FREE eBook

LEARNING

jdbc

Free unaffiliated eBook created from
Stack Overflow contributors.

#jdbc

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: jdbc

It is an unofficial and free jdbc ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official jdbc.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with jdbc

Remarks

JDBC, or Java DataBase Connectivity, is the Java specification for connecting to (relational) databases. JDBC provides a common API in the form of a number of interfaces and exceptions, and expectations (or requirements) of drivers.

The JDBC specification consists of two parts:

1. A specification document, available from the JSR-221 page
2. The API and its documentation, included with the Java SE API (packages `java.sql` and `javax.sql`)

Most relational databases, and some non-relational databases, provide a driver that implements the JDBC.

Versions

Version	Release Date
3.0	2002-02-06
4.0	2006-12-11
4.1	2011-07-07
4.2	2014-03-18

Examples

Creating a connection

To be able to use JDBC you need to have the JDBC driver of your database on the class path of your application.

There are multiple ways to connect to a database, but the common ways are to either use the `java.sql.DriverManager`, or to configure and use a database specific implementation of `javax.sql.DataSource`.

A simple example to create a connection to a database with the url `jdbc:somedb://localhost/foobar` and execute an update statement to give all employees a 5% raise:

```
try (Connection connection = DriverManager.getConnection(
        "jdbc:somedb://localhost/foobar", "anna", "supersecretpassword");
```

```
     Statement updateStatement = connection.createStatement()) {

    updateStatement.executeUpdate("update employees set salary = salary * 1.05");
}
```

For further details see creating a database connection

Read Getting started with jdbc online: https://riptutorial.com/jdbc/topic/1685/getting-started-with-jdbc

Chapter 2: Creating a database connection

Syntax

- DB_URL = "jdbc:DBMS://DB_HOST:DB_PORT/DB_NAME"
- DBMS: Data Base Driver Manager, this can be any DBMS (mysql, oracle, postgresql, sqlite, ...), exemple of mysql: "com.mysql.jdbc.Driver"
- DB_HOST: your database base host, the IP adress of your database exemple : 10.6.0.1, the default is localhost or 127.0.0.1
- DB_PORT: Database port, every DBMS has a defeaut port exemple mysql=3306, postegesql=5432
- DB_NAME: the name of your Database
- To connect you should to obtains a reference to the class object,
- Class.forName(DRIVER);
- And to connect to database, you need to create a connection
- java.sql.Connection con = DriverManager.getConnection(DB_URL, DB_USER_NAME, DB_PASSWORD);
- DB_USER_NAME : the username of your databse
- DB_PASSWORD : the password of your database

Examples

Introduction (SQL)

Since Java 6, the recommended way to access an SQL-based database in Java is via the JDBC(Java DataBase Connectivity) API.

This API comes in two packagages: `java.sql` and `javax.sql`.

JDBC defines database interactions in terms of `Connections` and `Drivers`.

A `Driver` interacts with the database, and provides a simplified interface for opening and managing connections. Most database server varieties (PostgreSQL, MySQl, etc.) have their own `Drivers`, which handle setup, teardown, and translation specific to that server. `Drivers` are usually not accessed directly; rather, the interface provided by the `DriverManager` object is used instead.

The `DriverManager` object is essentially the core of JDBC. It provides a (mostly) database-agnostic interface to create `Connections`. For older versions of the JDBC API, database-specific `Drivers` had

to be loaded before `DeviceManager` could create a connection to that database type.

A `Connection` is, as the name implies, a representation of an open connection to the database. `Connections` are database-agnostic, and are created and provided by the `DriverManager`. They provide a number of 'shortcut' methods for common query types, as well as a raw SQL interface.

Using the Connection (And Statements)

Once we've gotten the `Connection`, we will mostly use it to create `Statement` objects. `Statements` represent a single SQL transaction; they are used to execute a query, and retrieve the results (if any). Let's look at some examples:

```
public void useConnection() throws SQLException{


    Connection conn = getConnection();

    //We can use our Connection to create Statements
    Statement state = conn.getStatement();

    //Statements are most useful for static, "one-off" queries

    String query = "SELECT * FROM mainTable";
    boolean sucess = state.execute(query);

    //The execute method does exactly that; it executes the provided SQL statement, and
returns true if the execution provided results (i.e. was a SELECT) and false otherwise.

    ResultSet results = state.getResultSet();

    //The ResultSet object represents the results, if any, of an SQL statement.
    //In this case, the ResultSet contains the return value from our query statement.
    //A later example will examine ResultSets in more detail.

    ResultSet newResults = state.executeQuery(query)

    //The executeQuery method is a 'shortcut' method. It combines the execute and getResultSet
methods into a single step.
    //Note that the provided SQL query must be able to return results; typically, it is a
single static SELECT statement.
    //There are a number of similar 'shortcut' methods provided by the Statement interface,
including executeUpdate and executeBatch

    //Statements, while useful, are not always the best choice.

    String newQuery = "SELECT * FROM mainTable WHERE id=?";
    PreparedStatement prepStatement = conn.prepareStatement(newQuery);

    //PreparedStatements are the prefed alternative for variable statements, especially ones
that are going to be executed multiple times

    for(int id:this.ids){

        prepStatement.setInt(1,id);
        //PreparedStatements allow you to set bind variables with a wide variety of set
methods.
        //The first argument to any of the various set methods is the index of the bind
variable you want to set. Note that this starts from 1, not 0.
```

```
        ResultSet tempResults = prepStatement.executeQuery()
        //Just like Statements, PreparedStatements have a couple of shortcut methods.
        //Unlike Statements, PreparedStatements do not not take a query string as an argument
to any of their execute methods.
        //The statement that is executed is always the one passed to the
Connector.prepareStatement call that created the PreparedStatement
    }

}
```

Creating a connection using java.sql.DriverManager

To connect using `java.sql.DriverManager` you need a JDBC url to connect to your database. JDBC urls are database specific, but they are all of the form

```
jdbc:<subprotocol>:<subname>
```

Where `<subprotocol>` identifies the driver or database (for example `postgresql`, `mysql`, `firebirdsql`, etc), and `<subname>` is subprotocol-specific.

You need to check the documentation of your database and JDBC driver for the specific url subprotocol and format for your driver.

A simple example to create a connection to a database with the url `jdbc:somedb://localhost/foobar`:

```
try (Connection connection = DriverManager.getConnection(
        "jdbc:somedb://localhost/foobar", "anna", "supersecretpassword")) {
    // do something with connection
}
```

We use a try-with-resources here so the connection is automatically closed when we are done with it, even if exceptions occur.

4.0

On Java 6 (JDBC 4.0) and earlier, try-with-resources is not available. In those versions you need to use a `finally`-block to explicitly close a connection:

```
Connection connection = DriverManager.getConnection(
        "jdbc:somedb://localhost/foobar", "anna", "supersecretpassword");
try {
    // do something with connection
} finally {
    // explicitly close connection
    connection.close();
}
```

4.0

JDBC 4.0 (Java 6) introduced the concept of automatic driver loading. If you use Java 5 or earlier, or an older JDBC driver that does not implement JDBC 4 support, you will need to explicitly load

the driver(s):

```
Class.forName("org.example.somedb.jdbc.Driver");
```

This line needs to occur (at least) once in your program, before any connection is made.

Even in Java 6 and higher with a JDBC 4.0 it may be necessary to explicitly load a driver: for example in web applications when the driver is not loaded in the container, but as part of the web application.

Alternatively you can also provide a `Properties` object to connect:

```
Properties props = new Properties();
props.setProperty("user", "anna");
props.setProperty("password", "supersecretpassword");
// other, database specific, properties
try (Connection connection = DriverManager.getConnection(
        "jdbc:somedb://localhost/foobar", props)) {
    // do something with connection
}
```

Or even without properties, for example if the database doesn't need username and password:

```
try (Connection connection = DriverManager.getConnection(
        "jdbc:somedb://localhost/foobar")) {
    // do something with connection
}
```

Creating a connection to MySQL

To connect to MySQL you need to use the MySQL Connector/J driver. You can download it from http://dev.mysql.com/downloads/connector/j/ or you can use Maven:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.39</version>
</dependency>
```

The basic JDBC URL for MySQL is:

```
jdbc:mysql://<hostname>[:<port>]/<database>[?<propertyName>=<propertyValue>[&<propertyName>=<propertyVa
```

Where:

Key	Description	Example
`<hostname>`	Host name of the MySQL server	`localhost`
`<port>`	Port of the MySQL server (optional, default: 3306)	`3306`

Key	Description	Example
`<database>`	Name of the database	`foobar`
`<propertyName>`	Name of a connection property	`useCompression`
`<propertyValue>`	Value of a connection property	`true`

The supported URL is more complex than shown above, but this suffices for most 'normal' needs.

To connect use:

```
try (Connection connection = DriverManager.getConnection(
        "jdbc:mysql://localhost/foobardb", "peter", "nicepassword")) {
    // do something with connection
}
```

4.0

For older Java/JDBC versions:

4.0

```
// Load the MySQL Connector/J driver
Class.forName("com.mysql.jdbc.Driver");
```

```
Connection connection = DriverManager.getConnection(
        "jdbc:mysql://localhost/foobardb", "peter", "nicepassword");
try {
    // do something with connection
} finally {
  // explicitly close connection
  connection.close();
}
```

Connection to a Microsoft Access database with UCanAccess

UCanAccess is a pure Java `JDBC` driver that allows us to read from and write to Access databases without using `ODBC`. It uses two other packages, `Jackcess` and `HSQLDB`, to perform these tasks.

Once it has been set up*, we can work with data in .accdb and .mdb files using code like this:

```
import java.sql.*;

Connection conn=DriverManager.getConnection("jdbc:ucanaccess://C:/__tmp/test/zzz.accdb");
Statement s = conn.createStatement();
ResultSet rs = s.executeQuery("SELECT [LastName] FROM [Clients]");
while (rs.next()) {
    System.out.println(rs.getString(1));
}
```

*For more details see the following question:

Manipulating an Access database from Java without ODBC

Oracle JDBC connection

Driver:

- 12c R1
- 11g R2

(**Note:** the driver is not included in Maven Central!)

Driver class initialization:

```
    Class.forName("oracle.jdbc.driver.OracleDriver");
```

Connection URL

Older format, with SID

```
"jdbc:oracle:thin:@<hostname>:<port>:<SID>"
```

Newer format, with Service Name

```
"jdbc:oracle:thin:@//<hostname>:<port>/<servicename>"
```

Tnsnames like entry

```
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS=(PROTOCOL=TCPS)(HOST=<hostname>)(PORT=<port>))"
    +"(CONNECT_DATA=(SERVICE_NAME=<servicename>)))"
```

RAC cluster connection string for failover

```
"jdbc:oracle:thin:@(DESCRIPTION=(ADDRESS_LIST=(LOAD_BALANCE=OFF)(FAILOVER=ON)"
    +"(ADDRESS=(PROTOCOL=TCP)(HOST=<hostname1>)(PORT=<port1>))"
    +"(ADDRESS=(PROTOCOL=TCP)(HOST=<hostname2>)(PORT=<port2>)))"
    +"(CONNECT_DATA=SERVICE_NAME=<servicename>)(SERVER=DEDICATED)))"
```

Example

```
connection = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:orcl", "HR",
"HRPASS");
```

Read Creating a database connection online: https://riptutorial.com/jdbc/topic/2640/creating-a-database-connection

Chapter 3: JDBC - Statement Injection

Introduction

SQL injection is a code injection technique, used to attack data-driven applications, in which nefarious SQL statements are inserted into an entry field for execution (e.g. to dump the database contents to the attacker).

In this section we will talk about that and its relation with JDBC Statement.

Examples

Statement & SQL Injection evil

Note in this example we will use PostgreSQL DBMS, but you can use any DBMS

We will use a database `bd_test` witch contain a `Schema: sch_test` and two tables `users` and `test` :

```
CREATE TABLE sch_test.users
(
  id serial NOT NULL,
  username character varying,
  password character varying,
  CONSTRAINT utilisateur_pkey PRIMARY KEY (id)
)

CREATE TABLE sch_test.test
(
  id serial NOT NULL,
  "column" character varying
)
```

Simple login using Statement

```
static String DRIVER = "org.postgresql.Driver";
static String DB_USERNAME = "postgres";
static String DB_PASSWOR = "admin";
static String DB_URL = "jdbc:postgresql://localhost:5432/bd_test";

public static void sqlInjection() {
    try {
        Class.forName(DRIVER);
        Connection connection = DriverManager.getConnection(DB_URL, DB_USERNAME, DB_PASSWOR);
        Statement statement = connection.createStatement();
        String username = "admin";
        String password = "admin";
        String query = "SELECT * FROM sch_test.users where username = '"
                + username + "' and password = '" + password + "'";

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

```
        if (result.next()) {
            System.out.println("id = " + result.getInt("id") + " | username = "
                    + result.getString("username") + " | password = " +
result.getString("password"));
        }else{
           System.out.println("Login not correct");
        }

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

Until now every thing is normal and secure.

Login with fake username and password

The hacker or the tester can simply login or list all your users using this :

```
String username = " ' or ''='";
String password = " ' or ''='";
```

INSERT a new user

You can insert data in your table using :

```
String username = "'; INSERT INTO sch_test.utilisateur(id, username, password)
                        VALUES (2, 'hack1', 'hack2');--";
String password = "any";
```

DELETE All users

consider the hacker know the schema of your database so he can delete all your user

```
String username = "'; DELETE FROM sch_test.utilisateur WHERE id>0;--";
String password = "any";
```

DROP Table users

The hacker can also delete your table

```
String username = "'; drop table sch_test.table2;--";
String password = "any";
```

DROP DATABASE

The worst is to drop the database

```
String username = "'; DROP DATABASE bd_test;--";
String password = "any";
```

and there are many others.

Why all this?

All this because `Statement` is not secure enough it execute the query like is it, for that it is recommend to use `PreparedStatement` instead, it is more secure that `Statement`.

You can find here more details PreparedStatement

Read JDBC - Statement Injection online: https://riptutorial.com/jdbc/topic/9238/jdbc---statement-injection

Chapter 4: PreparedStatement

Remarks

A `PreparedStatement` declares the statement before it is executed, and allows for placeholders for parameters. This allows the statement to be prepared (and optimized) once on the server, and then reused with different sets of parameters.

The added benefit of the parameter placeholders, is that it provides protection against SQL injection. This is achieved either by sending the parameter values separately, or because the driver escapes values correctly as needed.

Examples

Setting parameters for PreparedStatement

Placeholders in the query string need to be set by using the `set*` methods:

```
String sql = "SELECT * FROM EMP WHERE JOB = ? AND SAL > ?";

//Create statement to make your operations
PreparedStatement statement = connection.prepareStatement(sql);

statement.setString(1, "MANAGER"); // String value
statement.setInt(2, 2850);         // int value
```

Special cases

Setting NULL value:

Setting a null value can not be accomplished using for example the `setInt` and `setLong` methods, as these use primitive types (`int` and `long`) instead of objects (`Integer` and `Long`), and would cause a `NullPointerException` to be thrown:

```
void setFloat(int parameterIndex, float x)
void setInt(int parameterIndex, int x)
void setLong(int parameterIndex, long x)
```

These cases can be handled by using `setNull`.

```
setNull(int parameterIndex, int sqlType)
```

It is typed, so the second parameter has to be provided, see java.sql.Types

```
//setting a NULL for an integer value
statement.setNull(2, java.sql.Types.INTEGER);
```

Setting LOBs

LOBs require special objects to be used.

```
Clob longContent = connection.createClob();
Writer longContentWriter = longContent.setCharacterStream(1); // position: beginning
longContentWriter.write("This will be the content of the CLOB");

pstmt = connection.prepareStatement("INSERT INTO CLOB_TABLE(CLOB_VALUE) VALUES (?)");
pstmt.setClob(1, longContent);
```

Exceptions on `set*` methods

> `SQLException` - if `parameterIndex` does not correspond to a parameter marker in the SQL statement; if a database access error occurs or this method is called on a closed `PreparedStatement`
>
> `SQLFeatureNotSupportedException` - if sqlType is a `ARRAY`, `BLOB`, `CLOB`, `DATALINK`, `JAVA_OBJECT`, `NCHAR`, `NCLOB`, `NVARCHAR`, `LONGNVARCHAR`, `REF`, `ROWID`, `SQLXML` or `STRUCT` data type and the JDBC driver does not support this data type

Basic usage of a prepared statement

This example shows how to create a prepared statement with an insert statement with parameters, set values to those parameters and then executing the statement.

```
Connection connection = ... // connection created earlier
try (PreparedStatement insert = connection.prepareStatement(
        "insert into orders(id, customerid, totalvalue, comment) values (?, ?, ?, ?)")) {
    //NOTE: Position indexes start at 1, not 0
    insert.setInt(1, 1);
    insert.setInt(2, 7934747);
    insert.setBigDecimal(3, new BigDecimal("100.95"));
    insert.setString(4, "quick delivery requested");

    insert.executeUpdate();
}
```

The question marks (`?`) in the insert statement are the parameter placeholders. They are positional parameters that are later referenced (using a 1-based index) using the `setXXX` methods to set values to those parameters.

The use of try-with-resources ensures that the statement is closed and any resources in use for that statement are released.

Read PreparedStatement online: https://riptutorial.com/jdbc/topic/2939/preparedstatement

Chapter 5: ResultSet

Introduction

A ResultSet object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The next method moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result se

Examples

ResultSet

To create a `ResultSet` you should to create a `Statement` or `PrepapredStatement` :

Create ResultSet with Statement

```
try {
    Class.forName(driver);
    Connection connection = DriverManager.getConnection(
            "jdbc:somedb://localhost/databasename", "username", "password");
    Statement statement = connection.createStatement();
    ResultSet result = statement.executeQuery("SELECT * FROM my_table");

} catch (ClassNotFoundException | SQLException e) {
}
```

Create ResultSet with PrepapredStatement

```
try {
    Class.forName(driver);
    Connection connection = DriverManager.getConnection(
            "jdbc:somedb://localhost/databasename", "username", "password");
    PreparedStatement preparedStatement = connection.prepareStatement("SELECT * FROM
my_table");
    ResultSet result = preparedStatement.executeQuery();

} catch (ClassNotFoundException | SQLException e) {
}
```

Check if your ResultSet have information or not

```
if (result.next()) {
   //yes result not empty
}
```

Get information from ResultSet

There are several type of information you can get from your `ResultSet` like `String`, `int`, `boolean`, `float`, `Blob`, ... to get information you had to use a loop or a simple if :

```
if (result.next()) {
    //get int from your result set
    result.getInt("id");
    //get string from your result set
    result.getString("username");
    //get boolean from your result set
    result.getBoolean("validation");
    //get double from your result set
    result.getDouble("price");
}
```

Read ResultSet online: https://riptutorial.com/jdbc/topic/9172/resultset

Chapter 6: ResultSetMetaData

Introduction

As we all know Metadata mean data about data.

To fetch metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

Examples

ResultSetMetaData

```
import java.sql.*;

class Rsmd {

    public static void main(String args[]) {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection(
                    "jdbc:oracle:thin:@localhost:1521:xe", "system", "oracle");

            PreparedStatement ps = con.prepareStatement("select * from emp");
            ResultSet rs = ps.executeQuery();
            ResultSetMetaData rsmd = rs.getMetaData();

            System.out.println("Total columns: " + rsmd.getColumnCount());
            System.out.println("Column Name of 1st column: " + rsmd.getColumnName(1));
            System.out.println("Column Type Name of 1st column: " +
rsmd.getColumnTypeName(1));

            con.close();
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Read ResultSetMetaData online: https://riptutorial.com/jdbc/topic/10126/resultsetmetadata

Chapter 7: Statement batching

Introduction

Statement batching is either executing multiple statements as one unit (with a normal `java.sql.Statement`), or a single statement with multiple sets of parameter values (with a `java.sql.PreparedStatement`).

Remarks

Statement batching allows a program to collect related statement, or in the case of prepared statements related parameter value sets, and send them to the database server as a single execute.

The benefits of statement batching can include improved performance. If and how these performance benefits are achieved depends on the driver and database support, but they include:

- Sending all statements (or all values sets) in one command
- Rewriting the statement(s) so they can be executed like one big statement

Examples

Batch insertion using PreparedStatement

Batch execution using `java.sql.PreparedStatement` allows you to execute a single DML statement with multiple sets of values for its parameters.

This example demonstrates how to prepare an insert statement and use it to insert multiple rows in a batch.

```
Connection connection = ...; // obtained earlier
connection.setAutoCommit(false); // disabling autoCommit is recommend for batching
int orderId = ...; // The primary key of inserting and order
List<OrderItem> orderItems = ...; // Order item data

try (PreparedStatement insert = connection.prepareStatement(
        "INSERT INTO orderlines(orderid, itemid, quantity) VALUES (?, ?, ?)")) {
    // Add the order item data to the batch
    for (OrderItem orderItem : orderItems) {
        insert.setInt(1, orderId);
        insert.setInt(2, orderItem.getItemId());
        insert.setInt(3, orderItem.getQuantity());
        insert.addBatch();
    }

    insert.executeBatch();//executing the batch
}

connection.commit();//commit statements to apply changes
```

Batch execution using Statement

Batch execution using `java.sql.Statement` allows you to execute multiple DML statements (`update`, `insert`, `delete`) at once. This is achieved by creating a single statement object, adding the statements to execute, and then execute the batch as one.

```
Connection connection = ...; // obtained earlier
connection.setAutoCommit(false); // disabling autocommit is recommended for batch execution

try (Statement statement = connection.createStatement()) {
    statement.addBatch("INSERT INTO users (id, username) VALUES (2, 'anna')");
    statement.addBatch("INSERT INTO userrole(userid, rolename) VALUES (2, 'admin')");

    statement.executeBatch();//executing the batch
}

connection.commit();//commit statements to apply changes
```

Note:

`statement.executeBatch();` will return `int[]` to hold returned values, you can execute your batch like this :

```
int[] stmExc = statement.executeBatch();//executing the batch
```

Read Statement batching online: https://riptutorial.com/jdbc/topic/2992/statement-batching

Credits

S. No	Chapters	Contributors
1	Getting started with jdbc	Community, Mark Rotteveel, YCF_L
2	Creating a database connection	F. Stephen Q, Gherbi Hicham, Gord Thompson, Mark Rotteveel , ppeterka, YCF_L
3	JDBC - Statement Injection	YCF_L
4	PreparedStatement	Gord Thompson, Gus, Mark Rotteveel, ppeterka, YCF_L
5	ResultSet	KIRAN KUMAR MATAM, YCF_L
6	ResultSetMetaData	KIRAN KUMAR MATAM, YCF_L
7	Statement batching	KIRAN KUMAR MATAM, Mark Rotteveel, ppeterka, YCF_L