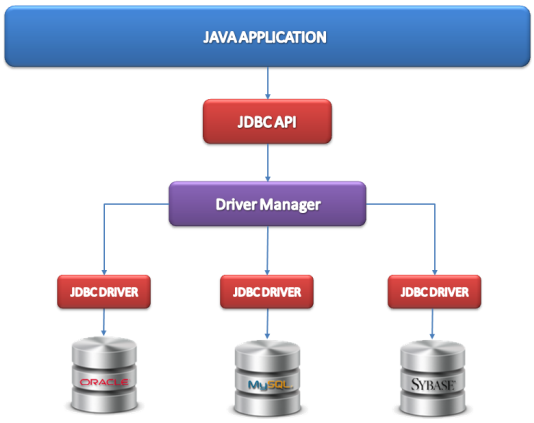
JDBC

Autori:
Goran Savić
Milan Segedinac

1. JDBC

U ovoj lekciji ćemo analizirati kako se perzistencija podataka u bazi podataka može vršiti iz Java aplikacije. JDBC (Java Database Connectivity) predstavlja osnovnu Java tehnologiju za pristup bazi podataka iz Java aplikacije. To je API koji specificira kako program može da ostvari takav pristup. API je implementiran u Java klasama koje su deo standardne Java biblioteke klasa i dizajniran je na generički način tako da podrži različite sisteme za upravljanje bazom podataka (SUBP). Za komunikaciju sa specifičnim SUBP, potrebno je implementirati klase koje poznaju protokol komunikacije sa tim SUBP. Ove klase implementiraju interfejse definisane u JDBC API-ju. Takve klase se nazivaju JDBC drajveri i distribuiraju se kao Java biblioteke u .jar formatu. Za sve popularne SUBP postoje dostupni JDBC drajveri. Ovo je ilustrovano na narednoj slici.



* preuzeto sa https://avaldes.com/connecting-to-mongodb-using-jdbc/

Na primer, za MySQL SUBP drajver je biblioteka pod nazivom *mysql-connector-java*, a glavna drajver klasa je *com.mysql.jdbc.Driver*. Da bi aplikacija mogla da komunicira sa SUBP, potrebno je da se drajver registruje pozivom metode *registerDriver* klase DriverManager. Sam drajver će pozvati ovaj kod i izvršiti registraciju, jer u sebi sadrži statički blok koda koji to radi. Kao što smo ranije objasnili, statički blok koda u klasi se izvršava kada JVM učitava klasu. Učitavanje klase se vrši pri instanciranju objekta, pri pristupu statičkom atributu ili pri pozivu statičke metode klase. Pošto je upravo cilj da kod koji radi sa bazom podataka što manje zavisi od konkretnog SUBP, Java aplikacija ne treba da instancira objekte specifičnog drajvera. Da bismo uprkos tome inicirali učitavanje drajvera od strane JVM, koristi se metoda forName klase Class. Ova metoda učitava klasu i vraća objekat klase Class koji reprezentuje klasu koju smo učitali. U skladu sa navedenim, dat je primer koda za učitavanje MySQL JDBC drajvera.

```
Class.forName("com.mysql.jdbc.Driver");
```

Nema potrebe za preuzimanjem objekta kojeg metoda vrati, jer je svrha pozivanja metode bilo samo učitavanje drajver klase kako bi se izvršio njen statički blok. Nakon što smo registrovali drajver za određeni SUBP, moguće je komunicirati sa tim SUBP.

Otvaranje konekcije

Prvi korak u komunikaciji sa SUBP je otvaranje konekcije. Konekcija je predstavljena klasom *Connection*. Obzirom da je potrebno uspostaviti konekciju ka specifičnom SUBP, instanca konekcije se dobavlja preko DriverManager klase koji upravlja JDBC drajverima ka različitim SUBP. Pri otvaranju konekcije potrebno je navesti parametre konekcije, što su adresa servera na kojem se SUBP nalazi, port za komunikaciju sa SUBP, baza podataka

sa kojom komuniciramo, kao i podatke o korisničkom nalogu koji vrši pristup. String u kojem definišemo ove podatke se naziva *connection string*. Naredni primer prikazuje otvaranje konekcije.

```
Connection conn = DriverManager.getConnection(  "jdbc:mysql://localhost:
3306/vp","root", "root");
```

Izvršavanje upita

Upit ka bazi se izvršava preko Statement objekta. Dat je primer kreiranja i izvršavanja upita koji preuzima sve države iz baze podataka.

```
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery("select * from country");
```

Vidimo da se upit kreira na osnovu Connection objekta i da se izvršava metodom *executeQuery* koja kao parametar dobija upit u SQL sintaksi. Obzirom da SELECT upit vraća slogove iz baze, ovi podaci se u memoriji smeštaju u objektu klase ResultSet. Ovaj objekat je memorijska reprezentacija slogova preuzetih iz baze. Nad objektom je moguće iterirati kroz slogove i preuzimati polja trenutnog sloga. Pogledajmo na primeru kako se iteracijom kroz slogove u ResultSet objektu iz prethodnog primera može dobiti ispis podataka o svakoj državi preuzetih iz baze podataka.

```
while (rset.next()) {
      String countryName = rset.getString("name");
      int population = rset.getInt("population");
      System.out.println(countryName + ", " + population);
}
```

ResultSet objekat sadrži informaciju o trenutnom slogu na koji je pozicioniran. Inicijalno nije pozicioniran ni na jedan slog, a pozicioniranje na sledeći slog se vrši pozivom metode next(). Metoda vraća vrednost false kada više nema sloga na koji bi se pozicioniralo jer se došlo do kraja skupa. Iz trenutnog sloga je moguće preuzeti vrednosti polja metodama getXXX. Naziv metode zavisi od tipa podatka u polju. Tako postoje metode getString, getInt, getBoolean, getDouble itd. Svaka od metoda vraća vrednost odgovarajućeg tipa.

Prethodni primer je pokazao kako možemo izvršiti upit na bazu koji vraća slogove kao rezultat. Upiti kao što su INSERT, UPDATE i DELETE ne vraćaju slogove iz baze nego je njihova svrha da klijent pošalje određene podatke ka bazi podataka. Dat je primer koda koji vrši ubacivanje nove države u bazu podataka.

```
Statement stmt = con.createStatement();
int noOfInsertedRows = stmt.executeUpdate(
    "insert into country (name, population) values ('Germany', 81410000)");
```

Vidimo da se za ovaj tip upita koristi takođe *Statement* objekat, ali njegova metoda *executeUpdate*. Povratna vrednost ove metode nisu slogovi kao u prethodnom slučaju nego broj slogova u bazi nastalih, izmenjenih ili obrisanih na osnovu izvršenog upita. U prikazanom primeru, ovaj broj će biti 1 jer je navedeni upit ubacio jednu novu državu u bazu podataka.

2. SQL Injection napad

U prethodnom primeru sa dodavanjem nove države, podaci o državi koja se dodaje su bili predefinisani u kodu. U realnom scenariju, podatke unosi korisnik, a aplikacija je zadužena da te podatke prosledi bazi podataka. Pogledajmo kako bi izgledala pretraga država u bazi. Korisnik bi imao mogućnost da u tekstualno polje unese državu koju traži i

recimo da je server primio tu vrednost i uskladištio je u promenljivu *countryName*. Upit koji bi preuzeo državu sa unešenim nazivom bi izgledao kao u narednom primeru.

```
stmt.executeQuery("select * from country where name = '" + countryName + "'");
```

Vidimo da je unešene vrednost za ime države ubačena u konačan SQL upit. Dakle, korisnikov unos se ubacuje kao deo SQL upita. Ako bi korisnik uneo tekst Germany, konačan SQL upit koji se šalje bazi bi bio

```
select * from country where name = 'Germany'
```

Šta bi se desilo ako bi maliciozan korisnik u polje za pretragu upisao sledeći tekst ?

```
abc'; drop table countries; --
```

Ova vrednost bi bila prosleđena kao *countryName* i nakon spajanja sa gore navedenim SQL upitom, konačan upit koji bi bio izvršen ka bazi bi bio:

```
select * from country where name = 'abc'; drop table countries; --'
```

Vidimo da se sada izvršavaju dva upita. Prvi upit traži državu i za malicioznog korisnika rezultat ovog upita je nebitan. Drugi upit uklanja sve države iz baze podataka. Znakovi -- su komentar u SQL-u i čine da ostatak upita nije važan i da dobijemo dva sasvim validna SQL upita.

Ovo se zove SQL Injection napad. Kao što smo videli, izvodi se injektovanjem malicioznog SQL koda tako da konačan upit izvrši operaciju koja korisniku nije dozvoljena.

JDBC ima podršku za sprečavanje SQL Injection napada putem definisanja pripremljenih SQL upita. Osnovna ideja pristupa je da se odvojeno tretira SQL kod koji definiše upit od podataka koje upit sadrži. Kod pripremljenih upita, podaci prosleđeni upitu se ne parsiraju kao SQL upiti.

Pogledajmo primer pripremljenog upita za prethodni primer sa pronalaženjem države.

```
PreparedStatement pstmt = conn.prepareStatement(
      "select * from country where name = ?");
pstmt.setString(1, countryName);
pstmt.executeQuery();
```

Vidimo da je pripremljeni upit predstavljen klasom *PreparedStatement*. Pri definisanju pripremljenog upita, samo se definiše upit, a sami podaci se parametrizuju znakom ?. Na mesto parametra će naknadno biti ubačena konkretna vrednost u upit, ali njen sadržaj neće biti parsiran kao SQL upit. Pri postavljanju parametra navodi se njegov redni broj (1 u ovom primeru) i tip kroz poziv odgovarajuće metode (setInt(), setString(), setDouble(), itd.).

Slično, ranije prikazani primer sa dodavanjem nove države bi korišćenjem pripremljenog upita izgledao kao u nastavku, ako bi unešene vrednosti za državu bile u promenljivima *countryName* i *countryPopulation*.

```
PreparedStatement pstmt = conn.prepareStatement(
      "insert into country (name, population) values (?, ?)");
pstmt.setString(1, countryName);
pstmt.setInt(2, countryPopulation);
pstmt.executeUpdate();
```

Osim zaštite od SQL Injection napada, pripremljeni upiti se koriste i za ubrzavanje izvršavanja upita u situaciji kada program veći broj puta izvršava isti upit nad različitim podacima. Tada se pri kreiranju pripremljenog upita, upit parsira samo jednom, a kasnije se samo različite vrednosti ubacuju u upit.

3. JDBC podrška za transakcije

Ranije smo objasnili da je određene operacije u bazi podataka potrebno izvršiti kao nedeljivu celinu, što nazivamo transakcijom. Korišćenjem JDBC možemo za više SQL upita definisati da pripadaju istoj transakciji.

Trenutak kada se potvrda transakcije vrši definisan je u klasi Connection atributom autoCommit, koji podrazumevano ima vrednost true. To znači da se svaki SQL automatski potvrđuje u bazi podataka (*commit* operacija), tako da je svaki novi SQL upit nova posebna transakcija. Vrednost ovog atributa se može postaviti na false, tako da korisnički kod preuzme odgovornost kreiranja i potvrđivanja transakcije. Pogledajmo to na ranijem primeru prebacivanja novca sa jednog bankovnog računa na drugi, što bi moralo biti izvršeno kao nedeljiva operacija.

```
try {
	conn.setAutoCommit(false);

	String query1 = "UPDATE account SET amount=400 WHERE id=1";
	String query2 = "UPDATE account SET amount=600 WHERE id=2";
	Statement stmt = conn.createStatement();
	stmt.executeUpdate(query1);
	stmt.executeUpdate(query2);
	stmt.close();

	conn.commit();

} catch (Exception ex) {
	try {
		if(conn != null)
			conn.rollback();
	} catch (SQLException e) {
		e.printStackTrace();
	}
}
```

Vidimo da se transakcija potvrđuje metodom *commit()*, a poništava metodom *rollback()* nad objektom klase Connection. Naglasimo da će baza podataka svakako poništiti transakciju ako je neka od njenih operacija neuspešno izvršena. Ipak, dobra je praksa da se u slučaju greške u *catch* bloku pozove metoda *rollback* nad konekcijom kako bi objekat klase *Connection* ostao u ispravnom stanju za buduće transakcije.

4. JDBC podrška za uskladištene procedure

U lekciji o bazama podataka smo pomenuli da je moguće napisati funkcije koje se direktno izvršavaju u okviru SUBP-a i da ove funkcije nazivamo uskladištene procedure. JDBC omogućuje i programski poziv uskladištene procedure. Ako je u okviru SUBP definisana uskladištena procedura sa zaglavljem *transfer(IN srcAccountId INT, IN dstAccountId INT, IN amount INT)*, i ako su u Java kodu oznaka izvornog računa, oznaka ciljnog računa i iznos koji se prenosi redom definisani u promenljivim *srcAccount*, *destAccount* i *amount*, ta procedura može biti pozvana na sledeći način.

```
CallableStatement stmt = dbConnection.prepareCall(
      "{call transfer(?, ?, ?)}");

stmt.setInt(1, srcAccount);
stmt.setString(2, destAccount);
stmt.setInt(3, amount);
stmt.execute();
```

Slično definisanje pripremljenog upita i ovde se prvo definiše šablon poziva, a onda se postavljaju parametri koji će biti prosleđeni proceduri. Sama procedura se poziva metodom *execute* klase *CallableStatement*.