

Introduction à Jakarta Persistence API (JPA)

Emmanuel BRUNO

Université de Toulon, LIS UMR CNRS 7020

emmanuel.bruno@univ-tln.fr

2026-10-14

Table des matières

1 Magics	1
2 Exemples	2
3 Introduction à Jakarta Persistence API (JPA)	2
4 Spécification	2
5 En pratique	2
6 JPA possède trois composantes principales	3
7 Les Entités dans Jakarta Persistence API (JPA)	3
8 Le gestionnaire d'entités	4
8.1 Exemple de persistence.xml	4
8.2 Création de l'EM et persistence d'une entité	4
9 Conseils pour l'utilisation de JPA avec Lombok	7

1 Magics

registered line magics:

- listLineMagic
- load
- reload-class
- addMavenDependencies, maven, addMavenDependency
- javadoc-html
- listMagic, list
- cmd
- git-graph-mermaid
- classpath-snapshot

- pom, loadFromPOM

- classpath

- listCellMagic

- printWithName

- printerPrefix

- write

- read

- class-info

- jars

- classDiagram

- addMavenRepo, mavenRepo

- tableSchema

- where, which

- commonshellcmd

registered cell magics:

- rdbmsSchema

- mycompile

- plantUML

- javasrcConstructorByName

- javasrcClassByName

- sqlAsTable

- compile

- write

- benchmark

- timeit, timeIt, time

- javasrcMethodByName

- javasrcMethodByAnnotationName

- plantUMLFile

- javasrcInterfaceByName

- shell

- tableSchema

- javasrcFieldByName

- classDiagram
- javasrcList
- pom, loadFromPOM
- javasrcJavadoc
- commonshell

2 Exemples

Les exemples présentés sont disponibles dans l'entrepôt: <https://github.com/ebpro/sample-hellojpa/>

Un exemple complet de base est disponible dans l'entrepôt: <https://github.com/ebpro/minimal-jpa/>

3 Introduction à Jakarta Persistence API (JPA)

Le Mapping Relationnel/Object (ORM) est une approche déclarative visant à établir un lien entre un modèle Orienté Objet, tel qu'un programme Java, et un modèle relationnel, matérialisé par des tables dans une base de données relationnelle. Cette démarche peut comprendre la création du schéma relationnel à partir du programme Java et l'automatisation des fonctions essentielles pour la gestion de la persistance des données. Cela améliore la portabilité des applications par une abstraction de la couche de persistance et donc une réduction de la dépendance au code SQL spécifique au fournisseur de base de données.

4 Spécification

Pour Java, Jakarta Persistence API (JPA) est une spécification (actuellement en version 3.1) concernant la liaison entre le modèle orienté objet de Java et le modèle relationnel (Object Relational Mapping - ORM). JPA s'appuie sur JDBC (Java Database Connectivity). JPA fournit une solution robuste et normalisée pour la

gestion des interactions entre une application Java et une base de données.

Plusieurs implantations de JPA existent dont l'implantation de référence EclipseLink et une autre très utilisée Hibernate. Cette introduction utilise Hibernate mais en respectant le standard.

5 En pratique

Il faut donc ajouter les éléments suivant au `pom.xml` de Maven.

```
<!-- L'API Standard -->
<!-- a priori inclue par
transitivité avec l'une des suivantes
-->
<dependency>
  <groupId>jakarta.persistence</
groupId>
  <artifactId>jakarta.persistence-
api</artifactId>
  <version>3.1.0</version>
</dependency>

<dependency>
  <groupId>org.hibernate.orm</
groupId>
  <artifactId>hibernate-core</
artifactId>
  <version>6.3.1.Final</version>
</dependency>

<!-- L'implantation de référence
-->
<!--
<dependency>

<groupId>org.eclipse.persistence</
groupId>
  <artifactId>eclipselink</
artifactId>
  <version>4.0.1</version>
```

```
</dependency>
-->
```

6 JPA possède trois composantes principales

- La description des « entités » persistentes.
- L'interaction de base avec la persistance via les opérations CRUD (Create, Read, Update, Delete).
- L'interrogation native ou via un langage dédié.

Pour utiliser JPA, il faut une base de données relationnelle. Nous utiliserons ici PostgreSQL via l'image Docker officielle.

Les exemples sont disponibles sur ce repository GitHub.

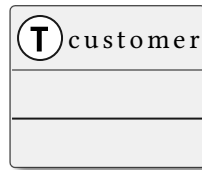
7 Les Entités dans Jakarta Persistence API (JPA)

Avec JPA une classe dont les instances doivent être persistantes est appelée une entité. Techniquement, une classe est une entité si elle annotée avec `@Entity`, qu'un ou plusieurs de ses attributs sont annotés avec `@Id` et qu'elle a au moins un constructeur sans paramètre (au plus `protected`). La classe sera alors associée à une relation du même nom, ayant une clé primaire (atomique ou composite)

La classe suivante est donc entité JPA :

```
Error: failed to read file /
home/jovyan/work/examples/github/ebpro/
sample-hellojpa/src/main/java/fr/
univtl/bruno/demos/jpa/hello/samples/
ex_simple/Customer.java: /home/jovyan/
work/examples/github/ebpro/sample-helloj-
pa/src/main/java/fr/univtl/bruno/demos/jpa/
hello/samples/ex_simple/Customer.java
```

associée à la relation :

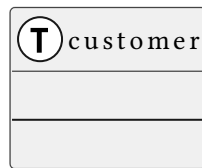


et à la table :

```
Table not found or has no columns:
ex_simple.customer
```

Le mapping peut être contrôlé finement par un ensemble d'annotations. La classe suivante illustre celles de base.

```
Error: failed to read file /
home/jovyan/work/examples/github/ebpro/
sample-hellojpa/src/main/java/fr/
univtl/bruno/demos/jpa/hello/samples/
ex_entity/Customer.java: /home/jovyan/
work/examples/github/ebpro/sample-helloj-
pa/src/main/java/fr/univtl/bruno/demos/jpa/
hello/samples/ex_entity/Customer.java
```



```
Table not found or has no columns:
customer
```

`@Entity` indique qu'une classe est une entité et permet de contrôler son nom (par défaut celui de la classe).

`@Id` est obligatoire et définit le ou les attributs qui composent la clé primaire.

`@GeneratedValue` contrôle la méthode utilisée pour générer la clé lors de l'insertion (automatique pour s'adapter à la base de données utilisée, identité pour MySQL, séquence pour

ceux qui le supporte ou Table qui est portable mais avec les moins bonnes performances et récemment UUID).

@Table contrôle finement le mapping au niveau de la table associée (nom spécifique, schema, catalogue, contraintes et index).

@Column contrôle finement la colonne associée au membre comme son nom ou ses contraintes (null, insertion ou mise à jour autorisés ou non ; unicité ; ...).

@Basic est l'annotation pour tout ce qui peut être représenté directement comme un attribut : les primitifs (et leurs classes d'enveloppement), les chaînes de caractères, les types temporels, les octets ou tableaux d'octets, les énumérés et tout ce qui implante l'interface Serializable. Cette annotation est optionnelle et permet de d'indiquer si l'attribut peut être nul dans la base (`optional`) et s'il doit être récupéré au changement de classe (par défaut, `fetch=EAGER`) ou uniquement lors de l'accès au membre (`fetch=LAZY`).

@Transient permet d'indiquer qu'un membre ne doit pas être persistant (s'il porte le modificateur `transient` c'est aussi le cas).

@Lob permet d'associer des grands objets binaires ou textes (BLOB et CLOB).

@Version définit un attribut utiliser pour assurer le verrouillage optimiste.

8 Le gestionnaire d'entités

Pour gérer les entité JPA s'appuie sur l'entity manager (EM). Il fournit les opération CRUD de base et une interface pour exécuter des requêtes. L'EM s'appuie sur des unités de persistances (des connexions à une base de données) qui sont définie dans un fichier XML `META-INF/persistence.xml` à la racine du classpath (donc dans `src/main/resources` avec Maven).

Son schéma est défini dans le standard <https://jakarta.ee/xml/ns/persistence>.

8.1 Exemple de persistence.xml

Un exemple de base est donné ci-dessous :

8.2 Création de l'EM et persistance d'une entité

Pour obtenir un EM, on utilise l'EntityManagerFactory (EMF) en indiquant le nom d'une persistance unit. La génération automatique éventuelle du schéma a lieu lors de la première utilisation de l'EMF.

A partir de l'entity manager, il est alors possible de commencer une transaction (`begin`), de rendre persistante une instance (`persist`) et de terminer la transaction (`commit` ou `rollback`). On note la mise à jour automatique de l'entity avec l'ID généré par la base de données. L'entity manager est prévu pour une utilisation locale et généralement brève, il doit être fermé après utilisation (`autocloseable`).

```
try (EntityManagerFactory emf = Persistence.createEntityManag
    pu"))
{
    try (EntityManager entityManager
        = emf.createEntityManager()) {
    }
}
```

La création d'un EMF est couteuse, on utilise donc généralement un singleton pour gérer l'EMF.

```
EntityManagerFactory emf = DatabaseManager.getEntityManag
Logger log =
LoggerFactory.getLogger("notebook");
```

```

06:45:20.927 [IJava-executor-0] INFO
f.u.b.d.jpa.hello.DatabaseManager
-- Database settings:
URL=jdbc:postgresql://dind/notebook-
db, User=dba
06:45:20.938 [IJava-executor-0] INFO
f.u.b.d.jpa.hello.DatabaseManager
-- Starting JPA with URL:
jdbc:postgresql://dind/notebook-db
and User: dba
06:45:22.194 [IJava-executor-0] INFO
com.zaxxer.hikari.HikariDataSource
-- HikariPool-1 - Starting...
06:45:22.235 [IJava-executor-0] INFO
com.zaxxer.hikari.pool.HikariPool --
HikariPool-1 - Added connection org.postgresql.jdbc.PgConnection@6091c637
06:45:22.237 [IJava-executor-0] INFO
com.zaxxer.hikari.HikariDataSource
-- HikariPool-1 - Start completed.

```

ce singleton retourne l'instance de l'EMF qui n'est qu'à la première demande.

```

import fr.univtln.bruno.demos.jpa.hello.s

try (EntityManager entityManager =
emf.createEntityManager();) {
    entityManager.getTransaction().begin()

    Customer customer =
Customer.of("pierre.durand@ici.fr");
    log.info("BEFORE PERSIST:
"+customer.toString());

entityManager.persist(customer);
    log.info("AFTER PERSIST:
"+customer.toString());

    entityManager.getTransaction().commi
}

```

```

06:45:24.515 [IJava-executor-0]
INFO notebook --
BEFORE PERSIST Customer(id=0,
email=pierre.durand@ici.fr,
name=null)
06:45:24.537 [IJava-executor-0]
INFO notebook --
AFTER PERSIST: Customer(id=1,
email=pierre.durand@ici.fr,
name=null)

```

A partir du moment où une entité est liée au contexte de persistance (en venant d'être créée ou récupérée), la mise à jour dans la base est implicite (pas de fonction dédiée de l'entity manager). La transaction n'est nécessaire que pour les modifications.

```

long newId;
Customer customer =
Customer.of("marie.dupond@la.fr");

try (EntityManager entityManager =
emf.createEntityManager();) {
    entityManager.getTransaction().begin();
    entityManager.persist(customer);

    customer.setName("Marie Dupond");

    entityManager.getTransaction().commit();
    log.info("AFTER COMMIT:
"+customer.toString());
}

newId=customer.getId();

```

```

06:45:24.682 [IJava-executor-0]
INFO notebook --
AFTER COMMIT: Customer(id=2,
email=marie.dupond@la.fr, name=Marie
Dupond)

```

L'entity manager propose une méthode de recherche par ID `find(Class<T> entityClass, Object primaryKey)` qui prend en paramètre la classe de l'entité recherchée et la valeur de la clé.

```
try (EntityManager entityManager =
emf.createEntityManager();) {
    Optional<Customer> customer = Optional.ofNullable(entityManager.find(Customer.class,
newId));

    if (customer.isEmpty())
        log.info("Id %d not in
database.".formatted(newId));
    else {
        log.info("Search cust. Id
%d: %s".formatted(newId, customer));
    }
}
```

```
06:45:24.846 [IJava-executor-0] INFO
notebook -- Search cust.
Id 2: Optional[Customer(id=2,
email=marie.dupond@la.fr, name=Marie
Dupond)]
```

L'entity manager propose une méthode `remove` pour supprimer une entité de la base de données.

```
try (EntityManager entityManager =
emf.createEntityManager();) {
    Optional<Customer> customer = Optional.ofNullable(entityManager.find(Customer.class,
newId));

    if (customer.isEmpty())
        log.error("Id %d not in
database.".formatted(newId));
    else {
        log.info("Removed Id %d from
the database.".formatted(newId));
        entityManager.getTransaction().begin();
        entityManager.remove(customer.get());
    }
}
```

```
entityManager.getTransaction().commit();
}
}
```

```
06:45:24.947 [IJava-executor-0] INFO
notebook -- Removed Id 2 from the
database.
```

`merge` attache un objet au contexte de persistance (par son id), `refresh` annule les modifications en cours de transaction sur une entité et `detach` indique à l'EM de ne plus gérer une entité.

```
Customer customer =
Customer.of("???");
customer.setId(newId);
log.info("BEFORE
MERGE "+customer.toString());

try (EntityManager entityManager =
emf.createEntityManager();) {
    entityManager.getTransaction().begin();
    customer =
entityManager.merge(customer);
    entityManager.refresh(customer);
    entityManager.getTransaction().commit();
}

log.info("AFTER MERGE+REFRESH
"+customer.toString());
```

```
06:45:25.302 [IJava-executor-0] INFO
notebook -- BEFORE MERGE
Customer(id=3, email=???, name=null)
06:45:25.365 [IJava-executor-0] INFO
notebook -- AFTER MERGE+REFRESH Customer(id=3,
email=pierre.durand@ici.fr,
name=null)
```

9 Conseils pour l'utilisation de JPA avec Lombok

Lorsque vous utilisez Lombok avec JPA (Java Persistence API), quelques précautions sont nécessaires pour assurer un fonctionnement harmonieux de votre application. Évitez l'utilisation de l'annotation `@Data`, car elle peut générer des méthodes indésirables dans le contexte de la persistance des données. De plus, faites attention avec `@EqualsAndHashCode`, car elle peut avoir des implications inattendues sur les relations Hibernate. Lorsque vous utilisez `@ToString`, assurez-vous d'exclure les attributs « Lazy » pour éviter les problèmes liés au chargement paresseux lors de l'affichage des objets. Enfin, pour garantir une construction cohérente des entités, ajoutez `@NoArgsConstructor(access=AccessLevel.PROTECTED)` en conjonction avec `@Builder` ou `@AllArgsConstructor`. En suivant ces bonnes pratiques, vous pouvez optimiser l'utilisation de Lombok avec JPA tout en évitant les écueils potentiels.