Accès aux bases de données : JDBC

JDBC (Java DataBase Connectivity) est l'API standard pour interagir avec les bases données relationnelles en Java. JDBC fait partie de l'édition standard et est donc disponible directement dans le JDK.

Préambule : try-with-resources

L'API JDBC donne accès à des objets qui correspondent à des ressources de base de données que le développeur doit impérativement fermer par l'appel à des méthodes *close()*. Ne pas fermer correctement les objets fournis par JDBC est un bug qui conduit habituellement à un épuisement des ressources système, empêchant l'application de fonctionner correctement.

Java 7 a introduit l'interface AutoCloseable ainsi qu'une nouvelle syntaxe dénommée try-with-resources. L'API JDBC utilise massivement l'interface AutoCloseable et autorise donc le try-with-resources. Ainsi, les deux codes ci-dessous sont équivalents puisque la classe java.sql.Connection implémente AutoCloseable :

Avec try-with-resources

```
try (java.sql.Connection connection = dataSource.getConnection()) {
  // ...
}
```

Sans try-with-resources

```
java.sql.Connection connection = dataSource.getConnection();
try {
  // ...
}
finally {
  if (connection != null) {
    connection.close();
  }
}
```

La version utilisant la syntaxe du try-with-resources est plus compacte et prend en charge automatiquement l'appel à la méthode _close(). Tout au long de ce chapitre sur JDBC, les exemples utiliseront alternativement l'une ou l'autre des syntaxes.

Le pilote de base de données

JDBC est une API indépendante de la base de données sous-jacente. D'un côté, les développeurs implémentent les interactions avec une base de données à partir de cette API. D'un autre côté, chaque fournisseur de SGBDR livre sa propre implémentation d'un pilote JDBC (JDBC driver). Pour pouvoir se connecter à une base de données, il faut simplement ajouter le driver (qui se présente sous la forme d'un fichier jar) dans le *classpath* lors de l'exécution du programme.

Des pilotes JDBC sont disponibles pour les SGBDR les plus utilisés : Oracle DB, MySQL, PostgreSQL, Apache Derby, SQLServer, SQLite, HSQLDB (HyperSQL DataBase)...

On peut rechercher le pilote souhaité sur le site du Maven Repository.

pour des raisons de licence, certains pilotes JDBC ne sont pas disponibles dans les référentiels Maven. C'est le cas notamment du pilote JDBC pour Oracle.

Création d'une connexion

Une connexion à une base de données est représentée par une instance de la classe Connection.

Comme nous l'avons précisé au début de ce chapitre, JDBC fait partie de l'API standard du JDK. Toute application Java peut donc facilement contenir du code qui permet de se connecter à une base de données. Pour cela, il faut utiliser la classe DriverManager pour enregistrer un pilote JDBC et créer une connexion :

Création d'une connexion MySQL avec le DriverManager

```
DriverManager.registerDriver(new com.mysql.jdbc.Driver());

// Connexion à la base myschema sur la machine localhost
// en utilisant le login "username" et le password "password"
Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost/myschema",
                                                    "username", "password");
```

Lorsque la connexion n'est plus nécessaire, il faut libérer les ressources allouées en la fermant avec la méthode *close()*. La classe Connection implémente AutoCloseable, ce qui l'autorise à être utilisée dans un try-with-resources.

```
connection.close();
```

L'URL de connexion et la classe des pilotes

Comme nous l'avons vu à la section précédente, pour établir une connexion, nous avons besoin de connaître la classe du pilote et l'URL de connexion à la base de données. Il n'existe pas vraiment de règle en la matière puisque chaque fournisseur de pilote décide du nom de la classe et du format de l'URL. Le tableau suivant donne les informations nécessaires suivant le SGBDR :

SGBDR	Nom de la classe du pilote	Format de l'URL de connexion
Oracle DB	oracle.jdbc.OracleDriver	[jdbc:oracle:thin:@[host]:[port]:schema]Ex : jdbc:oracle:thin:@localhost:1521:maBase
MySQL <= 5.1	com.mysql.jdbc.Driver	[jdbc:mysql://[host]:[port]/schema]Ex : jdbc:mysql://localhost:3306/maBase
MySQL >= 8	com.mysql.cj.jdbc.Driver	[jdbc:mysql://[host]:[port]/schema]Ex : jdbc:mysql://localhost:3306/maBase
MariaDB	org.mariadb.jdbc.Driver	[jdbc:mariadb://[host]:[port]/schema]Ex : jdbc:mariadb://localhost:3306/maBase
PosgreSQL	org.postgresql.Driver	[jdbc:postgresql://[host]:[port]/schema]Ex : jdbc:postgresql://localhost:5432/maBase

SGBDR	Nom de la classe du pilote	Format de l'URL de connexion
HSQLDB (mode fichier)	org.hsqldb.jdbcDriver	[jdbc:hsqldb:file:chemin du fichier]Ex : jdbc:hsqldb:file:maBase
HSQLDB (mode mémoire)	org.hsqldb.jdbcDriver	[jdbc:hsqldb:mem:schema]Ex : jdbc:hsqldb:mem:maBase

Note

Pour MySQL >= 8, il est parfois nécessaire de préciser le fuseau horaire à utiliser par le serveur en passant le paramètre `serverTimezone` dans l'URI de connexion :

```
jdbc:mysql://localhost:3306/maBase?serverTimezone=GMT
```

Les requêtes SQL (Statement)

L'interface Connection permet, entre autres, de créer des *Statements*. Un *Statement* est une interface qui permet d'effectuer des requêtes SQL. On distingue 3 types de Statement :

> - Statement : Permet d'exécuter une requête SQL et d'en connaître le résultat.
> - PreparedStatement : Comme le Statement, le PreparedStatement permet d'exécuter une requête SQL et d'en connaître le résultat. Le PreparedStatement est une requête paramétrable. Pour des raisons de performance, on peut préparer une requête et ensuite l'exécuter autant de fois que nécessaire en passant des paramètres différents. Le PreparedStatement est également pratique pour se prémunir efficacement des failles de sécurité par injection SQL.
> - CallableStatement : permet d'exécuter des procédures stockées sur le SGBDR. On peut ainsi passer des paramètres en entrée du CallableStatement et récupérer les paramètres de sortie après exécution.

Le Statement

Un Statement est créé à partir d'une des méthodes createStatement de l'interface Connection. À partir d'un Statement, il est possible d'exécuter des requêtes SQL :

```
java.sql.Statement stmt = connection.createStatement();

// méthode la plus générique d'un statement. Retourne true si la requête SQL
// exécutée est un select (c'est-à-dire si la requête produit un résultat)
stmt.execute("insert into myTable (col1, col2) values ('value1', 'value1')");

// méthode spécialisée pour l'exécution d'un select. Cette méthode retourne
// un ResultSet (voir plus loin)
stmt.executeQuery("select col1, col2 from myTable");

// méthode spécialisée pour toutes les requêtes qui ne sont pas de type select.
// Contrairement à ce que son nom indique, on peut l'utiliser pour des requêtes
```

```
// DDL (create table, drop table, ...) et pour toutes requêtes DML (insert,
update, delete).
stmt.executeUpdate("insert into myTable (col1, col2) values ('value1',
'value1')");
```

Prudence

Un Statement est une ressource JDBC et il doit être fermé dès qu'il n'est plus nécessaire :

```
stmt.close();
```

La classe Statement implémente AutoCloseable, ce qui l'autorise à être utilisée dans un try-with-resources.

Pour des raisons de performance, il est également possible d'utiliser un Statement en mode batch. Cela signifie, que l'on accumule l'ensemble des requêtes SQL côté client, puis on les envoie en bloc au serveur plutôt que de les exécuter séquentiellement.

```
java.sql.Statement stmt = connection.createStatement();
try {
  stmt.addBatch("update myTable set col3 = 'sameValue' where col1 = col2");
  stmt.addBatch("update myTable set col3 = 'anotherValue' where col1 <> col2");
  stmt.addBatch("update myTable set col3 = 'nullValue' where col1 = null and col2
= null");
  // les requêtes SQL sont soumises au serveur au moment de l'appel à
executeBatch
  stmt.executeBatch();
} finally {
  stmt.close();
}
```

Le ResultSet

Lorsqu'on exécute une requête SQL de type select, JDBC nous donne accès à une instance de ResultSet. Avec un ResultSet, il est possible de parcourir ligne à ligne les résultats de la requête (comme avec un itérateur) grâce à la méthode ResultSet.next. Pour chaque résultat, il est possible d'extraire les données dans un type supporté par Java.

Le ResultSet offre une liste de méthodes de la forme :

```
ResultSet.getXXX(String columnName)
ResultSet.getXXX(int columnIndex)
```

XXX représente le type Java que la méthode retourne. Si on passe un numéro en paramètre, il s'agit du numéro de la colonne dans l'ordre du select.

Prudence

Le numéro de la première colonne est **1**.

```
String request = "select titre, date_sortie, duree from films";

try (java.sql.Statement stmt = connection.createStatement();
     java.sql.ResultSet resultSet = stmt.executeQuery(request);) {
```

```
  // on parcourt l'ensemble des résultats retourné par la requête
  while (resultSet.next()) {
    String titre = resultSet.getString("titre");
    java.sql.Date dateSortie = resultSet.getDate("date_sortie");
    long duree = resultSet.getLong("duree");

    // ...
  }
}
```

Prudence

Un ResultSet est une ressource JDBC et il doit être fermé dès qu'il n'est plus nécessaire :

```
resultSet.close();
```

La classe ResultSet implémente AutoCloseable, ce qui l'autorise à être utilisée dans un try-with-resources.

Le PreparedStatement

Un PreparedStatement est créé à partir d'une des méthodes prepareStatement de l'interface Connection. Lors de l'appel à prepareStatement, il faut passer la requête SQL à exécuter. Cependant, cette requête peut contenir des **?** indiquant l'emplacement des paramètres.

L'interface PreparedStatement fournit des méthodes de la forme :

```
PreparedStatement.setXXX(int parameterIndex, XXX x)
```

XXX représente le type du paramètre, *parameterIndex* sa position dans la requête SQL (attention, le premier paramètre a l'indice **1**) et *x* sa valeur.

Note

Pour positionner un paramètre SQL à *NULL*, il faut utiliser la méthode setNull(int parameterIndex, int sqlType).

```
String request = "insert into films (titre, date_sortie, duree) values (?, ?,
?)";

try (java.sql.PreparedStatement pstmt = connection.prepareStatement(request)) {

  pstmt.setString(1, "live JDBC");
  pstmt.setDate(2, new java.sql.Date(System.currentTimeMillis()));
  pstmt.setInt(3, 120);

  pstmt.executeUpdate();
}
```

Prudence

Un PreparedStatement est une ressource JDBC et il doit être fermé dès qu'il n'est plus nécessaire :

```
pstmt.close();
```

La classe PreparedStatement implémente AutoCloseable, ce qui l'autorise à être utilisée dans un try-with-resources.

Le PreparedStatement reprend une API similaire à celle du Statement :

- une méthode execute pour tous les types de requête SQL
- une méthode executeQuery (qui retourne un ResultSet) pour les requêtes SQL de type select
- une méthode executeUpdate pour toutes les requêtes SQL qui ne sont pas des select

Le PreparedStatement offre trois avantages :

- il permet de convertir efficacement les types Java en types SQL pour les données en entrée
- il permet d'améliorer les performances si on désire exécuter plusieurs fois la même requête avec des paramètres différents. À noter que le PreparedStatement supporte lui aussi le mode batch
- il permet de se prémunir de failles de sécurité telles que l'injection SQL

L'injection SQL

L'injection SQL est une faille de sécurité qui permet à un utilisateur malveillant de modifier une requête SQL pour obtenir un comportement non souhaité par le développeur. Imaginons que le code suivant est exécuté après la saisie par l'utilisateur de son login et de son mot de passe :

```
public boolean isUserAuthorized(String login, String password) throws
SQLException {
  try (java.sql.Statement stmt = connection.createStatement()) {

    String request = "select * from users where login = '" + login
                   + "' and password = '" + password + "'";

    try (java.sql.ResultSet resultSet = stmt.executeQuery(request)) {
      return resultSet.next();
    }
  }
}
```

Le code précédent construit la requête SQL en concaténant des chaînes de caractères à partir des paramètres reçus. Il exécute la requête et s'assure qu'elle retourne au moins un résultat.

Un utilisateur mal intentionné peut alors saisir comme login et mot de passe : ' or '' = '. Ainsi la requête SQL sera :

```
select * from users where login = '' or '' = '' and password = '' or '' = ''
```

Cette requête SQL retourne toutes les lignes de la table users et l'utilisateur sera donc considéré comme autorisé par l'application.

Si on modifie le code précédent pour utiliser un PreparedStatement, ce comportement non souhaité disparaît :

```
public boolean isUserAuthorized(String login, String password) throws
SQLException {
  String request = "select * from users where login = ? and password = ?";
  try (java.sql.PreparedStatement stmt = connection.prepareStatement(request)) {

    stmt.setString(1, login);
    stmt.setString(2, password);

    try (java.sql.ResultSet resultSet = stmt.executeQuery()) {
      return resultSet.next();
    }
  }
}
```

Avec un PreparedStatement, login et password sont maintenant des paramètres de la requête SQL et ils ne peuvent pas en modifier sa structure. La requête exécutée sera équivalente à :

```
select * from users where login = ''' or '''' = ''' and password = ''' or '''' =
'''
```

Récupération des clés générées

Lorsque la base de données est responsable de générer la clé primaire lors d'une opération d'insertion (comme pour les colonnes *AUTO INCREMENT* de MySQL), il est parfois utile de pouvoir récupérer la valeur de cette clé pour pouvoir créer de nouvelles requêtes. La méthode getGeneratedKeys sur un Statement permet de récupérer une instance de ResultSet pour parcourir la liste des clés générées par une requête (la plupart du temps, une seule clé est générée). Ce ResultSet permet d'accéder à une colonne contenant la valeur de la clé. Attention, il est nécessaire de créer un Statement ou un PreparedStatement en passant l'option RETURN_GENERATED_KEYS pour informer JDBC que l'on souhaite récupérer la valeur de la ou des clés après un appel à executeUpdate.

Ci-dessous, un exemple avec un PreparedStatement :

```
String request = "insert into films (titre, date_sortie, duree) values (?, ?,
?)";

try (java.sql.PreparedStatement pstmt = connection.prepareStatement(request,
java.sql.Statement.RETURN_GENERATED_KEYS)) {

  pstmt.setString(1, "live JDBC");
  pstmt.setDate(2, new java.sql.Date(System.currentTimeMillis()));
  pstmt.setInt(3, 120);

  pstmt.executeUpdate();

  try (java.sql.ResultSet resultSet = pstmt.getGeneratedKeys()) {
    if (resultSet.next()) {
        // On récupère la clé dans la première colonne
        int key = resultSet.getInt(1);
        // ...
    }
  }
```

```
}
```

Le CallableStatement

Un CallableStatement permet d'appeler des procédures ou des fonctions stockées. Il est créé à partir d'une des méthodes prepareCall de l'interface Connection. Comme pour le PreparedStatement, il est nécessaire de passer la requête lors de l'appel à prepareCall et l'utilisation de **?** permet de spécifier les paramètres.

Cependant, il n'existe pas de syntaxe standard en SQL pour appeler des procédures ou des fonctions stockées. JDBC définit tout de même une syntaxe compatible avec tous les pilotes JDBC :

Requête JDBC pour l'appel d'une procédure stockée

```
{call nom_de_la_procedure(?, ?, ?, ...)}
```

Requête JDBC pour l'appel d'une fonction stockée

```
{? = call nom_de_la_fonction(?, ?, ?, ...)}
```

Un CallableStatement permet de passer des paramètres en entrée avec des méthodes de type *setXXX* comme pour le PreparedStatement. Il permet également de récupérer les paramètres en sortie avec des méthodes de type *getXXX* comme on peut trouver dans l'interface ResultSet. Comme pour le PreparedStatement, on retrouve les méthodes execute, executeUpdate et executeQuery pour réaliser l'appel à la base de données.

Exemple de procédure stockée MySQL

```
create procedure sayHello (in nom varchar(50), out message varchar(60))
begin
  select concat('hello ', nom, ' !') into message;
end
```

Pour appeler la procédure stockée définit ci-dessus :

```
String request = "{call sayHello(?, ?)}";

try (java.sql.CallableStatement stmt = connection.prepareCall(request)) {
  // on positionne le paramètre d'entrée
  stmt.setString(1, "the world");
  // on appelle la procédure
  stmt.executeUpdate();
  // on récupère le paramètre de sortie
  String message = stmt.getString(2);

  // ...
}
```

Prudence

Un CallableStatement est une ressource JDBC et il doit être fermé dès qu'il n'est plus nécessaire :

```
stmt.close();
```

La classe CallableStatement implémente AutoCloseable, ce qui l'autorise à être utilisée dans un try-with-resources.

La transaction

La plupart des SGBDR intègrent un moteur de transaction. Une transaction est définie par le respect de quatre propriétés désignées par l'acronyme ACID :

- Atomicité

  La transaction garantit que l'ensemble des opérations qui la composent sont soit toutes réalisées avec succès soit aucune n'est conservée.

- Cohérence

  La transaction garantit qu'elle fait passer le système d'un état valide vers un autre état valide.

- Isolation

  Deux transactions sont isolées l'une de l'autre. C'est-à-dire que leur exécution simultanée produit le même résultat que si elles avaient été exécutées successivement.

- Durabilité

  La transaction garantit qu'après son exécution, les modifications qu'elle a apportées au système sont conservées durablement.

Une transaction est définie par un début et une fin qui peut être soit une validation des modifications (*commit*), soit une annulation des modifications effectuées (*rollback*). On parle de **démarcation transactionnelle** pour désigner la portion de code qui doit s'exécuter dans le cadre d'une transaction.

Avec JDBC, il faut d'abord s'assurer que le pilote ne *commite* pas systématiquement à chaque requête SQL (l'auto commit). Une opération de *commit* à chaque requête SQL équivaut en fait à ne pas avoir de démarcation transactionnelle. Sur l'interface Connection, il existe les méthodes setAutoCommit et getAutoCommit pour nous aider à gérer ce comportement. Attention, dans la plupart des implémentations des pilotes JDBC, l'auto commit est activé par défaut (mais ce n'est pas une règle).

À partir du moment où l'auto commit n'est plus actif sur une connexion, il est de la responsabilité du développeur d'appeler sur l'instance de Connection la méthode commit (ou rollback) pour marquer la fin de la transaction.

Le contrôle de la démarcation transactionnelle par programmation est surtout utile lorsque l'on souhaite garantir l'atomicité d'un ensemble de requêtes SQL.

Dans l'exemple ci-dessous, on doit mettre à jour deux tables (*ligne_facture* et *stock_produit*) dans une application de gestion des stocks. Lorsqu'une quantité d'un produit est ajoutée dans une facture alors la même quantité est déduite du stock. Comme les requêtes SQL sont réalisées séquentiellement, il faut s'assurer que soit les deux requêtes aboutissent soit les deux requêtes échouent. Pour cela, on utilise la démarcation transactionnelle.

```
// si nécessaire on force la désactivation de l'auto commit
connection.setAutoCommit(false); boolean transactionOk = false; try {
// on ajoute un produit avec une quantité donnée dans la facture
String requeteAjoutProduit = "insert into ligne_facture (facture_id,
produit_id, quantite) values (?, ?, ?)"; try (PreparedStatement pstmt
= connection.prepareStatement(requeteAjoutProduit)) {
pstmt.setString(1, factureId); pstmt.setString(2, produitId);
pstmt.setLong(3, quantite); pstmt.executeUpdate(); } // on déstocke la
quantité de produit qui a été ajoutée dans la facture String
requeteDestockeProduit = "update stock_produit set quantite =
(quantite - ?) where produit_id = ?"; try (PreparedStatement pstmt =
connection.prepareStatement(requeteDestockeProduit)) {
pstmt.setLong(1, quantite); pstmt.setString(2, produitId);
pstmt.executeUpdate(); } transactionOk = true; } finally { //
L'utilisation d'une transaction dans cet exemple permet d'éviter
d'aboutir à // des états incohérents si un problème survient pendant
l'exécution du code. // Par exemple, si le code ne parvient pas à
exécuter la seconde requête SQL // (bug logiciel, perte de la
connexion avec la base de données, ...) alors // une quantité d'un
produit aura été ajoutée dans une facture sans avoir été // déstockée.
Ceci est clairement un état incohérent du système. Dans ce cas, // on
effectue un rollback de la transaction pour annuler l'insertion dans
// la table ligne_facture. if (transactionOk) { connection.commit(); }
else { connection.rollback(); } }
```

JDBC dans une application Web

JDBC (Java DataBase Connectivity) est l'API standard pour interagir avec les bases données relationnelles en Java. Cette API peut être utilisée dans une application Web.

Accès à une DataSource

Dans un serveur d'application, l'utilisation du DriverManager JDBC est remplacée par celle de la DataSource. L'interface DataSource n'offre que deux méthodes :

```
// Attempts to establish a connection with the data source
Connection getConnection()

// Attempts to establish a connection with the data source
Connection getConnection(String username, String password)
```

Il n'est pas possible de spécifier l'URL de connexion à la base de données avec une DataSource. En revanche, il est possible de configurer dans le serveur une DataSource en lui associant des paramètres de configuration tels que l'URL et les identifiants de connexion. Cette DataSource est gérée comme une ressource par le serveur. Un serveur Java EE maintient l'ensemble de ses ressources sous la forme d'une arborescence. Comme pour une arborescence de fichiers sur un disque ou sur un réseau, il est possible d'accéder aux ressources du serveur à partir de leur nom. La différence avec les gestion de fichiers sur un disque ou sur un réseau est que les ressources d'un serveur sont des interfaces Java.

L'accès aux ressources du serveur se fait par programmation grâce à l'API **JNDI** (*Java Naming and Directory Interface*). JNDI est une API standard de Java permettant de se connecter à des annuaires (notamment les annuaires LDAP). Un serveur Java EE dispose de sa propre implémentation interne d'annuaire pour la gestion de ses ressources.

Les ressources telles que les *DataSources* sont donc stockées dans un annuaire interne et il est possible d'y accéder avec l'API JNDI. Les ressources sont rangées dans une arborescence (comme le sont les fichiers dans des répertoires). Une ressource est stockée dans l'arborescence dont le chemin commence par **java:/comp/env**.

Exemple de récupération d'une DataSource en utilisant l'API JNDI

```
// javax.naming.InitialContext désigne le contexte racine de l'annuaire.
// Un annuaire JDNI est constitué d'instances de javax.naming.Context
// (qui sont l'équivalent des répertoires dans un système de fichiers).
Context envContext = InitialContext.doLookup("java:/comp/env");

// On récupère la source de données dans le contexte java:/comp/env
DataSource dataSource = (DataSource) envContext.lookup("nomDeLaDataSource");
```

La classe InitialContext agit comme une classe *factory*. Elle permet de fabriquer un contexte qui représente une position dans l'arborescence des ressources JNDI. Le contexte JNDI **java:/comp/env** est un contexte particulier. Il désigne l'ensemble des composants Java EE disponibles dans l'environnement (env) du composant Java EE (comp) courant (dans notre cas l'application Web). la méthode lookup permet ensuite de récupérer une ressource à partir de son

nom. Il faut effectuer un trans-typage (*cast*) vers le type réel de la ressource, pour le cas qui nous intéresse : une DataSource.

Injection d'une DataSource

Afin de simplifier l'accès à une DataSource, il est possible de l'injecter directement comme attribut d'un composant Java EE. Comme les Servlets sont des composants Java EE, il est possible d'ajouter l'annotation Resource sur un attribut de type DataSource :

Injection d'une DataSource dans une Servlet

```
import java.io.IOException;
import java.sql.Connection;

import javax.annotation.Resource;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.sql.DataSource;

@WebServlet("/MyServlet")
public class MyServlet extends HttpServlet {

  @Resource(name = "nomDeLaDataSource")
  private DataSource dataSource;

  @Override
  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
                          throws ServletException, IOException {

    try (Connection connection = dataSource.getConnection()) {
      // ...
    }

  }

}
```

L'annotation @Resource permet de spécifier le nom de la DataSource grâce à l'attribut *name*.

Avertissement

Pour avoir accès à l'annotation Resource, vous devez ajouter une dépendance dans votre projet. Pour un projet Maven, il suffit d'ajouter dans le fichier `pom.xml` la dépendance suivante :

```
<dependency>
  <groupId>javax.annotation</groupId>
  <artifactId>javax.annotation-api</artifactId>
  <version>1.3.2</version>
  <scope>provided</scope>
</dependency>
```

Déclaration de la DataSource dans le fichier web.xml

Le fichier de déploiement `web.xml` doit déclarer la DataSource comme une ressource de l'application. Cela va permettre au serveur d'application de permettre à l'application de se connecter à la base de données associée. Pour cela, on utilise l'élément `<resource-ref>` dans le fichier `web.xml` :

Déclaration de la DataSource dans le fichier web.xml

```
<resource-ref>
  <res-ref-name>nomDeLaDataSource</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
</resource-ref>
```

Mais comment le serveur d'application fait-il pour lier une DataSource avec une connexion vers une base de données ? Malheureusement, il n'existe pas de standard et chaque serveur d'application dispose de sa procédure. Nous allons voir dans la section suivante comment créer une DataSource spécifiquement pour Tomcat.

Déclaration d'une DataSource dans Tomcat

Tomcat n'est pas à proprement parler un serveur d'application, il s'agit juste d'un conteneur de Servlet. Néanmoins, il peut déployer des applications Web Java et il supporte l'annotation @Resource ainsi que la configuration d'une DataSource dans le serveur ou dans l'application.

Une connexion JDBC est réalisée à travers un pilote. Pour déclarer une DataSource vers une base de données MySQL, par exemple, nous devons installer le pilote MySQL dans le serveur. Pour cela, il vous faut télécharger le pilote et le placer dans le répertoire `lib` situé dans le répertoire d'installation du serveur.

Une fois, le pilote ajouté, il est possible de déclarer la DataSource dans le fichier `conf/server.xml`. Si vous utilisez Tomcat dans Eclipse, alors vous devez disposer d'un projet `Servers` qui a été créé automatiquement dans votre espace de travail. Dedans, se trouvent toutes les configurations des serveurs que vous avez créés. Pour Tomcat 9, le nom par défaut est `Tomcat v9.0 Server at localhost-config`. Dans ce répertoire, se trouve le fichier `server.xml` qu'il va falloir modifier.

Note

L'intégration de Tomcat dans Eclipse crée automatiquement une copie de la configuration originale du serveur pour chacune des instances de serveur créées dans Eclipse. Ainsi, il est possible de déclarer plusieurs serveurs Tomcat dans Eclipse qui possèdent chacun leur configuration spécifique.

Dans le fichier `server.xml`, vous devez ajouter avant la fin de l'élément `<Host />`, les informations de déploiement de votre application.

Avertissement

Si vous utilisez le serveur Tomcat dans Eclipse, la balise `<Context/>` est automatiquement ajoutée par Eclipse lorsque vous ajoutez votre application dans le serveur. Attention, cette balise sera aussi supprimée si vous supprimez l'application du serveur.

Exemple de balise de contexte de déploiement dans le fichier server.xml

```
<Context docBase="nomAppli"
         path="/nomAppli"
         reloadable="true"
         source="org.eclipse.jst.jee.server:jdbc">
  <Resource name="[nomDataSource]"
            auth="Container"
            type="javax.sql.DataSource"
            maxTotal="100"
            maxIdle="30"
            maxWaitMillis="10000"
            username="[USERNAME]"
            password="[PASSWORD]"
            driverClassName="com.mysql.jdbc.Driver"
            url="jdbc:mysql://[HOST]:3306/[NOM BASE]" />
</Context>
```

Si vous ne voulez pas modifier la configuration du serveur, il est possible d'ajouter un fichier `src/main/webapp/META-INF/context.xml` dans votre projet Maven et de déclarer à l'intérieur l'élément `<Resource />` :

Exemple de fichier context.xml dans l'application

```
<Context>

  <Resource name="[nomDataSource]"
            auth="Container"
            type="javax.sql.DataSource"
            maxTotal="100"
            maxIdle="30"
            maxWaitMillis="10000"
            username="[USERNAME]"
            password="[PASSWORD]"
            driverClassName="com.mysql.jdbc.Driver"
            url="jdbc:mysql://[HOST]:3306/[NOM BASE]" />

</Context>
```

Note

Pour une présentation de la déclaration des *data sources* pour différents SGBDR, reportez-vous à la documentation de Tomcat.

Dans sa gestion des *data sources*, Tomcat inclut la gestion d'un *pool* de connexions en s'appuyant sur la bibliothèque Apache DBCP. L'ensemble des paramètres de configuration du *pool* de connexions est disponibles dans la documentation de DBCP. Ces paramètres sont utilisables comme attributs de l'élément `<Resource />`.

Les entités JPA

Nous avons vu que l'API JDBC nous permet d'écrire des programmes Java qui interagissent avec des bases de données. JDBC nous assure que le code Java sera semblable quel que soit le SGBDR utilisé (mais le code SQL pourra bien sûr être différent en exploitant telle ou telle fonctionnalité non standard fournie par le SGBDR).

Néanmoins, JDBC a quelques inconvénients :

- l'API est verbeuse et répétitive. Pour un programme de quelques centaines de lignes de code, elle se révèle très efficace. Mais pour des applications plus volumineuses, la quantité de code nécessaire (notamment SQL) peut devenir une source de ralentissement du développement.
- la gestion des nombreuses ressources (Connection, Statement, ResultSet) est une source permanente de bugs pour les développeurs. Il est donc très facile d'écrire des applications qui perdent des ressources.
- JDBC n'offre qu'un service limité : un système d'échange avec une base de données (même s'il le fait très bien).

Les ORM (Object-Relational Mapping)

Les ORM sont des frameworks qui, comme l'indique leur nom, permettent de créer une correspondance entre un modèle objet et un modèle relationnel de base de données. Un ORM fournit généralement les fonctionnalités suivantes :

- génération à la volée des requêtes SQL les plus simples (CRUD)
- prise en charge des dépendances entre objets pour la mise en jour en cascade de la base de données
- support pour la construction de requêtes complexes par programmation

Java EE fournit une API standard pour l'utilisation d'un ORM : **JPA (Java Persistence API)** (la JSR-338). Il existe plusieurs implémentations open source qui respectent l'API JPA : EclipseLink (qui est aussi l'implémentation de référence), Hibernate (JBoss - Red Hat), OpenJPA (Apache).

Toutes ces implémentations sont bâties sur JDBC. Nous retrouverons donc les notions de pilote, de data source et d'URL de connexion lorsqu'il s'agira de configurer l'accès à la base de données.

Note

Pour ce cours, nous utiliserons comme exemple l'implémentation fournie par Hibernate. À version identique, le code présenté devrait être compatible avec les autres implémentations de JPA.

Les entités JPA

JPA permet de définir des entités (*entities*). Une entité est simplement une instance d'une classe qui sera *persistante* (que l'on pourra sauvegarder dans / charger depuis une base de données relationnelle). Une entité est signalée par l'annotation @Entity sur la classe. De plus, une entité JPA **doit disposer d'un ou plusieurs attributs définissant un identifiant** grâce à l'annotation @Id. Cet identifiant correspondra à la clé primaire dans la table associée.

Un exemple de classe entité avec la déclaration de son identifiant

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
```

```
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Individu {

    @Id
    // Permet de définir la statégie de génération
    // de la clé lors d'une insertion en base de données.
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
}
```

Il existe un grand nombre d'annotations JPA servant à préciser comment la correspondance doit être faite entre le modèle objet et le modèle relationnel de base de données. Il est possible de déclarer cette correspondance à l'aide du fichier XML `orm.xml`. Cependant, la plupart de développeurs préfèrent utiliser des annotations.

La liste ci-dessous résume les annotations les plus simples et les plus utiles pour définir une entité et ses attributs :

- @Entity

  Définit qu'une classe est une entité. Le nom de l'entité est donné par l'attribut `name` (en son absence le nom de l'entité correspond au nom de la classe).

- @Id

  Définit l'attribut qui sert de clé primaire dans la table. Il est recommandé au départ d'utiliser un type primitif, un wrapper de type primitif ou une String pour représenter un id. Pour les clés composites, la mise en œuvre est plus compliquée. Afin de ne pas se compliquer inutilement la tâche, il vaut mieux prévoir une clé simple pour chaque entité.

- @Basic

  Définit un mapping simple pour un attribut (par exemple `VARCHAR` pour String). Si on ne souhaite pas changer la valeur des attributs par défaut de cette annotation, alors il est possible de ne pas la spécifier puisqu'elle représente le mapping par défaut.

- @Temporal

  Pour un attribut de type java.util.Date et java.util.Calendar, cette annotation permet de préciser le type de mapping vers le type SQL (DATE, TIME ou TIMESTAMP).

- @Transient

  Indique qu'un attribut **ne doit pas** être persistant. Cet attribut ne sera donc jamais pris en compte lors de l'exécution des requêtes vers la base de données.

- @Lob

  Indique que la colonne correspondante en base de données est un LOB (large object).

Certaines annotations sont utilisées pour fournir des informations sur la base de données sous-jacente :

- @Table

  Permet de définir les informations sur la table représentant cette entité en base de données. Il est possible de définir le nom de la table grâce à l'attribut `name`. Par défaut le nom de la table correspond au nom de l'entité (qui par défaut correspond au nom de la classe).

- @GeneratedValue

  Indique la stratégie à appliquer pour la génération de la clé lors de l'insertion d'une entité en base. Les valeurs possibles sont données par l'énumération GenerationType. Si vous utilisez MySQL et la propriété `autoincrement` sur une colonne, alors vous devez utiliser `GenerationType.IDENTITY` (ce sera le cas pour les exemples de ce cours). Si vous utilisez Oracle et un système de séquence, alors vous devez utiliser `GenerationType.SEQUENCE` et préciser le nom de la séquence dans l'attribut `generator` de `@GeneratedValue`.

- @Column

  Permet de déclarer des informations relatives à la colonne sur laquelle un attribut doit être mappé. Si cette annotation est absente, le nom de la colonne correspond au nom de l'attribut. Avec cette annotation, il est possible de donner le nom de la colonne (l'attribut `name`) mais également si l'attribut doit être pris en compte pour des requêtes d'insertion (l'attribut `insertable`) ou de mise à jour (l'attribut `updatable`). Certains outils sont capables d'exploiter les annotations pour créer les bases de données. Dans ce cas, d'autres attributs sont disponibles pour ajouter toutes les contraintes nécessaires (telles que `length` ou `nullable`) et donner ainsi une description complète de la colonne.

Un exemple plus complet de classe entité

```
import java.util.Calendar;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Lob;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;
import javax.persistence.Transient;

@Entity
@Table(name="individu")
public class Individu {

    @Id
    @Column(name="individuId")
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    @Basic
    @Column(length=30, nullable=false)
```

```
    private String nom;

    @Basic
    @Column(length=30, nullable=false)
    private String prenom;

    @Transient
    private Integer age;

    @Temporal(TemporalType.DATE)
    private Calendar dateNaissance;

    @Temporal(TemporalType.TIMESTAMP)
    @Column(updatable=false)
    private Calendar dateCreation;

    @Lob
    @Basic(fetch=FetchType.LAZY)
    private byte[] image;

    // les getter/setter ont été omis pour faciliter la lecture
}
```

À l'entité JPA ci-dessus, on pourra faire correspondre la table MySQL :

La script de création de la table associée à l'entité

```
CREATE TABLE `individu` (
  `individuId` int NOT NULL AUTO_INCREMENT,
  `nom` varchar(30) NOT NULL,
  `prenom` varchar(30) NOT NULL,
  `dateNaissance` DATE,
  `dateCreation` TIMESTAMP,
  `image` BLOB,
  PRIMARY KEY (`individuId`)
);
```

L'EntityManager

Les annotations JPA que nous avons vues dans la section précédente, ne servent à rien si elle ne sont pas exploitées programmatiquement. Dans JPA, l'interface centrale qui va exploiter ces annotations est l'interface EntityManager.

Obtenir un EntityManager

JPA est une spécification. Pour pouvoir l'utiliser, il faut avoir à sa disposition une implémentation compatible avec JPA. Dans le cadre de ce cours, nous utiliserons Hibernate. Pour ajouter Hibernate dans un projet Java, nous pouvons utiliser Maven pour gérer notre projet et ajouter comme dépendances dans le fichier `pom.xml` :

```
<dependency>
  <groupId>javax.xml.bind</groupId>
  <artifactId>jaxb-api</artifactId>
  <version>2.3.1</version>
 </dependency>

<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>5.4.9.Final</version>
</dependency>
```

Template de projet JPA

Vous pouvez `télécharger le projet d'exemple`. Il s'agit d'un projet Maven avec une dépendance vers Hibernate et le pilote JDBC MySQL.

Il faut fournir à l'implémentation de JPA un fichier XML de déploiement nommé **persistence.xml**.

Contenu du fichier persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">
  <persistence-unit name="monUniteDePersistance">
    <!-- la liste des noms complets des classes représentant
         les entités gérées par cette unité de persistance  -->
    <class>ma.classe.Entite</class>
    <properties>
      <!-- une propriété de configuration propre à l'implémentation de JPA -->
      <property name="une propriété" value="une valeur" />
    </properties>
  </persistence-unit>
</persistence>
```

Dans ce fichier, on déclare une ou plusieurs unités de persistance grâce à la balise `<persitence-unit>`. Chaque unité de persistance est identifiée par un nom et contient la liste des classes entités gérées par cette unité avec la balise `<class>`. La balise `<properties>` permet de spécifier des propriétés propres à une implémentation de JPA et que indique comment se connecter au SGBDR.

Note

La liste complète des paramètres de configuration propres à Hibernate est disponible dans la documentation officielle.

Le fichier **persistence.xml doit se situer dans le répertoire META-INF** et être disponible dans le classpath à l'exécution. Dans un projet Maven, il suffit de créer ce fichier dans le répertoire **src/main/resources/META-INF** du projet (créez les répertoires manquants si nécessaire).

Astuce

Le fichier **persistence.xml** est déjà inclus dans le `projet d'exemple`.

On ajoute ensuite dans le fichier **persistence.xml** les propriétés permettant de décrire la connexion à la base de données.

Contenu du fichier persistence.xml avec les propriétés de connexion

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
             http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
             version="2.1">
  <persistence-unit name="monUniteDePersistance">
    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <properties>
      <property name="javax.persistence.jdbc.driver"
value="com.mysql.jdbc.Driver" />
      <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://localhost:3306/database" />
      <property name="javax.persistence.jdbc.user" value="root" />
      <property name="javax.persistence.jdbc.password" value="root" />
      <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQLDialect" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.format_sql" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```

Créer une fabrique d'EntityManager

Pour initialiser JPA, il faut utiliser la classe Persistence. Grâce à cette classe, nous allons pouvoir créer une instance de EntityManagerFactory. Cette dernière, comme son nom l'indique, permet de fabriquer une instance d"EntityManager.

Exemple d'initialisation de JPA

```
// on spécifie le nom de l'unité de persistence en paramètre
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("monUniteDePersistance");

EntityManager entityManager = emf.createEntityManager();
```

Il existe une méthode `Persistence.createEntityManagerFactory(java.lang.String, java.util.Map properties)` qui permet de spécifier des propriétés comme second paramètre. Ces propriétés s'ajoutent ou remplacent celles déclarées dans la balise `<properties>` du fichier *persistence.xml* pour l'unité de persistance.

Pour des raisons de performance, **une seule instance** de la classe EntityManagerFactory devrait être créée par unité de persistance et par application.

Par contre, une instance d"EntityManager n'est pas prévue pour être conservée trop longtemps. De plus, un EntityManager n'est pas conçu pour être utilisé dans un environnement concurrent. Pour des applications multi-threadées, on utilisera une instance d"EntityManager par thread.

Un EntityManagerFactory et un EntityManager représentent des ressources système et **doivent être fermées** par un appel à leur méthode `close()` dès qu'elles ne sont plus utiles.

```
EntityManager entityManager = emf.createEntityManager();
try {

  // ...

}
finally {
  entityManager.close();
}
```

Ni EntityManagerFactory ni EntityManager n'implémentent l'interface `AutoCloseable`. Il n'est donc pas possible d'utiliser la syntaxe du try-with-resources avec ces interfaces.

Prudence

Hibernate **impose** de fermer correctement les instances de type EntityManagerFactory et EntityManager. Si vous ne le faites pas, votre programme ne s'arrêtera pas même si vous arrivez à la fin de la méthode *main* de votre programme.

Note

Nous verrons par la suite que la procédure pour récupérer une instance d'un EntityManager est différente si nous développons une application Java EE destinée à être déployée dans un serveur d'application.

Manipuler des entités à partir d'un EntityManager

À partir d'une instance d"EntityManager, nous allons pouvoir manipuler les entités afin de les créer, les modifier, les charger ou les supprimer. Pour cela, nous disposons de six méthodes :

- find
- persist
- merge
- detach
- refresh
- remove

L"EntityManager va prendre en charge la relation avec la base de données et la génération des requêtes SQL nécessaires.

Exemples d'appel à l'EntityManager

```
EntityManager entityManager = ... // nous faisons l'hypothèse que nous disposons
d'une instance
Individu individu = new Individu();
individu.setPrenom("John");
individu.setNom("Smith");

// Demande d'insertion dans la base de données
entityManager.persist(individu);
```

```
// Demande de chargement d'une entité.
// Le second paramètre correspond à la valeur de la clé de l'entité recherchée.
individu = entityManager.find(Individu.class, individu.getId());

// Demande de suppression (delete)
entityManager.remove(individu);
```

De plus, l'implémentation JPA se charge d'extraire ou au contraire de positionner les attributs dans l'instance de l'entité. Par exemple, un appel à `find` retourne bien une instance de la classe spécifiée par le premier paramètre. Cette instance aura ses attributs renseignés à partir des valeurs des colonnes sur lesquelles ils ont été mappés.

Pour les opérations qui modifient une entité (telles que `persist` ou `remove`), il faut que l'appel se fasse dans le cadre d'une transaction. Grâce à la méthode `EntityManager.getTransaction()`, il est possible de récupérer la transaction est de gérer la démarcation comme ci-dessous :

Gestion de la transaction avec un EntityManager

```
EntityManager entityManager = ... // nous faisons l'hypothèse que nous disposons
d'une instance

entityManager.getTransaction().begin();
boolean transactionOk = false;
try {
// ..

transactionOk = true;
}
finally {
    if(transactionOk) {
        entityManager.getTransaction().commit();
    }
    else {
        entityManager.getTransaction().rollback();
    }
}
```

Note

Nous verrons plus tard que l'exemple ci-dessus **ne fonctionne pas** dans un serveur Java EE qui utilise l'API de gestion des transactions JTA.

Attention cependant à ne pas croire que JPA est simplement un framework pour générer du SQL. Une des difficultés dans la maîtrise de JPA consiste justement à comprendre comment il gère le cycle de vie des entités indépendamment de la base de données. Ainsi, on ne retrouve pas sur l'interface EntityManager des noms de méthodes qui correspondent aux instructions SQL `INSERT`, `SELECT`, `UPDATE` et `DELETE`. Il ne s'agit pas d'un effet de style, les méthodes pour manipuler les entités ont un comportement qui dépasse la simple exécution de requêtes SQL.

À votre avis

Quelles sont les requêtes SQL exécutées par le code ci-dessous ?

```
EntityManager entityManager = ... // nous faisons l'hypothèse que nous disposons
d'une instance
```

```
Individu individu = new Individu();
individu.setNom("David");
individu.setPrenom("Gayerie");

entityManager.getTransaction().begin();
boolean transactionOk = false;
try {
    entityManager.persist(individu);

    individu.setPrenom("Jean");

    entityManager.merge(individu);

    entityManager.remove(individu);

    transactionOk = true;
}
finally {
    if(transactionOk) {
        entityManager.getTransaction().commit();
    }
    else {
        entityManager.getTransaction().rollback();
    }
}
```

Un EntityManager cherche à limiter les interactions inutiles avec la base de données. Ainsi, tant qu'une transaction est en cours, le moteur JPA n'effectuera aucune requête SQL, à moins d'y être obligé pour garantir l'intégrité des données. Il attendra si possible le commit de la transaction. Ainsi si une entité est créée puis modifiée au cours de la même transaction, plutôt que d'exécuter deux requêtes SQL (INSERT puis UPDATE), l'"EntityManager attendra la fin de la transaction pour réaliser une seule requête SQL (INSERT) avec les données définitives.

La méthode persist

La méthode `persist` ne se contente pas d'enregistrer une entité en base, elle positionne également la valeur de l'attribut représentant la clé de l'entité. La détermination de la valeur de la clé dépend de la stratégie spécifiée par @GeneratedValue. L'insertion en base ne se fait pas nécessairement au moment de l'appel à la méthode `persist` (on peut toutefois forcer l'insertion avec la méthode `EntityManager.flush()`). Cependant, l'"EntityManager garantit que des appels successifs à sa méthode `find` permettront de récupérer l'instance de l'entité.

C'est une erreur d'appeler la méthode `EntityManager.persist` en passant une entité dont l'attribut représentant la clé est non null. La méthode jette alors l'exception EntityExistsException.

La méthode find

La méthode `EntityManager.find (Class<T>, Object)` permet de rechercher une entité en donnant sa clé primaire. Un appel à cette méthode ne déclenche pas forcément une requête `SELECT` vers la base de données.

En effet, un EntityManager agit également comme un cache au dessus de la base de données. Ainsi, il garantit l'unicité des instances des objets. Si la méthode `find` est appelée plusieurs fois sur la même instance d'un EntityManager avec une clé identique, alors l'instance retournée est toujours **la même**.

```
EntityManager entityManager = ... // nous faisons l'hypothèse que nous disposons
d'une instance

Individu individu  = entityManager.find(Individu.class, 1);
// Pour le second appel à find, aucune requête SQL n'est exécutée.
// L'EntityManager se contente de retourner la même instance que précédemment.
Individu individu2 = entityManager.find(Individu.class, 1);

// individu == individu2
```

Note

Il existe également la méthode `EntityManager.find(Class<T>, Object, LockModeType)`. Cette méthode permet de récupérer une entité en posant un verrou. Elle est utilisée pour réaliser un verrouillage optimiste ou pessimiste (appelé parfois `select for update` en SQL).

La méthode refresh

La méthode `EntityManager.refresh(Object)` annule toutes les modifications faites sur l'entité durant la transaction courante et recharge son état à partir des valeurs en base de données.

Note

Il existe également la méthode `EntityManager.refresh(Class<T>, Object, LockModeType)`. Cette méthode permet de rafraîchir une entité en vérouillant l'accès en écriture. Elle est utilisée pour réaliser un verrouillage optimiste ou pessimiste (appelé parfois `select for update` en SQL).

La méthode merge

La méthode `EntityManager.merge(T)` est parfois considérée comme la méthode permettant de réaliser les `UPDATE` des entités en base de données. Il n'en est rien et la sémantique de la méthode `merge` est très différente. En fait, il **n'existe pas** à proprement parlé de méthode pour réaliser la mise à jour d'une entité. Un EntityManager surveille les entités dont il a la charge et réalise les mises à jour si nécessaire au commit de la transaction. Par exemple le code ci-dessous suffit à déclencher une requête SQL UPDATE :

Mise à jour implicite d'une entité

```
EntityManager entityManager = ... // nous faisons l'hypothèse que nous disposons
d'une instance

entityManager.getTransaction().begin();
try {
    Individu individu = entityManager.find(Individu.class, 1L);
    if (individu != null) {
        individu.setPrenom("Vincent");
    }
    // Si le prénom a été modifié, JPA est
```

```
    // capable de le détecter et de déclencher un UPDATE
    // au moment du commit.
    entityManager.getTransaction().commit();
}
catch (RuntimeException e) {
    entityManager.getTransaction().rollback();
    throw e;
}
```

Si un EntityManager détecte automatiquement les modifications des entités dont il a la charge, à quoi peut donc servir la méthode `EntityManager.merge(T)` ? En fait si vous créez vous même une instance d'une entité et que vous positionnez la clé, cette entité n'est gérée par aucun EntityManager. Pour qu'un EntityManager prenne en compte votre entité, il faut appeler la méthode `merge` :

Utilisation de la méthode merge

```
EntityManager entityManager = ... // nous faisons l'hypothèse que nous disposons
d'une instance

entityManager.getTransaction().begin();

Individu individu = new Individu();
// on positionne explicitement l'id de l'entité
individu.setId(1L);

try {
    // il est très important de remplacer notre instance
    // par celle retournée par l'EntityManager après un merge.
    individu = entityManager.merge(individu);

    // on rafraîchit les données de la nouvelle entité
    entityManager.refresh(individu);

    // l'instance de individu contient bien le prénom stocké en base
    // de données (l'appel à merge à récupérer l'information)
    individu.setPrenom("Vincent");

    // JPA est capable de détecter que l'age de l'individu a été modifié
    // et qu'il faut réaliser un UPDATE SQL au moment du commit.
    entityManager.getTransaction().commit();
}
catch (RuntimeException e) {
    entityManager.getTransaction().rollback();
    throw e;
}
```

L'inverse de la méthode `EntityManager.merge(T)` est `EntityManager.detach(Object)` qui annule la gestion d'une entité par l'"EntityManager.

La méthode detach

Comme son nom l'indique, la méthode `EntityManager.detach(Object)` détache une entité, c'est-à-dire que l'instance passée en paramètre ne sera plus gérée par l'"EntityManager. Ainsi, lors du commit de la transaction, les modifications faites sur l'entité détachée ne seront pas prises en compte.

La méthode remove

La méthode `EntityManager.remove(Object)` supprime une entité. Si l'entité a déjà été persistée en base de données, cette méthode entraînera une requête SQL DELETE.

Les requêtes JPA

Les méthodes `find`, `persist`, `merge`, `detach`, `refresh` et `remove` disponibles avec une instance d'un EntityManager permettent de gérer simplement une entité mais ne permettent pas de réaliser des requêtes très élaborées.

Heureusement, un EntityManager fournit également différentes API pour exécuter des requêtes. Le principe est toujours le même :

1. On crée un objet de type Query ou TypedQuery grâce à l'API
2. Pour les requêtes paramétrées, on positionne la valeur des paramètres grâce aux méthodes `setParameter(String name, XXX xxx)` ou `setParameter(int position, XXX xxx)`
3. On peut optionnellement positionner plusieurs autres informations pour la requête (par exemple, le nombre maximum de résultats pour une consultation grâce à la méthode `setMaxResults(int)`)
4. On exécute la requête grâce aux méthodes `executeUpdate()` (pour un update ou un delete), `getSingleResult()` (pour une requête SELECT ne retournant qu'un seul résultat) ou `getResultList()` (pour une requête SELECT retournant une liste de résultats).

Pour les différents exemples de requêtes qui suivent, nous nous baserons sur l'entité JPA suivante :

```
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "individu")
public class Individu {
  @Id
  @Column(name = "individuId")
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @Basic
  @Column(length = 30, nullable = false)
```

```
  private String nom;

  @Basic
  @Column(length = 30, nullable = false)
  private String prenom;

  @Column(length = 3, nullable = false)
  private Integer age;

  // les getters et setters sont omis ...
}
```

Cette entité JPA correspond à la table MySQL :

```
CREATE TABLE `individu` (
  `individuId` int NOT NULL AUTO_INCREMENT,
  `nom` varchar(30) NOT NULL,
  `prenom` varchar(30) NOT NULL,
  `age` int(3) NOT NULL,
  PRIMARY KEY (`individu_id`)
);
```

Les requêtes Natives

Les requêtes natives en JPA désignent les requêtes SQL. On crée une requête native à partir des méthodes `EntityManager.createNativeQuery(...)`.

Exemple : récupérer tous les individus

```
List<Individu> individus = null;
individus = entityManager.createNativeQuery("select * from individu",
Individu.class)
                         .getResultList();
```

Exemple : récupérer tous les individus âgés au plus de ageMax

```
int ageMax = 25;
List<Individu> individus = null;
individus = entityManager
              .createNativeQuery("select * from individu where age <= ?",
Individu.class)
              .setParameter(1, ageMax)
              .getResultList();
```

Exemple : connaître le nombre d'individus enregistrés

```
long result = (Long) entityManager
                   .createNativeQuery("select count(1) from individu")
                   .getSingleResult();
```

Exemple : Suppression d'un individu dont l'id est individuId

```
long individuId = 1;
// Cette requête nécessite une transaction active
entityManager.createNativeQuery("delete from individu where individuId = ?")
             .setParameter(1, individuId)
             .executeUpdate();
```

Les requêtes JPQL

Avec JPA, il est possible d'utiliser un autre langage pour l'écriture des requêtes, il s'agit du **JPA Query Language** (JPQL). Ce langage est un langage de requête objet. L'objectif n'est plus d'écrire des requêtes basées sur le modèle relationnel des tables mais sur le modèle objet des classes Java.

Pour une **introduction à la syntaxe du JPQL**, reportez-vous au Wikibook.

On crée une requête JPQL à partir des méthodes `EntityManager.createQuery(...)`.

Exemple : récupérer tous les individus

```
List<Individu> individus = null;
individus = entityManager.createQuery("select i from Individu i", Individu.class)
                         .getResultList();
```

Exemple : récupérer tous les individus âgés au plus de ageMax

```
int ageMax = 25;
List<Individu> individus = null;
individus = entityManager
                .createQuery("select i from Individu i where i.age <= :ageMax",
Individu.class)
                .setParameter("ageMax", ageMax)
                .getResultList();
```

Exemple : connaître le nombre d'individus enregistrés

```
long result = (Long) entityManager.createQuery("select count(i) from Individu i")
                           .getSingleResult();
```

Exemple : Suppression d'un individu dont l'id est individuId

```
long individuId = 1;
// Cette requête nécessite une transaction active
entityManager.createQuery("delete from Individu i where i.id = :id")
             .setParameter("id", individuId)
             .executeUpdate();
```

Sur des exemples aussi simples que les exemples précédents, le JPQL semble très proche du SQL. Cependant, avec le JPQL, on ne fait référence qu'aux objets et à leurs attributs, **jamais** au nom des tables et des colonnes.

Ainsi, dans la requête JPQL suivante :

```
select individu from Individu individu
```

individu désigne la variable contenant l'instance courante de la classe Individu. Il ne s'agit absolument pas d'un alias de table comme en SQL. La déclaration d'une variable en JPQL est obligatoire ! Alors qu'un alias de table SQL est optionnel.

Enfin, le JPQL introduit une nouvelle façon de déclarer un paramètre dans une requête sous la forme `:nom`. Les paramètres disposent ainsi d'un nom explicite, rendant ainsi le code plus facile à lire et à maintenir.

Les requêtes par programmation

Lorsque l'on souhaite construire une requête JPQL dynamiquement, il n'est pas toujours très facile de construire la requête par simple concaténation de chaînes de caractères. Pour cela, JPA fournit une API permettant de définir entièrement une requête JPQL par programmation. On crée cette requête à travers un CriteriaBuilder que l'on peut récupérer grâce à la méthode `EntityManager.getCriteriaBuilder()`.

Exemple : récupérer tous les individus

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Individu> query = builder.createQuery(Individu.class);
Root<Individu> i = query.from(Individu.class);
query.select(i);

List<Individu> individus = entityManager.createQuery(query).getResultList();
```

Exemple : récupérer tous les individus âgés au plus de ageMax

```
int ageMax = 25;

CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Individu> query = builder.createQuery(Individu.class);
Root<Individu> i = query.from(Individu.class);
query.select(i);
query.where(builder.lessThanOrEqualTo(i.get("age").as(int.class), ageMax));

List<Individu> individus = entityManager.createQuery(query).getResultList();
```

Exemple : connaître le nombre d'individus enregistrés

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Long> query = builder.createQuery(Long.class);
Root<Individu> i = query.from(Individu.class);
query.select(builder.count(i));

long result = entityManager.createQuery(query).getSingleResult();
```

Pour des exemples supplémentaires d'utilisation du CriteriaBuilder, reportez-vous au Wikibook.

Utilisation de requêtes nommées

L'utilisation de requêtes peut rendre l'application difficile à comprendre et à faire évoluer. En effet, les requêtes sont mêlées au code source Java sous la forme de chaîne de caractères. Si vous préférez regrouper toutes vos requêtes dans des parties clairement identifiées du code, alors vous pouvez utiliser les requêtes nommées (*named queries*).

Une requête nommée permet d'associer un identifiant de requête à une requête JPQL. On utilise pour cela l'annotation @NamedQuery que l'on peut placer sur la classe de l'entité pour centraliser toutes les requêtes relatives à cette entité.

Déclaration d'une requête nommée

```
package dev.gayerie;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQuery;

@Entity
@NamedQuery(name="findIndividuByNom", query="select i from Individu i where i.nom
= :nom")
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String nom;

  // getters et setters omis

}
```

Dans l'exemple ci-dessus, on déclare une requête que l'on nomme « findIndividuByNom ».

Si vous voulez nommer plusieurs requêtes, vous devez utiliser l'annotation @NamedQueries pour les regrouper toutes sous la forme d'un tableau.

Déclaration de plusieurs requêtes nommées

```
package dev.gayerie;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;

@Entity
```

```
@NamedQueries({
  @NamedQuery(name="findIndividuByNom", query="select i from Individu i where
i.nom = :nom"),
  @NamedQuery(name="deleteIndividuByNom", query="delete from Individu i where
i.nom = :nom"),
  @NamedQuery(name="deleteAllIndividus", query="delete from Individu i")
})
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String nom;

  // getters et setters omis

}
```

On peut ensuite créer et exécuter des requêtes nommées à partir d'un EntityManager.

```
Individu individu = entityManager.createNamedQuery("findIndividuByNom",
Individu.class)
                                 .setParameter("nom", "David Gayerie")
                                 .getSingleResult();
entityManager.getTransaction().begin();
entityManager.createNamedQuery("deleteIndividuByNom")
             .setParameter("nom", "David Gayerie")
             .executeUpdate();
entityManager.getTransaction().commit();
```

Note

Il est également possible de créer des requêtes nommées en SQL en utilisant les annotations @NamedNativeQuery et @NamedNativeQueries sur le même principe qu'avec des requêtes nommées en JPQL.

Avertissement

Même si une requête nommée est déclarée dans une classe, son nom est globale à l'ensemble de l'unité de persistance. Il faut donc absolument éviter tout doublon dans le nom des requêtes.

Les relations avec JPA

Une des fonctionnalités majeures des ORM est de gérer les relations entre objets comme des relations entre tables dans un modèle de base de données relationnelle. JPA définit des modèles de relation qui peuvent être déclarés par annotation.

Les relations sont spécifiées par les annotations : @OneToOne, @ManyToOne, @OneToMany, @ManyToMany.

Pour une présentation approfondie des relations dans JPA, reportez-vous au Wikibook.

La relation 1:1 (one to one)

L'annotation @OneToOne définit une relation 1:1 entre deux entités. Si cette relation n'est pas forcément très courante dans un modèle relationnel de base de données, elle se rencontre très souvent dans une modèle objet.

Exemple de relation OneToOne

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @OneToOne
  private Abonnement abonnement;

  // getters et setters omis ...
}
```

La classe associée dans la relation OneToOne

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Abonnement {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  // getters et setters omis ...
}
```

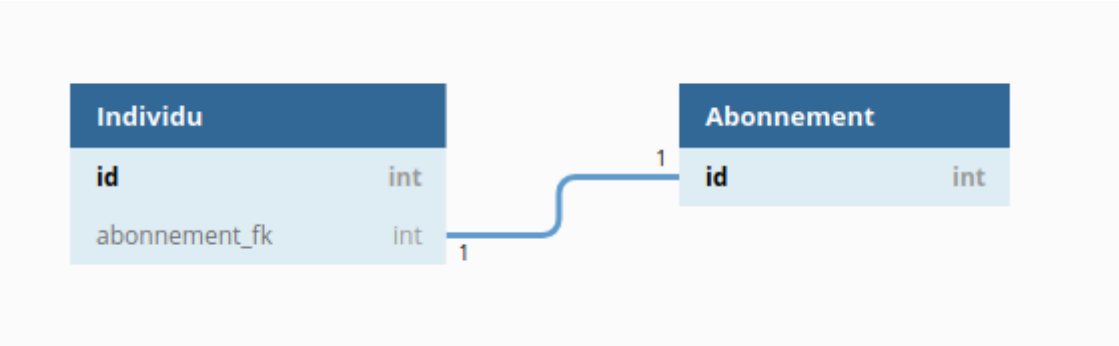
L'annotation @OneToOne implique que la table `Individu` contient une colonne qui est une clé étrangère contenant la clé d'un abonnement. Par défaut, JPA s'attend à ce que cette colonne se nomme `ABONNEMENT_ID`, mais il est possible de changer ce nom grâce à l'annotation @JoinColumn :

Déclaration de la clé étrangère

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
```

```
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;

@Entity
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @OneToOne
  @JoinColumn(name = "abonnement_fk")
  private Abonnement abonnement;

  // getters et setters omis ...
}
```

La déclaration de *mapping* ci-dessus correspond au modèle de base de données suivant :



Le modèle de base de données correspondant à la relation @OneToOne

Important

Dans une requête JPQL, une relation @OneToOne est navigable directement. Si on désire exprimer un requête de la forme

> Sélectionner l'individu avec l'abonnement dont l'id est 1.

alors il suffit d'écrire en JPQL :

> **select i from Individu i where i.abonnement.id = 1**

La relation n:1 (many to one)

L'annotation @ManyToOne définit une relation n:1 entre deux entités.

Exemple de relation ManyToOne

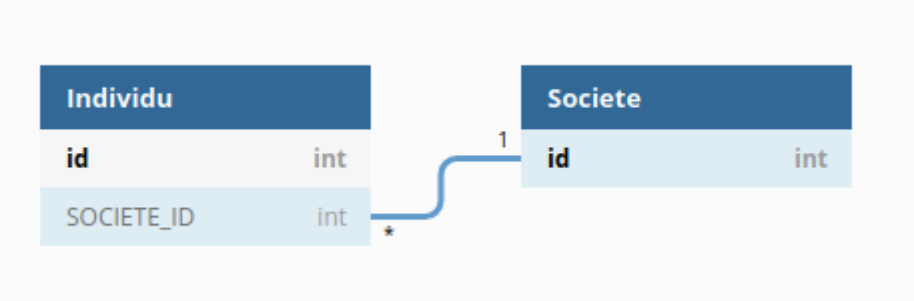
```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

@Entity
public class Individu {
```

```
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @ManyToOne
  private Societe societe;

  // getters et setters omis ...
}
```

La classe associée dans la relation ManyToOne

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Societe {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  // getters et setters omis ...
}
```

L'annotation @ManyToOne implique que la table `Individu` contient une colonne qui est une clé étrangère contenant la clé d'une société. Par défaut, JPA s'attend à ce que cette colonne se nomme `SOCIETE_ID`, mais il est possible de changer ce nom grâce à l'annotation @JoinColumn.



Le modèle de base de données correspondant à la relation @ManyToOne

Plutôt que par une colonne, il est également possible d'indiquer à JPA qu'il doit passer par une table d'association pour établir la relation entre les deux entités avec l'annotation @JoinTable :

Déclaration d'une table d'association

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToOne;
```

```
@Entity
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @ManyToOne
  // déclaration d'une table d'association
  @JoinTable(name = "societe_individu",
             joinColumns = @JoinColumn(name = "individu_id"),
             inverseJoinColumns = @JoinColumn(name = "societe_id"))
  private Societe societe;

  // getters et setters omis ...
}
```

Le modèle de base de données correspondant à la relation @ManyToOne avec une table de jointure

Important

Dans une requête JPQL, une relation @ManyToOne est navigable directement. Si on désire exprimer un requête de la forme

> Sélectionner les individus appartenant à la société avec l'id 1.

alors il suffit d'écrire en JPQL :

> **select i from Individu i where i.societe.id = 1**

La relation 1:n (one to many)

L'annotation @OneToMany définit une relation 1:n entre deux entités. Cette annotation ne peut être utilisée qu'avec une collection d'éléments puisqu'elle implique qu'il peut y avoir plusieurs entités associées.

Exemple de relation OneToMany

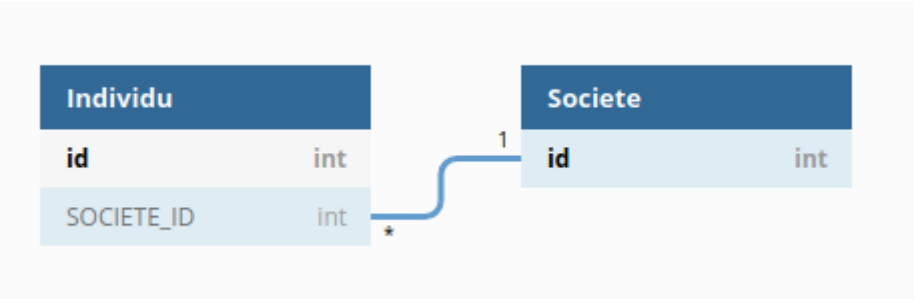
```
import java.util.ArrayList;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
```

```
@Entity
public class Societe {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @OneToMany
  private List<Individu> employes = new ArrayList<>();

  // getters et setters omis ...
}
```

La classe associée dans la relation OneToMany

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  // getters et setters omis ...
}
```

L'annotation @OneToMany implique que la table `Individu` contient une colonne qui est une clé étrangère contenant la clé d'une société. Par défaut, JPA s'attend à ce que cette colonne se nomme `SOCIETE_ID`, mais il est possible de changer ce nom grâce à l'annotation @JoinColumn.



Le modèle de base de données correspondant à la relation @OneToMany

Il est également possible d'indiquer à JPA qu'il doit passer par une table d'association pour établir la relation entre les deux entités avec l'annotation @JoinTable.

Exemple de relation OneToMany avec une table de jointure

```
import java.util.ArrayList;
import java.util.List;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
```

```
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;

@Entity
public class Societe {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @OneToMany
  @JoinTable(name = "societe_individu",
             joinColumns = @JoinColumn(name = "societe_id"),
             inverseJoinColumns = @JoinColumn(name = "individu_id"))
  private List<Individu> employes = new ArrayList<>();

  // getters et setters omis ...
}
```

Le modèle de base de données correspondant à la relation @OneToMany avec une table de jointure

La relation @OneToMany est la relation miroir de la relation @ManyToOne. Le modèle relationnel de base de données est le même. Par contre, du point de vue du modèle objet, cela dépend du *côté* de la relation que l'on veut représenter. On parle de *navigabilité* de la relation. Dans l'exemple ci-dessus, le modèle objet est navigable d'une société vers ses individus.

Important

Dans une requête JPQL, une relation @OneToMany **n'est pas navigable directement**. Si on désire exprimer un requête de la forme

> Sélectionner la société dont un des employés a l'id 1.

alors il faut utiliser en JPQL la syntaxe du fetch pour faire apparaître une variable intermédiaire dans la requête :

> **select s from Societe s join s.employes e where e.id = 1**

La relation n:n (many to many)

L'annotation @ManyToMany définit une relation n:n entre deux entités. Cette annotation ne peut être utilisée qu'avec une collection d'éléments puisqu'elle implique qu'il peut y avoir plusieurs entités associées.

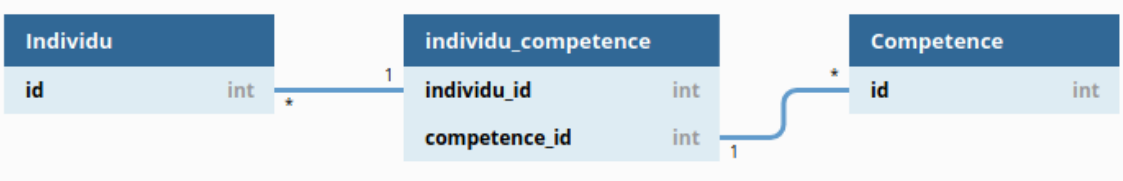
Exemple de relation ManyToMany

```
import java.util.List;
import java.util.ArrayList;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;

@Entity
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @ManyToMany
  @JoinTable(name = "individu_competence",
             joinColumns = @JoinColumn(name = "individu_id"),
             inverseJoinColumns = @JoinColumn(name = "competence_id"))
  private List<Competence> competences = new ArrayList<>();

  // getters et setters omis ...
}
```

La classe associée dans la relation ManyToMany

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Competence {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  // getters et setters omis ...
}
```

L'annotation @ManyToMany implique qu'il existe une table d'association. On utilise l'annotation @JoinTable pour préciser le nom de la table d'association et le nom des colonnes contenant la clé pour l'individu et la clé pour la compétence.



Le modèle de base de données correspondant à la relation @ManyToMany

Important

Dans une requête JPQL, une relation @ManyToMany **n'est pas navigable directement**. Si on désire exprimer un requête de la forme

> Sélectionner les individus dont une des competences à l'id 1.

alors il faut utiliser en JPQL la syntaxe du fetch pour faire apparaître une variable intermédiaire dans la requête :

> **select i from Individu i join i.competences c where c.id = 1**

Les relations bi-directionnelles

Parfois, il est nécessaire de construire un lien entre deux objets qui soit navigables dans les deux sens. D'un point de vue objet, cela signifie que chaque objet dispose d'un attribut pointant sur l'autre instance. Mais d'un point de vue du schéma de base de données relationnelle, il s'agit du même lien. Avec JPA, il est possible de qualifier une relation entre deux objets comme étant une relation inverse grâce à l'attribut `mappedBy` de l'annotation indiquant le type de relation.

Exemple de relation ManyToMany

```
import java.util.List;
import java.util.ArrayList;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;

@Entity
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @ManyToMany
  @JoinTable(name = "individu_competence",
             joinColumns = @JoinColumn(name = "individu_id"),
             inverseJoinColumns = @JoinColumn(name = "competence_id"))
  private List<Competence> competences = new ArrayList<>();

  // getters et setters omis ...
}
```

Exemple d'une relation bi-directionnelle avec `mappedBy`

```
import java.util.List;
import java.util.ArrayList;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
```

```
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Competence {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @ManyToMany(mappedBy = "competences")
  private List<Individu> individus = new ArrayList<>();

  // getters et setters omis ...
}
```

Dans l'exemple précédent, l'attribut `individus` dans la classe `Competence` correspond à la même relation que celle décrite par l'attribut `competences` dans la classe `Individu`. Ces deux attributs traduisent la même relation dans le modèle relationnel de base de données. Il faut donc indiquer à JPA qu'il existe un lien logique entre ces deux attributs. L'attribut `mappedBy` de l'annotation @ManyToMany permet d'indiquer le nom de l'attribut qui décrit la même relation dans l'autre entité. Dans notre exemple, on déclare dans la classe `Competence` que l'attribut `individus` correspond à l'attribut `competences` de la classe `Individu`.

Vous n'avez aucune obligation à déclarer une relation bi-directionnelle et, si vous le faites, il n'existe pas de règle pour décider lequel des deux attributs doit être déclaré `mappedBy`.

Par contre, vous **ne devez pas** utiliser `mappedBy` dans la déclaration des deux attributs mais uniquement pour celui qui est en quelque sorte le sens secondaire de la relation. L'attribut `mappedBy` est une façon de dire à JPA que l'attribut ne représente pas le sens privilégié de la relation et que, s'il désire obtenir des informations sur cette relation, il doit regarder la déclaration de l'autre attribut. Il s'agit d'un moyen d'éviter de la duplication d'informations. En effet, les annotations comme @JoinTable et @JoinColumn ne doivent être ajoutées qu'à l'attribut qui n'a pas la déclaration `mappedBy`.

Les relations bi-directionnelles ne sont pas limitées aux relations @ManyToMany. À une relation @ManyToOne peut correspondre une relation @OneToMany et à une relation @OneToOne peut correspondre une relation @OneToOne.

La propagation en cascade

Avec JPA, il est possible de propager des modifications à tout ou partie des entités liées. Les annotations permettant de spécifier une relation possèdent un attribut `cascade` permettant de spécifier les opérations concernées : `ALL`, `DETACH`, `MERGE`, `PERSIST`, `REFRESH` ou `REMOVE`.

Exemple d'opérations en cascade

```
import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
```

```
@Entity
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @ManyToOne(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
  private Societe societe;

  // getters et setters omis ...
}
```

Dans l'exemple précédent, on précise que l'instance de société doit être enregistrée en base si nécessaire au moment où l'instance d'Individu sera elle-même enregistrée. De plus, lors d'un appel à merge pour une instance d'Individu, un merge sera automatiquement realisé sur l'instance de l'attribut société.

Requêtes JPQL et jointure

Dès que l'on introduit des relations entre entités, on complexifie également le langage de requêtage. Comment par exemple demander la liste des individus travaillant pour une société dont on passe le nom en paramètre ? On peut utiliser une jointure avec une condition :

```
select i from Individu i join i.societe s where s.nom = :nom
```

Il est également possible d'utiliser le mot-clé `on` pour filtrer les entités à sélectionner dans la jointure :

```
select i from Individu i join i.societe s on s.nom = :nom
```

Comme en SQL, JPQL fait la différence entre une jointure et une jointure externe (*left outer join*). Avec une jointure simple, on élimine toutes les entités pour lesquelles la jointure n'existe pas. Alors qu'avec une jointure externe, nous préservons les entités pour lesquelles il n'existe pas de jointure.

```
select i from Individu i left join i.societe s on s.nom = :nom
```

Attention avec l'exemple de requête ci-dessus : elle retourne l'union des individus qui n'ont aucune relation avec l'entité société et ceux qui ont une relation avec une société dont le nom est donné par le paramètre `:nom`.

Fetch lazy ou fetch eager

Lorsque JPA doit charger une entité depuis la base de données (qu'il s'agisse d'un appel à `EntityManager.find(...)` ou d'une requête), la question est de savoir quelles informations doivent être chargées. Doit-il charger tous les attributs d'une entité ? Parmi ces attributs, doit-il charger les entités qui sont en relation avec l'entité chargée ? Ces questions sont importantes, car la façon d'y répondre peut avoir un impact sur les performances de l'application.

Dans JPA, l'opération de chargement d'une entité depuis la base de données est appelée **fetch**. Un fetch peut avoir deux stratégies : **eager** ou **lazy** (que l'on peut traduire respectivement et approximativement par *chargement immédiat* et *chargement différé*). On peut décider de la stratégie pour chaque membre de la classe grâce à l'attribut `fetch` présent sur les annotations @Basic, @OneToOne, @ManyToOne, @OneToMany et @ManyToMany.

- **eager** signifie que l'information doit être chargée systématiquement lorsque l'entité est chargée. Cette stratégie est appliquée par défaut pour @Basic, @OneToOne et @ManyToOne.
- **lazy** signifie que l'information ne sera chargée qu'à la demande (par exemple lorsque la méthode `get` de l'attribut sera appelée). Cette stratégie est appliquée par défaut pour @OneToMany et @ManyToMany.

Exemple d'utilisation de fetch lazy

```
import javax.persistence.Basic;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Lob;
import javax.persistence.ManyToOne;

@Entity
public class Individu {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  @ManyToOne(fetch = FetchType.LAZY)
  private Societe societe;

  /*
   * Stocke la photo d'identité sous une forme binaire. Comme l'information peut
être
   * volumineuse, on déclare un fetch lazy pour ne déclencher le chargement qu'à
l'appel
   * de getPhoto(), c'est-à-dire quand l'application en a vraiment besoin.
   */
  @Lob
  @Basic(fetch = FetchType.LAZY)
  private byte[] photo;

  // getters et setters omis ...
}
// Exécute une requête de la forme : SELECT i.id FROM Individu i WHERE i.id = ?
// Un appel à la méthode find ne charge pas les attributs marqués avec un fetch
lazy.
Individu persistedIndividu = entityManager.find(Individu.class, individuId);

// Exécute une requête de la forme : SELECT i.photo FROM Individu i WHERE i.id =
?
```

```
byte[] photo = persistedIndividu.getPhoto();

// Exécute une requête de la forme :  SELECT * FROM Societe WHERE id = ?
Societe societe = persistedIndividu.getSociete();
```

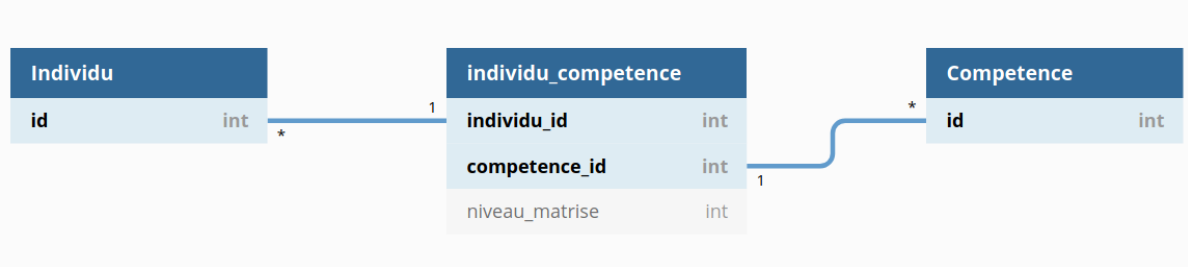
Il est possible d'utiliser une requête JPQL pour forcer la récupération d'informations en mode *lazy* si on sait qu'elle seront nécessaires plus tard dans le programme. Cela limite le nombre de requêtes à exécuter et peut se révéler nécessaire si la consultation des attributs *lazy* doit se faire à un moment où un *entity manager* n'est plus disponible. On utilise dans la requête l'option JOIN FETCH :

```
// Exécute une requête de type JOIN FETCH
String query = "select i from Individu i left join fetch i.societe where i.id =
:id";
Individu persistedIndividu = entityManager.createQuery(query, Individu.class)
                                          .setParameter("id", individuId)
                                          .getSingleResult();

// N'exécute aucune requête car la société a déjà été récupérée par le left join
fetch
Societe societe = persistedIndividu.getSociete();
```

La définition des stratégies de fetch est une partie importante du tuning dans le développement d'une application utilisant JPA.

Relation avec des attributs

Dans un modèle relationnel, il est courant que des tables d'association contiennent des colonnes pour caractériser une relation. Reprenons et complétons l'exemple de relation n:n présenté plus haut : un individu peut être associé à une ou plusieurs compétences et, pour chacune de ces compétences, il dispose d'un niveau de maîtrise (disons entre 0 et 5). Nous pouvons très facilement représenter cette relation avec le schéma suivant :



Le modèle de base de données avec un attribut sur la relation entre Individu et Competence

Cependant, dans le modèle objet, une relation entre deux objets ne peut pas avoir d'attribut. Pour ce type de *mapping*, nous avons une différence importante entre le modèle relationnel et le modèle objet. Si nous voulons traduire la colonne `niveau_maitrise` en Java, nous devons créer une entité pour représenter la table `individu_competence`. Ainsi, ce modèle ne peut pas être traduit comme une relation n:n entre deux entités `Individu` et `Competence` mais comme deux relations 1:n :

- une relation 1:N entre `IndividuCompetence` et `Individu`
- une relation 1:N entre `IndividuCompetence` et `Competence`

La classe `Competence` reste identique à l'exemple vu précédemment :

La classe Competence

```
import javax.persistence.Entity; import javax.persistence.GeneratedValue; import javax.persistence.GenerationType; import javax.persistence.Id; @Entity public class Competence { @Id @GeneratedValue(strategy = GenerationType.IDENTITY) private Long id; // getters et setters omis ... }
```

Pour réaliser le reste du *mapping*, nous avons le choix entre trois solutions. En effet, il existe plusieurs *mappings* pour représenter la clé composée de la table `individu_competence`.

Solution 1 : Référencer la clé composée avec @IdClass

La classe IndividuCompetence

```
import javax.persistence.Column; import javax.persistence.Entity; import javax.persistence.Id; import javax.persistence.IdClass; import javax.persistence.JoinColumn; import javax.persistence.ManyToOne; import javax.persistence.Table; @Entity @Table(name = "individu_competence") @IdClass(IndividuCompetencePK.class) public class IndividuCompetence { @Id @Column(name = "individu_id") private Long individuId; @Id @Column(name = "competence_id") private Long competenceId; @Column(name = "niveau_maitrise") private int niveauMaitrise; @ManyToOne @JoinColumn(name = "individu_id", insertable = false, updatable = false) private Individu individu; @ManyToOne @JoinColumn(name = "competence_id", insertable = false, updatable = false) private Competence competence; public int getNiveauMaitrise() { return this.niveauMaitrise; } public void setNiveauMaitrise(int niveauMaitrise) { this.niveauMaitrise = niveauMaitrise; } public Individu getIndividu() { return this.individu; } public void setIndividu(Individu individu) { this.individu = individu; this.individuId = individu.getId(); } public Competence getCompetence() { return this.competence; } public void setCompetence(Competence competence) { this.competence = competence; this.competenceId = competence.getId(); } }
```

La classe `IndividuCompetence` est l'entité qui représente la table `individu_competence`, elle déclare deux relations @ManyToOne : à la ligne 25 pour la relation avec l'entité `Individu` et à la ligne 29 pour la relation avec l'entité `Competence`. Elle dispose d'un attribut `niveauMaitrise` afin de fournir l'information sur la relation.

L'inconvénient est que cette entité dispose d'une clé primaire composée des deux clés étrangères de l'association (`individu_id` et `competence_id`). Nous sommes donc obligés de déclarer deux attributs avec l'annotation @Id pour identifier les deux attributs qui forment la clé composée (lignes 14 et 18). Remarquez que nous n'utilisons pas l'annotation @GeneratedValue pour ces attributs. En effet, la clé ne doit pas être générée puisqu'elle est la composition de deux autres clés déjà existantes : celle de `Individu` et celle de `Competence`.

Notre *mapping* associe deux fois les colonnes `individu_id` et `competence_id` à des attributs de la classe puisque ces deux colonnes sont à la fois des composantes de la clé primaire et des clés étrangères. Nous sommes donc obligés d'indiquer à JPA qu'il ne doit pas tenir compte de ces colonnes lorsqu'il doit faire des insertions et des mises à jour pour les colonnes de jointure. C'est la signification des attributs `insertable = false` et `updatable = false` pour les annotations @JoinColumn aux lignes 26 et 30.

Plutôt que de fournir des getters/setters aux attributs `individuId` et `competenceId`, nous préférons les mettre à jour lorsqu'on invoque les méthodes `setIndividu` (ligne 47) et `setCompetence` (ligne 56). En effet, ces identifiants sont en fait dupliqués puisqu'ils doivent être identiques à la valeur des identifiants des objets de type `Individu` et `Competence`.

Une entité JPA doit forcément disposer d'une classe qui représente sa clé. Jusqu'à présent, nous avons toujours déclaré des clés de type Long, une classe déjà fournie par la bibliothèque standard Java. Dans le cas de l'entité `IndividuCompetence`, nous avons déclaré deux clés dans la classe. Il faut donc fournir une classe supplémentaire pour représenter la composition de ces deux clés. Grâce à l'annotation @IdClass, nous indiquons à la ligne 11 le type de la classe Java qui représente la clé composée pour cette entité.

La classe IndividuCompetencePK

```
1 2 3
4 5 6 7
8 9 10
11 12
13 14
15 16
17 18
19 20
21 22
23 24
25 26
27 28
29 30
31 32
33 34
35 36
37 38
```

```
import java.io.Serializable; import java.util.Objects; public class IndividuCompetencePK implements Serializable { private Long individuId; private Long competenceId; public Long getIndividuId() { return this.individuId; } public void setIndividuId(Long individuId) { this.individuId = individuId; } public Long getCompetenceId() { return this.competenceId; } public void setCompetenceId(Long competenceId) { this.competenceId = competenceId; } public boolean equals(Object other) { if (other instanceof IndividuCompetencePK) { IndividuCompetencePK pk = (IndividuCompetencePK) other; return Objects.equals(this.individuId, pk.individuId) && Objects.equals(this.competenceId, pk.competenceId); } return false; } public int hashCode() { return Objects.hash(individuId, competenceId); } }
```

La classe `IndividuCompetencePK` représente la clé de la classe `IndividuCompetence`. Son implémentation doit respecter les contraintes suivantes :

- elle doit implémenter l'interface Serializable (ligne 4),
- elle doit déclarer tous les attributs qui composent la clé primaire (lignes 6 et 8),
- elle doit fournir une implémentation pour les méthodes equals (ligne 26) et hashCode (ligne 35).

La classe Individu

```
1 2
3 4
5 6
7 8
9 10
11
12
13
14
15
16
17
18
19
20
21
22
```

```
import java.util.ArrayList; import java.util.List; import
javax.persistence.CascadeType; import javax.persistence.Entity;
import javax.persistence.GeneratedValue; import
javax.persistence.GenerationType; import javax.persistence.Id; import
javax.persistence.OneToMany; @Entity public class Individu { @Id
@GeneratedValue(strategy=GenerationType.IDENTITY) private Long id;
@OneToMany(mappedBy="individu", cascade = CascadeType.ALL) private
List<IndividuCompetence> individuCompetences = new ArrayList<>(); //
getters et setters omis ... }
```

Pour notre exemple, il paraît utile qu'un objet de type `Individu` possède une liste de `IndividuCompetence` pour permettre au programme de connaître le niveau de compétence d'un individu donné. La relation entre `Individu` et `IndividuCompetence` est donc bi-directionnelle (ligne 18).

Notez que la relation avec la liste `IndividuCompetence` indique une propagation en cascade pour toutes les opérations. En effet, d'un point de vue objet, cette relation est très certainement une relation de composition (aussi appelée agrégation composite en UML) et dénote un lien très fort entre ces entités.

Solution 2 : Référencer la clé composée avec @EmbeddedId

L'implémentation précédente a un inconvénient : elle oblige à dupliquer les attributs représentant la clé composée de l'entité `IndividuCompetence` dans la classe `IndividuCompetencePK` qui représente la clé. Il est possible d'utiliser la notion de clé embarquée pour éviter cette duplication. Le code Java devient alors :

La classe IndividuCompetencePK

```
import java.io.Serializable; import java.util.Objects; import javax.persistence.Column; import javax.persistence.Embeddable; @Embeddable public class IndividuCompetencePK implements Serializable { @Column(name = "individu_id") private Long individuId; @Column(name = "competence_id") private Long competenceId; public Long getIndividuId() { return this.individuId; } public void setIndividuId(Long individuId) { this.individuId = individuId; } public Long getCompetenceId() { return this.competenceId; } public void setCompetenceId(Long competenceId) { this.competenceId = competenceId; } public boolean equals(Object other) { if (other instanceof IndividuCompetencePK) { IndividuCompetencePK pk = (IndividuCompetencePK) other; return Objects.equals(this.individuId, pk.individuId) && Objects.equals(this.competenceId, pk.competenceId); } return false; } public int hashCode() { return Objects.hash(individuId, competenceId); } }
```

La classe `IndividuCompetencePK` utilise l'annotation @Embeddable (ligne 9) pour indiquer qu'elle peut être utilisée dans une entité pour fournir une partie du *mapping*. Ainsi, elle déclare le *mapping* pour les attributs `individuId` et `competenceId`.

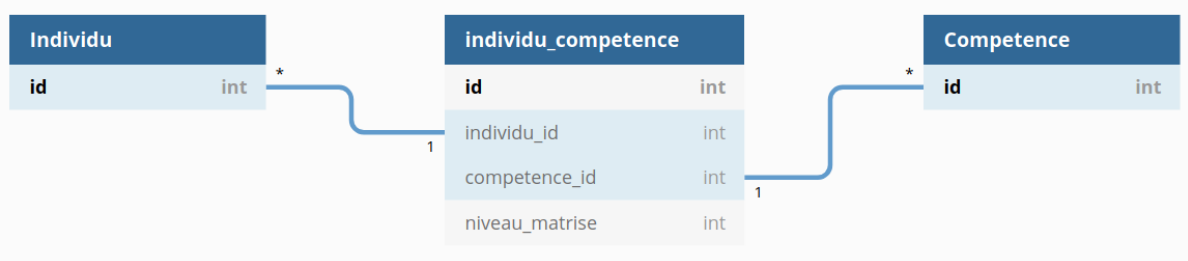
La classe IndividuCompetence

```
import javax.persistence.Column; import javax.persistence.EmbeddedId; import javax.persistence.Entity; import javax.persistence.JoinColumn; import javax.persistence.ManyToOne; import javax.persistence.Table; @Entity @Table(name = "individu_competence") public class IndividuCompetence { @EmbeddedId private IndividuCompetencePK id = new IndividuCompetencePK(); @Column(name = "niveau_maitrise") private int niveauMaitrise; @ManyToOne @JoinColumn(name = "individu_id", insertable = false, updatable = false) private Individu individu; @ManyToOne @JoinColumn(name = "competence_id", insertable = false, updatable = false) private Competence competence; public int getNiveauMaitrise() { return this.niveauMaitrise; } public void setNiveauMaitrise(int niveauMaitrise) { this.niveauMaitrise = niveauMaitrise; } public Individu getIndividu() { return this.individu; } public void setIndividu(Individu individu) { this.individu = individu; this.id.setIndividuId(individu.getId()); } public Competence getCompetence() { return this.competence; } public void setCompetence(Competence competence) { this.competence = competence; this.id.setCompetenceId(competence.getId()); } }
```

La classe `IndividuCompetence` n'utilise plus l'annotation @IdClass et ne déclare plus les différents attributs composant la clé. À la place, elle déclare un attribut de type `IndividuCompetencePK` avec l'annotation @EmbeddedId (ligne 12) pour signaler à JPA que cet attribut fournit le *mapping* de la clé composée. Le reste du code de la classe est adapté en fonction.

Solution 3 : Utiliser une clé technique simple

Il est important de comprendre que la complexité du *mapping* d'une table d'association en Java vient du fait qu'un modèle relationnel et un modèle objet ne se recoupent qu'imparfaitement. Dans les deux solutions vues précédemment, nous adaptons le modèle objet pour qu'il corresponde au modèle relationnel. Mais nous pouvons tout aussi bien adapter le modèle relationnel pour correspondre au modèle objet. Pour cela, il suffit de dénormaliser le modèle relationnel en ajoutant une colonne pour la clé primaire dans la table `individu_competence`.



Le modèle de base de données dénormalisé

Dans ce cas, nous n'avons plus besoin d'une classe spécifique pour représenter la clé et nous pouvons nous contenter de l'implémentation suivante pour la classe `IndividuCompetence` :

La classe IndividuCompetence

```
1 2
3 4 5
6 7 8
9 10
11 12
13 14
15 16
17 18
19 20
21 22
23 24
25 26
27
28
```

```
import javax.persistence.Column; import javax.persistence.Entity;
import javax.persistence.JoinColumn; import javax.persistence.Id;
import javax.persistence.ManyToOne; import javax.persistence.Table;
@Entity @Table(name = "individu_competence") public class
IndividuCompetence { @Id
@GeneratedValue(strategy=GenerationType.IDENTITY) private Long id;
@Column(name = "niveau_maitrise") private int niveauMaitrise;
@ManyToOne @JoinColumn(name = "individu_id") private Individu
individu; @ManyToOne @JoinColumn(name = "competence_id") private
Competence competence; // getters et setters omis ... }
```

L'héritage avec JPA

JPA offre plusieurs stratégies pour associer une relation d'héritage dans le modèle objet avec une représentation dans le modèle relationnel de données.

Pour décrire la relation d'héritage avec JPA, on utilise l'annotation @Inheritance ou @MappedSuperclass.

Note

Pour une présentation complète de l'héritage dans JPA, reportez-vous au Wikibook.

L'annotation @Inheritance

L'annotation @Inheritance permet d'indiquer qu'il existe une relation d'héritage dans le modèle objet et dans le modèle relationnel. Comme il existe plusieurs façon de représenter un héritage de données dans le modèle relationnel, l'annotation @Inheritance dispose de l'attribut `strategy` pour préciser la stratégie utilisée dans le modèle de données. Cette stratégie est une énumération du type InheritanceType et accepte les valeurs :

- `SINGLE_TABLE`

  l'héritage est représenté par une seule table en base de données

- `JOINED`

  l'héritage est représenté par une jointure entre la table de l'entité parente et la table de l'entité enfant

- `TABLE_PER_CLASS`

  l'héritage est représenté par une table par entité

Héritage avec une seule table

La stratégie `SINGLE_TABLE` permet de représenter en base de données un héritage avec une seule table. Une colonne contiendra un identifiant pour déterminer le type réel de la classe : on parle de la colonne *discriminante*. La table doit contenir toutes les colonnes nécessaires pour stocker les informations de la super classe et de l'ensemble des classes filles.

Note

L'inconvénient de la stratégie `SINGLE_TABLE` est qu'il n'est pas possible d'ajouter des contraintes de type `NOT NULL` sur les colonnes représentant les propriétés des classes filles.

Super classe

```
package dev.gayerie.vehicule;

import javax.persistence.DiscriminatorColumn;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="vehicule_type")
public abstract class Vehicule {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String marque;

    // getters/setters omis

}
```

Dans l'exemple précédent, la colonne discriminante est donnée par l'annotation @DiscriminatorColumn. Cette colonne n'est donc pas représentée par un attribut dans la classe.

Classe enfant

```
package dev.gayerie.vehicule;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue("V")
public class Voiture extends Vehicule {

    private int nbPlaces;
```

```
    // getters/setters omis

}
```

Une autre classe enfant

```
package dev.gayerie.vehicule;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue("M")
public class Moto extends Vehicule {

    private int cylindree;

    // getters/setters omis

}
```

L'annotation @DiscriminatorValue permet de préciser la valeur dans la colonne discriminante qui permet d'identifier un objet du type de cette classe. Cette valeur doit être unique pour l'ensemble des classes de l'héritage. Dans l'exemple ci-dessus, lors de la persistance d'un objet de la classe `Voiture`, JPA positionnera automatiquement la valeur `"V"` dans la colonne `vehicule_type`.

Pour l'exemple précédent, le schéma de base de données contiendra la table `Vehicule` qui peut être créée comme suit :

```
create table Vehicule (
    id int(11) AUTO_INCREMENT,
    vehicule_type varchar(1) NOT NULL,
    marque varchar(255),
    cylindree int,
    nbPlaces int,
    primary key (id)
) engine = InnoDB;
```

Héritage avec jointure de tables

La stratégie `JOINED` permet de représenter en base de données un héritage avec une table par entité. Pour les classes filles, JPA réalisera une jointure entre la table représentant l'entité et la table représentant l'entité de la super classe. L'implémentation est très proche de celle d'un héritage avec une seule table (seule la stratégie change) mais le schéma de la base de données est très différent.

Super classe

```
package dev.gayerie.vehicule;

import javax.persistence.DiscriminatorColumn;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
```

```
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminatorColumn(name="vehicule_type")
public abstract class Vehicule {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;

    private String marque;

    // getters/setters omis

}
```

Classe enfant

```
package dev.gayerie.vehicule;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue("V")
public class Voiture extends Vehicule {

    private int nbPlaces;

    // getters/setters omis

}
```

Une autre classe enfant

```
package dev.gayerie.vehicule;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue("M")
public class Moto extends Vehicule {

    private int cylindree;

    // getters/setters omis

}
```

Dans cette configuration, JPA attend 3 tables dans le schéma de base de données qui peuvent être créées par les instructions suivantes :

```
create table Vehicule (
    id int(11) AUTO_INCREMENT,
    vehicule_type varchar(1) NOT NULL,
    marque varchar(255),
    primary key (id)
) engine = InnoDB;

create table Voiture (
    id int(11) AUTO_INCREMENT,
    nbPlaces int,
    primary key (id),
    foreign key (id) references Vehicule(id)
) engine = InnoDB;

create table Moto (
    id int(11) AUTO_INCREMENT,
    cylindree int,
    primary key (id),
    foreign key (id) references Vehicule(id)
) engine = InnoDB;
```

Héritage avec une table par classe

La stratégie `TABLE_PER_CLASS` permet de représenter en base de données un héritage avec une table par entité. Les attributs hérités sont répétés dans chaque table. Ainsi, la notion d'héritage n'est pas exprimée dans le modèle relationnel de données.

Super classe

```
package dev.gayerie.vehicule;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Vehicule {

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

    private String marque;

    // getters/setters omis

}
```

Classe enfant

```
package dev.gayerie.vehicule;

import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name="Voiture")
public class Voiture extends Vehicule {

    private int nbPlaces;

    // getters/setters omis

}
```

Une autre classe enfant

```
package dev.gayerie.vehicule;

import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name="Moto")
public class Moto extends Vehicule {

    private int cylindree;

    // getters/setters omis

}
```

Dans cette configuration, JPA attend 2 tables dans le schéma de base de données qui peuvent être créées par les instructions suivantes :

```
create table Voiture (
    id int not null,
    marque varchar(255),
    nbPlaces int not null,
    primary key (id)
) engine = InnoDB;

create table Moto (
    id int not null,
    marque varchar(255),
    cylindree int not null,
    primary key (id)
) engine = InnoDB;
```

Note

La table `Vehicule` n'est pas nécessaire car la classe `Vehicule` est abstraite. Il n'est donc pas possible de créer des entités de type `Vehicule`.

Note

La stratégie `TABLE_PER_CLASS` est plus complexe à mettre en place pour la gestion des clés primaires. En effet, du point de vue de JPA, les objets de type `Voiture` et `Moto` sont également des objets de type `Voiture`. À ce titre, il ne peut pas exister deux objets avec la même clé primaire. Mais, comme ces objets sont représentés en base de données par deux tables différentes, une colonne de type `AUTO_INCREMENT` en MySQL ne suffit pas à garantir qu'il n'existe pas une voiture ayant la même clé qu'une moto.

Avec, la stratégie `TABLE_PER_CLASS`, il n'est pas possible d'utiliser l'annotation @GeneratedValue avec la valeur `IDENTITY`. On peut, par exemple, utiliser une table servant à générer une séquence de clés.

L'héritage et les requêtes JPQL polymorphiques

Lorsqu'il existe une relation d'héritage entre les entités, il est possible de créer des requêtes polymorphiques. Si on exécute la requête JPQL suivante :

```
select v from Vehicule v
```

Alors quelle que soit la stratégie d'héritage utilisée, la requête JPQL doit remonter l'ensemble des objets qui héritent de la classe `Vehicule`. Pour notre exemple, cette requête retourne la liste de tous les objets `Voiture` et `Moto`.

Néanmoins il est possible d'avoir un type plus précis si nécessaire. Pour retourner uniquement la liste des objets `Voiture` :

```
select v from Voiture v
```

Un cas à part : fusion de la super classe

Il arrive parfois que la relation d'héritage n'ait pas de sens dans le modèle relationnel. Dans ce cas, la classe parente n'est pas vraiment une entité au sens JPA, on parle de *mapped superclass*.

Super classe

```
package dev.gayerie.vehicule;

import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.MappedSuperclass;

@MappedSuperclass
public abstract class Vehicule {

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Long id;
```

```
  private String marque;

  // getters/setters omis

}
```

La classe `Vehicule` n'est plus déclarée avec l'annotation @Entity mais avec l'annotation @MappedSuperclass.

Classe enfant

```
package dev.gayerie.vehicule;

import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name="Voiture")
public class Voiture extends Vehicule {

  private int nbPlaces;

  // getters/setters omis

}
```

Une autre classe enfant

```
package dev.gayerie.vehicule;

import javax.persistence.Entity;
import javax.persistence.Table;

@Entity
@Table(name="Moto")
public class Moto extends Vehicule {

  private int cylindree;

  // getters/setters omis

}
```

Dans cette configuration, JPA attend 2 tables dans le schéma de base de données qui peuvent être créées par les instructions suivantes :

```
create table Voiture (
    id int not null auto_increment,
    marque varchar(255),
    nbPlaces int not null,
    primary key (id)
) engine = InnoDB;

create table Moto (
    id int not null auto_increment,
    marque varchar(255),
    cylindree int not null,
    primary key (id)
) engine = InnoDB;
```

L'utilisation de @MappedSuperclass implique qu'il n'existe pas de relation entre les classes filles pour JPA. Comme la super classe n'est pas un entité, il n'est pas possible d'effectuer des requêtes sur la super classe ni d'utiliser des requêtes polymorphiques.