Database Applications with JDBC in Java SE Applications

(Java SE 11 Developer Certification 1Z0-819)

Table des matières

I. *Introduction to relational database :*

Pour commencer, on peut séparer les bases de données en deux grosses catégories :

Sql (or relational)

No Sql
Data is not modeled using relationships

A. **Relationnelle** (SQL)

B. **Non-relationnelle** (plus souvent, un document au format json).

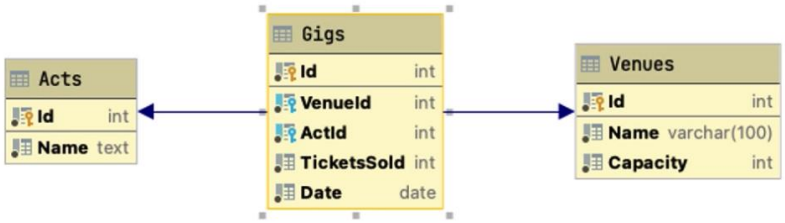
Une base de données relationnelle :

C'est une est une base de données dont, les données sont organisées dans des tables qui ont des lignes et des colonnes. ***Ces tables sont reliées entre-elles par le biais de clef.***

On a ici un exemple de trois tables qui représentent :

- Un concert (*gig*)
- Une salle (*venue*)
- Un acte (*acts*).

Ces trois tables possèdent chacune une id unique représenté par le nom ID et la petite clef. C'est la table *gigs* qui est liée au deux autres grâce à une clef étrangère (*foreign key*) représentée par une petite clef bleue.



Pour accéder à ces données on utilise le langage **SQL** (*structured query language*). Il permet 4 opérations de bases qu'on appelle les opérations **CRUD** :

- Créer
- Lire
- Mettre à jour
- Supprimer.

Quelques exemples de lecture de données :

1. On sélectionne la table *Gigs* et on lit toutes les lignes de la colonne *TicketSold*.
2. On sélectionne la table *Venues* et on récupère le contenu des colonnes ID et NOM uniquement quand le mot « arena » est dans le nom.
3. On sélectionne toutes les colonnes de la table *Gigs*, on la joint à la table Venues, on fait correspondre l'id de la table *Venues* à celui de *Gigs*, le tout en ayant mis un filtre identique à l'exemple précédant.
4. On demande de compter combien de lignes il y'a dans la table *Gigs* là où on a vendu moins de 30 tickets.

```
SELECT TicketsSold FROM Gigs                          ◂ Simple read


SELECT id, name FROM Venues                           ◂ Filters
WHERE name LIKE '%arena%'


SELECT * FROM gigs                                    ◂ Joins
JOIN venues ON venues.id = gigs.venueid
WHERE venues.name LIKE '%arena%'


SELECT COUNT(*) from Gigs                             ◂ SQL Functions
where TicketsSold < 30
```

Un exemple d'insertion de données. On doit spécifier la table (*Venues*), le nom des colonnes entre parenthèses, et les valeurs à insérer.

```
INSERT INTO                                           ◂ Insert into
venues (name, capacity)                               ◂ Table and columns to add
values ('The Arena', 100);                            ◂ Values to insert
```

Un exemple de mise à jour. On spécifie le nom de la table et dans ce cas ci on attribue la valeur 30 de la colonne *Capacity* quand l'ID vaut 4.

```
UPDATE venues                                         ◂ Update table
SET capacity=30                                       ◂ columns to update
WHERE id=4                                            ◂ Values to change to



UPDATE venues                                         ◂ BEWARE
SET capacity=30
```

Il faut juste faire attention à la deuxième instruction car elle va mettre à jour **TOUTES** les valeurs de la colonne capacity et leur attribuer 30.

Pareil pour la suppression. On sélectionne toutes les colonnes de la table *Venues* dans lesquelles l'Id vaut 5 et on les supprime.

Attention que la deuxième commande va **TOUT** supprimer.

```
DELETE FROM
venues
WHERE Id=5



DELETE FROM
venues
```

◂ DELETE from
◂ table
◂ filter
◂ BEWARE

II. Introduction to JDBC

A. Introduction :

Pour se connecter à la base de données il nous faut ce qu'on appelle un pilote (*driver*) JDBC. Ils seront matérialisés dans des classes Java prévues à cet effet et ils sont fournis par le fournisseur de la base de données. Ils sont spécifiques à un type de base de données (MySql driver, Oracle Drive etc…). Cette classe n'est pas utilisé explicitement, à la place on utilise une Interface (*Driver*).

Les principales interfaces de JDBC :

1. L'interface *Driver*, qui permet de créer une connexion.
2. Pour gérer la connexion on utilise l'interface *Connection*.
3. Pour exécuter des instructions **CRUD** on utilise l'interface *PreparedStatement*.
4. Il se peut que la base de données ait des procédures stockées. Pour les appeler (call) on utilise *CallableStatement*.
5. On pour récupérer les données utilisées dans les deux méthodes précédentes on utilise un *ResultSet*.

B. Demo : Loading the Driver :

Pour créer la connexion à la base de données, une fois qu'elle est créée, on a besoin de trois variables en String :

1. L'url (adresse de la base de données).
2. Le nom d'utilisateur.
3. Le mot de passe.

On créer une variable de type *Connection*, et on appelle la méthode qui comprend plusieurs paramètres : *DriverManager.getConnection*(URL, Username, password).

```
public class Application {

    static String url = "jdbc:mysql://localhost:3306/loboticket";
    static String userName = "loboticket";
    static String password = "p4ssw0rd";

    public static void main(String[] args) throws SQLException {
        try(Connection conn = DriverManager.getConnection(url, userName,
password)) {

            System.out.println(conn);
        }
    }
}
```

Si on fait un *print*, il montrera le nom complet de la classe. On notera qu'on met la méthode dans un **TRY/ressource** qui fermera la connexion automatiquement en dehors des accolades.

C. JDBC Interfaces summary :

Pour résumer :

JDBC Classes

The Driver Manager loads the driver

Driver provides implementations of the JDBC interfaces

These classes know how to talk to this specific database

On charge un driver qui va fournir l'implémentation des interfaces JDBC.

Ces interfaces vont nous permettre d'interagir avec la base de données.

III. Connecting to a Database :

A. Introduction :

On a vu précédemment que pour créer une connexion, il faut donner l'url de la base de données en paramètres *du DrivrerManager.getConnection()*.

Ce qu'il fait, c'est offrir cette url à toutes les classes qu'il contient pour qu'enfin la bonne classe reconnaisse qu'elle doit l'utiliser et elle va ainsi être chargée.

jdbc:mysql://localhost:3306/loboticket

Une url Jdbc est composée de trois parties :

1. Les 4 caractères jdbc qui sont le protocole pour l'url.
2. Le sous-protocole, ou le « fournisseur » du produit. Ici MySql.
3. Le sous nom, ou la chaine String de connexion à la base de données.

```
jdbc:postgresql://localhost/loboticket
jdbc:mysql://localhost:3306/loboticket
jdbc:oracle:thin:@123.123.123.123:5321:loboticket
jdbc:derby:loboticket
```

Figure 1Autres exemple

B. Database Connections :

Database Connections

These are native resources

Must be closed by the application after use

Use 'try with resources' to close connection

This will also close the statement and the result set

Comme on l'a dit précédemment, les connexions doivent être **FERMEE** par l'application après utilisation. Pour cela on utilise un bloc *try/ressource*.

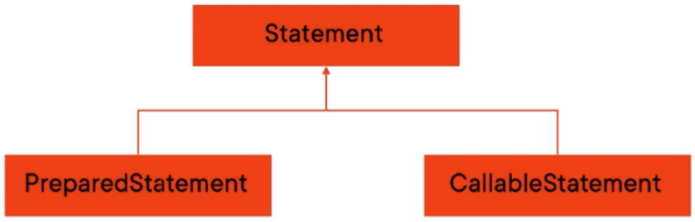
Dans l'exemple ci-dessous, on a le même début que l'exemple du chapitre 2 mais on rajoute une variable *PreparedStatement* et une variable *result* qui va stocker les résultats. Si on veut, on peut aussi mettre le *preparedStatement* dans le try/ressources pour le fermer, cela n'est pas obligatoire dans la mesure où on ferme déjà la connexion avec le *DriverManager* dedans, ce qui nous dispensera de mettre en plus cette ressource.

```
public class Application {

    static String url = "jdbc:mysql://localhost:3306/loboticket";
    static String userName = "loboticket";
    static String password = "p4ssw0rd";

    public static void main(String[] args) throws SQLException {
        try(Connection conn = DriverManager.getConnection(url, userName,
password)) {
            PreparedStatement stmt = conn.prepareStatement("select * from
Acts");
            var results = stmt.executeQuery();

            while(results.next()) {
                System.out.println(results.getString("Name"));
            }
        }
    }

}
```

IV. Using PreparedStatements :

A. Introduction :

JDBC a 2 différentes interfaces d'instruction (*statements*) qui implémentent une interface principale :



Pour la certification l'utilisation de *Statement* n'est pas demandée. A la place on utilisera soit *Prepared*, soit *Callable* qui dérivent toutes les deux de *Statement*.

PreparedStatement représente une instruction SQL qui sera envoyée à la base de données. On peut donc l'utiliser pour toutes les opérations **CRUD**.

Cette instruction a trois méthodes *execute*() différentes :

1. **executeUpdate** : permet de mettre à jour la bdd. Elle renvoie un int.
2. **executeQuery** : permet de d'executer un *PreparedStatement* qui effectue une requête sur la base de données. Elle renvoie un type *ResultSet*.
3. **execute** : c'est une méthode d'exécution générale. Elle renvoie un booléen.

Method	Return type	What is returned for SELECT	What is returned for INSERT/UPDATE/DELETE
execute	boolean	true	false
executeQuery	ResultSet	data	n/a
executeUpdate	int	n/a	number of rows changed

B. Create and Execute Queries :

Dans l'exemple qui suit tout a été mis dans une méthode. Il faudra quand même créer la connexion avec le driver dans le *main* et dans un *try/with*.

On peut voir qu'on procède en 3 étapes :

1. Ecriture de la requête SQL en string dans la variable « sql ».
2. Préparation de la déclaration : on met sql en argument dans *conn.prepareStatement* et on place le tout dans une variable de type VAR.
3. On exécute la déclaration avec *executeQuery*() et on place le tout dans une variable de type *Resultset*.

Pour terminer on peut parcourir le résultat à l'aide une boucle *while* et de la méthode *next()* :

```
private static void simpleReadWithExecuteQuery(Connection conn) throws
SQLException {
    String sql = "select name, capacity from venues";

    var stmt = conn.prepareStatement(sql);

    ResultSet rs = stmt.executeQuery();

    while(rs.next()) {
        System.out.print(".");
    }

    System.out.println();
}
```

Dans cet exemple on effectue une insertion de données. Outre une requête (*Query*) plus longue, il y'a une différence importante par rapport à l'exemple précédent, on utilise la méthode d'exécution *executeUpdate*() qui stockera le résultat dans un INT. Cela est dû au fait que ce qui est renvoyé est le nombre de ligne (*row*) qui ont été affectée.

```
private static void simpleInsertWithExecuteUpdate(Connection conn) throws
SQLException {
    String sql = "insert into venues (name, capacity) values ('The House
Next Door', 20 )";

    var stmt = conn.prepareStatement(sql);

    int result = stmt.executeUpdate();

    if(result > 0)
        System.out.println("Update the database");
}
```

A noter que cette syntaxe est aussi valable pour les instructions **UPDATE** et **DELETE**.

C. Using PreparedStatement's Execute Method :

Si on reprend l'exemple précédent avec la méthode d'insert mais qu'on change la troisième étape de *executeUpdate*() à un simple *execute*(), on obtient un booléen qui permet de savoir si on renvoie un *ResultSet* ou pas. Cela peut être utile dans la mesure ou la méthode *execute*() s'utilise avec tout le **CRUD**. Donc si elle renvoie *true*, on sait que ce n'est ni un *update*, ni un *delete*, ni un *insert* mais bien un **CREATE**.

```
 boolean result = stmt.execute();

if(!result) {
    if(stmt.getUpdateCount() > 0)
        System.out.println("Update the database");
    else
        System.out.println("No update");
} else {
    System.out.println("Result was not a count");
}
```

D. Parameterizing PreparedStatement :

Jusqu'à présent les déclarations que l'on a vues étaient des valeurs codées en « dure » (*hard coded values).* Pourtant les *preparedStatement* peuvent être paramétrés pour être utilisé avec plus de facilité.

Ces paramètres ont plusieurs caractéristiques :

- Ils sont représentés par un point d'interrogation (?) dans la chaine de caractère de la requête.
- Ils ont un type (*int*, *String*, etc).
- Ils ont une position qui démarre à 1 (et pas 0).
- Ils peuvent être utilisé avec **TOUS** les types de requête (CRUD).

On doit alors utiliser la méthode *set+Type()* sur la variable *preparedStatement*. Chaque *set* remplacera implicitement le point d'interrogation correspondant à la position qui est indiquée en paramètre.

preparedStatement Set Methods

Method	Parameter Type	Example Database Type
setBoolean	Boolean	BOOLEAN
setDouble	Double	DOUBLE
setInt	Int	INTEGER
setLong	Long	BIGINT
setObject	Object	Any type
setString	String	CHAR, VARCHAR

Ici on voit tous les types de « set » qui sont possibles.

A' noter qu'on peut utiliser objet pour une utilisation plus générale mais cela sera vu plus loin.

Dans cet exemple on peut voir qu'on écrit qu'une seule fois la requête mais qu'on change les valeurs insérées avec des *set*.

```
private static void insertMutlipleValues(Connection conn) throws
SQLException {
    String sql = "insert into venues (name, capacity) values (?, ?)";

    var stmt = conn.prepareStatement(sql);
    stmt.setString(1, "Empire State Park");
    stmt.setInt(2, 3000);

    int result = stmt.executeUpdate();

    if(result > 0)
        System.out.println("Update the database");


    stmt.setString(1, "London Hyde Park");

    result = stmt.executeUpdate();

    if(result > 0)
        System.out.println("Update the database");
}
```

Une chose à laquelle il faut faire attention est que toutes les valeurs doivent être définies. Il faut donc **IMPERATIVEMENT** un set qui défini les valeurs de **CHAQUE** paramètre.

V. Working with Data from a PreparedStatement :

A. JDBC ResultSet :

On l'a vu précédemment mais pour obtenir une variable de type *ResultSet*, on utilise une requête SELECT en appelant la méthode *execute*() ou *executeQuery*(). L'objet qui en résultera est muni d'un **CURSEUR** qui pointe **AVANT** la première ligne.

Pour déplacer le curseur : on utilise **NEXT**. Cela renverra un **TRUE** s'il y'a effectivement une ligne après, et **FALSE** si ce n'est pas le cas. On utilise donc **PRIORITAIREMENT** une boucle *while* pour parcourir les résultats de la requête car dès qu'il n'y en a plus, le *next* renverra un false qui interrompra le déroulement de la boucle.

Ici, on voit que sur le tableau, le curseur est en position initiale sur la ligne du nom des colonnes. Ensuite, dès que l'on fait *next*, le curseur descend de ligne.

Table: venues

Id	Name	Capacity
1	The Arena	100
2	The Bowl	150
3	The Garage	200

Ici un exemple qui ressemble fortement au premier. La seule différence est qu'on parcourt le *resultSet* (rs ici) et qu'on va fait un *print* du contenu de la colonne *NAME*, et de la colonne *CAPACITY*.

```
var sql = "select name, capacity from
venues";

try (PreparedStatement ps =

       conn.prepareStatement(sql)) {

  var rs = ps.executeQuery();

  while(rs.next()) {

    var name = rs.getString("name");

    var capacity = rs.getInt("capacity");

    System.out.println(name +

                       " has capacity " +

                       capacity);
```

◂ Create query

◂ Create the PreparedStatement

◂ Execute the query

◂ Move and check the cursor

◂ Get the values by name

◂ Use the values

Si on peut obtenir les données en insérant le **NOM** de la colonne : *rs.getString(« name «)*, on peut aussi les obtenir tout aussi facilement en insérant le **NUMERO** de la colonne : *rs.geString(1)*.

Ensuite si l'on veut lire qu'une seule ligne de la table, pas besoin d'utiliser un while, un *if* suffit :

```
if(rs.next()) {
    var numberOfVenues = rs.getInt(1);
  System.out.println(
      "Number of venues is: " +
                  numberOfVenues);
  }
```

```
var sql = "select count(*) as count from venues";
try (PreparedStatement ps = conn.prepareStatement(sql)) {
    var rs = ps.executeQuery();

    if(rs.next()) {
        var numberOfVenues = rs.getInt("count");
        System.out.println("Number of venues is: " + numberOfVenues);
    }
}
```

Enfin dans la requête, il est tout à fait possible d'attribuer un nom à la colonne qu'on sélectionne en utilisant AS+NOM. Ce nom pourra être réutilisé lorsqu'on qu'on souhaitera parcourir les résultats.

Récapitulatif des méthodes Get :

`resultSet` Get Methods

Method	Parameter Type
getBoolean	boolean
getDouble	double
getInt	int
getLong	long
getObject	Object
getString	String

B. Using Jdbc GetObject :

Contrairement aux autres méthodes, le *getObject*() est relativement plus complexe à utiliser :

```
private static void useGetObject(Connection conn) throws SQLException {
    String sql = "select capacity, name from venues where name like ?";

    var stmt = conn.prepareStatement(sql);
    stmt.setString(1, "%the%");

    ResultSet rs = stmt.executeQuery();

    String name = "";
    int capacity = 0;

    while(rs.next()) {
        Object nameValue = rs.getObject("name");

        Object capacityValue = rs.getObject("capacity");

        if(capacityValue instanceof Integer) capacity = (int)capacityValue;

        if(nameValue instanceof String) name = (String)nameValue;

        System.out.println(name + " has capacity " + capacity);
    }
}
```

1. On met les valeurs obtenues dans une variable de type Object (on peut utiliser le type Var aussi) : *NameValue* et *CapacityValue*.
2. Ensuite on vérifie que ces deux variables sont bien des instances d'objet de type *Integer* et *String*. Si la condition est remplie on les converti (*cast*) et on les met dans une nouvelle variable qu'on a déclaré plus tôt : *name* et *capacity*.

Si les performances sont moins bonnes et le code plus verbeux, l'avantage d'une telle procédure est de permettre plus de flexibilité dans le code.

C. Binding (lier) Parameters in a Select :

Une chose que l'on peut avoir à faire est d'insérer des valeurs nulles dans la table. Pour cela on doit anticiper cette possibilité.

Dans l'exemple ci-dessous, on veut faire une insertion qui comprend deux paramètres. L'un sera déterminé par la variable *name* et l'autre par la variable *recordLabel*.

S'il y a la possibilité que *recordLabel* puisse être nulle il faut l'anticiper.

Pour cela on met la variable dans un *if* et si elle est nulle, on fait un *setNull* avec en plus de sa posituin, son type en 2ème paramètre :

```
private static void insertAct(Connection conn, String name, String
recordLabel) throws SQLException{
    var sql = "insert into Acts (name, recordlabel) values(?, ?)";

    try(PreparedStatement ps = conn.prepareStatement(sql)) {
        ps.setString(1, name);

        if(recordLabel != null)
            ps.setString(2, recordLabel);
        else
            ps.setNull(2, Types.CHAR);

        ps.executeUpdate();
    }
}
```

VI. Working with a CallableStatement :

A. Introduction :

Pour faire simple cette méthode nous permet d'utiliser des méthodes qui ont été créée en langage SQL et qui sont stockées dans la base de données.

```
create procedure GetActs()
begin
    select acts.name, acts.recordlabel
    from acts
    where acts.recordlabel IS NOT null

    order by acts.name;
end
```

Stored Procedure

This gets all the acts with a record label

Imaginons que l'on crée une méthode de requête qui s'appelle GetActs() qui nous renvoie tous les résultats s'il y'en a (non null) et trié par nom.

On peut l'utiliser avec la méthode *callableStatement*() en utilisant une syntaxe particulière : accolade et nom de la procédure le tout entre les guillemets:

```
var sql = "{ call GetActs() }";
```

Le restant du code dans ce cas ci ne change pas des exemples précédents, on stocke le résultat, on le parcourt avec un *while* et un *next*, etc….

B. With IN parameters :

Tout comme *preparedStatement*, le *callable* peut utiliser des paramètres : IN, OUT, INOUT.

```
create procedure GigReport(IN startdate Date, IN enddate Date)
begin
    select gigs.date, acts.name 'Act', acts.recordlabel, venues.name 'Venue', ticketssold,
           venues.capacity
    from gigs join acts on acts.id = gigs.actid
             join venues on venues.id = gigs.venueid
    where date >= startdate and date <= enddate
    order by gigs.date;
end;
```

On commence tout d'abord par créer la méthode en sql avec deux paramètres qui seront implémentés grâce au mot clef **IN**. Ici ils sont de types date (start et end).

Enfin lorsqu'on intègrera la méthode dans une variable sql, tout comme avant on définit l'emplacement des paramètres par des point d'interrogation entre les parenthèses et séparés par une virgule. Enfin il suffira comme avant d'initialiser les valeurs avec un *set* adéquat.

```
var sql = "{ call GigReport(?, ?) }";

try (CallableStatement cs =
conn.prepareCall(sql)) {

    cs.setDate("startdate", ...);

    cs.setDate("enddate",...);
```

C. With OUT parameters :

Les méthodes stockées en sql peuvent aussi renvoyer des valeurs. L'écriture du paramètre en Java peut s'écrire de deux manières différentes selon qu'elles sont prises en charge ou non par ce type de driver.

Ce sera donc soit un point d'interrogation seul (?) ou suivi d'un égal (?=).

- **{ ?= call sproc_name(?) }**

Exemple :

```
create procedure GetTotalSales(OUT sales decimal(8, 2))
begin
    select sum(currentvalue) 'totalsales' from
        (select ticketssold, price, ticketssold*price 'currentvalue'
         from gigs) salestable
    into sales;
end;
```

Cette procédure renverra une valeur de type décimale qui correspond au total des ventes qu'on a réalisées.

On voit dans cet exemple qu'on utilise le point d'interrogation simple pour créer le point de sortie de la méthode (l'exemple est en mySql et ne lit pas la syntaxe ?=).

```
var sql = "{call GetTotalSales(?) }";

try (CallableStatement cs =
conn.prepareCall(sql)) {



    cs.registerOutParameter(1,

                        Types.DECIMAL);

    var result = cs.execute();

    System.out.println("Total sales is: "

                    + cs.getDouble(1));


}
```

◂ Create the SQL statement (note the '?')

◂ Prepare the call

◂ Register any out parameters

◂ Execute the query

◂ Use the data from the out parameter

Ensuite pour récupérer le résultat on utilise une nouvelle méthode, *registerOutParameter*() qui prend le numéro du paramètre et le type de retour défini dans la méthode sql.

D. With IN/OUT parameters :

Dans cet exemple on utilise le troisième mot clef : **INOUT** qui nous permet aussi bien une opération de sortie que d'entrée avec le même paramètre.

```
create procedure SetNewPrice(IN gigid int, IN percentage decimal(8,2), inout maxprice
decimal(8,2))
begin
    declare gigprice decimal(8,2) default 0.0;    declare proposedprice decimal(8,2);
    set gigprice = (select max(price) from gigs where id = gigid);
    set proposedprice = gigprice + (gigprice * percentage);
    if (proposedprice < maxPrice)
    then
        set maxprice = proposedprice;
        update gigs set price = proposedprice where id = gigid;
    else
        set maxprice = gigprice;
    end if;
end;
```

Ici on a créé une procédure qui met à jour le prix de vente pour un concert. On essaye de définir le prix, et si le prix est supérieur au maximum défini, on laisse le prix non-défini et on modifie le paramètre *maxPrice* en sortie pour qu'il soit le nouveau prix.

Donc en entrée on met l'Id du concert et le pourcentage d'augmentation qu'on souhaiterait avoir.

La particularité de cette écriture, c'est que le *InOut* fonctionne dans les deux sens. Il est d'abord un paramètre d'entrée et il est ensuite un paramètre de retour.

```
var sql = "{call SetNewPrice(?, ?, ?) }";

try (CallableStatement cs =
conn.prepareCall(sql)) {


  cs.setInt(1, 1);

  cs.setDouble(2, 0.1);

  cs.setDouble(3, 12.0);

  cs.registerOutParameter(3,

                          Types.DECIMAL);
```

◂ Create the SQL statement (note the '?')

◂ Prepare the call

◂ Set the IN values

◂ Set the value for the INOUT parameter

◂ Also register that as an OUT parameter

Donc comme pour les cas précédents il faudra un *set* pour chaque point d'interrogation. Le dernier *set* étant le fameux *maxPrice*. Ensuite on récupère la valeur de retour avec un *registerOutParameter* qui correspond au 3ème paramètre.