

Outline

1. Introduction to JDBC
2. Accessing a database: practical steps
3. Connection pools in Tomcat
4. Prepared statements



<http://dilbert.com/strips/comic/1995-11-17/>

Goals

- ▶ Access SQL DBMS's from JSP pages
 - ▶ JDBC technology
- ▶ Integrate SQL query results into the resulting HTML content
- ▶ Generate SQL queries according to FORM values



JDBC

- ▶ Standard library for accessing relational databases
- ▶ Compatible with most/all different databases
- ▶ JDBC : Java Database Connectivity
- ▶ Defined in package `java.sql` and `javax.sql`
- ▶ Documentation:
 - ▶ <http://java.sun.com/javase/technologies/database/index.jsp>

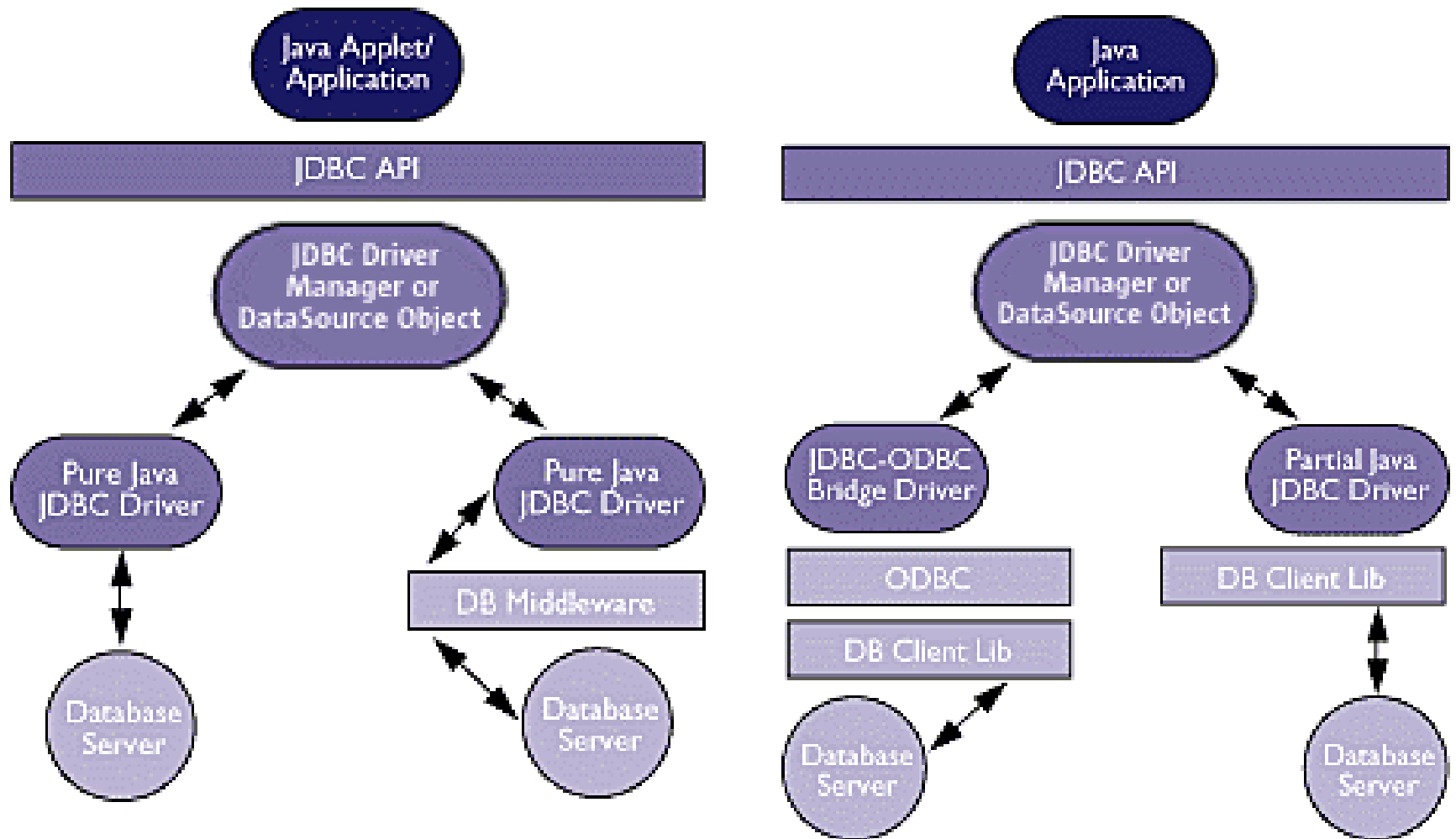


JDBC scope

- ▶ **Standardizes**
 - ▶ Mechanism for connecting to DBMSs
 - ▶ Syntax for sending queries
 - ▶ Structure representing the results
- ▶ **Does not standardize**
 - ▶ SQL syntax: dialects, variants, extensions, ...



Architecture



Main elements

- ▶ Java application (in our case, JSP)
- ▶ JDBC Driver Manager
 - ▶ For loading the JDBC Driver
- ▶ JDBC Driver
 - ▶ From DBMS vendor
- ▶ DBMS
 - ▶ In our case, MySQL



Types of drivers (1 / 3)

- ▶ **A JDBC-ODBC bridge**

- ▶ provides JDBC API access via one or more ODBC drivers. ODBC native code must be loaded on each client machine that uses this type of driver.

- ▶ **A native-API partly Java technology-enabled driver**

- ▶ converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS. Requires that some binary code be loaded on each client machine.



Types of drivers (2/3)

- ▶ **A net-protocol fully Java technology-enabled driver**
 - ▶ translates JDBC API calls into a DBMS-independent net protocol which is then translated to a DBMS protocol by a server. Specific protocol depends on the vendor. The most flexible alternative



Types of drivers (3 / 3)

- ▶ **A native-protocol fully Java technology-enabled driver**
 - ▶ converts JDBC technology calls into the network protocol used by DBMSs directly. Direct call from the client machine to the DBMS server. Many of these protocols are proprietary: the database vendors will be the primary source for this style of driver.





Accessing a database: practical steps

Database access and JDBC

Basic steps

1. Load the JDBC driver
2. Define the connection URL
3. Establish the connection
4. Create a statement object
5. Execute a query or update
6. Process the results
7. Close the connection



1. Loading the driver

- ▶ A Driver is a DMBS-vendor provided class, that must be available to the Java application
 - ▶ Must reside in Tomcat's CLASSPATH
- ▶ The application usually doesn't know the driver class name until run-time (to ease the migration to other DMBSs)
- ▶ Needs to find and load the class at run-time
 - ▶ Class.forName method in the Java Class Loader (not needed in recent versions)



MySQL JDBC driver

- ▶ MySQL Connector/J
 - ▶ <http://www.mysql.com/downloads/connector/j/>
- ▶ Provides mysql-connector-java-[version]-bin.jar
 - ▶ Copy into CLASSPATH
 - ▶ E.g.: c:\Program files\Java\jre1.5.0_09\lib\ext
 - ▶ Copy into Tomcat's libraries
- ▶ The driver is in class
 - ▶ `com.mysql.jdbc.Driver`



Loading the MySQL driver

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

// Notice, do not import com.mysql.jdbc.* or you will have problems!

public class LoadDriver {
    public static void main(String[] args) {
        try {
            // The newInstance() call is a work around for some
            // broken Java implementations

            Class.forName("com.mysql.jdbc.Driver").newInstance();
        } catch (Exception ex) { // mostly ClassNotFoundException
            // handle the error
        }
    }
}
```



Loading the MySQL driver

```
import java.sql.Connection;  
import java.sql.DriverManager;  
import java.sql.SQLException;
```

```
// Notice, do
```

```
public class L  
    public sta  
        try {
```

```
        //  
        //
```

```
        Cl
```

```
    } catch (Exception ex) { // mostly ClassNotFoundException  
        // handle the error  
    }
```

```
}
```

Note: in recent versions of the Java JVM, this step is no longer needed.

The class is looked up in all the libraries (.jar) found in the CLASSPATH

ve problems!

or some

stance());



2. Define the connection URL

- ▶ **The Driver Manager needs some information to connect to the DBMS**
 - ▶ The database type (to call the proper Driver, that we already loaded in the first step)
 - ▶ The server address
 - ▶ Authentication information (user/pass)
 - ▶ Database / schema to connect to
- ▶ **All these parameters are encoded into a string**
 - ▶ The exact format depends on the Driver vendor



MySQL Connection URL format

- ▶ `jdbc:mysql://[host:port],[host:port].../[database][?propertyName1]=[propertyValue1][&propertyName2]=[propertyValue2]...`
- ▶ `jdbc:mysql://`
- ▶ `host:port` (`localhost`)
- ▶ `/database`
- ▶ `?user=username`
- ▶ `&password=pppppppp`



3. Establish the connection

- ▶ Use `DriverManager.getConnection`
 - ▶ Uses the appropriate driver according to the connection URL
 - ▶ Returns a `Connection` object
- ▶ `Connection connection = DriverManager.getConnection(URLString)`
- ▶ Contacts DBMS, validates user and selects the database
- ▶ On the `Connection` object subsequent commands will execute queries



Example

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

    try {
        Connection conn = DriverManager.getConnection(
"jdbc:mysql://localhost/test?user=monty&password=secret");

        // Do something with the Connection
        ....

    } catch (SQLException ex) {
        // handle any errors
        System.out.println("SQLException: " + ex.getMessage());
        System.out.println("SQLState: " + ex.getSQLState());
        System.out.println("VendorError: " + ex.getErrorCode());
    }
```



4. Create a Statement object

- ▶ `Statement statement = connection.createStatement() ;`
- ▶ Creates a Statement object for sending SQL statements to the database.
- ▶ SQL statements without parameters are normally executed using Statement objects.
 - ▶ If the same SQL statement is executed many times, it may be more efficient to use a PreparedStatement object.



5. Execute a query

- ▶ Use the `executeQuery` method of the `Statement` class
 - ▶ `ResultSet executeQuery(String sql)`
 - ▶ `sql` contains a `SELECT` statement
- ▶ Returns a `ResultSet` object, that will be used to retrieve the query results



Other execute methods

- ▶ `int executeUpdate(String sql)`
 - ▶ For INSERT, UPDATE, or DELETE statements
 - ▶ For other SQL statements that don't return a resultset (e.g., CREATE TABLE)
 - ▶ returns either the row count for INSERT, UPDATE or DELETE statements, or 0 for SQL statements that return nothing

- ▶ `boolean execute(String sql)`
 - ▶ For general SQL statements



Example

```
String query = "SELECT col1, col2, col3 FROM  
sometable" ;
```

```
ResultSet resultSet =  
statement.executeQuery(query) ;
```



6. Process the result

- ▶ The ResultSet object *implements a “cursor”* over the query results
 - ▶ Data are available a row at a time
 - ▶ Method `ResultSet.next()` goes to the next row
 - ▶ The column values (for the selected row) are available through **getXXX** methods
 - ▶ `getInt`, `getString`, ...
 - ▶ Data types are converted from SQL types to Java types



ResultSet.getXXX methods

- ▶ **XXX** is the desired datatype
 - ▶ Must be compatible with the column type
 - ▶ String is almost always acceptable
- ▶ **Two versions**
 - ▶ `getXXX(int columnIndex)`
 - ▶ number of column to retrieve (starting from 1!!!!)
 - ▶ `getXXX(String columnName)`
 - ▶ name of column to retrieve



ResultSet navigation methods

- ▶ `boolean next()`
 - ▶ Moves the cursor down one row from its current position.
 - ▶ A `ResultSet` cursor is initially positioned **before the first row**:
 - ▶ the first call to the method `next` makes the first row the current row
 - ▶ the second call makes the second row the current row, ...



Other navigation methods (1 / 2)

- ▶ **Query cursor position**
 - ▶ `boolean isFirst()`
 - ▶ `boolean isLast()`
 - ▶ `boolean isBeforeFirst()`
 - ▶ `boolean isAfterLast()`



Other navigation methods (2/2)

▶ Move cursor

- ▶ `void beforeFirst()`
- ▶ `void afterLast()`
- ▶ `boolean first()`
- ▶ `boolean last()`
- ▶ `boolean absolute(int row)`
- ▶ `boolean relative(int rows) // positive or negative offset`
- ▶ `boolean previous()`



Example

- ▶ while(resultSet.next())
- ▶ {
- ▶ out.println("<p>" +
- ▶ resultSet.getString(1) + " - " +
- ▶ resultSet.getString(2) + " - " +
- ▶ resultSet.getString(3) + "</p>") ;
- ▶ }



Datatype conversions (MySQL)

These MySQL Data Types	Can always be converted to these Java types
CHAR, VARCHAR, BLOB, TEXT, ENUM, and SET	<code>java.lang.String</code> , <code>java.io.InputStream</code> , <code>java.io.Reader</code> , <code>java.sql.Blob</code> , <code>java.sql.Clob</code>
FLOAT, REAL, DOUBLE PRECISION, NUMERIC, DECIMAL, TINYINT, SMALLINT, MEDIUMINT, INTEGER, BIGINT	<code>java.lang.String</code> , <code>java.lang.Short</code> , <code>java.lang.Integer</code> , <code>java.lang.Long</code> , <code>java.lang.Double</code> , <code>java.math.BigDecimal</code>
DATE, TIME, DATETIME, TIMESTAMP	<code>java.lang.String</code> , <code>java.sql.Date</code> , <code>java.sql.Timestamp</code>



7. Close the connection

- ▶ Additional queries may be done on the same connection.
 - ▶ Each returns a different ResultSet object, unless you re-use it
- ▶ When no additional queries are needed:
 - ▶ `connection.close()` ;



Connection pooling

- ▶ **Opening and closing DB connection is expensive**
 - ▶ Requires setting up TCP/IP connection, checking authorization, ...
 - ▶ After just 1-2 queries, the connection is dropped and all partial results are lost in the DBMS
- ▶ **Connection pool**
 - ▶ A set of “already open” database connections
 - ▶ JSP pages “lend” a connection for a short period, running queries
 - ▶ The connection is then returned to the pool (not closed!) and is ready for the next JSP/Servlet needing it

Support in J2EE and Tomcat

- ▶ The Java EE Platform Specification requires:
 - ▶ Java EE Application Servers must provide a *DataSource* implementation
 - ▶ DataSource is a connection pool for JDBC connections
 - ▶ Tomcat implements this specification
- ▶ DataSource – interface `javax.sql.DataSource`
 - ▶ Alternative to DriverManager
 - ▶ DataSource implementations can be located through JNDI (Java Naming and Directory)
 - ▶ Tomcat implements a simplified JNDI service

Configure JNDI

- ▶ Tomcat's JNDI is stored in WEB-INF/web.xml
- ▶ Define a resource to access a DataSource object, with a symbolic reference name

```
<resource-ref>
  <description>
    Resource reference to a factory for java.sql.Connection
    instances that may be used for talking to a particular
    database that is configured in the <Context> configuration
    for the web application.
  </description>

  <res-ref-name>jdbc/TestDB</res-ref-name>

  <res-type>javax.sql.DataSource</res-type>

  <res-auth>Container</res-auth>

</resource-ref>
```

Configure the connection factory

- ▶ Implementation instructions are stored in WEB-INF/context.xml

```
<Context ...>
  ...
  <Resource
    name="jdbc/TestDB"
    auth="Container"
    type="javax.sql.DataSource"
    maxActive="100"
    maxIdle="30"
    maxWait="10000"
    username="utente1" password="utente1"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost:3306/nazioni?autoReconnect
      =true"
  />
  ...
</Context>
```

Get a connection from the pool

- ▶ Lookup the DataSource, then get a new connection

```
/* JNDI query to locate the DataSource object */
Context initContext = new InitialContext();

Context envContext =
(Context)initContext.lookup("java:/comp/env") ; // JNDI
standard naming root

DataSource ds = (DataSource)envContext.lookup("jdbc/TestDB");

/* Ask DataSource for a connection */
Connection conn = ds.getConnection();

... use this connection to access the database ...

conn.close() ; // return connection to the pool
```

Benchmarks

	100 Iterations	100 Iterations	1000 Iterations	3000 Iterations
Pooling	547 ms	<10 ms	47 ms	31 ms ¹
Non-Pooling	4859 ms	4453 ms	43625 ms	134375 ms

The first time, the connections must be created

Second time, reuse connections

Negligible overhead

10x slower

No improvement

Linear increase

What's wrong with statements?

- ▶ `String sql = "select * from users where username='" + request.getParameter("username") + "'";`
- ▶ **Security risk**
 - ▶ SQL injection – syntax errors or privilege escalation
 - ▶ Username: `' ; delete * from users ; --`
 - ▶ **Must** detect or escape all dangerous characters!
- ▶ **Performance limit**
 - ▶ Query must be re-parsed and re-optimized every time
 - ▶ Complex queries require significant set-up overhead

Prepared statements

- ▶ **Separate statement creation from statement execution**
 - ▶ At creation time: define SQL syntax (**template**), with placeholders for variable quantities (**parameters**)
 - ▶ At execution time: define actual quantities for placeholders (**parameter values**), and run the statement
- ▶ **Prepared statements can be re-run many times**
- ▶ **Parameter values are automatically**
 - ▶ Converted according to their Java type
 - ▶ Escaped, if they contain dangerous characters
 - ▶ Handle non-character data (serialization)

Example

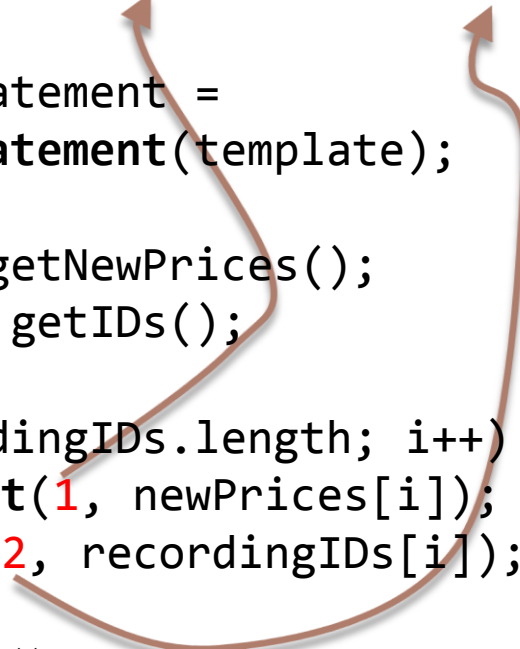
```
Connection connection =  
DriverManager.getConnection(url, username, password);
```

```
String template =  
"UPDATE music SET price = ? WHERE id = ?";
```

```
PreparedStatement statement =  
connection.prepareStatement(template);
```

```
float[] newPrices = getNewPrices();  
int[] recordingIDs = getIDs();
```

```
for(int i=0; i<recordingIDs.length; i++) {  
    statement.setFloat(1, newPrices[i]); // Price  
    statement.setInt(2, recordingIDs[i]); // ID  
  
    statement.execute();  
}
```



Callable statements

- ▶ Many DBMSs allow defining “stored procedures”, directly defined at the DB level
- ▶ Stored procedures are SQL queries (with parameters), or sequences of queries
 - ▶ Language for defining stored procedures is DBMS-dependent: not portable!
- ▶ MySQL: <http://dev.mysql.com/doc/refman/5.5/en/stored-programs-views.html> (chapter 18)
- ▶ Calling stored procedures: use CallableStatement in JDBC
 - ▶ Example: <http://dev.mysql.com/doc/refman/5.5/en/connector-j-usagenotes-basic.html#connector-j-examples-stored-procedure>

References

▶ JDBC Basics: Tutorial

- ▶ <http://java.sun.com/docs/books/tutorial/jdbc/basics/index.html>
- ▶ <http://pdf.coreservlets.com/Accessing-Databases-JDBC.pdf>

▶ JDBC reference guide

- ▶ <http://java.sun.com/javase/6/docs/technotes/guides/jdbc/getstart/GettingStartedTOC.fm.html>

▶ JDBC JavaDoc

- ▶ <http://java.sun.com/javase/6/docs/api/java/sql/package-summary.html>
- ▶ <http://java.sun.com/javase/6/docs/api/javax/sql/package-summary.html>

▶ Connection pooling

- ▶ Introduction: <http://www.datadirect.com/resources/jdbc/connection-pooling/index.html>
- ▶ with MySQL Connector/J: http://dev.mysql.com/tech-resources/articles/connection_pooling_with_connectorj.html
- ▶ <http://dev.mysql.com/doc/refman/5.5/en/connector-j-usagenotes-j2ee.html#connector-j-usagenotes-tomcat>
- ▶ Tomcat tutorial: <http://tomcat.apache.org/tomcat-5.5-doc/jndi-resources-howto.html#JDBC%20Data%20Sources>



Licenza d'uso



- ▶ Queste diapositive sono distribuite con licenza Creative Commons “Attribuzione - Non commerciale - Condividi allo stesso modo 2.5 Italia (CC BY-NC-SA 2.5)”
- ▶ Sei libero:
 - ▶ di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare quest'opera
 - ▶ di modificare quest'opera
- ▶ Alle seguenti condizioni:
 - ▶ **Attribuzione** — Devi attribuire la paternità dell'opera agli autori originali e in modo tale da non suggerire che essi avallino te o il modo in cui tu usi l'opera.
 - ▶ **Non commerciale** — Non puoi usare quest'opera per fini commerciali.
 - ▶ **Condividi allo stesso modo** — Se alteri o trasformi quest'opera, o se la usi per crearne un'altra, puoi distribuire l'opera risultante solo con una licenza identica o equivalente a questa.
- ▶ <http://creativecommons.org/licenses/by-nc-sa/2.5/it/>

