Access to databases (JDBC)

Juan M. Gimeno, Josep M. Ribó

February, 2008

Access to databases: JDBC

Access to databases: JDBC

- The JDBC approach to database access
- Driver loading
- Database connection
- Database manipulation
- Result management
- Errors
- Examples
- Limitations of the presented approach
- Appendix: The PostgreSQL DBMS

The Java approach to database connection

The JDBC technology provides two packages:

- `java.sql` and
- `javax.sql`

which contain interfaces and classes that support the connection of java applications (including web applications) to databases

- **The vendors of database management systems**:

  Provide a specific **JDBC driver** which implements the interfaces defined in the packages `java.sql` and `javax.sql` according to the specific requirements of their database management system

- **The java programmers**:

  1. **Charge the JDBC driver** specific for a database management system vendor and
  2. **Access the database through the operations defined in the** `java.sql/javax.sql` **packages** which will have been implemented in that driver

Specific procedure to access a database via JDBC

The procedure to connect a java application to a database can be synthesized as follows (in **JDBC 1.x**):

1. Charge dynamically the specific JDBC driver

   (it is provided by the database manager system vendor)

```
Class.forName("specificJDBCDriver");
```

2. Open a connection to a database

```
Connection con=
    DriverManager.getConnection("jdbc:DatabaseURL");
```

3. Create a SQL statement and process it

```
Statement stm=con.createStatement();
rs=stm.executeQuery("Select xxx from table.....");
```

4. Process the result set:

```
while (rs.next()){
   name=rs.getString("name");
   amount=rs.getInt("amount");
}
```

JDBC 2.0 supports further abstraction:

The database can be named with a logical name without referring to the JDBC driver used

This permits a higher level access to the database

```
InitialContext ctx= new InitialContext();

DataSource ds=(DataSource)
    ctx.lookup("logicalNameOfTheDataSource");
```

This will be studied in the **Design patterns for web applications** module (pattern **Service Locator**)

The `java.sql` and `javax.sql` packages

The most useful interfaces and classes of these packages are:

- `Connection`
  - A connection contains an active link to a database.
  - It is an abstraction of an access session to a specific database
  - A Java application can access a database through this connection
  - A connection to a database can be obtained by:
    `DriverManager.getConnection()`
    or
    `DataSource.getConnection() (JDBC 2.0)`

- `Statement`

  It encapsulates a SQL instruction and sends it through the connection

  A statement is created by: `Connection.createStatetment()`

- `ResultSet`

  A set of rows which is the result of some query to the database.

  It is obtained by:

  `Statement.executeQuery(sqlQueryString)`

- `DatabaseMetaData`

  Interface that describes the structure of a database

  It is obtained by calling `con.getMetaData()` (on a connection object `con`)

- `ResultSetMetaData`

  Interface that describes the structure of a result set

  It is obtained by calling `rs.getMetaData()` on a result set `rs`

- `DriverManager`

  This interface registers the JDBC drivers and provides some methods to get a connection to a database:

  `DriverManager.getConnection()`

- `DataSource`

  A factory that provides connections to a database in a more abstract (higher-level) way than `DriverManager`

  The `DataSource` interface is implemented by a driver vendor.

  An object that implements the DataSource interface will typically be registered with a naming service based on the Java Naming and Directory (JNDI) API (see **Design patterns for web applications** module (**Service Locator** Pattern)

```
InitialContext ctx= new InitialContext();

DataSource ds=(DataSource)
    ctx.lookup("logicalNameOfTHeDataSource");
```

Access to a database. Steps in detail

1. Driver loading
2. Database connection
3. Database manipulation
   - Queries
   - Insetions, updates, deletions
   - Table and index creation
4. Result management
5. Errors

1. Driver loading

A java program may connect with a database controled by a specific DBMS (e.g., mySQL, Postgresql...) by means of a *JDBC driver*.
A *JDBC driver* is **a java class** which:

(a) implements the interface *java.sql.Driver*

(b) is provided by the DBMS vendor

Before operating with a database, a java program must **load the JDBC driver**

1. Driver loading: How is a driver loaded?

A java program may load a JDBC driver by means of the sentence:
`Class.forName(nameOfTheClassDriver);`
For example:
`Class.forName("org.postgresql.Driver");`
`org.postgresql.Driver` is the name of a java class that contains the JDBC driver provided by the DBMS called *Postgresql*

The sentence `Class.forName("org.postgresql.Driver")`:

- Locates and loads the class named "org.postgresql.Driver"
- Initizalizes it.
- The initialization of the class includes:
  1. The creation of an instance of the driver class and
  2. Its registration as a driver to be managed by the `DriverManager` class

1. Driver loading: How is a driver found?

That is: How can the sentence
`Class.forName("org.postgresql.Driver")`
locate and load the driver?

(a) Finding a JDBC driver in usual java applications

The environment variable `CLASSPATH` should contain the directory where the driver class is located.

Example

The driver class is contained in a jar file called `postgresql-8.2dev-500.jdbc3.jar` in the directory `/usr/local/postgres/lib`

CLASSPATH should contain
`/usr/local/postgres/lib/postgresql-8.2dev-500.jdbc3.jar`

1. Driver loading: How is a driver found? (2)

(b) Finding a JDBC driver in web applications

- When Tomcat is initialized creates some *class loader* objects.

- When the load of a class is required, these class loaders search it in the following directories (among others):
  - `WEB-INF/classes` of your web application
  - `WEB-INF/lib/*.jar` of your web application
  - `TOMCAT_HOME/common/classes` (Tomcat 5.x)
  - `TOMCAT_HOME/common/endorsed/*.jar` (Tomcat 5.x)
  - `TOMCAT_HOME/common/lib/*.jar` (Tomcat 5.x)
  - `TOMCAT_HOME/shared/classes` (Tomcat 5.x)
  - `TOMCAT_HOME/shared/lib/*.jar` (Tomcat 5.x)
  - `TOMCAT_HOME/lib/*.jar` (Since Tomcat 6.0.x)

In particular, in Tomcat 5.x, the `shared` directory is the place to put those classes that may be accessed by any web application.

Therefore, we may put the JDBC driver class at:

1. `TOMCAT_HOME/shared/lib` (in Tomcat 5.5.x)
2. `TOMCAT_HOME/lib` (in Tomcat 6.0.x)
3. `WEB-INF/lib`

With 1 and 2 the JDBC driver will be accessible for all web applications

With 3 the JDBC driver will be accessible only for this web application

1. Driver loading: What kind of errors may be raised during driver loading?

`ClassNotFoundException`

This exception class is raised by `forName()` in the case that the JDBC driver class is not found by the class loader

Check that the JDBC driver class has been correctly located and (if it is not a web application) that the `CLASSPATH` variable is appropriately set

1. Driver loading: Example

```
<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<%@ page import="java.util.*" %>

<html>
<head>
<title> Resultats electorals </title>
</head>
<body>
<h2> Resultats electorals </h2>
<%
  String DRIVER= "org.postgresql.Driver";
  try{
   Class c=Class.forName(DRIVER);

    ......
  }
  catch (ClassNotFoundException e) {
     out.println("Driver class not found!!!");
  }
.......
</html>
```

2. Database connection

Once the JDBC driver has been loaded, it can be used to provide a connection to a database defined within the DBMS to which the driver is associated

The operation `getConnection(...)` of the class `DriverManager` is responsible for providing such a connection

```
public static Connection getConnection(String databaseURL)
```

- Attempts to establish a connection to the database identified by the database URL.
- The DriverManager attempts to select an appropriate driver from the set of registered JDBC drivers.
- It returns a connection to the database `databaseURL`. This connection is an object of the interface $java.sql.Connection$.

  The connection object encapsulates a session. At the end of the session the conection must be closed

This can only be done if the driver has been loaded first (when the driver is loaded it is also registered by the `DriverManager` class)

Another operation to establish a connection to a database:

```
public static Connection
        getConnection(String databaseURL,
                      String user,
                      String password)
```

Parameters:

- databaseURL - a database url
- user - the database user on whose behalf the connection is being made
- password - the user's password
- Returns: a connection to the URL

2. Database connection: URLs to identify a database

Databases are identified by means of a 3-tuple:

`protocol:subprotocol:databaseIdentifier`

- **Protocol:** `jdbc`

- **Subprotocol:** The identifier of the JDBC driver. Specific for each DBMS.

  The driver manager uses this string to select the correct driver which will provide the connection E.g.: PostgreSQL: `postgresql`

- **Database identifier:** It contains all the information needed by the driver to identify the database

  E.g.: PostgreSQL: `host:port:database name`

  - *host:* URL of the server where the database is located: `//ingrid.udl.net`
  - *port:* The port number the server is listening on (default: 5432)
  - *database name:* (database identifier within the server) Ex: `Partits`

Summing up:

```
jdbc:postgresql://ingrid.udl.net:5432/Partits
```

The connection is achieved by:

```
Connection con=DriverManager.getConnection
   ("jdbc:postgresql://ingrid.udl.net:5432/Partits");
```

2. Database connection: Connection permissions

In order to get the connection it is necessary to configure properly the DBMS so that the database to which the connection is established accepts the connection requests from the java application

Example:

In the `PostgreSQL` DBMS this means to put the right permissions in the configuration file `pg_hba.conf` associated to the database to which we want to make the connection

For example:

```
TYPE     DATABASE    IP_ADDRESS    MASK                AUTH_TYPE  AUTH_ARGUME

local      all                                          trust
host       all         127.0.0.1     255.255.255.255    trust
```

This configuration allows any local user to connect with any PostgreSQL username, over either UNIX domain sockets or IP without any need of authentication.

2. Database connection: Closing the connection

A connection to a database acts as a session. When the session finishes the connection should be closed:

```
con.close()
```

2. Database connection: Example

```
<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<%@ page import="java.util.*" %>
<html>
<%
  String DRIVER= "org.postgresql.Driver";
  String URL=
     "jdbc:postgresql://ingrid.udl.net:5432/Partits";
  Connection con=null;
  try{
    Class c=Class.forName(DRIVER);
    con=DriverManager.getConnection(URL,"josepma","");
     .........
 }
catch (Exception e) {
     out.println(e.getMessage());
}
finally{
  if (con!=null) {con.close();}
}
```

3. Database manipulation

Once we have established a connection with a database, we can start its manipulation

This means:

- Queries
- Insetions, updates and deletions of rows (records) of a table
- Table and index creation

These kinds of operations are performed in relational databases by means of the language SQL

In the following slides we will show how these operations can be performed from a java applications which uses the JDBC technology

3. Database manipulation: *java.sql.Statement*

java.sql.Statement provides an object-oriented interface to manipulate a relational database following the SQL language Any database manipulation operation will be performed through a *Statement* object

Creation of a Statement

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

Kinds of statements

The interface *Statement* provides 4 methods intended to manipulate the database:

- `executeQuery`: To execute `SELECT` SQL instructions
- `executeUpdate`: To execute `INSERT`, `UPDATE`, `DELETE` SQL instructions
- `execute`: More general method that can be used to execute any SQL instruction. In practice it is used when a SQL instruction may return multiple results
- `executeBatch`: Submits a batch of commands to the database for execution

We will use `executeQuery` and `executeUpdate` mostly

3. Database manipulation: `executeQuery`

`public ResultSet executeQuery(String sql)`

- Executes the given SQL statement on the database to which this statement is connected
- The data produced by the query execution is returned in a single ResultSet object

Example:

```
String sql=""
 +" select sigles,nvots "
 +" from partit"
 +" order by nvots desc";

Statement st= con.createStatement();
ResultSet rs=st.executeQuery(sql);
```

3. Database manipulation: `executeUpdate`

```
public int executeUpdate(String sql)
```

- Executes the given SQL statement on the database to which this statement is connected.
- The statement may be an INSERT, UPDATE, or DELETE statement or an SQL statement that returns nothing, such as an SQL DDL statement (e.g., `CREATE TABLE`, `CREATE INDEX`)
- Returns either the row count for INSERT, UPDATE or DELETE statements, or 0 for SQL statements that return nothing

Examples:

```
String sql=""
 +" insert into partit "
 +" values ('"
 +paramnom+"','"
 +paramsigles+"',0)";

Statement st= con.createStatement();
int j=st.executeUpdate(sql);
```

```
String sql=""
 +" update partit "
 +" set nvots= nvots+"
 +nvots
 +" where sigles='"+paramsigles+"'";


Statement st= con.createStatement();
int j=st.executeUpdate(sql);
```

3.1 Database Manipulation:

java.sql.PreparedStatement

```
PreparedStatement ps = con.prepareStatement(
    "insert into partit  VALUES (?, ?, ?)");
ps.setString(1, "Mentiders Units");
ps.setString(2, "MU");
ps.setInt(3, 0);
int j=ps.executeUpdate();
```

4. Result management: $java.sql.ResultSet$

The results of a call to `executeQuery` are returned in an object which implements the interface $java.sql.ResultSet$ An object of type $ResultSet$ encapsulates a sorted list that contains the rows of a database table that have been obtained from the processing of the query (`executeQuery(...)`)

Moving through the result set

The interface $ResultSet$ provides several methods to move along the list of the rows returned by the process of the query:

- `boolean absolute(int row)`
  Moves the cursor to the given row number in this ResultSet object.
- `void afterLast()`
  Moves the cursor to the end of this ResultSet object, just after the last row.
- `void beforeFirst()`
  Moves the cursor to the front of this ResultSet object, just before the first row.
- `boolean first()`
  Moves the cursor to the first row in this ResultSet object.

- `int getRow()`
  Retrieves the current row number.
- `boolean isAfterLast()`
  Retrieves whether the cursor is after the last row in this ResultSet object.
- `boolean isBeforeFirst()`
  Retrieves whether the cursor is before the first row in this ResultSet object.
- `boolean isFirst()`
  Retrieves whether the cursor is on the first row of this ResultSet object.
- `boolean isLast()`
  Retrieves whether the cursor is on the last row of this ResultSet object.
- `boolean last()`
  Moves the cursor to the last row in this ResultSet object.
- `boolean next()`
  Moves the cursor down one row from its current position.
- `boolean previous()`
  Moves the cursor to the previous row in this ResultSet object.
- `boolean relative(int rows)`
  Moves the cursor a relative number of rows, either positive or negative.

Getting the data of a row

When the cursor is on a specific row of the result set, it is possible to get the data contained in the different *columns* (*attributs* or *fields*) of that row:

- `int findColumn(String columnName)`
  Maps the given ResultSet column name to its ResultSet column index.
- `boolean getBoolean(int columnIndex)`
  Retrieves the value of the designated column in the current row of this ResultSet object as a boolean in the Java programming language.
- `boolean getBoolean(String columnName)`
  Retrieves the value of the designated column in the current row of this ResultSet object as a boolean in the Java programming language.
- `byte getByte(int columnIndex)`
  Retrieves the value of the designated column in the current row of this ResultSet object as a byte in the Java programming language.
- `byte getByte(String columnName)`
  Retrieves the value of the designated column in the current row of this ResultSet object as a byte in the Java programming language.
- `byte[] getBytes(int columnIndex)`
  Retrieves the value of the designated column in the current row of this ResultSet object as a byte array in the Java programming language.
- `byte[] getBytes(String columnName)`
  Retrieves the value of the designated column in the current row of this ResultSet object as a byte array in the Java programming language.
- `Date getDate(int columnIndex)`

Retrieves the value of the designated column in the current row of this ResultSet object as a java.sql.Date object in the Java programming language.

- `Date getDate(String columnName)`
  Retrieves the value of the designated column in the current row of this ResultSet object as a java.sql.Date object in the Java programming language.
- `double getDouble(int columnIndex)`
  Retrieves the value of the designated column in the current row of this ResultSet object as a double in the Java programming language.
- `double getDouble(String columnName)`
  Retrieves the value of the designated column in the current row of this ResultSet object as a double in the Java programming language.
- `int getInt(int columnIndex)`
  Retrieves the value of the designated column in the current row of this ResultSet object as an int in the Java programming language.
- `int getInt(String columnName)`
  Retrieves the value of the designated column in the current row of this ResultSet object as an int in the Java programming language.
- `ResultSetMetaData getMetaData()`
  Retrieves the number, types and properties of this ResultSet object's columns.
- `Object getObject(int columnIndex)`
  Gets the value of the designated column in the current row of this ResultSet object as an Object in the Java programming language.
- `Object getObject(int i, Map map)`
  Retrieves the value of the designated column in the current row of this ResultSet object as an Object in the Java programming language.

- `Object getObject(String columnName)`
  Gets the value of the designated column in the current row of this ResultSet object as an Object in the Java programming language.
- `int getRow()`
  Retrieves the current row number.
- `Statement getStatement()`
  Retrieves the Statement object that produced this ResultSet object.
- `String getString(int columnIndex)`
  Retrieves the value of the designated column in the current row of this ResultSet object as a String in the Java programming language.
- `String getString(String columnName)`
  Retrieves the value of the designated column in the current row of this ResultSet object as a String in the Java programming language.

The interface $java.sql.ResultSet$ contains many more methods

4. Result management: Example

```
ResultSet rs=st.executeQuery(sql);

while (rs.next()) {
   String sigles=rs.getString(1);
   int nvots=rs.getInt(2);
}
```

or

```
ResultSet rs=st.executeQuery(sql);

while (rs.next()) {
   String sigles=rs.getString("sigles");
   int nvots=rs.getInt("nvots");
}
```

In the second case, we use the names of the table columns (instead of their index) to retrieve the information

4. Result management: Additional Result set features

The queries executed in the framework of a statement created by the operation

`Statement createStatement()`

returns result sets which:

- Are navigable forward only (`TYPE_FORWARD_ONLY`)
- Are only readable (they cannot be updated: `CONCUR_READ_ONLY`)

The *navigation features* of a result set may be modified using another `createStatement(..)` operation:

```
public Statement createStatement(int resultSetType,
int resultSetConcurrency)
```

Creates a Statement object that will generate ResultSet objects with the given type and concurrency.

Parameters:

- `resultSetType`:
  - `ResultSet.TYPE_FORWARD_ONLY`,
  - `ResultSet.TYPE_SCROLL_INSENSITIVE` (Changes made by other users are not shown in the result set)
  - `ResultSet.TYPE_SCROLL_SENSITIVE` (Changes made by other users are shown in the result set)
- `resultSetConcurrency`:
  - `ResultSet.CONCUR_READ_ONLY`
  - `ResultSet.CONCUR_UPDATABLE`

4. Result management: Updating database tables through result sets

- The *ResultSet* interface provides methods `updateXXX(...)` to update the corresponding rows of the database
- These methods may be called only if the result set object is `CONCUR_UPDATABLE`
- The database update is effective only after calling the method:

  `updateRow()`

- Row insertion and deletion are also possible with the methods:

  `insertRow()`

  `deleteRow()`

5. Errors raised during database access

`SQLException`

Example: Managing the results of an election

Example location: jdbc/examples/ex1

This example shows how a database can be accessed and manipulated within a simple web application

We will show the contents of some of its jsp pages:

- `consultaResultat.jsp`

  It shows the results of the different parties taking part in an election

- `actualitzaPartit.jsp`

  It updates the results of a specific party in an election

- `inserirPartit.jsp`

  It inserts a new party in the database

Election results retrieval

File: consultaResultat.jsp

```
<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<%@ page import="java.util.*" %>

<h2> Resultats electorals</h2>
<%
  Statement st = null;
  ResultSet rs = null;
  Connection con = null;
  String DRIVER= "org.postgresql.Driver";
  String URL= "jdbc:postgresql://localhost:5432/Partits";
  try{
    Class c=Class.forName(DRIVER);
    con=DriverManager.getConnection(URL,"josepma","");
    String sql=""
     +" select sigles,nvots "
     +" from partit"
     +" order by nvots desc";
    st= con.createStatement();
    rs=st.executeQuery(sql);
%>
```

(continues.....)

(.....continues)

```
<table border=1 cellpadding=3>
<tr>
 <th>Partit</th> <th>Nombre vots</th>
</tr>
<%
    while (rs.next()) {
      String sigles=rs.getString("sigles");
      int nvots=rs.getInt("nvots");
%>
<tr>
   <td align="right"><%= sigles %></td>
   <td align="right"><%= nvots %></td>
</tr>
<%
    }
    con.close(); con = null;
    st.close();  st=null;
 }
 catch (Exception e) {
   out.println(e.getMessage());
 }
 finally{
  if (st != null) {st.close();}
  if (con != null) {con.close();}
}
%>
</table>
<p><p><p><A href="index.html">INICI</a>
```

Example 2: Updating of the results of a party

File: actualitzaPartit.jsp

```
<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<%@ page import="java.util.*" %>
<html><head>
<title> Actualitzacio del resultat d'un partit </title>
</head><body>
<h2> Actualitzacio del resultat electoral d'un partit</h2>
<%
  String DRIVER= "org.postgresql.Driver";
  String URL=
     "jdbc:postgresql://ingrid.udl.net:5432/Partits";
  Connection con=null;
  Statement st = null;
  try{
    Class.forName(DRIVER);
    con=DriverManager.getConnection(URL,"josepma","");
    String paramsigles=request.getParameter("partit");
    String nvots=request.getParameter("numvots");
    String sql=""
     +" update partit "
     +" set nvots= nvots+" +nvots
     +" where sigles='"+paramsigles+"'";
    Statement st= con.createStatement();
    int j=st.executeUpdate(sql);
%>
```

(continues.....)

(....continues)

```
<p><p>
<b>Actualitzacio feta</b>
<%
    st.close(); st=null;
    con.close(); con = null;
  }
  catch (Exception e)
        {out.println(e.getMessage(););
  }
  finally{
   if (st!=null) {st.close();}
   if (con != null) {con.close();}
  }
%>
</body>
</html>
```

Example 3: Inserting a new party

File: `inserirPartit.jsp`

```
<%@ page session="false" %>
<%@ page import="java.sql.*" %>
<%@ page import="java.util.*" %>
<h2> Insercio nou partit</h2>
<%
  String DRIVER= "org.postgresql.Driver";
  String URL=
      "jdbc:postgresql://ingrid.udl.net:5432/Partits";
  Connection con=null;
  Statement st = null;
  try{
    Class.forName(DRIVER);
    con=DriverManager.getConnection(URL,"josepma","");

    String paramsigles=request.getParameter("sigles");
    String paramnom=request.getParameter("nom");

    String sql=""
     +" insert into partit "
     +" values ('" +paramnom+"','"
     +paramsigles+"',0)";

    Statement st= con.createStatement();
    int j=st.executeUpdate(sql);
%>
```

(continues.....)

(....continues)

```
<p><p>
<b>Insercio feta</b>
<%
    st.close();
    st=null;
}

catch (ClassNotFoundException e) {
    out.println("driver no trobat");
}
finally{
  if (con!=null) con.close();
}
%>
```

And now what.....

In this module we have presented the basics of database access in a web application by using the JDBC API

However, several issues of the presented methodology can be clearly improved:

- **The user that access the database and the database name are hardcoded in the web application**
  - This is clearly inelegant: a change in the database name, location or access user would imply a change in several places of the web application code
  - This problem could be addressed using application-scoped attributes defined in the `web.xml` file
  - However, in module **Design patterns for web applications (Service locator pattern)** we will present a more elegant way of referring a database using the JNDI API

- **Each time a database is used, a new connection to that database is created**
  - This is a bad idea. The process of opening and closing a connection is expensive in terms of time and resources
  - This problem could be addressed by keeping the connection in a session-scoped attribute, which would be requested each time that the database were to be accessed.
    In this way only one connection is needed for a whole session. This is the solution taken in example 2.1 of the module **Tools for view-model separation**
  - However, this solution is not completely satisfactory since it introduces a coupling between the model and the view layers.
  - In the module **Design patterns for web applications** the access to the database will be encapsulated within a specific class (pattern DAO). That class should have access to the session attribute that keeps the connection. This coupling is not a good idea
  - The best solution is the use of `Datasources` and connection pooling. This will be presented in module **Design patterns for web applications**

- **Database abstraction**
  - In the module **Design patterns for web applications** some patterns will be introduced to design web applications as **actual object-oriented well designed applications**
  - One of the considered patterns will be DAO pattern, which will provide abstraction from the actual underlying database and from the specific way to access it
  - If the database is a relational one, the issue of **object-relational** mapping will be brought up
  - Furthermore, some mechanisms for automating this mapping will be presented. One of those mechanisms will be the `JPA` API (Java Persistence API)

Annex: The PostgreSQL DBMS

PostgreSQL basics

- PostgreSQL is a Database Management System (DBMS) which follows the relational model
- It has been developed at the University of California (Berkeley)
- It is an open-source descendant of the POSTGRES project
- `http://www.postgresql.org`

Installation procedure

Get the PostgreSQL package from

`http://www.postgresql.org`

The following slides assume that PostgreSQL has been appropriately installed in the computer that will act as database server

Server runtime environment

1. **Create an account to run the PostgreSQL database server and to store the data (specially, the databases) managed by it**

   This must be done being `root`

   ```
   $ useradd postgres
   ```

   The name of the account may be different from `postgres`. In the following we will assume that we have chosen `postgres`

2. Create a database cluster

This means create a storage area (a directory) which will contain all the databases managed by this database server

```
$ initdb -D /usr/local/pgsql/databases
```

This command will create the directory `/usr/local/pgsql/dat` and will install there all the information necessary to create and manage databases

This should be done being `postgres`

The location can be different. If the `postgres` account does not have permission to create that directory, it should be created as root and then transfer ownership to `postgres`:

```
root$ mkdir /usr/local/pgsql/databases
root$ chown postgres /usr/local/pgsql/databases
root$ su postgres
postgres$ initdb -D /usr/local/pgsql/databases
```

3. Configure the server

We mention just a basic permission policy to access to databases:

File: /usr/local/pgsql/databases/pg_hba.conf

```
# TYPE  DATABASE IP_ADDRESS  MASK            AUTH_TYPE

local    all                                  trust
host     all     127.0.0.1   255.255.255.255  trust
```

With this policy, the database server will allow any local user to connect with any PostgreSQL username, over either UNIX domain sockets or IP

4. Start the database server

```
$ postmaster -i  -D /usr/local/pgsql/databases
    >/usr/local/pgsql/databases/server.log 2>&1 &
```

Or, alternatively:

```
$ pg_ctl start -o "-i" -D /usr/local/pgsql/databases
        -l /usr/local/pgsql/databases/server.log
```

The database server may be shutdown with:

```
$ pg_ctl stop  -D /usr/local/pgsql/databases
```

5. **Create new users**

   The user `postgres` can create new users that are allowed to work with the postgreSQL (e.g., create or use databases...).

   This can be done, from the postgres account:

```
$ createuser josepma
```

   Now, the user `josepma` can interact with postgreSQL with the rights stated in `pg_hba.conf`

6. **Create a database**

   Any authorized user can do that with the command:

```
$ createdb mydb
```

7. **Manage the database**

   PostgreSQL provides a database monitor which may be used to manage the database (create tables, add rows, perform queries....)

```
$ psql mydb
```

   This command enters the monitor environment

Example 1: Management of a database

```
$ psql mydb
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

mydb=# CREATE TABLE teatre (
  nom varchar(60),
  adreca varchar(200),
  poblacio varchar(30),
  telefon varchar(15),
  http varchar(60)
);
CREATE

mydb=# INSERT INTO teatre VALUES
 ('Teatre Principal','Carrer Major-2','Lleida',
  '973.202020','http://principal.com');
INSERT 16578 1

mydb=# INSERT INTO teatre VALUES
 ('Teatre Secundari','Carrer Menor-2','Lleida',
  '973.101010','http://secundari.com');
INSERT 16579 1
```

```
mydb=# SELECT nom FROM teatre
  WHERE poblacio='Lleida'
  ORDER BY nom;

       nom
------------------
 Teatre Principal
 Teatre Secundari
(2 rows)

teatres=#
```

Example 1: Management of a database

The commands may be read from a file

```
$ psql mydb
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

teatres=# \i teatres.sql
CREATE
INSERT 16578 1
INSERT 16579 1
       nom
------------------
 Teatre Principal
 Teatre Secundari
(2 rows)

teatres=#
```

More information

PostgreSQL documentation

In /usuaris/documents/blocTAP/*********

- tutorial.pdf
- instalacio.pdf
- programmerguide.pdf
- userguide.pdf

Annex: The JavaDB DBMS

JavaDB basics

- Originally developed by IBM as Cloudscape
- Code donated to Apache project -¿ Apache Derby
- Distributed by Sun as JavaDB
  - As a separate product
  - As part of J2SE 6 SDK
  - As part of Glassfish V2 application server

JavaDB characteristics

- Implemented entirelly in Java
- With two modes of operation:
  - As an embedded database (in the same JVM as the program)
  - As a network database server

JavaDB installation

- Get distribution from Sun or Apache (javadb-10_3_2_1-linux.bin)
- Execute and a directory javadb is created
- Define DERBY_HOME as this directory
- Add $DERBY_HOME/bin to PATH (administration utilities)

JavaDB drivers

Different JDBC drivers are available depending on the environment you choose for Derby.

- org.apache.derby.jdbc.EmbeddedDriver
  A driver for embedded environments, when Derby runs in the same JVM as the application.

- org.apache.derby.jdbc.ClientDriver
  A driver for the Network Server environment. The Network Server sets up a client/server environment.

JavaDB connection url

jdbc:derby://server[:port]/databaseName[;URLAttributes=value[;...]]

- username=jmgimeno; password=mypassword
- create=true

JavaDB basic use

- Starting the server
  - From the directory we want the database file to be, issue command `startNetworkServer`
- Creating a database
  - Issue command line utility `ij`
  - connect 'jdbc:derby://localhost:1527/mydb;create=true';
- Now you can use SQL