Práce s relační databází 10

Relační databáze představují velmi silný nástroj pro perzistentní uložení dat a práci s nimi, a proto se s nimi dá setkat v celé škále aplikací. V tomto semináři si ukážeme, jak propojit relační databázi a aplikace naprogramované v Javě.

1 Jemný úvod do SQL

S ohledem na to, že ne všichni účastníci tohoto semináře již absolvovali kurz zaměřený na databázové systémy, uvedeme si zde stručné shrnutí, jak se s relačními databázemi pracuje.

Značně zjednodušeně můžeme říct, že základem relačních databází jsou tabulky. Každá tabulka představuje množinu řádků, kde každý řádek nese informaci o nějaké entitě (např. zaměstnanci, studentovi, knize, filmu) a každý sloupec představuje jeden atribut této entity (např. jméno, věk, plat, autora, rok vydání), jak ukazuje například Obrázek 1. Z podstaty relačních databází u těchto tabulek nehraje roli pořadí řádků ani sloupců. Pokud chceme přistupovat k nějakému konkrétnímu řádku (potažmo entitě), musíme použít tzv. primární klíč, tj. jeden nebo více atributů, kterými lze jednoznačně identifikovat každý řádek v tabulce.

S daty, která máme uložena v tabulkách, se pracuje pomocí dotazů, jimiž můžeme vybírat řádky splňující daná kritéria, sloupce (atributy), které nás zajímají, agregovat hodnoty, spojovat informace v tabulkách a řadu dalších věcí, které jdou nad rámec tohoto semináře.

1.1 SQL

Základním prostředkem, kterým se pracuje s relační databází, je jazyk SQL, který umožňuje zejména definovat tabulky, měnit jejich obsah a získat informace v nich uložené.

jméno	rok	režisér	hodnocení
Forrest Gump	1994	R. Zemeckis	8.8
Psycho	1960	A. Hitchcock	8.6
Psycho	1998	G.V. Sant	4.8
Pulp Fiction	1994	Q. Tarantino	8.8
Kill Bill: Vol.1	2003	Q. Tarantino	8.1

Obrázek 1: Ukázková tabulka s informacemi o filmech

1.1.1 Vytvoření tabulky

Nová tabulka se v SQL vytváří pomocí příkazu `CREATE TABLE`:

```
CREATE TABLE tabulka
(sloupec1 typ-dat [vlastnosti-sloupce],
 sloupec2 typ-dat [vlastnosti-sloupce],
 ...,
 sloupecN typ-dat [vlastnosti-sloupce],
 [omezeni-pro-vice-sloupcu]);
```

Jako nejčastější datové typy se používají:

- `integer` nebo `int` – 4 bytové celé číslo se znaménkem (-2147483648 až +2147483647),
- `varchar`(n) – řetezec proměnlivé velikosti mající maximálně n znaků,
- `NUMERIC`(*počet cifer*, *počet desetinných míst*) – slouží k práci s čísly s jednoznačně definovanou přesností. Parametr *počet cifer* udává celkový počet cifer v daném čísle (před i za desetinnou čárkou), *počet desetinných míst* udává počet cifer za desetinnou čárkou. Např. `NUMERIC(6,2)` umožňuje přesně pracovat s čísly ve tvaru 1234.56. Přesnost operací je u tohoto datového typu vykoupena pomalejší prací s těmito hodnotami.
- `real`, `double precision` – typy odpovídající číslům s plovoucí řádovou čárkou podle normy IEEE 754, tj. typům `float` a `double` z jazyka C. U těchto datových typů není zaručena přesnost a nejsou proto vhodné pro ukládání dat, kde na přesnosti záleží, typicky u finančních operací. Na druhou stranu díky přímé podpoře hardwaru jsou operace s hodnotami v těchto sloupcích relativně rychlé.

Jazyk SQL umožňuje definovat celou řadu omezení ať už jednotlivých atributů, tak jejich skupin. My si pro jednoduchost vystačíme s primárním klíčem. Ten lze nastavit buď k jednomu sloupci (k danému sloupci napíšeme modifikátor `PRIMARY KEY`) nebo k více sloupcům, v takovém případě použijeme na závěr deklarace klauzuli `PRIMARY KEY` (*sloupec*$_1$, ..., *sloupec*$_n$).

Ukažme si použití příkazu `CREATE TABLE` na vytvoření tabulky, která obsahuje seznam filmů s atributy *jméno filmu* (`name`), *rok natočení* (`released_in`), *režisér* (`director`) a *hodnocení* (`rating`)[1].

```
CREATE TABLE movies
(name        varchar(50),
 released_in int,
 director    varchar(35),
 rating      double precision,
 PRIMARY KEY(name, released_in));
```

Svádělo by to jako primární klíč zvolit pouze sloupec `name`, ale protože existují remaky filmů, je žádoucí je identifikovat i podle roku vydání.

[1] Předpokládáme hodnocení z intervalu 0.0 až 10.0.

1.1.2 Vkládání řádků do tabulky

Data se do tabulek vkládají příkazem `INSERT INTO` v následujícím tvaru:

`INSERT INTO` *tabulka* `(`*sloupec*$_1$, *sloupec*$_2$, ..., *sloupec*$_n$`) VALUES (`*hodnota*$_1$, *hodnota*$_2$, ..., *hodnota*$_n$`);`

Seznam sloupců lze vynechat. V takovém případě se předpokládá, že budou zadány hodnoty pro všechny sloupce v pořadí, jak byly sloupce zadány příkazem `CREATE TABLE`.

Je možné vložit několik řádek pomocí jednoho příkazu `INSERT INTO`. V části `VALUES` jsou jednotlivé řádky odděleny závorkami.

```
INSERT INTO movies VALUES ('Forrest Gump', 1994, 'R. Zemeckis', 8.8);
INSERT INTO movies (name, released_in, director, rating)
    VALUES ('Psycho', 1960, 'A. Hitchcock', 8.6);
INSERT INTO movies (name, released_in, director, rating)
    VALUES ('Psycho', 1998, 'G.V. Sant', 4.8);
INSERT INTO movies (name, released_in, director, rating) VALUES
    ('Pulp Fiction', 1994, 'Q. Tarantino', 8.8),
    ('Kill Bill: Vol.1', 2003, 'Q. Tarantino', 8.1);
```

1.1.3 Vylistování obsahu tabulky

Nyní již máme data v tabulce a můžeme s nimi pracovat. K tomu slouží dotaz `SELECT`, který má širokou škálu možností, jak data zpracovat a vrátit uživateli. My si ukážeme tři základní varianty.

K získání celého obsahu tabulky slouží dotaz:

`SELECT * FROM` *tabulka*`;`

Dotaz, kterým získáme obsah tabulky `movies`, má tvar:

```
SELECT * FROM movies;
```

Výsledek dotazu bude vypadat následovně:

```
NAME              |RELEASED_IN|DIRECTOR       |RATING
-------------------------------------------------------
Forrest Gump      |1994       |R. Zemeckis    |8.8
Psycho            |1960       |A. Hitchcock   |8.6
Psycho            |1998       |G.V. Sant      |4.8
Pulp Fiction      |1994       |Q. Tarantino   |8.8
Kill Bill: Vol.1  |2003       |Q. Tarantino   |8.1
```

Pozor: V tomto případě SQL nezaručuje, že řádky budou seřazeny v nějakém konkrétním pořadí, např. jak byly zadány, i když to tak v řadě případů může být.

Pokud by nás nezajímala všechna data, můžeme omezit počet sloupcí (provést tzv. *projekci*) tak, že za klíčové slovo `SELECT` jednotlivé sloupce vyjmenujeme. Například následovně:

```
SELECT name, released_in FROM movies;
```

Pokud nás zajímají jen určité řádky (této operaci se říká *selekce* nebo *restrikce*), můžeme použít klíčové slovo `WHERE` a uvést podmínku, kterou mají řádky splňovat. Například následovně:

```
SELECT * FROM movies WHERE director = 'Q. Tarantino';
SELECT name, released_in FROM movies WHERE (released_in >= 1990) AND (released_in <= 1999);
```

První dotaz vrátí seznam filmů, které režíroval „Q. Tarantino". Druhý vrátí jména a rok vydání filmů z devadesátých let.

1.1.4 Odstranění řádků z tabulky

Odstranění řádků z tabulky obstarává příkaz `DELETE`, který má tvar:

`DELETE FROM` *tabulka* `WHERE` *podminka*;

Tento příkaz odstraní z *tabulky* všechny řádky splňující danou podmínku. Například následující příkazy odstraní (i) remake filmu Psycho z roku 1998 a (ii) všechny filmy vzniklé před rokem 2000.

```
DELETE FROM movies WHERE (name = 'Psycho') AND (released_in = 1998);
DELETE FROM movies WHERE released_in < 2000;
```

1.1.5 Změna řádku v tabulce

K aktualizaci jednotlivých řádků tabulky slouží příkaz `UPDATE`, který má následující tvar:

`UPDATE` *tabulka* `SET` $sloupec_1 = vyraz_1, \ldots, sloupec_n = vyraz_n$ `WHERE` *podminka*;

Provedení příkazu update u všech řádků splňujících danou podmínku změní hodnoty v určených sloupcích na hodnoty daného výrazu.

Následující dva příklady (i) změní zkrácené jméno A. Hitchcocka na plné jméno, (ii) zvednou rating filmů Q. Tarantina o 10 %.

```
UPDATE movies SET name = 'Alfred HitchCock' WHERE director = 'A. Hitchcock';
UPDATE movies SET rating = rating * 1.1 WHERE director = 'Q. Tarantino';
```

1.2 Relační databázové systémy

Relační databázové systémy někdy též (systémy řízení báze dat) zajišťují nejen práci s daty, ale celou řadu dalších činností, např. konzistenci dat, přístup více uživatelů, správu oprávnění, distribuci výpočtu. V praxi se můžeme setkat se dvěma variantami relačních databázových systémů.

Na jedné straně jsou to plnohodnotné databázové systémy, které fungují jako síťová služba (samostatný proces, někdy též databázový server), ke které se připojují jednotliví klienti, kteří se službou komunikují pomocí protokolu dané služby a jazyka SQL. Do této kategorie se řadí zejména databázové systémy jako Oracle, PostgreSQL, SQL Server, MySQL/MariaDB a další.

Vedle těchto tradičních databázových systémů existují tzv. embedded databázové systémy, které jsou realizovány jako knihovny, které jsou připojeny k programu (nepotřebují instalaci) a program s nimi

komunikuje pomocí volání funkcí/metod a jazyka SQL. Tato databáze je obvykle určena pro přístup z jednoho procesu. Typickým zástupcem této katogorie databází je SQLite.

V tomto semináři budeme používat Apache Derby, což je databázový systém naprogramovaný kompletně v Javě, který je pro nás zajímavý z toho důvodu, že nevyžaduje složitou instalaci a umí pracovat v režimu síťové služby i jako embedded databázový systém. Pokud máte zkušenost s jiným databázovým systémem, můžete jej v rámci tohoto semináře použít místo něj.

1.3 Začínáme s Apache Derby

Databázový systém Apache Derby[2] stačí stáhnou[3] a rozbalit do vhodného adresáře.

Databázový systém spustíme příkazem `bin/startNetworkServer`[4]
a vypneme jej příkazem `bin/stopNetworkServer`.

Máme-li spuštěný databázový server, můžeme se k němu přípojit jednoduchým konzolovým nástrojem `bin/ij`. Po jeho spuštění nás bude očekávat vstup ve tvaru.

```
ij version 10.15
ij>
```

Nejdříve se k databázi přípojíme příkazem `connect` a URL identifikující naši databázi.

```
ij> connect 'jdbc:derby://localhost:1527/moviedb;create=true';
```

Na začátku URL je specifikovaný protokol `jdbc:derby`, `localhost` udává adresu počítače, kde databázový systém běží a následně je uveden i port[5]. Za lomítkem je název databáze (`moviedb`) a za středníkem je doplňující parametr `create=true`, který zajistí, že v případě že databáze dosud neexistuje, bude vytvořena. Zde podotkněme, že parametr `create=` je specifický pro Apache Derby a není nic neobvyklého, že jedna instance databázového systému může obsahovat více databází.

Když jsme připojeni, můžeme s databází pracovat, vytvářet tabulky, apod. pomocí příkazů, které jsme si ukázali v této kapitole.

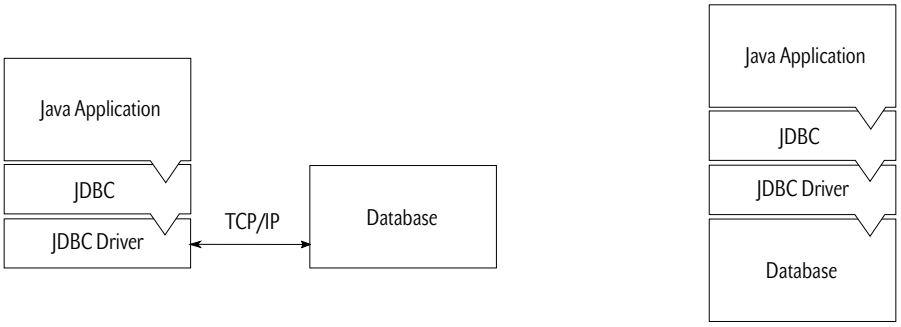
Úkol: Vytvořte v databázi tabulku, naplňte ji daty a vypište její obsah.

2 Přístup k relačním databázím na platformě Java

Platforma Java nabízí rozhraní JDBC (Java Database Connectivity), které zajišťuje jednotný přístup k různým relačním databázím. Architekturu aplikace využívající relační databázi ukazuje Obrázek 2. Důležitou částí této architektury je ovladač, který zajišťuje komunikaci mezi rozhraním JDBC a samotným relační databázovým systémem, s kterým je obvykle dodáván.

V případě Apache Derby je tento ovladač v souborech `lib/derbyclient.jar` a `lib/derbyshared.jar` a je potřeba je oba přidat do *classpath* projektu, aby s ním mohla aplikace pracovat.

[2] `http://db.apache.org/derby/`
[3] `http://db.apache.org/derby/derby_downloads.html`
[4] Na Windows je potřeba přidat `.bat`.
[5] Apache Derby implicitně běží na portu 1527.



Obrázek 2: Přístup k databázi s oddděleným databázovým serverem (vlevo), embedded databáze (vpravo)

2.1 Vytvoření spojení k databázi

Pro potřeby tohoto semináře budeme využívat databázi a tabulku, kterou jsme si vytvořili v předchozí kapitole.

Přístup k databázi si budeme demonstrovat na jednoduchých třídách. První třída bude určena pro přímou manipulaci s tabulkou filmů.

Vytvoříme si proto třídu `DatabaseClient`, která bude mít dvě metody `void connect()` a `void close()`, které se postarají o vytvoření a uzavření spojení do databáze. Tato třída může vypadat následovně.

```
public class DatabaseClient {
    private static final String DB_NAME = "moviedb";
    private static final String HOST    = "localhost:1527";
    private Connection con;

    public void connect() throws SQLException {
        String connectionURL = "jdbc:derby://" + HOST + "/" + DB_NAME;
        con = DriverManager.getConnection(connectionURL);
    }
    public void close() throws SQLException {
        con.close();
    }
}
```

Na začátku máme konstanty udávající jméno databáze a adresu počítače, kde běží databázový systém. Dále máme atribut, do nějž bude uložen objekt reprezentující spojení do databáze.

Metoda, která sestaví spojení má v tomto případě dvě části. V první se sestaví tzv. *connection string*, kterým specifikujeme databázi[6], ke které se připojujeme. V druhém kroce se vytvoří spojení.

Uzavření spojení spočívá v zavolání metody `Connection.close()`.

[6]Tento řetězec se může mezi různými databázovými systémy lišit.

2.2 Dotazy

Asi nejčastější operací, která se používá, je tzv. dotazování, kdy čteme data uložená v databázi. Ukážeme si to na příkladu metody, která vypíše úplný obsah tabulky `movies`.

```
public void listMovies() throws SQLException {
    try (Statement stmt = con.createStatement()) {
        stmt.executeQuery("SELECT * FROM movies");
        try (ResultSet results = stmt.getResultSet()) {
            while (results.next()) {
                String name     = results.getString(1);
                int releasedIn  = results.getInt(2);
                String director = results.getString("director");
                double rating   = results.getDouble("rating");
                System.out.println(name + ", " + director + ", " + releasedIn + ", " + rating);
            }
        }
    }
}
```

V této metodě nejdříve vytvoříme prázdný databázový příkaz (*statement*) a zavoláním metody `Statement.executeQuery(String)` předáme databazi SQL dotaz, který chceme zpracovat. Výsledek je vrácen jako objekt typu `ResultSet`, na který můžeme nahlížet jako na iterátor, kterým procházíme řádky vrácené tabulky. Na počátku je ukazatel v `ResultSet` nastaven před první řádek výsledku a pomocí metody `boolean ResutlSet.next()` se můžeme posunout na další řádek. Pokud se nedá na další řádek posunout, vrací metoda `false`.

K jednotlivým políčkům (sloupcům, atributům) se přistupuje pomocí metod `ResultSet.getInt(int)`, `ResultSet.getString(int)`, `ResultSet.getDouble(int)`, `ResultSet.getInt(String)`, `ResultSet.getString(String)`, `ResultSet.getDouble(String)`, apod. Kdy zvolenou metodou specifikujeme typ hodnoty, kterou v daném sloupci očekáváme, a argumentem určíme sloupec. Ten je možné zadat buď číselně nebo jeho jménem. Avšak pozor! Sloupce jsou indexovány od jedničky a ne od nuly, jak je dnes běžné.[7]

S většinou objektů, se kterými se pracuje, jsou obvykle spojeny i další zdroje, proto je potřeba je korektně uzavírat. Všimněme si proto použití *try-with-resources*.

Objekt `ResultSet` umožňuje získat metadata o výsledku, jako je počet sloupců, jejich jména, typy, apod. Viz přiložený kód.

Nyní již máme kód, který si můžeme vyzkoušet.

```
DatabaseClient client = new DatabaseClient();
client.connect();
client.listMovies();
client.close();
```

[7] Je to nejspíš relikt minulosti, kdy některé databázové systémy počítaly/počítají sloupce od 1.

Nejdříve vytvoříme objekt typu `DatabaseClient`, pak spojení do databáze, vypíšeme obsah tabulky a spojení zase uzavřeme.

2.3 Aktualizace dat

Na přidání nového filmu do databáze si ukážeme problematiku aktualizaci dat. V kódu budeme postupovat velice podobně jako u dotazování.

```
public void insertMovie(String name, int releasedIn, String director, double rating)
         throws SQLException {
    try (Statement stmt = con.createStatement()) {
        String query = "INSERT INTO movies (name, released_in, director, rating) VALUES ('"
                       + name + "', " + releasedIn + ", '" + director + "', " + rating + ")";
        stmt.executeUpdate(query);
    }
}
```

Vytvoříme si prázdný příkaz. Dál si sestavíme řetězec, který odpovídá tomu, co se má provést na straně databáze, tj. vznikne nám příkaz:

```
INSERT INTO movies (name, released_in, director, rating)
VALUES ('The Simpsons Movie', 2007, 'David Silverman', 7.3)
```

Pomocí metody `Statement.executeUpdate(String)` jej necháme provést.

Když nově vytvořenou metodu zavoláme s vhodnými daty, například takto:

```
client.insertMovie("The Simpsons Movie", 2007, "David Silverman", 7.3);
```

Vše proběhne, jak má. A pravděpodobně to bude platit pro velké množství vstupních hodnot. Avšak v programu máme závažnou chybu.

Zkusme vložit jiný animovaný film:

```
client.insertMovie("A Bug's Life!", 1998, "John Lasseter", 7.2);
```

V tomto případě program skončí chybou. Podíváme-li se na problém podrobněji, zjistíme, že se chyba nachází u slova `Bug`[8]. Pokud sestavíme dotaz pomocí spojení řetězců, bude apostrof v dotazu brán jako ukončení řetezce v rámci dotazu a celý požadavek se „rozbije", jak ukazuje následující kód:

```
INSERT INTO movies (name, released_in, director, rating)
VALUES ('A Bug's Life!', 1998, 'John Lasseter', 7.2)
```

Kromě toho, že jsme vytvořili program, který vyvolá výjimku, pokud je na vstupu apostrof, vytvořili jsme program, který obsahuje vážnou (protože snadno zneužitelnou) bezpečnostní chybu, tzv. SQL injection attack[9].

[8] Jak ironické!

[9] `https://en.wikipedia.org/wiki/SQL_injection`

Uvažujme, že máme webovou aplikaci pro fanoušky filmu, kam uživatelé sami mohou pomocí formuláře zadávat filmy. Pokud by se objevil uživatel, který vloží film s názvem[10]

```
A Bug', 1998, 'Jim Hacker', 2.5); DELETE FROM movies; --
```

Můžete mít potenciálně vážný problém, protože dotaz bude interpretován následovně:

```
INSERT INTO movies (name, released_in, director, rating)
VALUES ('A Bug', 1998, 'Jim Hacker', 2.5); DELETE FROM movies; --', 1998, 'John Lasseter', 7.2)
```

Většina databázových systémů se tomuto konkrétnímu útoku dokáže bránit tím, že ve výchozím nastavení je možné zavolat jenom jeden požadavek, ale chyba je zneužitelná i dalšími způsoby, a proto je nutné se jí vyhnout za každou cenu.

2.4 Připravené dotazy (Prepared Statements)

K vyřešení předchozího problému a zároveň jako optimalizaci můžeme použít tzv. *připravené dotazy* (prepared statement). Kdykoliv, když databázový systém obdrží SQL dotaz, musí jej převést z textové reprezentace do nějaké interní formy, nad kterou pak provádí různé optimalizace, pak vytvoří plán provedení dotazu a nakonec jej vykoná. Aby se tento proces co nejvíce urychlil, může klientský program databázovému systému poslat dotazy, které bude v budoucnu provádět. Databázový systém si tyto dotazy předzpracuje (převede do interní formy, zoptimalizuje), a pak klient již jen posílá parametry dotazu a databázový systém vrací výsledky.

Metoda pro přidání nového řádku s filmem by s použitím připraveného dotazu mohla vypadat následovně:

```
public void insertMovie(String name, int releasedIn, String director, double rating)
        throws SQLException {
    String query = "INSERT INTO movies (name, released_in, director, rating) VALUES (?, ?, ?, ?)";
    try (PreparedStatement stmt = con.prepareStatement(query)) {
        stmt.setString(1, name);
        stmt.setInt(2, releasedIn);
        stmt.setString(3, director);
        stmt.setDouble(4, rating);
        stmt.executeUpdate();
    }
}
```

Nejdříve si vytvoříme řetezec s operací, kterou s databází chceme provést. Na místa, kam budeme chtít vložit hodnoty, dáme znak `?`. Vytvoříme si připravený příkaz, pomocí metody `Connection.prepareStatement(String)` a nastavíme hodnoty jednotlivých parametrů[11]. Pak necháme dotaz zavolat pomocí metody `executeUpdate()`. Pokud bychom jako připravený dotaz měli běžný výběr

[10] V podobném duchu můžeme zneužít jakýkoliv vstup, např. jméno režiséra, viz `https://xkcd.com/327/`.

[11] Opět číslovány od jedné.

(SELECT), použili bychom metodu executeQuery() a s vráceným výsledkem bychom pracovali běžným způsobem.

V tomto příkladu jsme využili jenom jednu vlastnost připravených dotazů, tj. schopnost korektně zpracovat vstupy a vyhnout se tak SQL injection attacku. Nevyužíváme už však optimalizací, které připravené dotazy nabízí, a navíc kód ukázkového programu není úplně dobře navržen.

2.5 Rozhraní pro přístup k datům

Jednou z velmi náročných úloh při návrhu programu je vyřešení problému, jak a kde pracovat s načítáním a ukládáním dat. Je otázka, zda má být součástí objektu nebo to má být delegované do samostatných tříd. Hodně záleží na konkrétní situaci. Autorovi semináře se osvědčilo delegovat problematiku ukládání a načítání objektů z databáze do samostatných tříd. Má to jednak výhodu, že si jasně definujeme, které operace se mají s objekty (ve smyslu načítání a ukládání) dít. Navíc tím, že je kód pro ukládání a načítání na jednom místě, umožní nám to v budoucnu jeho snadné úpravy, např. při přechodu na jinou databázi či technologii ukládání dat. Nevýhodou je, že toto řešení představuje nemalé množství kódu, které musí programátor vytvořit.

Jak takto navržený program vypadá ukazují přiložené soubory:

(i) Movie – třída reprezentující film (POJO),

(ii) MovieDatabase – třída obsluhující mechanismus pro ukládání a načítání objektů,

(iii) MovieDBException – výjimka vyvolaná při problému v MovieDatabase,

(iv) MovieApp – ukázková aplikace.

U třídy Movie není moc co probírat, protože se jedná o anemický objekt bez nějaké logiky. Jen poznamenáme, že u tohoto typu tříd má smysl atributy, kterou jsou zároveň primárním klíčem označit jako final. Když už něco jednoznačně identifikuje objekt, nemělo by to být možné měnit.

Třída MovieDatabase představuje prostředek pro nakládání s daty. V konstruktoru inicializujeme všechny připravené dotazy, které jsou používány v metodách pro práci s daty. Zde už s největší pravděpodobností využijeme optimalizací, které nabízí připravené dotazy. Jaké metody by taková třída měla mít, záleží hodně na konkrétní situaci.

Všimněte si, že u všech metod odchytáváme výjimky typu SQLException a zabalíme je do vlastní výjimky MovieDBException. Na první pohled to vypadá zbytečně a řada programátorů by tam nechala SQLException. Není to ale dobré řešení, protože takto začne z objektu prosakovat, že k ukládání dat používáme SQL. Co když jednoho dne přijde někdo a bude to chtít změnit a ukládat data jinam, např. do nějaké NoSQL databáze jako je MongoDB? V takovém případě budeme mít problém. Proto dává smysl vyvolat vlastní výjimku a tu původní do ní zabalit, abychom neztratili informace o tom, k jaké chybě došlo.

Pro usnadnění práce s databází vznikla celá řada ORM[12] frameworků, které umožňují mapovat objekty na řádky v tabulce a obráceně. Tyto frameworky mají však celou řadu úskalí. Podrobněji se práci s nimi a tomu proč je nepoužívat, bude věnovat předmět Platforma Java.

[12] Object-relational mapping

2.6 Embedded databáze

Závěrem si ukažme, jak lze Apache Derby použít jako embedded databázi. Nejdříve musíme do classpath programu přidat balíky `derby.jar` a `derbyshared.jar`.

Připojení k této databázi je velice podobné, jen s tím rozdílem, že nebudeme uvádět adresu počítače a port, kde databáze běží. Vytvoření připojení tedy může vypadat následovně:

```
String connectionURL = "jdbc:derby:" + dbName + ";create=true";
Connection con = DriverManager.getConnection(connectionURL);
```

Nyní můžeme s databází pracovat, jak jsme si ukázali v tomto semináři. Je tu však jedno drobné „ale". V předchozích kapitolách jsme využili toho, že databáze byla inicializována ručně pomocí nástroje `ij`. Nyní to musí obstarat program sám.

Zavedeme si proto metodu `initializeTable(Connection)`, která se postará o vytvoření tabulky a její naplnění při prvním spuštění:

```
private static void initializeTable(Connection con) throws SQLException {
    System.out.println("Vytvarim tabulku i s daty");
    try (Statement stmt = con.createStatement()) {
        stmt.addBatch("CREATE TABLE movies\n" +
                "(name        varchar(50),\n" +
                " released_in int,\n" +
                " director    varchar(35),\n" +
                " rating      double precision,\n" +
                " PRIMARY KEY(name, released_in))");
        stmt.addBatch("INSERT INTO movies VALUES ('Forrest Gump', 1994, 'R. Zemeckis', 8.8)");
        stmt.addBatch("INSERT INTO movies (name, released_in, director, rating) "
            + "VALUES ('Psycho', 1960, 'A. Hitchcock', 8.6)");
        stmt.executeBatch();
    }
}
```

Tento kód je velice podobný tomu, co už jsme tu viděli, je zde však použito dávkové zpracování, tj. všechny požadavky jsou zaslány naráz.

Pokud tento kód zařadíme do následujícího programu a spustíme, mělo by vše proběhnout dle očekávání:

```
public static void main(String[] args) throws SQLException {
    String connectionURL = "jdbc:derby:" + DB_NAME + ";create=true";
    try (Connection con = DriverManager.getConnection(connectionURL)) {
         initializeTable(con);
    }
}
```

Avšak pokud program spustíme podruhé, skončí chybou, protože tabulka `movies` již existuje.

Abychom zjistili, zda tabulka existuje, budeme potřebovat prozkoumat metadata databáze, která mimo jiné obsahují i seznam tabulek. Pro stručnost si ukažme metodu, která zjistí, zda v databázi existuje tabulka s názvem `MOVIES`.

```
private static boolean isReady(Connection con) throws SQLException {
    DatabaseMetaData dbm = con.getMetaData();
    try (ResultSet tables = dbm.getTables(null, null, "MOVIES", null)) {
        return tables.next();
    }
}
```

Nejdříve získáme objekt zpřístupňující metadata a následně metodou `getTables` získáme seznam všech tabulek, co se jmenují `MOVIES`[13]. V případě Apache Derby ostatní parametry nedávají smysl, a jsou proto `null`. Pokud taková tabulka existuje, bude o ní uložen záznam v objektu `ResultSet` a můžeme tak zjistit, zda databáze již byla inicializována či nikoliv.

V kódu by to mohlo být shrnuto následovně:

```
try (Connection con = DriverManager.getConnection(connectionURL)) {
    if (!isReady(con)) initializeTable(con);
    try (MovieDatabase movies = new MovieDatabase(con)) {
        System.out.println(movies.getMovieNames());
    }
}
```

Zde lze vidět, že je kód pro práci s daty je dostatečně univerzální a umí pracovat jak s běžnou síťovou databází, tak s embedded databází.

[13] Apache Derby, podobně jako celá řada dalších databázových systémů, převádí interně jména na velké znaky.