

Práctica 04

Programación Web Dinámica con Interpretación en el Servidor

Fundamentos de Aplicaciones y Servicios Telemáticos

2º Curso Grado en Ingeniería de Tecnologías de Telecomunicación

Departamento de Ingeniería Telemática (DIT)

Universidad de Sevilla



Ignacio Campos Rivera

Francisco José Fernández Jiménez

José Ángel Gómez Argudo

Francisco Javier Muñoz Calle

Juan Antonio Ternero Muñiz

@2018

1	Objetivos y alcance	1
1.1	Introducción.....	1
1.2	Objetivo de la práctica.....	1
1.3	Documentación de apoyo	1
1.4	Preparación del entorno de trabajo	1
1.5	Descripción del entorno de trabajo.....	2
1.5.1	SGBD PostgreSQL.....	2
2	PostgreSQL	3
2.1	Introducción.....	3
2.2	Iniciando una sesión en PostgreSQL.....	3
2.3	Creación de tablas	4
2.4	Llenado de tablas.....	7
2.5	Realización de consultas	8
2.6	Borrado de datos.....	10
2.7	Eliminación de tablas	11
3	JSP: acceso a BBDD	12
3.1	Introducción: JDBC.....	12
3.1.1	Conexión al SGBD	13
3.1.2	Creación de sentencias SQL.....	18
3.1.3	Ejecución de sentencia SQL.....	20
3.1.4	Procesamiento de resultados.....	21
3.1.5	Cierre de la conexión.....	24
3.1.6	Resumen de funciones JDBC	25
3.2	Acceso al SGBD de PostgreSQL	26
3.2.1	Ejemplo 1: administrador de usuarios web	26
3.2.2	Ejemplo 2: listado de mascotas	28
3.2.3	Ejemplo 3: agenda de contactos	32
3.2.4	Ejemplo de uso de javabean y EL	41
4	Ejemplo de integración.....	45
4.1	Introducción.....	45

4.2	Ejercicio	46
5	Anexo no evaluable	50
5.1	Instalación de postgresql	50
5.2	Creación de usuario en SGBD.....	50
5.3	Cliente de PostgreSQL (intérprete de comandos)	50
5.4	Creación de una Base de Datos	51
5.5	Eliminación de BBDD.....	52

1 Objetivos y alcance

1.1 Introducción

Las aplicaciones Web pueden almacenar información de forma persistente en una Base de Datos, y para ello necesitan comunicarse con el SGBD (*Sistema de Gestión de Bases de Datos*) correspondiente. Aunque la mayoría de SGBDs aceptan órdenes en el lenguaje estándar SQL (*Structured Query Language*), la forma de comunicación puede variar. Una forma de conseguir que una misma aplicación Web programada en Java se pueda comunicar fácilmente con distintos SGBDs sin tener que cambiar código es mediante la API (*Application Program Interface*) estándar JDBC (*Java Database Connectivity*). De esta forma JDBC actúa de puente entre el código Java de la aplicación Web y los distintos SGBDs.

1.2 Objetivo de la práctica

Esta práctica tiene como objetivo que el alumno se familiarice con el uso del lenguaje SQL, para realizar consultas, inserción y borrado de datos de tablas de una Base de Datos. En particular, emplearemos el SGBD de PostgreSQL, así como su integración con el lenguaje de programación Java a través de JDBC.

1.3 Documentación de apoyo

Lenguaje SQL

- <http://www.w3schools.com/sql/>

PostgreSQL

- <http://www.postgresql.org/docs/9.1/static/index.html>

JDBC

- <https://docs.oracle.com/javase/7/docs/api/java/sql/package-summary.html>

1.4 Preparación del entorno de trabajo

Para acceder a las URLs mediante el protocolo HTTP (en el navegador debe aparecer `http://` y no `file://`), es necesario que el servidor web se encuentre en ejecución, (Tomcat hace de servidor web, arrancado desde Eclipse, recuerde cómo hacerlo en las prácticas anteriores) así como el Sistema Gestor de Bases de Datos (SGBD o servidor de BBDD) **PostgreSQL** (para PostgreSQL ver apartados 1.5.1 y 2).

La interpretación de **Servlets** y páginas **JSP**, también la hace **Tomcat**, que es contenedor de servlets.

IMPORTANTE: Para la realización de la práctica, utilice el usuario **dit**.

DESCARGA DE LOS FICHEROS DE ESTA PRÁCTICA

En esta práctica se van a usar `FAST_t04-ficheros.tar.gz`, y también los mismos ficheros que en la práctica anterior `FAST_t03-ficheros.tar.gz`, pero es conveniente volverlos a descargar para disponer de la última versión.

Recuerde también borrar el directorio `/home/dit/workspace` y desplegar el fichero `workspace.tar.gz` (tal como se vio en la práctica P01).

1.5 Descripción del entorno de trabajo

Durante el desarrollo de la práctica es necesario que utilice el IDE (entorno de desarrollo integrado) Eclipse. Además, al arrancarlo debe escoger el directorio de trabajo:

```
/home/dit/workspace
```

El directorio donde se han de ubicar los ficheros para la publicación de páginas webs a través de Eclipse se encuentra en:

```
/home/dit/workspace/AppWeb/WebContent
```

Desde Eclipse se puede acceder al contenido de estos desde el explorador de proyectos (Project Explorer) dentro de AppWeb:

```
AppWeb/WebContent/
```

Eclipse copia los ficheros editados en ese directorio en el correspondiente de Tomcat (que es quien responde al navegador):

```
/home/dit/tomcat/webapps/AppWeb/
```

Para acceder al servidor web de Tomcat, la URL a utilizar es:

```
http://localhost:8080/AppWeb
```

1.5.1 SGBD PostgreSQL

Para realizar la práctica, es necesario que exista el usuario o rol “dit” en el SGBD de PostgreSQL, así como la base de datos (BBDD) “dit” (no es lo mismo que el usuario dit de Linux, aunque tenga el mismo nombre y password). Actualmente ya existe ese usuario (rol) y su base de datos. Si desea saber cómo crear otro usuario en el SGBD consulte el anexo.

Para poner en marcha el servidor del SGBD use el comando `start-postgresql`.

Para detener el servidor del SGBD use el comando `stop-postgresql`.

2 PostgreSQL

2.1 Introducción

En la máquina virtual correspondiente a la asignatura debe estar ejecutándose (tal como se ha visto en la tarea anterior) el servidor de PostgreSQL (es el SGBD) aceptando peticiones en el puerto por defecto (5432), y además debe estar creado el usuario `dit` con autorización para la creación de bases de datos.

En esta sección, emplearemos el directorio “**spj**”, ubicado en:

```
/home/dit/workspace/AppWeb/spj.
```

2.2 Iniciando una sesión en PostgreSQL

El comando `psql` pone en marcha un cliente de PostgreSQL que hace de intérprete de comandos y se comunica con el servidor (en este caso el cliente y el servidor están en la misma máquina).

Para acceder al intérprete de comandos de PostgreSQL, con las condiciones dadas en la máquina virtual (servidor escuchando en el puerto por defecto, usuario o rol `dit` y base de datos `dit` existentes), bastará con ejecutar el comando `psql`, sin parámetros. Si desea conocer más detalles del comando y cómo crear otras bases de datos consulte el anexo.

Ya en el intérprete de comandos, puede observar que el prompt es ahora “`dit=>`”, donde `dit` indica la base de datos activa, y la secuencia de símbolos `=>` indica que se está a la espera de la entrada de una sentencia.

El intérprete de comandos admite dos tipos de órdenes (sentencias):

- sentencias SQL: finalizan con un `;` seguido de un retorno de carro. Las sentencias SQL pueden ocupar más de una línea; en este caso, el prompt se va modificando para indicar en qué momento de la edición estamos. Así, por ejemplo, si el prompt termina en “`->`”, indica que no se ha dado por terminado la sentencia (no se ha introducido el punto y coma), mientras que si termina en “`(>`” indica que antes de dar por finalizado la sentencia es necesario cerrar un paréntesis.
- sentencias propias de PostgreSQL: comienzan por el símbolo `\`, y finalizan con un retorno de carro. Son sentencias que permiten obtener información sobre el SGBD. Puede obtener una relación de las sentencias de este grupo mediante la orden `\?` (sólo algunas se utilizarán). Es especialmente interesante la sentencia `\h`, que proporciona ayuda sobre las sentencias SQL. Con la sentencia `\q` se abandona el intérprete de comandos y se vuelve al shell.

TAREAS

Abra un intérprete de comandos `psql`, y ejecute las siguientes sentencias:

```
dit@localhost:~$ start-postgresql
dit@localhost:~$ psql
psql (9.5.5)
Type "help" for help.

dit=> help
dit=> \?
dit=> \h
dit=> \h update
dit=> \du
dit=> \l
dit=> \q
dit@localhost:~$
```

Compruebe que:

- `\?` muestra ayuda de órdenes `psql`
- `\h` muestra ayuda de órdenes SQL
- `\h update` muestra la ayuda para la orden SQL `update`
- `\du` muestra los usuarios (roles) definidos en el SGBD (`dit` entre ellos)
- `\l` muestra las bases de datos (`dit` entre ellas)
- `\q` sale del intérprete de comandos

El intérprete de comandos `psql` admite distintas opciones (puede consultarlas con `psql --help`). Si no se especifica ninguna opción se usa como nombre de usuario el mismo de linux y como base de datos la del mismo nombre que el usuario. Por ejemplo, si se quiere usar el usuario `dit2` y la base de datos `fast2` (si existieran):

```
dit@localhost:~$ psql -U dit2 -d fast2
```

Nota: aunque normalmente se utilicen mayúsculas para las sentencias SQL y los nombres de las tablas, no es necesario y se puede utilizar minúsculas indistintamente (también en los nombres de los campos).

2.3 Creación de tablas

La primera acción a realizar en una base de datos consiste en la creación de las tablas. Para la realización de esta práctica se definirán cuatro tablas (S, P, J, SPJ). En este apartado, trabajaremos con los ficheros contenidos en la carpeta “spj”.

Para la creación de las tablas, se hará uso de una de las sentencias propias de PostgreSQL, `\i`, que permite leer las sentencias a ejecutar de un fichero de texto. El fichero a leer debe estar accesible, y una de las formas es que esté en el mismo directorio desde el que se ejecutó el comando `psql`.

TAREAS

Cambie el directorio de trabajo (se muestra en la captura). Abra un intérprete de comandos sobre la base de datos de dit (psql), y ejecute la siguiente sentencia (para ahorrarse la escritura vea la siguiente nota):

```
dit@localhost:~$ cd /home/dit/workspace/AppWeb/spj
dit@localhost:~/workspace/AppWeb/spj$ cat s.sql
-- Table: S
CREATE TABLE S (
  ids INTEGER,
  noms CHAR(20) NOT NULL,
  estado CHAR(25),
  ciudad CHAR(25) NOT NULL,
  PRIMARY KEY (ids)
);
dit@localhost:~/workspace/AppWeb/spj$ psql
dit=> \i s.sql
```

Nota: lo anterior es equivalente a (dentro del intérprete de comandos psql):

```
dit@localhost:~/workspace/AppWeb/spj$ psql
dit=> CREATE TABLE S (
dit(>   ids INTEGER,
dit(>   noms CHAR(20) NOT NULL,
dit(>   estado CHAR(25),
dit(>   ciudad CHAR(25) NOT NULL,
dit(>   PRIMARY KEY (ids)
dit(> )
dit-> ;
```

Ejecute lo siguiente (siempre que aparezca `dit=>`, significa dentro del intérprete de comandos psql) para ver la estructura de la tabla creada:

```
dit=> \d S
```

Con esta sentencia se ha creado una tabla de nombre **s** con cuatro campos:

- Un campo `ids`, que es de tipo entero; además es la clave primaria de la tabla (ver “PRIMARY KEY” en la última línea), lo que supone que este campo no puede estar vacío, y además no puede repetirse en los registros de la tabla, ya que ese campo identifica a un registro (una fila dentro de la tabla)
- Un campo `noms`, de hasta 20 caracteres. Tampoco puede estar vacío.
- Un campo `estado`. De hasta 25 caracteres.
- Un campo `ciudad`, de hasta 25 caracteres, que tampoco puede estar vacío.

Para la creación de las tablas restantes, se hará uso de otros ficheros de texto y de la sentencia de postgresSQL `\i`.

TAREAS

a) Cree una nueva tabla, de nombre **P**, con los cinco campos siguientes:

- Un campo entero, **idp**, que será la clave primaria de la tabla.
- Un campo de hasta 20 caracteres, no vacío, de nombre **nomp**.
- Un campo de hasta 10 caracteres, no vacío, de nombre **color**.
- Un campo entero, no vacío, de nombre **peso**.
- Un campo de hasta 25 caracteres, no vacío, de nombre **ciudad**.

Para ello ejecute la siguiente sentencia:

```
dit=> \i p.sql
```

b) Para crear las tablas **J** y **SPJ**, compruebe que también se pueden leer las sentencias sql de un fichero, sin tener que entrar en el intérprete de comandos **psql** en modo interactivo:

```
dit@localhost:~/workspace/AppWeb/spj$ psql < j.sql
dit@localhost:~/workspace/AppWeb/spj$ psql < spj.sql
```

c) Observe los ficheros **p.sql**, **j.sql** y **spj.sql**, y podrá comprobar que no son más que sentencias SQL dentro de un fichero.

d) Compruebe que se han creado las tablas con la orden **\d**

```
dit=> \d
```

IMPORTANTE:

En la sentencia de creación de la tabla **SPJ** puede observar dos especificaciones que no aparecen en las anteriores:

- **REFERENCES S (ids)** (también con **P (idp)** y con **J (idj)**): indica que el valor de este campo sólo puede ser uno que ya aparezca en el campo **ids** (o **idp**, o **idj**) de la tabla **S** (o **P** o **J**). Esto es así porque **S** contiene datos de suministradores, **P** contiene datos de piezas, **J** contiene datos de proyectos y **SPJ** contiene en cada registro o fila la cantidad de una pieza (identificada por **idp**) suministrada a un proyecto (identificado por **idj**) por un suministrador (identificado por **ids**). Así los valores que aparecen en **SPJ** tienen que corresponderse con suministradores, piezas y proyectos que existan (identificados respectivamente por **ids**, **idp** e **idj**)
- La especificación **PRIMARY KEY** (clave primaria) hace referencia a más de un campo (tres en concreto). Con esto se indica que lo que no se pueden repetir en los diferentes registros o filas son las combinaciones de los valores de los tres campos en conjunto, no de cada uno de ellos individualmente. Es decir, en la tabla **SPJ** lo que no se puede repetir es la combinación de identificador de suministrador, pieza y proyecto (**ids**, **idp**, **idj**). En cambio dentro de la tabla **S** lo que no se puede repetir es el identificador de suministrador **ids**, ya que **ids** es la clave primaria, y de forma similar ocurre con **idp** en **P** e **idj** en **J**.

2.4 Llenado de tablas

El último paso para tener disponible la base de datos consiste en añadir registros a las tablas. Para ello se utiliza la sentencia SQL `“INSERT”`. Como regla general, para insertar valores es necesario indicar en qué tabla hay que añadir los valores, qué campos hay que rellenar y con qué valores.

Por ejemplo, para introducir un registro en la tabla `S`, con un valor `"1"` en el campo `ids`, `“Suministros 01”` en el campo `noms`, y `“Madrid”` en el campo `ciudad`, habría que ejecutar la siguiente sentencia:

```
dit=> INSERT INTO S
dit->   (ids, noms, ciudad)
dit->   VALUES
dit->   (1, 'Suministros 01', 'Madrid');
```

Puede observarse que no es necesario especificar todos los campos de la tabla a la hora de insertar un nuevo registro; los campos no indicados se inicializan a `NULL`. Lógicamente sólo podrán estar ausentes de la inserción los campos de la tabla que no estén marcados como `NOT NULL`. Por ejemplo, no será posible insertar un nuevo registro sin asignar valor al campo `noms`.

TAREAS

- Ejecute la sentencia anterior (use comillas simples para los valores).
- Inserte ahora un registro en la tabla `S` con valores `'11'` para `ids`, `'Italia'` para `estado` y `'Florenia'` para `ciudad`, y observe el error, ya que no se le está dando ningún valor a `noms`.

Alternativamente, cuando el volumen de los datos a introducir es muy elevado, puede utilizarse la sentencia `\copy`, que permite rellenar una tabla con los valores previamente escritos en un fichero.

TAREAS

Para rellenar la tabla `S` con los datos almacenados en el fichero `s.txt`, ejecute la sentencia:

```
dit=> \copy S FROM 's.txt'
```

Si observa el fichero `s.txt` puede comprobar cómo cada línea del fichero se corresponde con un registro de la tabla, donde los campos están por orden y separados por un tabulador. Además puede observar que el valor `NULL` en el fichero se representa mediante la secuencia `\N`. Vea también que por tratarse de una sentencia propia de PostgreSQL no hay que finalizar la sentencia con punto y coma.

TAREAS

- Repita la sentencia anterior para rellenar las tablas `P` (fichero `p.txt`), `J` (fichero `j.txt`) y `SPJ` (fichero `spj.txt`).
- Inserte ahora un registro en la tabla `J` con valores `'1'` para `idj`, `'Edificio 21'` para `nomj` y `'Badajoz'` para `ciudad`, y observe el error, ya que `idj` es clave primaria (recuerde el uso de comillas simples para los valores).

2.5 Realización de consultas

Una vez completada la inicialización de la base de datos, pueden realizarse consultas sobre ellas.

TAREAS

Para comenzar, liste el contenido de todas las tablas de la base de datos, una a una. Para hacerlo sobre la tabla “S”, tendrá que ejecutar la sentencia:

```
dit=> SELECT * FROM S;
```

```
dit=> SELECT * FROM S;
ids |          noms          | estado |          ciudad
-----+-----+-----+-----
  1 | Suministros 01         |         | Madrid
  2 | Suministros 02         |         | Londres
  3 | Suministros 03         |         | Sevilla
  4 | Suministros 04         |         | Londres
  5 | Suministros 05         |         | París
  6 | Suministros 06         |         | New York
  7 | Suministros 07         |         | Atenas
  8 | Suministros 08         |         | Sevilla
  9 | Suministros 09         |         | Roma
 10 | Suministros 10         |         | Múnich
(10 rows)
```

TAREAS

Repita la sentencia para las otras tres tablas de la base de datos (J, P y SPJ).

```
dit=> SELECT * FROM J;
dit=> SELECT * FROM P;
dit=> SELECT * FROM SPJ;
```

La función más básica de una consulta a una base de datos es seleccionar registros de una tabla que cumplan con algún criterio. Para ello se utiliza la cláusula `WHERE` de la sentencia `SELECT`. Mediante esa sentencia se pueden expresar condiciones que deben cumplir los registros seleccionados.

Por ejemplo, para conocer los suministradores (tabla S) de la ciudad de Sevilla, se escribirá:

```
dit=> SELECT * FROM S WHERE ciudad = 'Sevilla';
ids |          noms          | estado |          ciudad
-----+-----+-----+-----
  3 | Suministros 03         |         | Sevilla
  8 | Suministros 08         |         | Sevilla
(2 rows)
```

Las consultas también permiten seleccionar los campos de la tabla que se desean obtener. Por ejemplo, si lo que se quiere es el nombre y el peso de las piezas (tabla P) con peso superior a 50, habrá que ejecutar la sentencia:

```
dit=> SELECT nomb, peso FROM P WHERE peso > 50;

nomb          | peso
```

```

-----+-----
Pieza 01      | 100
Pieza 04      | 75
Pieza 07      | 80
(3 rows)

```

Las condiciones pueden ser compuestas; para conocer el color de las piezas en Sevilla con peso mayor que 30, tendríamos:

```

dit=> SELECT color FROM P
dit-> WHERE peso > 30 AND ciudad = 'Sevilla';
      color
-----
rosa
(1 rows)

```

Las consultas pueden afectar a varias tablas, pudiendo establecer condiciones entre campos de distintas tablas. Por ejemplo, para conocer el color de las piezas que forman parte del proyecto 3, deberá ejecutar la sentencia:

```

dit=> SELECT color FROM P, SPJ
dit-> WHERE SPJ.idj = 3 AND SPJ.idp = P.idp;
      color
-----
azul
rojo
rosa
(3 rows)

```

En la anterior consulta se han usado dos tablas en la cláusula FROM porque hace falta relacionar el color de la pieza (que está en la tabla P) con el identificador del proyecto en el que se suministra una pieza (que está en la tabla SPJ). El incluir dos tablas produce el producto cartesiano (todas las combinaciones posibles) y sólo tienen sentido aquellas filas en las que el identificador de la pieza que proviene de SPJ sea igual al identificador que proviene de P.

Para entender mejor el producto cartesiano, pruebe a ejecutar:

```
dit=> SELECT * FROM P, SPJ;
```

En las condiciones de la cláusula WHERE también se puede usar el operador IN, donde el operador de la derecha es una subconsulta (una consulta entre paréntesis). Por ejemplo, para obtener los nombres de los proyectos a los que suministra el suministrador "1" se puede hacer de varias formas (nótese que cuando se utiliza el producto cartesiano hay que usar DISTINCT para eliminar los resultados repetidos, pero si se utiliza IN no hace falta). Con la cláusula ORDER BY se ordena el resultado:

```

dit@localhost:~/workspace/AppWeb/spj$ cat con_nomj_sum_1.sql

-- Obtener los nombres de los proyectos en los que el suministrador es "1".

SELECT DISTINCT nomj FROM j,spj WHERE spj.idj = j.idj AND spj.ids = '1' ORDER BY nomj;

SELECT nomj FROM j WHERE idj IN (SELECT idj FROM spj WHERE ids = '1') ORDER BY nomj;
dit@localhost:~/workspace/AppWeb/spj$ psql < con_nomj_sum_1.sql

```

NOTA: Puede usar el fichero `recarga_spj.sql` para asegurarse el estado de la base de datos (la sentencia `DROP` se ve más adelante):

```
dit@localhost:~/workspace/AppWeb/spj$ psql < recarga_spj.sql
```

TAREAS

Realice las siguientes consultas:

- a) Obtener todos los datos de los proyectos.
- b) Obtener el nombre y la ciudad de todos los suministradores.
- c) Obtener el nombre y el color de todas las piezas cuyo color es 'rojo'.
- d) Obtener el nombre de los suministradores que suministran la pieza '2'.
- e) Obtener el nombre de los suministradores que no suministran la pieza '2' (use `NOT IN`).

NOTA: Recuerde ejecutar `\q` para salir del intérprete `psql`.

2.6 Borrado de datos

La sintaxis básica para el borrado de datos de una tabla es la siguiente:

```
DELETE FROM tabla [ WHERE condicion ];
```

Si no se incluye la cláusula `WHERE` se borran todos los datos. Cuando se borran los datos de una tabla, la tabla puede quedar vacía, pero sigue existiendo, y se puede volver a introducir datos.

IMPORTANTE: no elimine datos, ni tablas, ni BBDD, pues se emplearán en el desarrollo de la práctica. Al finalizar la misma, puede probar las sentencias que considere oportunas. Basta realizar los pasos de los apartados anteriores para crear de nuevo las estructuras de datos y llenarlas. También puede usar el fichero `recarga_spj.sql`

TAREAS

Ejemplo con la tabla “P”:

```
DELETE FROM P WHERE peso > 30 AND ciudad = 'Sevilla';
```

Ejecute la sentencia anterior y compruebe que se produce un error y no se borra (ya que esa fila está referenciada desde SPJ).

Muestre las filas que se quieren borrar:

```
SELECT * FROM P WHERE peso > 30 AND ciudad = 'Sevilla';
```

Muestre el identificador de las filas que se quieren borrar:

```
SELECT idp FROM P WHERE peso > 30 and ciudad = 'Sevilla';
```

Muestre las filas de SPJ que hacen referencia a las filas de P que se quieren borrar:

```
SELECT * FROM SPJ WHERE idp IN (SELECT idp FROM P WHERE peso > 30 AND ciudad = 'Sevilla');
```

Borre las filas de SPJ que hacen referencia a las filas de P que se quieren borrar:

```
DELETE FROM SPJ WHERE idp IN (SELECT idp FROM P WHERE peso > 30 AND ciudad = 'Sevilla');
```

Borre las filas que se quieren borrar:

```
DELETE FROM P WHERE peso > 30 AND ciudad = 'Sevilla';
```

Ahora ya no se produce error.

Si quiere volver al estado inicial:

```
\i recarga_spj.sql
```

2.7 Eliminación de tablas

La sintaxis para la eliminación o borrado de una tabla es la siguiente:

```
DROP TABLE nombre_tabla
```

Cuando se elimina una tabla, ya deja de existir, y si se quiere usar hay que volver a crearla para poder introducir datos.

3 JSP: acceso a BBDD

3.1 Introducción: JDBC

Con JSP, es posible acceder a BBDD mediante JDBC (*Java Database Connectivity*, API abstracta) desde el código Java incrustado en las páginas web. Cada SGBD al que se quiera acceder mediante **JDBC** debe contar con un controlador (driver).

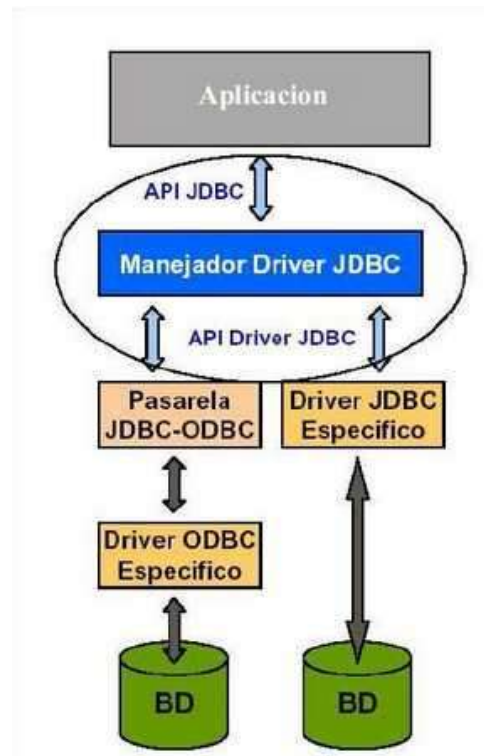
JDBC es una API de Java que permite ejecutar sentencias SQL, y consta de un conjunto de clases e interfaces escritos en lenguaje de programación Java.

Empleando JDBC, es fácil ejecutar sentencias SQL accediendo a cualquier base de datos relacional. Por esta razón, no es necesario escribir un programa para acceder a una base de datos tipo Access, otro programa para acceder a una base de datos tipo Oracle y así para cada tipo de base de datos. Se puede escribir un solo programa usando la API JDBC y el programa será capaz de enviar sentencias SQL a la base de datos apropiada. Además con una aplicación escrita en Java, se puede ejecutar en diferentes plataformas.

El funcionamiento práctico de JDBC comprende las siguientes fases:

- 1) Establecimiento de conexión con el SGBD.
- 2) Creación de sentencia SQL.
- 3) Ejecución de sentencia SQL.
- 4) Procesamiento de resultados.
- 5) Cierre de la conexión.

Un ejemplo de un fragmento de código para acceder desde un cliente en java a un SGBD postgresQL podría ser (sería necesario importar la librería java.sql con `import java.sql.*`):



```
_____ .java

try {
    Class.forName("org.postgresql.Driver");

    String url = "jdbc:postgresql://localhost:5432/dit";
    String user = "dit";
    String pass = "dit";
    Connection conn = DriverManager.getConnection(url, user, pass);

    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery("SELECT * FROM usuarios");
    while(rs.next())
    {
        String name = rs.getString("name");
        String password = rs.getString("password");
        System.out.println("name=" + name + " password=" + password);
    }

    rs.close();
    st.close();
    conn.close();

} catch (SQLException e) {
    out.println("Excepción SQL Exception: " + e.getMessage());
    e.printStackTrace();
}
```

En el ejemplo anterior se observan las 5 fases. A continuación, se verán cada una de estas fases en detalle.

3.1.1 Conexión al SGBD

3.1.1.1 Conexión mediante DriverManager

La conexión implica el uso de dos objetos:

- 1) DriverManager
- 2) Connection

- **DriverManager**

La clase `DriverManager` es la capa gestora de JDBC, trabajando entre la aplicación de usuario y el controlador (driver). Se encarga de seguir el rastro de los controladores que están disponibles y establecer la conexión entre la base de datos y el controlador apropiado. La clase `DriverManager` mantiene una lista de clases `Driver` que se han registrado llamando al método `DriverManager.registerDriver`. Un usuario normalmente no llamará al método `DriverManager.registerDriver` directamente, sino que será llamado automáticamente por el controlador (driver) cuando éste se carga.

El usuario lo que hace es forzar que se cargue el driver, lo cual puede hacerse de dos formas, aunque la recomendada es llamando al método `Class.forName()`. Esto carga la clase driver explícitamente. Para PostgreSQL, el parámetro debe ser:

```
"org.postgresql.Driver"
```

Nota: desde la versión 4 de JDBC no es necesario usar el método `Class.forName()` en cada conexión (sólo es necesario una vez al comienzo de la aplicación), aunque se puede usar en cada conexión. Tampoco es necesario el método `Class.forName().newInstance()`, aunque se suele encontrar por compatibilidad hacia atrás. El protocolo de comunicación con el SGBD está indicado también en la url usada en el método `DriverManager.getConnection(url, user, pass)`.

- **Connection**

Un objeto `Connection` representa una conexión a una base de datos. Una sesión con una conexión incluye las sentencias SQL que son ejecutadas, y los resultados que son devueltos a través de dicha conexión. Una misma aplicación puede tener una o más conexiones con una sola base de datos o puede tener conexiones con varias bases de datos diferentes.

La forma estándar de establecer una conexión con una base de datos es llamando al método `DriverManager.getConnection`. Este método toma como parámetro una cadena de caracteres que contiene una URL de conexión. La clase `DriverManager` trata de localizar el driver que pueda conectar con la base de datos representada por esa URL.

Ejemplo:

```
import java.sql.*;

public static void conexion ( ) throws Exception {

try {
    // Carga del driver
    Class.forName("org.postgresql.Driver");

    // Definición de la cadena de conexión
    String url = "jdbc:postgresql://localhost:5432/dit";
    String user = "dit";
    String pass = "dit";

    // Creación del objeto Connection a través de DriverManager
    Connection conn = DriverManager.getConnection(url, user, pass);

    System.out.println("La conexion establecida es: "+ conn);

} catch(ClassNotFoundException cnfe) {

    System.err.println(cnfe);

} catch(SQLException sqle) {

    System.err.println(sqle);

};
};
```

TAREAS

- a) Consulte la documentación de JavaSE y busque los paquetes `java.sql` y `javax.sql`:
<http://docs.oracle.com/javase/7/docs/api/>
- b) Consulte la documentación anterior y busque la especificación del método `getConnection` de la clase `DriverManager`. Compruebe que es estático (calificador `static` y por lo tanto se usa directamente con el nombre de la clase), que el nombre del método está sobrecargado y que uno de los métodos toma 3 parámetros.
- c) Consulte la documentación de PostgreSQL y compruebe el significado de los 3 parámetros del método anterior:
<https://jdbc.postgresql.org/documentation/91/connect.html>

Compruebe que si el servidor estuviera escuchando en el puerto 5000 y el nombre de la base de datos fuera `ditprueba`, el valor de la url sería:

```
"jdbc:postgresql://localhost:5000/ditprueba"
```

- d) Compruebe el funcionamiento de:
<http://localhost:8080/AppWeb/jspdb/buscaPiezas1.jsp>

Asegúrese antes de que el SGBD está en servicio y la BBDD tienen datos:

```
dit@localhost:~/workspace/AppWeb/spj$ start-postgresql
dit@localhost:~/workspace/AppWeb/spj$ psql < recarga_spj.sql
```

- e) Observe el código del fichero `jsp` y compruebe cómo se accede al SGBD. Haga esa misma petición con el comando `psql`:

```
dit@localhost:~/workspace/AppWeb/spj$ psql
dit=> SELECT * FROM ... WHERE ... ;
```

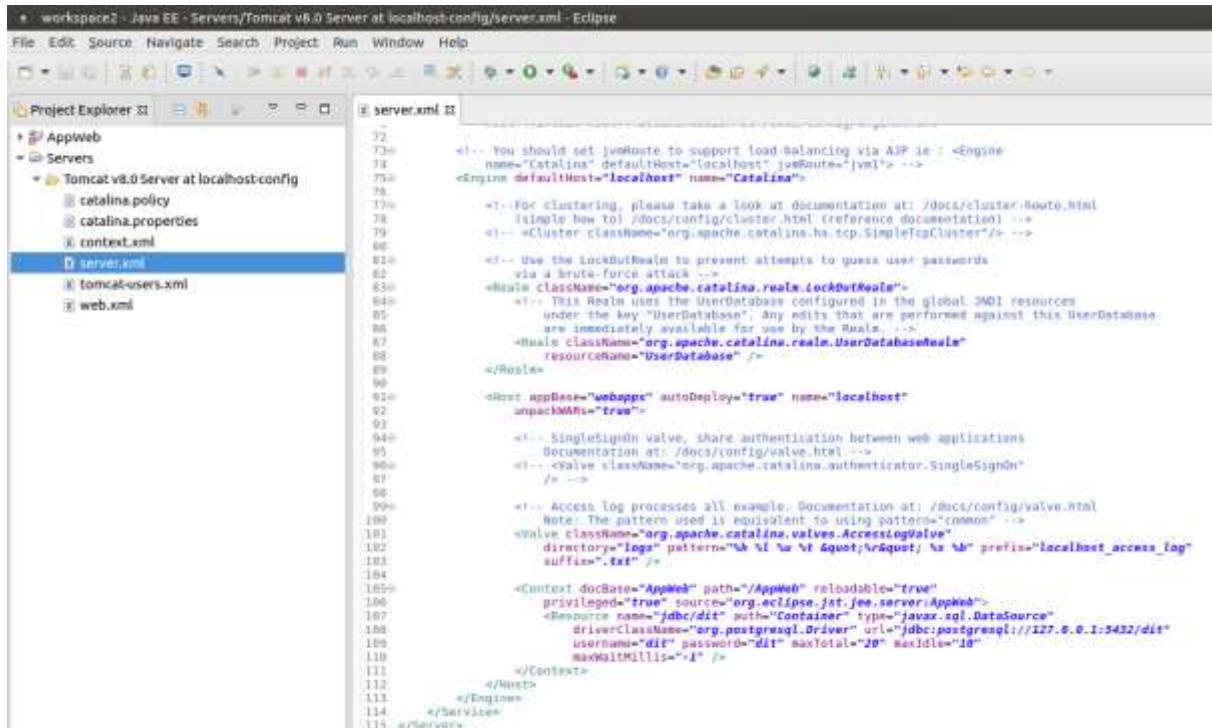
- f) ¿Cuál es la url asociada al formulario (etiqueta `action`)?
- g) ¿Cómo funciona el fichero `jsp` para que la primera vez que se accede aparezca sólo el formulario, pero no la cabecera de los datos?
- h) Si tuviera que cambiar el usuario o password para acceder al SGBD ¿qué fichero habría que modificar?

3.1.1.2 Conexión mediante referencia a DataSource

Otra forma más eficiente de acceder al SGBD es, tal como se ha visto en teoría, mediante referencia a `DataSource`. Puede consultar información relativa en:

<http://localhost:8080/docs/jndi-datasource-examples-howto.html>

Para ello, comprobar el contenido del fichero `server.xml` de configuración de Tomcat:



En concreto, comprobar que contiene el recurso de nombre jdbc/dit (con la url que identifica el driver, la dirección IP, el puerto y la base de datos dit, el username que identifica al usuario dit y el password que es dit) en la aplicación AppWeb:

/home/dit/workspace/Servers/Tomcat v8.0 Server at localhost-config/server.xml

```
...
<Context docBase="AppWeb"
  path="/AppWeb"
  reloadable="true"
  privileged="true"
  source="org.eclipse.jst.jee.server:AppWeb">
  <Resource name="jdbc/dit"
    auth="Container"
    type="javax.sql.DataSource"
    driverClassName="org.postgresql.Driver"
    url="jdbc:postgresql://127.0.0.1:5432/dit"
    username="dit"
    password="dit"
    maxTotal="20"
    maxIdle="10"
    maxWaitMillis="-1" />
</Context>
...
```

Y también, comprobar que el fichero web.xml de configuración de la aplicación AppWeb contiene (la descripción es opcional) la referencia al recurso de nombre jdbc/dit:

home/dit/workspace/AppWeb/WebContent/WEB-INF/web.xml

```
...
<resource-ref>
  <description>    postgresQL Datasource </description>
  <res-ref-name>    jdbc/dit </res-ref-name>
  <res-type>        javax.sql.DataSource </res-type>
</resource-ref>
```

```
<res-auth>          Container          </res-auth>
</resource-ref>
...
```

Una vez que están bien configurados los ficheros server.xml y web.xml, desde la aplicación se puede acceder al SGBD mediante (observe que son necesarios nuevos import):

```
import java.sql.*;

import javax.sql.DataSource
import javax.naming.InitialContext
import javax.naming.Context

public static void conexion ( ) throws Exception {

try {
    InitialContext ctx = new InitialContext();
    DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/dit");
    Connection conn = ds.getConnection();

    System.out.println("La conexion establecida es: "+ conn);

} catch(ClassNotFoundException cnfe) {

    System.err.println(cnfe);

} catch(SQLException sqle) {

    System.err.println(sqle);

};
};
```

TAREAS

- Consulte la documentación y busque la especificación de InitialContext y DataSource:
<https://docs.oracle.com/javase/7/docs/api/javax/naming/InitialContext.html>
<https://docs.oracle.com/javase/7/docs/api/javax/sql/DataSource.html>
- Consulte la documentación anterior y busque la especificación del método lookup ¿a qué interfaz pertenece?
- Consulte la documentación anterior y busque la especificación del método getConnection ¿qué devuelve?
- Compruebe el funcionamiento de:
<http://localhost:8080/AppWeb/jspdb/buscaPiezas1bis.jsp>

Asegúrese antes de que el SGBD está en servicio y la BBDD tienen datos:

```
dit@localhost:~/workspace/AppWeb/spj$ start-postgresql
dit@localhost:~/workspace/AppWeb/spj$ psql < recarga_spj.sql
```

- Observe el código del fichero jsp y compruebe cómo se accede al SGBD. Observe la diferencia con buscaPiezas1.jsp
- Si tuviera que cambiar el usuario o password para acceder al SGBD ¿qué fichero habría que modificar?

3.1.2 Creación de sentencias SQL

Para crear una sentencia se pueden usar dos tipos de objetos:

- Statement
- PreparedStatement

3.1.2.1 Uso de Statement

El objeto `Statement` se utiliza para enviar sentencias SQL a una base de datos. Una vez que se ha establecido una conexión con una base de datos particular, esa conexión puede ser usada para enviar sentencias SQL. Antes de crear el objeto se debe tener una conexión establecida con el SGBD.

Tal como se ha visto en el apartado anterior, puede ser usando `DriverManager`:

```
Connection conn = DriverManager.getConnection(url, user, pass);
```

O de la forma más eficiente, usando `Datasource`:

```
Connection conn = ds.getConnection();
```

Una vez que se tiene la conexión, un objeto `Statement` se crea con el método `createStatement` de `Connection`. Ejemplo:

```
Statement sentencia = conn.createStatement();
```

Por ejemplo, suponiendo que `conn` es una referencia a la conexión (ver apartado anterior) y que se quiere usar en la consulta un parámetro de la petición de nombre ciudad, la creación de la sentencia y su posterior ejecución (`executeQuery` se verá en el apartado siguiente) sería:

```
Statement st = conn.createStatement();  
String sql =  
    "SELECT * FROM p WHERE ciudad='"+request.getParameter("ciudad")+"'";  
ResultSet rs = st.executeQuery(sql);
```

TAREAS

- a) Consulte la documentación la clase `DriverManager` y compruebe que el valor devuelto por `getConnection` implementa la interfaz `Connection`:
<http://docs.oracle.com/javase/7/docs/api/java/sql/DriverManager.html>
- b) Consulte la documentación la interfaz `DataSource` y compruebe que el valor devuelto por `getConnection` implementa la interfaz `Connection`:
<https://docs.oracle.com/javase/7/docs/api/javax/sql/DataSource.html>
- c) Consulte la documentación la interfaz `Connection` y busque la especificación del método `createStatement`. Compruebe que no es estático (no tiene calificador `static`) y que el valor devuelto implementa la interfaz `Statement`.
<http://docs.oracle.com/javase/7/docs/api/java/sql/Connection.html>
- d) Compruebe el funcionamiento de:
<http://localhost:8080/AppWeb/jspdb/buscaPiezas1bis.jsp>

Asegúrese antes de que el SGBD está en servicio y la BBDD tienen datos:

```
dit@localhost:~/workspace/AppWeb/spj$ start-postgresql
dit@localhost:~/workspace/AppWeb/spj$ psql < recarga_spj.sql
```

- e) Observe el código del fichero `jsp` y compruebe cómo se crea la sentencia `sql`. ¿El valor del parámetro debe ir entre comillas? ¿Es necesario poner el carácter `'` en la sentencia `sql`? Pruebe con y sin `'`.
- f) Desde el navegador, introduzca en el campo del formulario una comilla simple y observe el error que aparece, le está indicando la sentencia `sql` que se ejecuta en el servidor.

Observe que la comilla simple aparece entre otras dos comillas simples.

- g) Introduzca en el campo del formulario lo siguiente (tal cual, con la comilla simple al principio) y observe el error:

```
' or 1=1
```

¿Le sugiere cómo puede modificar la entrada para obtener información extra del servidor?

- h) Introduzca en el campo del formulario lo siguiente (tal cual, con la comilla simple al principio y delante del segundo 1):

```
' or 1='1
```

¿Obtiene información no esperada del servidor? Esto es un ejemplo de inyección SQL, y es consecuencia de no tratar adecuadamente los datos de entrada que provienen del cliente.

3.1.2.2 Uso de `PreparedStatement`

El objeto `PreparedStatement` es similar a `Statement`, pero permite evitar la inyección de SQL. Es muy útil cuando dentro de la sentencia se va a usar parámetros que se obtienen de la petición, y que podrían contener caracteres “peligrosos” (pueden modificar la consulta).

Antes de crear el objeto se debe tener una conexión establecida con el SGBD. si se quiere hacer una consulta igual a la del apartado anterior:

```
String sql = "SELECT * FROM p WHERE ciudad=?";
PreparedStatement st = conn.prepareStatement(sql);
st.setString(1, request.getParameter("ciudad"));
```

Observe que la cadena de caracteres contiene el carácter interrogación (puede aparecer más de una vez) y que con el método `setString` se asigna un valor a la posición donde aparece una interrogación. si hubiera dos interrogaciones, habría que usar un 2 como primer parámetro para asignar el valor a la segunda interrogación.

TAREAS

- Consulte la documentación la interfaz `PreparedStatement` y busque la especificación del método `setString`. Compruebe que la numeración de los índices empieza en 1 y no en 0.
<https://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html>
- Consulte la documentación la interfaz `Connection` y busque la especificación del método `prepareStatement`. Compruebe que el valor devuelto implementa la interfaz `PreparedStatement`.
<http://docs.oracle.com/javase/7/docs/api/java/sql/Connection.html>
- Compruebe el funcionamiento de:
<http://localhost:8080/AppWeb/jspdb/buscaPiezas2.jsp>
Asegúrese antes de que el SGBD está en servicio y la BBDD tienen datos:

```
dit@localhost:~/workspace/AppWeb/spj$ start-postgresql
dit@localhost:~/workspace/AppWeb/spj$ psql < recarga_spj.sql
```
- Observe el código del fichero `jsp` y compruebe cómo se crea la sentencia `sql`.
¿Es necesario introducir el carácter interrogación entre comillas?
- Intente realizar la inyección SQL del aparatado anterior.
- Observe que se ha utilizado `DriverManager` para el acceso al SGBD. Copie `buscaPiezas2.jsp` en `buscaPiezas2bis.jsp` y modifíquelo para que el acceso sea mediante `Datasource`. El funcionamiento debe ser igual.

3.1.3 Ejecución de sentencia SQL

La sentencia SQL se ejecuta a través de los métodos `executeQuery` o `executeUpdate`, dependiendo de que exista o no modificación de los datos, respectivamente. Esto es:

- Consulta de datos mediante `executeQuery()`: es una operación `SELECT` que devuelve el resultado de una consulta encapsulado en un objeto de tipo `ResultSet`.
- Actualización de datos mediante `executeUpdate()`: es una operación de creación, modificación o borrado de tuplas de una tabla (`INSERT`, `UPDATE` o `DELETE`). También se utiliza para sentencias de tipo DDL: creación de tablas (`CREATE`), destrucción de tablas (`DROP`), modificación de estructuras de tablas (`ALTER`), etc. El resultado es un valor entero indicando el número de tuplas afectadas por esta operación.

Ejemplos:

```

Connection conn = DriverManager.getConnection(url, user, pass);
Statement sentencia = conn.createStatement();
ResultSet resultado = sentencia.executeQuery("SELECT a, b, c FROM
tabla2");
Int filas_afectadas = sentencia.executeUpdate("UPDATE personas SET
nombre='Marina' WHERE id=4");
Int filas_afectadas = sentencia.executeUpdate("INSERT INTO contacto
(nombre, apellidos, telefono) VALUES ('Juan', 'Gomez', '123')");

```

TAREAS

- a) Consulte la documentación la interfaz Statement y compruebe que contiene los métodos `executeQuery` y `executeUpdate`.
<http://docs.oracle.com/javase/7/docs/api/java/sql/Statement.html>
- b) Consulte la documentación y compruebe que el valor devuelto por `executeQuery` implementa la interfaz `ResultSet`, pero el valor devuelto por `executeUpdate` es del tipo `int`.

3.1.4 Procesamiento de resultados

Un objeto `ResultSet` contiene todos los registros (tuplas, filas) que satisfacen las condiciones impuestas en una sentencia SQL ejecutada, y proporciona acceso a los datos en dichos registros a través de un conjunto de métodos `get`, que permiten acceder a los diferentes campos o atributos (columnas) del registro actual. Un objeto `ResultSet` mantiene un cursor que apunta al registro actual. El método `ResultSet.next()` se usa para moverse al siguiente registro del `ResultSet`, haciendo el siguiente registro el actual.

Los métodos `get` proporcionan los medios para obtener los valores de los campos, atributos o columnas del registro actual. Para cada registro, los valores de las columnas pueden ser obtenidos en cualquier orden, pero para la mayor portabilidad, se debe hacer de izquierda a derecha y leer el valor de la columna sólo una vez. Tanto el nombre de la columna como el número de ésta se pueden emplear para designar la columna de la cual se quiere obtener el valor. Por ejemplo, si la columna “a” es de tipo entero, la “b” del tipo cadena de caracteres y la “c” de tipo coma flotante, la siguiente porción de código imprimiría los valores de todos los registros:

```

ResultSet resultado = sentencia.executeQuery("SELECT a, b, c FROM
tabla2");

while(resultado.next())
{
    int i = resultado.getInt("a");
    String s = resultado.getString("b");
    Float f = resultado.getFloat("c");
    System.out.println("FILA= " + i + " " + s + " " + f);
}

```

O también:

```

while(resultado.next())
{

```



```

int i = resultado.getInt(1);
String s = resultado.getString(2);
Float f = resultado.getFloat(3);
System.out.println("FILA= " + i + " " + s + " " + f);
}

```

TAREAS

- Consulte la documentación la interfaz `ResultSet` y compruebe que contiene el método `next` y que devuelve un `boolean`.
<http://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html>
- Consulte la documentación la interfaz `ResultSet` y compruebe que contiene los métodos `getInt`, `getString` y `getFloat`.
- Consulte la documentación y compruebe que los nombres de los métodos están sobrecargados y que uno de los métodos toma como parámetro `String` y otro toma `int`.

En la tabla siguiente se muestra el método adecuado para recuperar los distintos tipos de datos:

TIPO DE DATO SQL	TIPO DE DATO JAVA DEVUELTO POR <code>GETOBJECT()</code>	MÉTODO <code>GET</code> APROPIADO
BIGINT	Long	long <code>getLong()</code>
BINARY	byte[]	byte[] <code>getBytes()</code>
BIT	Boolean	boolean <code>getBoolean()</code>
CHAR	String	String <code>getString()</code>
DATE	java.sql.Date	java.sql.Date <code>getDate()</code>
DECIMAL	java.math.BigDecimal	java.math.BigDecimal <code>getBigDecimal()</code>
DOUBLE	Double	double <code>getDouble()</code>
FLOAT	Double	double <code>getDouble()</code>
INTEGER	Integer	int <code>getInt()</code>
LONGVARBINARY	byte[]	InputStream <code>getBinaryStream()</code>
LONGVARCHAR	String	InputStream <code>getAsciiStream()</code> InputStream <code>getUnicodeStream()</code>
NUMERIC	java.math.BigDecimal	java.math.BigDecimal <code>getBigDecimal()</code>
REAL	Float	float <code>getFloat()</code>
SMALLINT	Integer	short <code>getShort()</code>
TIME	java.sql.Time	java.sql.Time <code>getTime()</code>
TIMESTAMP	java.sql.Timestamp	java.sql.Timestamp <code>getTimestamp()</code>
TINYINT	Integer	byte <code>getByte()</code>
VARBINARY	byte[]	byte[] <code>getBytes()</code>

VARCHAR	String	String getString()
VARCHAR2	String	String getString()

Tabla 2. Tipos de datos SQL y Java, y método getXxx asociado para recuperarlos.

3.1.4.1 Métodos relacionados con el cursor de ResultSet

Dentro del conjunto de datos devueltos en el objeto `ResultSet`, se puede trabajar con el cursor con diferentes métodos.

TAREAS

- Consulte la documentación la interfaz `ResultSet` y compruebe que contiene los siguientes métodos (que mueven el cursor sobre los datos devueltos):
 - `first()`: moverá el cursor al principio.
 - `last()`: moverá el cursor al final
 - `beforeFirst()`: moverá el cursor antes del principio.
 - `afterLast()`: moverá el cursor después del final.
 - `absolute (int num)`: moverá el cursor al número de fila indicado en su argumento. Si el número es positivo, el cursor se mueve al número de posiciones a contar desde el principio; por ejemplo, `absolute (1)` ubica el cursor en la primera fila. Si el número es negativo, mueve el cursor al número dado desde el final; por ejemplo, `absolute (-1)` ubica el cursor en la última fila.
 - `next()`: mueve el cursor a la fila siguiente.
 - `previous()`: mueve el cursor a la fila anterior.
- Consulte la documentación y compruebe que contiene los siguientes métodos (que comprueba la posición del cursor sobre los datos devueltos):
 - `isFirst()`
 - `isLast()`
 - `isBeforeFirst()`
 - `isAfterLast()`
- Consulte la documentación y compruebe que contiene el siguiente método (que indica la posición del cursor sobre los datos devueltos):
 - `getRow()`: permite comprobar el número de fila donde está el cursor.

3.1.4.2 Tratamiento de excepciones

La mayor parte de las operaciones que nos proporciona el API JDBC lanzarán la excepción `java.sql.SQLException` en caso de que se produzca algún error en el acceso a la base de datos. Ejemplo de errores: errores en la conexión, sentencias SQL incorrectas, falta de privilegios, etc. Por este motivo, es necesario dar un tratamiento adecuado a estas excepciones y encerrar todo el código JDBC entre bloques `try/catch`.

Ejemplo:

```
try{
```

```
// Conexión a BBDD y procesamiento de resultados

...

}catch(SQLException se){
// Informamos al usuario de los errores SQL que se han producido
out.println("SQL Exception: " + se.getMessage());
se.printStackTrace(System.out);
}
```

3.1.5 Cierre de la conexión

Para finalizar correctamente el procedimiento de conexión a BBDD y procesamiento de datos, es necesario cerrar adecuadamente el objeto con los resultados (ResultSet), la sentencia (Statement o PreparedStatement) y la conexión (Connection):

```
resultado.close();

sentencia.close();

conn.close();
```

TAREAS

- a) Consulte la documentación y compruebe que contienen el método `close`.
<http://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html>
<http://docs.oracle.com/javase/7/docs/api/java/sql/Statement.html>
<http://docs.oracle.com/javase/7/docs/api/java/sql/Connection.html>
- b) Compruebe que no devuelve nada (`void`).

3.1.6 Resumen de funciones JDBC

A continuación, un listado de las funciones JDBC empleadas para la conexión a BBDD y ejecución y procesamiento de consultas SQL sobre las mismas.

Etapas	Descripción	Java	
A	Abrir conexión	Class.forName("sgbd.Driver") DriverManager. getConnection(url,user,pass)	InitialContext. lookup("java:comp/env/jdbc/dit"); DataSource. getConnection()
B	Creación de la sentencia SQL	Connection. createStatement()	Connection. prepareStatement(sql)
C	Ejecución de sentencia	Statement. executeQuery(sql) executeUpdate(sql)	PreparedStatement. executeQuery() executeUpdate()
D	Procesamiento de resultados	ResultSet. getMetaData() ResultSetMetaData. getColumnCount() getColumnLabel() getColumnClassName() ...	
		ResultSet. next() getFloat() getInt() getString() ...	
E	Cierre de la conexión (con liberación de recursos)	ResultSet.close() Statement.close() Connection.close()	

Tabla 3. Funciones Java para acceso a BBDDs desde Servlets/JSP mediante JDBC.

3.2 Acceso al SGBD de PostgreSQL

Durante la práctica, realizaremos conexiones y consultas mediante JDBC (desde páginas JSP) al servidor local de PostgreSQL. Veremos algunos ejemplos y ejercicios propuestos.

3.2.1 Ejemplo 1: administrador de usuarios web

En este ejemplo, veremos cómo acceder a los registros almacenados en una tabla de usuarios desde una página JSP, y los presentamos en una tabla HTML al navegador de un cliente remoto.

TAREAS

Desde el directorio donde están los ficheros con las sentencias sql, accediendo al intérprete psql, crearemos una tabla de nombre “usuarios”, que almacenará usuarios y sus contraseñas asociadas, insertamos 2 usuarios y salimos del intérprete:

```
dit@localhost:~$ cd /home/dit/workspace/AppWeb/spj
dit@localhost:~/workspace/AppWeb/spj$ cat usuarios.sql
-- Table: usuarios
CREATE TABLE usuarios (
  name CHAR(20) NOT NULL,
  password CHAR(20) NOT NULL,
  PRIMARY KEY (name)
);
INSERT INTO usuarios (name,password) VALUES ('user1', 'pass1');
INSERT INTO usuarios (name,password) VALUES ('user2', 'pass2');
SELECT * FROM usuarios;
dit@localhost:~/workspace/AppWeb/spj$ psql < usuarios.sql
dit@localhost:~/workspace/AppWeb/spj$
```

Observe el contenido del siguiente fichero, el cual obtiene los nombres y contraseñas almacenados en la tabla “usuarios” de la BBDD “dit” del servidor PostgreSQL local:

AppWeb/WebContent/jspdb/usuarios.jsp

```
<%@ page language="java" import="java.sql.*" contentType="text/html;
charset=UTF-8" pageEncoding="UTF-8"%>
<%
  try {
    Class.forName("org.postgresql.Driver");

    String url = "jdbc:postgresql://localhost:5432/dit";
    String user = "dit";
    String pass = "dit";
    Connection conn = DriverManager.getConnection(url, user, pass);

    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery("SELECT * FROM usuarios");

%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml">

  <head>
    <title>JSP: conexion JDBC a BBDD PostgreSQL</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  </head>

  <body>
    <table>
      <tr>
        <th>User</th><th>Password</th>
      </tr>
      <%
        while(rs.next())
        {
          String name = rs.getString("name");
          String password = rs.getString("password");
        %>

        <tr>
          <td><%=name%></td><td><%=password%></td>
        </tr>

        <%
          }

          rs.close();
          st.close();
          conn.close();

          } catch (SQLException e) {
            out.println("Excepción SQL Exception: " + e.getMessage());
            e.printStackTrace();
          }
        %>

      </table>

    </body>
  </html>
```

TAREAS

- a) Observe el contenido de

```
/home/dit/workspace/AppWeb/WebContent/jspdb/usuarios.jsp
```

y acceda a

<http://localhost:8080/AppWeb/jspdb/usuarios.jsp>

para visualizar los resultados. Nota: recuerde que tienen que estar ejecutándose los servidores (tomcat y postgresql) y que si tomcat informa que el recurso no está disponible es posible que tenga que recargar las aplicaciones (ver manager en práctica anterior).

- b) Pruebe a añadir más registros de usuario/contraseña desde el intérprete psql y recargue la página del navegador para ver los resultados.

```
dit@localhost:~/workspace/AppWeb/spj$ cat usuarios.txt
user3 pass3
user4 pass4
dit@localhost:~/workspace/AppWeb/spj$ psql
dit=> \copy usuarios from usuarios.txt
```

3.2.2 Ejemplo 2: listado de mascotas

En este ejemplo, veremos cómo acceder a los registros almacenados en una tabla de animales desde una página JSP, y los presentamos en una tabla HTML al navegador de un cliente remoto.

TAREAS

a) Acceda al directorio:

```
/home/dit/workspace/AppWeb/WebContent/jspdb
```

b) En primer lugar, accediendo al intérprete `psql`, crearemos una tabla de nombre “mascotas”, que almacenará animales y propietarios (ambos clave primaria) y otros datos:

```
dit@localhost:~/workspace/AppWeb/WebContent/jspdb$ cat mascotas.sql
DROP TABLE mascotas;
CREATE TABLE mascotas(
nombre VARCHAR(20),
propietario VARCHAR(20),
especie VARCHAR(20),
sexo CHAR(1),
nacimiento DATE,
fallecimiento DATE,
PRIMARY KEY (nombre,propietario));
dit@localhost:~/workspace/AppWeb/WebContent/jspdb$ psql < mascotas.sql
```

c) A continuación, utilizamos el fichero

```
dit@localhost:~/workspace/AppWeb/WebContent/jspdb$ psql
dit=> \copy mascotas from mascotas.txt
```

Por último, salimos del intérprete `psql`:

```
dit=> \q
```

Observe el contenido del siguiente fichero, el cual obtiene los registros contenidos en la tabla “mascotas” del servidor local de PostgreSQL, y los muestra en la ventana del navegador:

AppWeb/WebContent/jspdb/mascotas.jsp

```
<%@ page language="java" import="java.sql.*" contentType="text/html;
charset=UTF-8" pageEncoding="UTF-8" %>
<%
try {

    Class.forName("org.postgresql.Driver");

    String url = "jdbc:postgresql://localhost:5432/dit";
    String user = "dit";
    String pass = "dit";
    Connection conn = DriverManager.getConnection(url, user, pass);

    Statement st = conn.createStatement();
    ResultSet rs = st.executeQuery("SELECT * FROM mascotas");

}%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
```



```

<head>
  <title>JSP: conexion JDBC a BBDD PostgreSQL</title>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
</head>

<body>
  <table>
    <tr>

<th>Nombre</th><th>Propietario</th><th>Especie</th><th>Sexo</th><th>Nacimie
nto</th><th>Fallecimiento</th>
    </tr>
    <%
      while(rs.next())
      {
        String nombre = rs.getString("nombre");
        String prop = rs.getString("propietario");
        String esp = rs.getString("especie");
        String sexo = rs.getString("sexo");
        String nac = rs.getString("nacimiento");
        String ff = rs.getString("fallecimiento");
      %>

    <tr>
      <td><%=nombre%></td>
      <td><%=prop%></td>
      <td><%=esp%></td>
      <td><% if(sexo!=null) out.print(sexo); else out.print("-");%></td>
      <td><%=nac%></td>
      <td><% if(ff!=null) out.print(ff); else out.print("-");%></td>
    </tr>

    <%
      }

      // Liberamos los recursos para realizar otra consulta
      rs.close();

      // Mascotas que nacieron despues del anyo 2000:
      rs = st.executeQuery("SELECT nombre FROM mascotas WHERE nacimiento >
'1999-12-31'");

    %>
  </table>

  <br/><br/>

  <span>Consulta (a): Mascotas que nacieron despues de 2000:</span>

  <br/><br/>

  <table>
    <tr>
      <th>Nombre</th>

```

```
</tr>
<%
    while(rs.next())
    {
        String nombre = rs.getString("nombre");
    %>
    <tr>
        <td><%=nombre%></td>
    </tr>

    <%
        }

        rs.close();

        st.close();
        conn.close();

    } catch (SQLException e) {
        out.println("Excepción SQL Exception: " + e.getMessage());
        e.printStackTrace();
    }
    %>
</table>

</body>
</html>
```

TAREAS

- a) Observe el contenido de

```
/home/dit/workspace/AppWeb/WebContent/jspdb/mascotas.jsp
```

y acceda a

<http://localhost:8080/AppWeb/jspdb/mascotas.jsp>

para visualizar los resultados.

- b) Pruebe a añadir más registros de animales desde el intérprete psql y recargue la página del navegador para ver los resultados.

```
dit=> \copy mascotas from 'mascotas2.txt';
```

3.2.2.1 Ejercicio: consultas sobre mascotas

TAREAS

Modifique el fichero “mascotas.jsp” para que contenga las siguientes consultas:

- Mostrar los nombres de las mascotas que nacieron en el año 2000 o posterior (ya está hecha)
- Listar los perros hembra.
- Listar las aves y los gatos.

3.2.3 Ejemplo 3: agenda de contactos

TAREAS

- 1) En primer lugar, sobre la BBDD “dit” de PostgreSQL, creamos la tabla “contactos”:

```
dit@localhost:~/workspace/AppWeb/WebContent/jspdb $ cat contactos.sql
DROP TABLE contactos;
CREATE TABLE contactos(
num SERIAL,
nombre VARCHAR(20),
apellidos VARCHAR(20),
email VARCHAR(20),
telefono VARCHAR(10),
PRIMARY KEY (num));
dit@localhost:~/workspace/AppWeb/WebContent/jspdb$ psql < contactos.sql
```

- 2) Observe el contenido de los ficheros agenda_inserta.jsp y agenda_lista.jsp que se exponen a continuación:

AppWeb/WebContent/jspdb/agenda_inserta.jsp

```

<%@ page language="java" import="java.sql.*" contentType="text/html;
charset=UTF-8" pageEncoding="UTF-8"%>
<%
    try {

        Class.forName("org.postgresql.Driver");

        String url = "jdbc:postgresql://localhost:5432/dit";
        String user = "dit";
        String pass = "dit";
        Connection conn = DriverManager.getConnection(url, user, pass);

        Statement st = conn.createStatement();
        int n = st.executeUpdate("INSERT INTO contactos (nombre, apellidos,
email, telefono) VALUES ('Pedro', 'Diaz Lopez', 'pdlopez@afast.us.es',
'666777888')");

        n = st.executeUpdate("INSERT INTO contactos (nombre, apellidos, email,
telefono) VALUES ('Ana', 'Sierra Collado', 'asc@afast.us.es', '666888999')");

        st.close();
        conn.close();

    } catch (SQLException e) {
        out.println("Excepción SQL Exception: " + e.getMessage());
        e.printStackTrace();
    }
%>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

    <head>
        <title>Agenda</title>
        <meta http-equiv="content-type" content="text/html; charset=utf-8" />
        <link rel="stylesheet" type="text/css" href="estilo.css" />
    </head>

    <body>
        <p>Contactos insertados.</p>
    </body>
</html>

```

AppWeb/WebContent/jspdb/agenda_lista.jsp

```

<%@ page language="java" import="java.sql.*" contentType="text/html;
charset=UTF-8" pageEncoding="UTF-8"%>
<%
    try {

        Class.forName("org.postgresql.Driver");

        String url = "jdbc:postgresql://localhost:5432/dit";
        String user = "dit";
        String pass = "dit";
        Connection conn = DriverManager.getConnection(url, user, pass);

        Statement st = conn.createStatement();
        ResultSet rs = st.executeQuery("SELECT * FROM contactos");
    }%>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

    <head>
        <title>Agenda</title>
        <meta http-equiv="content-type" content="text/html; charset=utf-8" />
        <link rel="stylesheet" type="text/css" href="estilo.css" />
    </head>

    <body>

        <%
            while(rs.next())
            {
                String nombre = rs.getString("nombre");
                String apellidos = rs.getString("apellidos");
                String email = rs.getString("email");
                String telefono = rs.getString("telefono");
            }%>

        <table>
            <tr>
                <td>Nombre: <strong><%=nombre%></strong></td>
                <td>Apellidos: <strong><%=apellidos%></strong></td>
            </tr>
            <tr>
                <td>e-mail: <strong><%=email%></strong></td>
                <td>Telefono: <strong><%=telefono%></strong></td>
            </tr>
        </table>
        <hr/>

        <%
            }

            // Liberamos los recursos para realizar otra consulta

```

```
rs.close();  
st.close();  
conn.close();  
  
} catch (SQLException e) {  
    out.println("Excepción SQL Exception: " + e.getMessage());  
    e.printStackTrace();  
}  
%>  
  
</body>  
</html>
```

TAREAS

- 1) Razone el contenido de los ficheros anteriores y ejecútelos en orden (accediendo desde un navegador a la ruta **http://localhost:8080/AppWeb/jspdb/...**): primero “agenda_inserta.jsp” y después “agenda_lista.jsp”.
- 2) Por último, desde un terminal ejecute lo siguiente:

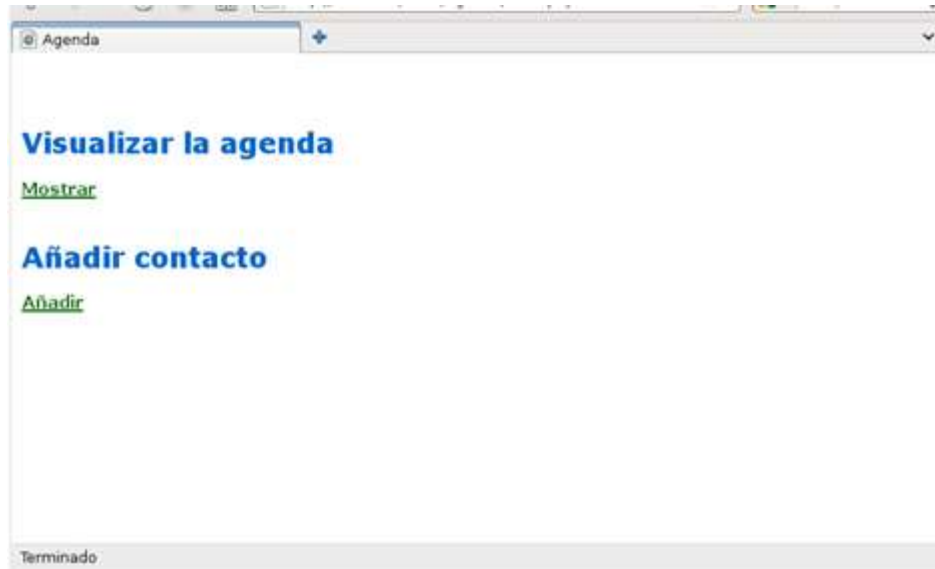
```
dit@localhost:~/workspace/AppWeb/WebContent/jspdb $ psql  
dit=> select * from contactos;
```

Contraste los resultados con los presentados anteriormente en el navegador.

3.2.3.1 Ejercicio: modificación de agenda de contactos

Se propone realizar el siguiente sitio web, basándose en el ejemplo anterior: inicialmente, se le mostrará al usuario una página de inicio (*inicio.html*) con dos enlaces, uno para mostrar la agenda (*agenda_recupera.jsp*, muy similar al ya usado *agenda_lista.jsp*) y otro para añadir contactos a la agenda (*agenda_add.jsp*).

La siguiente imagen presenta el resultado de la página *inicio.html*:



TAREAS

Complete el fichero `inicio.html` mediante el siguiente esqueleto:

AppWeb/WebContent/jspdb/inicio.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">

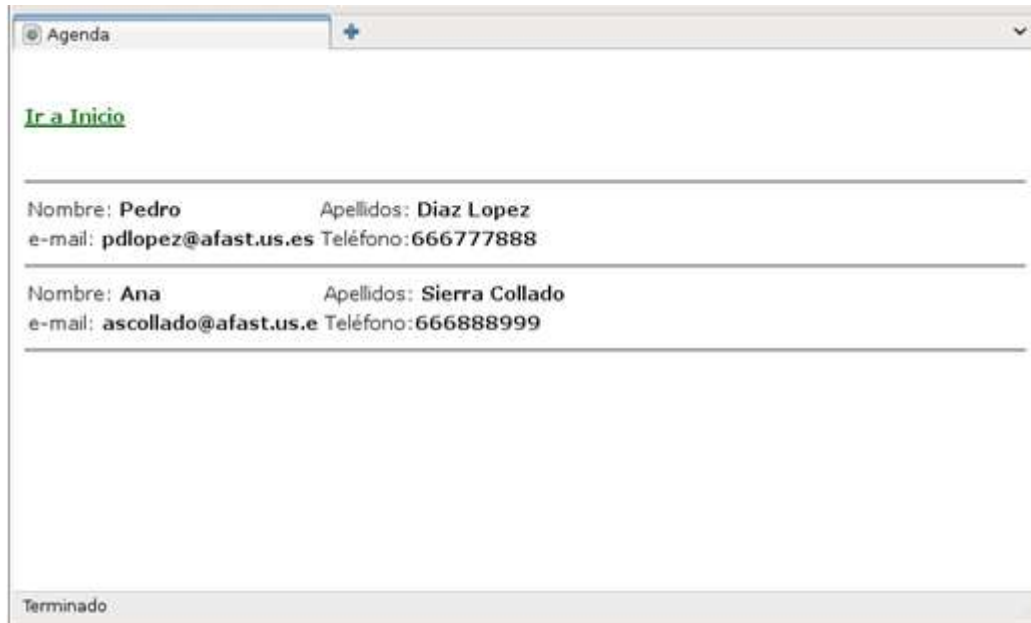
  <head>
    <title>Agenda</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-
8" />
    <link rel="stylesheet" type="text/css" href="estilo.css" />
  </head>

  <body>
    <!-- Enlaces para visualizar la agenda y añadir nuevo contacto:
-->
    ...

  </body>
</html>
```

La página `agenda_recupera.jsp` muestra los datos contenidos en la tabla “contactos”. También presenta un enlace a la página de inicio.

Observe la siguiente imagen para ver su funcionamiento:

**TAREAS**

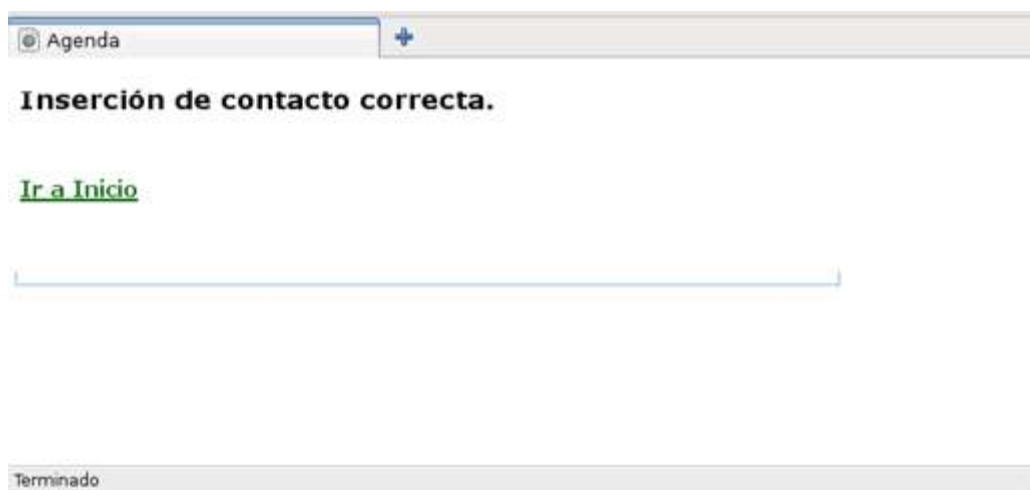
Copie el fichero `agenda_lista.jsp` en `agenda_recupera.jsp` y modifíquelo para que:

- Contenga un enlace a `inicio.html`
- Use el fichero de estilos `estilo.css`

El fichero `agenda_add.jsp`, encargado de añadir un contacto a la agenda, en primer lugar deberá presentar un **formulario**, y una vez introducidos los datos (nombre, apellidos, email y teléfono, todos campos obligatorios, excepto el teléfono – utilice *ECMAScript* para validar los datos –) los volcará a la tabla de contactos de la BBDD `dit`. Observe las siguientes imágenes para ver el funcionamiento:



The screenshot shows a web browser window titled "Agenda". The main heading is "Insertar nuevo contacto:". Below it is a form with four fields: "Nombre(*)" containing "Jose", "Apellidos(*)" containing "Perez Ruiz", "e-mail(*)" which is empty, and "Telefono:" which is empty. An "Add" button is at the bottom left of the form. A JavaScript alert dialog box is displayed over the form, with the title "La pagina en http://localhost dice:" and the message "Debe introducir los campos obligatorios (*)." with a yellow warning triangle icon. An "Aceptar" button is in the bottom right of the dialog. The browser's status bar at the bottom says "Terminado".



The screenshot shows the same "Agenda" browser window. The heading now reads "Inserción de contacto correcta." in bold. Below the heading is a green text link that says "Ir a Inicio". The status bar at the bottom still says "Terminado".

TAREAS

Complete el fichero `agenda_add.jsp` mediante el siguiente esqueleto:

AppWeb/WebContent/jspdb/agenda_add.jsp

```

<%@ page language="java" import="java.sql.*" contentType="text/html;
charset=UTF-8" pageEncoding="UTF-8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Agenda</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-
8" />
    <link rel="stylesheet" type="text/css" href="estilo.css" />
    <script type="text/javascript">
      function validaform() {

          //Validacion de campos de formulario

          //...

      }
    </script>

  </head>

  <body>
    <%
    if( ) {

        // Conexion a BBDD
        //...

        // Modifica la BBDD
        //...

    }

    else
    {

    %>

    <br/><br/>
    <h1>Insertar nuevo contacto:</h1>

    <form class="borde" action="agenda_add.jsp" method="post"
id="formulario" name="formulario" onsubmit="return validaform()">
    <table>
      <tr>
        <td><label for="nombre">Nombre(*):</label></td>
        <td><input type="text" name="nombre" id="nombre"/></td>
      </tr>

```

```

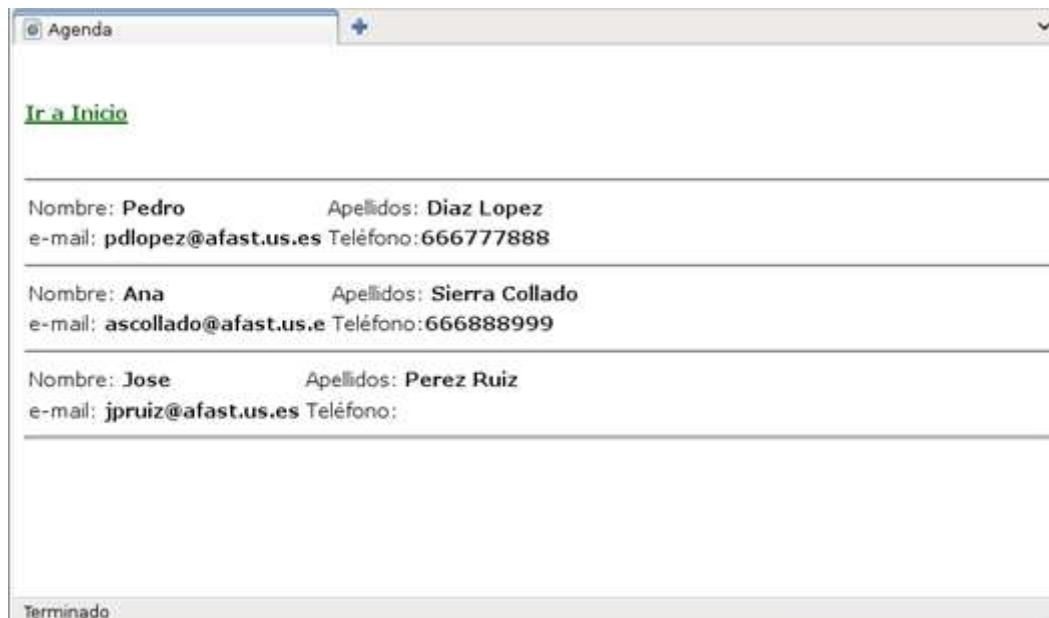
        <tr>
            <td><label for="ap">Apellidos(*):</label></td>
            <td><input type="text" name="apellidos" id="ap"/></td>
        </tr>
        <tr>
            <td><label for="mail">e-mail(*):</label></td>
            <td><input type="text" name="email" id="mail"/></td>
        </tr>
        <tr>
            <td><label for="tel">Teléfono:</label></td>
            <td><input type="text" name="telefono" id="tel"/></td>
        </tr>
    </table>
    <p><input type="submit" name="add" value="Add"/></p>
</form>

<%
}
%>

</body>
</html>

```

Ejemplo de listado de todos los contactos insertados, accediendo a `agenda_recupera.jsp`:



3.2.4 Ejemplo de uso de javabean y EL

En este apartado se va a hacer uso de un bean para acceder al SGBD y EL para mostrar la información. También se va a usar el bean para obtener los datos del formulario (ya visto en la práctica anterior).

Se parte de

<http://localhost:8080/AppWeb/jspdb/buscaPiezas5.jsp>

Observe el código y compruebe que se usa `DataSource` para acceder al SGBD y `PreparedStatement` para la sentencia sql.

A partir de este fichero se ha creado `buscaPiezas6.jsp` que tiene la misma funcionalidad, pero que usa los beans `Pieza` y `PiezasPorCiudad` (dentro del paquete `fast.db`).

Onserve la clase `Pieza`. Sus propiedades coinciden con los atributos de la tabla `p` de la BBDD, y tienen los métodos setters y getters correspondientes.

AppWeb/src/fast/db/Pieza.java

```
package fast.db;

import java.io.Serializable;

public class Pieza implements Serializable {

    private static final long serialVersionUID = 1L;

    Integer idp;
    String nombre;
    String color;
    Integer peso;
    String ciudad;

    public Integer getIdp() {
        return idp;
    }
    public void setIdp(Integer idp) {
        this.idp = idp;
    }
    public String getNombre() {
        return nombre;
    }
    public void setNombre(String nombre) {
        this.nombre = nombre;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    public Integer getPeso() {
        return peso;
    }
    public void setPeso(Integer peso) {
        this.peso = peso;
    }
    public String getCiudad() {
        return ciudad;
    }
}
```

```

    }
    public void setCiudad(String ciudad) {
        this.ciudad = ciudad;
    }
}

```

Observe la clase `PiezasPorCiudadBean`. Observe que tiene implementado sólo los métodos `setCiudad()` y `getPiezas()`, y también un constructor sin parámetros `PiezasPorCiudadBean()`. Compruebe que:

- al invocar al constructor se accede al SGBD y se prepara la sentencia sql.
- al llamar a `setCiudad()` se le da valor a la propiedad ciudad.
- al llamar a `getPiezas()` se devuelve una lista con el resultado de la consulta, usando como parámetro el valor de la propiedad ciudad.

AppWeb/src/fast/db/PiezasPorCiudadBean.java

```

package fast.db;

import java.io.Serializable;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import javax.naming.InitialContext;
import javax.sql.DataSource;

public class PiezasPorCiudadBean implements Serializable {

    private static final long serialVersionUID = 1L;

    String ciudad;
    Connection conn;
    PreparedStatement st;
    boolean conError = false;

    public PiezasPorCiudadBean() {
        try {
            // Usando DataSource ya definido en el servidor
            InitialContext ctx = new InitialContext();
            // /jdbc/dit is the name of the resource above
            DataSource ds = (DataSource)
ctx.lookup("java:comp/env/jdbc/dit");
            conn = ds.getConnection();

            String sql = "SELECT * FROM p WHERE ciudad=?";
            // Statement st = conn.createStatement();
            st = conn.prepareStatement(sql);
        } catch (Exception e) {
            conError = true;
        }
    }
}

```

```

public void setCiudad(String ciudad) {
    this.ciudad = ciudad;
}

public List<Pieza> getPiezas() throws SQLException {
    List<Pieza> resultado = null; // null = error
    if (!conError) {
        st.setString(1, ciudad);
        ResultSet rs = st.executeQuery();
        resultado = new ArrayList<Pieza>();
        while (rs.next()) {
            Pieza aux = new Pieza();
            aux.setIdp(rs.getInt("idp"));
            aux.setNombre(rs.getString("nomp"));
            aux.setColor(rs.getString("color"));
            aux.setPeso(rs.getInt("peso"));
            aux.setCiudad(rs.getString("ciudad"));
            resultado.add(aux);
        }
        rs.close();
    }
    return resultado;
}
}

```

Compruebe ahora el funcionamiento de:

<http://localhost:8080/AppWeb/jspdb/buscaPiezas6.jsp>

Asegúrese antes de que el SGBD está en servicio y la BBDD tiene datos:

```

dit@localhost:~/workspace/AppWeb/spj$ start-postgresql
dit@localhost:~/workspace/AppWeb/spj$ psql < recarga_spj.sql

```

Observe ahora el código de buscaPiezas6.jsp y compruebe que:

el dato del formulario se obtiene con:

```
<jsp:setProperty name="piezasPorCiudad" property="*" />
```

al crearse el bean se conecta al SGBD y se prepara la sentencia sql (ya que se invoca al constructor):

```
<jsp:useBean id="piezasPorCiudad" class="fast.db.PiezasPorCiudadBean" >
```

al llamar al método getPiezas() dentro del bucle for se hace la consulta al SGBD y la lista devuelta se usa en la variable p (que es un elemento de la lista):

```
for (Pieza p: piezasPorCiudad.getPiezas()) {
```

para poder acceder a la información desde una EL, la variable p se asocia con el atributo de mismo nombre:

```
pageContext.setAttribute("p", p);
```

una vez que el atributo es un bean, se puede acceder desde una EL a sus propiedades con el operador punto:

```
<td>${p.idp}</td><td>${p.nombre}</td><td>${p.color}</td>
```

AppWeb/WebContent/jspdb/buscaPiezas6.jsp

```
%@page import="fast.db.Pieza"%>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Buscador de piezas. Versión 6</title>
<link href="estiloconsulta.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <h1>Buscador de piezas de ciudades (v6)</h1>
    <div id="formulario">
        <form action="" method="post">
            <label for="ciudad">Ciudad donde está la pieza:</label> <input
                type="text" name="ciudad" /> <input type="submit" value="Enviar"
            />
        </form>
        <!-- Ejemplo de inyección SQL usando como ciudad:
        ' union select idj, nomj, nomj as color, idj as peso, ciudad from j --
        --%>
    </div>
    <%
    if (request.getParameter("ciudad") != null) {
    %>
    <div id="resultado">
        <!-- Usando un bean --%>
        <jsp:useBean id="piezasPorCiudad" class="fast.db.PiezasPorCiudadBean"
    >
            <jsp:setProperty name="piezasPorCiudad" property="*" />
        </jsp:useBean>

        <hr />
        <h2>Resultado de la búsqueda:</h2>
        <table id="resultados">
            <tr>

<th>Idp</th><th>Nombre</th><th>Color</th><th>Peso</th><th>Ciudad</th>
            </tr>
            <%
            for (Pieza p: piezasPorCiudad.getPiezas()) {
                pageContext.setAttribute("p", p);
                %>
                <tr>
                    <td>${p.idp}</td><td>${p.nombre}</td><td>${p.color}</td>
                    <td>${p.peso}</td><td>${p.ciudad}</td>
                </tr>
```

```
<%  
}  
%>  
  
</table>  
</div>  
<%  
}  
%>  
</body>  
</html>
```

TAREAS

- Modifique el código de `Pieza.java` y `PiezasPorCiudadBean.java`, cambiando el nombre de la propiedad ciudad por `city`, pero sin modificar el nombre de los métodos ¿hay que cambiar el código del jsp?
- A partir de `buscaPiezas6.jsp`, cree un nuevo fichero `buscaSuministradores1.jsp` que funcione de forma similar. Cree los beans necesarios.

4 Ejemplo de integración

4.1 Introducción

Este ejemplo forma parte de un ejercicio especialmente largo que usa conocimientos adquiridos en toda la asignatura, y en esta práctica debe **centrarse en la parte de BBDD** (vea más adelante el apartado **6.2 y siguientes** del Ejercicio en 4.2). La solución del resto del ejercicio le puede servir de ejemplos resueltos.

Se desea desarrollar una aplicación web llamada “Bloc de Notas – FAST”, cuya funcionalidad sea crear y mostrar notas (texto con un título y opcionalmente una imagen). Vea cómo funciona la aplicación ya resuelta (para ello siga las instrucciones del apartado del Ejercicio en 4.2):

<http://localhost:8080/AppWeb/blocSol/inicio.jsp>

Puede hacer pruebas en:

<http://localhost:8080/AppWeb/blocSol>

Para acceder a los ficheros originales desde Eclipse, búsquelos en el directorio:

AppWeb/WebContent/bloc

En el anterior directorio están los ficheros originales por si quisiera hacer todos los apartados. Busque en el fichero solución la solución a los apartados no relacionados con BBDD (HTML, CSS y ECMAScript). Ahora ya sólo tiene que modificar los ficheros relacionados con las notas y la base de datos para llegar a la solución, siguiendo las instrucciones del apartado 6.2 y siguientes del Ejercicio en 4.2.

4.2 Ejercicio

La aplicación se realizará en JSP utilizando sesiones y las notas se almacenarán en una BBDD PostgreSQL. Se utilizará HTML y CSS (separando totalmente el contenido del formato) para la presentación. Por último, también se utilizará ECMAScript (validación de formularios, AJAX, etc.).

Parte de la aplicación ya está creada, pero requiere modificaciones. A lo largo de este ejercicio deberá modificar los ficheros que ha descargado previamente, no tendrá que crear ningún fichero nuevo.

Puede ver capturas de pantalla de la solución final esperada en el directorio “capturas” que está junto con el código.

Apartado 1: Puesta en marcha de la aplicación descargada

Antes de nada, debe preparar el entorno y probar la aplicación en su estado inicial:

1. Cree la BBDD “notasfast”. Utilice el archivo “bbdd/tablas.sql” para crear las tablas y datos de ejemplo.
2. Estudie la estructura de las tablas. En este ejercicio no haremos uso de la tabla “usuarios”. Usaremos un único usuario con nombre “usuario” y clave “clave” con identificador “1”.
3. Resto de ficheros:
 - `index.html`: solo se usa para hacer pruebas. No hace falta entregarlo. Utilícelo como medio para probar las distintas páginas web y como acceso rápido durante el desarrollo (por ejemplo, para poder continuar si no consigue realizar uno de los apartados).
 - `detallenota.jsp`: no es una página web, solo genera código JSON con información de una determinada nota.
 - `borrarnota.jsp`: no es una página web, solo genera código JSON con información de si se ha producido algún error al borrar una determinada nota.
 - `cabecera.jsp`: cabecera común a todas las páginas. Incluye la etiqueta de comienzo `<html>`, el elemento `<head>` y la etiqueta de comienzo `<body>`. Inicia la sesión web, y verifica que la página puede ser accedida por el usuario actual.
 - `pie.jsp`: pie común a todas las páginas. Incluye las etiquetas de cierre de `body` y `html`.
 - `inicio.jsp`: página de inicio a la aplicación. También sirve para cerrar la sesión si se le pasa el parámetro “salir”.
 - `menu.jsp`: menú de acciones a realizar con las notas, que actualmente, son crear y mostrar las notas.
 - `crearnota.jsp`: página que contiene un formulario para crear nuevas notas.
 - `listarnotas.jsp`: página que muestra todas las notas del usuario. Inicialmente mostrará un listado con todos los títulos de las notas y cuando el usuario seleccione un título, recuperará el resto de detalles de la nota usando AJAX y además mostrará un botón para poder borrar la nota.
 - `listarnotas.js`: código ECMAScript utilizado en `listarnotas.jsp`.
 - `estilo.css`: estilos CSS utilizados en todas las páginas.
 - Subdirectorio `imagen`: imágenes usadas en la aplicación.

Apartado 2: Modificación del pie de página

Haga las modificaciones oportunas para hacer que el pie de página:

1. contenga sus apellidos y nombre.
2. sea siempre visible (aunque el tamaño de la ventana sea muy pequeño),
3. esté situado en la parte inferior de la ventana y ocupe todo el ancho de la ventana,
4. tenga un borde superior de color gris y el color de fondo sea blanco,
5. el tamaño de letra será un 85% menor al tamaño normal.

Capturas útiles: en todas menos las “detalle-x.png” puede ver el aspecto final del pie de página. En “inicio-2.png” puede observar cómo el pie de página siempre está visible.

Apartado 3: Modificación de la cabecera de página

Haga los siguientes cambios en la cabecera común:

1. La imagen con el logotipo no debe aparecer en la página inicial, pero si en el resto.
2. Insertar un enlace en la imagen, para que al hacer click se muestre la página `menu.jsp`.
3. Los elementos DIV de clase “acceso” deben estar situados a la derecha de la ventana, a la misma altura que la imagen del logotipo.
4. Añada código JavaScript a las funciones “muestraFormAcceso” y “ocultaFormAcceso”, de tal manera que el formulario de acceso se muestre o se oculte, respectivamente.
5. Añada un nuevo campo al formulario de acceso para escribir la clave, de nombre “clave” y que oculte los caracteres que escriba el usuario.

Capturas útiles: en “inicio-1.png” puede ver que en la página de inicio no aparece el logotipo en la cabecera. En “inicio-3.png” e “inicio-4.png” puede ver el aspecto del formulario de acceso. En “inicio-5.png” puede ver el resultado de intentar ver una página sin autorización.

Apartado 4: Modificación de la página de inicio

Haga los siguientes cambios:

1. Introduzca la imagen “imagen/blocnotasfast.png” después de la línea de encabezado de nivel 1. Ponga como texto alternativo “Bloc de Notas FAST”.
2. Modifique el estilo para los elementos con identificador “bienvenida”, “menu”, “crear” y “lista” para que el texto aparezca centrado y no permita solapamientos de elementos flotantes ni a izquierda ni a derecha.
3. Evite que se cierre la sesión cuando se cargue esta página a menos que se le pase en la petición HTTP el parámetro “salir” (ya sea mediante GET o POST).

Capturas útiles: en “inicio-1.png” puede ver el logotipo insertado. En “inicio-1.png”, “menu-1.png”, “crear-1.png” y “lista-1.png” puede ver como aparece el texto central. En “inicio-6.png”, se muestra el acceso a la página de inicio con la sesión abierta.

Apartado 5: Modificación de la página de menú

Haga los siguientes cambios:

1. Haga que los elementos div contenidos dentro del elemento con identificador “menu” tengan como imagen de fondo “imagen/fondonota.jpg”.

2. Haga que el texto de los elementos del menú (“Nueva” y “Mostrar”) queden aproximadamente en el centro de la imagen de fondo del punto anterior.
3. La variable `$usuario` solo debería ser definida si en la petición HTTP (GET o POST) se pasan como parámetros “usuario” (con valor “usuario”) y “clave” (con valor “clave”). Si se cumple esa condición la variable `$usuario` tendrá el mismo valor que el parámetro “usuario”.

Capturas útiles: en “inicio-5.png” puede ver el resultado de acceder con un usuario y/o clave incorrectos. En “menu-1.png” y “menu-2.png” se muestra el aspecto final del menú.

Apartado 6: Modificación de la página de creación de notas

Haga los siguientes cambios:

1. Complete la función JavaScript “validacion” y modifique el formulario, de tal manera que antes de enviar los datos, si el campo “titulo” está vacío se aborte el envío mostrando un mensaje de error (use la función “alert” de JavaScript para esto).
2. Modifique el código JSP para que se guarde la nota en la BBDD correctamente. Las líneas de código que tiene que modificar están marcadas.

En este ejercicio supondremos que el usuario nunca introducirá en ninguno de los campos de la nota los caracteres ' comilla simple, “ doble comilla o intro. Cuando haga pruebas, tenga en cuenta esto, ya que pueden dar errores de distinto tipo. Una aplicación real debería verificar completamente la información introducida por el usuario.

Capturas útiles: en “crear-1.png” puede ver el aspecto final de la página. En “crear-2.png” se ve el error que se muestra cuando se intenta crear una nota sin título. En “crear-3.png” se ve el error que se muestra cuando se intenta crear una nota con un título que incluye una comilla simple. En “crear-4.png” se ve el mensaje de éxito que aparece cuando se inserta una nota correctamente.

Apartado 7: Modificación de la página de listado de notas

Haga los siguientes cambios:

1. En el código suministrado, no se obtiene el listado de notas de la BBDD, sino que se está utilizando unos valores fijos. Modifique el código JSP para generar el listado con todas las notas del usuario actual haciendo una petición a la BBDD. La consulta debe obtener los campos “id” y “titulo” de todas las notas del usuario cuyo identificador está almacenado en la variable de sesión ‘usuario’. En el código fuente está indicado donde debe hacer la modificación.
2. Modificar la función JavaScript “mostrar” para que obtenga los detalles utilizando AJAX. La petición que hay que hacer ya está almacenada en la variable `peticion`. La página `detallenota.jsp` generará una respuesta JSON. Esta respuesta deberá ser convertida a un objeto JavaScript y ser pasada a la función “muestraDetalle” que ya está creada.
3. Modificar la función JavaScript “borrar” para que borre una nota utilizando AJAX. La petición que hay que hacer ya está almacenada en la variable `peticion`. La página `borrarnota.jsp` generará una respuesta JSON. Esta respuesta deberá ser convertida a un objeto JavaScript y ser pasada a la función “procesarResultadoBorrar” que ya está creada.

Capturas útiles: en “lista-1.png” puede ver el aspecto final de la página, mostrando las notas de ejemplo. En “lista-2.png” se ve el resultado de seleccionar la segunda nota. En “lista-3.png” se ve el resultado de seleccionar la primera nota (y mantener el cursor sobre ella).

Apartado 8: Modificación de detallenota.jsp

Haga el siguiente cambio:

1. El código actual genera siempre una respuesta JSON de muestra. Modifique el código para que la respuesta se genere con los datos almacenados en la BBDD. Si no ha realizado el apartado 7.2, puede hacer pruebas con el enlace que aparece en la página `index.html`.

Capturas útiles: en “detalle-1.png” se ve la salida al preguntar sobre los detalles de la nota con id=1 (una de las notas de ejemplo). En “detalle-2.png” se ve la salida al preguntar sobre los detalles de la nota con id=0 (una que no existe). En ambos casos la sesión debe ser iniciada previamente (con usuario y clave correctos).

Apartado 9: Modificación de borrarnota.jsp

Haga el siguiente cambio:

1. El código actual genera siempre una respuesta JSON de muestra. Modifique el código para que la respuesta se genere con el resultado real del borrado. Si no ha realizado el apartado 7.3, puede hacer pruebas con el enlace que aparece en la página `index.html`.

Capturas útiles: en “borrarnota-1.png” se ve la salida al intentar borrar la nota con id=1 (una de las notas de ejemplo). En “borrarnota-2.png” se ve la salida al intentar borrar la nota con id=0 (una que no existe). En ambos casos la sesión debe ser iniciada previamente (con usuario y clave correctos).

5 Anexo no evaluable

5.1 Instalación de postgresql

Si no está instalado en el sistema, puede hacerlo como superusuario:

```
su -  
apt-get update  
apt-get install postgresql
```

En el siguiente apartado se explica cómo crear el usuario dit2 (en postgresql) y su base de datos asociada dit2. Si no tiene ningún usuario creado, puede seguir las instrucciones y crear el usuario dit en vez de dit2 (en postgresql) sustituyendo dit2 por dit. Si desea que no haya que introducir la clave cuando se conecte, cree un fichero de nombre .pgpass (tenga en cuenta que empieza por punto) que contenga el password (dit).

5.2 Creación de usuario en SGBD

Actualmente ya existe el usuario dit, así que se va a crear el usuario “dit2”, con password “dit2” (dentro del SGBD, no del sistema Linux). Para crear el usuario en el SGBD, realice las siguientes tareas.

TAREAS

- d) Desde el usuario root iniciar el SGBD si no está iniciado, y cambiar al usuario postgres:

```
su -  
service postgresql status  
service postgresql start  
su postgres
```
- e) Creación del usuario dit2 con password dit2, con permisos de creación de BBDD:

```
psql -c "CREATE USER dit2 CREATEDB PASSWORD 'dit2'" template1
```
- f) Comprobación de los usuarios existentes:

```
psql -c "\du"
```
- g) Ya puede volver al usuario dit:

```
exit  
exit
```

5.3 Cliente de PostgreSQL (intérprete de comandos)

La sintaxis del comando es:

```
psql [<opciones>] [<bbdd> [<usuario>]]
```

donde

- `<bddd>` es el nombre de la base de datos con la que se desea trabajar. Por defecto es aquella de idéntico nombre al del usuario.
- `<usuario>` indica el propietario de la base de datos con la que se desea trabajar. Si no se indica, es el mismo usuario que está ejecutando la Shell.

Y algunas de las opciones pueden ser:

- `-h <servidor>` especifica nombre o dirección IP en la que se encuentra el servidor PostgreSQL. Si no se especifica, considera que el servidor está en la máquina local.
- `-p <puerto>` fija el puerto en el que está escuchando el servidor. En ausencia de este parámetro, se supone que está escuchando en el puerto por defecto que es el 5432.
- `-c "<comando>"` sólo se utiliza si se desea ejecutar un único comando, y no ejecutar la Shell interactiva de PostgreSQL.
- `-U <usuario>` indica el propietario de la base de datos con la que se desea trabajar. Si no se indica, es el mismo usuario que está ejecutando la Shell.

En nuestro caso, con el usuario `dit`, bastará con escribir:

```
psql [-c "<comando>"] [<base de datos>]
```

5.4 Creación de una Base de Datos

El primer paso para trabajar con una base de datos es crearla dentro del SGBD. Para ello se utiliza la sentencia SQL `"CREATE DATABASE"`, que admite como parámetro el nombre de la base de datos a crear. En nuestro caso ya está creada una base de datos con nombre `dit`, con lo que en lo sucesivo no tendremos que indicar el nombre de la base de datos con la que deseamos trabajar al ejecutar el comando `psql`. Si quisiéramos crear otra base de datos `ditdbprueba`, como en `psql` es imprescindible trabajar con una base de datos, nos podemos apoyar en una vacía (`template1`) que incorpora PostgreSQL.

Ejecute lo siguiente:

```
psql -c "CREATE DATABASE ditdbprueba" template1
```

Ejecute las siguientes sentencias para listar los usuarios y las BBDD existentes, comprobar que se ha creado la base de datos “ditdbprueba” y eliminarla (la sentencia DROP DATABASE se verá más adelante), ya que se usará la base de datos dit (las sentencias que comienzan con \ se explican en el apartado siguiente):

```
psql
dit=> \du
dit=> \l
dit=> \c ditdbprueba
ditdbprueba=> \c dit
dit=> drop database ditdbprueba;
dit=> \q
```

Ejecute las siguientes sentencias para crear las BBDD ditdb2 y ditdb22 (y luego eliminar esta última) para el usuario dit2 (recuerde que su contraseña es dit2)

```
psql -c "CREATE DATABASE ditdb2" -h localhost template1 dit2
psql -c "CREATE DATABASE ditdb22" -h localhost template1 dit2
psql -h localhost ditdb2 dit2
ditdb2=> \l
ditdb2=> \c ditdb22
ditdb22=> \c ditdb2
ditdb2=> drop database ditdb22;
ditdb2=> \q
```

Alternativamente, PostgreSQL proporciona el comando createdb para el mismo efecto. Puede ver su funcionamiento ejecutando:

```
createdb --help
```

5.5 Eliminación de BBDD

La sintaxis SQL para la eliminación de una BBDD es la siguiente:

```
DROP DATABASE database
```