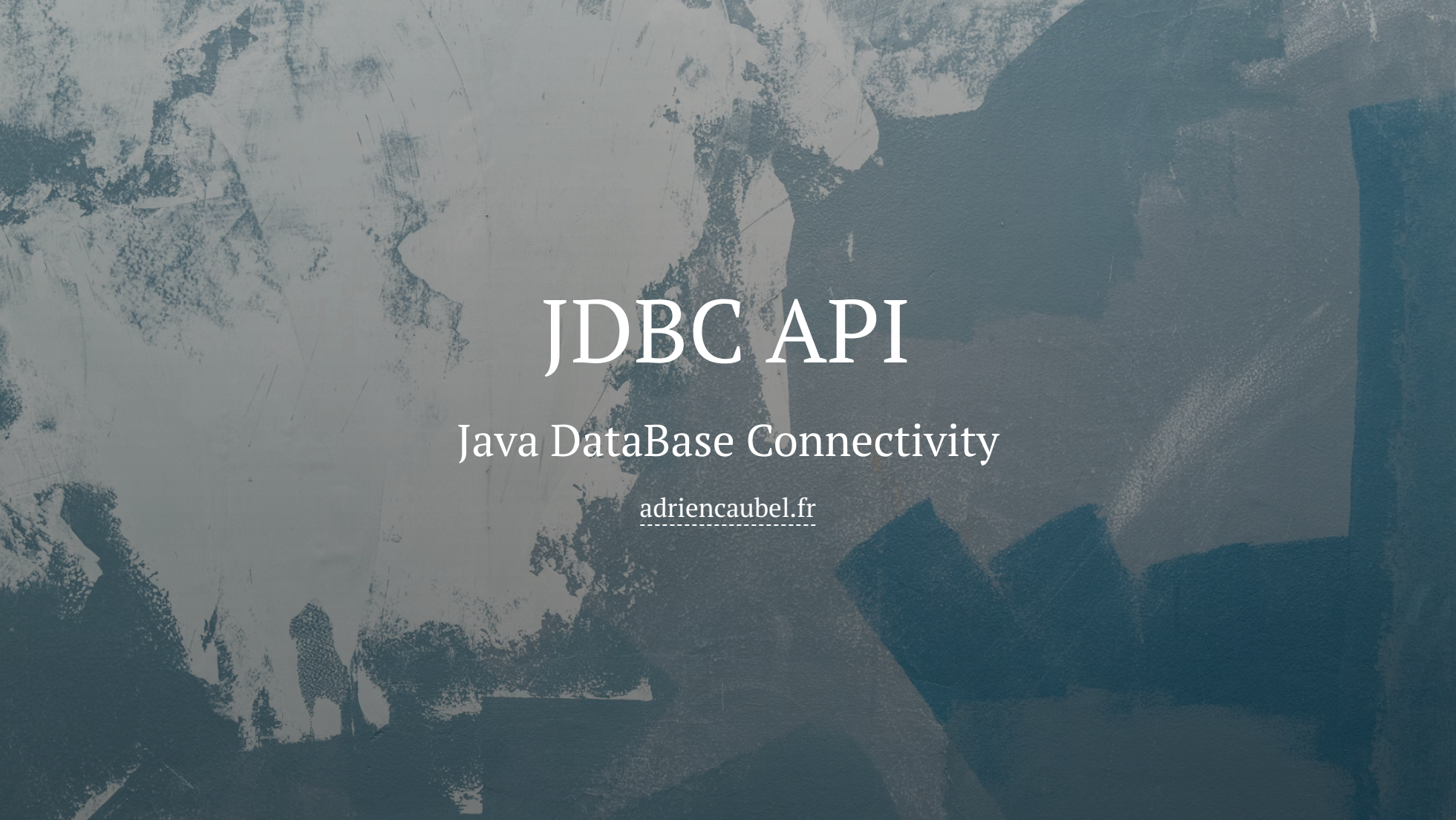


JDBC API

Java DataBase Connectivity

adriencaubel.fr

Ressources

- https://www.marcobehler.com/guides/jdbc

Table des matières

L’API JDBC

Définition

Java DataBase Connectivity (JDBC) est une API de bas niveau permettant à une application Java de se connecter et d'interagir avec une base de données relationnelle.

Exemple

```
Connection conn =
    DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/mabase",
        "utilisateur",
        "motdepasse"
    );

ResultSet rs = conn
    .createStatement()
    .executeQuery("SELECT * FROM matable");
```

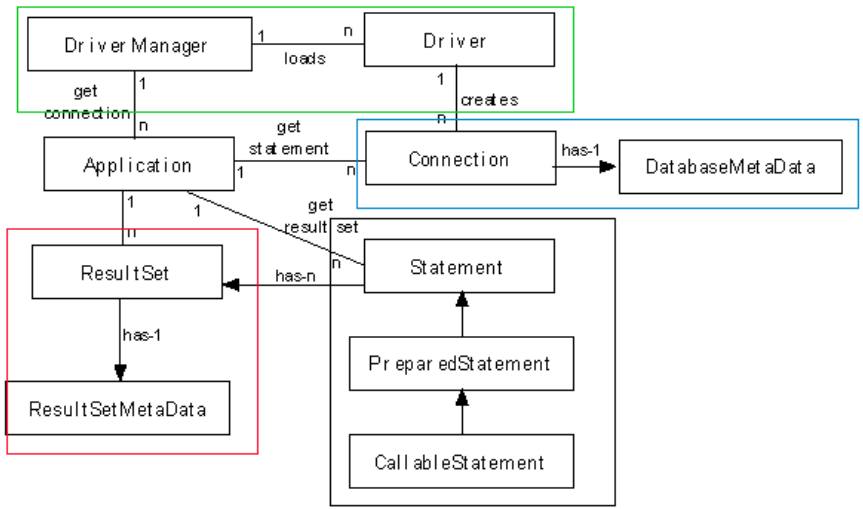
1. Créer une Connection
2. Executer une requête

Architecture de JDBC

- **Niveau supérieur : API JDBC**
  - package `java.sql`

- **Niveau inférieur : DRIVERS**
  - interface entre les accès bas niveau au moteur du SGBD et l'application
  - chaque SGBD utilise un pilote (driver) particulier
  - => Les drivers permettent de traduire les requêtes JDBC dans le langage du SGBD

L'API JDBC



4 Types

- DriverManager
- Connection
- Statement
- ResultSet

=> Ensemble, ces types permettent d'interagir avec une base de données.

Intéragir avec le BDD

Ouvrir une connexion

```
Class.forName("com.mysql.jdbc.Driver");
Connection connnection = DriverManager.getConnection("jdbc:mysql://localhost:3306/mabase", "user", "pdw")
```

1. Charger le pilote (Driver) puis 2. Établir la connexion à la BDD

Établir un statement

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

Via l'object `connection` nous créons un `Statement` que nous allons exploiter

Exécuter une requête

```
Statement stmt = conn.createStatement();
ResultSet resultats = stmt.executeQuery("SELECT * FROM client");
```

- Executer une requête SQL
  - `executeQuery()` pour effectuer un `SELECT`
  - `executeUpdate()` pour effectuer un `INSERT`, `UPDATE` ou `DELETE`

Traitement du résultat

```
ResultSet rs = stmt.executeQuery(requete);

while (rs.next()) {
   String nom = rs.getString("nom");
   int age = rs.getInt("age");
}
```

- Itérer sur les lignes de l'objet `ResultSet`

Fermeture de la connexion

Chaque objet possède une méthode close() :

- `resultset.close();`
- `statement.close();`
- `connection.close();`

Code complet

```
Class.forName("com.mysql.jdbc.Driver");
Connection conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/mabase", "user", "pdw")

Statement stmt = conn.createStatement();

ResultSet rs = stmt.executeQuery("SELECT * FROM client");

while (rs.next()) {
   String nom = rs.getString("nom");
   int age = rs.getInt("age");
}

conn.close();
stmt.close();
rs.close();
```

Compléments

Traduction des types de données SQL/Java

Type SQL	Type Java
CHAR, VARCHAR, LONGCHAR	String
NUMERIC, DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float

Traduction des types de données SQL/Java

Type SQL	Type Java
FLOAT, DOUBLE	double
BINARY, VARBINARY, LONGVARBINARY	byte
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

PrepareStatement

```
Statement stmt = con.createStatement();
String sql = "SELECT * FROM users WHERE name + 'name';"

ResultSet rs = stmt.executeQuery(sql)
```

- L'écriture du `Statement` précédent permet des Injections SQL
- Une façon plus simple est d'éviter de les utiliser et préférer les `PrepareStatement`

PrepareStatement

```
PrepareStatement ps = con.createPrepareStatement("SELECT * FROM users WHERE name = ?");
ps.setString(1, "Paul"); // /!\ On commence à 1 et pas à 0

ResultSet rs = ps.executeQuery();
```

1. La requête SQL est paramétrable via le `?`
2. Puis on précise la valeur du paramètre
3. Et finalement on exécute `executeQuery()` ou `executeUpdate()` sans paramètres

DataSource

Pour le moment nous créons une connexion à la base de données via le DriverManager. Mais depuis la version 2.0 de l'API JDBC nous pouvons utiliser l'interface DataSource

The simplest `javax.sql.DataSource` implementation could delegate connection acquisition requests to the underlying `DriverManager`, and the connection request workflow would look like this:

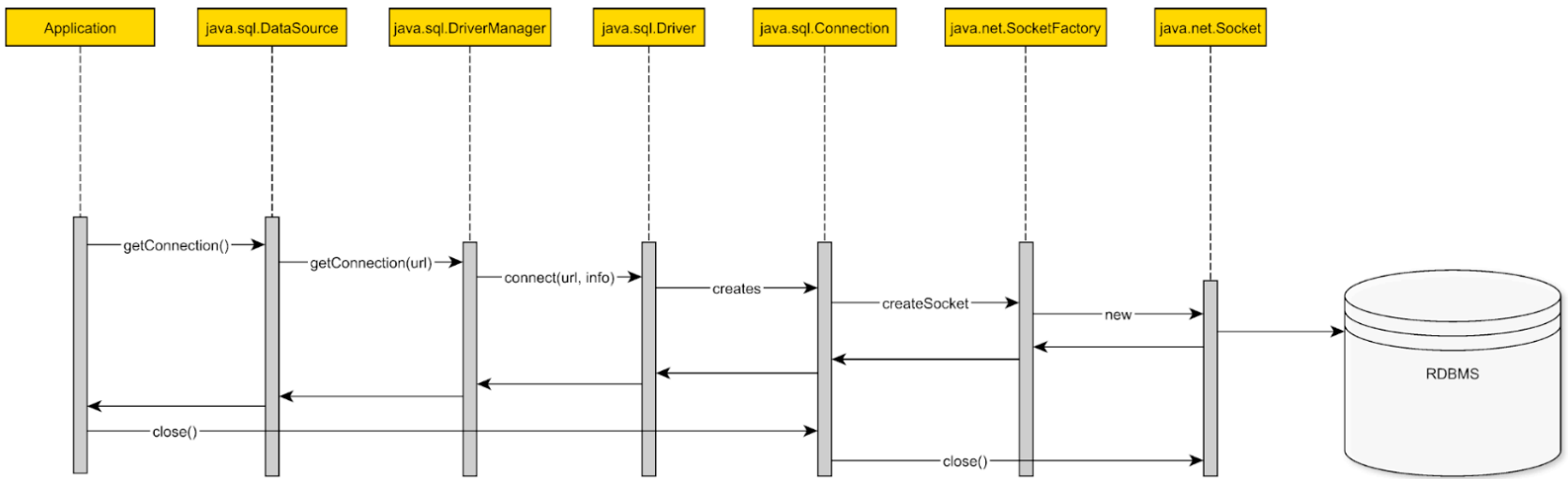


Figure 3.5: DataSource without connection pooling

Avantage DataSource

Nous ne sommes pas obligé de donnée l'url de connexion exacte qui est propre à chaque drivers :

- `jdbc:mysql://`
- `jdbc:postgres://`

Nous allons simplement donner le serveur, le port ou encore la table

```
dataSource.setServeurName("localhost");
dataSource.setPort(3306);
dataSource.setDatabaseName("client");
...
```

Pool de connexion

L'ouverture et la fermeture des connexions à la base de données sont des opérations très coûteuses.
Donc réutiliser des connexions présente les avantages suivants :

- évite à la fois la surcharge de la base de données et celle du pilote pour établir une connexion TCP
- empêche la destruction des tampons de mémoire temporaires associés à chaque connexion à la base de données
- réduit les déchets d'objets JVM côté client.

Table 3.1: Database connection establishing overhead vs connection pooling

Metric	Time (ms) DB_A	Time (ms) DB_B	Time (ms) DB_C	Time (ms) DB_D	Time (ms) HikariCP
min	11.174	5.441	24.468	0.860	0.001230
max	129.400	26.110	74.634	74.313	1.014051
mean	13.829	6.477	28.910	1.590	0.003458
p99	20.432	9.944	54.952	3.022	0.010263

Pooling fonctionnement

1. Lorsqu'une connexion est demandée, le pool recherche les connexions non attribuées.
2. Si le pool en trouve une, il la transmet au client.
3. Sinon, le pool tente d'atteindre sa taille maximale autorisée.
4. Si le pool a déjà atteint sa taille maximale, il réessaie plusieurs fois avant d'abandonner avec une exception d'échec d'acquisition de connexion.
5. Lorsque le client ferme la connexion logique, celle-ci est libérée et retourne au pool sans fermer la connexion physique sous-jacente.

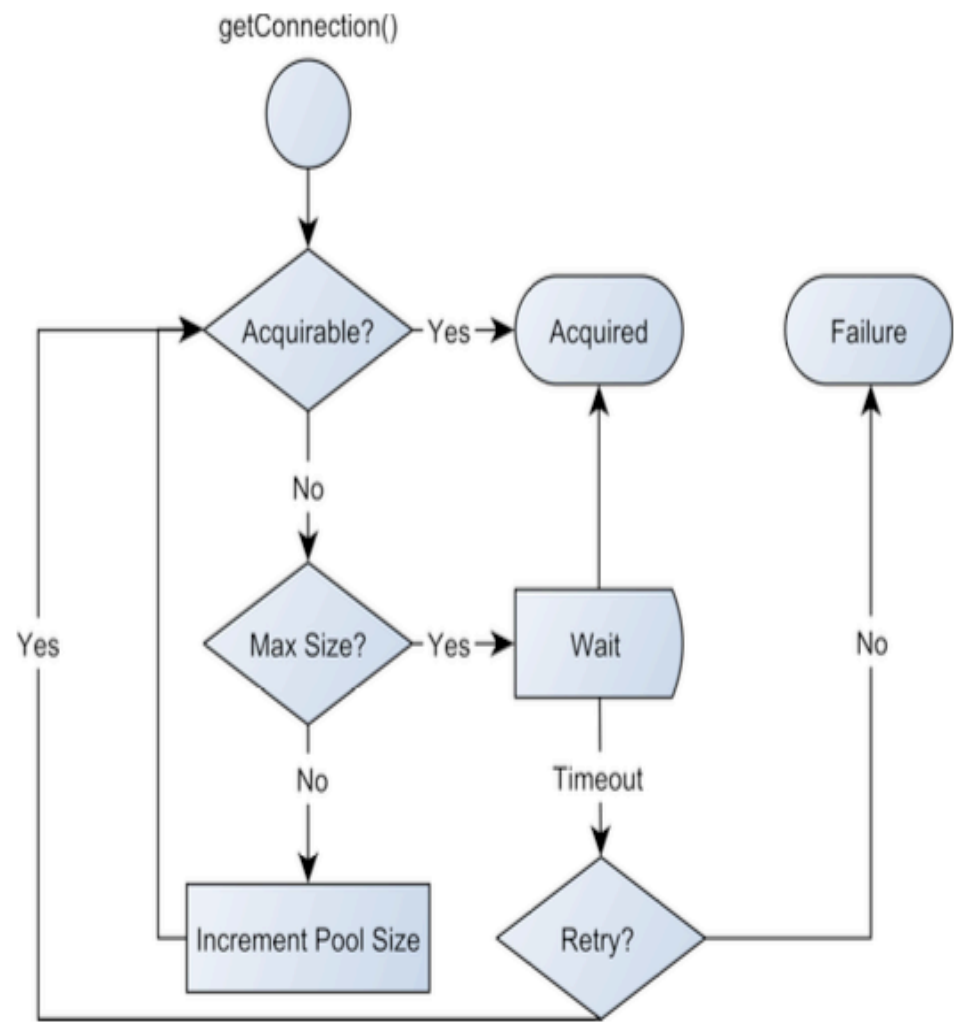


Figure 3.6: Connection acquisition request flow

Pooling cycle de vie

Contrairement à une utilisation sans pool, où la connexion physique est retournée telle quelle, un pool de connexions fournit un proxy.

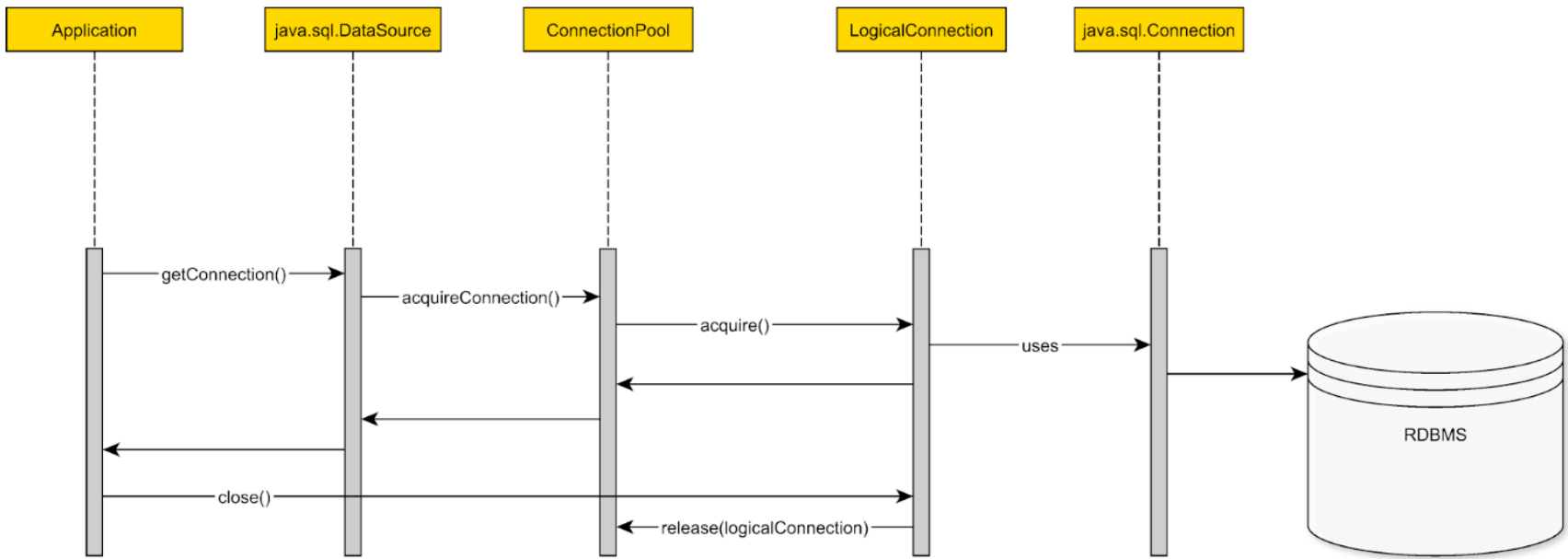


Figure 3.7: DataSource connection

Pooling cycle de vie

Ce proxy permet notamment de détourner le comportement des méthodes `getConnection()` et `close()`.

- Lorsqu'une connexion est empruntée, le pool marque la connexion comme « allouée » afin d'empêcher son utilisation simultanée par plusieurs consommateurs.
- À l'appel de `close()`, le proxy intercepte l'opération et notifie le pool, qui remet alors la connexion dans l'état « non allouée », la rendant à nouveau disponible sans fermer la connexion physique sous-jacente.

Pooling exemple

```
BasicDataSource dataSource = new BasicDataSource();
dataSource.setUrl("url");
dataSource.setUsername("username");
dataSource.setPassword("password");
dataSource.setInitialSize(5);  // Initialiser 5 connexions physiques

Connection connection = dataSource.getConnection(); // Demande une connexion physique
...
connection.close(); // La connection ne sera pas fermée MAIS rendu dans le pool (la marque disponible)
```