

Databázy a JDBCTemplate,
TestContainers

UINF/PAZ1c
5. prednáška

A black and white photograph of a modern building's curved glass facade. The word "ORACLE" is prominently displayed in large, white, sans-serif capital letters on the upper part of the curve. Several horizontal metallic bands or railings are visible across the glass surface, creating a sense of depth and structure. The background is a clear, light sky.

ORACLE

Relačné databázy

dominantný spôsob ukladania dát

hojný výskum

kilá implementácií

Naša entita User v relačnej databáze

```
CREATE TABLE users (  
  id BIGINT NOT NULL AUTO_INCREMENT,  
  chip_id VARCHAR(20) NOT NULL,  
  name VARCHAR(45) NOT NULL,  
  active BOOLEAN,  
  PRIMARY KEY (`id`)  
);
```

Ako prepojiť
Javu a DB?



JDBC = Java API
pre SQL



IT'S A CRAP!

JDBC

- Funkčné, ale ťažkopádne
- Potenciálne nebezpečné
 - Nutnosť korektne obsluhovať a uvoľňovať prostriedky
 - V tutoriáloch sa kvôli prehľadnosti ignorujú závažné chyby

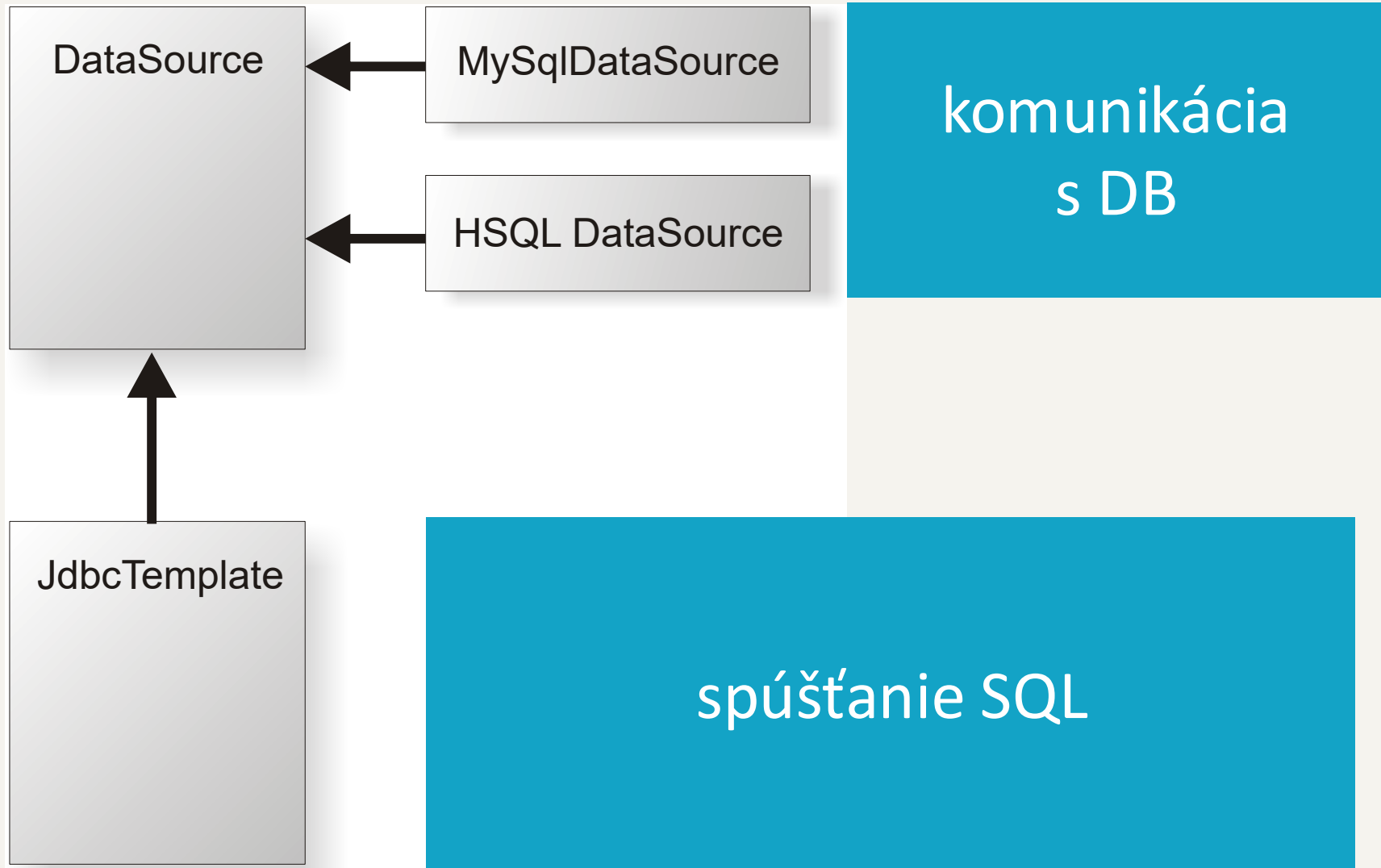
DB spojenia

- Časté vytváranie spojenia je drahé
 - Bežne chceme niekoľko (aj tisíce) dopytov za sekundu
 - Robiť všetko cez jedno trvalé spojenie je pomalé
 - Trpíme: „head of line blocking“ – krátke dopyty zbytočne čakajú na predchádzajúce dlhé operácie
- Typicky databáza dovolí max. 20 - 50 paralelných spojení
 - Potrebujeme to nejako spravovať

Spring
Framework

„Budujme lepšie enterprise aplikácie“

Spring JDBC



JdbcTemplate
v Mavene

mysql-connector-j

spring-jdbc

Inicializácia vo Factory

```
// definovanie systémových premenných, ktoré môžeme nastavovať "zvonka" kódu -  
pri štarte appky, pri štarte testoch atď.  
var dbUser = System.getProperty("DB_USER", "entrance");  
var dbPassword = System.getProperty("DB_PASSWORD", "entrance");  
var url = System.getProperty("DB_JDBC", "jdbc:mysql://localhost:3307/entrance");  
  
var dataSource = new MysqlDataSource();  
dataSource.setUser(dbUser);  
dataSource.setPassword(dbPassword);  
dataSource.setUrl(url);  
  
this.jdbcOperations = new JdbcTemplate(dataSource);
```

Dopyty

```
String sql = "SELECT id, chip_id, name, active FROM users";  
RowMapper<User> rowMapper = new UserRowMapper();  
List<User> users = jdbcTemplate.query(sql, rowMapper);
```

- Pre každý riadok výslednej tabuľky sa zavolá metóda `mapRow()` objektu `rowMapper`
 - mapovač vráti jednu inštanciu triedy `User`
- Metóda `query()` vyzbiera všetky inštancie a vráti ich ako zoznam

Dopyty

```
String sql = "SELECT id, chip_id, name, active FROM user";  
RowMapper<User> rowMapper = new UserRowMapper();  
List<User> users = jdbcTemplate.query(sql, rowMapper);
```

```
public class UserRowMapper implements RowMapper<User> {  
    public User mapRow(ResultSet rs, int rowNum)  
        throws SQLException {  
        User user = new User();  
        user.setId(rs.getLong("id"));  
        /* ďalšie volania setterov */  
        return user;  
    }  
}
```

Dopyty s parametrom

```
String sql = "SELECT id, chip_id, name, active FROM users  
WHERE id = ?";  
RowMapper<User> rowMapper = new UserRowMapper();  
List<User> users = jdbcTemplate.query(sql, rowMapper);
```

```
public class UserRowMapper implements RowMapper<User> {  
    public User mapRow(ResultSet rs, int rowNum)  
        throws SQLException {  
        User user = new User();  
        user.setId(rs.getLong("id"));  
        /* ďalšie volania setterov */  
        return user;  
    }  
}
```

Zmena dát cez JdbcTemplate

```
String sql = "INSERT INTO users (chip_id, name, active)  
VALUES (?, ?, ?)";
```

```
jdbcTemplate.update(sql, user.chipId(), user.name(),  
user.active());
```

- Konverziu necháme na Spring
 - Escapovanie stringov, null sa nevloží ako "null", ...
 - Žiadne SQL injection

Zmena dát cez JdbcTemplate

```
String sql = "INSERT INTO users (chip_id, name, active)  
VALUES (?, ?, ?)";
```

```
jdbcTemplate.update(sql, user.chipId(), user.name(),  
user.active());
```

- Naplnenie id necháme na AUTOINCREMENT
 - Žiadne "SELECT max(id) FROM user" a potom "INSERT ... id+1..."

Ak chceme vedieť aké ID bolo pridelené riadku

```
var keyHolder = new GeneratedKeyHolder();
jdbc.update(connection -> {
    var ps = connection.prepareStatement(sql,
        Statement.RETURN_GENERATED_KEYS);
    ps.setString(1, user.chipId());
    ps.setString(2, user.name());
    ps.setBoolean(3, user.active());
    return ps;
}, keyHolder);

var id = keyHolder.getKey().longValue();
user.setId(id);
```

DAO testovanie

Testovanie "čistých" metód

- **Čistá metóda**
 - Ak ju zavoláme 2x nad tým istým vstupom, dostaneme ten istý výsledok
 - Napr. "sort()" je čistá metóda
- Testy sú ľahké
 - **TestSort** proste zavolá x-krát **sort** s rôznymi vstupmi a porovná assert-ami výstupy
 - Test priamo volá metódu bez potreby nejakého štelovania

Test "nečistých" metod

- **"Nečistá" metoda**

- metoda s postranným efektem
- Typicky:
 - pracuje so súborom alebo sieťou
 - robí systémové volania
 - alebo len pristupuje k meniteľnej inštančnej premennej
- V Jave ju typicky poznáme podľa toho, že hádže kontrolované výnimky

Test "nečistých" metód

- **Postranný efekt**

- externé javy ovplyvňujú správanie metódy
- *dao.findByld(42)* nám raz nájde entitu s daným id, o 10 sekúnd však identické volanie s rovnakým id môže vyhodiť `NotFoundException`, alebo vrátiť `null`, lebo medzitým niekto túto entitu vymazal

- Test potrebuje dôkladne naštellovať zdroj postranného efektu

DAO je nečisté

- DAO je abstrakcia nad nejakým úložiskom
 - MemoryDao, FileDao, MysqlDao
- Úložisko je bašta postranných efektov
- Čím je úložisko komplexnejšie, tým ťažšie sa píše testy
- Databáza je veľmi komplexné úložisko
- V teste DAO treba úložisko dôkladne štelovať

Nečisté DAO

- Pre MySqlConnection teoreticky nevieme napísať unit test
- "Test, ktorý potrebuje externý program nie je unit test"
 - Lebo súčasťou testu je rozbehnutie DB, pripojenie atd.

Teória testov

1. Unit test podľa definície má testovať triedu v "laboratórnych podmienkach"
2. Má testovať iba triedu a nič viac.

Avšak typická trieda potrebuje iné triedy

- **Controller** potrebuje **Dao**
- **Dao** potrebuje **Jdbc**

TestDao teda bude tranzitívne testovať aj **Jdbc**, čím poruší bod 2.

Mockovanie

- Tradičný spôsob unit testovania DAO tried
- Technika "sfalšovania" komplexných závislostí
 - Napr. knižnica Mockito
- Naše MysqlDao používa JdbcTemplate
 - JDBC je most medzi DAO a samotnou DB
 - Mockovanie podhodí do DAO "falošný" JdbcTemplate

Unpopular opinion: Mockovanie DAO je nanič

- DAO posiela SQL dopyty
- Mockovanie obchádza celý ten mechanizmus
 - Donúti metódy JdbcTemplate vracať hard-coded hodnoty
- Testy s mockmi **neodhalia** chybu v dopyte
 - Dá sa to obísť, no s pochybnou úspešnosťou a veľmi komplexnými mockmi
- Najlepšie je testovať nad reálnym úložiskom

Testcontainers

- Súbor knižníc na unit testovanie nad reálnou DB



```
GenericContainer redis = new GenericContainer("redis:5.0.3-alpine")  
    .withExposedPorts(6379);
```

- Ako Docker Compose, ale v Java syntaxi

Testcontainers setup

```
<dependency>  
  <groupId>org.testcontainers</groupId>  
  <artifactId>testcontainers</artifactId>  
  <version>2.0.5</version>  
  <scope>test</scope>  
</dependency>
```

```
<dependency>  
  <groupId>org.testcontainers</groupId>  
  <artifactId>testcontainers-mysql</artifactId>  
  <version>2.0.5</version>  
  <scope>test</scope>  
</dependency>
```

Testovací továrna

```
public enum TestFactory {  
    INSTANCE;  
  
    private Object lock = new Object();  
  
    private MySQLContainer mysql;  
  
    public MySQLContainer getMysql() {  
        if (this.mysql == null) {  
            synchronized (this.lock) {  
                if (this.mysql == null) {  
                    var mysql = new MySQLContainer("mysql:9.7.2")  
                        .withDatabaseName("entrance")  
                        .withCopyFileToContainer(  
                            MountableFile.forHostPath("./init.sql"),  
                            "/docker-entrypoint-initdb.d/init.sql"  
                        );  
                    mysql.start();  
                    this.mysql = mysql;  
                }  
            }  
        }  
    }  
  
    return this.mysql;  
}
```

Testovanie rozhrania

- Mysql DAO a Memory DAO implementujú rovnaké rozhranie
- Rozhranie = interface = kontrakt

Abstraktný UserDaoTest

- Testy Mysql a Memory DAO by mali testovať kontrakt
- NIE implementačné detaily

Abstraktný UserDaoTest

- findByld, findAll, create, update, delete by sa mali správať z vonkajšieho pohľadu rovnako
- Pre náš kontrolér by sa obe DAO mali správať rovnako
- Ak delete v MemoryUserDao iba nastaví *active = false*, tak aj MysqlUserDao to musí urobiť

Abstraktný test

```
public abstract class UserDaoTest {  
    protected UserDao userDao;  
  
    @Test  
    void testUserDao() throws Exception {  
        // copy-paste z MemoryUserDao  
    }  
}
```

MemoryUserDaoTest

```
class MemoryUserDaoTest extends UserDaoTest {  
  
    @BeforeEach  
    void setUp() {  
        this.userDao = new MemoryUserDao();  
    }  
}
```

MysqlUserDaoTest

```
public class MysqlUserDaoTest extends UserDaoTest {  
  
    @BeforeEach  
    void setUp() {  
        var mysql = TestFactory.INSTANCE.getMysql();  
  
        var dataSource = new MysqlDataSource();  
        dataSource.setUser(mysql.getUsername());  
        dataSource.setPassword(mysql.getPassword());  
        dataSource.setUrl(mysql.getJdbcUrl());  
  
        var jdbcOperations = new JdbcTemplate(dataSource);  
  
        this.userDao = new MysqlUserDao(jdbcOperations);  
    }  
}
```

Memory vs Mysql DAO test

- Obidve DAO majú rovnaký kontrakt
 - Obidva testy dedia od spoločného predka
- Kontrakt nič nehovorí o konštruktore
 - Vytvorenie Mysql DAO je úplne iné a komplexnejšie
 - Oba testovacie triedy sa teda líšia iba v *setUp* metóde

Testcontainers zhrnutie

- Z hľadiska teórie SW testovania je takýto test viac tzv. **integračný test**
 - test, čo testuje prepojenie appky s inými komponentami (Java trieda -> MySQL)
- Avšak vďaka Dockeru je setup takmer triviálny
- Programátori aj automatizované pipeline vedia test s TC púšťať rovnako ľahko ako štandardný Unit test bez potreby inštalácie DB.
- Hranica medzi Unit a Int. testom sa ztiera

Nevýhody Testcontainers

- Nová technika, veľa existujúcich Java projektov stále používa iba mocky
- Testcontainers sú náročnejšie na CPU a/alebo RAM oproti mockom
- Štandardne 1 inštancia DB zdieľaná pre všetky testy
 - DB predstavuje postranný efekt
 - Každý test musí po sebe poriadne upratať, alebo treba **písať testy tak, aby sa neovplyvňovali**
 - Žiadne hardcoded ID, atď.
- Možno urobiť TestFactory aby každý test mal vlastný kontajner s DB
 - Veľmi náročné na RAM

Otázky?