

# Accès aux bases de données : JDBC

JDBC (Java DataBase Connectivity) est l'API standard pour interagir avec les bases de 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<sup>1</sup>

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

Sans try-with-resources<sup>1</sup>

```
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

[illegible]

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	<a href="#">jdbc:oracle:thin:@[host]:[port]:[schema]</a> Ex : <a href="#">jdbc:oracle:thin:@localhost:1521:maBase</a>
MySQL	com.mysql.jdbc.Driver	<a href="#">jdbc:mysql://[host]:[port]/[schema]</a> Ex : <a href="#">jdbc:mysql://localhost:3306/maBase</a>
PostgreSQL	org.postgresql.Driver	<a href="#">jdbc:postgresql://[host]:[port]/[schema]</a> Ex : <a href="#">jdbc:postgresql://localhost:5432/maBase</a>
HSQLDB (mode fichier)	org.hsqldb.jdbcDriver	<a href="#">jdbc:hsqldb:file:[chemin du fichier]</a> Ex : <a href="#">jdbc:hsqldb:file:maBase</a>
HSQLDB (mode mémoire)	org.hsqldb.jdbcDriver	<a href="#">jdbc:hsqldb:mem:[schema]</a> Ex : <a href="#">jdbc:hsqldb:mem:maBase</a>

## 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')");
```

UnStatement 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.

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");

        // ...
    }
}
```

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.

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();
}
```

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 '' = ''
```

## 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<sup>¶</sup>

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

Requête JDBC pour l'appel d'une fonction stockée<sup>¶</sup>

```
{? = 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<sup>¶</sup>

```
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);

    // ...
}
```

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();
    }
}
```

---

Revision #1

Created 10 February 2025 23:30:58 by Nicolas

Updated 10 February 2025 23:33:12 by Nicolas