



UNIVERSIDAD DE CÓRDOBA

PROGRAMACIÓN WEB - JSP-II

J2EE/JSP, JDBC,
DAO, DTO, MVC

Dr. José Raúl Romero Salguero
jrromero@uco.es



JSP & Servlet

Contenidos

1. Instalación
2. Elementos de JSP
3. Objetos implícitos en JSP
4. Acceso a datos desde Java
5. Patrón Modelo-vista-controlador (MVC)
6. Recomendaciones de diseño

4.

Acceso a datos

Accediendo a la capa de datos

Bases de datos en la UCO

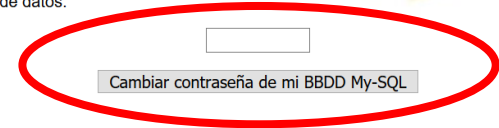


Bases de datos UCO

- ▶ Para la elaboración de prácticas, **utilice la base de datos MySQL** dispone en la UCO
- ▶ Todo estudiante de la asignatura debería registrado en <http://oraclepr.uco.es/abd/>
 - ❑ En caso contrario, contacte con su profesor de prácticas
- ▶ Utilice **phpMyAdmin** para la gestión de la base de datos
- ▶ **Recuerde cambiar la clave** de la base de datos en el portal UCO para la elaboración de las prácticas

Cambiar la contraseña de acceso

En este apartado podrá cambiar la contraseña de acceso al servidor de Base de Datos MySQL, con el fin de que a clave utilizada en procedimientos PHP que puedan ser consultados por sus compañeros, no figure su clave personal. Recuerde que deberá modificar la contraseña en las cadenas de conexión de todos los programas que accedan a la base de datos.



http://www.uco.es

Usuario: IN

Ampliación de BBDD

Mi Base de Datos

I. Estado actual de su BBDD

Su BBDD personal **EXISTE**. Estos son los parámetros que debe de utilizar

- Nombre de la BBDD: in1rosaj
- Usuario: in1rosaj
- Clave: su clave de correo electrónico, si nó la cambió por otra desde aquí
- Servidor: oraclepr.uco.es

Introducción

En esta sección del panel de control podrá dar de alta su base de datos en un servicio MySQL; también puede dar de baja esta base de datos si la hubiese dado de alta previamente.

Cuando se da de alta una base de datos, el sistema le asigna el usuario y la clave que deberá utilizar para conectarse al servidor desde PHP o cualquier otra utilidad. El nombre de la base de datos que se crea coincide también con este nombre de usuario. La contraseña es la que ha introducido para la autenticación en esta página. Se recomienda no utilizar contraseñas de usuario y por tanto, fíjese Ud. la posibilidad de cambiar la

Bases de datos UCO

| | |
|-----------------------|---|
| <i>Usuario:</i> | <Login de usuario> |
| <i>Password:</i> | <Clave indicada> |
| <i>Servidor BBDD:</i> | http://oraclepr.uco.es/ |
| <i>Nombre BBDD:</i> | <Nombre creado> |

Consideraciones:

- ▷ Un equipo utilizará una única base de datos de un miembro del equipo
- ▷ Indique un *password* a la base de datos que pueda revelar, ya que el código deberá ser visible al profesorado (la clave puede cambiarse)
- ▷ El nombre de la base de datos, al igual que el de las tablas y atributos, deben ser claros y descriptivos
- ▷ Incluya contenidos de ejemplo – creíbles y profesionales

Acceso a bases de datos - JDBC

- ▷ JDBC es la [API de Java para acceso a bases de datos SQL](#)
- ▷ MySQL podría requerir añadir como dependencia externa `mysql-connector.jar`
- ▷ Requiere importar el paquete `java.sql` y ofrece las siguientes clases importantes:

| Clase Java | Descripción |
|--------------------------------|---|
| <code>DriverManager</code> | Carga el driver de la base de datos |
| <code>Connection</code> | Establece la conexión con la base de datos |
| <code>Statement</code> | Ejecuta sentencias SQL (principalmente para consultas) |
| <code>PreparedStatement</code> | Ejecuta sentencias SQL en base datos, especialmente cuando la sentencia requiere más de una ejecución (p.ej. transacciones o reiteradas invocaciones) o se ejecuta con parámetros |
| <code>ResultSet</code> | Guarda el resultado de una consulta |

JDBC - Establecimiento de conexión

- ▶ La conexión se abre utilizando el *driver* correspondiente y tomando la cadena de conexión del ejemplo (en cursiva, según cada caso)
- ▶ Se debe ajustar el diseño del código para no abrir/cerrar la conexión constantemente, sino para aprovechar una conexión para múltiples operaciones (si procede)
- ▶ **Connection** es la clase que representa la conexión abierta
- ▶ Los datos de conexión deberían estar almacenados en el archivo de configuración
- ▶ Debe tenerse en cuenta la captura de la excepción **SQLException**

```
public Connection getConnection(){
    Connection con = null;
    try {
        Class.forName("com.mysql.jdbc.Driver");
        con=DriverManager.getConnection("jdbc:mysql://oraclepr.uco.es:3306/basedatos","usuario","passwd");
    } catch(Exception e) {
        System.out.println(e);
    }
    return con;
}
```


JDBC - Consultas

- ▷ **Statement** es la interfaz que representa a una sentencia SQL, que no admite parámetros
- ▷ El resultado de una sentencia generará objetos de clase **ResultSet**
- ▷ Para acceder a los datos de **ResultSet** se pueden utilizar métodos *getter* de la forma **getString**, **getInt**, **getLong**, etc. bien por índice de columna (más eficiente) o por alias de la columna (en consulta)
 - ❑ El índice de columnas es 1-index y se leen una única vez de izquierda a derecha
 - ❑ También es posible acceder con un cursor: **next**, **previous**, **first**, **last**, **beforeFirst**, **afterLast**, etc.

```
//añade un año a todos los usuarios
while (userRS.next()) {
    int age = userRS.getInt("age");
    userRS.updateInt( "age", age + 1);
    userRS.updateRow();
}
```

```

public Hashtable<String,String> queryById (int id) {
    Statement stmt = null;
    Hashtable<String,String> resul = null;
    try {
        Connection con=getConnection();
        stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("select last, first, age from User where id = " +
                                         id);

        while (rs.next()) {
            String last = rs.getString("last");
            String first = rs.getString("first");
            int age = rs.getInt("age");
            resul = new Hashtable<String,String>();
            resul.put("id", Integer.toString(id));
            resul.put("last", last);
            resul.put("first", first);
            resul.put("age", Integer.toString(age));
            System.out.println(id + "\t" + last +
                               "\t" + first + "\t" + age);
        }
        if (stmt != null) stmt.close();
    } catch (Exception e) { System.out.println(e); }
    return resul;
}

```

JDBC - Inserción, actualización y borrado

- ▶ **PreparedStatement** es la interfaz utilizada cuando se trata de sentencias SQL complejas, recurrentes y/o con parámetros

| Método | Descripción |
|--|---|
| <code>public void setInt(int paramIndex, int value)</code> | Establece el valor entero del parámetro <code>paramIndex</code> |
| <code>public void setString(int paramIndex, String value)</code> | Establece el valor cadena para el parámetro <code>paramIndex</code> |
| <code>public void setFloat(int paramIndex, float value)</code> | Establece el valor flotante del parámetro <code>paramIndex</code> |
| <code>public void setDouble(int paramIndex, double value)</code> | Establece el valor doble del parámetro <code>paramIndex</code> |
| <code>public int executeUpdate()</code> | Ejecuta una sentencia de actualización del tipo <i>create, drop, insert, update, delete</i> etc. |
| <code>public ResultSet executeQuery()</code> | Ejecuta una consulta devolviendo un <code>ResultSet</code> . Es utilizado en transacciones o consultas con parámetros o accesos múltiples |

```
public int save(int id, String last, String first, int age){
    int status=0;
    try{
        Connection con=getConnection();
        PreparedStatement ps=con.prepareStatement("insert into User (id,last,first,age) values(?,?,?,?)");
        ps.setInt(1,id);
        ps.setString(2,last);
        ps.setString(3,first);
        ps.setInt(4,age);
        status = ps.executeUpdate();
    } catch(Exception e) { System.out.println(e); }
    return status;
}
```

```
public int update(int id, String last, String first, int age){
    int status=0;
    try{
        Connection con=getConnection();
        PreparedStatement ps=con.prepareStatement("update User set last=?,first=?,age=? where id=?");
        ps.setString(1,last);
        ps.setString(2,first);
        ps.setInt(3,age);
        ps.setInt(4,id);
        status=ps.executeUpdate();
    }catch(Exception e){System.out.println(e);}
    return status;
}
```

```
public int delete(int id){  
    int status=0;  
    try{  
        Connection con=getConnection();  
        PreparedStatement ps=con.prepareStatement("delete from User where id=?");  
        ps.setInt(1,id);  
        status=ps.executeUpdate();  
    }catch(Exception e){System.out.println(e);}
  
    return status;  
}
```

El Patrón *Data Access Object*

Objetos DAO

DAO (*Data Access Object*)

- ▷ Los **objetos DAO** encapsulan el acceso a los datos, independientemente de la fuente en la que se almacenen, y los ponen en disposición de la capa de lógica (*business*)
- ▷ El acceso es **transparente al cliente** (generalmente en capa *business*)
- ▷ Los DAO **no tienen que corresponderse exactamente con tablas** o ficheros
 - ❑ Se puede crear un objeto similar a *User* (tabla de la base de datos) pero que se corresponda con una vista de los datos que nos interese contemplar
 - ❑ También pueden ser **vistas de negocio** de los datos: por ejemplo, podemos tener una clase de objetos **Purchase**, que recoja información de los productos, de los usuarios, del proceso de compra, etc. El acceso a las tablas de la base de datos implicadas, la obtención de los datos y el parseado lo realizaría la clase **PurchaseDAO**, que gestionaría la transacción

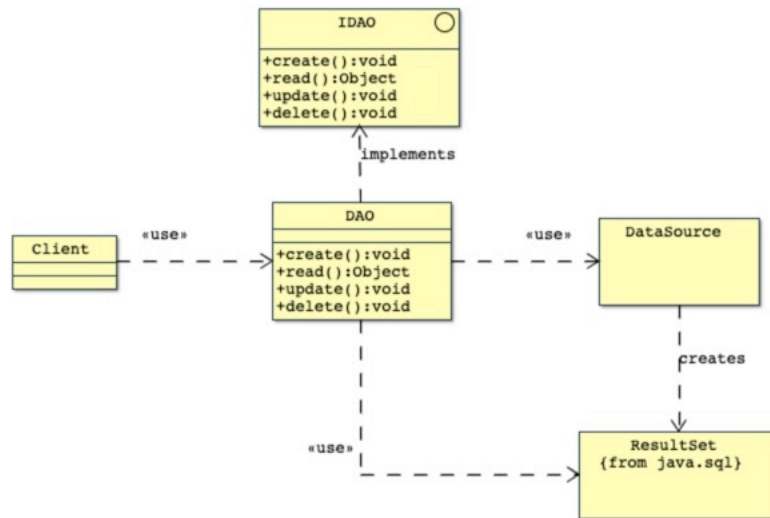
DAO (*Data Access Object*)

- ▷ Es habitual implementar una **interfaz IDAO** o heredar de una **clase genérica DAO**, y ofrecer una **excepción DAOException**
- ▷ Ofrecen las **operaciones CRUD** (*Create, Read, Update, Delete*)
- ▷ Ocultan la dificultad de acceso al **ResultSet** (en JDBC) al cliente, ofreciendo operaciones **más acorde al dominio de negocio** (*capa business*)
- ▷ Pueden tener **métodos de filtrado** de datos, **filter()**, de acceso por clave, **getById()**, o de acceso a todos los elementos, **getAll()**

DAO (*Data Access Object*)

- ▷ En general, se utiliza un **objeto DAO** distinto para cada objeto del modelo de objetos de la aplicación
 - ❑ Por ejemplo, si hay clientes que realizan pedidos, lo más habitual será tener **ClienteDAO** y **PedidoDAO**

Es frecuente utilizar un patrón **Factoría** para la creación de los distintos DAO de la aplicación (ClienteDAO, UsuarioDAO, FacturaDAO, etc.)



El Patrón *Data Transfer Object*

Objetos DTO

DTO (*Data Transfer Object*)

- ▷ Los **objetos DTO** son artefactos de diseño que encapsulan información sobre objetos del dominio de aplicación (business), generalmente producidos a partir de abstracciones de los casos de uso
- ▷ Erróneamente llamados **Value Objects** (error muy frecuente)
 - ❑ En realidad, los *value objects* son objetos con listas de valores (**enum** en Java)
- ▷ La utilidad de los DTO es la de **transferir datos entre componentes/elementos de la aplicación** – sirven en cualquier capa (presentación, negocio, integración...)
 - ❑ La tendencia por defecto sería crear **un DTO para cada objeto del modelo de dominio**
 - ❑ **No son reflejo del mundo real ni de la base de datos**, ya que pueden omitir atributos (p.ej., identificadores) o derivar otros nuevos (p.ej., se puede almacenar la fecha de nacimiento pero utilizar la edad en el dominio)

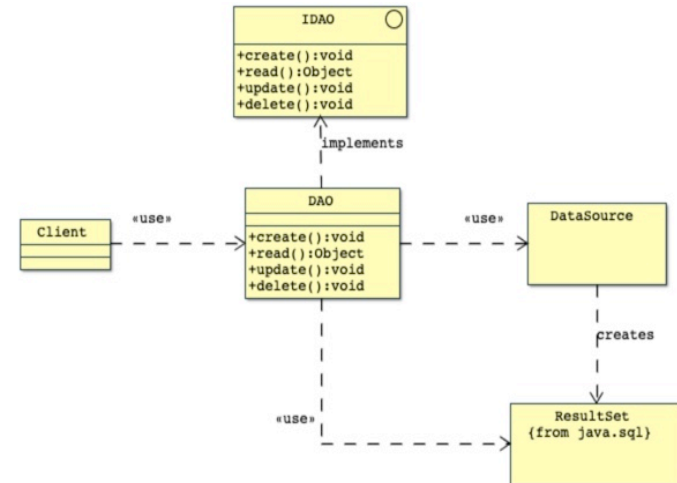
DTO (*Data Transfer Object*)

- ▷ Son **sólo objetos de Java**, esto es, **POJOs** (*Plain Old Java Objects*) – *Martin Fowler, Rebecca Parsons & Josh MacKenzie*
 - ❑ Objetos simples y “simplemente orientados a objetos”
 - ❑ No dependen de ningún *framework*
 - ❑ No extienden ni implementan nada (tan sólo `Serializable`)
 - ❑ Sólo contienen atributos, si bien pueden tener getters y setters para establecer la visibilidad
 - ❑ Similares a los *java beans*

DTO + DAO

- ▶ Lo más frecuente es utilizar una estrategia simultánea de DAO + DTO
 - ❑ Los DAO reciben las llamadas del Cliente, ocultan el acceso a la fuente de datos y transforman los **ResultSet** en objetos de negocio (DTO)
- ▶ Las invocaciones a **Create**, **Update** y **Delete** pueden pasar un DTO como parámetro
- ▶ Las invocaciones a **Read** devuelven un DTO

- 😊 Permite **transparencia** en el acceso a datos por parte del cliente
- 😊 Reduce la **complejidad** del código
- 😞 Posible **explosión de clases** → esfuerzo de implementación, aunque necesario



5.

Patrón Modelo-Vista- Controlador (MVC)

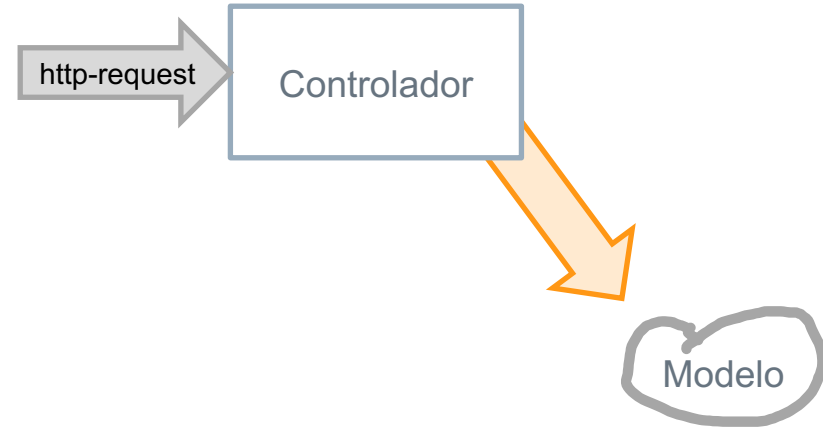
**La base de la implementación de aplicaciones web
con/sin *frameworks***

Esquema general



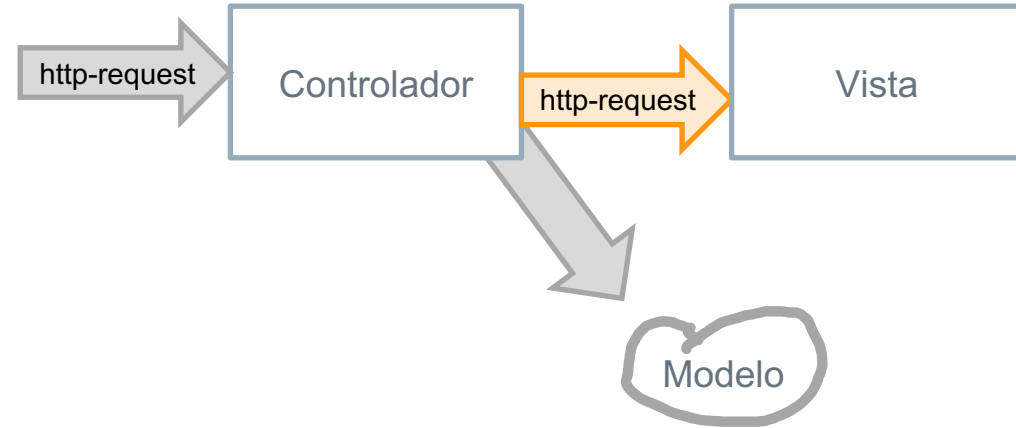
- ❑ La solicitud http-request siempre llega **primero al Controlador**
- ❑ Se recuperan los parámetros de la solicitud
- ❑ Se recupera la información de sesión, como **CustomerBean**
- ❑ Se realiza el control de privilegios y de seguridad (p.ej., mediante ACL)
- ❑ Se comprueban las restricciones y se ejecuta la lógica de negocio requerida
- ❑ El controlador no implementa ningún tipo de código de agente de usuario
- ❑ Se accede a fuentes de datos para recuperar toda la información necesaria

Esquema general



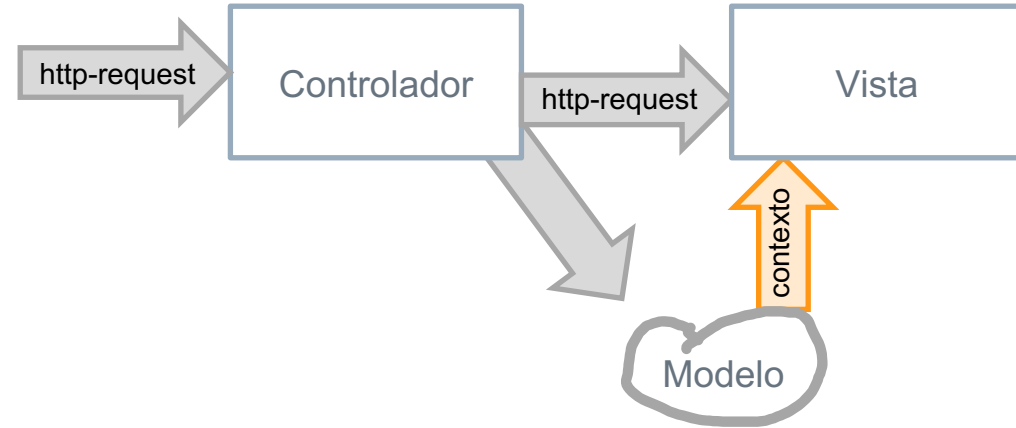
- ❑ A través de los DAO, los DTO se transfieren a modelos
- ❑ Los modelos se guardan en el **scope** necesario para su posterior recuperación
- ❑ En general, los modelos son *javabeans*

Esquema general



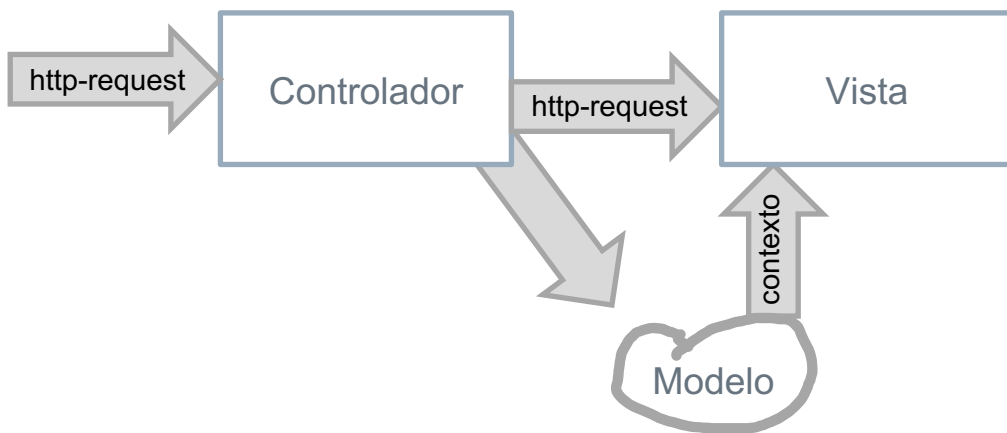
- ❑ El flujo de navegación se redirige a la vista mediante un **`jsp:forward`**
- ❑ En caso de datos sencillos o requeridos en la `http-request`, se incluirán parámetros en la solicitud mediante **`jsp:param`**

Esquema general



- ❑ En caso de datos más complejos, o datos contenidos en modelos, la vista accederá a ellos a través de los objetos implícitos (p.ej., **session**) o de *javabeans* (**jsp:useBean**)
- ❑ Se realiza el control de privilegios y de seguridad (p.ej., mediante ACL)

Esquema general

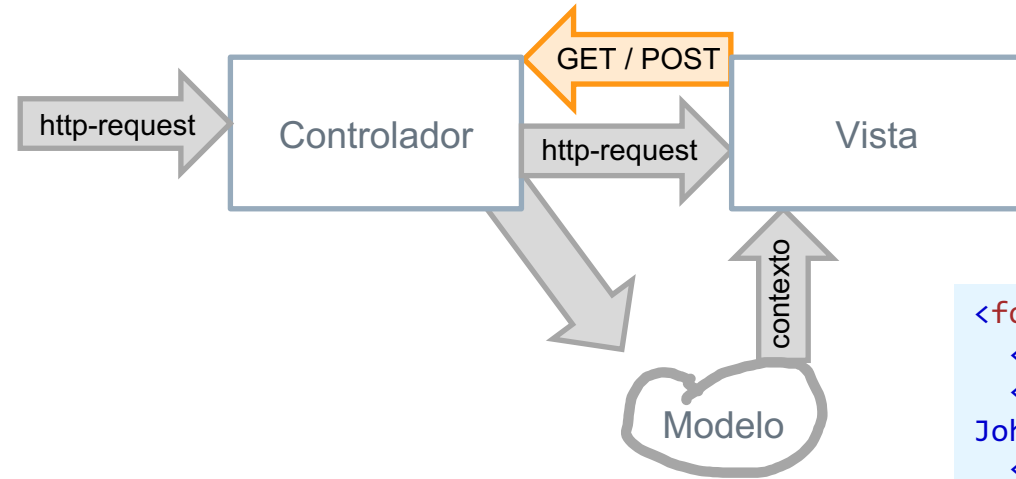


- ❑ La vista muestra toda la información necesaria a partir de parámetros y modelos
- ❑ La interfaz de usuario se construye conforme a lo implementado en HTML5, CSS3 y Javascript
- ❑ Las interacciones del usuario generan eventos cuya lógica de respuesta puede estar implementada en el cliente (vista)
- ❑ Puede contener llamadas asíncronas que, según el caso, requerirían de la intermediación (o no) del controlador
- ❑ El usuario podrá continuar la navegación desde la vista de 2 formas habituales:
 - Mediante enlace a otro controlador (****)
 - Mediante paso de datos al controlador (**<form method="post" ...>**)

Se recuerda que el modelo presentado es genérico y agnóstico de cualquier *web framework*

En caso de utilizar web frameworks, sean *full-stack* / *front-end* / *back-end*, estos WF impondrán restricciones que podrán modificar la estructura del patrón MVC, así como su secuencia de uso. Igualmente, los WF tomarán el control de la lógica de la aplicación y podrán generar automáticamente partes del código, haciendo al desarrollador transparente la implementación de este (MVC) y otros patrones (DTO, DAO, localizadores de servicios, etc.)

Esquema general



- ❑ Los formularios de HTML son los mecanismos más habituales para la devolución de datos del usuario al controlador

```
<form method="post" action=".../loginController.jsp">
  <label for="login">Login:</label><br>
  <input type="text" id="login" name="login" value="
JohnDoe"><br>
  <label for="passwd">Passwd:</label><br>
  <input type="password" id="passwd" name="lname" va
lue="Doe"><br><br>
  <input type="submit" value="Submit">
</form>
```

Esquema general

- ▷ Los **objetos DAO** encapsulan el acceso a los datos, independientemente de la fuente en la que se almacenen, y los ponen en disposición de la capa de lógica (*business*)
- ▷ El acceso es **transparente al cliente** (generalmente en capa *business*)
- ▷ Los DAO **no tienen que corresponderse exactamente con tablas** o ficheros
 - ❑ Se puede crear un objeto similar a *User* (tabla de la base de datos) pero que se corresponda con una vista de los datos que nos interese contemplar
 - ❑ También pueden ser **vistas de negocio** de los datos: por ejemplo, podemos tener una clase de objetos **Purchase**, que recoja información de los productos, de los usuarios, del proceso de compra, etc. El acceso a las tablas de la base de datos implicadas, la obtención de los datos y el parseado lo realizaría la clase **PurchaseDAO**, que gestionaría la transacción

6.

Recomendaciones de diseño

**Ideas básicas a considerar en el diseño
y desarrollo de la aplicación web**

➤ Se recomienda el uso estricto del **patrón arquitectónico MVC**:

➤ **Modelo:**

- ❑ Un *JavaBean* que contiene los **datos de la funcionalidad**, accediendo mediante **DAOs a base de datos** para la recuperar y salvaguardar los mismos desde el controlador (manteniendo el control de la secuencia de invocaciones a capa de datos en la capa de negocio)
- ❑ Las **consultas SQL** se recomienda que se declaren en un **fichero de propiedades** externo, de modo que –si cambia el gestor de base de datos o la versión– se pueda actualizar el acceso a datos sin necesidad de recompilar el código Java

➤ **Controlador:**

- ❑ Al principio del controlador se puede **comprobar si hay parámetros para controlar el flujo de ejecución**. Por ejemplo, si no hay parámetros, viene de otra funcionalidad o de `index.jsp`; si hay parámetros (p.ej. puede ser de un campo de tipo *hidden*) viene de una vista
- ❑ Se verifica la **lista de control de acceso**, de modo que sólo los usuarios con rol permitido para la funcionalidad, pueden acceder
- ❑ No debe incluirse **ningún tipo de salida a la interfaz** en el controlador

- Se recomienda el uso estricto del **patrón arquitectónico MVC**:
 - **Vista(s)**:
 - ❑ Únicamente obtienen sus datos de los *JavaBeans* (modelos) creados desde el controlador
 - ❑ En ningún caso invocan a la base de datos ni realizar funciones de la capa de negocio
 - **Variantes**:
 - ❑ En ocasiones, algunas aplicaciones recomiendan la **modificación del patrón MVC** para adaptarlo a necesidades específicas del sistema
 - ❑ Puede requerirse que los **beans sean lo más ligeros posible**, en cuyo caso conviene que el acceso a la base de datos lo realice el controlador (objetos DAO) e inmediatamente instancie el *bean* con los resultados obtenidos de invocar a los métodos de estos objetos
 - ❑ Es importante mantener siempre la **coherencia en el uso** en toda la aplicación

- Se deben **capturar las excepciones** para que no se propaguen y, en cualquier caso, deben **controlarse siempre las páginas de error** a las que se dirige el flujo de la aplicación
 - ❑ Se recomienda que **el usuario siempre pueda recuperar el control** de la navegación, incluso después de haber incurrido en un error
 - ❑ En el fichero `web.xml` también puede indicarse la página de error para cada tipo de mensaje de error HTTP o para errores genéricos:

```
<web-app>
[...
  <error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/error.jsp</location>
  </error-page>
[...
</web-app>
```

➤ **Estructure el código** adecuadamente y conforme a las recomendaciones

Ejemplo de estructura:

webapp:

```
\
  index.jsp
  acercaDe.html
\include
  header.jsp
  validation.jsp
  errorpage.jsp
\mvc
  \control
    \user
      loginController.jsp
      editProfileController.jsp
    \alert
      addNewAlertController.jsp
      displayAlertDetailController.jsp
      ...
  \view
    \user
      loginView.jsp
      loginViewFailure.jsp
      editProfileView.jsp
    \alert
      ...
```

src:

```
es.uco.pw.
  display.
    javabean.
      CustomerBean.java
      LoginBean.java
      AlertDetailBean.java
  business.
    common.
      PropertyMgr.java
      PropertyNotFoundException.java
  user.
    User.java
    UserMgr.java
    InterestMgr.java
  bulletin.
  ...
  data.
  dao.
    common.
      ConexionBD.java
      DAO.java
      DAOException.java
  user.
    DAOUser.java
  bulletin.
    DAOAlert.java
    DAOBulletin.java
  ...
```



Programación Web

Presentación de la asignatura__ **Curso 2020/21**