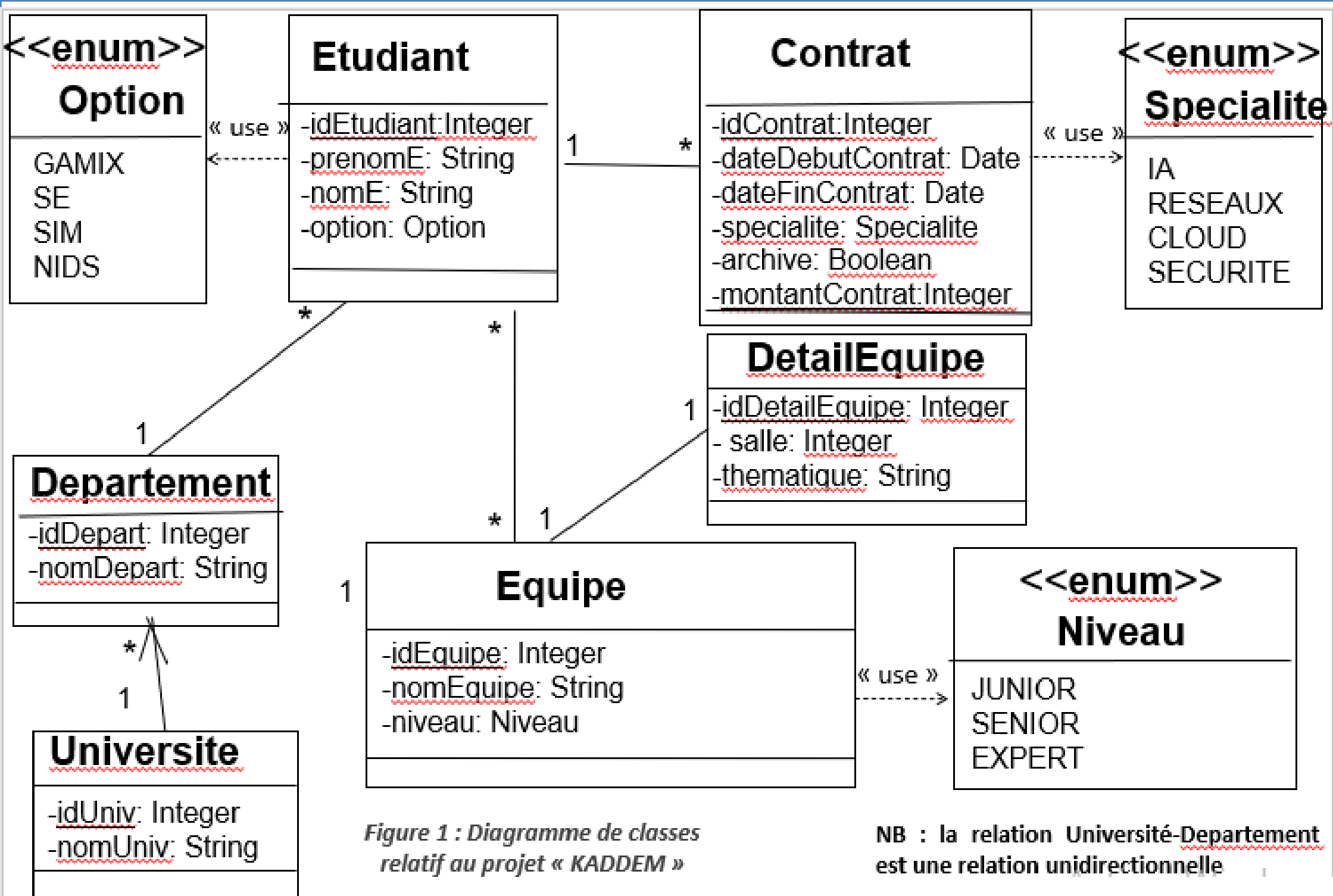


Figure 1 : Diagramme de classes  
relatif au projet « KADDEM »

NB : la relation Université-Departement  
est une relation unidirectionnelle

# SPRING DATA JPA – Première Entité

Si vous avez des questions, n'hésitez pas à nous contacter :

**Département Informatique**  
**UP Architectures des Systèmes d'Information**  
Bureau E204