JAVA DATABASE CONNECTIVITY (JDBC)

O modalitate de a se asigura persistența datelor în cadrul unei aplicații o reprezintă utilizarea unei *baze de date*. O bază de date este gestionată de un sistem de gestiune a bazelor de date (SGBD) dedicat, de obicei aflat pe un server, astfel încât baza de date poate fi utilizată, în mod independent și transparent, de mai multe aplicații, posibil implementate în limbaje de programare diferite.

Java DataBase Connectivity (JDBC) este un API dedicat accesării bazelor de date din cadrul unei aplicații Java, care permite conectarea la un server de baze de date, precum și executarea unor instrucțiuni SQL. Accesarea unei baze de date din cadrul unei aplicații Java se realizează într-o manieră transparentă, independentă de sistemul de gestiune al bazelor de date utilizat. Practic, pentru fiecare SGBD există un driver dedicat (un program instalat local) care transformă cererile efectuate din cadrul programului Java în instrucțiuni care pot fi înțelese de către SGBD-ul respectiv (Fig. 1). Există mai multe tipuri de drivere disponibile, însă, în prezent, cele mai utilizate sunt *driverele native Java*. Acestea sunt scrise complet în limbajul Java și folosesc socket-uri pentru a comunica direct cu o bază de date, obținându-se astfel o performanță ridicată din punct de vedere al timpului de executare.

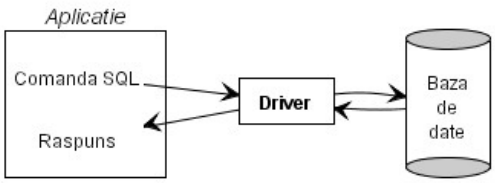

Figura 1. Comunicarea dintre o aplicație Java și un SGBD (https://profs.info.uaic.ro/~acf/java/slides/ro/jdbc_slide.pdf)

Procedura de instalare a unui drivere poate fi diferită de la un driver la altul. De exemplu, în cazul driverului MySQL, driverul este o arhivă de tip jar. Într-un mediu de dezvoltare, driverul poate fi specificat sub forma unei biblioteci atașată proiectului sau poate fi deja disponibil (de exemplu, versiunile noi de NetBeans conțin suport implicit pentru MySQL). Astfel, JDBC dispune de clasa `DriverManager` care administrează încărcarea driverelor, precum și obținerea conexiunilor către baza de date (Fig. 2). Odată conexiunea deschisa, JDBC oferă clientului un API care nu depinde de softul de baze de date folosit, ceea ce facilitează eventuale migrări între diferite SGBD-uri. Cu alte cuvinte, nu este necesar să scriem un program pentru a accesa o bază de date Oracle, alt program pentru a accesa o bază de date Sybase etc., ci este suficient să scriem un singur program folosind API-ul JDBC, iar acesta va fi capabil să comunice cu drivere diferite, trimițând secvențe SQL către baza de date dorită.

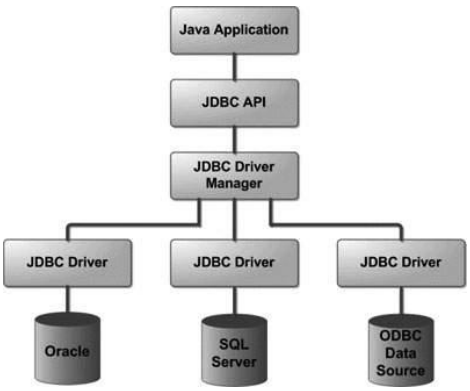

Figura 2. JDBC Driver Manager (https://www.tutorialspoint.com/jdbc/jdbc-introduction.htm)

Arhitectura JDBC

Nucleul JDBC conține o serie de clase și interfețe aflate în pachetul `java.sql`, precum:

- Clasa `DriverManager`**:** gestionează driver-ele JDBC instalate și alege driver-ul potrivit pentru realizarea unei conexiuni la o bază de date;
- Interfața `Connection`**:** gestionează o conexiune cu o bază de date (orice comandă SQL este executată în contextul unei conexiuni);
- Interfețele `Statement / PreparedStatement / CallableStatement`**:** sunt utilizate pentru a executa comenzi SQL în SGBD sau pentru a apela proceduri stocate;
- Interfața `ResultSet`**:** stochează sub forma tabelară datele obținute în urma executării unei comenzi SQL;
- Clasa `SQLException`**:** utilizată pentru tratarea erorilor specifice JDBC.

Etapele realizării unei aplicații Java folosind JDBC

1. Încărcarea driver-ului specific

Prima etapă constă din înregistrarea, pe mașina virtuală unde rulează aplicația, a driver-ului JDBC necesar pentru comunicarea cu baza de date respectivă. Acest lucru presupune încărcarea în memorie a clasei care implementează driver-ul și poate fi realizată prin apelul metodei statice `void forName(String driver)` din clasa `Class`. De exemplu, încărcarea unui driver pentru o conexiune cu MySQL se poate realiza prin `Class.forName("com.mysql.jdbc.Driver")`, iar pentru o conexiune cu Oracle prin `Class.forName("oracle.jdbc.OracleDriver")`.

Începând cu versiunea JDBC 4.0, inclusă în Java SE 6, acest pas nu mai este obligatoriu, deoarece, la prima încercare de conectare la o bază de date, mașina virtuală Java va încărca automat toate driver-ele disponibile (pe care le găsește în *class path*).

2. Stabilirea unei conexiuni cu o bază de date

După înregistrarea unui driver JDBC, acesta poate fi utilizat pentru a stabili o conexiune cu o bază de date de tipul respectiv. O *conexiune* (sesiune) la o bază de date reprezintă un context prin care sunt trimise secvențe SQL din cadrul aplicației către SGBD și sunt primite înapoi rezultatele obținute.

Având în vedere faptul ca pot exista mai multe drivere încărcate în memorie, se va specifica, pe lângă un identificator al bazei de date, și driverul care trebuie utilizat. Acest lucru se realizează prin intermediul unei adrese specifice, numită JDBC URL, având formatul `jdbc:sub-protocol:identificator`, unde:

- câmpul `sub-protocol` specifică tipul de driver care va fi utilizat (de exemplu `sqlserver`, `mysql`, `postgresql` etc.);
- câmpul `identificator` specifică adresa unei mașini gazdă (inclusiv un număr de port), numele bazei de date și, eventual, numele utilizatorului și parola sa.

De exemplu, pentru o conexiune cu o bază de date denumită `BD`, care este stocată local folosind SGBD-ul MySQL, poate fi utilizat URL-ul de tip JDBC `jdbc:mysql://localhost:3306/BD`, iar dacă s-ar utiliza SGBD-ul Oracle, atunci URL-ul ar putea fi `jdbc:oracle:thin:@localhost:1521:BD`.

Deschiderea unei conexiuni se realizează prin intermediul metodelor statice `Connection getConnection(String url)` sau `Connection getConnection(String url, String user, String password)` din clasa `DriverManager`. Ambele metode returnează un obiect de tip `Connection`, clasă care conține o serie de metode pentru a gestiona conexiunea cu baza de date.

Exemplu:

- Deschiderea unei conexiuni la baza de date `Firma`, găzduită local utilizând `MySQL`:

```
Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/Firma");
```
sau
```
Connection con=DriverManager.getConnection ("jdbc:mysql://localhost:3306/Firma",
                                                    "popescuion","12345");
```

- Deschiderea unei conexiuni la baza de date `Firma`, găzduită local utilizând `Apache Derby`:

```
Connection con=DriverManager.getConnection("jdbc:derby://localhost:1527/Firma");
```
sau
```
Connection con=DriverManager.getConnection ("jdbc:derby://localhost:1527/Firma",
                                                    "popescuion","12345");
```

La primirea unui URL de tip JDBC, DriverManager-ul va parcurge lista driverelor încărcate în memorie (de exemplu, începând cu JDK 6, driver-ul pentru Apache Derby este încărcat automat), până când unul dintre ele va recunoaște URL-ul respectiv. Dacă nu există nici un driver potrivit, atunci va fi lansată o excepție de tipul `SQLException`, cu mesajul "`No suitable driver found for ...`".

3. Crearea unui obiect de tip `Statement`

După realizarea unei conexiuni cu o bază de date, acesta poate fi folosită pentru executarea unor comenzi SQL (interogarea sau actualizarea bazei de date), precum și pentru extragerea unor informații referitoarea baza de date (meta-date).

Obiectele de tip `Statement` sunt utilizate pentru a executa instrucțiuni SQL (interogări, actualizări ale datelor sau modificări ale structurii) în cadrul unei conexiuni.

JDBC pune la dispoziția programatorului 3 tipuri de statement-uri, sub forma a 3 interfețe:
- `Statement` – pentru comenzi SQL simple, fără parametri;
- `PreparedStatement` – pentru comenzi SQL parametrizate;
- `CallableStatement` – pentru apelarea funcțiilor sau procedurilor stocate.

Interfața `Statement`

Crearea unui obiect `Statement` se realizează apelând metoda `Statement createStatement()` pentru un obiect de tip `Connection`: `Statement stmt = con.createStatement()`.

Executarea unei secvențe SQL poate fi realizată prin intermediul următoarelor metode:

a) metoda `ResultSet executeQuery(String sql)` – este folosită pentru executarea interogărilor de tip `SELECT` și returnează un obiect de tip `ResultSet` care va conține rezultatul interogării sub o formă tabelară, precum și meta-datele interogării (de exemplu, denumirile coloanelor selectate, numărul lor etc.).

Exemplu
Extragerea datelor despre o persoană stocate în tabela `Angajati` din baza de date `Firma`:
```
String sql = "SELECT * FROM Angajati";
ResultSet rs = stmt.executeQuery(sql);
```

Pentru a parcurge înregistrările rezultate în urma unei interogări de tip `SELECT`, un obiect de tip `ResultSet` utilizează un cursor, poziționat inițial înaintea primei linii. În clasa `ResultSet` sunt definite mai multe metode pentru a muta cursorul în cadrul structurii tabelare, în scopul parcurgerii sale: `boolean first()`, `boolean last()`, `boolean next()`, `boolean previous()`. Toate cele 4 metode întorc valoarea `true` dacă mutarea cursorului a fost efectuată cu succes sau `false` în caz contrar.

Pentru a extrage informațiile de pe fiecare linie se utilizează metode de forma `TipData getTipData(int coloană)` sau `TipData getTipData(String coloană)`, unde `TipData` reprezintă tipul de dată al unei coloane, iar argumentul `coloană` indică fie numărul de ordine din cadrul tabelului (începând cu 1), fie numele acesteia.

Exemplu:
Afișarea datelor angajaților stocate în tabela `Angajati` din baza de date `Firma`:
```
while(rs.next())
   System.out.println(rs.getString("Nume") + " " + rs.getInt("Varsta") + " "
   + rs.getDouble("Salariu"));
```

b) metoda `int executeUpdate(String sql)` – este folosită pentru executarea unor interogări SQL de tipul Data Manipulation Language (DML), care permit actualizări ale datelor de tipul `UPDATE/INSERT/DELETE`, sau de tipul Data Definition Language (DDL) care permit manipularea structurii bazei de date (de exemplu, `CREATE/ALTER/DROP TABLE`). Metoda returnează numărul de linii modificate în urma efectuării unor interogări de tip DML sau 0 în cazul interogărilor de tip DDL.

Exemple:
```
String qrySQL = "INSERT INTO Angajati VALUES('1234567890999',
                                                'Albu Ioan',3210.10)";
```
sau
```
String qrySQL = "UPDATE Angajati SET Salariu = 1.10*Salariu
                                                WHERE Salariu <= 2500";
```
sau
```
String qrySQL = "DELETE FROM Angajati WHERE Nume LIKE 'Geo%'";

int n = stmt.executeUpdate(qrySQL);
System.out.println("Au fost modificate " + n + " înregistrări!");
```

Interfața `PreparedStatement`

Crearea unui obiect de tip `PreparedStatement` se realizează apelând metoda `PreparedStatement prepareStatement(String sql)` pentru un obiect de tip `Connection` și primește ca argument o instrucțiune SQL cu unul sau mai mulți parametri. Fiecare parametru este specificat prin intermediul unui semn de întrebare (?). Obiectele de tip `PreparedStatement` sunt utilizate în cazul în care este necesară executarea repetată a unei interogări SQL, eventual cu valori diferite ale parametrilor, deoarece aceasta va fi precompilată, deci se va executa mai rapid.

Exemplu:
```
String sql = "UPDATE persoane SET nume=? WHERE cod=?";
Statement pstmt = con.prepareStatement(sql);
```

Obiectul `pstmt` conține o instrucțiune SQL precompilată care este trimisă către baza de date, însă pentru a putea fi executată este necesară stabilirea valorilor pentru fiecare parametru. Setarea valorilor parametrilor se realizează prin metode de tip `void setTipData(int index, TipData valoare)`, unde `TipData` este tipul de date corespunzător parametrului respectiv, iar prin argumentele metodei se specifică indexul parametrului (începând de la 1) și valoarea pe care dorim să i-o atribuim.

Exemplu:
```
String sql = "UPDATE persoane SET nume=? WHERE cod=?";
Statement pstmt = con.prepareStatement(sql);
pstmt.setString(1, "Ionescu");
pstmt.setInt(2, 45);
pstmt.executeUpdate();
```

Executarea unei instrucțiuni SQL folosind un obiect de tip `PreparedStatement` se realizează apelând una dintre metodele `ResultSet executeQuery()` sau `int executeUpdate()`, asemănătoare cu cele definite pentru un obiect de tip `Statement`.

Interfața `CallableStatement`

Această interfață este utilizată pentru executarea subprogramelor atașate unei baze de date, respectiv funcții și proceduri stocate. Diferențele dintre funcții și proceduri stocate sunt următoarele:

- procedurile sunt folosite pentru a efectua prelucrări în baza de date (de exemplu, operații de actualizare), în timp ce funcțiile sunt folosite pentru a efectua calcule (de exemplu, pentru a determina numărul de angajați care au salariul maxim);
- procedurile nu returnează nimic prin numele lor (dar pot returna mai multe valori prin parametrii de intrare-ieșire, într-un mod asemănător funcțiilor de tip `void` din C/C++), în timp ce funcțiile returnează o singură valoare (într-un mod asemănător funcțiilor din C/C++ care returnează un tip de date primitiv, diferit de `void`);
- procedurile pot avea parametrii de intrare, de ieșire și de intrare-ieșire, în timp ce funcțiile pot avea doar parametrii de intrare;
- funcțiile pot fi apelate în proceduri, dar invers nu.

Funcțiile și procedurile stocate sunt utilizate într-o bază de date pentru a efectua calcule sau prelucrări complexe. De exemplu, se poate implementa o funcție stocată care să calculeze profitul mediu adus de contractele pe care o firmă le are cu o altă firmă într-un anumit interval calendaristic sau se poate implementa o procedură stocată care să efectueze prelucrări complexe ale mai multor tabele.

Exemple:

a) o funcție stocată care calculează suma salariilor tuturor bărbaților sau tuturor femeilor dintr-o firmă, în raport de valoarea parametrului `Tip`:

```
CREATE FUNCTION totalSalarii(Tip VARCHAR(1)) RETURNS double
BEGIN
   DECLARE total DOUBLE;
   DECLARE aux CHAR;

   IF(Tip = 'B') THEN
        SET aux = '1';
   ELSE
        SET aux = '2';
   END IF;

   SELECT SUM(Salariu) INTO total FROM Angajati WHERE LEFT(CNP,1) = aux;
   RETURN total;
END
```

b) o procedură stocată care verifică dacă un angajat există în tabela `Angajați` (pe baza `CNP`-ului) și în raport de rezultatul obținut inserează sau actualizează datele sale:

```
CREATE PROCEDURE inserareAngajat(IN CNP VARCHAR(13), IN Nume VARCHAR(45),
                                          IN Salariu DOUBLE, OUT rezultat INT)
BEGIN
   DECLARE cnt INT;
   SELECT COUNT(*) INTO cnt FROM angajati WHERE angajati.CNP = CNP;

   IF(cnt = 0) THEN
      INSERT INTO Angajati VALUES(CNP,Nume,Salariu);
      SET rezultat = 1;
   ELSE
      UPDATE Angajati SET Angajati.Nume = Nume, Angajati.Salariu = Salariu
                                                        WHERE Angajati.CNP =CNP;
      SET rezultat = 2;
   END IF;
END
```

Apelarea funcții stocate `totalSalarii` definită mai sus necesită efectuarea următorilor pași:

- se creează un obiect `sfunc` de tip `CallableStatement` folosind un obiect `conn` de tip `Connection`:

```
sfunc = conn.prepareCall("{?=call totalSalarii(?)}");
```

Primul ? reprezintă valoarea returnată de funcție (“parametrul de ieșire”), iar cel dintre paranteze reprezintă parametrul de intrare. Dacă funcția ar fi avut mai mulți parametri de intrare, atunci se punea câte un ? pentru fiecare, de exemplu `funcție(?,?,?)`.

- se specifică tipul rezultatului întors de funcție - se spune că “se înregistrează parametrul de ieșire (valoarea returnată de funcție)”:

```
sfunc.registerOutParameter(1, Types.DOUBLE);
```

Valoarea 1 identifică primul ? din apelul metodei `prepareCall` de mai sus!

- se setează valorile parametrilor de intrare, folosind metode de tipul `setTipData`, asemănătoare cu cele definite pentru `PreparedStatement`:

```
sfunc.setString(2, "B");
```

Deoarece valoarea 1 identifică valoarea returnată de funcție, parametrii de intrare sunt numerotați de la 2!

- se execută funcția stocată:

```
sfunc.execute();
```

- se preia rezultatul întors de funcția stocată, folosind metode de tipul `getTipData(int index_parametru_de_intrare)`:

```
double total = sfunc.getDouble(1);
```

Valoarea parametrului este 1 deoarece rezultatul întors de funcție se identifică prin numărul de ordine 1!

Apelarea procedurii stocate `inserareAngajat` definită mai sus necesită efectuarea următorilor pași:

- se creează un obiect `sproc` de tip `CallableStatement` folosind un obiect `conn` de tip `Connection`:

```
sproc = conn.prepareCall("{call inserareAngajat(?,?,?,?)}");
```

Semnele de întrebare dintre paranteze reprezintă parametrii de intrare/ieșire/intrare-ieșire ai procedurii.

- se specifică tipurile parametrilor de ieșire ai procedurii:

```
sproc.registerOutParameter(4, Types.DOUBLE);
```

- se setează valorile parametrilor de intrare, folosind metode de tipul `setTipData`:

```
sproc.setString(1, "1234567890999");
sproc.setString(2, "Vasilescu Ion");
```

```
sproc.setDouble(3, 3333.33);
```

- se execută procedura stocată:

```
sproc.execute();
```

- se preiau eventualele rezultatele întoarse de procedura stocată, folosind metode de tipul `getTipData(int index_parametru_de_ieșire)`:

```
double rezultat = sproc.getInt(4);
```

4. Închiderea unei conexiuni cu o bază de date

Conexiunea cu o bază de date se închide utilizând metoda `void close()` din clasa `Connection`, dacă nu este utilizat un bloc de tip `try-with-resources`.