Desarrollo Web en entorno servidor

Unidad 3:
Aplicaciones Web en PHP
Extensiones. Manejo de bases de datos.

Tabla de contenido

1. Introducción.

En esta última parte del tema 3 vamos a ver la aplicación de la programación orientada a objetos para poder enviar correos electrónicos, manipular documentos XML y trabajar con bases de datos. Para ello, introduciremos la biblioteca (librería) phpMailer y las extensiones Simple-XML y PDO.

Las extensiones[1] son módulos que proporcionan alguna funcionalidad concreta al motor de PHP. La mayoría de funciones **y** clases predefinidas vienen de extensiones que vienen ligadas a las diferentes versiones de PHP. Muchas de estas extensiones vienen compiladas con cada versión de PHP, de forma que no pueden eliminarse, como las correspondientes a las funciones SPL PHP: Funciones SPL - Manual

Existen extensiones que, por defecto, no están habilitadas, lo cual puede hacerse a través del archivo php.ini. También hay otras que no están incluidas directamente, pero pueden ser descargadas a través de la comunidad que las gestiona: PHP Extension Community Library PECL. Una vez que la extensión está instalada (lo que significa que tenemos el archivo DLL o .so en el directorio de extensiones PHP), necesita ser activada.

Podemos ver el listado de extensiones habilitadas ejecutando el siguiente código.

```php
<?php
echo "<pre>";
print_r(get_loaded_extensions());
echo "</pre>";
?>
```

Por otra parte, existen bibliotecas que podemos utilizar y que no están incluidas en php. Es el caso de la biblioteca phpMailer. Para gestionar este tipo de bibliotecas en nuestro proyectos es habitual hacer uso de herramientas como Composer[2]. Composer nos permite gestionar las dependencias en PHP. Nos permite declarar las bibliotecas de las que dependen nuestros proyectos y las administrará (instalará / actualizará) por nosotros.

Para profundizar más en el tema de las extensiones en PHP, visitar Extensiones en PHP

2. La biblioteca phpMailer

En PHP existen funciones que no permiten enviar correos directamente, como la función `mail()`, sin embargo es habitual utilizar bibliotecas que nos permitan adecuar el formato debido a las limitaciones de dicha función.

La biblioteca más habitual es phpMailer. Para instalarla podemos utilizar Composer o descargarnos directamente los archivos que la componen a través de su página de GIThub GitHub - PHPMailer/PHPMailer: The classic email sending library for PHP

[1] Para profundizar más en el tema de las extensiones en PHP, visitar Extensiones en PHP

[2] Podemos instalar Composer y acceder a su documentación a través del siguiente enlace: Composer (getcomposer.org)

De cualquier forma, aunque técnicamente sea posible configurar nuestro servidor SMTP local para enviar correos electrónicos, el hecho de no estar registrado correctamente hará que los filtros antispam impidan el envío de los mismos.

Si no disponemos de un servicio de correo en Internet, la opción más cómoda para enviar un correo electrónico desde nuestra aplicación web es a través de una cuenta de Gmail, ya que desde estas cuentas tenemos la opción de activar la opción “Permitir aplicaciones menos seguras” en la sección de ajustes de la cuenta.

Por ello, no vamos a realizar este tipo de configuración durante el curso, limitándonos a exponer el código de ejemplo para enviar correos electrónicos haciendo uso de esta biblioteca. Lo podemos ver en ejemplo_correo.php.

```
 1:  1 <?php
 2:
 3:     use PHPMailer\PHPMailer\PHPMailer;
 4:
 5:     require "vendor/autoload.php";
 6:     $mail = new PHPMailer();
 7:     $mail->IsSMTP();
 8:     // cambiar a 0 para no ver mensajes de error
 9:     $mail->SMTPDebug  = 2;
10:     $mail->SMTPAuth   = true;
11:     $mail->SMTPSecure = "tls";
12:     $mail->Host       = "smtp.gmail.com";
13:     $mail->Port       = 587;
14:     // introducir usuario de google
15:     $mail->Username   = "";
16:     // introducir clave
17:     $mail->Password   = "";
18:     $mail->SetFrom('user@gmail.com', 'Test');
19:     // asunto
20:     $mail->Subject    = "Correo de prueba";
21:     // cuerpo
22:     $mail->MsgHTML('Prueba');
23:     // adjuntos
24:     $mail->addAttachment("empleado.xsd");
25:     // destinatario
26:     $address = "destino@servidor.com";
27:     $mail->AddAddress($address, "Test");
28:     // enviar
29:     $resul = $mail->Send();
30:     if (!$resul) {
31:         echo "Error" . $mail->ErrorInfo;
32:     } else {
33:         echo "Enviado";
34:     }
```

Las líneas 3, 4 y 5 las podemos reemplazar por las páginas php específicas de la biblioteca phpMailer mínimas para que este ejemplo funcione, las cuales son

```
3:  include "PHPMailer.php";
```

```
4:  include "SMTP.php";
5:  include "exception.php";
```

En la línea 5 del ejemplo vemos la referencia a incluir una carpeta, vendor, y un documento, autoload.php. Tanto la carpeta como el documento php son creados por Composer en nuestro proyecto si hemos realizado adecuadamente los pasos para su instalación.

Por otra parte, en la línea 3 vemos la sentencia: `use PHPMailer\PHPMailer\PHPMailer.` La palabra reservada `use` nos permite definir los llamados “Espacios de nombres”.

Por ejemplo, si creamos una función llamada db_connect, y el código de alguien más que usamos en nuestra paágina (por ejemplo, cuando usamos include) tiene una función con ese mismo nombre se produciría una colisión de nombres.

Para solucionar ese problema, tenemos varias opciones. Una opción es cambiar el nombre a nuestra función, por ejemplo a Mi_funcion_db_connect, esto hace que nuestro código vaya ganando cada vez más complejidad, sea más largo y más difícil de leer.

Los espacios de nombres buscan solucionar este problema. Con ellos podemos mantener el nombre de nuestra función separado del nombre de la función de cualquier otra persona. Lo podemos ver en los siguientes ejemplos extraídos de la documentación de php.

Espacio_nombres_1.php

```
 1:  <?php
 2:
 3:  namespace Foo\Bar\subespacio_de_nombres;
 4:
 5:  const FOO = 1;
 6:  function foo()
 7:  {
 8:  }
 9:  class foo
10:  {
11:      static function método_estático()
12:      {
13:      }
14:  }
```

Espacio_nombres_2.php

```
 1:  <?php
 2:
 3:  namespace Foo\Bar;
 4:
 5:  include 'espacio_nombres_1.php';
 6:
 7:  const FOO = 2;
 8:  function foo()
 9:  {
10:  }
11:  class foo
12:  {
```

```
13:       static function método_estático()
14:       {
15:       }
16:   }
17:
18:   /* Nombre no cualificado */
19:   foo(); // se resuelve con la función Foo\Bar\foo
20:   foo::método_estático(); // se resuelve con la clase Foo\Bar\foo, método
método_estático
21:   echo FOO; // se resuelve con la constante Foo\Bar\FOO
22:
23:   /* Nombre cualificado */
24:   subespacio_de_nombres\foo(); // se resuelve con la función
Foo\Bar\subespacio_de_nombres\foo
25:   subespacio_de_nombres\foo::método_estático(); // se resuelve con la clase
Foo\Bar\subespacio_de_nombres\foo,
26:   // método método_estático
27:   echo subespacio_de_nombres\FOO; // se resuelve con la constante
Foo\Bar\subespacio_de_nombres\FOO
28:
29:   /* Nombre conmpletamente cualificado */
30:   \Foo\Bar\foo(); // se resuelve con la función Foo\Bar\foo
31:   \Foo\Bar\foo::método_estático(); // se resuelve con la clase Foo\Bar\foo,
método método_estático
32:   echo \Foo\Bar\FOO; // se resuelve con la constante Foo\Bar\FOO
33:
```

Para más información, consultar la documentación de php: [PHP: Espacios de nombres - Manual]

3. Documentos estructurados: XML

XML es un lenguaje de marcas similar a HTML. Ambos tipos de documentos producen documentos con características parecidas: ambos contienen texto sin formato y etiquetas delimitadas por los símbolos < y >.

Existen dos grandes diferencias entre XML y HTML:

- En XML no se definen los conjuntos específicos de etiquetas que se deben utilizar.
- XML es extremadamente exigente con la estructura del documento.

Debido a esta primera diferencia, XML nos da mucha más libertad que HTML en cuanto a la estructura de los documentos, ya que HTML tiene un determinado conjunto de etiquetas, por ejemplo: las etiquetas <a> rodean un enlace, el párrafo <p> comienza, etc.

En los documentos XML podemos utilizar las etiquetas que deseemos, teniendo en cuenta que siempre es obligatorio cerrarlas. Por ejemplo, podemos utilizar las etiquetas <nombre> y <apellidos>, con sus correspondientes cierres </nombre> y </apellidos>, para guardar los nombres y apellidos de diferentes empleados.

La sintaxis básica de un documento XML es la siguiente:

```
<nombre_del_elemento atributo1 atributo2>
… contenido …
</nombre_del_elemento>
```

Donde

- nombre_del_elemento es, como su nombre indica, el nombre del elemento o etiqueta.
- atributo1, atributo2 son atributos del elemento separados por espacios en blanco. Un atributo define una propiedad del elemento. Es decir, se asocia un nombre con un valor, que es una cadena de caracteres.

Cuando un elemento no tiene contenido, hablamos del elemento vacío y podemos sustituir las etiquetas de inci y cierre con una etiqueta que termine en />. Por ejemplo, `<nombre></nombre>` equivale a `<nombre/>`

Veamos el documento empleados.xml

```
 1:  <?xml version="1.0" encoding="UTF-8"?>
 2:  <departamento nombre="Marketing" jefeDep="e101">
 3:     <empleado codEmple="e101">
 4:         <nombre>Ana</nombre>
 5:         <apellidos>Frutos Jarama</apellidos>
 6:         <edad>28</edad>
 7:         <carnetConducir/>
 8:     </empleado>
 9:     <empleado codEmple="e102">
10:         <nombre>Antonio</nombre>
11:         <apellidos>Miguel Bustos</apellidos>
12:         <edad>23</edad>
13:     </empleado>
14:  </departamento>
```

Uno de los ejemplos más paradigmático de uso de este tipo de documentos lo encontramos en la plataforma de contratación del estado:

Plataforma de Contratación del Sector Público (contrataciondelestado.es)

Para poder trabajar en PHP con este tipo de documentos hacemos uso de la extensión SimpleXML. Podemos encontrar toda su información en https://www.php.net/manual/es/book.simplexml.php

Básicamente, esta extensión nos proporciona acceso estructurado al documento XML, permitiéndonos filtrar y recorrer estos documentos gracias a convertirlos a objetos que podemos manipular.

SimpleXML define la clase SimpleXMLElement. Para crear un objeto de esta clase a partir de un documento XML, pasamos el nombre del documento XML a la función `simplexml_load_file()`, la cual devuelve el objeto de la clase.

Veamos el ejemplo `xml_simplexml_con.php`.

```
1:  <?php
2:     $datos = simplexml_load_file("empleados.xml");
3:     if($datos===false){
4:         echo "Error al leer el fichero";
5:     }
```

```
    echo "<pre>";
    foreach($datos as $valor){
        print_r($valor);
        echo "<br>";
    }
    echo "</pre>";
```

El método xpath() permite seleccionar ciertos elementos usando una expresión XPath. En el siguiente ejemplo, xml_xpath.php se seleccionan solo los elementos <edad>.

```
<?php
    $datos = simplexml_load_file("empleados.xml");
    $edades = $datos->xpath("//edad");
    foreach($edades as $valor){
        print_r($valor);
        echo "<br>";
    }
```

3.1. Validación de XML. Esquemas XSD

Para comprobar la validez de la estructura de los documentos XML tenemos dos herramientas. Los documentos DTD y los documentos XSD.

Los documentos DTD pueden encontrarse al comienzo de los archivos XML o en documentos aparte con extensión .dtd. Veamos el siguiente ejemplo:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE address [
   <!ELEMENT address (name,company,phone)>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT company (#PCDATA)>
   <!ELEMENT phone (#PCDATA)>
]>
<address>
   <name>Tanmay Patil</name>
   <company>TutorialsPoint</company>
   <phone>(011) 123-4567</phone>
 </address>
```

Los esquemas XSD son documentos XML que contienen la estructura del documento, definiendo de forma adecuada los elementos y atributos que puede contener el XML original, el tipo de contenido simple si no tiene elementos descendientes o complejo si tiene elementos descendientes), así como las características de estos elementos complejos (orden de aparición, número de ocurrencias, grupos, restricciones, …).

Veamos el ejemplo departamento.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="departamento">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="empleado" maxOccurs="unbounded"/>
```

```
 7:         </xs:sequence>
 8:         <xs:attribute name="jefeDep" type="xs:string" use="required"/>
 9:         <xs:attribute name="nombre" type="xs:string" use="required"/>
10:       </xs:complexType>
11:     </xs:element>
12:     <xs:element name="empleado">
13:       <xs:complexType>
14:         <xs:sequence>
15:           <xs:element name="nombre" type="xs:string"/>
16:           <xs:element name="apellidos" type="xs:string"/>
17:           <xs:element name="edad" type="xs:string"/>
18:           <xs:element name="carnetConducir" minOccurs = "0" type="xs:string"/>
19:         </xs:sequence>
20:         <xs:attribute name="codEmple" type="xs:string" use="required"/>
21:       </xs:complexType>
22:     </xs:element>
23:   </xs:schema>
```

Ejercicio 1

Comparar el documento departamento.xsd y el documento empleados.xml para ver las diferencias entre ambos y entender la estructura de departamento.xsd.

Para más información sobre esquemas XSD visitar https://www.ticarte.com/contenido/que-son-los-esquemas-xsd

Para validar el empleados.xml con departamento.xsd usamos el método `schemavalidate()` de la clase `DOMDocument` definida en la extensión DOM. Veamos el ejemplo xml_validacion.php

```
 1:  <?php
 2:     $dept = new DOMDocument();
 3:     $dept->load( 'empleados.xml' );
 4:     $res = $dept->schemaValidate('departamento.xsd');
 5:     if ($res){
 6:        echo "El fichero es válido";
 7:     }
 8:     else {
 9:        echo "Fichero no válido";
10:     }
```

DOM son las siglas de Document Object Model y en PHP nos permite manipular documentos XML según el estándar DOM.

3.2. Transformaciones

También es posible utilizar transformaciones XSLT con la extensión XSL, la cual implementa el estándar XSL. Este estándar nos permite definir adecuadamente hojas de estilo para los documentos XML y, combinando ambos, generar páginas HTML.

De forma resumida, una hoja de estilos XSL es un documento XML con extensión .xsl, en el que se define las transformaciones a realizar en el documento XML de partida, mediante una serie de elementos y atributos.

La hoja de estilo incluye el elemento stylesheet, que declara el espacio de nombres para poder utilizar los elementos y atributos XSL: <xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">

El elemento utilizado en la hoja de estilos para realizar las transformaciones, es la plantilla.

La plantilla se define utilizando el elemento xsl:template, el cual tiene el atributo match, que indica la rama del árblol XML sobre la que se aplica la plantilla, pudiendo seleccionar el árbol completo usando simplemente "/". Ver la línea 3 del siguiente ejemplo.

Veamos la plantilla departamento.xslt.

```
 1:  <?xml version="1.0" encoding="UTF-8"?>
 2:  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
 3:    <xsl:template match="/">
 4:      <html>
 5:        <head>
 6:          <title>Empleados del departamento</title>
 7:        </head>
 8:        <body>
 9:          <table>
10:            <tr>
11:              <th>Nombre</th>
12:              <th>Apellidos</th>
13:              <th>Edad</th>
14:            </tr>
15:            <xsl:for-each select="departamento/empleado">
16:              <tr>
17:                <td>
18:                  <xsl:value-of select="nombre"/>
19:                </td>
20:                <td>
21:                  <xsl:value-of select="apellidos"/>
22:                </td>
23:                <td>
24:                  <xsl:value-of select="edad"/>
25:                </td>
26:              </tr>
27:            </xsl:for-each>
28:          </table>
29:        </body>
30:      </html>
31:    </xsl:template>
32:  </xsl:stylesheet>
```

Podemos ver otros elementos propios de este tipo de documento, como en la línea 15, `<xsl:for-each select="">` o en las líneas 18, 21 y24 `<xsl:value-of select="">`

Ejercicio 2
Comparar el documento departamento.xslt y el documento empleados.xml para ver las diferencias entre ambos y entender la estructura de departamento.xslt.

El ejemplo xml_tranformacion.php utiliza una transformación XSLT para mostrar un fichero XML como una tabla HTML Tanto la transformación como el fichero XML se cargan utilizando la clase DOMDocument. La transformación se realiza con un objeto de la clase XSLTProcessor que se asocia a la transformación usando el método `importStylsheet()`.

```
 1:  <?php
 2:     // cargar el fichero a transformar
 3:     $dept = new DOMDocument();
 4:     $dept->load( 'empleados.xml' );
 5:     // cargar la transformacion
 6:     $transformacion = new DOMDocument();
 7:     $transformacion->load( 'departamento.xslt' );
 8:     // crear el procesador
 9:     $procesador = new XSLTProcessor();
10:     // asociar el procesador y la transformacion
11:     $procesador-> importStylesheet($transformacion) ;
12:     // transformar
13:     $transformada = $procesador->transformToXml($dept);
14:     echo $transformada;
```

Sin embargo, esta extensión no viene activada por defecto en PHP, por lo que hemos de configurar adecuadamente el archivo php.ini. Para ello, abrimos dicho archivo y buscamos ***;extension=xsl***

Borramos el punto y coma inicial, guardamos el archivo y reiniciamos el servidor para que la configuración se aplicada.

4. Bases de datos. La extensión PDO

Una de las aplicaciones más frecuentes de PHP es generar una interface web para acceder y gestionar la información almacenada en una base de datos. Usando PHP podemos mostrar en una página web información extraída de la base de datos, o enviar sentencias al gestor de la base de datos para que elimine o actualice algunos registros.

PHP soporta más de 15 sistemas gestores de bases de datos (SGBD): SQLite, Oracle, SQL Server, PostgreSQL, IBM DB2, MySQL, etc.

El acceso a estas bases de datos se puede realizar de manera específica, con una extensión propia para cada gestor de base de datos, **método habitualmente desaconsejado y en muchos casos obsoleto,** o bien mediante una única extensión que nos permite de forma sencilla cambiar de gestor de base de datos, la extensión PDO.

Utilizando PDO no podemos olvidarnos completamente del SGBD utilizado, pero la mayor parte del código es independiente del SGBD y sólo en algunas partes del programa (en la conexión con el SGBD o en la creación de tablas, por ejemplo) el código es específico del SGBD.

Pese a la multitud de problema que resuelve la capa de abstracción aportada por la extensión PDO, no resuelve del todo la conversión de los objetos definidos en la aplicación en entidades que se pueden almacenar en una base de datos relacional.

Para ello, existen capas de abstracción de nivel superior a PDO y que se engloban bajo el acrónimo ORM (Object-relational mapping, mapeo objeto-relacional), las cuales trataremos en los temas

dedicados al framework Symfony y su ORM Doctrine. Estas capas permiten, en teoría, olvidarse completamente del SGBD ya que son capaces de reescribir las sentencias SQL o simular características no presentes en el SGBD.

4.1. PHPmyadmin y MYSQL

Para crear nuestra primera base de datos vamos a abrir phpMyAdmin.

En cambio, localmente (en nuestro servidor de pruebas) podremos crear una nueva base de datos para cada proyecto en el que estemos trabajando.

Para crear una nueva base de datos, dentro del phpMyAdmin escribiremos (en la zona que se resalta a continuacion) el nombre que le queremos dar. Vamos a denominarla "empresa"):

Figura 1.- Pestaña para crear bases de datos en phpMyAdmin

Pulsamos el **botón Crear** y, a continuación, el nombre de la base recién creada aparecerá en la **columna de la izquierda**, debajo del menú de selección que nos muestra todas las bases de datos que tengamos en nuestro servidor, así como también aparece el nombre de la base de datos activa en la ruta superior que siempre nos indica donde estamos ubicados.

Si accedemos a C:/xampp/mysql/data/ y allí encontraremos una carpeta por cada base de datos que hayamos creado; en este caso, vemos, al lado de las bases que vienen por defecto, nuestra nueva base "empresa".

Para dotar de contenido nuestra base de datos podemos trabajar manualmente para crea cada una de las tablas que debe contener.

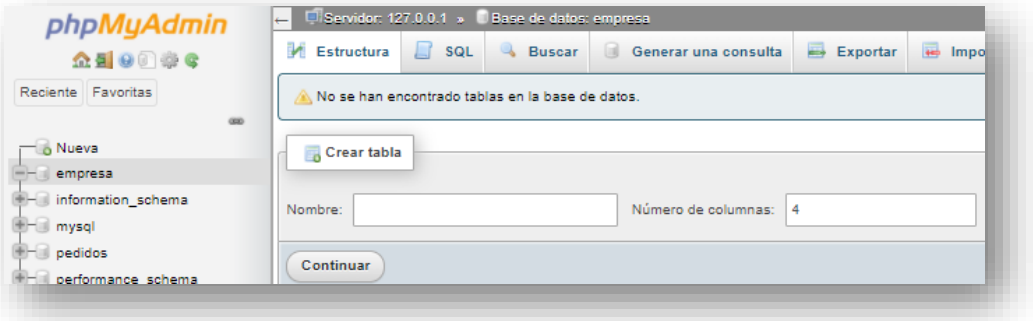

Figura 2.- Creación de tablas

Vamos a crear una tabla de nombre ***usuarios*** con cuatro campos o columnas:

- **Código**: es la clave primaria, un campo autonumérico
- **Nombre**: es el nombre de usuario y está marcado como unique
- **Clave**: es una cadena con la clave de acceso al sistema.
- **Rol**: es un entero que representa el rol del usuario dentro del sistema.

También podríamos haber usado la pestaña SQL y, con las instrucciones adecuadas, haber generado la tabla usuarios.

```
CREATE TABLE `empresa`.`usuarios2` (
`Código` INT NOT NULL AUTO_INCREMENT ,
`Nombre` VARCHAR(65) NOT NULL ,
`Clave` VARCHAR(65) NOT NULL ,
`Rol` INT NOT NULL ,
PRIMARY KEY (`Código`), UNIQUE (`Nombre`)
) ENGINE = InnoDB;
```

El código anterior lo podemos obtener si seleccionamos el botón “Previsualizar SQL” en la misma pestaña donde hemos introducido los campos de la tabla

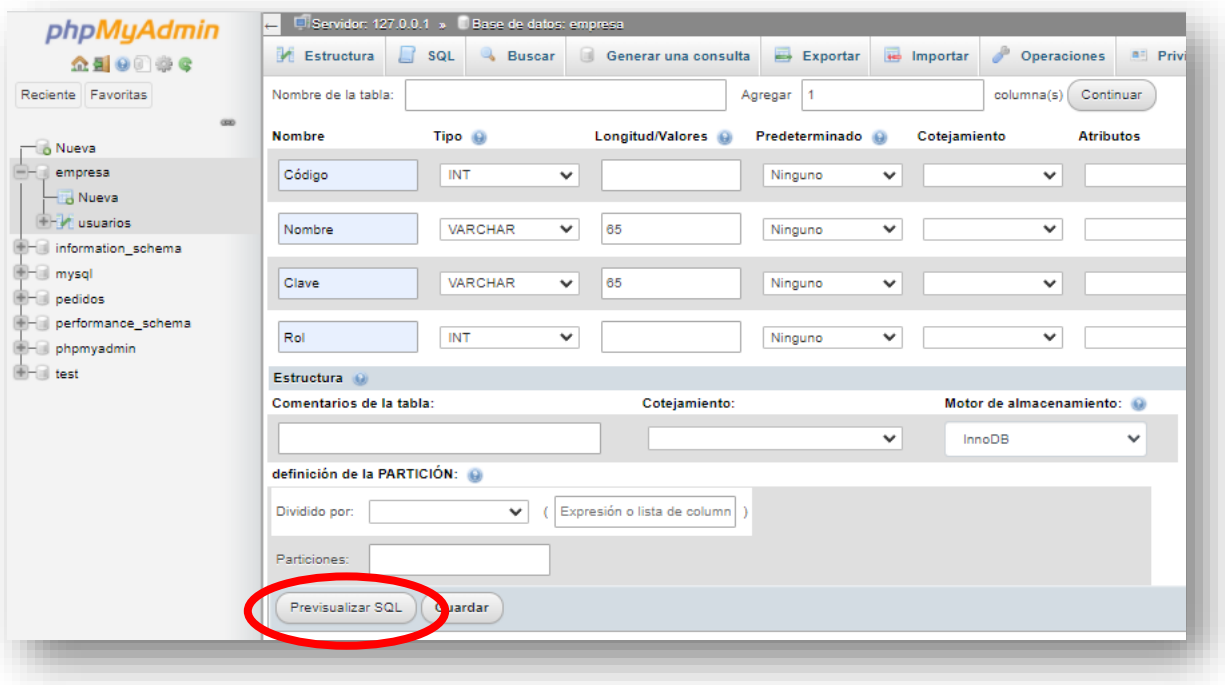

Figura 3.- Previsualizar SQL

En la última línea del código SQL generado, nos aparece la sentencia “`ENGINE = INNODB;`“. Engine hace referencia al motor de almacenamiento que gestiona cada una de las tablas de la base de datos. Existen varios motores en MySQL, los podemos ver desde phpMyAdmin haciendo clic en la pestaña “Motores”. Cada uno tiene sus propias características.

El motor por defecto al crear una tabla es INNODB.

Si necesitamos cambiar el motor de almacenamiento de una tabla basta con seleccionarla en nuestra base de datos, y, haciendo clic en operaciones, seleccionamos en la sección “Opciones de tabla” el motor correspondiente.

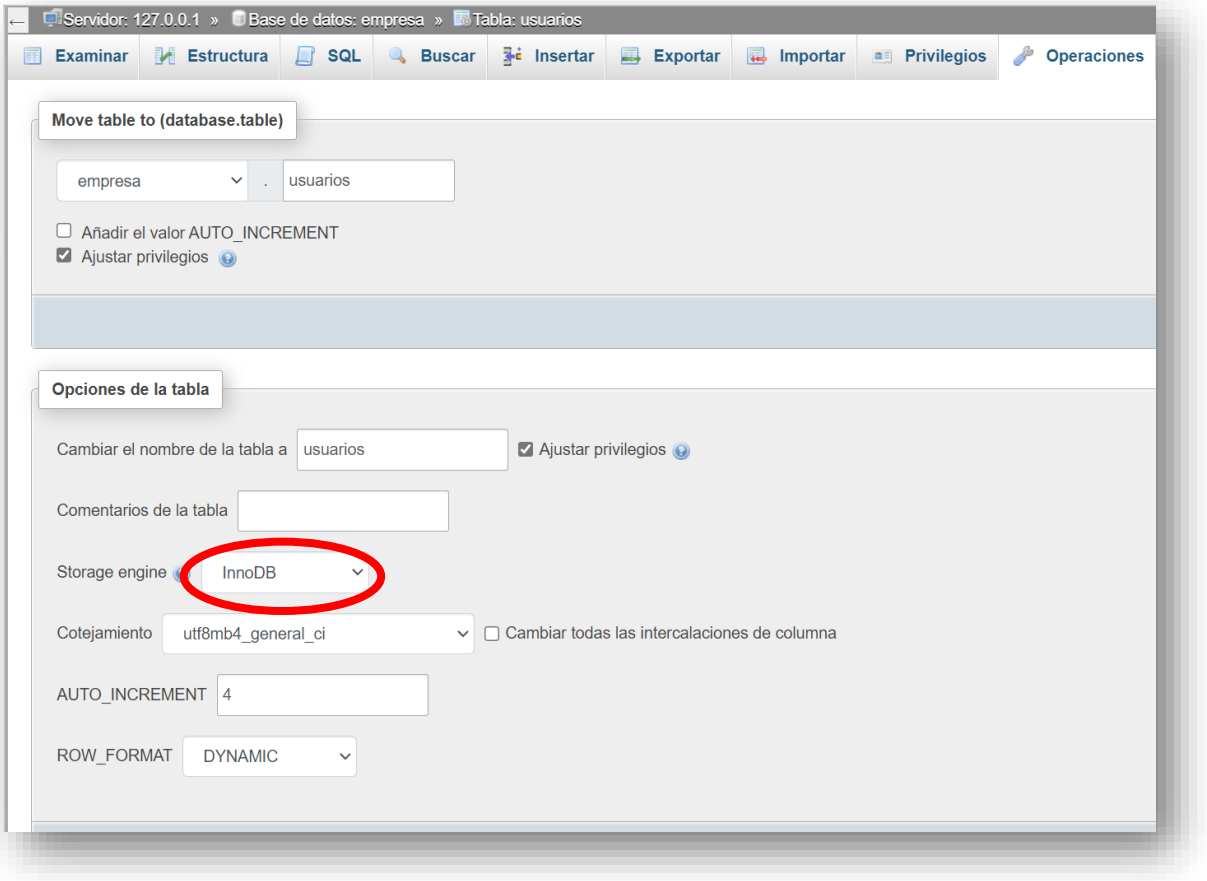

Figura 4.- Cambio de motor de almacenamiento predeterminado

Por último, vamos a incorporar algunos datos a nuestra tabla de usuarios. Para ello nos dirigimos a la pestaña SQL, introducimos las siguientes sentencias,

```
INSERT INTO empresa.usuarios(nombre, clave, rol) VALUES ('Carlos', '11111', '0');
INSERT INTO empresa.usuarios(nombre, clave, rol) VALUES ('Alejandro', '22222', '0');
INSERT INTO empresa.usuarios(nombre, clave, rol) VALUES ('Marcos', '33333', '0');
```

y hacemos clic en continuar.

Las sentencias anteriores son equivalentes a:

```
INSERT INTO empresa.usuarios(nombre, clave, rol) VALUES ('Carlos', '11111',
'0'),('Alejandro', '22222', '0'),('Marcos', '33333', '0');
```

Sin embargo, para que no haya problemas a la hora de copiar y pegar en la pestaña de SQL y para facilitar su lectura, las mostramos en tres instrucciones SQL separadas.

4.2. La clase PDO

A lo largo de esta sección utilizaremos de manera específica la clase PDO perteneciente a la extensión PDO.

4.2.1. Conexión a la base de datos

El primer paso para trabajar con una base datos es obtener una conexión a la misma. Para presentar la conexión se usa un objeto de clase PDO. El constructor es:

`public PDO::__construct(string $dsn [, string $username [, string $passwd [, array $options]]])` El primer parámetro es una cadena que especifica qué driver hay que usar, la localización y el nombre de la base de datos. Para una base de datos MySQL se usaria:

`mysql:dbname=<base de datos>;host=<ip o nombre>`

Los siguientes parámetros son el nombre de usuario y clave para acceder a la base de datos. El último parámetro, opcional, es un array de opciones.

Si se puede establecer la conexión, se usará el nuevo objeto PDO para manejar la base de datos. Si no se puede conectar con la base de datos, el constructor lanza una excepción PDOException.

Veamos el ejemplo bd_conect.php

```
 1:  <?php
 2:  $cadena_conexion = 'mysql:dbname=empresa;host=127.0.0.1';
 3:  $usuario = 'root';
 4:  $clave = '';
 5:  try {
 6:      $bd = new PDO($cadena_conexion, $usuario, $clave);
 7:      echo 'Conectado';
 8:  } catch (PDOException $e) {
 9:      echo 'Error con la base de datos: ' . $e->getMessage();
10:  }
```

En principio, al acabar el script se cierra la conexión a la base de datos, pero también es posible usar conexiones persistentes, que no se cierran automáticamente al terminar el script.

Quedan abiertas y si se vuelven a utilizar no es necesario reestablecer la conexión, por lo pueden ser más eficientes en determinadas ocasiones. Para crear una conexión persistente se utiliza la opción PDO::ATTR_PERSISTENT en el constructor:

`$bd=new PDO($cadena_conexion, $usuario, $clave, array(PDO::ATTR_PERSISTENT => true));`

4.2.2. Recuperación y presentación de datos

El método `query($cad)` de la clase PDO ejecuta la cadena que recibe como argumento en la base de datos, como se haría desde la línea de comando SQL. El argumento `$cad` tiene que ser una instrucción SQL válida. Devuelve `FALSE` si hubo algún error o un **objeto PDOStatement** si la cadena se ejecutó con éxito.

Vamos a ver qué información tiene el objeto que obtenemos al ejecutar una consulta sobre nuestra tabla usuarios. Para ello, usaremos el método `fetchAll` y la función `vardump()`.

Abrimos el script **bd_mostrando_consulta.php**

Visto esto, podemos preguntarnos si es posible encontrar un atributo en la clase `PDOStatement` que guarde el contenido de `fetchAll`, tal y como hemos hecho en la sección de programación

orientada a objetos. Sin embargo, si consultamos la documentación vemos que no existe ningún atributo para dicha clase.

No contentos con ello, puesto que sabemos que la documentación de php puede contener alguna imprecisión, podemos buscar el repositorio de php en GitHub, en particular el de la extensión PDO.

Accediendo a pdo_stmt.stub.php y a pdo_stmt.c vemos, que solo están declarado métodos en la clase `PDOStatement`. En particular, `fetchAll` se encuentra en la línea 1249 del segundo enlace.

También podemos ejecutar instrucciones SQL desde nuestro código con el método `exec()`. La diferencia con `query()` radica en que la primera nos devuelve, exclusivamente, el número de filas afectadas por la instrucción SQL. Si queremos obtener el número de filas afectadas con `query()` basta con llamar al método `rowCount()` desde el objeto PDOStatement que invoca a `query()`.

Por otra parte, si se trata de una consulta de selección, es posible recorrer las filas devueltas con un `foreach`. En cada iteración del bucle se tendrá una fila, representada como un array en que las claves son los nombres que aparecen en la cláusula Select.

Veamos el ejemplo **bd_listar_usuario_1.php**

```
1:  <?php
2:  $cadena_conexion = 'mysql:dbname=empresa;host=127.0.0.1';
3:  $usuario = 'root';
4:  $clave = '';
5:  try {
6:      $bd = new PDO($cadena_conexion, $usuario, $clave);
7:     echo "Conexión realizada con éxito<br>";
8:     $sql = 'SELECT nombre, clave, rol FROM usuarios';
9:     $usuarios = $bd->query($sql);
10:    echo "Número de usuarios: " . $usuarios->rowCount() . "<br>";
11:    foreach ($usuarios as $usu) {
12:        print "Nombre : " . $usu['nombre'];
13:        print " Clave : " . $usu['clave'] . "<br>";
14:    }
15:    } catch (PDOException $e) {
16:        echo 'Error con la base de datos: ' . $e->getMessage();
17:    }
```

Cuando en `foreach` hacemos la asignación (`$usuarios as $usu`) hemos de entenderlo como la sentencia "`$arr = $usuarios->fetchAll();`" es decir, guardamos el contenido del objeto en un array.

Ejercicio 3

Escribe un formulario que reciba el código de un usuario y muestre por pantalla todos sus datos recogidos en la tabla usuarios.

4.2.3. Sentencias SQL en PHP. Instrucciones preparadas.

A la de elaborar sentencias SQL en PHP hemos de tener especial cuidado en su construcción. Sabemos que una sentencia SQL válida ha de tener la estructura concreta de la operación que queramos realizar (SELECT, INSERT INTO, UPDATE, DELETE) pero, a mayores, hemos de escribir adecuadamente los valores con los que vamos a trabar en la consulta.

Por ejemplo, para seleccionar los datos de la tabla usuarios usamos la sentencia sql:

```
$sql = 'SELECT codigo, nombre FROM usuarios';
```

Si quisiéramos seleccionar los usuarios de código 1, usamos:

```
$sql = 'SELECT codigo, nombre FROM usuarios WHERE codigo = 1';
```

Pero, si seleccionamos los usuarios por nombre, el valor que usamos en la clausula WHERE ha de ir entre comillas:

```
$sql = 'SELECT codigo, nombre FROM usuarios WHERE codigo = 'Alberto'';
```

Sin embargo, esta última sentencia no es válida en php, marcando como error `Alberto''`.

Ocurre que php se encuentra con una cadena (`'SELECT … codigo = '`) que va seguida de otra cadena de caracteres sin entrecomillar (`Alberto'';`), con lo que no puede interpretar a que nos referimos.

Para solucionar esto podemos utilizar comillas dobles, las cuales hacen que las comillas simples se interpreten como un carácter más de la cadena y no como delimitadores de la cadena en sí.

```
$sql = "SELECT codigo, nombre FROM usuarios WHERE codigo = 'Alberto'";
```

Hemos de tener esto en cuenta cuando trabajemos con sentencias SQL en el que nuestros valores estén almacenados en variables. Por ejemplo:

```
$nombre = 'Alberto';
$sql = "SELECT codigo, nombre FROM usuarios WHERE codigo = '" . $nombre . "'";
```

Esta última sentencia, también puede escribirse como:

```
$sql = "SELECT codigo, nombre FROM usuarios WHERE codigo = '$nombre'";
```

ya que las comillas dobles toman las comillas simples como parte de la cadena principal y la variable `$nombre` queda reemplazada por su valor, `'Alberto'`.

Trabajar con consultas de este estilo está completamente desaconsejado ya que son muy susceptibles a la inyección de código SQL.

Por ejemplo, en el ejercicio 3, si utilizamos cualquiera de las expresiones anteriores para manejar las sentencias SQL, podemos inyectar código SQL haciendo uso de las comillas simples y -- , los cuales delimitan comentarios en SQL.

De esta forma, si un ataque introdujera como nombre *'; TRUNCATE usuarios;--* perderíamos los datos en la tabla usuarios, ya que la consulta sql que se resolvería sería la siguiente:

```
$sql = "SELECT nombre, clave FROM usuarios WHERE nombre = ''; TRUNCATE usuarios;--'";
```

De la misma forma, podría conseguir acceso a la aplicación añadiendo usuarios. Por ejemplo, si se introduce en el campo del nombre *'; INSERT INTO usuarios(nombre, codigo) VALUES (user, user);--*.

Para evitar las inyecciones de SQL, existen varias alternativas, como por ejemplo comprobar expresamente el tipo de dato recibido que se va a utilizar en la consulta. Funciones que nos ayudan a ello son: is_numeric, ctype_digit, sprintf, settype.

Sin embargo, la solución general es usar instrucciones SQL preparadas, las cuales nos permiten utilizar parámetros, los cuales serán utilizados apropiadamente en la consulta, impidiendo inyecciones SQL como las vistas anteriormente.

Se inicializan una sola vez con el método `prepare()` y luego se ejecutan las veces necesarias con `execute()`, con diferentes valores para los parámetros.

Las características de las instrucciones preparadas son las siguientes:

- Permiten reutilizar las consultas
- Previenen la inyección de código
- Mejoran el rendimiento

Hay dos opciones para indicar los parámetros de la consulta: **por posición** y **por nombre**. En el primer caso se utiliza el **símbolo de interrogación** para indicar un parámetro. Al ejecutarla **se asocian por orden** los símbolos de interrogación con los **valores del array** que se pasa argumento a `execute()`

```
/* consulta preparada, parametros por orden */
$preparada = $bd->prepare("select nombre from usuarios where rol = ?");
$preparada->execute( array(0));
echo "Usuarios con rol 0: " .  $preparada->rowCount() . "<br>";
foreach ($preparada as $usu) {
    print "Nombre : " . $usu['nombre'] . "<br>";
}
```

Si se utilizan nombres en la instrucción preparada se han de usar esos nombres como **claves** del array de `execute()`.

```
/* consulta preparada, parametros por nombre */
$preparada_nombre = $bd->prepare("select nombre from usuarios where rol = :rol");
$preparada_nombre->execute( array(':rol' => 0));
echo "Usuarios con rol 0: " .  $preparada->rowCount() . "<br>";
foreach ($preparada_nombre  as $usu) {
    print "Nombre : " . $usu['nombre'] . "<br>";
}
```

Podemos ver ambos ejemplos en **bd_listar_usuario_2.php**

Ejercicio 4

Modifica el formulario sesiones1_login.php de la carpeta "Scripts de la segunda parte", para que compruebe usuario y contraseña usando la tabla usuarios de la base de datos *empresa*.

4.2.4. Inserción, borrado y actualización

Para insertar, borrar o actualizar simplemente hay que ejecutar la sentencia SQL correspondiente. Puede ser una sentencia preparada o no. Veamos **bd_insert_update_delete.php**

```php
<?php
// datos conexión
$cadena_conexion = 'mysql:dbname=empresa;host=127.0.0.1';
$usuario = 'root';
$clave = '';
try {
   // conectar
    $bd = new PDO($cadena_conexion, $usuario, $clave);
   echo "Conexión realizada con éxito<br>";
   // insertar nuevo usario
   $ins = "insert into usuarios(nombre, clave, rol) values('Alberto', '33333',
'1');";
   $resul = $bd->query($ins);
   //comprobar errores
   if($resul) {
       echo "insert correcto <br>";
       echo "Filas insertadas: " . $resul->rowCount() . "<br>";
   }else print_r( $bd -> errorinfo());
   // para los autoincrementos
   echo "Código de la fila insertada" . $bd->lastInsertId() . "<br>";
   // actualizar
   $upd = "update usuarios set rol =  0 where rol = 1";
   $resul = $bd->query($upd);
   //comprobar errores
   if($resul){
       echo "update correcto <br>";
       echo "Filas actualizadas: " . $resul->rowCount() . "<br>";
   }else print_r( $bd -> errorinfo());
   // borrar
   $del = "delete from usuarios where nombre = 'Luisa'";
   $resul = $bd->query($del);
   //comprobar errores
   if($resul){
       echo "delete correcto <br>";
       echo "Filas borradas: " . $resul->rowCount() . "<br>";
   }else print_r( $bd -> errorinfo());

} catch (PDOException $e) {
   echo 'Error con la base de datos: ' . $e->getMessage();
}
```

En la línea 19 utilizamos uno de los métodos de la clase PDO que nos permite obtener datos sobre la última fila insertada en la base de datos. Sin embargo, hemos de tener especial cuidado con esta función dado a que se comporta de una manera especial.

Veamos el ejemplo **bd_ultima_fila.php**

```php
<?php
// datos conexión
$cadena_conexion = 'mysql:dbname=empresa;host=127.0.0.1';
```

```
 4:  $usuario = 'root';
 5:  $clave = '';
 6:  try {
 7:     // conectar
 8:      $bd = new PDO($cadena_conexion, $usuario, $clave);
 9:     echo "Conexión realizada con éxito.<br>";
10:     echo "Truncamos la tabla usuarios.<br>";
11:     echo "<br>";
12:     // Trucamos la tabla
13:     $bd->query("TRUNCATE usuarios");
14:     // insertar solo un nuevo usario
15:     $ins = "INSERT INTO usuarios(nombre, clave, rol) VALUES ('Ana', '11111',
'0'),('Sofia', '22222', '0'),('Rosa', '33333', '0')";
16:     $resul = $bd->query($ins);
17:     //comprobar errores
18:     if($resul) {
19:         echo "[1] Insertamos tres usuarios<br>";
20:         echo " · Filas insertadas: " . $resul->rowCount() . "<br>";
21:         // para los autoincrementos
22:         echo " · Código de la fila insertada: " . $bd->lastInsertId() . "<br>";
23:     }else print_r( $bd -> errorinfo());
24:     echo "<br>";
25:     // insertar varios nuevos usarios
26:     $ins = "INSERT INTO usuarios(nombre, clave, rol) VALUES ('Carmen', '44444',
'0'),('Cristina', '55555', '0')";
27:     $resul = $bd->query($ins);
28:     //comprobar errores
29:     if($resul) {
30:         echo "[2] Insertamos dos usuarios<br>";
31:         echo " · Filas insertadas: " . $resul->rowCount() . "<br>";
32:     // para los autoincrementos
33:     echo " · Código de la fila insertada: " . $bd->lastInsertId() . "<br>";
34:     }else print_r( $bd -> errorinfo());
35:  } catch (PDOException $e) {
36:     echo 'Error con la base de datos: ' . $e->getMessage();
37:  }
38:
```

En la línea 13 usamos la instrucción `"TRUNCATE usuarios"` de esta manera borramos los datos de la tabla y ajustamos las variables autoincreméntales a cero. Si solo borrásemos los datos con `"DELETE FROM usuarios"` la columna Código seguiría incrementando su valor, pese a solo tener 5 datos en la tabla.

Para que el ejemplo funcione, tiene que estar activada la opción de autocommit en la base datos. Esta opción controlar el comportamiento de una operación. Cuando está activada, cada sentencia SQL se considera una ***transacción*** completa y se realiza un *guardado* (commit) cuando terminan.

4.2.5. Transacciones

Una transacción consiste en un conjunto de operaciones que deben realizarse de forma automática. Es decir, o se realizan todas o no se realiza ninguna. Por ejemplo, una transferencia de saldo entre dos clientes implica dos operaciones:

- Quitar saldo al cliente que envía la transferencia.
- Aumentar el saldo del cliente que recibe la transferencia.

Si por cualquier motivo la segunda operación falla, hay que deshacer la primera operación para que el sistema quede en un estado consistente. Las dos operaciones forman una única transacción Para indicar el comienzo de una transacción se utiliza el método `begintransaction()`.

Este método desactiva el modo 'autocomit', con lo que no se alterará el contenido de la base de datos hasta que no se realice el 'commit'. Esto nos permite encadenar cualquier número de instrucciones SQL, las cuales solo producirán un cambio efectivo en la base de datos cuando hallamos comprobado que todas son correctas y llamememos al método `commit()`.

En el caso de que alguna instrucción sea errónea durante una transacción, será necesario deshacer las operaciones realizadas hasta el momento y, para ello, invocaremos al método `rollBack()`

En el ejemplo **bd_transaccion.php** se realizan dos inserciones como una transacción. Como la segunda falla (tiene un valor repetido para el campo nombre que es unique), la primera se deshace.

```
 1:  <?php
 2:  $cadena_conexion = 'mysql:dbname=empresa;host=127.0.0.1';
 3:  $usuario = 'root';
 4:  $clave = '';
 5:  try {
 6:      $bd = new PDO($cadena_conexion, $usuario, $clave);
 7:     echo "Conexión realizada con éxito<br>";
 8:     // comenzar la transacción
 9:     $bd->beginTransaction();
10:     $ins = "insert into usuarios(nombre, clave, rol) values('Fernando', '33333',
'1')";
11:     $resul = $bd->query($ins);
12:     // se repite la consulta
13:     // falla porque el nombre es unique
14:     $resul = $bd->query($ins);
15:     if(!$resul){
16:         echo "Error: " . print_r($bd->errorinfo());
17:         // deshace el primer cambio
18:         $bd->rollback();
19:         echo "<br>Transacción anulada<br>";
20:     }else{
21:         // si hubiera ido bien...
22:         $bd->commit();
23:     }
24:  } catch (PDOException $e) {
25:      echo 'Error al conectar: ' . $e->getMessage();
26:  }
```

Ejercicio 5

Modifica el formulario sesiones1_principal.php para que cuando se acceda desde sesiones1_login.php con un perfil de administrador se muestren los datos de los usuarios registrados en una tabla.

Ejercicio 6

Crear un formulario solo visible para los administradores en sesiones1_principal.php en el que se pueda introducir el código de un usuario y, una vez enviado el formulario, nos redirija a la página editar.php.
En la página editar.php se han de mostrar los datos del usuario seleccionado en la página anterior dentro de un formulario que permita cambiarlos. Agregar un enlace para volver a la página anterior.
editar.php solo puede ser accesible si hemos iniciado sesión como administrador. En caso contrario hemos de ser redirigidos a sesiones1_principal.php (Ver sesiones1_logout.php)