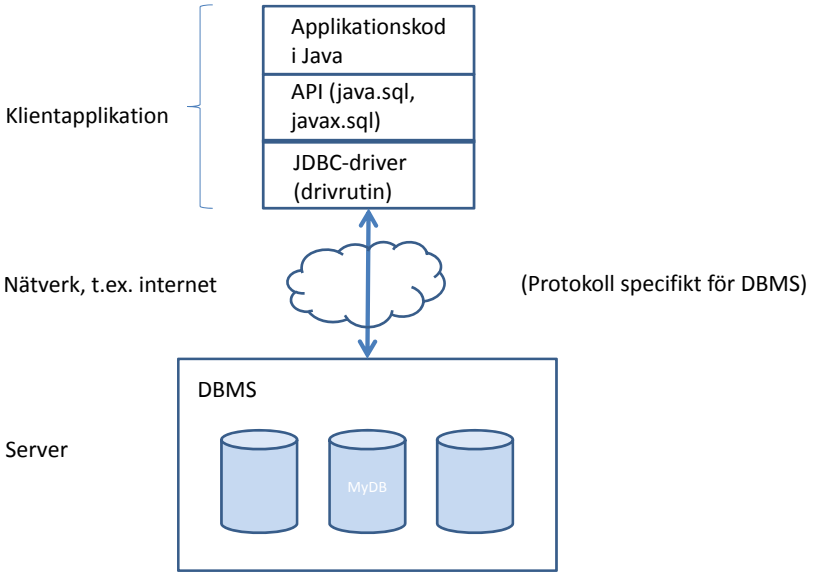
Klientprogrammering mot Databaser

Java DataBase Connectivity, JDBC

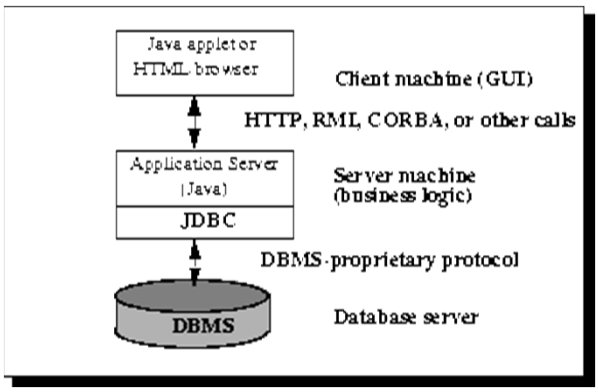
Klientprogrammering mot databaser

- Native API
  - olika för olika DBMS, ofta i C/C++
- ODBC, Open Datbase Connectivity
  - samma API för olika databashanterare. C/C++
- JDBC, Java Database Connectivity
  - samma API för olika databashanterare. Java
- Webbserver + skriptspråk för dynamiska webbsidor
  - PHP, Active Server Pages (ASP), Java Server Pages (JSP) m.fl.

JDBC, Arkitektur



Treskiktsarkitektur



Källa: http://docs.oracle.com/javase/tutorial/jdbc/index.html

Komponenter i JDBC

- DriverManager – laddar rätt drivrutin (om den finns på systemet) och skapar en connection.
- Connection – representerar en session med databasen och sköter kommunikationen.
- Statement/PreparedStatement/CallableStatement – används för att exekvera SQL-kommandon. Fås från Connection-objektet.
- ResultSet – en tabell data, resultatet av en SQL-fråga.

Skapa en anslutning

```
String server = "jdbc:postgresql://localhost:5432/" +
nameOfDatabase + "?UseClientEnc=UTF8";

Class.forName("org.postgresql.Driver");
Connection con =   DriverManager.getConnection(
                                        server, user, pwd);

...
con.close();
```

Exekvera en SQL-sats

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

String sql = ”DELETE FROM Employee WHERE
                                    name LIKE ’Anders’”;
int n = stmt.executeUpdate(sql);
```

- executeUpdate kan exekvera satser med INSERT, DELETE och UPDATE-satser (och DDL-satser)

Exekvera en SQL-fråga, SELECT

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

String sql = ”SELECT * FROM Employee”;
ResultSet rs = stmt.executeQuery(sql);

while(rs.next()) {
        int eno = rs.getInt(”eNo”);
        String name = rs.getString(”name”);
        . . .
}

. . .
stmt.close();
```

ResultSet

- Cursor (ung. en iterator) pekar på aktuell rad, next() förflyttar cursorn 1 rad
- Metoder för att hämta alla vanliga datayper, t.ex.
  - getDate(String columnName)
  - getDate(int column) - första index är *1*
- getMetaData() – antal, typer, namn m.m. för kolumnerna
- NB! – ResultSet är ”temporärt”, om stmt stängs, eller ny fråga exekveras på samma stmt, stängs rs.

PreparedStatment

- Använd Prepared statement då en specifik sats exekveras ofta
- Satsen kompileras första gången den exekveras och kan ”återanvändas” tills den uttryckligen stängs (eller session avslutas)
- String sql = ”INSERT . . .”;
  PreparedStatement pstmt =
  con.*prepareStatement*(sql);
  int n = pstmt.executeUpdate();

PreparedStatement

- Man kan, och bör, använda in-parametrar till ett PreparedStatement
- String sql = "INSERT INTO Emp_Proj VALUES (*?*, *?*, *?*) ";
  PreparedStatement addEmpToProj =
  con.prepareStatement(sql);
- addEmpToProj.setInt(*1*, 107);
  addEmpToProj.setInt(*2*, 1002);
  addEmpToProj.setDouble(*3*, 100);
- int n = addEmpToProj.executeUpdate();

CallableSatement

- Används för att anropa lagrade procedurer
- String call = *"{? = call proj_hours(?)}"*;
  CallableStatement cstmt =
  con.prepareCall(call);
  cstmt.registerOutParameter(*1*, Types.BIGINT);
- cstmt.setInt(*2*, projNo);
  cstmt.execute();
  int hours = cstmt.getInt(*1*);

Felhantering

- SQLException och subklasser
- Checked exceptions
- - se.getSQLState()SQL-state enligt SQL:2003 konventioner
  - se.getErrorCode()
  - se.getMessage()
- Länk till ursprunglig(a) exception(s)

Felhantering

- Viktigt att allokerade resurser frigörs (ex statement stängs) oavsett om fel uppstår eller ej
- Om du inte kan hantera felet i metoden – kasta det vidare till anropande metod

- public void sqlMethod(. . .) *throws SQLException* {

```
    Statement stmt  null;
    try {
            stmt = con.createStatement();
            . . .
    }
    finally {
            stmt.close();
    }
}
```

Transaktioner

1. Slå av auto-commit (som är default)
2. Exekvera sql-satserna som ingår i transaktionen
3. Kontrollera om transaktionen verkligen ska genomföras. . .
4. Utifrån svaret ovan: con.commit() eller con.rollback()
5. Slå på auto-commit igen

Transaktioner

```
Statement stmt = null;
try {
       stmt = con.createStatement();
       con.setAutoCommit(false);
       stmt.executeUpdate("INSERT . . .");
       stmt.executeUpdate("UPDATE . . .");
       // . . .
       con.commit();
}
catch (SQLException e) {
       if (con != null) con.rollback();
       throw e;
}
finally {
       if (stmt != null)  stmt.close();
       con.setAutoCommit(true);
}
```

Användare(!)

- Connection con = DriverManager.getConnection(*server*, *user*, *pwd*);
  Vilken användare?
- Skapa en användare som har på lämpligt sätt begränsade rättigheter till din databas
- Ex för PostgreSQL:
  CREATE USER clienttapp PASSWORD 'qwerty'
  GRANT SELECT ON Employee, Department, Project TO clienttapp
- Alternativt ge rättigheter endast till specialiserade vyer

SQL-injection(!)

- String sql = ”SELECT name, phone FROM Employee WHERE name = ’” + userInput + ”’ AND secretPhone = FALSE”;
- Vad blir frågan om användaren matar in ”Anders’ OR ’x’=’x OR ’x’=’x”?
- Vill du slippa problem med SQL-injection
  - använd PreparedStatement, där parameterar skickas till den förkompilerade frågan:

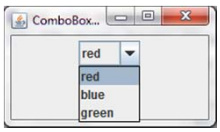
  pstmt.setString(*1*, userInput);

Applikationen

- Model-View-Controller
  - Model är klasser som kommunicerar med databasen via JDBC
- Ev – ”mappa” ResultSet mot objekt, ex ArrayList<Employee> som sedan skickas vidare till view
- Använd gärna PreparedStatements, effektivare och säkrare
- Exceptions som fångas först i UI bör åtminstone resultera i ett kort felmeddelande (använd JOptionPane).

Applikationen, user interface

- Om endast vissa värden är tillåtna, gör det omöjligt att mata in annat, använd t.ex. JComboBox



- Inloggning sker lämpligen i ett separat fönster, använd JDialog
- Resultat av sökningar presenteras snyggt i en JTable (+ JTableModel)
- BorderLayout ger enkelt ett snyggt och intuitivt ui