Bases de Datos con JAVA

Juan Carlos Pérez Rodríguez

Sumario

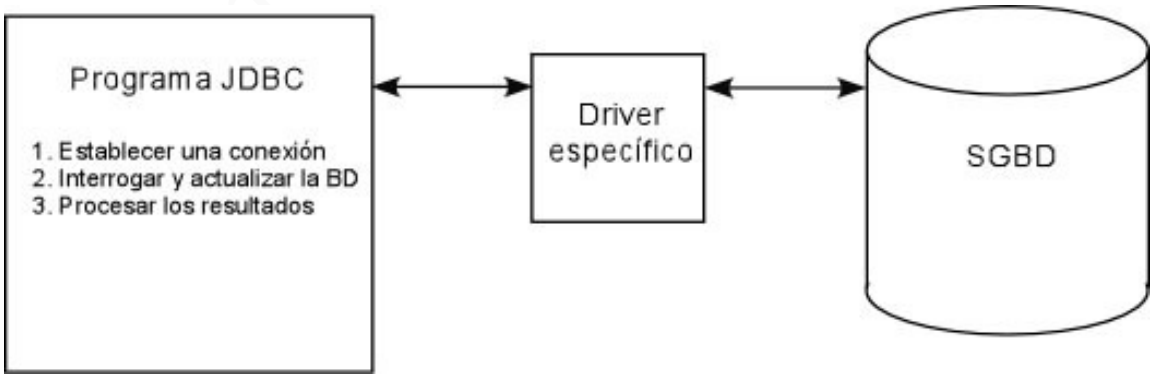
Introducción

Las bases de datos será nuestra forma habitual de mantener la información de nuestra aplicación. Son sistemas optimizados y por tanto los más eficientes para gestionar la persistencia de nuestra aplicación.

Dejaremos la parte de persistencia que corresponde a los ficheros a casos muy específicos (por ejemplo, en un editor de texto. Las preferencias de usuario de nuestra aplicación etc) Pero ¿ cómo nos conectaremos a la base de datos ?

JDBC: Java Database Connectivity (en español: Conectividad a bases de datos de Java), más conocida por sus siglas JDBC, es una API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java, independientemente del sistema operativo donde se ejecute o de la base de datos a la cual se accede, utilizando el dialecto SQL del modelo de base de datos que se utilice. (fuente wikipedia)

JDBC nos permitirá acceder a bases de datos (BD) desde Java. Con JDBC no es necesario escribir distintos programas para distintas BD, sino que un único programa sirve para acceder a BD de distinta naturaleza. Incluso, podemos acceder a más de una BD de distinta fuente (Oracle, Access, MySql, etc.) en la misma aplicación. Podemos pensar en JDBC como el puente entre una base de datos y nuestro programa Java.



La impedancia objeto-relacional

El desfase objeto-relacional, también conocido como impedancia objeto-relacional, consiste en la diferencia de aspectos que existen entre la programación orientada a objetos y la base de datos. Estos aspectos se puede presentar en cuestiones como:

Lenguaje de programación. El programador debe conocer el lenguaje de programación orientada a objetos (POO) y el lenguaje de acceso a datos.

Tipos de datos: en las bases de datos relacionales siempre hay restricciones en cuanto a los tipos de datos que se pueden usar, que suelen ser sencillos, mientras que la programación orientada a objetos utiliza tipos de datos más complejos.

Paradigma de programación. En el proceso de diseño y construcción del software se tiene que hacer una traducción del modelo orientado a objetos de clases al modelo Entidad-Relación (E/R) puesto que el primero maneja objetos y el segundo maneja tablas y tuplas o filas, lo que implica que se tengan que diseñar dos diagramas diferentes para el diseño de la aplicación.

El modelo relacional trata con relaciones y conjuntos debido a su naturaleza matemática. Sin embargo, el modelo de Programación Orientada a Objetos trata con objetos y las asociaciones entre ellos. Por esta razón, el problema entre estos dos modelos surge en el momento de querer persistir los objetos de negocio

La escritura (y de manera similar la lectura) mediante JDBC implica: abrir una conexión, crear una sentencia en SQL y copiar todos los valores de las propiedades de un objeto en la sentencia, ejecutarla y así almacenar el objeto. Esto es sencillo para un caso simple, pero trabajoso si el objeto posee muchas propiedades, o bien se necesita almacenar un objeto que a su vez posee una colección de otros elementos. Se necesita crear mucho más código, además del tedioso trabajo de creación de sentencias SQL.

Este problema es lo que denominábamos impedancia Objeto-Relacional, o sea, el conjunto de dificultades técnicas que surgen cuando una base de datos relacional se usa en conjunto con un programa escrito en con u lenguajes de Programación Orientada a Objetos.

Podemos poner como ejemplo de desfase objeto-relacional, un Equipo de fútbol, que tenga un atributo que sea una colección de objetos de la clase Jugador. Cada jugador tiene un atributo "teléfono". Al transformar éste caso a relacional se ocuparía más de una tabla para almacenar la información, implicando varias sentencias SQL y bastante código.

Procedimiento para conectar mediante JDBC

Para usar JDBC hay que seguir los siguientes pasos:

1.- Incluir el jar con el driver del SGBD. Para esto se puede ir a la página de la empresa (por ejemplo mysql da la posibilidad de descargar el driver jdbc) y luego incluirlo entre las librerías. Otra alternativa (mejor) es usar maven o gradle para gestionar las dependencias

Si lo hacemos con maven podemos buscar en google (nos dará un enlace a maven central) maven mysql-connector-java

Una vez en maven central tomamos la versión que queramos del driver. Por ejemplo veamos la 8.0.20:

```
<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.20</version>
</dependency>
```

2.- Registrar el driver. Para que se pueda usar el driver tenemos que "cargarlo"

El siguiente método se puede usar para cargar un driver mysql:

```
    private static  void cargarDriverMysql(){
      try {
        Class.forName("com.mysql.cj.jdbc.Driver");
      } catch(ClassNotFoundException ex) {
        System.err.println("no carga el driver");
        System.exit(1);
      }
    }
```

3.- Establecer la conexión.

Los controladores se identifican con un URL JDBC de la forma:

```
jdbc:subprotocolo:localizadorBD
```

• El subprotocolo indica el tipo de base de datos específico

• El localizador permite referenciar de forma única una BD

  • Host y opcionalmente puerto

  • Nombre de la base de datos

La conexión a la BD se hace con el método getConnection()

- public static Connection getConnection(String url)

- public static Connection getConnection(String url, String user, String password)

- public static Connection getConnection(String url, Properties info)

Tener en cuenta que todos pueden lanzar la excepción SQLException

Ejemplo:

```
try {
  conexion = DriverManager.getConnection(
     "jdbc:mysql://localhost/tienda","pruebas", "clavepruebas");
} catch (SQLException ex) {
// Tratar el error
}
```

En el código anterior le decimos que el subprotocolo(tipo de DB específica) es: mysql. En la parte de localizador aparece: **localhost** que indica que es en la propia máquina donde está la base de datos. Y la parte que dice: **tienda** especifica que la base de datos se llama: tienda. Finalmente

nos aparece el nombre del usuario que se conecta: **pruebas** y la password para ese usuario: **clavepruebas**

4.- Crear y ejecutar operaciones en la DB

Prepararemos la sentencia a ejecutar (createStatement() o prepareStatement()) la lanzamos (excuteUpdate(), executeQuery(), execute(),…) Para finalmente tomar los resultados devueltos (ResultSet) y cerraremos la conexión

Ejemplo:

Veamos la base de datos:

```
create database oficina;
use oficina;
SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
START TRANSACTION;
SET time_zone = "+00:00";
CREATE TABLE `lapices` (
  `idlapiz` int NOT NULL,
  `marca` char(30) NOT NULL,
  `numero` int DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

INSERT INTO `lapices` (`idlapiz`, `marca`, `numero`) VALUES
(1, 'staedtler', 2),
(2, 'alpino', 1),
(3, 'alpino', 3),
(4, 'staedtler', 1);
ALTER TABLE `lapices`
  ADD PRIMARY KEY (`idlapiz`);
COMMIT;
```

Una primera versión podría ser:

```
public class Main {
    private static  void cargarDriverMysql(){
      try {
        Class.forName("com.mysql.cj.jdbc.Driver");
      } catch(ClassNotFoundException ex) {
        System.err.println("no carga el driver");
        System.exit(1);
      }
    }

    public static void main(String[] args) {

        cargarDriverMysql();
        Connection conexion = null;
        try {
            conexion = DriverManager.getConnection(
                     "jdbc:mysql://localhost/oficina?serverTimezone=UTC","root",
"1q2w3e4r");

            Statement s = conexion.createStatement();
            String sql = "select * from lapices";
            ResultSet rs = s.executeQuery(sql);

            while(rs.next()){

                int id = rs.getInt("idlapiz");
                String marca = rs.getString("marca");
                int numero = rs.getInt("numero");

                System.out.println("Lápiz: " + id + " " + marca + " " + numero);

            }
            s.close();
            conexion.close();

        } catch (SQLException ex) { ex.printStackTrace(); }
    }
}
```

El código anterior primero registra el driver y luego intenta hacer una conexión:

conexion = DriverManager.getConnection(

"jdbc:mysql://localhost/oficina?serverTimezone=UTC","root", "1q2w3e4r")

la uri nos indica que nos conectamos a mysql, a la base de datos: oficina con el usuario: root y la password: 1q2w3e4r Todo eso lo hemos comentado antes. Pero ahora observamos el parámetro: serverTimezone=UTC

Bien, este parámetro tiene lugar porque las versiones del driver desde hace algún tiempo precisan conocer de la zona horaria. En nuestro caso siempre agregaremos el servertimezone con UTC establecido

Veamos las siguiente 3 líneas:

```
Statement s = conexion.createStatement();
String sql = "select * from lapices";
ResultSet rs = s.executeQuery(sql);
```

Hay varias formas de ejecutar una consulta contra la DDBB. Una es mediante: createStatement() Básicamente estamos lanzando una consulta en sql plano (sin compilar ni parametrizar) mediante el método executeQuery() y tomamos los resultados En un conjunto: ResultSet

ResultSet contiene todas las filas que satisfacen la consulta sql lanzada. Para ir avanzando al siguiente registro usamos: ResultSet.next()

¡¡**importante!!** resultset no está apuntando al primer registro una vez obtenido mediante executeQuery(). Para ubicarnos en el primer registro tenemos que ejecutar ResultSet.next()

Veamos ahora el recorrido de los registros:

```
while(rs.next()){
    int id = rs.getInt("idlapiz");
    String marca = rs.getString("marca");
    int numero = rs.getInt("numero");

    System.out.println("Lápiz: " + id + " " + marca + " " + numero);
}
s.close();
```

Observar que con rs.next() estamos avanzando de un registro a otro. Una vez ubicados en un registro, tomamos la información de cada campo mediante métodos como: rs.getInt(), rs.getString(),..

Los métodos rs.getInt(), rs.getString() reciben un nombre de campo en String o un entero que hace referencia al número de columna que corresponde a un campo en concreto.

Así en el ejemplo anterior sería coincidente:

```
String marca = rs.getString("marca");

String marca = rs.getString(2);
```

En ambas sentencias obtenemos el campo marca del registro, ya que marca es la segunda columna de la query lanzada

Finalmente cerramos la statement y después la conexión

Práctica 1: Crear la aplicación anterior en la que el usuario introduzca el nombre de la marca y se le muestren los registros correspondientes a la marca solicitada. Debe acceder convenientemente a la base de datos y hacer uso de una clase java: Lapiz que permita reconstruir un arraylist de lapices desde el resultset obtenido. Hacer una versión de texto y otra gráfica. Observar que podemos obtener todos los lapices en el resultset y únicamente poner en el arraylist los de la marca solicitada. También podemos ejecutar la query con una cláusula where

Práctica 2: Crear una versión de la aplicación anterior en la que tengamos una clase llamada: GestorLapices con un constructor que reciba el nombre de la base de datos (en el ejemplo que hemos realizado el nombre era: oficina) y que tenga un método llamado: obtenerLapicesPorMarca() que se le pase el nombre de la marca y devuelva un arraylist de lapices

En las actividades anteriores se ha nombrado que se puede lanzar la query a la DDBB con un where respecto a marca o también filtrar los datos recibidos en Java ¿ qué supone hacerlo en Java ?

Sabemos que las bases de datos son eficientes con las consultas y también sabemos que si nuestra base de datos está en red si solicitamos más datos de los que precisamos aumenta el tráfico innecesariamente. Normalmente evitaremos hacer el filtrado de datos en Java si podemos pedírselo directamente a la base de datos

Ahora estudiaremos como lanzar las diferentes consultas:

- Statement

- PreparedStatement

- CallableStatement

Statement

El objeto Statement al ser ejecutado, generan ResultSet que son una tabla de datos representantes de un conjuto de datos de la base de datos. Para crear un objeto statement debemos partir de un objeto Connection. En nuestro ejemplo, lo hacíamos mediante:

```
Statement s = conexion.createStatement();
```

Tenemos 3 formas de ejecutar una Statement:

execute

execute() devuelve true si se obtiene un objeto resultset. Mediante execute() podemos ejecutar sentencias sql de lectura: select, pero también de grabación: insert, o actualización: update

Veamos un ejemplo:

```
Statement stmt = con.createStatement();
//The query can be update query or can be select query
String query = "select * from emp";
boolean status = stmt.execute(query);
if(status){
    //query is a select query.
    ResultSet rs = stmt.getResultSet();
    while(rs.next()){
        System.out.println(rs.getString(1));
    }
    rs.close();
} else {
    //query can be update or any query apart from select query
    int count = stmt.getUpdateCount();
    System.out.println("Total records updated: "+count);
}
```

Observar que primero se analiza la variable status (que es lo que devuelve execute) para saber si era una consulta de selección o no lo era. Si es true, entonces es de seleccióny mediante: getResultSet() se obtiene y se recorre de forma parecida a executeQuery(), si devuelve false es que era de inserción, update, delete. En ese caso tiene un comportamiento similar que executeUpdate ya que tomamos el número de filas afectadas mediante: getUpdateCount()

executeQuery

executeQuery() únicamente admite consultas de selección y devuelve un resultset

Ya hemos visto al principio del tema un ejemplo con executeQuery():

```
            Statement s = conexion.createStatement();
            String sql = "select * from lapices";
            ResultSet rs = s.executeQuery(sql);
```

executeUpdate

Este método permite insert/update/delete (no permite select) Devuelve un entero que representa la cantidad de filas afectadas. Adicionalmente también permite DDL (crear tablas, borrar tablas, etc.)

```
s = c.createStatement();
    // Suprimir la tabla de prueba, si existe.
    try {
        s.executeUpdate("DROP TABLE MYLIBRARY.MYTABLE");
    } catch (SQLException e) {    }

    // Ejecutar una sentencia SQL que crea una tabla en la base de datos.
       s.executeUpdate("CREATE  TABLE  MYLIBRARY.MYTABLE  (NAME  VARCHAR(20),  ID
INTEGER)");

    // Ejecutar algunas sentencias SQL que insertan registros en la tabla.
     s.executeUpdate("INSERT INTO MYLIBRARY.MYTABLE (NAME, ID) VALUES ('RICH',
123)");
```

Práctica 3: DELETE, UPDATE de los lápices
El método: actualizar(Lapiz) recibe un objeto Lapiz y lo modifica en la base de datos con UPDATE
El método: borrar(int idlapiz) recibe el id del objeto Lapiz y borra el objeto en la base de datos.
Hacer también las correspondientes partes de interacción con el usuario para que pueda modificar o borrar un lápiz

Hemos dejado fuera de la práctica anterior el INSERT. Lo estamos haciendo porque si el id es autoincremental tenemos el problema de insertar el objeto como un registro en la base de datos sin saber que ID va a tener. Al final habría una descompensación entre lo que tenemos en Java y lo registrado en la base de datos ya que nuestro objeto Lápiz no tiene id pero en la base de datos sí lo tiene

Veamos como hacer un INSERT y tomar la información del id generado:

```
    public Lapiz saveLapiz(Lapiz m) throws SQLException{

        Lapiz resultado = null;
        Connection conexion = gcdb.getConnection();

             String sql2    = "INSERT INTO lapices ( marca, numero) VALUES
("+m.getMarca()+","+m.getNumero()+")";
        Statement st = conexion.createStatement();
        st.executeUpdate(sql2, Statement.RETURN_GENERATED_KEYS);
        ResultSet rs1 = st.getGeneratedKeys();
        while(rs1.next()){
            int id = rs1.getInt(1);
            resultado.setIdlapiz(id);
            resultado.setMarca(m.getMarca());
            resultado.setNumero(m.getNumero());
        }
        st.close();
        conexion.close();
        return resultado
    }
```

Como vemos aquí la clave está en pasarle a executeUpdate el dato de que queremos que nos devuelva las claves generadas: RETURN_GENERATED_KEYS y luego recogemos lo que nos haya dado en un resultset mediante: st.getGeneratedKeys()

Observar que estamos devolviendo un objeto Lapiz diferente del que hemos recibido. Desde el controlador, o el lugar al que se llame este método se tiene que tener en cuenta que hay que desechar el objeto Lapiz viejo y quedarse con el nuevo

Práctica 4: INSERT de los lápices

El método: save(Lapiz) recibe un objeto Lapiz y lo inserta en la base de datos con INSERT

Observar que el código del ejemplo precedente no se preocupa de si el objeto Lapiz ya existía en la base de datos. Modificar convenientemente para que primero mire si el Lapiz recibido ya tiene un ID Si tiene un id > 0 no hace la inserción y devuelve null (otra opción es lanzar una excepción)

Otro abordaje posible,que queda como actividad opcional adicional, es tratar de guardar con el id que nos ha dado el usuario (pero entonces se está adulterando la gestión con autoincremental) Si no existe en la base de datos y es mayor de cero: id > 0 entonces es que el usuario quiere guardar con ese ID específico el Lapiz.

Las Statement tienen el problema del SQL Injection esto es: Tenemos una página que le pide al alumno 3 que modifique su fecha de nacimiento. La query sería parecida a:

```
            Scanner sc = new Scanner(System.in);
            int idalumno = 3;
            Connection conexion = gc.getConnection();
            try {
                  Statement s = conexion.createStatement();
s.execute("UPDATE alumnos SET fechanacimiento=" + sc.nextLine() +
" where idalumno="+idalumno);
            } catch (SQLException e) {
                  e.printStackTrace();
            }
```

Ahora bien. Si es un usuario malintencionado puede hacer un gran destrozo si lo que envía por teclado es:

```
"2000-01-01 where idalumno=3; delete from matriculas; update alumnos set
nombre='sqlinjection' "
```

Lo anterior se denomina SQL injection. Veamos la definición de wikipedia:

SQL Injection

“Inyección SQL es un método de infiltración de código intruso .

El origen de la vulnerabilidad radica en la incorrecta comprobación o filtrado de las variables utilizadas en un programa que contiene, o bien genera, código SQL.

Se dice que existe o se produjo una inyección SQL cuando, de alguna manera, se inserta o "inyecta" código SQL invasor dentro del código SQL programado, a fin de alterar el funcionamiento normal del programa y lograr así que se ejecute la porción de código "invasor" incrustado, en la base de datos”

Si observamos el ejemplo anterior, vemos que el desarrollador estaba solicitando el contenido de una **variable** (fechadenacimiento) Que es justo lo que se indica en la definición de wikipedia:

```
UPDATE alumnos SET fechanacimiento="
```

Y el atacante en lugar de darle únicamente un valor, agrega, “inyecta” código SQL (observar el concepto de que es código, no un valor, porque ese es el matiz para tratar de controlar la vulnerabilidad como veremos en la siguiente sección)

Para lidiar con este tipo de ataques tenemos opciones como las **PreparedStatement** que vemos en la siguiente sección

PreparedStatement

Lo primero y más importante que debemos entender es que estas sentencias admiten que les pasemos parámetros (observar el interrogante: “?” que es el lugar donde va el parámetro)

```
"SELECT * FROM asignaturas WHERE idasignatura =  ? "
```

Estas sentencias pasan de esta forma (literalmente) a nuestro SGBD . Allí se compila la sentencia. Después se inserta el valor (únicamente admite valores para la parte de interrogantes, no admite nada más) que se le pase como parámetro, y finalmente ejecuta

De lo anterior podemos concluir dos ventajas: La principal que podemos combatir mejor contra el sql injection ya que lo que envíe el usuario y que sea permitido serán únicamente valores que se reemplazarán allí donde haya interrogantes. La segunda es que al quedar compilada en el SGBD puede recurrirse a ella nuevamente para ejecuciones posteriores más rápidas

Veamos un ejemplo:

```
String query = "INSERT INTO asignaturas(nombre,curso) VALUES(?,?) " ;

// cn es de tipo Connection

PreparedStatement ps;
ps = cn.prepareStatement(query,PreparedStatement.RETURN_GENERATED_KEYS);

ps.setString(1, "BAE");
ps.setString(2, "1º DAM");

ps.executeUpdate();
```

Observar que le pasamos los parámetros. El primer interrogante que aparezca será el parámetro 1, el segundo interrogante será el parámetro 2 etc.

Así en el código anterior, BAE se está poniendo como nombre y 1ºDAM como curso

Por lo demás es muy parecido a Statement, existen executeUpdate, executeQuery,…

Cuidado al hacer uso de: executeQuery(), executeUpdate() porque hay dos versiones: una recibe una String (que realmente no es propio de la clase PreparedStatement sino de su padre: Statement) y la otra versión no lleva String, que es la que realmente queremos usar.

El siguiente código es correcto:

```
String query = "SELECT * FROM asignaturas WHERE idasignatura =  ? " ;
PreparedStatement ps = cn.prepareStatement(query);
ps.setInt(1, (int)id);

     ResultSet rs = ps.executeQuery();
```

Pero si en lugar de la última línea hubiéramos puesto:

```
ResultSet rs = ps.executeQuery(query);
```

entonces habría fallado. Ya que se desencadenaría la versión del método executeQuery() perteneciente a la clase Statement en lugar de la de PreparedStatement

Un ejemplo que refleja como buscar un registro en la tabla asignaturas:

```
int id = 1;
String query = "SELECT * FROM asignaturas WHERE idasignatura =  ? " ;
try (
      Connection cn = gc.getConnection();
    PreparedStatement ps = cn.prepareStatement(query);
 ){
  ps.setInt(1, id);

      ResultSet rs = ps.executeQuery();
      if(rs.next()) {
            asignatura = new Asignatura();
            asignatura.setIdasignatura(id);
            asignatura.setNombre(rs.getString("nombre"));
            asignatura.setCurso(rs.getString("curso"));
      }
} catch (SQLException e) { e.printStackTrace();}
```

Práctica 5: Repetir/modificar las prácticas anteriores que usaban Statement por PrepareStatement. Comprobar que si se le pasa una String SQL tanto a executeQuery como a executeUpdate tiene un comportamiento no deseado. ¿ Genera algún tipo de error ?

Procedimientos Almacenados

Un procedimiento almacenado es un procedimiento o subprograma que está almacenado en la base de datos.

Un procedimiento almacenado MySQL no es más que una porción de código que puedes guardar y reutilizar. Es útil cuando repites la misma tarea repetidas veces, siendo un buen método para encapsular el código. Al igual que ocurre con las funciones, también puede aceptar datos como parámetros, de modo que actúa en base a éstos.

Muchos sistemas gestores de bases de datos los soportan, por ejemplo: MySQL, Oracle, etc.

Tipos:

- Procedimientos almacenados.
- Funciones, las cuales devuelven un valor que se puede emplear en otras sentencias SQL.

Al reducir la carga en las capas superiores de al aplicación, se reduce el tráfico de red y, si el uso de los procedimientos almacenados es el correcto, puede mejorar el rendimiento.

Al encapsular las operaciones en el propio código SQL, nos aseguramos de que el acceso a los datos es consistente entre todas las aplicaciones que hagan uso de ellos.

En cuanto a seguridad, es posible limitar los permisos de acceso de usuario a los procedimientos almacenados y no a las tablas directamente. Des este modo evitamos problemas derivados de una aplicación mal programada que haga un mal uso de las tablas.

Un procedimiento almacenado típico tiene:

Un nombre.

Una lista de parámetros.

Unas sentencias SQL.

Veamos un ejemplo de sentencia para crear un procedimiento almacenado sencillo para MySQL, aunque sería similar en otros sistemas gestores:

```
DELIMITER $$
DROP PROCEDURE IF EXISTS alumnossubnombre$$
CREATE PROCEDURE alumnossubnombre(IN subnombre VARCHAR(50))
BEGIN
  SELECT *
  FROM alumnos
  WHERE alumnos.nombre LIKE  subnombre;
END
$$
```

Observar que el delimitador de sentencias habitual es: “;” Se cambia para que el sistema entienda que es una única sentencia (la creación de un procedimiento almacenado) ya que dentro las acciones que se realizan pueden ser varias y precisar de finalización. En ese caso el punto y coma que se ponga no lo interpretará como final del procedimiento almacenados

En el anterior ejemplo se crea un procedimiento que devuelve el listado de productos para una variable que toma de entrada. La forma de llamar al procedimiento es:

```
Connection cn = gc.getConnection();
try {

      CallableStatement pc = cn.prepareCall("call alumnossubnombre(?)");
      pc.setString(1,"%a%");
      ResultSet rs = pc.executeQuery();
      while(rs.next()) {
            System.out.println(rs.getString("nombre"));
      }

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

La consulta anterior devuelve todos los alumnos que tienen una letra a en su nombre. Observar que hemos puesto el tanto por ciento: **"%"** como un parámetro. Este es el comportamiento habitual que haremos. NO pondremos dentro de la query el: "%" en las prepareStatement ni en las prepareCall sino que se lo pasaremos al enviar el parámetro

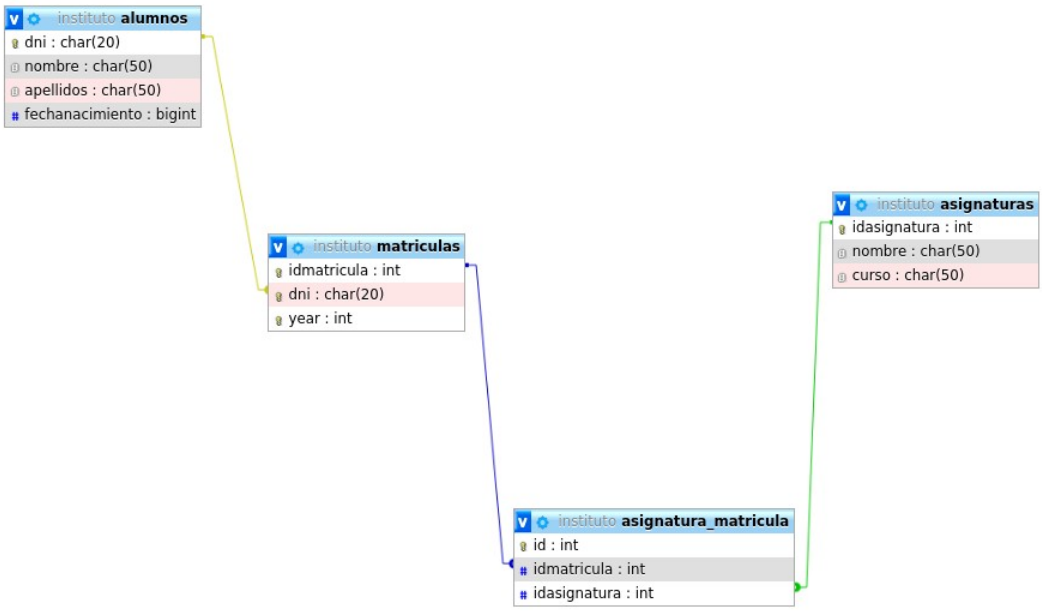
Por lo demás tiene un comportamiento similar a lo que esperamos en prepareStatement

Práctica 6: Crear un procedimiento almacenado que obtenga lápices de la tabla lapices recibiendo un parámetro (por ejemplo el número del lápiz) y convertirlo a un arraylist de lápices Java

Bien, ya tenemos las herramientas básicas para empezar una aplicación que nos guíe durante el aprendizaje. El resto de conceptos los iremos introduciendo con esa aplicación

Descripción de la aplicación a desarrollar

Tomemos como referencia el siguiente diagrama (representa una base de datos que gestiona las matrículas en un Instituto)



Los alumnos se identifican por DNI. Intencionadamente se ha registrado fechanacimiento como bigint, no como Date. Esto es para prepararnos para situaciones como en SQLite que no se dispone del tipo de datos Date

Los alumnos pueden matricularse hasta una vez al año en el Instituto

alumnos – 1:N –> matriculas

matriculas(idmatricula,dni,year) Podríamos tener como clave primaria de matriculas el DNI del alumnos matriculado y el año. Sin embargo se ha escogido crear un id autoincremental y se ha establecido que no sea posible dos registros con mismo par: dni-año (restricción Unique)

Si bien, pareciera lógico disponer de una tabla con los Cursos disponibles se ha saltado por sencillez. En este caso, la información de curso que aparece está registrada directamente en asignaturas

asignaturas(idasignatura,nombre,curso) Asignaturas es una tabla con id autoincremental y donde no se puede repetir el par: nombre-curso (restricción Unique)

La relación entre matrículas y asignaturas es de muchos a muchos:

matriculas — N:M — asignaturas

Para ponerlo en relacional agregamos una nueva tabla: matricula_asignatura. Otra vez lo más interesante sería disponer de una primary-key compuesta de las dos claves foráneas. Sin embargo como en otras tablas hay un ID autoincremental y una resticción Unique para las dos claves foráneas

Realizaremos una aplicación web que permita gestionar:

Pantalla inicial (es únicamente un archivo html. No es una página dinámica)

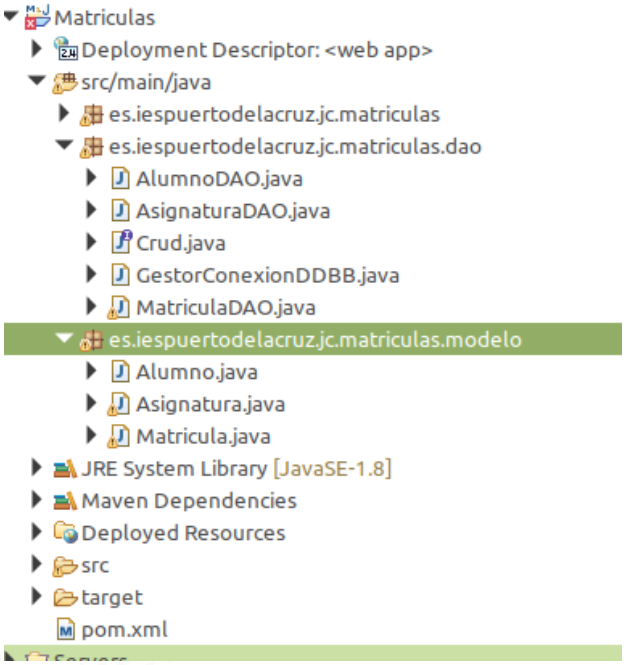
La página que gestiona los alumnos:

La página que gestiona las matrículas:

La pantalla de gestión de las asignaturas se deja a decisión creativa del alumno

Circunstancias técnicas a tener en cuenta con la aplicación

Se espera una separación en paquetes amplia y coherente. Se propone algo similar a:



Observar que hay una capa DAO (explicación más adelante). Tenemos un paquete también para las entities (paquete modelo en este caso) y un paquete matriculas que es donde vamos a poner nuestros Servlets

Una de las partes que lleva tiempo de decisión es que hacer cuando una clase (por ejemplo Alumno) está en una relación: 1:N con otra clase en DDBB (por ejemplo Matrícula) y si cuando se busca un Alumno se debe traer la información de la lista de objetos del lado N de la relación (en este caso sería Matrícula)

Observar que un alumno realiza varias matrículas (una por año como mucho) y eso hace que la clase Alumno pueda ser conveniente que tenga una lista de matriculas:

```
Alumno alumno42131234V = alumnoDAO.findById("42131234V");
```

La anterior línea es lo que se ejecuta para buscar en la base de datos a un alumno con el dni: “42131234V” y debe devolver un Alumno ¿ Debe de tener la lista de matrículas incluida ?

Vamos a ver con más detalle el problema: (no preocuparse ahora de las anotaciones json)

```
public class Alumno {

	String dni;
	String nombre;
	String apellidos;
	@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd")
	Date fechanacimiento;
	@JsonIgnore
	ArrayList<Matricula> matriculas;
}
```

Es importante que nos fijemos que traer los objetos de tipo Matrícula cuando solicitamos un Alumno implica hacer más consultas a la base de datos. Además tener en cuenta que a su vez la clase Matrícula tiene implicaciones en otras tablas:

```
public class Matricula {

	int idmatricula;
	Alumno alumno;

	@JsonProperty("año")
	int year;
	ArrayList<Asignatura> asignaturas;
}
```

Sabemos que las matrículas se hacen con un grupo de asignaturas. Pero en nuestro caso relacional esto corresponde con dos tablas más: asignatura_matricula, asignaturas

El caso es que si se resuelven TODAS las implicaciones de objetos con el modelo relacional pueden ser muchas consultas SQL y repercutir en el rendimiento de la aplicación y la saturación del SGBD

Lo cierto es que no hay una regla general que se pueda aplicar, debe adaptarse al caso particular que se trate.

A modo de idea, los frameworks ORM el comportamiento que tienen por defecto es:

- Las relaciones 1:N no se resuelven salvo que sea solicitado por el usuario. Esto es, en el ejemplo anterior, se reconstruye un Alumno con los datos de la base de datos de la tabla alumnos sin hacer la consulta relacionada a la tabla matriculas (a esto le dicen comportamiento Lazy)

- Las relaciones N:1 sin embargo SÍ se resuelven. Esto es, si creas un objeto: Matricula tomando la información de la base de datos de la tabla matriculas, también haces la consulta pertinente para reconstruir el Alumno que aparece como atributo en el objeto Matricula

Así pues, tenemos que ver el caso en concreto y las necesidades que pudiera tener nuestra aplicación. En nuestro caso vamos a considerar las siguiente implicaciones:

Respecto a Matricula:

- NO vamos a guardar en cascada la relación entre Matricula y Alumno. Esto es, si se crea un objeto Matricula que tenga incorporado un nuevo objeto Alumno, NO se hace el INSERT en la tabla Alumno. De hecho, lo que haremos es que se rechace grabar la nueva matrícula en la base de datos al no existir el alumno previamente. Sin embargo si el Alumno ya existe en la base de datos SÍ vamos grabar en cascada la Matricula y la Inserción que corresponde en la tabla: asignatura_matricula. Observar que es lógico que cuando alguien se apunte con una matrícula también se detalle a la vez las asignaturas que va a cursar.

Éste va a ser en general el comportamiento que aplicaremos con la relación Matricula y Asignatura Tanto para grabar una nueva matrícula (insert), modificar la matrícula (update) o borrar la matrícula (delete) Cada una de esas acciones afectará a las dos tablas: matricula, asignatura_matricula

- Si obtenemos un objeto Matricula ya almacenado en la base de datos y lo vamos a recuperar (por ejemplo, método: findByIdmatricula()) también vamos a crear el objeto Alumno (los foreign key sí los resolvemos aunque implique realizar una consulta adicional a la base de datos. En estos casos vamos a comportarnos con el mismo comportamiento por defecto que tienen los ORM)

Respecto a Alumno:

- Guardar un objeto Alumno, únicamente implicará modificar la tabla alumnos (hacer un INSERT) no afectará a ninguna otra tabla

- Recuperar un objeto Alumno (por ejemplo, mediante: findByDNI()) no implicará a otras tablas (se hará un SELECT a la tabla alumnos). De igual forma será si se recupera un grupo de alumnos (por ejemplo, todos los que tengan el mismo nombre)

- Modificar un alumno (UPDATE) únicamente implicará a la tabla alumnos a ninguna otra tabla

- Borrar un alumno no será posible si no se ha borrado previamente las matrículas que ha realizado (esto es, no se admite el borrado en cascada: borrar un alumno, borraría también las matrículas que ha hecho). Y cuando se haga el borrado únicamente se hará un delete en la tabla alumnos, no habrá otras consultas SQL

- Se puede recuperar un Alumno tanto por DNI, como recuperar todos los alumnos (que será únicamente consultas a la tabla alumnos, no implicará a otras tablas)

Especial: Vamos a habilitar una consulta compleja que permite recuperar el conjunto de alumnos para una asignatura un año en concreto (esto es: asignatura.idasignatura, matricula.year)

Respecto a Asignatura:

- Nada especial. Todo se ejecuta únicamente contra la tabla asignaturas. Tanto si se modifica una asignatura como si se crea nueva o si se trata de recuperar (select) Por otro lado tampoco se permitirá el borrado en cascada, así que NO se va a poder borrar una asignatura si tiene relación con alguna matrícula (tiene primero que modificarse las matrículas implicadas para que no aparezca la asignatura en ellas antes de poder borrar la asignatura)

DAO

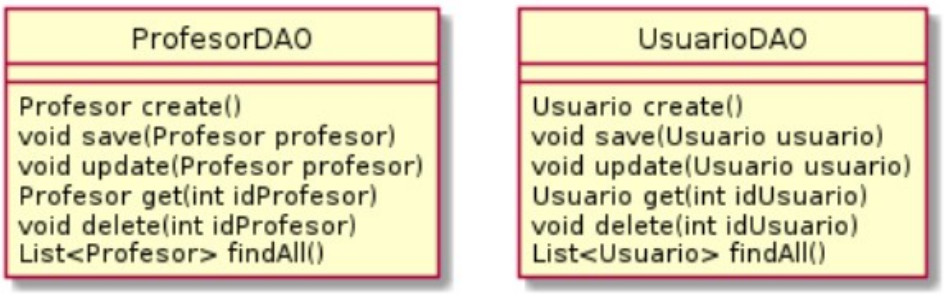
Según wikipedia:

En software de computadores, un objeto de acceso a datos (en inglés, data access object, abreviado DAO) es un componente de software que suministra una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos, tales como una Base de datos o un archivo.

Los Objetos de Acceso a Datos son un Patrón de los subordinados de Diseño Core J2EE y considerados una buena práctica. La ventaja de usar objetos de acceso a datos es que cualquier objeto de negocio (aquel que contiene detalles específicos de operación o aplicación) no requiere conocimiento directo del destino final de la información que manipula.

Los Objetos de Acceso a Datos pueden usarse en Java para aislar a una aplicación de la tecnología de persistencia Java subyacente (API de Persistencia Java), la cual podría ser JDBC, JDO, Enterprise JavaBeans, TopLink, EclipseLink, Hibernate, iBATIS, o cualquier otra tecnología de persistencia

El modelo DAO podría incluso tomar por cada clase que necesitamos crear una clase equivalente DAO. Por ejemplo, supongamos que nuestra aplicación usará dos clases que queremos persistir. Una se llama Profesor y la otra Usuario entonces el patrón DAO nos podría dar:



Si nos fijamos, entonces con la línea:

```
profesorDAO.update( profesor )
```

estaríamos en ese método update() tocando la base de datos para actualizar la información del objeto: profesor que tenemos en memoria

Se propone pues, crear un objeto DAO por cada una de las tablas/objetos necesarias. En nuestra aplicación precisaríamos como mínimo:

AlumnoDAO, AsignaturaDAO, MatriculaDAO

Se propone que TODOS ellos implementen el siguiente interfaz:

```
public interface Crud<T,E> {

	T save(T dao);

	T findById(E id);

	boolean update(T dao);

	boolean delete(E id);

	ArrayList<T> findAll();
}
```

Donde el tipo genérico: T hace referencia a la clase que queremos trabajar (por ejemplo en AlumnoDAO sería Alumno) y el tipo: E hace referencia al ID que usamos para esa clase (el ID de un Alumno es un dni que es de tipo String)

Veamos como quedaría el prototipo de AlumnoDAO entonces:

```
public class AlumnoDAO implements Crud<Alumno,String>{

	GestorConexionDDBB gc;
	public AlumnoDAO(GestorConexionDDBB gc) {
		this.gc = gc;
	}

	@Override
	public Alumno save(Alumno dao) {
		String query="INSERT INTO alumnos(dni,nombre ...";
		try (
			Connection cn = gc.getConnection();
		    PreparedStatement ps = cn.prepareStatement(query);
		 ){
			//aquí el código que corresponda
		}
		return null;
	}

	@Override
	public Alumno findById(String id) {
		return null;
	}

	@Override
	public boolean update(Alumno dao) {
		return false;
	}

	@Override
	public boolean delete(String id) {
		return false;
	}

	@Override
	public ArrayList<Alumno> findAll() {
		return null;
	}
}
```

Observar que todos los override son exactamente lo que viene del interfaz

Por otro lado aparece un constructor que recibe `GestorConexionDDBB` eso es debido a que hemos creado una clase que maneja la conexión. Vamos a verla

```
public class GestorConexionDDBB {

    String jdbcUrl;
    String usuario;
    String clave;

    public Connection getConnection() {

        Connection con=null;
        try {
            con = DriverManager.getConnection(jdbcUrl, usuario, clave);
        } catch (SQLException ex) {
            System.exit(1);
        }
        return con;
    }

    public GestorConexionDDBB(String ddbb,String nombreUsuario, String password)
{
        jdbcUrl = "jdbc:mysql://localhost/"+ddbb+"?serverTimezone=UTC";
        usuario = nombreUsuario;
        clave = password;
        cargarDriverMysql();
    }

    private static  void cargarDriverMysql(){
      try {
        Class.forName("com.mysql.cj.jdbc.Driver");
      } catch(ClassNotFoundException ex) {
        System.err.println("no carga el driver");
        System.exit(1);
      }
    }
}
```

Como vemos, lo único que hace es establecer y fijarnos todo para que lo único que tengamos que hacer cuando queremos una nueva conexión sea: GestorConexionDDBB.getConnection()

Transacciones

Pongámonos en el siguiente caso de nuestra aplicación. Tenemos que guardar una nueva matrícula. Como dijimos ya, parece lógico que en el momento de la matrícula se establezcan cuáles son las asignaturas que se van a cursar. Eso significa que tenemos que hacer un INSERT en la tabla matriculas y tantos INSERT como asignaturas se vayan a cursar en la tabla: asignatura_matricula

¿ Qué ocurre si insertamos correctamente el registro en la tabla matrícula y sin embargo falla la inserción en la tabla asignatura_matricula ? Si lo dejáramos así sería que el alumno estará matriculado pero no tiene asignaturas que cursar. Eso no tiene sentido. Debiéramos entender toda la acción como una única y atómica, de tal forma que si no se consiguen insertar todos los registros que corresponde para dar de alta la matrícula, entonces que no se tenga en cuenta ninguno

Pues precisamente eso es lo que es una **transacción**: Agrupa un conjunto de operaciones como una única operación. Si alguna de ellas falla entonces se deshacen las otras

Las bases de datos que soportan transacciones son mucho más seguras y fáciles de recuperar si se produce algún fallo en el servidor que almacena la base de datos, ya que las consultas se ejecutan o no en su totalidad.

Al ejecutar una transacción, el motor de base de datos garantiza: atomicidad, consistencia, aislamiento y durabilidad (ACID) de la transacción (o conjunto de comandos) que se utilice.

El ejemplo típico que se pone para hacer más clara la necesidad de transacciones en algunos casos es el de una transacción bancaria. Por ejemplo, si una cantidad de dinero es transferida de la cuenta de un cliente, pongamos: clienteEmisor a otro cliente, pongamos: clienteReceptor, hace falta dos operaciones en DDBB:

```
UPDATE cuentas SET saldo = saldo - cantidad WHERE cliente = "clienteEmisor";

UPDATE cuentas SET saldo = saldo + cantidad WHERE cliente = "clienteReceptor";
```

Si la segunda operación no se pudiera realizar pero la primera sí ocurriría que el dinero "desaparecería" ya que habría sido quitado de la cuenta del emisor pero no agregado a la del receptor

Commit, Rollback

Una transacción tiene dos finales posibles, COMMIT o ROLLBACK. Si se finaliza correctamente y sin problemas se hará con COMMIT, con lo que los cambios se realizan en la base de datos, y si por alguna razón hay un fallo, se deshacen los cambios efectuados hasta ese momento, con la ejecución de ROLLBACK.

Habitualmente en los SGBD, e**n una conexión trabajamos en modo autocommit con valor true**. Eso significa que cada consulta es una transacción en la base de datos.

Por tanto, si queremos definir una transacción de varias operaciones, estableceremos el modo autocommit a false con el método setAutoCommit de la clase Connection.

En modo **no autocommit** las transacciones quedan definidas por las ejecuciones de los métodos commit y rollback. Una transacción abarca desde el último commit o rollback hasta el siguiente commit.

Ej:

```
BEGIN

...

SET AUTOCOMMIT OFF
update cuenta set saldo=saldo + 250 where dni="12345678-L";
update cuenta set saldo=saldo - 250 where dni="89009999-L";
COMMIT;
...

EXCEPTION
   WHEN OTHERS THEN
      ROLLBACK ;
END;
```

La anterior transacción es código de un procedimiento anónimo que se ejecuta DENTRO DE LA BASE DE DATOS. Así que ahora sabemos que las transacciones son nativas. Veamos como enviamos las instrucciones desde Java con el ejercicio de matrículas:

Cuando modificamos una Matricula, implica también a las asignaturas de la matrícula en la tabla asignatura_matricula

Así se necesitan estas 3 query:

```
String queryDelete = "DELETE FROM asignatura_matricula  WHERE idmatricula =  ? "
;
String queryUpdate = "UPDATE matriculas SET dni= ?, year= ?  WHERE idmatricula =
? " ;


String queryInsert = "INSERT INTO asignatura_matricula(idasignatura,idmatricula)
VALUES(?,?) " ;
```

Primero borramos (delete) toda asociación de las asignaturas con la matrícula (las filas de asignatura_matricula son la clave aquí)

Después modificamos (update) la tabla matriculas para poner los posibles cambios introducidos (el dni del alumno, el año de la matricula)

finalmente agregamos (insert) en la tabla asignatura_matricula las nuevas asociaciones que nos hayan dado

Lo lógico es que **ninguna modificación tenga lugar si alguna de las sentencias tenga lugar.** Así que **hay que cubrir con una transacción**

Veamos un trozo del código necesario:

```
String queryDelete = "DELETE FROM asignatura_matricula  WHERE idmatricula =  ? " ;
String queryUpdate = "UPDATE matriculas SET dni= ?, year= ?  WHERE idmatricula =  ? " ;


String queryInsert = "INSERT INTO asignatura_matricula(idasignatura,idmatricula)
VALUES(?,?) " ;
try (
      Connection cn = gc.getConnection();

      PreparedStatement psDelete = cn.prepareStatement(queryDelete);
      PreparedStatement psUpdate = cn.prepareStatement(queryUpdate);
      PreparedStatement psInsert = cn.prepareStatement(queryInsert);
){
      cn.setAutoCommit(false);

      psDelete.setInt(1, dao.getIdmatricula());
      int respuesta = psDelete.executeUpdate();

      psUpdate.setString(1, dao.getAlumno().getDni());
      psUpdate.setInt(2, dao.getYear());
      psUpdate.setInt(3, dao.getIdmatricula());


      respuesta = psUpdate.executeUpdate();

      ok = respuesta > 0;  //entendemos que debe haber alguna fila borrada si ok

      if(ok ) {

            //… .. .. ←Aquí el resto del código
            //si todo sale bien, al final hay un commit
            cn.commit();
            cn.setAutoCommit(true);

       }else{   //aquí salió mal así que “rompemos la transacción”
          cn.rollback();
          cn.setAutoCommit(true);
       }
}
```

Vemos que hemos usado un try-with-resources en el código anterior. Y es que estamos siempre expuestos al: SQLException

Excepciones.

Las conexiones con una base de datos consumen muchos recursos en el sistema gestor, y por lo tanto en el sistema informático en general. Por ello, conviene cerrarlas con el método `close` siempre que vayan a dejar de ser utilizadas, en lugar de esperar a que el garbage collector de Java las elimine.

También conviene cerrar las consultas (`Statement` y `PreparedStatement`) y los resultados (`ResultSet`) para liberar los recursos.

Una excepción es una situación que no se puede resolver y que provoca la detención del programa de manera abrupta. Se produce por una condición de error en tiempo de ejecución.

En base de datos es especialmente interesante el uso de try with resourses porque el proceso de manejo y cierre de excepciones puede ser un poco tedioso

Pongamos como ejemplo una situación que se da en la aplicación de matrículas. En el código que vimos antes de modificación de una matrícula hay que abrir y ejecutar múltiples preparedStatement y todas ellas pueden ser susceptibles de lanzar excepción. Habría que cubrirse con un try y finalmente cerrar todo correctamente. Con try-with-resources se nos facilita bastante ya que tienen implementado el autocierre.

Uso de JSON en las clases del modelo (las Entities)

Para finalizar vamos a documentar un poco las instrucciones pertinentes para la conversión a JSON y vicerversa

Ya hemos hablado de la librería Jackson. Volvemos a poner la dependencia Maven:

Una dependencia maven válida podría ser:

```
<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-core -->
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
   <version>2.12.2</version>
</dependency>
```

Únicamente veremos la transformación a String (con formato JSON) y la construcción de objetos Java desde una string en formato JSON

El objetivo es:

- Paso de objeto Java a String JSON para luego mostrar al usuario en ese formato

Si tenemos un objeto Alumno y lo queremos pasar a JSON:

```
ObjectMapper mapper = new ObjectMapper();
String strAlu = mapper
            .writerWithDefaultPrettyPrinter()
            .writeValueAsString(alumno);
```

El problema está en las referencias cruzadas… Por ejemplo, un alumno tiene matrículas y luego cada matrícula tiene referencia a ese alumno. Así que cuando vaya a escribir el JSON entra en un bucle infinito

@JsonIgnore

Podemos usar: @JsonIgnore si no queremos que un atributo participe de la conversión a JSON Por ejemplo, podemos evitar las referencias cruzadas si impedimos que se escriba el arraylist de matriculas al estar escribiendo un Alumno

```
class Alumno{
	@JsonIgnore
	ArrayList<Matricula> matriculas;
}
```

@JsonFormat

Las fechas querremos que aparezcan en un formato específico. Para conseguir ese formato podemos usar JsonFormat. Por ejemplo, En la clase Alumno al nombrar la fecha de nacimiento:

```
	@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "yyyy-MM-dd")
	Date fechanacimiento;
```

@JsonProperty

En ocasiones queremos que el JSON tenga un nombre en los atributos generados distinto del que declaramos en la clase Java. Por ejemplo, nuestros atributos Java en la clase Matricula para hablar del año de la matrícula lo llamamos: year sin embargo para mostrárselo al usuario se lo damos en castellano: año Eso lo conseguimos con JsonProperty

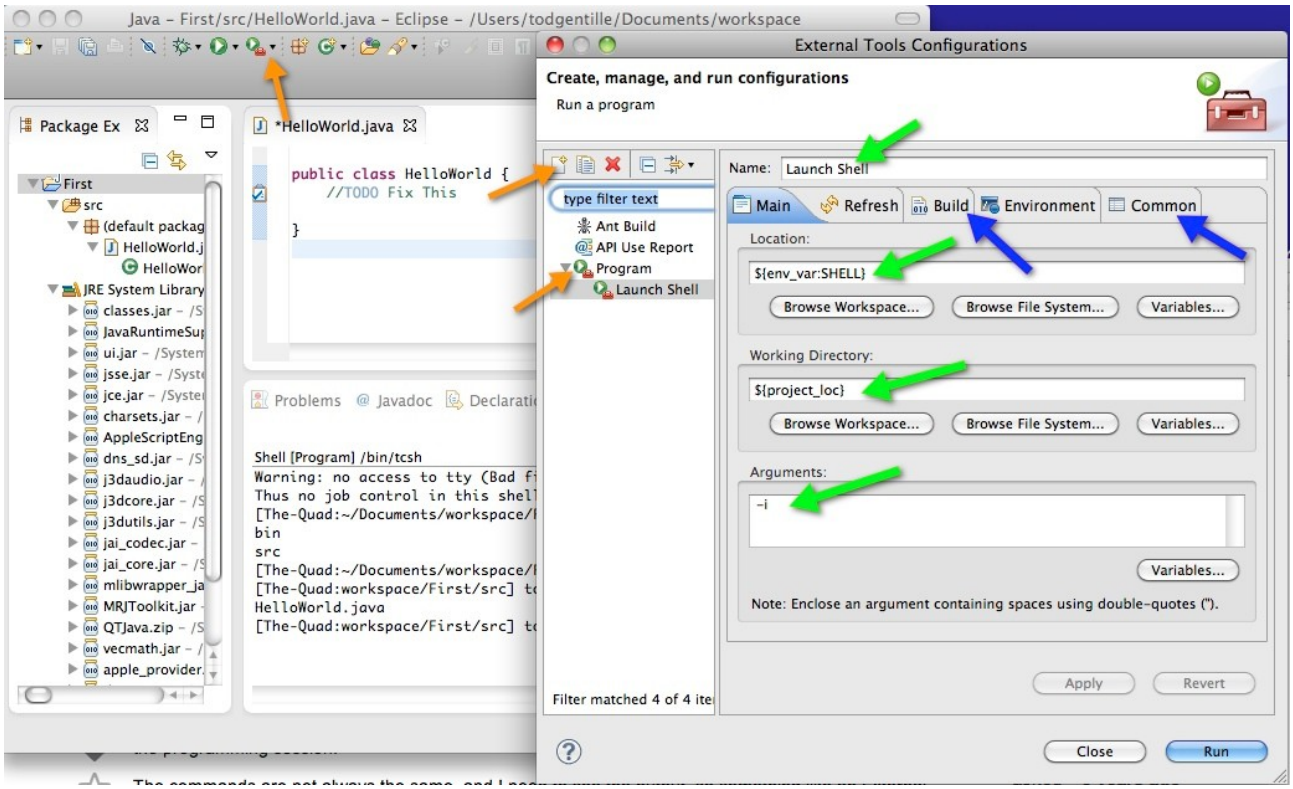
```
public class Matricula {

	@JsonProperty("año")
	int year;
}
```

Anexo Agregar shell a Eclipse

Hay shell integrado que se puede habilitar con: CTRL + ALT + SHIFT + T

para versiones más antiguas de eclipse el procedimiento es un poco más tedioso:



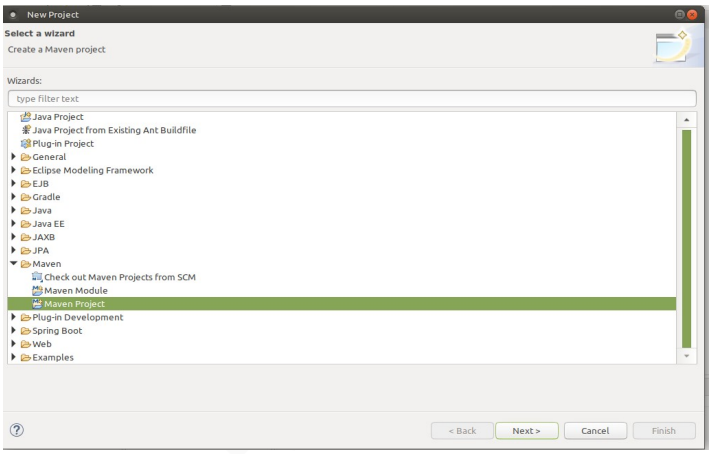
Lo encontramos documentado en: https://qastack.mx/programming/1562600/is-there-an-eclipse-plugin-to-run-system-shell-in-the-console

Anexo: Creando un proyecto web java que accede a bases de datos con Eclipse

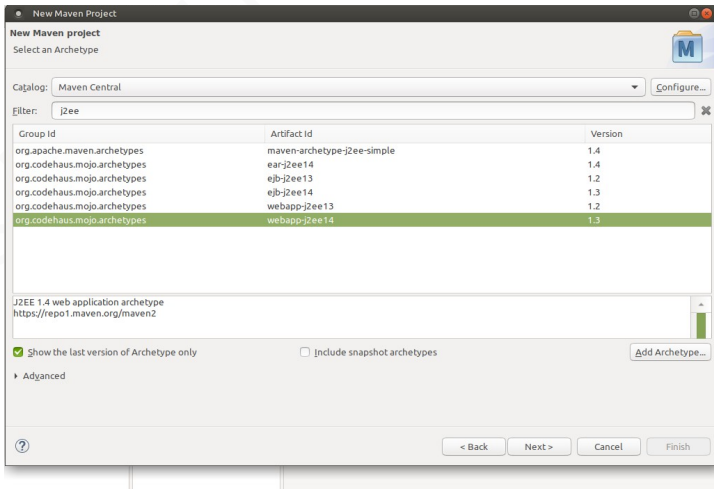
Creando proyecto web con eclipse para Alumnos Asignaturas

1)

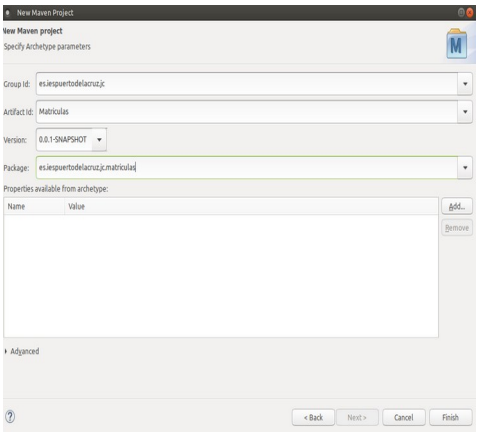
seleccionamos Maven project



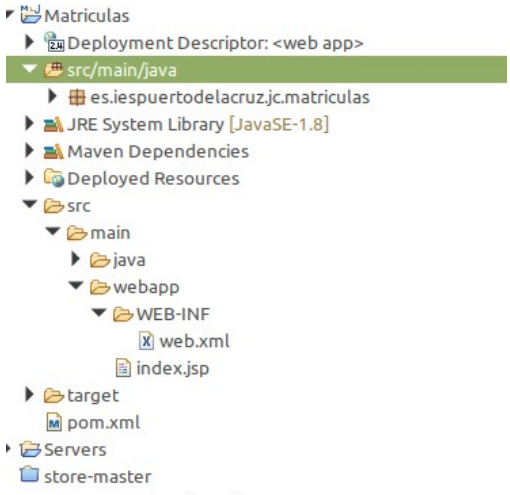
2) En la parte de archetype podemos elegir varias opciones que nos funcionarían. En esta ocasión usamos: webapp-j2ee14



3) Ponemos los datos de nuestro proyecto maven. Observar que se ha pasado a minúscula el nombre del paquete ya que eclipse copia literalmente lo que ponemos en artifact id



La estructura del proyecto que nos ha generado:



Observar que nos ha generado:

- index.jsp (los ficheros .jsp son las vistas y aquí tenemos la página de inicio del proyecto)

- WEB-INF/web.xml La carpeta WEB-INF es muy especial. Es una carpeta que no está pública para nuestros clientes. Lo que pongamos en esa carpeta no será visible sino únicamente para la propia aplicación. Ahí dentro tenemos web.xml que es el fichero de configuración Es en ese fichero donde se dice cuál es la página inicial, se dice donde y cuáles son nuestros controladores de nuestra aplicación (los Servlet), si hay algún filtro a los accesos de usuario etc.

- El fichero pom.xml donde se ponen las declaraciones maven

5) La archetype de maven que elegimos para aplicación web nos pone una versión muy antigua de java. Como mínimo pasamos a la 8: 1.8 en source y target del pom

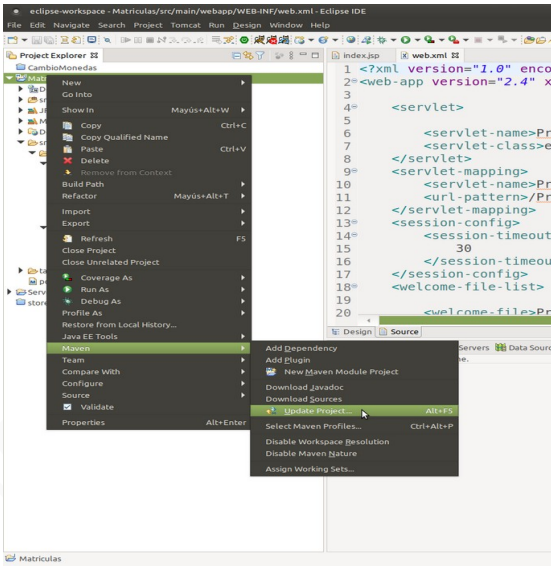
Hay que cambiar el pom.xml

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.0.2</version>
      <configuration>
        <source>1.8</source>
```

```
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

5) Debemos actualizar el proyecto maven para que tome la referencia de 1.8. Para ello:

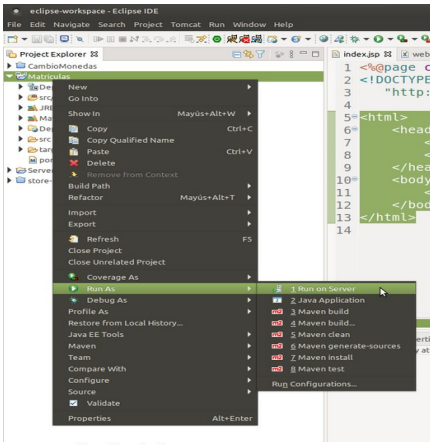
Botón derecho → Maven → update project



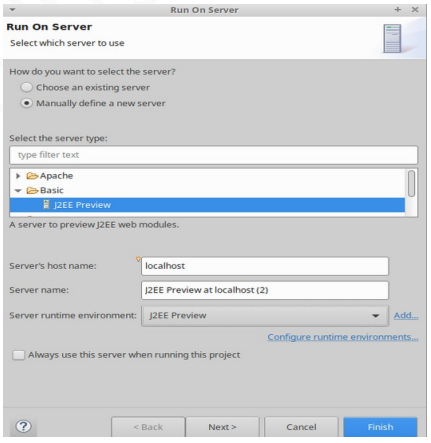
Ahora vamos a tratar de ejecutar el proyecto para verlo en ejecución

Ejecutando la aplicación y agregando un servidor web

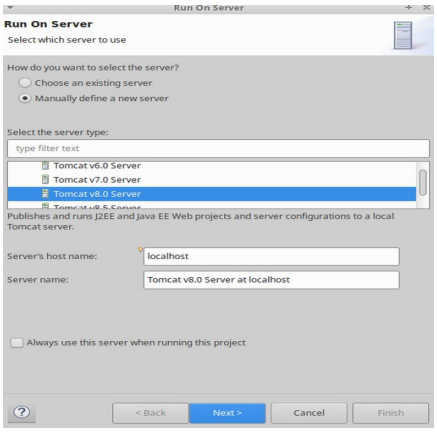
Botón derecho sobre el proyecto → Runas - → run on server



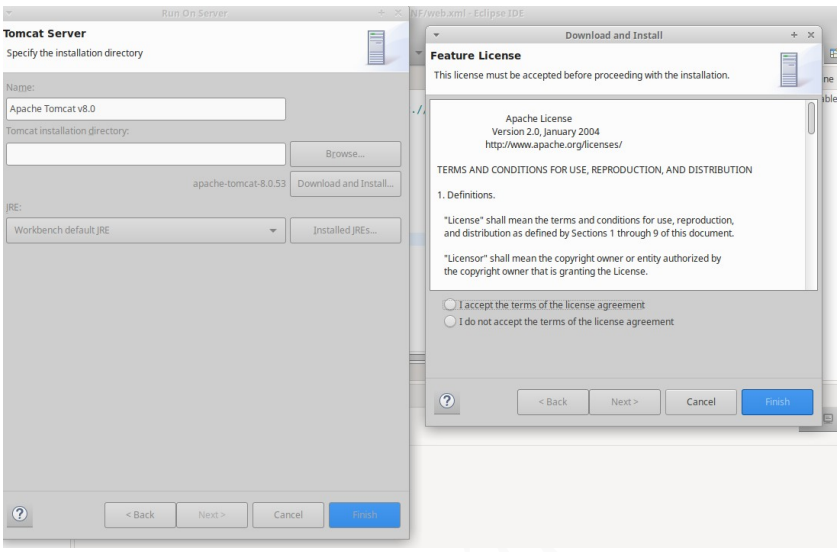
El wizard nos dará la opción de un embebido j2ee preview. Nosotros vamos a usar un tomcat. Para ello observar el enlace que dice: Add..



En el ejemplo se ha elegido un tomcat 8 para agregar



Aceptamos las condiciones de la licencia para poder continuar

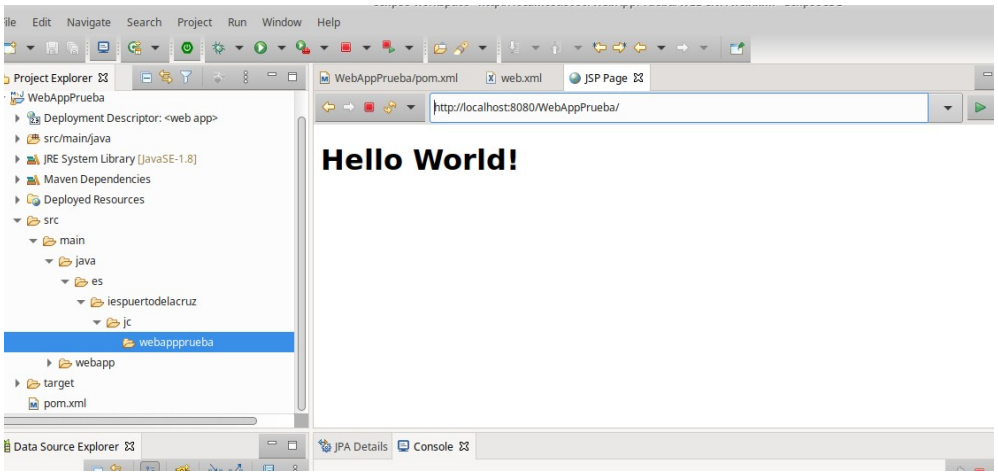


Ahí se descarga una versión de tomcat (lo podemos poner en la misma carpeta donde tenemos eclipse)

Puede que aparezca un error al ejecutar la aplicación:

class java.lang.String cannot be cast to class org.eclipse.wst.server.core.util.HttpLaunchable (java.lang.String is in module java.base of loader 'bootstrap'; org.eclipse.wst.server.core.util.HttpLaunchable is in unnamed module of loader org.eclipse.osgi.internal.loader.EquinoxClassLoader @2eaecd47)

No pasa nada Al ir a la ruta de la página se observa que funciona correctamente:



POM de nuestra aplicación

Observar que con los procedimientos descritos tendremos ya un POM. Los cambios que podemos tener que aplicar adicionalmente serán en las <dependencies > y en plugin maven compiler Que es lo que vamos a poner ahora:

```
<dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.1.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet.jsp</groupId>
            <artifactId>javax.servlet.jsp-api</artifactId>
            <version>2.3.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.el</groupId>
            <artifactId>javax.el-api</artifactId>
            <version>3.0.0</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version>
        </dependency>
        <dependency>
            <groupId>taglibs</groupId>
            <artifactId>standard</artifactId>
            <version>1.1.2</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>8.0.26</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-core -->
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>2.12.2</version>
        </dependency>
</dependencies>
```

Ahora la parte del compilador maven (para especificar la versión de Java) En esta parte lo único que se cambiará es que aparezca 1.8 en source y lo mismo en target

```
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.6.0</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
```

El web.xml puede mostrar algún error (realmente no es relevante pero mejor que no aparezca) Debe tener la siguiente cabecera para que coincida apropiadamente con el pom descrito:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns="http://Java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
      version="3.0">
```

Para usar jstl en las página jsp tenemos que tener un comienzo de página parecido a este (la primera línea **NO** se puede cambiar)

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
```

Anexo: Operaciones Java Web elementales

Servlets

Es una clase en el lenguaje de programación Java, utilizada para ampliar las capacidades de un servidor. Aunque los servlets pueden responder a cualquier tipo de solicitudes, estos son utilizados comúnmente para extender las aplicaciones alojadas por servidores web

Muy útiles para leer cabeceras de mensajes, datos de formularios, gestión de sesiones, procesar información, etc.Pero tediosos para generar todo el código HTML El mantenimiento del código HTML es difícil

Lo siguiente es una sección de los import de un servlet Java y la declaración de la clase java que soporta servlet:

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 *
 * @author carlos
 */
public class NewServlet extends HttpServlet {
```

Los servlets heredan de la clase HttpServlet y permiten gestionar elementos HTTP mediante las clases (se muestran en los import de arriba):

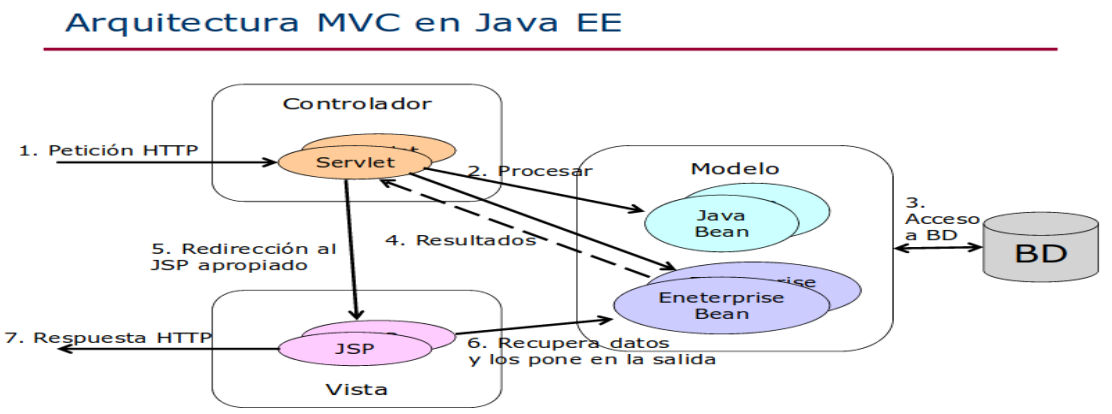
* **HttpServletRequest**: recibe la petición

* **HttpServletResponse**: genera la respuesta.

* **HttpSession**: permite crear una sesión común a un conjunto de request (ámbito de sesión).

* **ServletContext**: gestiona la información común a todas las peticiones realizadas sobre la aplicación (ámbito de aplicación). Se obtiene a partir del método getServletContext() de la clase HttpServlet.

Los servlets en un modelo MVC está ubicado en la parte del controlador, ya que devuelve una respuesta (**Response**) (que podría ser una página jsp, un fichero, un video, audio,…) y debe interactuar con las peticiones (**Request**) de los usuarios



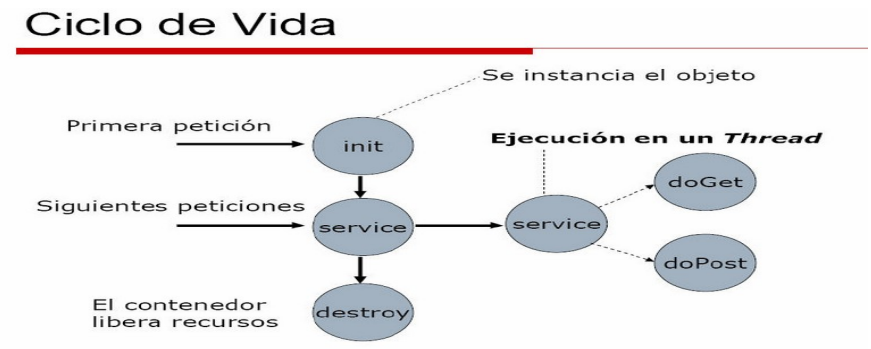
Los servlets no tienen el método main() como los programas Java, sino que se invocan unos métodos cuando se reciben peticiones. A esta metodología se le llama ciclo de vida de un servlet y viene dado por tres métodos: init, service, destroy:

INICIALIZACIÓN: Una única llamada al método “init” por parte del servlet. Incluso se pueden recoger unos parámetros concretos con “getInitParameter” de “ServletConfig” inicialales y que operarán a lo largo de toda la vida del servlet.

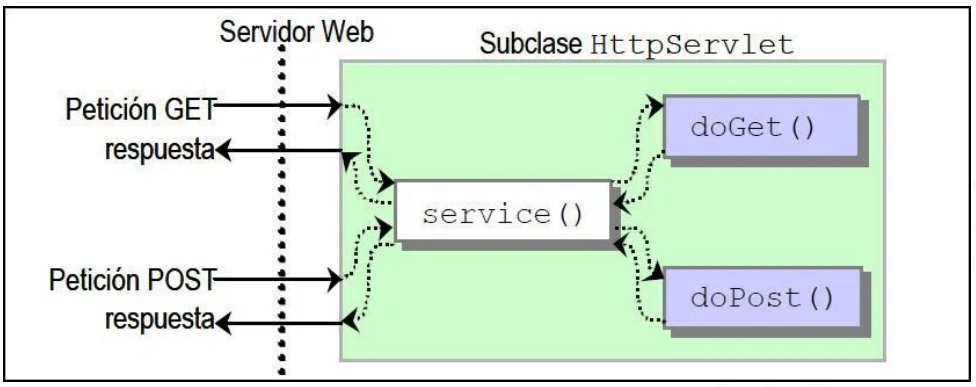
SERVICIO: una llamada a service() por cada invocación al servlet para procesar las peticiones de los clientes web.

DESTRUCCIÓN: Cuando todas las llamadas desde el cliente cesen o un temporizador del servidor así lo indique o el propio administrador así lo decida se destruye el servlet. Se usa el método “destroy” para eliminar al servlet y para “recoger sus restos” (garbage collection).

Cada vez que el servidor pasa una petición (distinta a la primera) a un servlet se invoca el método service(), este método habrá que sobreescribirlo



(override). Este método acepta dos parámetros: un objeto petición (request) y un objeto respuesta. Los servlets http, que son los que vamos a usar, tienen ya definido un método service() que llama a doXxx(), con Xxx el nombre de la orden que viene en la petición al servidor web. Estos dos métodos son **doGet() y doPost()** y nos sirven para atender las peticiones específicamente provinientes de métodos GET o POST respectivamente,



El código que genera Eclipse al crear un servlet:

```
public class Principal extends HttpServlet {
	private static final long serialVersionUID = 1L;

    public Principal() {
        super();
    }

	protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {

     }
}
```

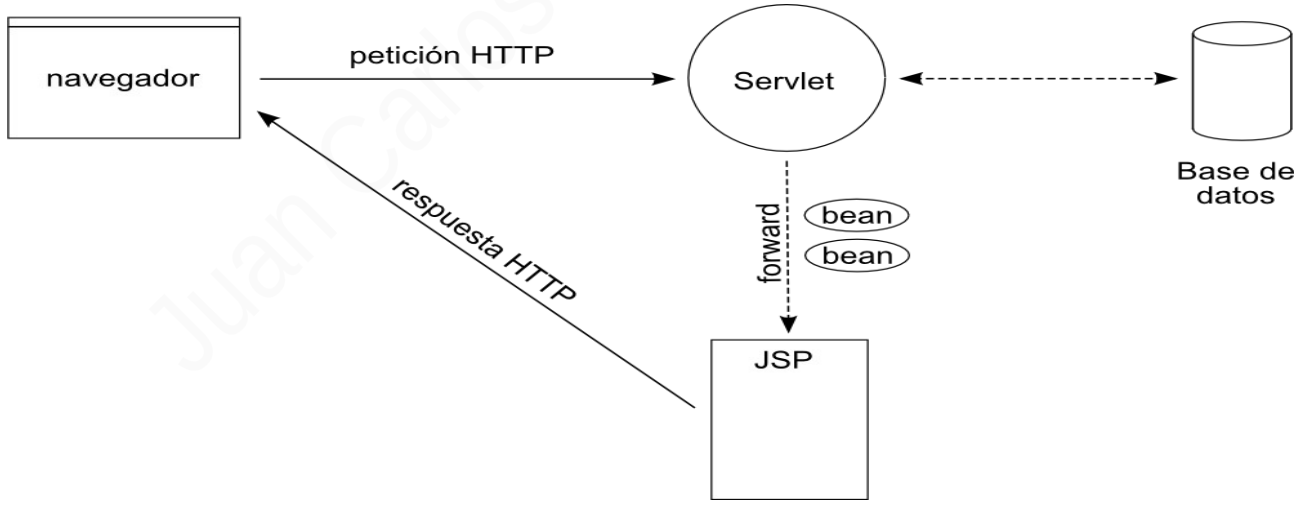
Ahí, dentro del método doGet() es donde respondemos a una petición (request) del usuario

Ahora veremos como podemos devolverle una página web de respuesta a una petición del usuario de tipo GET:

```
       protected  void  doGet(HttpServletRequest  request,  HttpServletResponse
response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {

            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet </title>");
            out.println("</head>");
            out.println("<body>");
             out.println("<h1>Servlet NewServlet at " + request.getContextPath()
+ "</h1>");
            out.println("</body>");
            out.println("</html>");
    }
```

Lo anterior es válido para devolver directamente una página html. Pero el procedimiento habitual es normalmente que el Servlet haga las veces de controlador y que derive a una vista JSP que sea la que se encargue de mostrar la página al usuario:



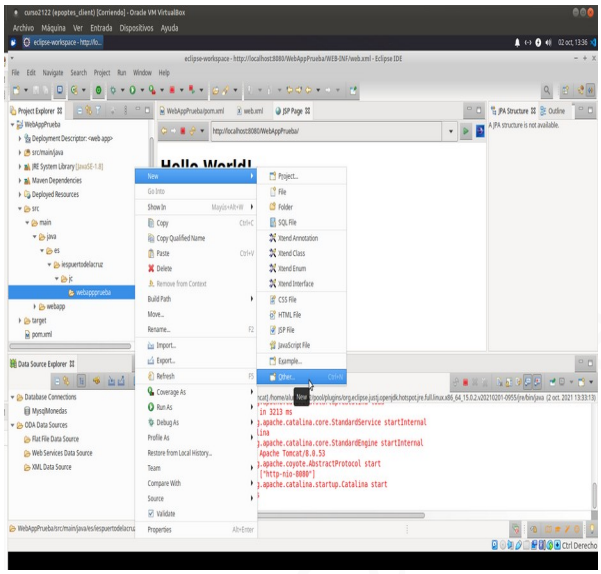
Observar en el gráfico que la petición la recibe el Servlet. Luego el servlet hace las consultas pertinentes a la base de datos y envía lo que ha obtenido a la página JSP (fijarse donde dice: bean eso son los datos que tomó de la base de datos y se los envía a JSP)

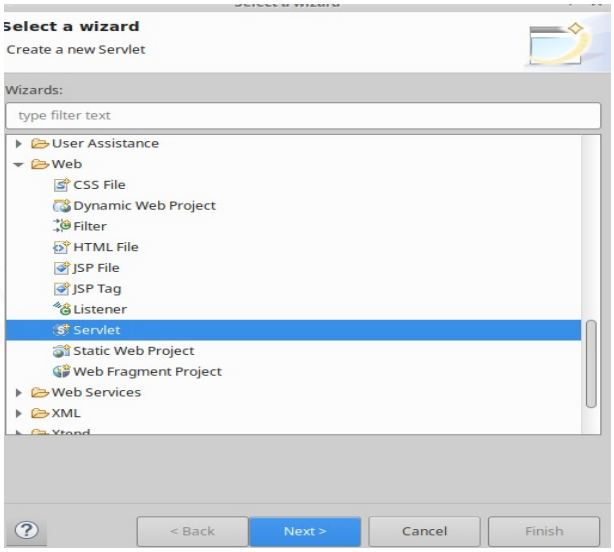
Agregando un Servlet (un controlador) a la app

El funcionamiento de Java Web hace que hasta nuestras vistas (las páginas .jsp) tienen un Servlet detrás. Esto es, una clase Java que va volcando en la salida out lo que va a devolver el protocolo http

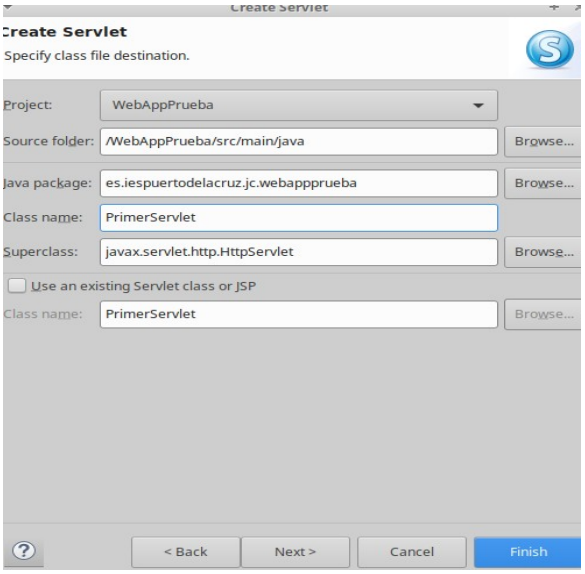
En general asociaremos los Servlet a nuestros controladores web. Y las páginas jsp a las vistas. **De hecho, una buena práctica es que la consulta del cliente vaya a un controlador-servlet y éste envíe lo que corresponda a la vista JSP**.

Para agregar un Servlet: Botón derecho →New → Other → Web → Servlet

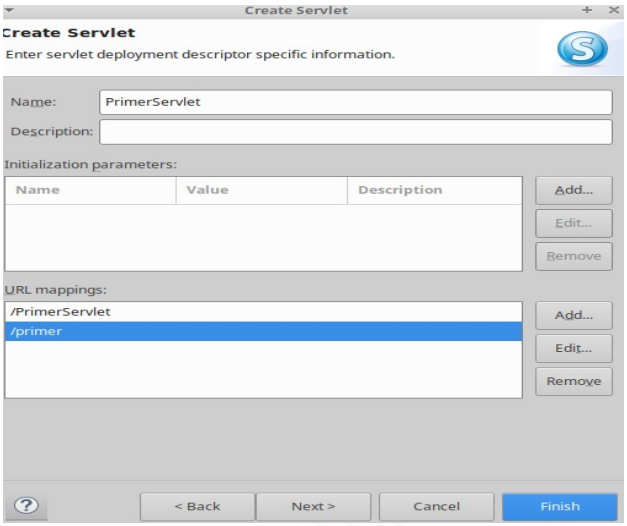




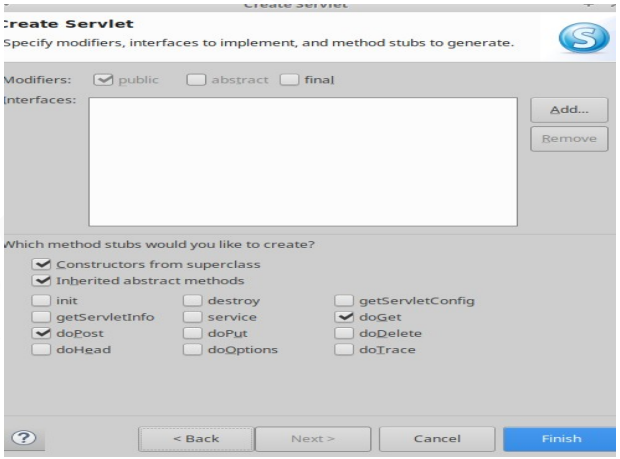
En la siguiente pantalla le ponemos nombre al Servlet. En el ejemplo se llama PrimerServlet:



En la siguiente pantalla especificamos el mapping (la url con la que se va a acceder al servlet). Observar que en este caso se puede acceder al servlet, agregando a la ruta base de nuestra app: /primer

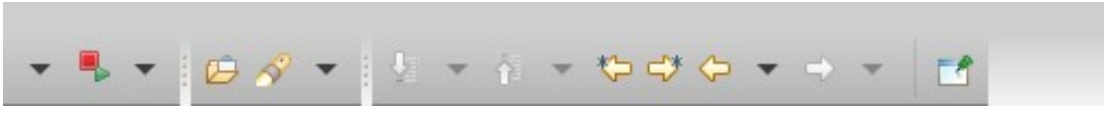


La siguiente pantalla nos establece a qué va a responder (peticiones get, post, etc)



Por defecto el Servlet creado devuelve (lo recoge y lo muestra el navegador) ya una frase.

Para que la aplicación reconozca el Servlet podemos reiniciar a la vez el servidor (botón rojo con flecha verde: relaunch tomcat) :



Según lo que hayamos puesto, y la versión de Eclipse pueden darse errores. Uno de ellos es que el fichero web.xml tenga error en su declaración. Aprovechamos para ver un ejemplo de ese fichero y entender un poco

Ejemplo de un web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://JAVA.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-
app_2_4.xsd">

    <servlet>

        <servlet-name>Primero</servlet-name>
        <servlet-class>es.iespuertodelacruz.jc.matriculas.Primero</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Primero</servlet-name>
        <url-pattern>/primero</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>

        <welcome-file>primero</welcome-file>
        <welcome-file>index.jsp</welcome-file>
        </welcome-file-list>
    </web-app>
```

Vemos que aparecen nuestros servlet, donde está alojado y el nombre que le hemos puesto. Así como la ruta url que nos permite acceder (url-pattern)

session-timeout es la cantidad de tiempo que tienen las sesiones ociosas antes de desconectar. Y welcome-file especifica cuál es el primer fichero que se va a ejecutar cuando se accede a la aplicación. En este caso hemos puesto el servlet: Primero y en su defecto (si fallara) index.jsp

El error que puede darse es tener mal puesto el namespace. Usando minúsculas en la palabra JAVA cuando corresponde mayúscula:

posible error que aparece a veces en web.xml:

`xmlns="http://java.sun.com/xml/ns/javaee"`

lo cambiamos a:

`xmlns="http://JAVA.sun.com/xml/ns/javaee"`

Para hacer pruebas en la aplicación web respecto a si funciona bien los DAO se muestra un ejemplo

```
GestorConexionDDBB gc = new GestorConexionDDBB("instituto","root","1q2w3e4r");
AsignaturaDAO aDAO = new AsignaturaDAO(gc);

Asignatura asignatura = aDAO.findById(1);
asignatura.setNombre("BAEN");
aDAO.update(asignatura);
asignatura = aDAO.findById(1);

if(asignatura != null)
      request.setAttribute("asignatura", asignatura);

request.getRequestDispatcher("index.jsp").forward(request, response);
```

Observar: **getRequestDispatcher** es muy imporante porque se encarga de redirigir hacia la página JSP (la vista)

Redireccionar a la vista (JSP) desde un Servlet

Vamos a analizar ahora las redirecciones. Algo habitual es que un servlet redireccione a una página u otra según lo que tenga lugar. En ese proceso de redirección puede proceder a establecer información adicional para la pagina a la que va a redirigir.

Haremos uso de: request.setAttribute() para establecer información a la página que vamos a redirigir y usaremos: request.getRequestDispatcher(“nombrepagina.jsp”); y RequestDispatcher.forward() para hacer la redirección

Ejemplo:

```
request.setAttribute("prueba", 5); // establecemos el parámetro: prueba a: 5

// reenviamos:

RequestDispatcher rd = request.getRequestDispatcher("unaPagina.jsp");

rd.forward(request, response);
```

El equivalente en JSP de request.setAttribute() es: request.getServletContext().setAttribute()

Observar que esto es diferente de **parameter** que se usa cuando se recoge información del usuario.

Veamos un caso en que un usuario nos envíe un parámetro y queramos recogerlo (por ejemplo, nos envía un formulario)

```
request.getAttribute(name)  // donde name es el nombre del atributo
```

Debemos entender que getParameter() nos devuelve los parámetros. AQUELLOS PARÁMETROS QUE EL CLIENTE ENVÍA AL SERVIDOR. getAttribute() es desde el lado del servidor, son datos que establece un servlet y por ejemplo, se envían a un jsp. Adicionalmente observar que getParameter() devuelve String mientras que getAttribute() devuelve cualquier cosa

Vamos a ver lo anterior todo con un ejemplo completo:

la página **index.html**

```
<!DOCTYPE html>
<html>
    <head>
        <title>Start Page</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
    </head>
    <body>
        <h1>Hello World!</h1>
        <a href="procesar?unParametro=7" >pulsa aquí por favor </a>
    </body>
</html>
```

Observar que se llama a un servicio llamado: procesar y se le está enviando por método GET un parámetro llamado: unParametro con valor: 7

El contenido del fichero: **web.xml**:

```
<?xml version="1.0" encoding="UTF-8" ?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xmlns:xsi="
    <servlet>
        <servlet-name>Pablo</servlet-name>
        <servlet-class>controller.ServletPrueba</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Pablo</servlet-name>
        <url-pattern>/procesar</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
</web-app>
```

Observar que "**/procesar**" está mapeado a un nombre de Servlet que es: "**Pablo**" y a su vez tenemos la ruta de ese nombre de Servlet que realmente coincide con una clase Java llamada: "**ServletPrueba**"

Veamos ahora parte del contenido de ServletPrueba.java:

```
public class ServletPrueba extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");

        //vamos a tomar el dato que nos envió la url ( mediante GET )
        String strUnParametro = request.getParameter("unParametro");

        int unParametro = 0;
        try{
            unParametro = Integer.parseInt(strUnParametro);
        }catch(Exception ex){}

        //vamos a sumar 5 al dato que nos venga en unParametro y lo vamos a enviar
        //mediante un atributo llamado: prueba
        request.setAttribute("prueba", unParametro + 5);


        //request.getRequestDispatcher("unaPagina.jsp").forward(request,response);
        RequestDispatcher rd = request.getRequestDispatcher("unaPagina.jsp");
        rd.forward(request, response);
    }
```

Vemos que se usa: request.getParameter() para obtener la información que nos enviaron

Después vemos que creamos un nuevo atributo de la request llamado: "prueba" y estamos ingresando el número que nos enviaron dentro del parámetro: "unParametro" y le sumamos 5

Luego vemos que redireccionamos hacia una página JSP llamada: **unaPagina.jsp**

Veamos ahora el contenido de: unaPagina.jsp

```jsp
<%@page import="java.util.Enumeration"%>
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Hello World!</h1>
        <%
            Enumeration<String> atributos = request.getAttributeNames();

            while( atributos.hasMoreElements()){
                String atributo = atributos.nextElement();
                out.write(atributo +"<br>");
                out.write(request.getAttribute(atributo) +"<br><br>");
                // atributo es la key, request.getAttribute(atributo) sería el value



            }


        %>
        <p>Efecto usando getParameter():          <%= request.getParameter("prueba") %>
        </p>
        <p>Efecto usando getAttribute():          <%= request.getAttribute("prueba") %>
        </p>

    </body>
</html>
```

Observamos como podemos recorrer todos los atributos sin conocer sus nombres

Y también que si buscamos el dato como un parámetro no lo encontraremos: (request.getParameter()) pero que si lo buscamos como atributo sí: request.getAttribute()

Tener en cuenta que con getAttribute() podemos tomar cualquier tipo de objeto, getParameter() devuelve únicamente String

Usando las plantillas JSTL

Hoy en día tenemos diferentes opciones para crear nuestras vistas. Pero para una visión rápida y sencilla, vamos a explicar como incorporamos jstl a nuestro proyecto maven-eclipse

Agregamos al pom.xml:

```
<!--  dependencias agregadas para que funcione jstl -->
<dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-api</artifactId>
    <version>6.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.5</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.2</version>
</dependency>
```

Y ahora **importante**, ¡¡ Justo al inicio de la página JSP !! agregamos:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

A partir de ahora ya podemos hacer uso de expresiones jstl en las páginas jsp:

```
<c:forEach items="${lista}" var="alumno">
```

En el ejemplo anterior se recibe un atributo del controlador llamado: lista y vamos tomando en cada iteración un objeto de esa lista en la variable alumno

Disponemos de if, forEach, choose (pensar en el que usamos con XML en lenguaje de marcas), etc

Para una información completa de lo que podemos hacer con JSTL:

https://www.javatpoint.com/jstl