

## Tema 4: Aplicaciones web con Java

### Parte I:

Introducción. API básico

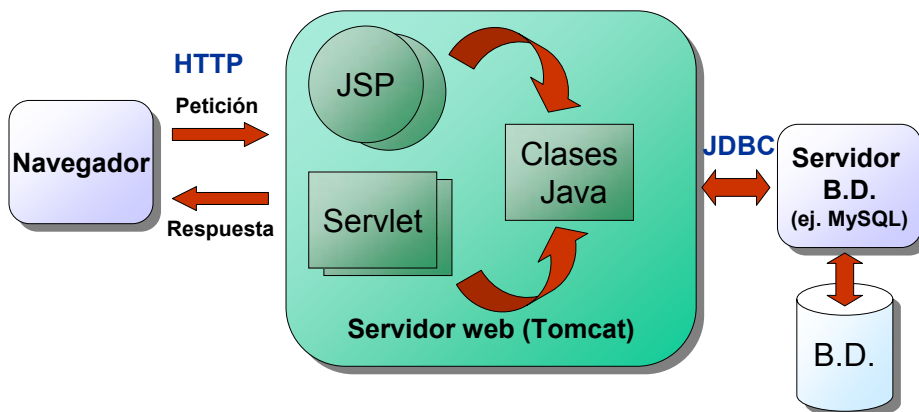
JDBC

Arquitectura de aplicaciones web



Tecnologías  Web

## Aplicaciones web con Java



Tecnologías  Web

## 2. API de JSP básico

### Inserción de java en páginas web



## Ejemplo de página JSP

```
<html>
<head>
  <title>Ejemplo de JSP</title>
</head>
<body>
<h1>
<% String nombre;
  nombre = request.getParameter("nombre"); %>
  Hola <% out.print(nombre); %>
</h1>
</body>
</html>
```



## Características básicas de un JSP

- El código Java no necesita ponerse dentro de una clase, ni de ningún método, se ejecuta directamente cuando se hace una petición HTTP
- Hay una serie de variables predefinidas
  - out: *salida*
  - request: *petición HTTP*
  - response: *respuesta HTTP*
- Se puede colocar código de 3 formas distintas
  - Declaraciones: `<%! ... %>`      `<%! int visitas = 0 %>`
  - Acciones: `<% ... %>` (se ejecutan)      `<% visitas++ %>`
  - Expresiones: `<%= ... %>` (su valor se escribe en el cuerpo de la respuesta HTTP)      `<%= visitas %>`



## 3 formas de escribir en la respuesta (variable out)

- Todo lo que no esté entre `<% y %>` (generalmente código HTML)
- Lo que esté entre `<%= y %>` (expresiones Java)

```
<table>
  <% for int i=0; i<5; i++ { %>
    <tr>
      <td> <%= nombre[i] %> </td>
      <td> <%= nota[i] %> </td>
    </tr>
  <% } %>
</table>
```
- Lo que esté entre `<% y %>` con instrucciones `out.print()`

```
<tr> es lo mismo que <% out.print("<tr>") %>
```



## Salida condicional

```
<%@ page import = "java.util.*" %>
<html>
<head>
<title>Saludo</title>
</head>
<body>
<%
    Calendar ahora = Calendar.getInstance();
    int hora = ahora.get(Calendar.HOUR_OF_DAY);
%>
Hola mundo,
<% if ((hora>20)|| (hora<6)) { %>
    buenas noches
<% }
    else if ((hora>=6)&&(hora<=12)) { %>
        buenos días
<%
    }
    else { %>
        buenas tardes
<%
    } %>
</body>
</html>
```



## Interacción básica con petición/respuesta

### ■ **request:** representa la petición HTTP

- **request.getParameter(nombre):** devuelve un parámetro HTTP como una cadena (hay que convertir al tipo requerido)
  - Si el parámetro no existe, devuelve **null**
  - Si el parámetro es vacío, devuelve la cadena vacía
- **request.getHeader(nombre):** devuelve el valor de una cabecera HTTP

### ■ **response:** representa la respuesta HTTP

- **response.sendRedirect(url):** redirección a la URL especificada
- **response.setHeader(nombre, valor):** envía una cabecera HTTP en la respuesta con un valor específico



## 2. JDBC

### Acceso a bases de datos en Java



## JDBC

- No se hace necesariamente siempre desde web, es una librería aparte
- Independiza en gran medida nuestro código de la B.D. empleada: no importa que usemos MySQL, Access u Oracle (salvo por el dialecto de SQL)
- Pasos para realizar una operación con la B.D.
  - Establecer la **conexión** con el servidor de B.D. (**Connection**)
  - Crear una **sentencia SQL** (**Statement**)
  - Ejecutar la **sentencia**
  - Procesar los **resultados** (**ResultSet**) si es SELECT
  - Cerrar la **conexión**



## ¿Cómo se consigue una conexión?

- **Antes que nada:** necesitamos un driver para nuestro motor de bases de datos (normalmente un .jar)
- **Manualmente:**

```
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection("jdbc:mysql://localhost/ejemplotw");
```

  - **Problema:** abrir una conexión es costoso (¡milisegundos!). Si cada operación con la BD requiere esto, apañados vamos
- **Pool de conexiones: nada más arrancar abrimos unas cuantas conexiones y las vamos reusando a medida que las necesitamos**
  - **Gestionado por Tomcat**
  - **Gestionado por librerías de terceros o por nosotros**



## Conexiones gestionadas por Tomcat

- **Archivo *context.xml* : datos de la conexión**

```
<Context>
  <Resource name="jdbc/tw" auth="Container" type="javax.sql.DataSource"
    username="tw" password="tw"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost/tw" maxActive="8" maxIdle="4"/>
</Context>
```
- **Código Java para acceder a una conexión**

```
Context contextoInicial = new InitialContext();
Context contextoEntorno = (Context) contextoInicial.lookup("java:comp/env");
ds = (DataSource) contextoEntorno.lookup("jdbc/tw");
Connection con = ds.getConnection()
...
con.close(); //una vez hecho el trabajo, hay que acordarse de cerrar la conexión
```



## Crear y ejecutar la sentencia

### ■ Crear la sentencia

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

### ■ Ejecutarla

- Si es un SELECT, método **executeQuery**. Devuelve un objeto de tipo **ResultSet** con los registros

```
ResultSet datos = sentencia.executeQuery("select * from clientes");
```

- Si es un INSERT, UPDATE o DELETE, método **executeUpdate**. Devuelve un entero indicando el número de registros afectados

```
int filas = sentencia.executeUpdate("delete from clientes where cod=1");
```



Tecnologías  Web

## SQL con “parámetros”: opciones

### ■ Crear SQL “sobre la marcha” concatenando cadenas

```
public obtenerCliente(String codCli {  
    Statement sql = con.createStatement()  
    ResultSet rs = sql.executeQuery("select * from clientes  
        where cod=" + codCli + """);  
    ...  
}
```

### ■ PreparedStatement: SQL precompilado en la BD a falta de los parámetros

```
PreparedStatement pstat = con.prepareStatement("UPDATE CLIENTES  
    SET saldo=? WHERE id=?");  
  
pstat.setFloat(1,150.25);  
pstat.setInt(2,1)  
pstat.executeUpdate();  
...
```

### ■ PreparedStatement no solo es útil por eficiencia, también para evitar “SQL injection”

- En *obtenerCliente*, ¿Qué pasa si cliente="1; or 1=1"?



Tecnologías  Web

## Recorrer registros en SELECT

### ■ Método **next()** de ResultSet.

- Devuelve **true** si hay siguiente registro y avanza a él.
- Inicialmente hay que hacer **next()** para apuntar al 1er registro. Si devuelve *false* es que no hay registros.
- Por tanto, **siempre hay que hacer next()**, aunque solo haya 1 registro (en este caso solo se hará next() una vez)

### ■ Métodos **getXXX(nombre\_campo)** de ResultSet. Obtienen el valor del campo nombre\_campo del registro actual

- getString(...), getInt(...), getFloat(...), getBoolean(), getDate(),...

```
ResultSet datos = sentencia.executeQuery("select * from clientes");
while (datos.next()) {
    out.println(datos.getString("apellidos"));
    out.println(datos.getInt("edad"));
}
```



## ■ 3. Arquitectura de aplics. web

### ■ Cómo estructurar el código



## Modelo monolítico

```
<% Context contextInicial = new InitialContext();
Context contextoEntorno = (Context) contextInicial.lookup("java:comp/env");
ds = (DataSource) contextoEntorno.lookup("jdbc/tw");
Connection con = ds.getConnection();
Statement sentencia = con.createStatement();
ResultSet rs = sentencia.executeQuery("select * from entradas order by fecha desc
limit " + request.getParameter("cuantas") %>
<html>
  <head><title>Blog de TW: últimas entradas</title></head>
<body>
  <% while (rs.next()) { %>
    <h1> <%= rs.getString("titulo")%> </h1>
    <p> <%= rs.getString("texto")%> </p>
    <a href="verEntrada.jsp?id=<%=rs.getInt("idEntrada")%>">Leer más</a>
  <% } %>
</body>
</html>
```



Tecnología Web

## Ventajas y problemas

- **Para aplicaciones pequeñas**, el código monolítico es sencillo y compacto, todo lo de la página está contenido en un solo archivo .jsp
- Pero a medida que crece la aplicación
  - **No es modular**: el acceso a la base de datos, la lógica y la presentación están mezcladas. El código está desorganizado
    - Si hay un **equipo de trabajo** se hace difícil separar responsabilidades (diseñador-HTML, programador-BD, lógica)
  - **Cualquier cambio afecta potencialmente a todo el código** (por ejemplo cambia el nombre de una tabla, o los campos).
  - No se puede **reutilizar el código para otro cliente** que no sea el navegador
    - Acceso a través de dispositivos móviles
    - Servicios web



Tecnología Web

## Introduciendo algo de estructura...

```
<%
  ArrayList<Entrada> ultimas;
  EntradasBD buscador;
  buscador = new EntradasBD();
  ultimas = buscador.verUltimasEntradas(request.getParameter("cuantas"));
%>
<html>
  <head><title>Blog de TW: últimas entradas</title></head>
<body>
  <% for (Entrada e : ultimas) %>
    <h1> <%= e.getTitulo() %> </h1>
    <p> <%= e.getTexto() %> </p>
    <a href="verEntrada.jsp?id=<%=e.getIdEntrada()%"%>">Leer más</a>
  <% } %>
</body>
</html>
```



Tecnologías  Web

## Clase java aparte

***/\* CUIDADO, esto no se corresponde exactamente con el código de la plantilla de la práctica, está modificado para que se vea mejor en la transparencia \*/***

```
public class EntradasBD {
  ...
  public ArrayList<Entrada> verUltimasEntradas(int numero) {
    Connection con; Statement sql; ResultSet rs;
    Entrada e; ArrayList<Entrada> ultimas;

    con = FuenteDatos.getConnection();
    sql = con.createStatement();
    rs = sql.executeQuery("select * from entradas order by fecha desc limit " + numero);
    ultimas = new ArrayList<Entrada>();
    while (rs.next()) {
      e = new Entrada();
      e.setIdEntrada(rs.getInt("idEntrada");
      e.setTitulo(rs.getString("titulo"));
      ...
      ultimas.add(e);
    }
    return ultimas;
  }
}
```



Tecnologías  Web

## En plan *fino*...

- Los arquitectos de aplicaciones web suelen distinguir 3 capas
- Cada capa es un "módulo" que solo depende de las capas con que se comunica
  - **Presentación: mostrar datos**
    - Página con formulario para meter una nueva entrada en el blog
  - **Lógica de negocio: las "reglas del juego"**
    - No puede haber una entrada del *blog* sin título
    - Para poder editar una entrada, hay que haber hecho login
    - ...
  - **Acceso a datos: meter y sacar información de la B.D.**
    - Insertar un registro en la tabla "entradas"

