Université Bretagne Sud

R3.07. SQL dans un langage de programmation

1. Abstractions pour l'interaction avec des sources de données relationelles avec JDBC

Hélène Evenas, Selena Lamari et Chouki Tibermacine

Vannes
iut
Université Bretagne Sud

Plan du cours

Introduction à JDBC

Définition

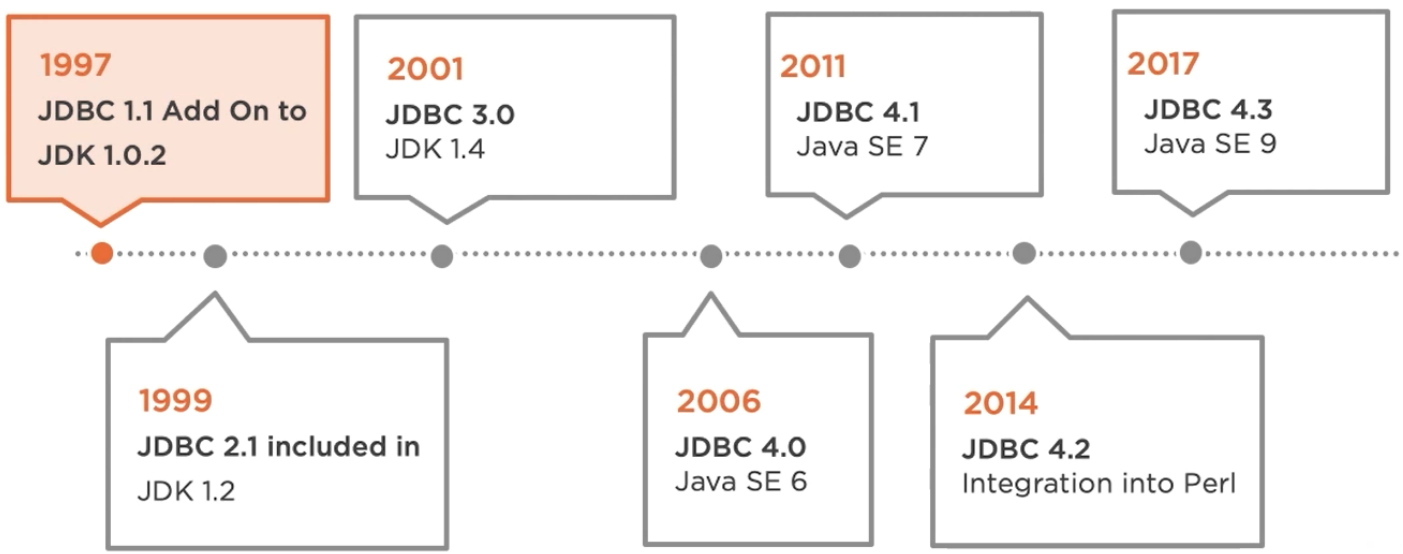
JDBC (**Java Database Connectivity**) est une **API Java** standard qui permet aux applications Java d'interagir avec des bases de données relationnelles.

À quoi ça sert ?

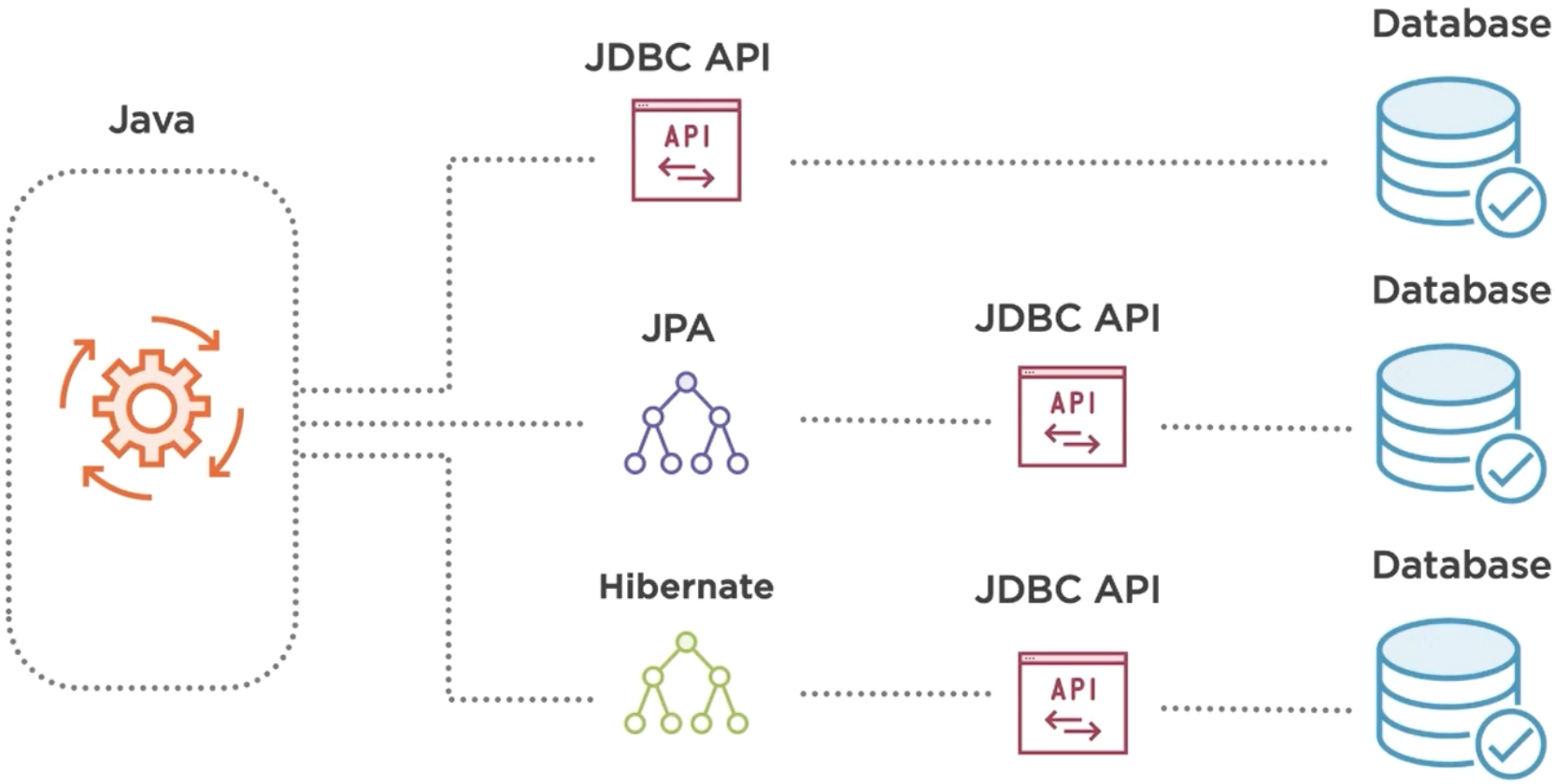
- Établir une connexion entre une application Java et une base de données (MySQL, PostgreSQL, Oracle, etc.).
- Exécuter des requêtes SQL (SELECT, INSERT, UPDATE, DELETE).
- Récupérer et manipuler les résultats des requêtes.
- Gérer les transactions.

Pourquoi utiliser JDBC ?

- Indépendance vis-à-vis du SGBD (grâce aux **pilotes JDBC**).
- Intégration native dans l'écosystème Java.
- Support des fonctionnalités avancées (transactions, batch, métadonnées).
- L'une des premières API pour app d'entreprise Java :



JDBC partout où Java interagit avec une BdD



Installer un serveur

- Nous allons utiliser Postgres comme serveur de données
- Démarrer un serveur Postgres en utilisant Docker
- D'abord s'assurer qu'il n'y a pas un service Postgres qui est actif sur votre machine (sous Windows, voir la configuration des services)
- Si vous n'avez pas Docker sur votre machine, l'installer : `https://docs.docker.com/get-docker/`
- Pour démarrer un container Docker avec un serveur Posgres :
  - Ouvrir un terminal
  - Exécuter la commande :

```
docker run --name postgres-db --rm -P -p
127.0.0.1:5432:5432 -e POSTGRES_USER=postgres -e
POSTGRES_PASSWORD=postgres -d postgres
```

Installer un serveur -suite-

- Tester le bon démarrage du container/serveur avec la commande : `docker ps`

  Ceci doit vous indiquer la présence d'un container démarré sous le nom `postgres-db`
- Pour arrêter le container : `docker stop postgres-db`
- Pour supprimer le container : `docker rm postgres-db`

Installer un serveur -suite-

- Tel que nous l'avons créé et démarré, le container Docker ne permet pas de garder les données de la BdD sur le *file system* de la machine hôte, si jamais on l'arrête (les données sont perdues)
- Pour avoir des données persistantes, il faut créer/utiliser un volume :
  1. Arrêter le container : `docker stop postgres-db`
  2. Créer un volume : `docker volume create app_vol`
  3. Démarrer le container en utilisant ce volume :

```
docker run --name postgres-db --rm -P -p
127.0.0.1:5432:5432 -e POSTGRES_USER=postgres -e
POSTGRES_PASSWORD=postgres -v
app_vol:/var/lib/postgresql -d postgres
```

Accéder au serveur en ligne de commande avec PSQL

- Pour accéder au serveur, taper la commande :
  `docker exec -it postgres-db psql -U postgres`

  Ceci démarre un shell PSQL dans le container postgres-db avec l'utilisateur (option -U) `postgres`
- Dans le Shell, lister les bases de données avec *\l*
- Vous pouvez lister les tables de la base de données courante (qui s'appelle postgres, créée par défaut) avec *\d*
- Au départ, ceci vous indique qu'il n'y a pas de tables (relations)
- Créer une base de données avec la requête SQL :
  `CREATE DATABASE app_db;`

  Avec la commande *\l*, vous allez voir cette base de données

Exécuter des requêtes en ligne de commande sur PSQL

- Se connecter à cette base : \c *app_db*
- Créer une table dans cette base de données en utilisant la requête suivante :

```
CREATE TABLE users(
 user_id serial NOT NULL PRIMARY KEY,
 first_name varchar(30) NOT NULL,
 last_name varchar(30) NOT NULL,
 email varchar(80) NOT NULL,
 phone_number varchar(20) NOT NULL,
 password varchar(100) NOT NULL
);
```

  Vérifier la présence de la table dans la base (\d)
- Vérifier le schéma de la table avec : \d *users*

Exécuter des requêtes en ligne de commande sur PSQL -suite-

- Dans le Shell PSQL, ajouter l'extension qui nous permettra d'exécuter des requêtes SQL où l'on peut crypter des mots de passe :

  ```
  CREATE EXTENSION pgcrypto;
  ```

- Insérer un tuple (record) dans la table users : (attention aux apostrophes quand vous copiez/collez depuis le PDF)

  ```
   INSERT INTO
  users(first_name,last_name,email,phone_number,password)
   VALUES('Harry','Potter','harry.potter@gmail.com',
    '+44 123456789', crypt('1234',gen_salt('bf', 8)));
  ```

  Noter à la dernière ligne l'utilisation des fonctions de `pgcrypto` pour le hachage du mot de passe en utilisant l'algorithme BF (BlowFish) avec un sel

Exécuter des requêtes en ligne de commande sur PSQL -suite-

- Vérifier la présence de ce tuple dans la table :

```
SELECT * FROM users;
```

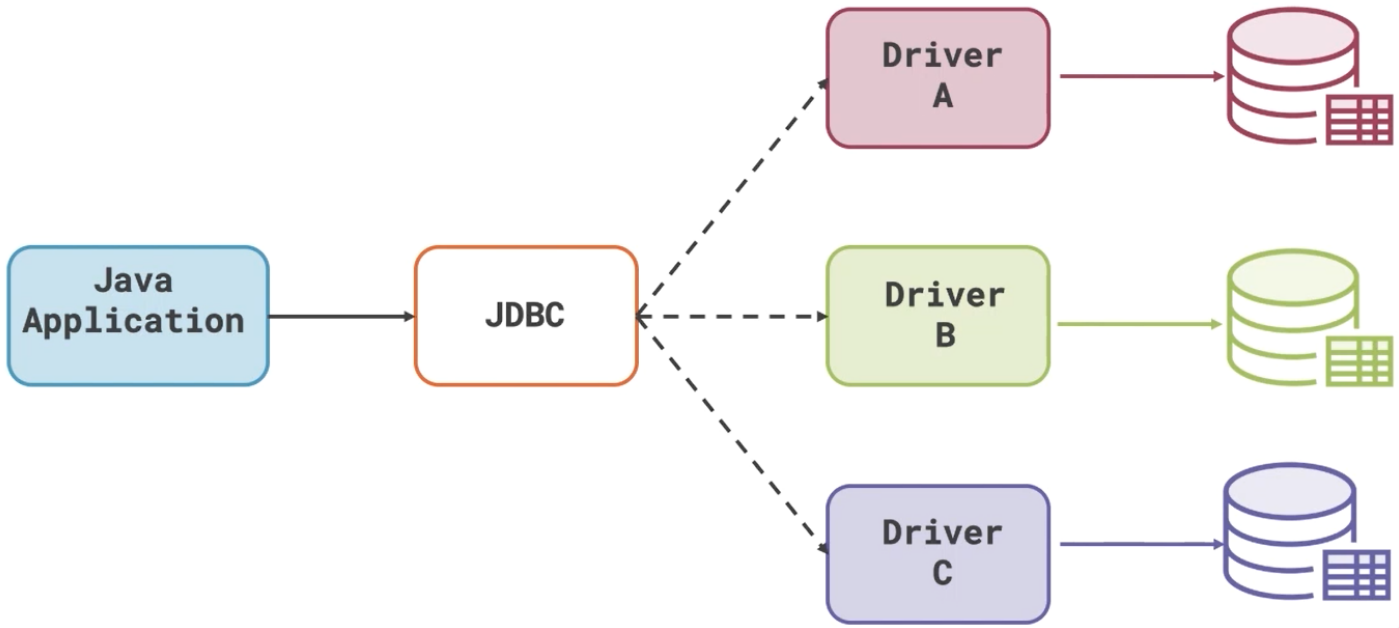
- Tester d'autres requêtes SQL
- Pour quitter le Shell : $\backslash q$

Noter que le serveur Postgres est par défaut accessible sur le port 5432 (modifiable dans la commande de démarrage du container Docker)

Plan du cours

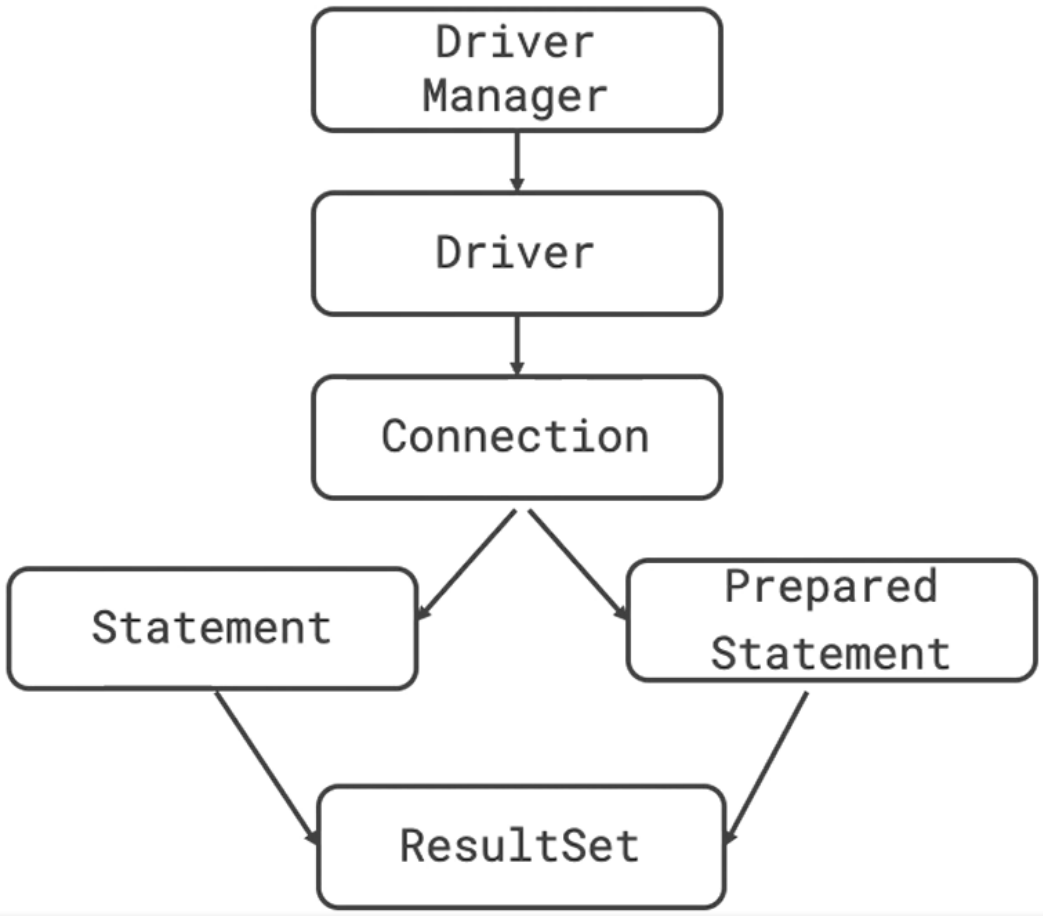
Plan du cours

JDBC utilise le patron Façade



- Derrière l'API JDBC, il y a des pilotes (*Drivers*) spécifiques aux serveurs de BdD
- Ils sont utilisés à l'exécution (implémentent l'API)

Interfaces de l'API



Interfaces de l'API -suite-

- `DriverManager` gère l'annuaire de drivers
- `Driver` est une interface implémentée sous la forme d'une classe unique (théoriquement) par les différents fournisseurs (*vendors*) de serveurs de BdD
  - A son chargement par la JVM une instance est créée
  - Cette instance s'enregistre auprès du `DriverManager`
  - Elle permet d'obtenir une connexion au serveur
- `Connection` permet d'établir une connexion avec le serveur de BdD afin d'exécuter des requêtes
- `Statement` et `PreparedStatement` sont des abstractions des requêtes SQL
- `ResultSet` représente les résultats de l'exécution des requêtes

Connexion au serveur

- Pour se connecter au serveur avec JDBC, on utilise :

```
Connection connection =
 DriverManager.getConnection(
  "jdbc:postgresql://localhost:5432/app_db",
  "postgres", "postgres");
```

  où le premier paramètre est l'URL JDBC de connexion à un serveur Postgres
- Pour d'autres fournisseurs de serveurs (Oracle, MySQL, ...) l'URL est légèrement différent :
  `jdbc:mysql://localhost:3306/app_db`
  pour MySQL, par ex
- Écrire le code ci-dessus dans la méthode main d'une classe Java que vous créez dans un projet Maven sous IntelliJ
- On complètera ce code dans les prochains transparents

Tester la connexion et fermer la connexion

- Pour tester la connexion :

```
connection.isValid(2);
```

  Le paramètre de `isValid()` est un timeout en secondes

- Il faut toujours fermer proprement la connexion à une base de données, sinon consommation importante de ressources systèmes (mémoire, sockets et threads) :

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

  Le faire proprement veut dire qu'il faut soit mettre l'instruction dans une clause `finally` d'un bloc `try` ou bien utiliser un bloc `try-with-resources` (voir cours de gestion des exceptions IG3, ou exemples suivants)

Utiliser un Driver précis

- Pour le moment, on utilise des interfaces et classes abstraites de l'API JDBC (package java.sql)
- Mais comment on indique à la JVM quel Driver (classe d'implémentation de l'interface Driver) précis(e) utiliser ?
  1. Ajouter un dossier `META-INF/services` dans le dossier `src/main/resources` de votre projet IntelliJ
  2. Ajouter dans ce dossier un fichier texte et le nommer : `java.sql.Driver`
  3. Dans ce fichier, on peut mettre la liste de tous les drivers qu'on veut utiliser dans notre projet
  4. Pour notre exemple (Postgres), on ajoute la ligne : `org.postgresql.Driver` (c'est la classe d'implémentation de l'interface `java.sql.Driver`)

Utiliser un Driver précis -suite-

D'où vient la classe `org.postgresql.Driver` ?

1. Déclarer la dépendance Maven suivante :

```
<dependencies>
 <!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->
 <dependency>
  <groupId>org.postgresql</groupId>
  <artifactId>postgresql</artifactId>
  <version>42.7.8</version>
 </dependency>
</dependencies>
```

2. Ouvrir la tool-window Maven sur la bande verticale à droite de la fenêtre IntelliJ et appuyer sur le bouton *reload*

   Ceci va provoquer le téléchargement de la dépendance (et donc du JAR qui comporte la classe `org.postgresql.Driver`)

Applications légataires

- Dans certaines applications légataires, qui utilisent d'anciennes versions de JDBC (avant 2006), vous trouverez l'instruction suivante, avant `getConnection()` :

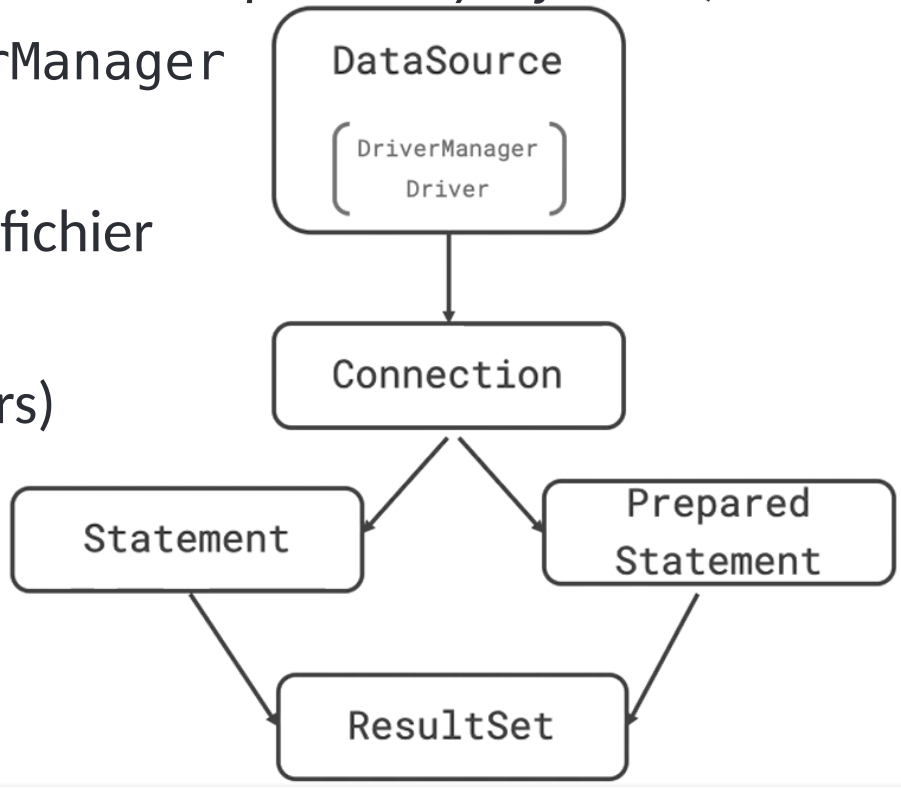
```
Class.forName("org.postgresql.Driver").newInstance();
```

  Ceci permet de déclencher le chargement de la classe Driver et de son instanciation

- Depuis Java 6, et l'arrivée du système de services, les drivers sont enregistrés automatiquement depuis le fichier `java.sql.Driver`, vu précédemment

Autre méthode pour obtenir une connexion au serveur de BdD

- Autre méthode : en utilisant un objet de type `DataSource`
- Avec JNDI de JEE ou avec des annotations @Resource de JEE ou @Inject de CDI (*Contexts and Dependency Injection*)
- Abstraction de `DriverManager` et `Driver`
- URL et Driver dans un fichier de config
- Voir JEE (prochain cours)



Plan du cours

Création de requêtes SQL

- Pour exécuter une requête, on doit s'appuyer sur des objets de type `Statement`
- Pour créer de tels objets, on utilse l'objet de type `Connection` :

```
Statement statement = connection.createStatement();
```

- Ensuite, on peut exécuter des requêtes SQL :

```
ResultSet resultSet = statement.executeQuery(
                        "SELECT * FROM users;");
```

  Cette méthode retourne un objet enveloppant le résultat

- Il existe plusieurs méthodes pour exécuter des requêtes : `execute()`, `executeQuery()`, `executeUpdate()`, ... :

https://docs.oracle.com/en/java/javase/15/docs/api/java.sql/java/sql/Statement.html

Libération des ressources

- Il est important de libérer les objets utilisés pour se connecter à la base de données immédiatement après les avoir utilisé
- Ceci permet de ne plus les attacher à la BdD ⇒ libérer les ressources de l'OS
- Il faudra donc invoquer la méthode close() sur chacun, dans l'ordre inverse de leur création :

```
resultSet.close();
statement.close();
connection.close();
```

- Sinon utiliser des blocs imbriqués try-with-resources

Récupération des résultats

- Si on veut itérer sur les résultats retournés par un SELECT par exemple, il nous faudra une boucle :

```
while(resultSet.next()) {
  // Faire quelque chose avec le tuple courant
  // ...
}
```

- Un objet `ResultSet` maintient un curseur pour se déplacer dans les tuples du résultat
- La première invocation à `next()` place le curseur sur le premier tuple dans les résultats et retourne true
- Les invocations suivantes déplacent le curseur jusqu'à la fin (où elle retourne false)

Récupération des résultats -suite-

- Contrairement à JPA (qui fait le mapping objet/relationnel), JDBC donne accès aux données brutes de la BdD
- Il faudra donc passer par les multiples méthodes de l'interface `ResultSet` pour récupérer chaque champ des tuples : `getBoolean`, `getLong`, `getString`, ...

  https://docs.oracle.com/en/java/javase/15/docs/api/java.sql/java/sql/ResultSet.html
- Les champs peuvent être obtenus par leur position (numéro de colonne dans la table) ou par leur nom : `getString(4)` ou `getString("email")`
- Navigations possibles avec : `next()`, `previous()`, `first()`, `last()`, ...

Mapping des types de données

- Il n'y a pas de mapping direct (1-1) entre les types génériques SQL et les types Java
- Les types SQL dépendent du fournisseur du serveur de BdD
- Exemple : le type varchar a des limitations différentes selon le serveur (+65k caractères max sur MySQL, 8k sur SQL Server de Microsoft, et 4k sur Oracle)
- Sur certains serveurs, on a un type Money ...
- En général, le mieux est de ne pas faire d'hypothèses sur les types spécifiques aux serveurs (Money) et définir ses propres classes qui représentent les types complexes (pour rester indépendant d'un fournisseur de serveur particulier)
- Types JDBC :

`https://docs.oracle.com/en/java/javase/15/docs/api/java.sql/java/sql/Types.html`

Exécution de requêtes SQL avec `PreparedStatement`

- Meilleur mécanisme pour exécuter des requêtes SQL

```
PreparedStatement statement =
 connection.prepareStatement("SELECT * FROM users;");
ResultSet resultSet = statement.executeQuery();
```

- En quoi c'est meilleur ?
  1. Possibilité d'écrire des requêtes paramétrées par des données utilisateurs
  2. Protection des requêtes paramétrées des attaques (par injection SQL, par ex)
  3. Meilleure performance : la BdD anticipe le parsing et l'optimisation d'un plan de requêtes (contrairement à `Statement`)

Exécution de requêtes paramétrées SQL

- Créer une requête préparée et lui passer un paramètre :

```
PreparedStatement statement =
 connection.prepareStatement("SELECT * FROM users
    where user_id=?;");
statement.setInt(1,1);
ResultSet resultSet = statement.executeQuery();
```

- `setInt()` permet de passer un paramètre
- Le première paramètre indique le numéro du paramètre : 1, 2, ... (dans l'exemple, on a un seul paramètre : 1)
- Le deuxième paramètre indique la valeur du paramètre passé

Exemple plus complet

```
import java.sql.*;
public class UserManager {
 public boolean printUser() throws SQLException {
  boolean isValid = false;
  try (Connection connection =
    DriverManager.getConnection("jdbc:postgresql://localhost:5432/app_db",
                                "postgres", "postgres")){
   isValid = connection.isValid(2);
   try(PreparedStatement statement =
     connection.prepareStatement("SELECT * FROM users WHERE user_id=?;");){
    statement.setInt(1, 1);
     try (ResultSet resultSet = statement.executeQuery()) {
      while (resultSet.next()) {
       System.out.println(resultSet.getString("email"));
      }
     }
   }
  }
  return isValid;
 }
 public static void main(String... args) throws Exception {
  new UserManager().printUser();
 }
}
```

Mise à jour des données

- Invoquer la méthodes `executeUpdate()` sur un objet `PrepraredStatement` (utiliser la même méthode pour l'insertion et la suppression de donnéess)

```
public void updateEmail(int user_id) throws SQLException {
 try (Connection connection =
  DriverManager.getConnection("jdbc:postgresql://localhost
    :5432/app_db", "postgres", "postgres")){
  String query = "UPDATE users SET email=?
                  WHERE user_id=?;";
  try(PreparedStatement statement =
   connection.prepareStatement(query);){
   statement.setString(1, "harry.potter@umontpellier.fr");
   statement.setInt(2,user_id);
   statement.executeUpdate();//retourne nb tuples mis a jour
  }
 }
}
```

Conteneuriser l'application Java avec Docker

- Construisez un JAR exécutable de votre application, en utilisant Maven :
  - récupérez le fichier pom.xml sur Moodle (on a ajouté dedans un plugin maven, qui s'appelle *shade*, et qui permet de créer un JAR exécutable)
  - éditez le fichier pom.xml pour changer le nom de la classe qui contient le main
- Ajoutez dans votre projet le Dockerfile qui est sur Moodle
- Ce fichier va vous permettre de construire une image Docker de votre application comme suit :

  `docker build -t example_jdbc .`

  où example_jdbc est le nom de votre image (celle-ci sera taggée `latest`, sinon utilisez `example_jdbc:1.0.0`)
- Démarrez votre app dans un container : (l'app ne marchera pas)

  `docker run example_jdbc`

Lier les containers par un réseau Docker

Pourquoi l'app ne marchait pas ? parce que les containers (Postgres et celui qui va exécuter votre app Java) sont isolés et ne partagent aucun réseau

- Créez un réseau utilisateur Docker :
  `docker network create app_net`
- Au démarrage de chacun des deux containers, utilisez l'option `--network` comme suit :
  `docker run ... --network app_net postgres`
  `docker run --network app_net example_jdbc`
- Dans le code Java, remplacez, dans l'url de connexion, `localhost` par le nom du container Docker : `postgres-db`
- Utilisez le port 5432. Le port mapping n'est pas utile. Vous pouvez le supprimer de la commande `docker run ...` qui démarre le container postgres (l'option `-p`)

Plan du cours

CLOB et BLOB

- De larges morceaux de données texte (CLOB) ou binaires (BLOB)
- *CLOB : Character Large OBject*
- *BLOB : Binary Large OBject*
- Exemples de CLOB : des documents texte, des descriptions HTML, ...
- Exemples de BLOB : des images, des vidéos, ...

Écrire un CLOB dans une BdD

- D'abord insérer une nouvelle colonne cv dans la table users : Sur le Shell psql

```
ALTER TABLE users ADD COLUMN cv text;
```

- Ensuite, lire un texte depuis un InputStreamReader et utiliser JDBC pour l'ajouter à la requête SQL update ou `insert` :

```
String query = "UPDATE users SET cv=? WHERE user_id=?;";
try(PreparedStatement statement =
 connection.prepareStatement(query);){
 InputStreamReader inputStreamReader =
  new InputStreamReader(getClass().getClassLoader().
     getResourceAsStream("cv.html"));
 statement.setCharacterStream(1, inputStreamReader);
 statement.setInt(2,1);
 statement.executeUpdate();
 return true;
}
```

Exemple précédent

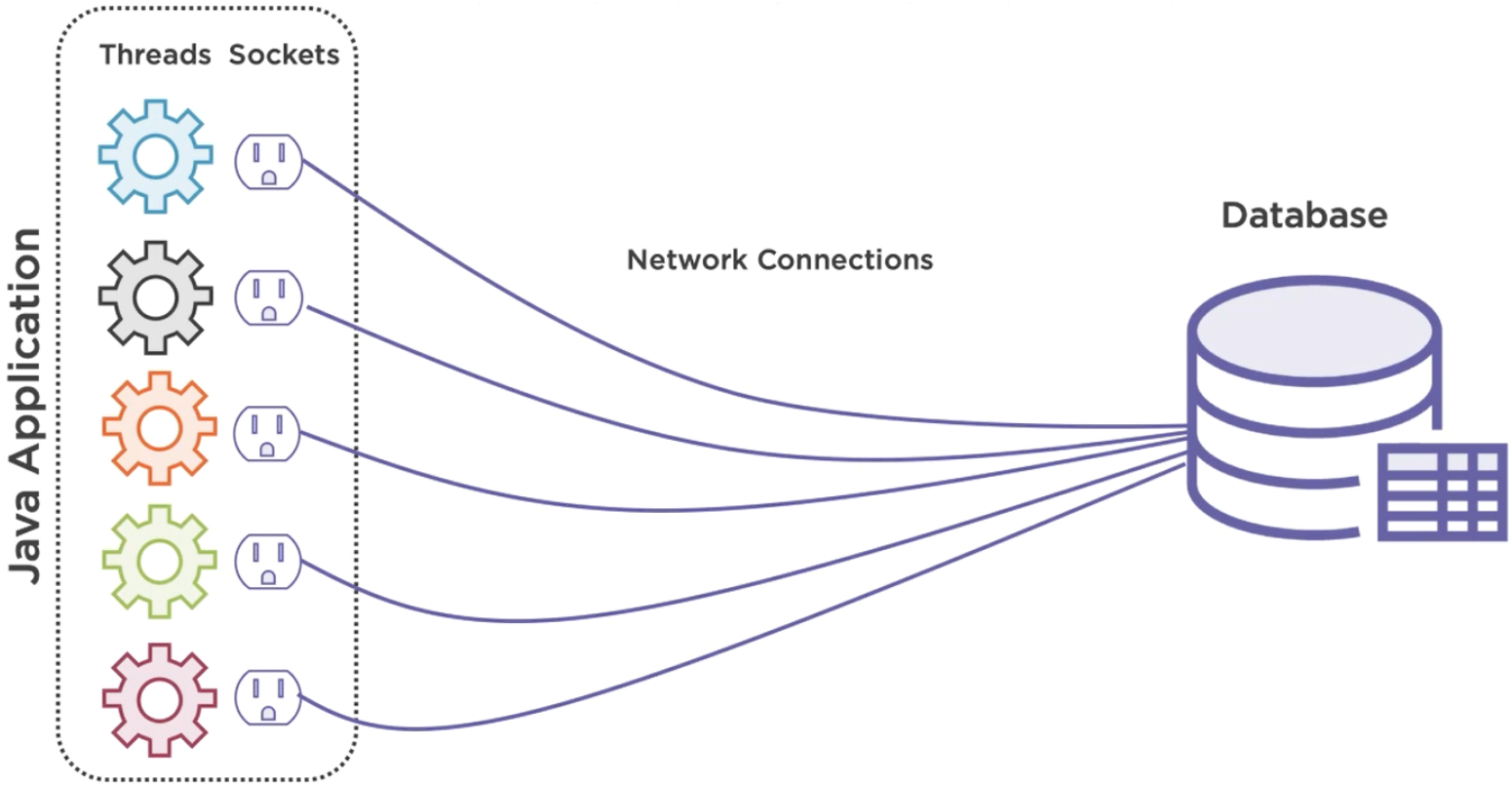
- Dans la diapo précédente, nous avons utilisé la méthode `setCharacterStream()` de `PreparedStatement` pour ajouter le texte lu depuis le stream dans la requête
- Pour lire le texte depuis le résultat de l'exécution d'une requête, on utilise `getCharacterStream()` de `ResultSet`
- Cette méthode prend en paramètre la position de la colonne ou son nom (comme les autres méthodes getXxx() de `ResultSet`)
- A vos claviers : tester l'exemple

Lire et écrire des BLOB

- Pour les fichiers binaires (données BLOB), utiliser `FileInputStream` à la place de `InputStreamReader`
- Utiliser la méthode `setBinaryStream()` à la place de `setCharacterStream()`
- Utiliser la méthode `getBinaryStream()` à la place de `getCharacterStream()`
- A vos claviers : tester l'exemple d'un fichier contenant la photo de l'utilisateur

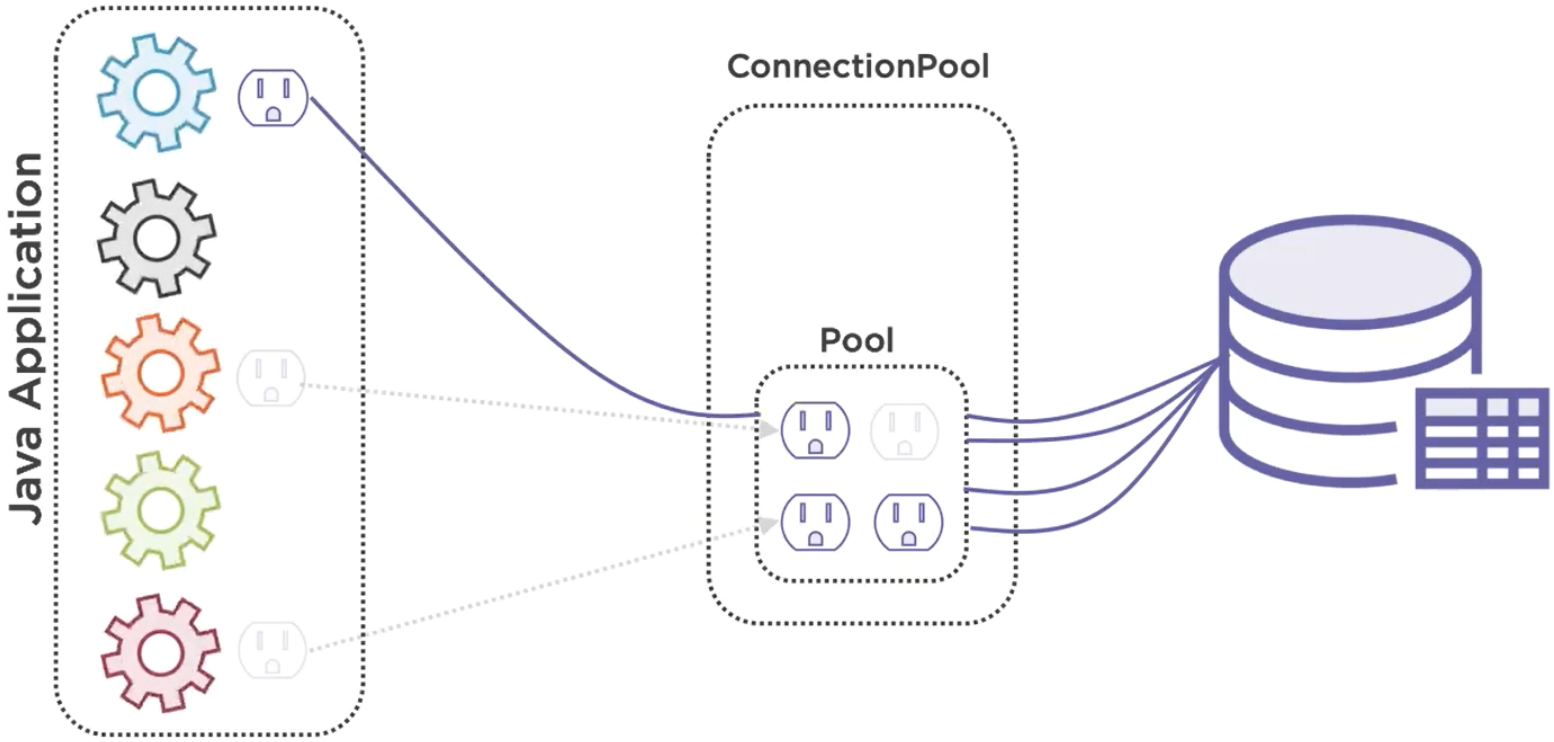
Viviers de connexions (*Connection Pools*)

Inconvénient des connexions individuelles



Trop de ressources systèmes consommées

Viviers de connexions (*Connection Pools*)



Viviers de connexions (*Connection Pools*)

- Ce mécanisme est spécifié dans JDBC comme alternative pour se connecter à un serveur de BdD (l'API JDBC reste inchangée que l'on utilise un vivier ou pas)
- Un fournisseur de *Connection Pools* gère un cache avec un vivier de connexions qui ne sont pas fermés après chaque invocation de `close()`
- Les connexions sont libérées et peuvent être réutilisées par d'autres threads
- Si toutes les connexions d'un vivier sont utilisées, ce fournisseur en crée de nouvelles
- Il existe plusieurs fournisseurs : HikariCP, Apache DBCP, C3PO, ...
- Les serveurs d'applications (comme Java EE – prochain cours) fournissent ce genre de mécanismes (scalabilité, perf, ...)

Transactions dans JDBC

- Plusieurs requêtes à exécuter de façon atomique
- Par défaut, JDBC fait un autocommit après l'exécution de chaque requête
- Si jamais on a, par exemple, une première requête qui s'exécute avec succès et la suivante qui échoue, la BdD peut se retrouver dans un état incohérent
- Solution dans JDBC : utiliser l'objet Connection
  1. désactiver l'autocommit au début : `connection.setAutoCommit(false);`
  2. invoquer `connection.commit()` si succès de toutes les requêtes de la transaction et `connection.rollback()` sinon

Les objets RowSet

- `RowSet` est une extension de `ResultSet`
- Un objet `RowSet` peut être sérialisé (stocké, envoyé, ...)
- `RowSet` peut être utilisée même lors d'une déconnexion du serveur de la BdD :
  - `ResultSet` maintient les ressources système (threads, sockets, ...) jusqu'à l'invocation de `close()`
  - `RowSet` peut utiliser un cache en mémoire (`CachedRowSet`). ça libère donc la connexion au serveur
- Avantage : + rapide, mais peut consommer beaucoup de mémoire et les données peuvent être obsolètes
- A utiliser avec des données qui ne sont pas fréquemment mises à jour

Les objets RowSet -suite-

- Exemple d'utilisation d'un `CachedRowSet` :

```
String query = "SELECT * FROM users;";
RowSetFactory factory = RowSetProvider.newFactory();
CachedRowSet rowSet = factory.createCachedRowSet();
rowSet.setUrl("jdbc:postgresql://localhost:5432/app_db?
    user=postgres&password=postgres");
rowSet.execute();
```

Après l'invocation de `execute()`, on peut invoquer les méthodes `getXxx()` héritées de `ResultSet`