API for database applications: JDBC

Maciej Grzenda
Warsaw University of Technology
Faculty of Mathematics and Information Science
M.Grzenda@mini.pw.edu.pl
http://www.mini.pw.edu.pl/~grzendam
JDBC

- Stands for Java Database Connectivity
- Provides universal data access from the Java programming language to different data sources including:
  - Relational databases,
  - Spreadsheets,
  - Flat files.
- JDBC API is composed of:
  - `java.sql` package,
  - `javax.sql` package which contains server-side extensions.
- Latest version: JDBC 4.1, a part of Java SE 7
JDBC API

• JDBC API guide:
  http://docs.oracle.com/javase/7/docs/technotes/guides/jdbc/

• Tutorial – JDBC basics:

• JDBC overview:
  http://www.oracle.com/technetwork/java/overview-141217.html
Why JDBC API?

- The JDBC API consists of a set of classes and interfaces written in the Java programming language that provide a standard API for tool/database developers and makes it possible to write industrial strength database applications using an all-Java API.
- The JDBC API makes it easy to send SQL statements to relational database systems and supports all dialects of SQL. But the JDBC 2.0 API goes beyond SQL, also making it possible to interact with other kinds of data sources, such as files containing tabular data.
- The value of the JDBC API is that an application can access virtually any data source and run on any platform with a Java Virtual Machine.
- One can write a single program using the JDBC API, and the program will be able to send SQL or other statements to the appropriate data source.


SQL statements can query and modify the data in a database.
Establishing a connection – the first option

• The first option is to load a specified JDBC driver in order to establish a connection
• This is used when a client (frequently a thick client) connects directly to a database
• Advantages:
  – The same API can be used to access different databases
• Disadvantages:
  – JDBC driver name hard coded in the application
  – It takes time to establish and close a connection; whenever possible connection pooling should be applied.
import java.sql.*;
public class ConnectionTester {
    public ConnectionTester() {
    }
    public static void main(String[] args) {
        ...

        Connection con = null;
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:ORCL", "hr", "hrpass");

            Statement statement = con.createStatement();
            ResultSet rs = statement.executeQuery("SELECT * FROM Jobs");
while (rs.next()) {
    System.out.println("Job: "+rs.getString("Job_Title")+" with salary starting from "+rs.getString("Min_Salary"));
}
statement.close();
rs.close();
con.close();

} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (SQLException e) {
    e.printStackTrace();
}

JDBC Oracle driver is used. Different driver would be needed to access another DBMS.
Establishing a connection – the 2\textsuperscript{nd} option

- The second option is to use a connection obtained from a data source.
- This is used by server-side applications (e.g. web applications, web services,...) i.e. the applications having business logic components executed on the servers.
- Advantages:
  - The same API can be used to access different databases.
  - The connection attributes including driver type and DBMS platform are not hard coded.
  - Connection pool can be accessed.
- Disadvantages:
  - Application server is required to host the Java application.
A data source is used by the code located is such application.
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.sql.DataSource;

public class testServlet extends HttpServlet {
    private static final String CONTENT_TYPE = "text/html";

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        response.setContentType(CONTENT_TYPE);
        PrintWriter out = response.getWriter();
        out.println("Hello, World!");
        out.close();
    }
}

http://www.mini.pw.edu.pl/~grzendam
Connection from a DataSource – part II

```java
public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException,
        IOException {
    response.setContentType(CONTENT_TYPE);
    PrintWriter out = response.getWriter();
    out.println("<html>);
    Connection con = null;
    try {
        InitialContext ic=null;
        try {
            ic = new InitialContext();
        } catch (NamingException e) {
            ...
        }
        DataSource ds=null;
        try {
            ds = (DataSource) ic.lookup("jdbc=DBTestDS");
        } catch (NamingException e) {
            ...
        }
        con = ds.getConnection();
    } catch (NamingException e) {
        ...
    }
    out.println("</html>");
    out.flush();
}
```

A reference to JNDI DataSource is made. Thus, the real configuration of a connection (the type and location of a DBMS, driver type, security credentials) is no longer hard coded.
Connection from a DataSource – part III

```java
Statement statement = con.createStatement();
ResultSet rs = statement.executeQuery("SELECT * " + "FROM Jobs");

while (rs.next()) {
    out.println("<li> Job: "+rs.getString("Job_Title")+" with salary starting from "+rs.getString("Min_Salary");
    }
    statement.close();
    rs.close();
    con.close();

} catch (SQLException e) {
    e.printStackTrace();
}
out.println("</body></html>");
out.close();
```

The rest of the code is quite similar to the previous example. The code is based on JavaServlet. In fact Java EE application server providing JNDI services is used in the example.
Data sources - defined

```xml
<?xml version = '1.0' encoding = 'windows-1250'?>
<data-sources xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://xmlns.oracle.com/oracleas/schema/data-sources-10_1.xsd"
xmlns="http://xmlns.oracle.com/oracleas/schema">
  <connection-pool name="jdev-connection-pool-DBTest">
    <connection-factory factory-class="oracle.jdbc.pool.OracleDataSource" user="hr"
      password="->DataBase_User_iJr2Heyyw8WoyfiTIBHQSSzBbm45XmM"
      url="jdbc:oracle:thin:@localhost:1521:ORCL"/>
  </connection-pool>

  <managed-data-source name="jdev-connection-managed-DBTest"
    jndi-name="jdbc/DBTestDS" connection-pool-name="jdev-connection-pool-DBTest"/>

  ...
</data-sources>
```

The content can be found in `<ApplicationName>-data-sources.xml`, which is a part of Java application configuration. The configuration is used by Java EE application server to establish data sources used by Java applications e.g. JavaServlet-based web applications.
Web applications and databases

1. HTTP request is sent

0. the user wants to display a document

2a. Document is read

2b. Scripts are read and executed

3. Response is received

Web server

DBMS#1

Database

Web browser

4. Response is displayed to the user

In case web documents are generated, this usually relies on a database content.

http://www.mini.pw.edu.pl/~grzendam
Connection pooling

Thin client (usu. web client)

Thin client (usu. web client)

Thin client (usu. web client)

LAN/WAN

Application server

Java application

JDBC

Data source/Connection pool #1

Data source/Connection pool #n

DBMS#1

DBMS#n

Thin client

Thin client

Thin client

Thin client

Thin client

Thin client

Thin client

Thin client

Thin client
Case study

Thin client (usu. web client)

Thin client (usu. web client)

Thin client (usu. web client)

LAN/WAN

Application server: Oracle WebLogic Server

Java application

JDBC

Data source/Connection pool #1

Oracle11gR2

Database

Maciej Grzenda

http://www.mini.pw.edu.pl/~grzendam
Connection pooling

- A pool of connections is created by an application server.
- `DataSource` object can be used to access such connections. In that case:
  - `getConnection()` usu. returns an existing connection,
  - `close()` releases a connection so that it can be reused.
- Existing connections can be reused instead of being closed.
- This may result in significant performance improvements.
- The application server maintains a pool of connections, as configured by the administrator.
- Obtaining a connection through a `DataSource` is a suggested solution in multi-user enterprise applications.

Whether a `DataSource` operates a connection pool or not depends on the developers implementing it. The `DataSource` implementations are provided by DBMS vendors and third-party vendors.
Data modifications - PreparedStatement

```java
...
    con = ds.getConnection();

    PreparedStatement ps = con.prepareStatement(
        "UPDATE Jobs SET MIN_SALARY = ? WHERE
        JOB_Title = ?");

    ps.setInt(1,100);
    ps.setString(2,"Finance Manager");

    ps.executeUpdate();
    con.commit();
```

Whenever SQL Statement is supposed to be executed a number of times, PreparedStatement should be preferred. It creates a precompiled statement on the DBMS, which allows to significantly reduced time required to execute a query. The solution can be used both for queries selecting and modifying the data. A regular Statement can be used otherwise.
Discussion

- ResultSet objects have to be closed as soon as possible
- Connection objects have to be closed as soon as possible
- Hint:
  - Manage all the resources of this kind in an organised way
  - Monitor the number of used database objects (result sets, connections,…) to avoid resource leak