Tema 7
JDBC y pools de conexiones

JDBC. Qué es

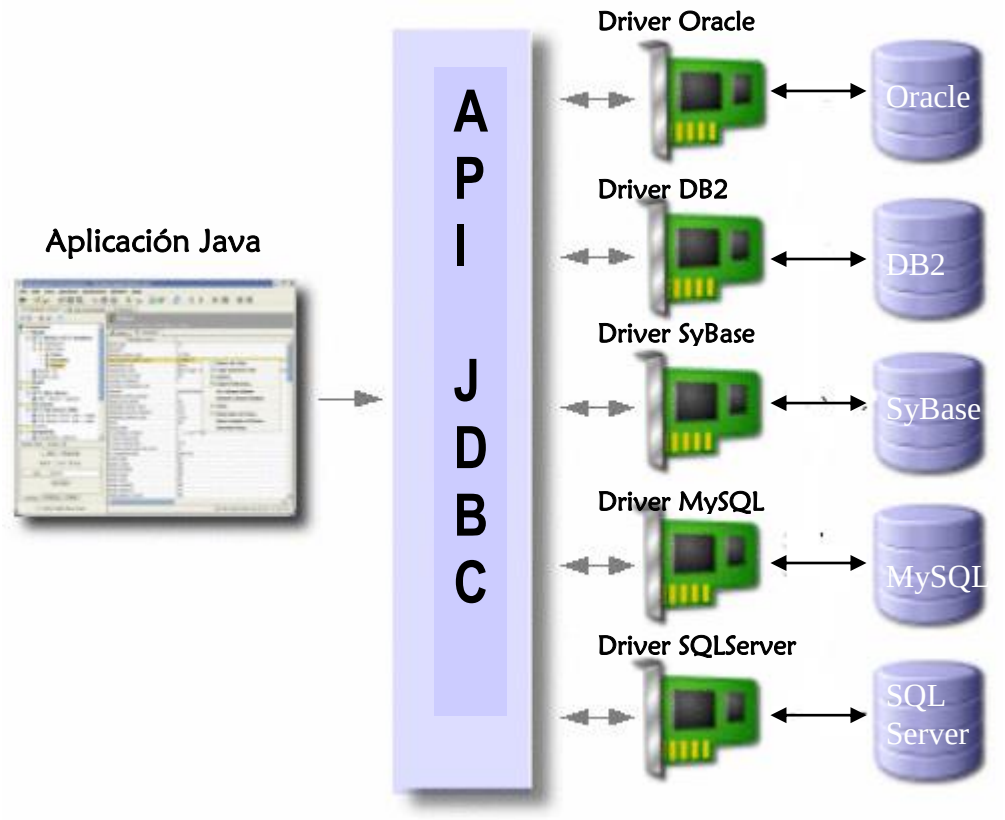
Es una API para la ejecución de operaciones sobre bases de datos relacionales desde Java, independientemente del sistema operativo donde se ejecute o del SGBD accedido.
Permite, pues, que Java pueda acceder a cualquier base de datos relacional (Oracle, MySQL, Sybase, Informix, Access, SQLServer etc.) de forma transparente al tipo de la misma.
JDBC está formado por un conjunto de clases e interfaces programadas en el propio Java.

Drivers JDBC

Para acceder a BBDD en Java es necesario:

1. El entorno de desarrollo de Java (JDK)
2. La API JDBC (incluida en el JDK)
3. **Disponer del driver Java específico para el gestor de BBDD escogido**

Para usar JDBC con un sistema gestor de base de datos en particular, es necesario disponer del driver JDBC apropiado que haga de intermediario entre ésta y JDBC.
Un driver JDBC es el programa que permite a la aplicación Java interactuar con la base de datos. En concreto, es una clase que implementa la interfaz JDBCDriver, proporciona la conexión al SGBD y convierte solicitudes SQL para una BD en particular



¿Qué haremos con JDBC?

- Establecer una conexión con la base de datos.
- Enviar sentencias SQL, de selección y actualización.
- Procesar los resultados.

El siguiente fragmento de código nos muestra un ejemplo básico de estas tres cosas:

```
Connection con = DriverManager.getConnection (
                  " jdbc:mysql://localhost:3306/bd ", "login", password");
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("SELECT a, b, c FROM Tabla1");
while (rs.next()) {
      int x = rs.getInt("a");
      String s = rs.getString("b");
      float f = rs.getFloat("c");
}
rs.close();
st.close();
con.close();
```

Clases principales

Las clases JDBC se incluyen en el paquete **java.sql**, siendo éstas las más utilizadas.

TIPO	Clase JDBC
Implementación	**java.sql.Driver** **java.sql.DriverManager** **java.sql.DriverPropertyInfo**
Conexión a base de datos	**java.sql.Connection**
Sentencias SQL	**java.sql.Statement** **java.sql.PreparedStatement** **java.sql.CallableStatement**
Datos	**java.sql.ResulSet**
Errores	**java.sql.SQLException** **java.sql.SQLWarning**

SQLException

La mayoría de los métodos del API JDBC pueden lanzar la excepción SQLException

- Esta excepción se produce cuando se da algún error en el acceso a la base de datos: error de conexión, sentencias SQL incorrectas,…
- Es un tipo de excepción "catched" (que hay que tratar)

```
try{ .....
}
catch(SQLException e){
    //Interesante utilizar: e.getMessage() con el error concreto generado por el SGBD.
    //Por ejemplo: "ORA-00942: la tabla o vista no existe"
}
```

Clase DriverManager

Registro del driver JDBC

Para poder conectarse a una BD, es preciso antes cargar el driver encargado de esta función.

Class.forName(String driver)

Ejemplos

Class.forName("com.mysql.jdbc.Driver"); ← Registro de driver para MySql

Class.forName("oracle.jdbc.driver.OracleDriver"); ← Registro de driver para Oracle

** tendréis que incluir en el proyecto la librería (.jar) correspondiente*

Establecimiento de una conexión

El método getConnection establece una conexión a la base de datos cuya url se pasa como parámetro, normalmente aportando unas credenciales.

static Connection **getConnection** (String url, String user, String password)

jdbc:mysql://<host>:<puerto>/< base de datos> ← url de connexión a mysql

jdbc:oracle:thin:@<host>:<puerto>:<base de datos> ← url de connexión a Oracle

Ejemplos

```
Connection con=DriverManager.getConnection
                    ("jdbc:mysql://localhost:3306/bdciclos", "Juan","1234");
```

```
Connection con = DriverManager.getConnection
                    ("jdbc:oracle:thin:@localhost:1521:orcl",  "c##scott", "tiger");
```

Clase Connection

Un objeto de la clase Connection representa una sesión conectados con una base de datos.
Se establece, como acabamos de ver, mediante el método getConnection de DriverManager.

Creación de un Statement

La función principal de la clase Connection es crear objetos tipo Statement , mediante el método createStatement

Statement **createStatement**()	Crea un objeto que permitirá enviar consultas SQL a la BD. Puede haber varios Statement simultáneos sobre la misma conexión.

Ejemplo:

```
Statement st=con.createStatement( );
Statement st2=con.createStatement( );
```

void **close**( )	Libera la conexión con la base de datos y los recursos JDBC

Clase Statement

Un Statement representa el canal por el que se envían instrucciones SQL a la base de datos y se reciben los resultados. Vale para instrucciones DML (incluido SELECT) y DDL

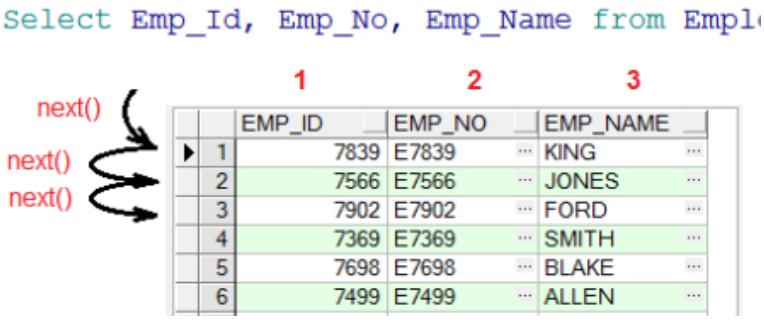
ResultSet **executeQuery**(String sql)	Ejecuta una sentencia SQL SELECT que devuelve un objeto ResultSet (conjuntos de tuplas).
int **executeUpdate**(String sql)	Ejecuta una sentencia INSERT, UPDATE, DELETE, o una sentencia DDL (cualquier sentencia SQL que no devuelva nada) Devuelve 0 o el Nº de filas afectadas por la actualización
void **close**( )	

Ejemplos:

```
Connection con = DriverManager.getConnection
        ("jdbc:oracle:thin:@localhost:1521:orcl",  "c##scott", "tiger");
Statement st=con.createStatement( );
ResultSet productos = st.executeQuery("SELECT * FROM productos WHERE precio<10");
st.executeUpdate("INSERT INTO cafes VALUES ('Colombiano',9)");
st.close();
```

Clase ResultSet

Contiene los datos resultado (filas) de una sentencia SQL Select.
Simula una tabla con las filas y columnas devueltas.
Existen métodos para recuperar secuencialmente las filas de un ResultSet, mediante un cursor que se desplaza automáticamente



---

boolean **next**()

Avanza a la siguiente fila, desde la posición actual.
En un principio, el cursor está situado ANTES de la primera fila del ResultSet, por lo que será necesario una primera llamada a next() para acceder a ella.
Devuelve **false** cuando el cursor se sale de la hoja de resultados

** **IMPORTANTE:** incluso cuando sepamos que el resultset consta de una sola fila/registro, es necesario hacer next() para acceder a ella y recuperar valores*

---

int **getInt** (int columna)
int **getInt** (String nombre_columna)

float **geFloat** (int columna)
float **getFloat** (String nombre_columna)

String **getString** (int columna)
String **getString** (String nombre_columna)

java.sql.Date **getDate**(int columna)
java.sql.Date **getDate**(String nombre_colum)
etc

Métodos que permiten inspeccionar el 'ResultSet':
Devuelven, para la fila actual, el valor de una columna a partir de su nombre o de su posición
(ojo! **comenzando en 1,** no es un índice),

El método getString permite leer cualquier campo de la tabla (aunque sea numérico) como cadena de caracteres.

Hay que tener en cuenta que getDate recupera campos fecha con el tipo java.sql.Date.

Correspondencia de
tipos SQL ←→ método getXXX

SQL Type	Java Method
BIGINT	getLong()
BINARY	getBytes()
BIT	getBoolean()
CHAR	getString()
DATE	getDate()
DECIMAL	getBigDecimal()
DOUBLE	getDouble()
FLOAT	getDouble()
INTEGER	getInt()
LONGVARBINARY	getBytes()
LONGVARCHAR	getString()
NUMERIC	getBigDecimal()
OTHER	getObject()
REAL	getFloat()
SMALLINT	getShort()
TIME	getTime()
TIMESTAMP	getTimestamp()
TINYINT	getByte()
VARBINARY	getBytes()
VARCHAR	getString()

---

void **close**()

Ejemplos

Dadas la tabla de clientes y compras de una BD MySQL **bdcompras**:

clientes : Tabla

id	nombre	pais	edad	vip
1	Joao	Portugal	44	
2	Boris	Croacia	34	
3	Pierre	Canadá	49	
4	Julio	Portugal	24	
5	Pavel	Croacia	2	

Compras : Tabla

idcompra	idcliente	concepto	importe
1	3	AB209	100
2	1	X9222	50
3	3	AC300	150
4	4	D299	300
5	5	D900	45
6	2	D299	300
7	2	W255	20
8	3	L73	120
9	5	AC300	400

Ejemplo

Visualizar nombre y edad de clientes cuya edad es mayor de la media

```
try {
        Class.forName("com.mysql.jdbc.Driver" );
        String url = " jdbc:mysql://localhost:3306/bdcompras ";
        Connection con = DriverManager.getConnection(url, "Juan", "1234");

        String sql="select nombre, edad from clientes where edad >(select avg(edad) from clientes)";
        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery(sql);

        while(rs.next()){
                S.O.P("Nombre:"+ rs.getString(1));
                S.O.P(",edad: "+ rs.getInt("edad"));
        }
        rs.close();
        st.close();
        con.close();

} catch (ClassNotFoundException e) {
        System.out.println("Error driver");

} catch (SQLException e) {
        System.out.println(e.toString());
}
```

Visualiza

```
Nombre:Joao,edad: 44
Nombre:Boris,edad: 34
Nombre:Pierre,edad: 49
```

Configurar como clientes vip (actualizar campo vip ← true) aquellos clientes cuyas compras sumen más de 400 euros

```
Class.forName(”com.mysql.jdbc.Driver” );
String url = " jdbc:mysql://localhost:3306/bdcompras ";
Connection con = DriverManager.getConnection(url, "Juan", "1234");
String sql="UPDATE clientes SET vip = true WHERE id in
        (select idcliente from compras group by idcliente having sum(importe)>400)";
Statement st = con.createStatement( );
int n=st.executeUpdate(sql);
System.out.println(n + " clientes actualizados");
st.close();
con.close();
```

clientes : Tabla

	id	nombre	pais	edad	vip
+	1	Joao	Portugal	44	☐
+	2	Boris	Croacia	34	☐
+	3	Pierre	Canadá	49	☑
+	4	Julio	Portugal	24	☐
+	5	Pavel	Croacia	2	☑

Clase **PreparedStatement** (Derivada de Statement)

Representa objetos que contienen órdenes SQL precompiladas
Se utiliza cuando se quiere ejecutar varias veces la misma consulta con distintos valores. Al estar precompilada, la ejecución será más rápida.

La consulta SQL contenida en un objeto **PreparedStatement** puede tener **uno o más parámetros** cuyo valor no se especifica cuando se crea. Dichos parámetros se establecen mediante interrogantes: **?**
Cada interrogante se identifica posteriormente con un número.
Finalmente, al ejecutar la consulta, rellenaremos esos valores con los métodos **setXXX**()
Los métodos setXXX tienen 2 parámetros:

- El primer parámetro es el número del interrogante, de forma que el primer interrogante que aparece es el 1, el segundo el 2, etc.
- El segundo es el valor a insertar en el lugar del interrogante

En general, hay un método setXXX para cada tipo Java (setInt, setFloat, setString, etc)

Ejemplo 1

```
Connection con = DriverManager.getConnection(……..);

List<String> lista=new ArrayList<String>();
PreparedStatement ps = con.prepareStatement("SELECT * FROM Clientes WHERE pais = ?");

ps.setString(1, txtPais.getText());

ResultSet rs = ps.executeQuery();
while (rs.next()) {
        lista.addItem(rs.getString("Titulo"));
}
rs.close();
ps.close();
con.close();
```

Ejemplo 2

```
Cliente[] clientes={
        new Cliente(10,"Jon",40}, new Cliente(11,"Ana",35}, ….new Cliente(12,"Jim",60)};
……
Connection con = DriverManager.getConnection (……);

String sql ="INSERT INTO clientes (id,nombre, edad) VALUES( ? , ?,  ? )";
PreparedStatement ps = con.prepareStatement (sql); );   → Se precompila la consulta

for (int i=0; i<clientes.length; i++){
        ps.setString (1,clientes[i].getId());
        ps.setString (2, clientes[i].getNombre());
        ps.setInt (3, clientes[i].getEdad());
        int n = ps.executeUpdate( );          → Cada ejecución de la consulta es más rápida,
                                                  al estar precompilada}
}
ps.close();
con.close();
```

Ejemplo 3: Insertar nuevas filas en la tabla de compras anterior (extraídas de un fichero de texto)

Compras : Tabla

idcompra	idcliente	concepto	importe
1	3	AB209	100
2	1	X9222	50
3	3	AC300	150
4	4	D299	300

Fichero de texto **compras.txt**
(En una línea el tabulador separa los datos)

10	4	PK400	27.5
11	2	D299	300
12	1	KL9001	70.09
...	..	.....	.....

```
Connection con = DriverManager.getConnection(……);

PreparedStatement ps = con.prepareStatement("INSERT INTO Compras "+
                        "(idcompra,idcliente, concepto, importe) VALUES ( ?, ?, ?, ?)");

BufferedReader br=new BufferedReader(new FileReader("compras.txt"));

String str=br.readLine();
while(str!=null ){
        String[ ] partes = str.split("\t");

        ps.setString(1, partes[0]);
        ps.setString(2, partes[1]);
        ps.setString(3, partes[2]);
        ps.setFloat(4, Float.parseFloat(partes[3]));

        ps.executeUpdate();
        str=br.readLine( );
}
br.close();
ps.close();
con.close();
```

Utilizar un pool de conexiones en JavaEE

El patrón singleton (una sola conexión en cada momento) es viable para una aplicación de escritorio, pero no para una aplicación Web.

Si prevemos que nuestra aplicación web tendrá cientos o miles de usuarios concurrentes, es necesario preocuparse por el rendimiento del servidor.

- En este sentido, la conexión/desconexión a bases de datos es una tarea que consume muchos recursos (CPU, Memoria)
- Además, si no se cierran las conexiones aun teniendo pocos usuarios es probable que acabemos saturando al servidor de conexiones sin usar.

Por ello, conviene gestionar los accesos a las BBDD mediante un pool de conexiones.

Un **Pool de Conexiones** es un conjunto de conexiones ya establecidas a una base de datos (gestionado por el servidor de aplicaciones), pudiendo ser estas conexiones reutilizadas por las diferentes peticiones.
Tener un Pool de Conexiones permite centralizar el acceso a la base de datos, ya que de dicho acceso se encarga el servidor de aplicaciones.

La idea de usar un Pool de Conexiones es que cada vez que un cliente necesita una conexión a la base de datos, la solicita al Pool, no al SGBD

- Dicha conexión se asigna al cliente hasta que no la necesita más, y en ese momento, en lugar de cerrarla, se devuelve al pool, para poder ser aprovechada por otro cliente.
- Si un cliente necesita una conexión y no hay ninguna disponible, se queda esperando hasta que alguna conexión del pool se libere.

Las conexiones en un Pool de Conexiones se mantienen abiertas desde el principio, por lo que los clientes se ahorran el tiempo de conexión a la Base de Datos.
(Una petición web clásica –sin pool- que acceda a una BD se conecta, consulta o actualiza y se desconecta. Dicha conexión y desconexión consume recursos valiosos para el servidor)

Vamos a ver 2 maneras de utilizar pools de conexiones desde una aplicación Java:

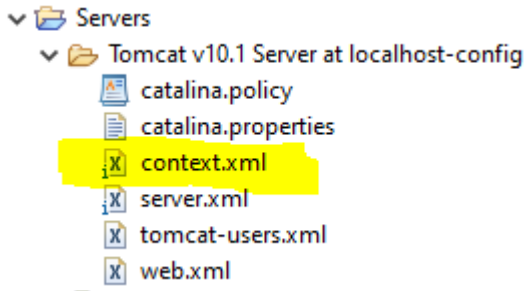
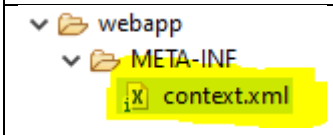
- Configurar pools de conexiones manejados por nuestro servidor de aplicaciones (Tomcat, GlassFish), publicándolos como recurso

- Utilizar un framework universal: Apache Commons DBCP, HikariCP y C3PO son los más conocidos a día de hoy.

Creación de un pool de conexiones en Tomcat (Eclipse)

1. Copiar mysql-connector-xxxx.jar a la carpeta lib de Tomcat (C:\.....\apache-tomcat-10.1.15\lib)

2. Crear un <Resource> de tipo DataSource/fuente de datos: una conexión a una base de datos concreta de un tipo concreto y que utiliza un pool de conexiones.
   Lo haremos en un archivo **context.xml**

	Si modificamos context.xml de Tomcat la fuente de datos será compartido por todas las aplicaciones desplegadas en Tomcat
	Si lo creamos en la carpeta META-INF de nuestra aplicación, será accesible solamente por esta aplicación.

```
<Context>
        <Resource name="jdbc/DSHospital"
                        auth="Container"
                        type="javax.sql.DataSource"
                        driverClassName="com.mysql.cj.jdbc.Driver"
                        url="jdbc:mysql://localhost:3306/bdhospital"
                        username="root"     password=""
                        maxTotal="100" maxIdle="30" maxWaitMillis="10000"
        />
        <Resource name="jdbc/DSSubastas"
                        auth="Container"
                        type="javax.sql.DataSource"
                        driverClassName="org.mariadb.jdbc.Driver"
                        url="jdbc:mariadb://localhost:3306/bdsubastas"
                        username="root"     password=""
                        maxTotal="100" maxIdle="30" maxWaitMillis="10000"
        />
</Context>
```

3. Finalmente, en nuestra clase Java, recuperaremos la fuente de datos así:

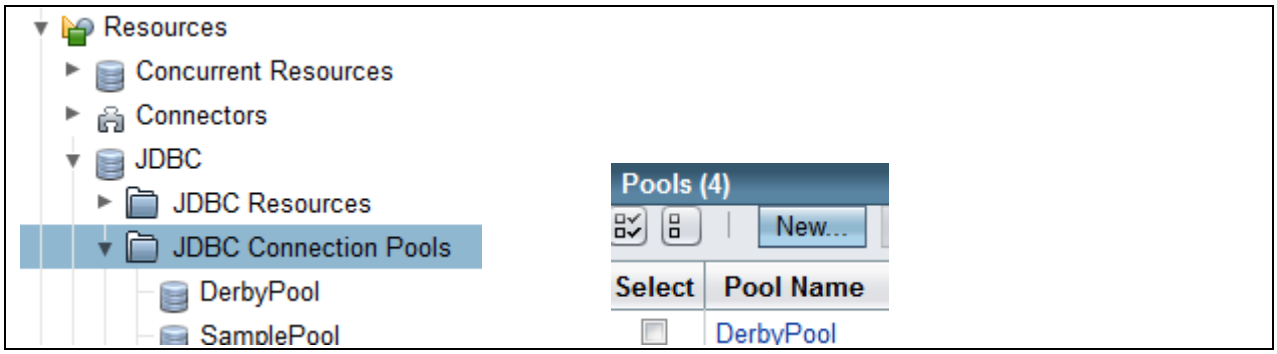
```
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("java:/comp/env/jdbc/DSHospital");
```

Creación de un pool de conexiones en GlassFish

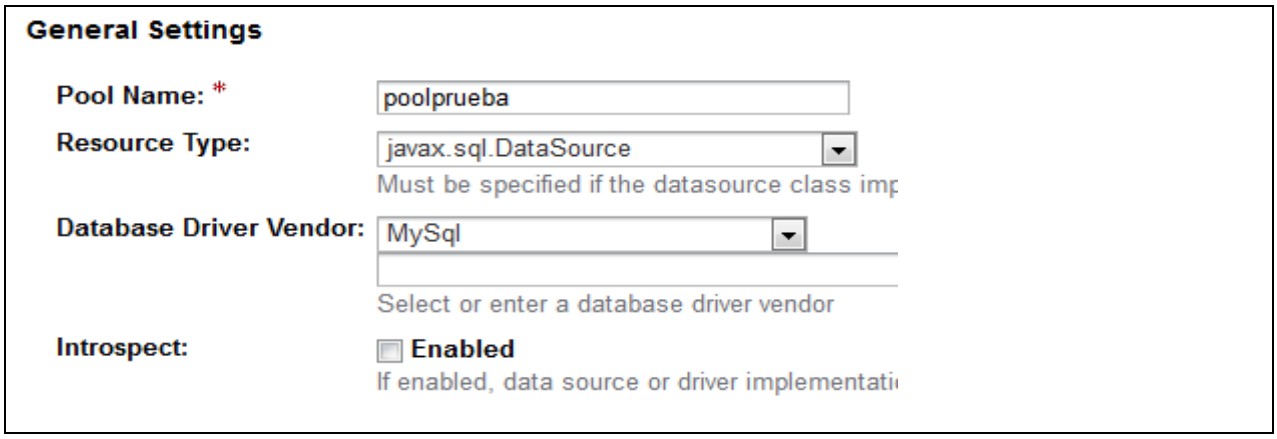
1. Copiar mysql-connector-xxxx.jar a la carpeta lib de GlassFish (C:\....\GlassFish_Server\glassfish\lib)

2. Con GlassFish en marcha, entrar a su consola de Admon http://localhost:4848

3. Entrar a "JDBC Connection Pools" y añadir un nuevo pool en "New"



4. Darle 1 **nombre**, de tipo **javax.sql.DataSource** y Driver: **MySql**



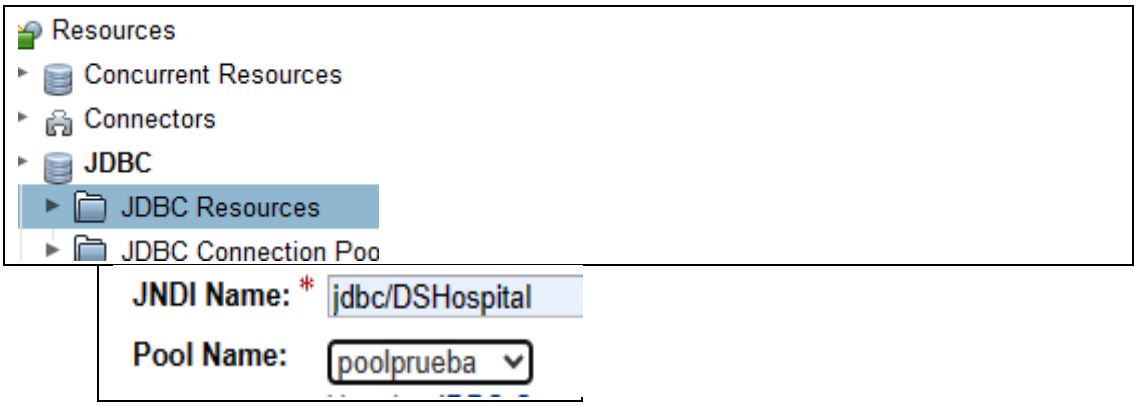
5. Meter propiedades **servidor**, **nombrebd**, **puerto** (3306 para mysql), **usuario**, **password**, y **useSSL**

Name	Value
password	123456
databaseName	bdhospital
serverName	localhost
user	root
portNumber	3306
useSSL	false

Y cambiar → **Datasource Classname:** com.mysql.cj.jdbc.MysqlDataSource

6. Hacer ping para comprobar que se conecta:

7. Crear un recurso JDBC con un nombre JNDI, asociado al pool anteriormente creado



8. En nuestra clase Java, recuperaremos la fuente de datos así:

```
InitialContext ctx = new InitialContext();
DataSource ds = (DataSource) ctx.lookup("jdbc/DSHospital");
```

Connection Pool Hikari (NetBeans)

Hikari provee una implementación sencilla de un pool de conexiones, para cuando no tengamos la opción de configurarla desde el servidor.

Su clase **HikariDataSource** crea un DataSource con pool de conexiones: hay que indicar en un lugar su configuración e invocar **getConnection** para tomar una conexión del pool, por cada consulta. Cuando invoquemos *java.sql.Connection.close()* realmente la conexión no se cerrará sino que se devolverá al pool de conexiones al que pertenece.

1. Añadir dependencia (Dependencies/add Dependency/Query/ …. hikaricp…)

```
<dependency>
        <groupId>com.zaxxer</groupId>
        <artifactId>HikariCP</artifactId>
        <version>2.6.1</version>
 </dependency>
```

2. Copiar librería: mysql-connector-java-8….. a la carpeta lib/ de GlassFish

3. Clase para establecer un DataSource /pool de conexiones estático

```
public class ConnPool {
   private static HikariDataSource dataSource = null;

   static {
                HikariConfig config = new HikariConfig();
                config.setJdbcUrl("jdbc:mysql://localhost:3306/bdhospital?useSSL=false");
                config.setUsername("root");
                config.setPassword("123456");
                config.addDataSourceProperty("minimumIdle", "5");
                config.addDataSourceProperty("maximumPoolSize", "25");
                dataSource = new HikariDataSource(config);
   }

   public static Connection dameConexion() throws SQLException{
        return dataSource.getConnection();
   }

}
```

4. Clase Dao (métodos sql) que toma conexiones del pool

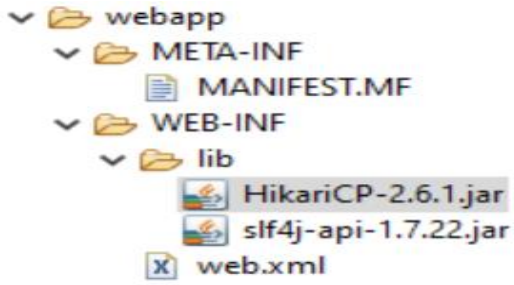
```
public static HashMap<String,String> mapa (){
      HashMap<String,String> map=new HashMap <String,String>();
      try (
          Connection cn=  ConnPool.dameConexion();
          PreparedStatement ps=cn.prepareStatement("select * from tabla");
          ResultSet rs=ps.executeQuery();
          )
       {
          while (rs.next()){
              map.put(rs.getString("campo1"), rs.getString("campo2"));
          }
       }
      catch (SQLException ex) {
       }
       catch (Exception e){
     }
     return map;

}
```

try ( ) { } catch { }
es un ejemplo de try-with-resources.
Todos los recursos (cn, st,…) que se abran en el paréntesis se cerrarán automáticamente (close implícito) en orden inverso al de apertura

Connection Pool Hikari (Eclipse)

1. Añadir librerías:



2. Copiar mysql-connector-xxxx.jar a la carpeta lib de Tomcat: C:\.....\apache-tomcat-10.1.15\lib

3. Clase para establecer un DataSource /pool de conexiones estático

```
public class ConnPool {
   private static HikariDataSource dataSource = null;
   static {
                HikariConfig config = new HikariConfig();
                config.setJdbcUrl("jdbc:mysql://localhost:3306/bdhospital?useSSL=false");
                config.setUsername("root");
                config.setPassword("123456");
                config.addDataSourceProperty("minimumIdle", "5");
                config.addDataSourceProperty("maximumPoolSize", "25");
                dataSource = new HikariDataSource(config);
   }
   public static Connection dameConexion() throws SQLException{
        return dataSource.getConnection();
   }
}
```

4. Clase Dao (métodos sql) que toma conexiones del pool

```
public static HashMap<String,String> mapa (){
      HashMap<String,String> map=new HashMap<String,String>();
      try (
         Connection cn=  ConnPool.dameConexion();
         PreparedStatement ps=cn.prepareStatement("select * from tabla");
         ResultSet rs=ps.executeQuery();
         )
      {
         while (rs.next()){
            map.put(rs.getString("campo1"), rs.getString("campo2"));
         }
      } catch (SQLException ex) {      }      catch (Exception e){      }
      return map;
   }
```