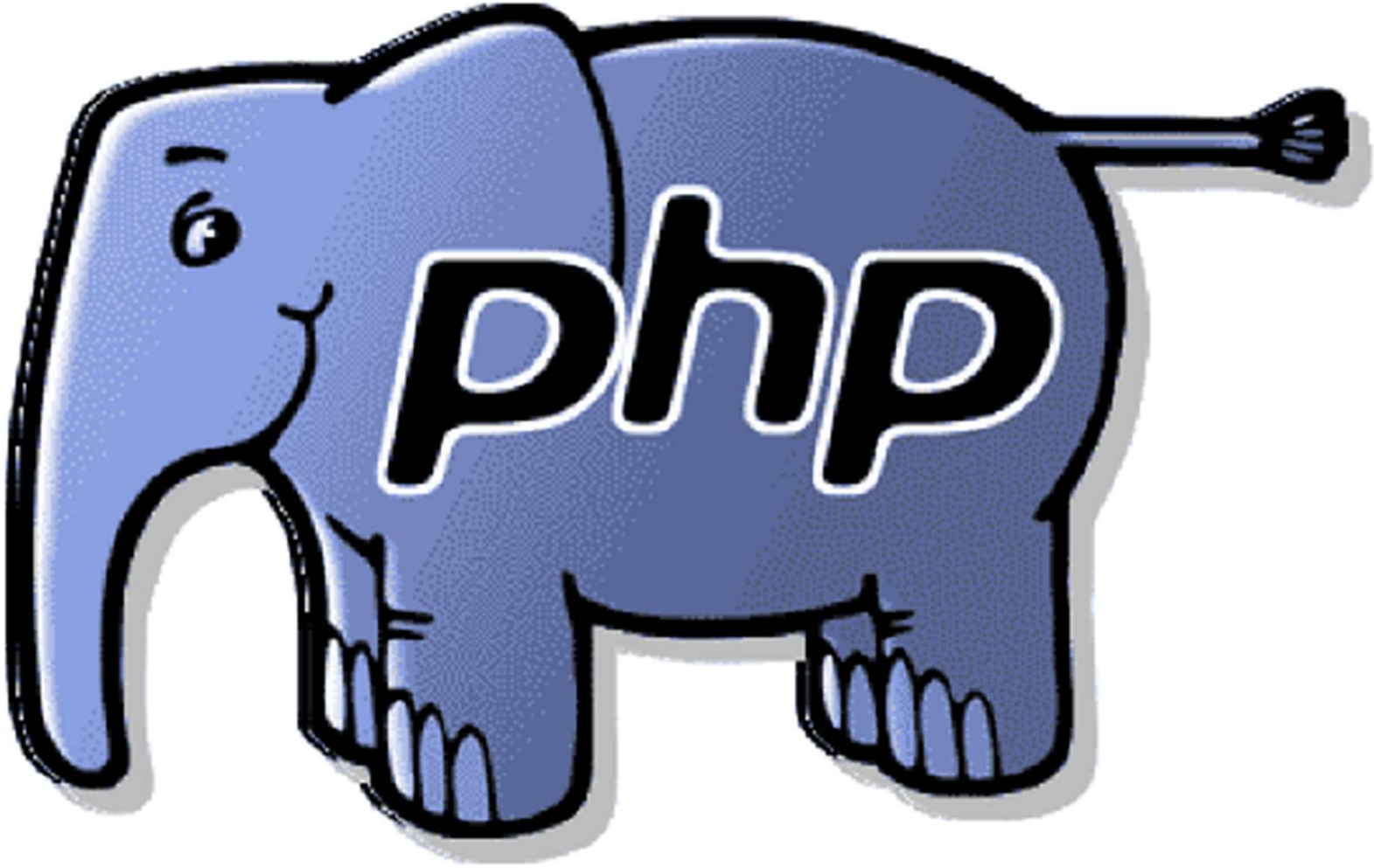
Utilización de técnicas de acceso a datos



Desarrollo Web en Entorno Servidor

Contenido

1. Acceso a bases de datos desde PHP
2. Características básicas de la utilización de objetos en PHP
3. MySQL
4. Utilización de bases de datos MySQL en PHP
5. Extensión MySQLi
6. PHP Data Objects (PDO)
7. Errores y manejo de excepciones

1. Acceso a bases de datos desde PHP

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**. Con PHP se puede enviar sentencias al gestor de una base de datos para que consulte, añada, elimine o actualice registros de la base de datos.

PHP soporta más de 15 sistemas gestores de bases de datos: SQLite, Oracle, SQL Server, PostgreSQL, IBM DB2, MySQL, etc. Hasta PHP 5, el acceso a las bases de datos se hacía utilizando extensiones específicas para cada sistema gestor de base de datos (**extensiones nativas**). Es decir, si se quería acceder a una base de datos concreta, se debía instalar y utilizar la extensión de ese gestor en concreto.

A partir de PHP 5 se introdujo una extensión para acceder de forma común a distintos sistemas gestores: **PDO**. La ventaja de PDO es poder seguir utilizando una misma sintaxis aunque se cambie el motor de la base de datos. Por el contrario, las extensiones nativas normalmente ofrecen más potencia y mayor velocidad.

De los distintos SGBD existentes, se utilizará **MySQL**, y se verá cómo **acceder desde PHP a bases de datos MySQL** utilizando tanto **PDO** como la extensión nativa **MySQLi**.

Para el seguimiento de esta unidad se utilizará SQL y orientación a objetos.

2. Características básicas de la utilización de objetos en PHP

- **Las clases: *class***

Una clase es un conjunto de variables (atributos) y funciones (métodos) que trabajan sobre esas variables. Las clases son definiciones a partir de las cuales se crean objetos.

Las clases en PHP se definen así:

```
<?
class Caja{
   var $alto;
   var $ancho;
   var $largo;
   var $contenido;
   var $color;
   function introduce($cosa){
      $this->contenido = $cosa;
   }
   function muestra_contenido(){
      echo $this->contenido;
   }
}
?>
```

Ej.: La clase "Caja" tendrá por atributos características como: dimensiones, color, contenido y cosas semejantes. Las funciones o métodos que se podrán incorporar a la clase "caja" serán funcionalidades que se desea que realice la caja: introduce(), muestra_contenido(), vaciate(), comprueba_si_cabe()...

En el ejemplo se ha creado la clase Caja, con atributos: ancho, alto, largo, color y contenido de la caja. Se han creado un par de métodos: uno para introducir un elemento en la caja y otro para mostrar el contenido.

Los atributos se definen declarando unas variables al principio de la clase. Los métodos se definen declarando funciones dentro de la clase.

2. Características básicas de la utilización de objetos en PHP

- **Utilizar la clase**

Las clases solamente son definiciones. Si se quiere utilizar la clase se tiene que crear un ejemplar de dicha clase, instanciar un objeto de una clase.

$micaja = new Caja;

Con esto se ha creado, o instanciado, un objeto de la clase Caja llamado $micaja.

$micaja->introduce("algo");

$micaja->muestra_contenido();

Con estas dos sentencias se introduce "algo" en la caja y luego se muestra ese contendido en el texto de la página.

Los métodos de un objeto se llaman utilizando el código "->".

nombre_del_objeto->nombre_de_metodo()

Para acceder a los atributos de una clase también se accede con el código "->".

nombre_del_objeto->nombre_del_atributo

- **La variable *$this***

Dentro de un método, la variable $this hace referencia al objeto sobre el que se invoca el método.

En la invocación $micaja->introduce("algo") se está llamando al método introduce sobre el objeto $micaja. Cuando se está ejecutando ese método, se vuelca el valor que recibe por parámetro en el atributo contenido. En ese caso $this->contenido hace referencia al atributo contenido del objeto $micaja.

3. MySQL

MySQL es un sistema gestor de bases de datos (SGBD) relacionales.

Es de código abierto con licencia GNU GPL. También hay una licencia comercial para desarrollar aplicaciones de código propietario.

Incorpora múltiples motores de almacenamiento, siendo **InnoDB** el utilizado por defecto.

MySQL se emplea en múltiples aplicaciones web, muy ligado al lenguaje **PHP** y al servidor web **Apache**. Utiliza SQL para la gestión, consulta y modificación de la información, y soporta ANSI **SQL 99**.

Elementos de instalación:

- Programa **servidor**: necesario para gestionar las bases de datos y permitir conexiones desde el equipo local o través de red.
- Programas **cliente**: se conectan al servidor MySQL, desde el mismo equipo que ejecuta el servidor o a través de red.

Herramientas de administración:

Existen muchas, algunas se ejecutan en la línea de comandos, otras tienen un interface gráfico; unas se incluyen con el propio servidor, y otras se instalan de forma independiente.

3. MySQL

Con el servidor MySQL se incluyen herramientas de administración en línea de comandos:

- **mysql**. Permite conectarse a un servidor MySQL para ejecutar sentencias SQL.
- **mysqladmin**. Es un cliente específico para tareas de administración.
- **mysqlshow**. Muestra información sobre bases de datos y tablas.

Herramientas de administración independientes:

- **MySQL Workbench**: permite administrar tanto el servidor como acceder a las bases de datos.
- **phpMyAdmin**: aplicación web para la administración de las bases de datos y la gestión de la información que maneja el servidor MySQL.

  Puede venir integrado en paquetes de software, como LAMP, WAMP, MAMP o XAMPP.

  Una vez instalado se puede acceder por la URL: "**http://localhost/phpmyadmin/**".

  Mediante phpMyAdmin se podrá: ver y modificar la **estructura** de la base de datos, ejecutar sentencias **SQL**, **buscar** información en la base de datos, **generar una consulta** utilizando un asistente, **exportar** e **importar** información, **diseñar** las relaciones existentes entre las tablas, hacer una copia de la base de datos, etc.

4. Utilización de bases de datos MySQL en PHP

Como se ha visto, existen dos **formas de comunicarse con una base de datos desde PHP**:

1. utilizar una extensión nativa programada para un SGBD concreto,

2. utilizar una extensión que soporte varios tipos de bases de datos.

Ello pasa por elegir entre **MySQLi** (extensión nativa) y **PDO**.

Con cualquiera de ambas extensiones, **se podrán realizar acciones** sobre las bases de datos como:

- ✓ Establecer conexiones.
- ✓ Ejecutar sentencias SQL.
- ✓ Obtener los registros afectados o devueltos por una sentencia SQL.
- ✓ Emplear transacciones.
- ✓ Ejecutar procedimientos almacenados.
- ✓ Gestionar los errores que se produzcan durante la conexión o en el establecimiento de la misma.

5. Extensión MySQLi

Esta extensión ofrece una **interface de programación dual**, pudiendo accederse a las funcionalidades de la extensión utilizando objetos o funciones de forma indiferente.

Por ejemplo, para establecer una conexión con un servidor MySQL y consultar su versión, podemos utilizar cualquiera de las siguientes formas:

// utilizando **programación orientada a objetos**:

```
$conexion = new mysqli('localhost', 'usuario', 'contraseña', 'base_de_datos');
print $conexion->server_info;
```

// utilizando **llamadas a funciones**:

```
$conexion = mysqli_connect('localhost', 'usuario', 'contraseña', 'base_de_datos');
print mysqli_get_server_info($conexion);
```

La utilización de los métodos y propiedades que aporta la clase **mysqli** normalmente produce un código más corto y legible que si se utilizan llamadas a funciones.

5. Extensión MySQLi

- **Establecimiento de conexiones**

Si se utiliza MySQLi, establecer una conexión significa **crear una instancia de la clase mysqli**.
El constructor de la clase puede recibir seis **parámetros**, todos opcionales:

- El nombre o dirección IP del servidor MySQL al que se va a conectar.
- Un nombre de usuario con permisos para establecer la conexión.
- La contraseña del usuario.
- El nombre de la base de datos a la que conectarse.
- El número del puerto en que se ejecuta el servidor MySQL.
- El socket o la tubería con nombre (named pipe) a usar.

Utilizando el constructor de **la clase**, para conectarte a la base de datos "dwes":

```
$dwes = new mysqli('localhost', 'dwes', 'abc123.', 'dwes');
```

También se puede primero crear la instancia, y después utilizar el método connect:

```
$dwes = new mysqli();
$dwes->connect('localhost', 'dwes', 'abc123.', 'dwes');
```

Utilizando el **interface procedimental** de la extensión (llamadas a funciones):

```
$dwes = mysqli_connect('localhost', 'dwes', 'abc123.', 'dwes');
```

5. Extensión MySQLi

Es importante **verificar** que **la conexión** se ha establecido correctamente:

- **connect_errno** (o función **mysqli_connect_errno**) devuelve el número de error o null si no se produce ningún error.
- **connect_error** (o función **mysqli_connect_error**) devuelve el mensaje de error o null si no se produce ningún error.

Ej.: comprobar el establecimiento de una conexión con la base de datos "dwes":

```
@ $dwes = new mysqli('localhost', 'dwes', 'abc123.', 'dwes');
$error = $dwes->connect_errno;
if ($error != null) {
    echo "<p>Error $error conectando a la base de datos: $dwes->connect_error</p>";
    exit();
}
```

(En PHP se puede anteponer a cualquier expresión el **operador de control de errores @** para que se ignore cualquier posible error que pueda producirse al ejecutarla).

Una vez finalizadas las tareas con la base de datos, se utiliza el método **close** (o la función **mysqli_close**) para **cerrar la conexión** con la base de datos y liberar los recursos que utiliza.

```
$dwes->close();
```

5. Extensión MySQLi

- **Ejecución de consultas**

La forma de ejecutar una consulta es el método **query**, (o función **mysqli_query)**. Si se ejecuta una **consulta de acción** que **no devuelve datos** (sentencia SQL de tipo **UPDATE**, **INSERT** o **DELETE**), la llamada devuelve true si se ejecuta correctamente o false en caso contrario. El número de registros afectados se puede obtener con la propiedad **affected_rows** (o la función **mysqli_affected_rows**).

```
@ $dwes = new mysqli('localhost', 'dwes', 'abc123.', 'dwes');
$error = $dwes->connect_errno;
if ($error == null) {
    $resultado = $dwes->query('DELETE FROM stock WHERE unidades=0');
    if ($resultado)
        print "<p>Se han borrado $dwes->affected_rows registros.</p>";
    $dwes->close();
}
```

En el caso de ejecutar una **sentencia SQL que devuelva datos** (**SELECT**), éstos se devuelven en forma de un objeto resultado (de la clase **mysqli_result**).

Los resultados obtenidos se almacenarán en memoria mientras se estén usando. Cuando ya no se necesitan, se pueden liberar con el método **free** de la clase **mysqli_result** (o la función **mysqli_free_result**): $resultado->free();

5. Extensión MySQLi

- **Obtención y utilización de conjuntos de resultados**

Para trabajar con los datos obtenidos del servidor, hay varias posibilidades:

- **fetch_array** (función **mysqli_fetch_array)**. Obtiene un registro completo del conjunto de resultados y lo almacena en un array. Por defecto el array contiene tanto claves numéricas como asociativas.

  (Para acceder al primer campo devuelto, se puede utilizar como clave el 0 o su nombre).

```
$resultado = $dwes->query('SELECT producto, unidades FROM stock WHERE unidades<2');
$stock = $resultado->fetch_array(); // Obtenemos el primer registro
$producto = $stock['producto']; // O también $stock[0];
$unidades = $stock['unidades']; // O también $stock[1];
print "<p>Producto $producto: $unidades unidades.</p>";
```

  Este comportamiento por defecto se puede modificar utilizando un parámetro opcional, que puede tomar los siguientes valores:

  - **MYSQLI_NUM**. Devuelve un array con claves numéricas.
  - **MYSQLI_ASSOC**. Devuelve un array asociativo.
  - **MYSQLI_BOTH**. Es el comportamiento por defecto, en el que devuelve un array con claves numéricas y asociativas.

5. Extensión MySQLi

- **fetch_assoc** (función **mysqli_fetch_assoc)**. Idéntico a fetch_array pasando como parámetro **MYSQLI_ASSOC**.
- **fetch_row** (función **mysqli_fetch_row)**. Idéntico a fetch_array pasando como parámetro **MYSQLI_NUM**.
- **fetch_object** (función **mysqli_fetch_object)**. Similar a los métodos anteriores, pero devuelve un objeto en lugar de un array. Las propiedades del objeto devuelto se corresponden con cada uno de los campos del registro.

Parar **recorrer todos los registros de un array**, se puede hacer un bucle teniendo en cuenta que cualquiera de los métodos o funciones anteriores devolverá null cuando no haya más registros en el conjunto de resultados.

```
$resultado = $dwes->query('SELECT producto, unidades FROM stock WHERE unidades<2');
$stock = $resultado->fetch_object();
while ($stock != null) {
    print "<p>Producto $stock->producto: $stock->unidades unidades.</p>";
    $stock = $resultado->fetch_object();
}
```

(Ver ejemplo: conjuntosderesultados.php asociado a dwes.sql)

5. Extensión MySQLi

- **Transacciones**

Para utilizar transacciones hay que asegurarse de que éstas estén soportadas por el motor de almacenamiento que gestiona las tablas en MySQL. Por defecto cada consulta individual se incluye dentro de su propia transacción. Este comportamiento se puede gestionar con el método **autocommit** (función **mysqli_autocommit**).

```
$dwes->autocommit(false); // deshabilita el modo transaccional automático
```

Al deshabilitar las transacciones automáticas, las siguientes operaciones sobre la base de datos iniciarán una transacción que se finalizará utilizando:

- **commit** (o función **mysqli_commit**). Realizar una operación "commit" de la transacción actual, devolviendo true si se ha realizado correctamente o false en caso contrario.
- **rollback** (o función **mysqli_rollback**). Realizar una operación "rollback" de la transacción actual, devolviendo true si se ha realizado correctamente o false en caso contrario.

```
…
$dwes->query('DELETE FROM stock WHERE unidades=0'); // Inicia una transacción
$dwes->query('UPDATE stock SET unidades=3 WHERE producto="STYLUSSX515W"');
…
$dwes->commit(); // Confirma los cambios
```

Una vez finalizada esa transacción, comenzará otra de forma automática.

(Ver ejemplo: transaccion.php asociado a dwes.sql)

5. Extensión MySQLi

- **Consultas preparadas**

Cuando se envía una consulta al servidor, éste debe analizarla antes de ejecutarla. Algunas sentencias SQL, como las de inserción, deben repetirse de forma habitual en un programa. Para acelerar este proceso, MySQL admite consultas preparadas. Estas consultas se almacenan en el servidor listas para ser ejecutadas cuando sea necesario.

Para trabajar con consultas preparadas con MySQLi, se utiliza la clase **mysqli_stmt**. Con el método **stmt_init** de la clase **mysqli** (o la función **mysqli_stmt_init**) se obtiene un objeto de dicha clase.

```
$dwes = new mysqli('localhost', 'dwes', 'abc123.', 'dwes');
$consulta = $dwes->stmt_init();
```

Los pasos a seguir para ejecutar una consulta preparada son:

- Preparar la consulta en el servidor: método **prepare** (función **mysqli_stmt_prepare**).
- Ejecutar la consulta (veces necesarias): método **execute** (función **mysqli_stmt_execute**).
- Cuando ya no se necesite: ejecutar el método **close** (función **mysqli_stmt_close**).

P.ej.: preparar y ejecutar una consulta que inserta un nuevo registro en la tabla familia:

```
$consulta = $dwes->stmt_init();
$consulta->prepare('INSERT INTO familia (cod, nombre) VALUES ("TABLET", "Tablet PC")');
$consulta->execute();
$consulta->close();
$dwes->close();
```

5. Extensión MySQLi

Pero no sirve preparar una consulta de inserción, si los valores que inserta son siempre los mismos. Por ello las consultas preparadas admiten parámetros: en lugar de valores se debe indicar con el signo ? su posición dentro de la sentencia SQL.

```
$consulta->prepare('INSERT INTO familia (cod, nombre) VALUES (?, ?)');
```

Y antes de ejecutar la consulta hay que utilizar el método **bind_param** (o la función **mysqli_stmt_bind_param**) para sustituir cada parámetro por su valor. El primer parámetro del método bind_param es una cadena de texto en la que cada carácter indica el tipo de un parámetro, según la siguiente tabla:

Caracteres indicativos del tipo de los parámetros en una consulta preparada.	
Carácter.	**Tipo del parámetro.**
I.	Número entero.
D.	Número real (doble precisión).
S.	Cadena de texto.
B.	Contenido en formato binario (BLOB).

En el caso anterior, si se almacenan los valores a insertar en variables:

```
$consulta = $dwes->stmt_init();
$consulta->prepare('INSERT INTO familia (cod, nombre) VALUES (?, ?)');
$cod_producto = "TABLET";
$nombre_producto = "Tablet PC";
$consulta->bind_param('ss', $cod_producto, $nombre_producto);
$consulta->execute();
$consulta->close();
$dwes->close();
```

5. Extensión MySQLi

Cuando se usa **bind_param** para enlazar los parámetros de la consulta preparada con sus valores, se usarán siempre variables, como en el ejemplo anterior. No utilizar literales:

```
$consulta->bind_param('ss', 'TABLET', 'Tablet PC'); // Genera un error
```

El método **bind_param** permite tener una consulta preparada en el servidor MySQL y ejecutarla tantas veces como se necesite cambiando ciertos valores cada vez. Además, en el caso de las consultas que devuelven valores, se puede utilizar el método **bind_result** (función **mysqli_stmt_bind_result**) para asignar a variables los campos que se obtienen tras la ejecución. Utilizando el método **fetch** (**mysqli_stmt_fetch**) se recorren los registros devueltos.

Ejemplo:

```
$consulta = $dwes->stmt_init();
$consulta->prepare('SELECT producto, unidades FROM stock WHERE unidades<2');
$consulta->execute();
$consulta->bind_result($producto, $unidades);
while($consulta->fetch()) {
    print "<p>Producto $producto: $unidades unidades.</p>";
}
$consulta->close();
$dwes->close();
```

(Ver consultaspreparadas.php asociado a dwes.sql)

6. PHP Data Objects (PDO)

Si se va a programar una aplicación que utilice como sistema gestor de bases de datos MySQL, la extensión **MySQLi** es una buena opción. Ofrece acceso a todas las características del motor de base de datos, a la vez que reduce los tiempos de espera en la ejecución de sentencias.

Sin embargo, **si en el futuro hay que cambiar el SGBD** por otro distinto, se tendrá que volver a programar gran parte del código de la misma. En el caso de que exista la posibilidad, presente o futura, de utilizar otro servidor, se deberá adoptar una capa de abstracción para el acceso a los datos. Existen varias alternativas, pero sin duda la opción más recomendable en la actualidad es **PDO**.

El objetivo es que, si llegado el momento se necesita cambiar el servidor de base de datos, las modificaciones a realizar en el código sean mínimas. PDO no abstrae de forma completa el sistema gestor que se utiliza. Por ejemplo, no modifica las sentencias SQL para adaptarlas a las características específicas de cada servidor.

PDO se basa en las características de orientación a objetos de PHP, pero al contrario que la extensión MySQLi, **no ofrece una interface de programación dual**. Para acceder a las funcionalidades de la extensión **hay que emplear los objetos que ofrece, con sus métodos y propiedades**. No existen funciones alternativas.

6. PHP Data Objects (PDO)

- **Establecimiento de conexiones**

Para establecer una conexión con una base de datos utilizando PDO, se debe instanciar un objeto de la clase PDO pasándole los siguientes parámetros (solo el primero es obligatorio):

- **Origen de datos** (DSN). Cadena de texto que indica qué **controlador** se va a utilizar y, separados por dos puntos, los parámetros necesarios para el controlador, como: el **nombre o dirección IP del servidor** y el **nombre de la base de datos**.
- **Nombre de usuario** con permisos para establecer la conexión.
- **Contraseña del usuario**.
- **Opciones** de conexión, almacenadas en forma de array.

P.ej.: Para establecer una conexión con la base de datos 'dwes‘:

```
$dwes = new PDO('mysql:host=localhost;dbname=dwes', 'dwes', 'abc123.');
```

Si se utiliza el controlador para **MySQL**, los parámetros específicos a utilizar en la cadena DSN (separados por punto y coma) a continuación del prefijo **mysql:** son los siguientes:

- **host**. Nombre o dirección IP del servidor.
- **port**. Número de puerto TCP en el que escucha el servidor.
- **dbname**. Nombre de la base de datos.
- **unix_socket**. Socket de MySQL en sistemas Unix.

6. PHP Data Objects (PDO)

Si se quiere indicar al servidor MySQL que utilice **codificación UTF-8** para los datos que se transmitan, se puede usar una opción específica de la conexión:

```
$opciones = array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8");
$dwes = new PDO('mysql:host=localhost;dbname=dwes', 'dwes', 'abc123.', $opciones);
```

Una vez establecida la conexión, se puede utilizar el método **getAttribute** para obtener información del estado de la conexión y **setAttribute** para modificar algunos parámetros que afectan a la misma.

P.ej.: para obtener la versión del servidor:

```
$version = $dwes->getAttribute(PDO::ATTR_SERVER_VERSION);
print "Versión: $version";
```

Y si se quiere que devuelva todos los nombres de columnas en mayúsculas:

```
$version = $dwes->setAttribute(PDO::ATTR_CASE, PDO::CASE_UPPER);
```

6. PHP Data Objects (PDO)

- **Ejecución de consultas**

Para ejecutar una consulta SQL utilizando PDO, se debe diferenciar aquellas sentencias SQL que no devuelven como resultado un conjunto de datos, de aquellas otras que sí lo devuelven.

En el caso de las consultas de acción, como **INSERT**, **DELETE** o **UPDATE**, el método **exec** devuelve el número de registros afectados.

```
$registros = $dwes->exec('DELETE FROM stock WHERE unidades=0');
print "<p>Se han borrado $registros registros.</p>";
```

Si la consulta genera un conjunto de datos, como es el caso de **SELECT**, se utilizará el método **query**.

```
$dwes = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123.");
$resultado = $dwes->query("SELECT producto, unidades FROM stock");
```

6. PHP Data Objects (PDO)

- **Obtención y utilización de conjuntos de resultados**

Al igual que con la extensión MySQLi, en PDO hay varias posibilidades para tratar el conjunto de resultados devuelto por el método **query**. La más utilizada es el método **fetch**. Este método devuelve un registro del conjunto de resultados, o false si ya no quedan registros por recorrer.

```
$dwes = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123.");
$resultado = $dwes->query("SELECT producto, unidades FROM stock");
while ($registro = $resultado->fetch()) {
    echo "Producto ".$registro['producto'].": ".$registro['unidades']."<br />";
}
```

Por defecto, el método **fetch** genera y devuelve, a partir de cada registro, un array con claves numéricas y asociativas. Para cambiar su comportamiento, admite un parámetro opcional que puede tomar uno de los siguientes valores:

- **PDO::FETCH_ASSOC**. Devuelve solo un array asociativo.
- **PDO::FETCH_NUM**. Devuelve solo un array con claves numéricas.
- **PDO::FETCH_BOTH**. Devuelve un array con claves numéricas y asociativas. Es el comportamiento por defecto.

6. PHP Data Objects (PDO)

- **PDO::FETCH_OBJ**. Devuelve un objeto cuyas propiedades se corresponden con los campos del registro.

```
$dwes = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123.");
$resultado = $dwes->query("SELECT producto, unidades FROM stock");
while ($registro = $resultado->fetch(PDO::FETCH_OBJ)) {
    echo "Producto ".$registro->producto.": ".$registro->unidades."<br />";
}
```

- **PDO::FETCH_LAZY**. Devuelve tanto el objeto como el array con clave dual anterior.
- **PDO::FETCH_BOUND**. Devuelve true y asigna los valores del registro a variables, según se indique con el método **bindColumn**. Este método debe ser llamado una vez por cada columna, indicando en cada llamada el número de columna (empezando en 1) y la variable a asignar.

```
$dwes = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123.");
$resultado = $dwes->query("SELECT producto, unidades FROM stock");
$resultado->bindColumn(1, $producto);
$resultado->bindColumn(2, $unidades);
while ($registro = $resultado->fetch(PDO::FETCH_BOUND)) {
    echo "Producto ".$producto.": ".$unidades."<br />";
}
```

(Ver ejemplo: conjuntosderesultadosPDO.php)

6. PHP Data Objects (PDO)

- **Transacciones**

Por defecto PDO trabaja en modo "autocommit", confirma de forma automática cada sentencia que ejecuta el servidor. Para trabajar con transacciones, PDO incorpora tres métodos:

- **beginTransaction**. Deshabilita el modo "autocommit" y comienza una nueva transacción, que finalizará cuando se ejecute uno de los dos métodos siguientes.
- **commit**. Confirma la transacción actual.
- **rollback**. Revierte los cambios llevados a cabo en la transacción actual.

Una vez ejecutado un commit o un rollback, se volverá al modo de confirmación automática.

```
$ok = true;
$dwes->beginTransaction();
if($dwes->exec('DELETE …') == 0) $ok = false;
if($dwes->exec('UPDATE …') == 0) $ok = false;
…
if ($ok) $dwes->commit(); // Si todo fue bien confirma los cambios
else $dwes->rollback(); // y si no, los revierte
```

(Ver ejemplo: transaccionPDO.php)

6. PHP Data Objects (PDO)

- **Consultas preparadas**

Utilizando PDO también se pueden preparar consultas parametrizadas en el servidor para ejecutarlas de forma repetida.

Para preparar la consulta en el servidor MySQL, se deberá utilizar el método **prepare** de la clase PDO.

Este método devuelve un objeto de la clase PDOStatement. Los **parámetros** se pueden marcar utilizando **signos de interrogación**.

```
$dwes = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123.");
$consulta = $dwes->prepare('INSERT INTO familia (cod, nombre) VALUES (?, ?)');
```

O también utilizando **parámetros con nombre**, precediéndolos por el símbolo de dos puntos (**:**).

```
$dwes = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123.");
$consulta = $dwes->prepare('INSERT INTO familia (cod, nombre) VALUES (:cod, :nombre)');
```

Antes de ejecutar la consulta hay que asignar valores a los parámetros utilizando el método **bindParam.**

Si se utilizan signos de interrogación para marcar los parámetros, el procedimiento es equivalente al método **bindColumn**.

6. PHP Data Objects (PDO)

```
$cod_producto = "TABLET";
$nombre_producto = "Tablet PC";
$consulta->bindParam(1, $cod_producto);
$consulta->bindParam(2, $nombre_producto);
```

Si se utilizan parámetros con nombre, se debe indicar ese nombre en la llamada a **bindParam**.

```
$consulta->bindParam(":cod", $cod_producto);
$consulta->bindParam(":nombre", $nombre_producto);
```

Importante: Cuando se use **bindParam** para asignar los parámetros de una consulta preparada, se deberán usar siempre variables.

Una vez preparada la consulta y enlazados los parámetros con sus valores, se ejecuta la consulta utilizando el método **execute**.

```
$consulta->execute();
```

Alternativamente, es posible **asignar los valores de los parámetros en el momento de ejecutar la consulta, utilizando un array** (asociativo o con claves numéricas dependiendo de la forma en que se haya indicado los parámetros) en la llamada a **execute**.

```
$parametros = array(":cod" => "TABLET", ":nombre" => "Tablet PC");
$consulta->execute($parametros);
```

(Ver ejemplo: consultaspreparadasPDO.php)

7. Errores y manejo de excepciones

PHP define una clasificación de los errores que se pueden producir en la ejecución de un programa y ofrece métodos para ajustar el tratamiento de los mismos. Para hacer referencia a cada uno de los **niveles de error**, PHP define una serie de **constantes**. Cada nivel se identifica por una constante.

P.ej.: la constante **E_NOTICE** hace referencia a avisos que pueden indicar un error al ejecutar el guión, y la constante **E_ERROR** engloba errores fatales que provocan que se interrumpa forzosamente la ejecución.

La configuración inicial de tratamiento de errores según su nivel se realiza en **php.ini** (fichero de configuración de PHP). Los parámetros que se pueden ajustar son:

- **error_reporting**. Indica qué tipos de errores se notificarán. Su valor se forma combinando las constantes anteriores. Su valor predeterminado es **E_ALL & ~E_NOTICE** que indica que se notifiquen todos los errores (**E_ALL**) salvo los avisos en tiempo de ejecución (**E_NOTICE**).
- **display_errors**. En su valor por defecto (**On**), hace que los mensajes se envíen a la salida estándar (y por lo tanto se muestren en el navegador). Se debe desactivar (**Off**) en los servidores que se usan para producción.

7. Errores y manejo de excepciones

Desde el código, se puede usar la función **error_reporting()** con las constantes anteriores para establecer el nivel de notificación en un momento determinado. P.ej.: si en algún lugar figura una división en la que exista la posibilidad de que el divisor sea cero, cuando ello ocurra se obtendrá un mensaje de error en el navegador. Para evitarlo, se puede desactivar la notificación de errores de nivel **E_WARNING** antes de la división y restaurarla a su valor normal a continuación:

```
error_reporting(E_ALL & ~E_NOTICE & ~E_WARNING);
$resultado = $dividendo / $divisor;
error_reporting(E_ALL & ~E_NOTICE);
```

Al usar la función **error_reporting()** solo se controla qué tipo de errores va a notificar PHP. Puede ser suficiente, pero para obtener más control sobre el proceso existe la posibilidad de reemplazar la gestión de los mismos. Es decir, es posible programar una función para que sea la que ejecuta PHP cada vez que se produce un error. El nombre de esa función se indica utilizando **set_error_handler()**, y debe tener dos parámetros obligatorios (el nivel del error y el mensaje descriptivo) y hasta otros tres opcionales (el nombre del fichero en que se produce, el número de línea, y un volcado del estado de las variables en ese momento).

7. Errores y manejo de excepciones

```
set_error_handler("miGestorDeErrores");
$resultado = $dividendo / $divisor;
restore_error_handler();

function miGestorDeErrores($nivel, $mensaje)
{
	switch($nivel) {
		case E_WARNING:
			echo "Error de tipo WARNING: $mensaje.<br />";
			break;
		default:
			echo "Error de tipo no especificado: $mensaje.<br />";
	}
}
```

La función **restore_error_handler()** restaura el manejador de errores original de PHP (el que se estaba usando antes de la llamada a **set_error_handler()**).

7. Errores y manejo de excepciones

- **Excepciones**

A partir de la versión 5 PHP utiliza un modelo de excepciones similar al de otros lenguajes:

- ✓ El código susceptible de producir algún error se introduce en un bloque **try**.
- ✓ Cuando se produce algún error, se lanza una excepción utilizando la instrucción **throw**.
- ✓ Después del bloque try debe haber como mínimo un bloque **catch** encargado de procesar el error.
- ✓ Si una vez acabado el bloque try no se ha lanzado ninguna excepción, se continúa con la ejecución en la línea siguiente al bloque o bloques catch.

P.ej.: para lanzar una excepción cuando se produce una división por cero se podría hacer:

```
try {
        if ($divisor == 0)
                throw new Exception("División por cero.");
        $resultado = $dividendo / $divisor;
}
catch (Exception $e) {
        echo "Se ha producido el siguiente error: ".$e->getMessage();
}
```

7. Errores y manejo de excepciones

PHP ofrece una clase base **Exception** para utilizar como manejador de excepciones. Para lanzar una excepción no es necesario indicar ningún parámetro, aunque de forma opcional se puede pasar un mensaje de error y también un código de error.

Entre los métodos que se pueden usar con los objetos de la clase **Exception** están:

- **getMessage**. Devuelve el mensaje, en caso de que se haya puesto alguno.
- **getCode**. Devuelve el código de error si existe.

Las funciones internas de PHP y muchas extensiones como MySQLi usan el sistema de errores visto anteriormente. Solo las extensiones más modernas orientadas a objetos, como es el caso de PDO, utilizan este modelo de excepciones.

Concretamente, la clase **PDO** permite definir la fórmula que usará cuando se produzca un error, utilizando el atributo **PDO::ATTR_ERRMODE**. Las posibilidades son:

- **PDO::ERRMODE_SILENT**. No se hace nada cuando ocurre un error. Comportamiento por defecto.
- **PDO::ERRMODE_WARNING**. Genera un error de tipo **E_WARNING** cuando se produce un error.
- **PDO::ERRMODE_EXCEPTION**. Cuando se produce un error lanza una excepción utilizando el manejador propio **PDOException**.

7. Errores y manejo de excepciones

Es decir, para utilizar excepciones con la extensión **PDO**, se debe configurar la conexión haciendo:

```
$dwes->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

P.ej.: el siguiente código:

```
$dwes = new PDO("mysql:host=localhost; dbname=dwes", "dwes", "abc123.");
$dwes->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
try {
        $sql = "SELECT * FROM stox";
        $result = $dwes->query($sql);
        …
}
catch (PDOException $p) {
        echo "Error ".$p->getMessage()."<br />";
}
```

Captura la excepción que lanza PDO debido a que la tabla no existe. El bloque catch muestra el siguiente mensaje:

```
Error SQLSTATE[42S02]: Base table or view not found: 1146 Table 'dwes.stox' doesn't exist
```

(Ver ejemplo: controldeexcepciones.php)