TP Jakarta Persistence

Jusqu'à présent nous avons vu comment créer une base de données et comment récupérer les valeurs avec la norme JDBC et un connecteur MySQL qui implémente cette norme.

Aujourd'hui nous allons voir qu'il existe une autre norme pour les entreprises et qui s'appelle JPA (pour Java Persistence API) et qui a été renommée en **Jakarta Persistence** depuis peu. Elle est mieux adaptée pour les entreprises qui ont des besoins de sécurité et de robustesse différents des applications standards.

Jakarta Persistence est une spécification qui fait partie de **Jakarta EE** (anciennement connu sous le nom **Java EE** ou encore **J2E** pour **Java2 Entreprise Edition**).

Les spécifications **Jakarta EE** comprennent les spécifications suivantes :

- **Jakarta RESTful Web Services** : support des transferts HTTP/JSON avec le pattern architectural RESTful. Cette spécification permet la connexion rapide du front-end avec le back-end.
- **Jakarta Context and Dependency Injection** (CDI) : support pour l'injection de dépendances dans les composants. Cette spécification permet de gérer le cycle de vie de vos composants et la manière dont ils se lient entre eux. Composant est un synonyme de "singleton" (une seule instance).
- **Jakarta Concurrency** : permet la gestion de la sécurité d'une requête à une autre, entre les threads de votre application.
- **Jakarta Persistence** : support de la persistance de vos objets en base de données. C'est cette dernière spécification sur laquelle nous allons travailler aujourd'hui.

Pour activer le support de Jakarta EE dans IntelliJ, nous devons utiliser la facette "Jakarta EE".

> *Une facette est une qualité qu'on donne à un projet et qui permet d'activer un ensemble de fonctionnalités dans l'IDE pour augmenter notre productivité en réalisant à notre place toutes les étapes pénibles.*

Vous pouvez créer votre projet directement avec la facette Jakarta EE ou bien la lui donner plus tard, via le menu "File > Project Structure" et l'ajout du Module "JPA" et de la Facette "JPA".

Note : Jakarta Persistence n'est qu'une spécification et nous devrons l'utiliser avec une implémentation. Les plus fréquemment utilisées sont **Hibernate** (JBoss) et **EclipseLink** (Fondation Eclipse). Nous utiliserons Hibernate pour la suite du TP.

> *Dans tous les cas, ces deux implémentations respectent le standard Jakarta Persistence (que j'appellerai JPA dans la suite du document pour abréger). En théorie, vous pourrez donc utiliser aussi bien une implémentation EclipseLink qu'une implémentation Hibernate une fois que votre mapping Objet-Relationnel sera configuré.*

Contexte

Voici le contexte du TP : une association sportive vous a contacté car elle souhaite mettre en place un logiciel pour **suivre les cotisations de ses licenciés**. Une base de données vous a été fournie sous forme d'un script SQL permettant de créer les tables.

Votre travail durant ce TP sera le suivant :

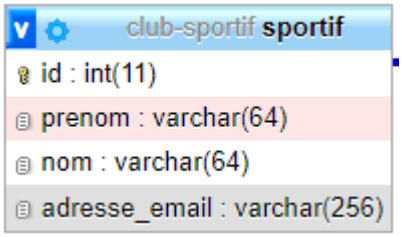
1. importer la base de données fournie
2. mettre en place le mapping objet-relationnel avec IntelliJ
3. afficher la liste des adhérents qui ne sont pas à jour de leur cotisation
4. insérer des données dedans en passant par Jakarta Persistence

Qu'est-ce qu'un mapping Objet-Relationnel ?

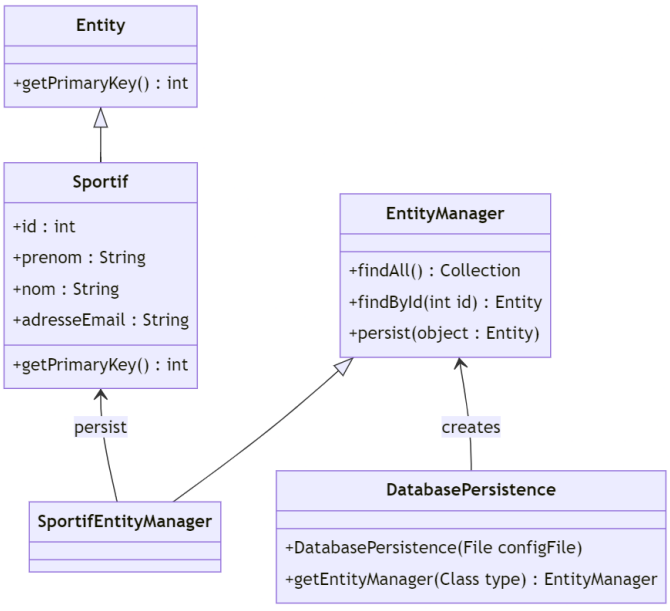
Un mapping Objet-Relationnel (ORM) permet de manipuler les données des tables d'une base de données relationnelle comme s'il s'agissait de classes Java.

Jusqu'à présent nous le faisions manuellement avec JDBC (`Connection, ResultSet, Statement`) et des POJOs (*Plain Old Java Objects*) contenant des variables et des *getters/setters*.

Prenons l'exemple de la table "sportif" de la base de données utilisée sur ce TP :



Pour stocker des données dans la BDD dans notre application, nous aurons besoin des classes suivantes :



Pour nous éviter d'écrire manuellement toutes ces classes (c'est à dire éviter de faire des erreurs ET bénéficier de toutes les bonnes pratiques), nous utilisons JPA et IntelliJ pour les générer à notre place :)

IntelliJ va créer d'une part un fichier de configuration "`META-INF/persistence.xml`" et d'autre part générer nos *DOs* (pour "*Data Objects*", c'est-à-dire les classes qui correspondent aux tables).

De l'autre côté, les classes de support (`Persistence`, `EntityManagerFactory`, `EntityManager` ...etc) sont apportées par JPA et Hibernate. Il nous restera à écrire seulement les méthodes dites "métier" comme par exemple une méthode "`getAdherentsAvecCotisationIncomplete()`" qui fera une requête JPQL (équivalent à SQL) et retournera les adhérents qui n'ont pas encore payé une cotisation complète pour chaque licence.

Ce qu'on appelle "métier" c'est tout ce qui n'est pas du code technique et qui n'est pas transposable d'une application à l'autre.

De quoi se compose JPA ?

Le cœur de JPA réside dans 2 choses :

1. Le fichier de configuration **persistence.xml**
2. Les annotations JPA (nous avons déjà vu ce qu'est une annotation avec JUnit : ce sont des tags qu'on peut utiliser pour "marquer" notre code à la manière de hashtags, comme par exemple @Test ou @BeforeAll).

Le fichier "persistence.xml"

Voici un exemple de fichier **persistence.xml** valable pour Jakarta Persistence 3.x :

```
<persistence>
    <persistence-unit name="club-sportif">
        <properties>
            <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver" />
            <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/club-sportif" />
            <property name="jakarta.persistence.jdbc.user" value="myuser" />
            <property name="jakarta.persistence.jdbc.password" value="mypasswd" />
        </properties>
        <class>fr.simplon.clubsportif.Sportif</class>
    </persistence-unit>
</persistence>
```

Vous constatez qu'on retrouve les éléments de configuration JDBC que nous mettions jusqu'à présent dans une classe **DataAccess** (ou **DatabaseConnection** ou bien encore **Database** peu importe comment elle s'appelait).

On y trouve également le nom des *Data Objects* qui vont contenir les annotations JPA (`fr.simplon.clubsportif.Sportif`)

Pour plus d'informations sur ce fichier de configuration faites des recherches sur le net. Le lien suivant est un bon début : https://thorben-janssen.com/jpa-persistence-xml/

Les annotations JPA

Les annotations JPA sont placées au-dessus d'une classe ou d'un attribut pour spécifier à quelle entité de la base de données cette classe ou cet attribut est lié.

Voici un exemple de *POJO* annoté avec les annotations JPA (ce qui en fait donc un *Data Object*) :

```
@Entity
@Table(name = "sportif")
public class Sportif implements Serializable { // Serializable obligatoire !!!

   @Id
   @Column(name = "id")
   private Long id;

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

   // ...
}
```

`@Entity`	Permet de désigner votre POJO comme une classe persistante
`@Table(name = "sportif")`	Permet de désigner la table où seront persistés les objets de votre classe
`@Id`	Désigne l'attribut associé à la clé primaire de la table
`@Column(name = "nom")`	Spécifie le nom de la colonne de la table dans laquelle est stocké l'attribut

Exemple de classe de lecture de données annotée avec les annotations JPA (hors périmètre pour ce TP mais ce sera à connaître pour le module Back-end JEE) :

```
@ApplicationScoped
@Transactional
public class SportifEntityManager {

   @PersistenceContext(unitName="club-sportif")
   private EntityManager em;

   public Sportif findSportif(Long id) {
      return em.find(Sportif.class, id);
   }
}
```

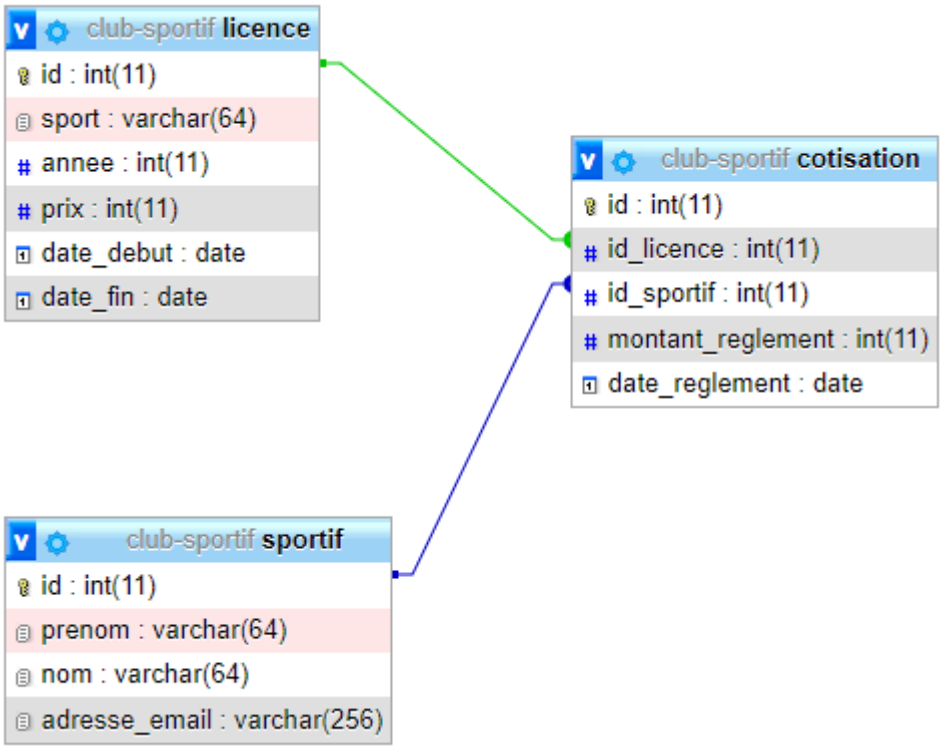
@ApplicationScoped	Annotation qui définit le périmètre du cycle de vie de ce bean. Pour plus d'informations sur les Entreprise Java Beans, lisez cette courte page : https://quarkus.io/guides/cdi#bean-scope-available
@Transactional	Permet d'activer les capacités transactionnelles. Quand une méthode démarre, une transaction sera démarrée automatiquement. À la fin de la méthode, la transaction sera validée si tout s'est bien passé ou bien sera

	rollback si une exception est survenue.
@PersistenceContext	Permet de désigner quelle "Persistence Unit" il faut utiliser. Pour rappel il s'agit du fichier persistence.xml. Si vous ne chargez qu'un seul context de persistance dans votre application, pas besoin de spécifier le nom de la persistence unit à utiliser. Sinon vous devrez préciser le nom avec

Il existe d'autres annotations pour gérer les relations entre vos classes de données mais nous les aborderons plus loin. Je vous recommande un bon article sur JPA : https://blog.payara.fish/getting-started-with-jakarta-ee-9-jakarta-persistence-api-jpa

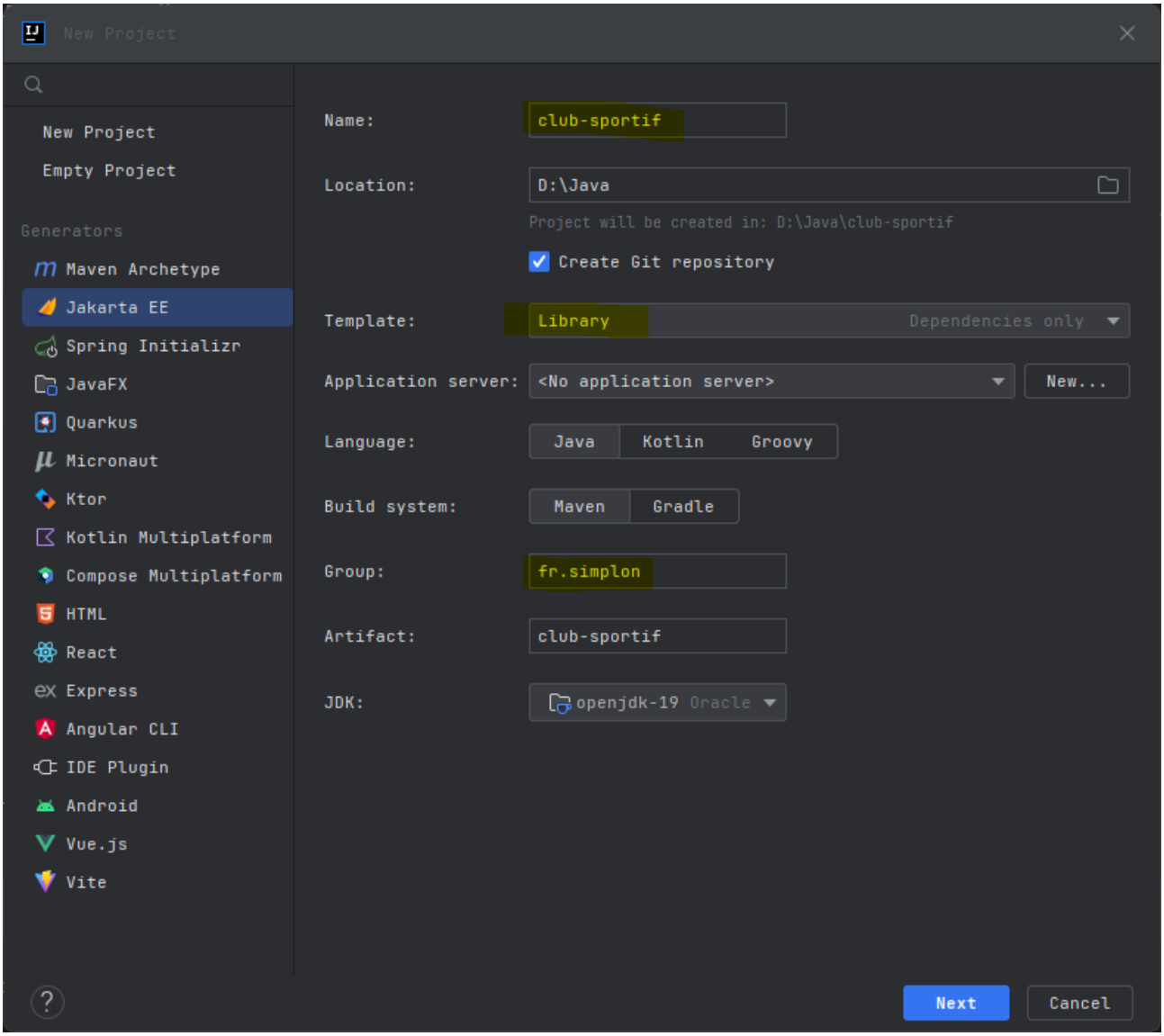
Import de la BDD

Le script SQL vous est fourni, vous avez normalement toutes les compétences pour être en autonomie sur ce sujet. À l'issue de votre import, la base ressemblera à ceci :

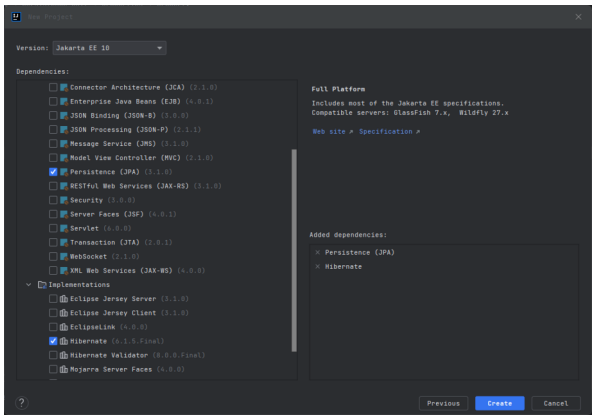


Création d'un projet Jakarta EE

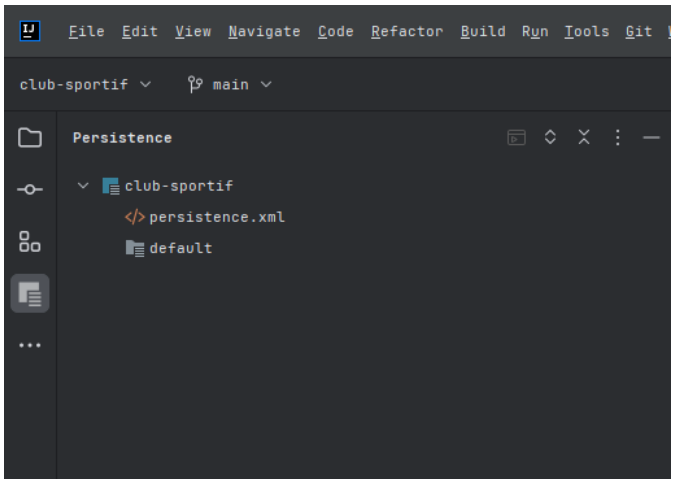
Créez un nouveau projet via le menu `File > New project`… et remplissez les champs comme ceci (mettez votre projet dans C:/Users/…./IdeaProjects)



- Cliquez sur Next.



- Sélectionnez la version 9.1 et cochez ensuite les cases **Persistence** et **Hibernate** comme sur la capture d'écran suivante.
- Ouvrez ensuite l'outil Persistence depuis le menu View > Tool Windows > Persistence. Vous obtenez cette fenêtre :

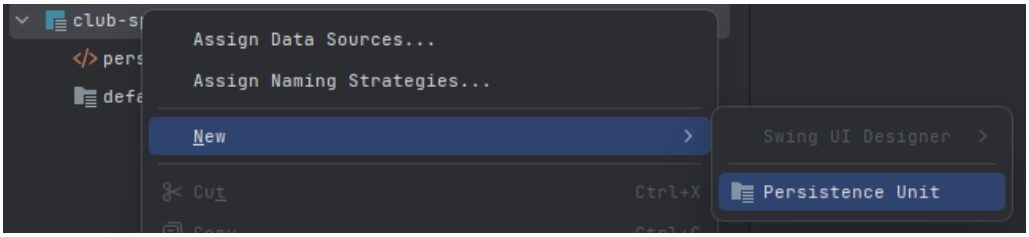


Création d'une *Persistence Unit*

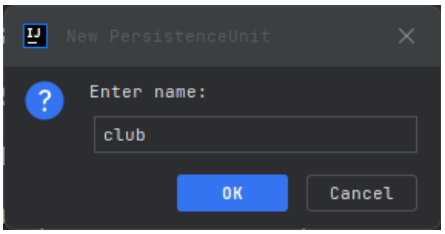
Nous allons maintenant créer une **unité de persistence** pour configurer Jakarta Persistence et Hibernate. Cette "*Persistence Unit*" contiendra toute la configuration nécessaire pour se connecter automatiquement à la base de données sans écrire une seule ligne de code.

D'autre part nous allons **générer automatiquement les classes Java de notre modèle de données**.

- Faites un clic droit sur l'élément "club-sportif" puis cliquez sur "New > Persistence Unit"



- Puis renseignez le nom de votre *PU* (note : c'est une bonne idée de l'appeler "club-sportif" et non pas juste "club" comme c'est le cas sur ma capture d'écran…) :



Génération des classes de mapping

Le reste est présenté sous forme de vidéos sur youtube :

https://www.youtube.com/playlist?list=PLRYQwkQcKLfBY414L88_aB2JwToLEhkrC

Travail à faire

1. Servez-vous du code que vous avez vu dans les vidéos pour écrire un programme qui affiche les sportifs qui sont en base de données et pour chacun d'entre eux qui affiche ses cotisations.

2. Il vous est également demandé d'afficher en fin de programme la liste de tous les sportifs qui ne sont pas à jour de leurs cotisation, avec la somme restante due par rapport au prix de la licence.

3. Une fois que vous aurez réussi, insérez les données suivantes en BDD (avec JPA) :

- 1 nouveau type de licence pour le sport que vous aimez

- 1 nouvel adhérent qui pratique ce sport et a payé 2 cotisations cette année (la somme des cotisations payées par ce nouvel adhérent ne doit pas atteindre le prix de la licence)

Utilisez la méthode `EntityManager.persist()`

4. Transformez vos requêtes JPQL en `@NamedQuery` (cf. internet). Interdiction de les mettre dans vos classes *DOs* puisque ce sont des classes générées par IntelliJ et que par conséquent elles peuvent être supprimées à tout moment.

→ https://gayerie.dev/epsi-b3-orm/javaee_orm/jpa_queries.html#utilisation-de-requetes-nommees