

JDBC adatbázis-hozzáférés Java-ból

Áttekintés

- ▶ Bevezetés
- ▶ Kapcsolat objektumok
- ▶ SQL parancs küldés
- ▶ Válasz feldolgozása
- ▶ Tranzakciók
- ▶ DAO tervezési minta

JDBC

JDBC API:

Java osztályokat és interfészeket tartalmazó csomagok (java.sql, javax.sql), melyek egy standard API-t biztosítanak adatbázist használó alkalmazások valamint adatbázis-keretrendszerek fejlesztésére.

- ▶ A JDBC API előnye abban áll, hogy elvileg bármilyen adatbázist elérhetünk vele bármilyen platformról, melyen a Java virtuális gép fut.
- ▶ Nem kell tehát minden adatbázisszerverre külön adatbázis hozzáférési réteget írni, hanem ugyanaz a kód működni fog bármely adatbázisszerverrel (feltéve, hogy nem használunk adatbázis-specifikus SQL kiterjesztéseket).

JDBC meghajtó szolgáltatók, lásd:

<http://developers.sun.com/product/jdbc/drivers>

A JDBC alkalmazása

mire használható a JDBC:

1. Kapcsolat objektumok lekérése egy adatforrástól (data source)
2. Lekérdező (select) adat-módosító (insert, update, delete) valamint adatbázis struktúra módosító parancsokat lehet küldeni az adatbázisnak (azaz bármilyen érvényes SQL parancsot).
3. A lekérdezés eredményét fel lehet dolgozni

Adatbáziskapcsolat objektumok (Connection)

Pl.

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/MyDB");
Connection con =
    ds.getConnection("myLogin", "myPassword");

Statement stmt = con.createStatement();
ResultSet rs =
    stmt.executeQuery("SELECT a, b, c FROM Table1");

while (rs.next()) {
    int x = rs.getInt("a");
    String s = rs.getString("b");
    float f = rs.getFloat("c");
}
```

- ▶ Egy Connection objektum egy adatbáziskapcsolatnak felel meg.
- ▶ Egy adatbáziskapcsolat lehetővé teszi SQL parancsok hívását, illetve az eredmények kinyerését a kapcsolat-objektumon keresztül.
- ▶ Egy alkalmazás használhat egy vagy több kapcsolatot egy vagy több adatbázishoz.
- ▶ A kapcsolathoz tartozó adatbázisról a `Connection.getMetaData()` metódussal kaphatunk információt.
- ▶ Ez egy `DatabaseMetaData` objektumot ad vissza, amelyik az adatbázistáblákról, tárolt eljárásokról, a kapcsolat tulajdonságairól szolgáltat információt.

pl.

TypeInfo, JDBCMeta, DatabaseInfoServlet (servlet)

Kapcsolat létrehozása

Két módon történhet:

1. DriverManager
2. DataSource

1. DriverManager (régebbi)

- ▶ A `DriverManager.getConnection()` metódusát használjuk, mely paraméterként egy URL-t kap.
- ▶ A `DriverManager` osztály tartalmaz egy listát a regisztrált driverekkel.
- ▶ A `getConnection()` metódus hívásakor megpróbálja megtalálni a megfelelő drivert, mely kapcsolódni tud az URL-ben megadott adatbázishoz (sorba kiprobálja a drivereket, míg egyet talál, amely kapcsolódik a megfelelő URL segítségével)
- ▶ Ezt a manager-szintet el lehet kerülni közvetlen `Driver` metódus hívásával. (csak ritkán használjuk, pl. ha két driver is van, amelyik hozzá tud kapcsolódni egy bizonyos adatbázishoz és explicit szeretnénk meghatározni, hogy melyikkel akarunk kapcsolódni.)

Pl.:

```
Class.forName("jdbc.odbc.JdbcOdbcDriver");
//betölti a driver-t a memóriába
String url = "jdbc:odbc:myDatabase";
Connection con = DriverManager.getConnection(url,
    "myUsername", "myPassword");
```

lásd: pl.

driverManager

2. DataSource

A DataSource interfész a DriverManager alternatívájaként egy kapcsolat létrehozásának az inkább ajánlott módja.

Előnyei:

- ▶ a DataSource-al létrehozott kapcsolatok részt vehetnek "connection pooling"-ban (kapcsolatobjektum készlet) valamint osztott tranzakciókban.
- ▶ a DataSource objektum JNDI-n keresztül is lekérhető, és az alkalmazástól függetlenül lehet telepítve és létrehozva:
 - ▶ a JDBC driver tartalmaz egy DataSource implementációt, a rendszeradminisztrátor regisztrálja ezt a JNDI névszolgáltatóval, az alkalmazás pedig a JNDI szerveren regisztrált DataSource-ot egyszerűen lekéri név alapján.
(Az alkalmazás tehát nem kell driver információkat hardkódoljon, hanem egy logikai nevet használ a DataSource eléréséhez, így a DataSource megváltoztatható az alkalmazáskód módosítása nélkül)

Példák:

lásd: pl.

- ▶ dataSource.DataSourceTableView (DataSource közvetlen létrehozása -nem tipikus-),
- ▶ metaData.DatabaseInfoServlet (kapcsolatobjektum-készletet –"pooling"– használó DataSource)

Datasource létrehozása és regisztrálása

A kód csak illusztrálásként szolgál, tipikusan az alkalmazásszerver konfigurációs fájljában állítjuk be. (Ehhez hasonló kód az alkalmazásszerverben fut le.)

```
VendorDataSource vds = new VendorDataSource();
```

```
vds.setServerName("my_database_server");  
vds.setDatabaseName("my_database");
```

```
Context ctx = new InitialContext();  
ctx.bind("jdbc/MyDB", vds);
```

lásd: pl.

jndi.TestDSBind (NEM tipikus beállítás), jndi.TestDSLookup

Kapcsolódás a DataSource-hoz

```
Context ctx = new InitialContext();  
DataSource ds = (DataSource)ctx.lookup("jdbc/MyDB");  
Connection con = ds.getConnection("myUsername",  
"myPassword");
```

A DataSource interfész implementálása háromféle lehet:

1. Alap DataSource osztály: a driver szolgáltató (vendor) adja
2. DataSource osztály, amelyik "connection pooling"-ot szolgáltat: alkalmazásszerver- vagy driverszolgáltató adja.
3. DataSource osztály, amelyet osztott tranzakciókban használhatunk: az alkalmazásszerver szolgáltató adja (pl. EJB konténer szolgáltató).

- ▶ Egy alap DataSource-al létrehozott kapcsolat-objektum –akárcsak a DriverManager-el létrehozott– egy fizikai kapcsolat objektumot hoz létre
- ▶ "Connection pooling"-ot implementáló DataSource viszont csak egy PooledConnection objektumot ad vissza, amely nem közvetlenül egy fizikai kapcsolat objektum.
Lásd pl.: <http://commons.apache.org/dbcp/> (szabadon használható adatbáziskapcsolat-készlet implementáció)
- ▶ Az alkalmazás kódja ugyanúgy használja fel a kapcsolatobjektumot, függetlenül attól, hogy DataSource-ból vagy DriverManager-ből hozzuk-e létre, illetve hogy "pool"-t használ vagy sem.
- ▶ Lényeges, hogy az alkalmazáskód egy finally block-ot kell tartalmazzon, biztosítva ezáltal, hogy a kapcsolat bezáródik akkor is, ha hiba történt (kivétel dobódott).
- ▶ Ez még fontosabb készletet (pool) használó kapcsolatok esetén, hogy az illető kapcsolatot visszajuttassuk a pool-ba az újból rendelkezésre álló kapcsolatok közé.

Pl.

```
try {
    Connection con = ds.getConnection("user", "secret");
    // ...üzleti logika kod
} catch {
    // SQLException-t kezelo kod
} finally {
    if (con != null)
        con.close();
}
```

- ▶ Amint a kapcsolat létrejött, az adatbázisnak SQL parancsokat küldhetünk.
- ▶ A JDBC API nem korlátoz a kiadható SQL parancsok tekintetében: bármilyen parancs használható, amit a JDBC driver elfogad.
- ▶ Azt azonban biztosítanunk kell, hogy az adatbázis fel tudja dolgozni a parancsokat.
Pl. hívhatunk tárolt eljárásokat egy olyan adatbázisra, amelyik nem támogatja ezeket, de egy kivétel fog dobódni.

A JDBC API három interfészt biztosít SQL parancsok küldésére:

1. Statement
2. PreparedStatement
3. CallableStatement

Statement

Statement

- ▶ A kapcsolat objektum `createStatement()` metódusával hozhatjuk létre.
- ▶ Ezt paraméter nélküli SQL parancsok hívása használja

```
Statement stmt = con.createStatement();
//lekérdezés: SELECT
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
//módosítás: INSERT, UPDATE, DELETE ES DDL (CREATE TABLE, DROP TABLE)
int affectedRows = stmt.executeUpdate("UPDATE...");
stmt.execute(...)
```

pl.

`basic.BasicSample, basic.Join, execute.ExecuteSQL`

PreparedStatement

- ▶ A kapcsolat objektum `prepareStatement()` metódusával hozhatjuk létre.
- ▶ Előfordított (precompiled) SQL parancsok hívására használjuk.

Előnyök a Statement-hez képest:

- ▶ egy vagy több paramétert adhatunk meg neki.
- ▶ hatékonyabb, mert le lesz fordítva és ez el lesz mentve. Többszöri felhasználás esetén érdemes tehát ezt használni.

Mielőtt futtatnánk a PreparedStatement-et, mindegyik paraméternek értéket kell adjunk:

```
PreparedStatement pstmt = con.prepareStatement(  
    "UPDATE table1 SET name = ? WHERE id = ?");  
pstmt.setString(1, "Joe");  
pstmt.setLong(2, 1000);  
ResultSet rs = pstmt.executeQuery();
```

- ▶ Kiterjeszti a Statement interfészt, tehát örökli ennek metódusait, viszont saját verziókat definiál az `executeQuery`, `executeUpdate` és `execute` metódusokra.
- ▶ Mivel a Statement objektumok nem tartalmazzák az SQL parancsot, ezért paraméterként adjuk meg ezt a fenti metódusoknak.
- ▶ A PreparedStatement objektumok nem adják át paraméterként az SQL parancsokat ezeknek a metódusoknak, mivel ezek már tartalmazzák az SQL parancs előre lefordított változatát.
- ▶ Kivétel dobódik, ha PreparedStatement objektum esetén SQL parancsot adunk át paraméterként ezeknek az örökölt metódusoknak

CallableStatement

- ▶ A kapcsolat objektum `prepareCall` metódusával hozhatjuk létre
- ▶ Tárolt eljárások hívására használjuk.

Kötegelt módosítások (Batch Updates)

- ▶ Egy Statement objektum több módosító parancsot egy egységként (kötegelve) küldhet a szervernek.
- ▶ Ez bizonyos esetekben lényeges teljesítménynövekedéshez vezethet.

Pl.

```
Statement stmt = con.createStatement();
con.setAutoCommit(false);

stmt.addBatch("INSERT INTO employees VALUES (1000, 'Joe Jones')");
stmt.addBatch("INSERT INTO departments VALUES (260, 'Shoe')");
stmt.addBatch("INSERT INTO emp_dept VALUES (1000, '260')");

int [] updateCounts = stmt.executeBatch();
```

- ▶ Egy kötegen belül mindegyik SQL parancs módosító kell legyen.
- ▶ Hiba esetén BatchUpdateException kivétel dobódik vagy nem dobódik kivétel és tovább fut a köteg, de az updateCounts megfelelő elemére Statement.EXECUTE_FAILED íródik. Ez JDBC driver függő. Az eredménytől függően commit vagy rollback következhet.
- ▶ A JDBC driver nem kötelező, hogy implementálja a kötegelt módosításokat. A DatabaseMetaData supportsBatchUpdates tulajdonsága alapján lekérdezhetjük.

pl.

coffee.CoffeeBatchInsert – kötegelt módosítások (beszúrások)

ResultSet

- ▶ A ResultSet egy Java objektum, amelyik egy SQL lekérdezés eredményét tartalmazza.
- ▶ A különböző mezőkhöz egy soron belül set és get metódusokkal férünk hozzá és a next metódussal megyünk a következő sorra.

Pl.

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table");
while (rs.next()) {
    // az aktualis rekord ertekeinek kinyerese es kiirasa
    int i = rs.getInt("a");
    String s = rs.getString("b");
    float f = rs.getFloat("c");
    System.out.println("ROW = " + i + " " + s + " " + f);
}
```

Kurzorok

- ▶ A ResultSet objektum tartalmaz egy kurzort, amelyik az aktuális sorra mutat.
- ▶ A ResultSet objektum létrehozásakor a kurzor az első sor elé van beállítva, és a next metódus első hívása beállítja az első elemre.
- ▶ Gördíthető ResultSet-ek esetében több metódust használhatunk:
 - ▶ previous,
 - ▶ first,
 - ▶ last,
 - ▶ absolute,
 - ▶ relative,
 - ▶ afterLast,
 - ▶ beforeFirst

pl.

coffee.ScrollableResultSet

ResultSet típusok

- ▶ `TYPE_FORWARD_ONLY`: csak előre gördíthető
- ▶ `TYPE_SCROLL_INSENSITIVE`: előre-hátra gördíthető vagy egy konkrét pozícióra állítható
- ▶ `TYPE_SCROLL_SENSITIVE`: ezen kívül érzékeli az adatváltozásokat, amelyek azóta történtek, amióta kinyitottuk a `ResultSet`-et.

Konkurencia típusok

- ▶ `CONCUR_READ_ONLY`: nem módosítható, read-only lock-okat használ, tehát több felhasználó is hozzáférhet az adatokhoz egyidőben (read-only lock akárhány lehet ugyanarra az adatra)
- ▶ `CONCUR_UPDATABLE`: a `ResultSet` módosítható, tehát a módosított adatokat visszaírja az adatbázisba (write-only lock-okat használ azaz csak egy felhasználó fér egyidőben hozzá az adatokhoz)

pl.

`resultSet.ResultSetUpdate`

Példa különböző típusú ResultSet-ek létrehozására:

```
Statement stmt = con.createStatement(  
    ResultSet.TYPE_SCROLL_SENSITIVE,  
    ResultSet.CONCUR_UPDATABLE);  
ResultSet rs = stmt.executeQuery(  
    "SELECT EMP_NO, SALARY FROM EMPLOYEES");  
  
PreparedStatement pstmt = con.prepareStatement(  
    "SELECT EMP_NO, SALARY FROM EMPLOYEES WHERE EMP_NO = ?",  
    ResultSet.TYPE_SCROLL_SENSITIVE,  
    ResultSet.CONCUR_UPDATABLE);  
pstmt.setString(1, "1000010");  
ResultSet rs = pstmt.executeQuery();
```

Módosítások

Csak `CONCUR_UPDATABLE` típusú `Statement` esetében használható.

P1.1 – Módosítás:

```
rs.absolute(4);  
rs.updateString(2, "321 Kasten");  
rs.updateFloat(3, 10101.0f);  
rs.updateRow();
```

P1.2 – Módosítás:

```
rs.absolute(4);  
rs.updateString("ADDRESS", "321 Kasten");  
rs.updateFloat("AMOUNT", 10101.0f);  
rs.updateRow();
```

P1. – Törlés:

```
rs.first();  
rs.deleteRow();
```

P1. – Beszúrás:

```
rs.moveToInsertRow();  
rs.updateObject(1, myArray);  
rs.updateInt(2, 3857);  
rs.updateString(3, "Mysteries");  
rs.insertRow();  
rs.first();
```

- ▶ Egy tranzakció egy vagy több parancsból áll, amelyek lefutottak és vagy mind sikeresen el lett végezve (*commit*) vagy visszagördültek (*roll back*).
- ▶ Mikor egy *commit* vagy *rollback* hívódik az aktuális tranzakció befejeződik és egy új kezdődik.

Egy új kapcsolat (Connection) objektum alapértelmezésben általában *auto-commit* módban van, ami azt jelenti, hogy a *commit* metódus automatikusan meghívódik a parancs lefutását követően, azaz a tranzakció egyetlenegy SQL parancsból áll.

- ▶ A kapcsolat objektumok részt vehetnek osztott tranzakciókban is (több adatbázisszervert magukba foglaló tranzakciók).
- ▶ Ehhez azonban a kapcsolat objektumot kötelezően egy DataSource objektumból kell kinyerjük, amely úgy van implementálva, hogy együttműködjön egy alkalmazásszerver osztott tranzakciós infrastruktúrájával.
- ▶ Ellentétben a DriverManager-ből létrehozott kapcsolatokkal, az ilyen DataSource által létrehozott kapcsolatoknak az auto-commit módja alapértelmezésben ki van kapcsolva.
- ▶ (A DataSource standard implementációja viszont ugyanolyan kapcsolatobjektumokat hoz létre, mint amelyet a DriverManager osztály.)

- ▶ Ha a kapcsolat objektum osztott tranzakcióban vesz részt, a tranzakció manager határozza meg, hogy a commit ill. rollback metódusok mikor lesznek meghívva.
- ▶ Ilyenkor tehát nem hívhatjuk meg közvetlenül ezeket a metódusokat valamint nem állíthatjuk az auto-commit módot, mert keresztbe teszünk a tranzakció manager-nek.

Tranzakciós elszigetelési szintek

- ▶ Ha egy adatbázisszerver támogatja a tranzakciókezeléseket, mód van arra, hogy potenciális konfliktusokat elkerüljön, melyek abból adódnak, hogy két vagy több tranzakció fut az adatbázison egyidőben.
- ▶ A kapcsolat objektumnak beállíthatjuk a tranzakciós elszigetelési szintjét, ami megadja, hogy az adatbázisszerver milyen szinten gondoskodik a potenciális konfliktusok megoldásáról.
PI. mi történjen akkor, ha egy tranzakció megváltoztat egy értéket, és egy másik olvassa azt mielőtt az befejeződött commit-al vagy rollback-el?

Hogy ezt megengedjük, beállíthatjuk a megfelelő szintet:

```
con.setTransactionIsolation(TRANSACTION_READ_UNCOMMITTED);
```

- ▶ Minél magasabb az elszigetelési szint, annál nagyobb hangsúly lesz fektetve a konfliktusok elkerülésére, viszont annál lassúbb lesz a szerver (a megnövekedett zárok (locks) kezelése valamint a felhasználók csökkentett párhuzamos hozzáférése miatt).
- ▶ A `Connection` interfész öt szintet definiál.
- ▶ A valódi szintek száma természetesen adatbázisszerver-függő.
- ▶ A `setTransactionIsolation` metódussal beállíthatjuk a kapcsolat elszigetelési szintjét, és ez vonatkozik a kapcsolat további szesszióira.

Savepoints

- ▶ A Savepoint interfészt a JDBC 3.0 API vezette be.
- ▶ Egy `SavePoint` egy ellenőrzőpontot jelöl meg egy tranzakción belül és lehetővé teszi, hogy egy tranzakció visszagördüljön addig a pontig ahelyett, hogy a teljes tranzakció visszagördüljön.

PI.:

```
Statement stmt = con.createStatement();
int rows =
    stmt.executeUpdate("INSERT INTO AUTHORS VALUES " +
        "(LAST, FIRST, HOME) 'TOLSTOY', 'LEO', 'RUSSIA'");
Savepoint save1 = con.setSavepoint("SAVEPOINT_1");

int rows =
    stmt.executeUpdate("INSERT INTO AUTHORS VALUES " +
        "(LAST, FIRST, HOME) 'MELVOY', 'HAROLD', 'FOOLAND'");
...
con.rollback(save1);
```

- ▶ Egy tranzakcióhoz több Savepoint-ot rendelhetünk.
- ▶ Ezek automatikusan törölődnek *commit* vagy teljes *roll back* esetében.
- ▶ Ha egy bizonyos Savepoint-ig gördítünk vissza, az utána definiált Savepoint-ok törölődnek.
- ▶ Expliciten is törölhetünk Savepoint-ot:
`con.releaseSavepoint(save1);`
- ▶ Ha egy automatikusan vagy expliciten törölt Savepoint-ra hivatkozunk, `SQLException` kivétel dobódik.

DAO (Data Access Object)

szétválasztja:

- ▶ az adatok feldolgozását (üzleti logika)
- ▶ az adat tárolási logikától

előnyei:

- ▶ rugalmasabb alkalmazás (adatbázisszerver v. adatbázis-kezelő keretrendszer megváltoztatása egyszerűbb)
- ▶ átlátható (könnyebben karbantartható) kód

DAO

használata:

- ▶ DAO interfész – az üzleti logika kód számára ajánlott ennek a metódusait hívni
- ▶ interfész implementálása – tipikusan egy adatbázistáblával függ össze pl. új bejegyzés beszúrása, összes bejegyzés lekérése, keresési műveletek, stb. (egyúttal a kapcsolat nyitása/zárása)

részletesebb leírás, pl.:

<http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html>

Összefoglalás

Ami JDBC-vel kapcsolatban web-alkalmazás esetén különösen fontos:

- ▶ mivel egy web-alkalmazás esetén egyidőben nagyon sok AB-művelet elvégzését is igénylő kérés érkezik
 - ▶ lehetőleg kapcsolatobjektum-készletben (connection pool) részt vevő kapcsolatobjektumot használjunk
 - ▶ amikor épp nincs szükség a kapcsolatobjektumra, zárjuk azt, szabaddá téve a többi kérés számára (minden egyes AB-művelet vagy több, összefüggő művelet végrehajtását követően)
 - ▶ a kapcsolat zárását (készletbe való visszajuttatását) helyezzük finally blokkba
- ▶ komplex web-alkalmazás esetén ajánlott az adathozzáférési logikát különválasztani az adatok feldolgozását végző üzleti logikától (lásd. DAO tervezési minta)