

## **U05. UTILIZACIÓN DE TÉCNICAS DE ACCESO A DATOS**

### **1.- Acceso a bases de datos desde PHP**

Una de las aplicaciones más frecuentes de PHP es generar un 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: SQLite, Oracle, SQL Server, PostgreSQL, IBM DB2, MySQL, etc. Hasta la versión 5 de PHP, el acceso a las bases de datos se hacía principalmente utilizando extensiones específicas para cada sistema gestor de base de datos (extensiones nativas). Es decir, que si queríamos acceder a una base de datos de PostgreSQL, deberíamos instalar y utilizar la extensión de ese gestor en concreto. Las funciones y objetos a utilizar eran distintos para cada extensión.

A partir de la versión 5 de PHP se introdujo en el lenguaje una extensión para acceder de una forma común a distintos sistemas gestores: PDO. La gran ventaja de PDO está clara: podemos seguir utilizando una misma sintaxis aunque cambiemos el motor de nuestra base de datos. Por el contrario, en algunas ocasiones preferiremos seguir usando extensiones nativas en nuestros programas. Mientras PDO ofrece un conjunto común de funciones, las extensiones nativas normalmente ofrecen más potencia (acceso a funciones específicas de cada gestor de base de datos) y en algunos casos también mayor velocidad.

De los distintos SGBD existentes utilizaremos MySQL. MySQL es un gestor de bases de datos relacionales de código abierto bajo licencia GNU GPL. Es el gestor de bases de datos más empleado con el lenguaje PHP. Es la letra "M" que figura en los acrónimos LAMP, WAMP y XAMPP.

Para acceder a bases de datos MySQL desde PHP utilizaremos tanto PDO como la extensión nativa MySQLi. Para el acceso a las funcionalidades de ambas extensiones se utilizan objetos.

#### **Características básicas de la utilización de objetos en PHP**

La programación orientada a objetos es una metodología de programación avanzada y bastante extendida, en la que los sistemas se modelan creando clases, que son un conjunto de datos y funcionalidades. Las clases son definiciones, a partir de las que se crean objetos. Los objetos son ejemplares de una clase determinada y como tal, disponen de los datos y funcionalidades definidos en la clase.

La programación orientada a objetos permite concebir los programas de una manera bastante intuitiva y cercana a la realidad. La tendencia es que un mayor número de lenguajes de programación adopten la programación orientada a objetos como paradigma para modelizar los sistemas. Prueba de ello es que desde la versión 5 de PHP se implanta la programación orientada a objetos como metodología de desarrollo. También Microsoft ha dado un vuelco hacia la programación orientada a objetos, ya que .NET dispone de varios lenguajes para programar y todos orientados a objetos.

Así pues, la programación orientada a objetos es un tema de gran interés, pues es muy utilizada y cada vez resulta más esencial para poder desarrollar en casi cualquier lenguaje moderno. A continuación haremos un breve repaso de POO bajo el enfoque de PHP.

## Las clases: *class*

Una clase es un conjunto de variables, llamados atributos o variables miembro, y funciones, llamadas métodos, que trabajan sobre esas variables. Las clases son, al fin y al cabo, una definición: una especificación de propiedades y funcionalidades de elementos que van a participar en nuestros programas.

Por ejemplo, la clase "Caja" tendría como atributos características como las dimensiones, color, contenido y cosas semejantes. Las funciones o métodos que podríamos incorporar a la clase "caja" son las funcionalidades que deseamos que realice la caja, como introduce(), muestra\_contenido(), comprueba\_si\_cabe() , vaciate() ...

Las clases en PHP se definen de la siguiente manera:

```
<?php
class Caja{
    private $alto;
    private $ancho;
    private $largo;
    private $contenido;
    private $color;

    function introduce($cosa){
        $this->contenido = $cosa;
    }
    function muestraContenido() {
        echo $this->contenido, "\n";
    }
}
$c=new Caja();
$c->introduce("Libro");
$c->muestraContenido();
?>
```

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

Si nos fijamos, los atributos se definen declarando unas variables al principio de la clase. Los métodos se definen declarando funciones dentro de la clase. La variable \$this se refiere al objeto instanciado desde el que se llama al método, así en la llamada a **\$c->muestraContenido()**, dentro del método **\$this** representa al objeto creado con el nombre **\$c**.

En el ejemplo anterior hemos **instanciado** un objeto utilizando el operador **new**, y a continuación hemos hecho uso de los métodos **introducir** y **muestraContenido**. Como se ve en el ejemplo para acceder a los métodos y/o variables miembro se utiliza el operador **->**.

## Constructores

Los constructores son funciones, o métodos, que se encargan de realizar las tareas de inicialización de los objetos al ser instanciados. Es decir, cuando se crean los objetos a partir de las clases, se llama a un constructor que se encarga de inicializar los atributos del objeto y realizar cualquier otra tarea de inicialización que sea necesaria.

No es obligatorio disponer de un constructor, pero resultan muy útiles y su uso es muy habitual. En el ejemplo de la caja, que comentábamos anteriormente, lo normal sería inicializar las variables como color o las relacionadas con las dimensiones y, además, indicar que el contenido de la caja está vacío. Si no hay un constructor no se inicializan ninguno de los atributos de los objetos.

Podríamos añadir un constructor al ejemplo anterior como sigue:

```
<?php
class Caja{
    private $alto;
    private $ancho;
    private $largo;
    private $contenido;
    private $color;

    public function __construct($alto, $ancho, $largo=5, $contenido='Verduras', $color='Verde') {
        $this->alto=$alto;
        $this->ancho=$ancho;
        $this->largo=$largo;
        $this->contenido=$contenido;
        $this->color=$color;
    }
    public function introduce($cosa){
        $this->contenido = $cosa;
    }
    public function muestraContenido() {
        echo $this->contenido, "\n";
    }
}
$c=new Caja(7,7,7,'Frutas','Verde');
$c->introduce("Naranjas");
$c->muestraContenido();
$c1=new Caja(5,5);
$c1->muestraContenido();
?>
```

## 2.- MySQL

MySQL es un sistema gestor de bases de datos (SGBD) relacionales. Incorpora múltiples motores de almacenamiento, cada uno con características propias: unos son más veloces, otros, aportan mayor seguridad o mejores capacidades de búsqueda. Cuando se crea una base de datos, se puede elegir el motor de base de datos en función de las características propias de la aplicación. Si no se especifica, el motor que se utiliza por defecto se llama MyISAM, que es muy rápido pero a cambio no contempla integridad referencial ni tablas transaccionales ( conjunto de operaciones sobre los datos que se han de realizar de forma conjunta, una sola vez, e independientemente del resto de manipulaciones sobre los datos).

Toda transacción debe cumplir cuatro propiedades: atomicidad, consistencia, aislamiento y permanencia. El motor InnoDB es un poco más lento pero sí soporta tanto integridad referencial como tablas transaccionales.

### Atomicidad

La Atomicidad requiere que cada transacción sea "todo o nada": si una parte de la transacción falla, todas las operaciones de la transacción fallan, y por lo tanto la base de datos no sufre cambios. Un sistema atómico tiene que garantizar la atomicidad en cualquier operación y situación, incluyendo fallas de alimentación eléctrica, errores y caídas del sistema.

### Consistencia

La propiedad de Consistencia se asegura que cualquier transacción llevará a la base de datos de un estado válido a otro estado válido. Cualquier dato que se escriba en la base de datos tiene que ser válido de acuerdo a todas las reglas definidas, como son *constraints*, *cascades*, *triggers* y/o cualquier combinación de ellos.

### Aislamiento

El aislamiento ("Isolation" en inglés) se asegura que la ejecución concurrente de las transacciones resulte en un estado del sistema que se obtendría si estas transacciones fueran ejecutadas una detrás de otra. Cada transacción debe ejecutarse en aislamiento total; por ejemplo, si T1 y T2 se ejecutan concurrentemente, luego cada una debe mantenerse independiente de la otra.

### Durabilidad

La durabilidad significa que una vez que se confirmó una transacción (*commit*), quedará persistentemente, incluso ante eventos como pérdida de alimentación eléctrica, errores y caídas del sistema. Por ejemplo, en las bases de datos relacionales, una vez que se ejecuta un grupo de sentencias SQL, los resultados tienen que almacenarse inmediatamente (incluso si la base de datos se cae inmediatamente después).

MySQL se emplea en múltiples aplicaciones web, ligado en la mayor parte de los casos al lenguaje PHP y al servidor web Apache. Utiliza SQL para la gestión, consulta y modificación de la información almacenada. Soporta la mayor parte de las características de ANSI SQL, añadiendo algunas extensiones propias.

### Instalación

Dependiendo de la distribución de **Linux** que estemos usando, es posible que MySQL se haya reemplazado por MariaDB. Según fuera el caso, instalaremos ejecutando:

```
sudo apt install mysql-server mysql-client
ó
sudo apt install mariadb-server mariadb-client
```

Para controlar el servicio utilizaremos la herramienta **systemctl**:

```
sudo systemctl start | stop | restart mysql
```

---

**ESTO YA NO ES NECESARIO (pero por si acaso)** Para que nos deje conectarnos con el usuario root sin hacer sudo:  
`UPDATE mysql.user SET plugin = 'mysql_native_password' WHERE user = 'root' AND plugin = 'unix_socket';`  
`ALTER USER root@localhost IDENTIFIED WITH mysql_native_password by 'mipassword';`  
`FLUSH PRIVILEGES;` (Ya no es necesario, hacer prueba creando una base de datos con usuario)

---

El puerto que utiliza tanto MySQL como MariaDB por defecto es el 3306 de TCP.

Los ficheros de configuración se encuentran, en las últimas versiones, en la carpeta `/etc/mysql`.

Podemos ejecutar scripts de mysql en línea de comandos, de la siguiente forma:

```
[sudo] mysql -u usuario -p[password] < script.sql
```

Para comunicarse con el servidor existen numerosas herramientas, como son: **phpmyadmin**, **mysql-workbench**, el cliente de línea de comandos visto anteriormente, etc., etc., etc.

## **3.- Acceso a MySQL desde PHP**

Existen dos formas de comunicarse con una base de datos desde PHP: utilizar una extensión nativa programada para un SGBD concreto, o utilizar una extensión que soporte varios tipos de bases de datos. Tradicionalmente las conexiones se establecían utilizando la extensión nativa mysql. Esta extensión se mantiene en la actualidad para dar soporte a las aplicaciones ya existentes que la

utilizan, pero no se recomienda utilizarla para desarrollar nuevos programas. Lo más habitual es elegir entre MySQLi (extensión nativa) y PDO.

Con cualquiera de ambas extensiones, se pueden 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.

PDO y MySQLi (y también la antigua extensión mysql) utilizan un driver de bajo nivel para comunicarse con el servidor MySQL. Hasta hace poco el único driver disponible para realizar esta función era libmysql, que no estaba optimizado para ser utilizado desde PHP. A partir de la versión 5.3, PHP viene preparado para utilizar también un nuevo driver mejorado para realizar esta función, el Driver Nativo de MySQL, mysqlnd.

#### **4.- Extensión MySQLi**

Esta extensión se desarrolló para aprovechar las ventajas que ofrecen las versiones 4.1.3 y posteriores de MySQL, y viene incluida con PHP a partir de la versión 5. Ofrece un 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:

```
<?php
// utilizando constructores y métodos de la programación orientada a objetos
$host='localhost'; $usuario='productos'; $pass='productos2021'; $bd='productos';
$conexion = new MySQLi($host, $usuario, $pass, $bd);
echo $conexion->server_info;
// utilizando llamadas a funciones
$conexion = MySQLi_connect($host, $usuario, $pass, $bd);
echo MySQLi_get_server_info($conexion);
?>
```

En ambos casos, la variable \$conexion es un objeto. 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.

Toda la información relativa a la instalación y utilización de la extensión, incluyendo las funciones y métodos propios de la extensión, se puede consultar en el manual de PHP:

<http://es.php.net/manual/es/book.MySQLi.php>

En *Debian/Ubuntu* instalaremos la extensión ejecutando:

***sudo apt install php-mysql*** (que también instala el soporte para **PDO**)

Entre las mejoras que aporta sobre la antigua extensión mysql, figuran:

- Interface orientado a objetos.
- Soporte para transacciones.
- Soporte para consultas preparadas.
- Mejores opciones de depuración.

Las opciones de configuración de PHP se almacenan en el fichero **php.ini** y los ficheros incluidos en la carpeta **conf.d**. En el fichero **php.ini** hay una sección específica para las opciones de configuración propias de esta extensión. Entre las opciones que se pueden configurar para la extensión MySQLi están:

- MySQLi.allow\_persistent. Permite crear conexiones persistentes.
- MySQLi.default\_port. Número de puerto TCP predeterminado a utilizar cuando se conecta al servidor de base de datos.
- MySQLi.reconnect. Indica si se debe volver a conectar automáticamente en caso de que se pierda la conexión.
- MySQLi.default\_host. Host predeterminado a usar cuando se conecta al servidor de base de datos.
- MySQLi.default\_user. Nombre de usuario predeterminado a usar cuando se conecta al servidor de base de datos.
- MySQLi.default\_pw. Contraseña predeterminada a usar cuando se conecta al servidor de base de datos.

En la documentación de PHP se incluye una lista completa de las directivas relacionadas con la extensión MySQLi que se pueden utilizar en php.ini:

<https://www.php.net/manual/es/book.MySQLi.php>

#### Establecimiento de conexiones.

Para poder comunicarte desde un programa PHP con un servidor MySQL, el primer paso es establecer una conexión. Toda comunicación posterior que tenga lugar, se hará utilizando esa conexión. Si utilizas la extensión MySQLi, establecer una conexión con el servidor significa crear una instancia de la clase MySQLi. El constructor de la clase puede recibir seis parámetros, todos opcionales, aunque lo más habitual es utilizar los cuatro primeros:

- El nombre o dirección IP del servidor MySQL al que te quieres 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 puerto para establecer la conexión (3306 por defecto)
- El "Unix domain socket" (p.ej. /run/mysql/mysqld.sock)

Ya se vio antes como establecer una conexión:

```
<?php
// utilizando constructores y métodos de la programación orientada a objetos
$host='localhost'; $usuario='productos'; $pass='productos2021'; $bd='productos';
$conexion = new MySQLi($host, $usuario, $pass, $bd);
echo $conexion->server_info;
?>
```

También podemos crear un objeto con el constructor vacío y llamar al método **connect**:

```
<?php
$host='localhost'; $usuario='productos'; $pass='productos2021'; $bd='productos';
$conexion = new MySQLi();
$conexion->connect($host, $usuario, $pass, $bd);
?>
```

Es importante verificar que la conexión se ha establecido correctamente. Para comprobar el

error, en caso de que se produzca, puedes usar las siguientes propiedades (o funciones equivalentes) de la clase MySQLi:

- connect\_errno (o la función MySQLi\_connect\_errno ) devuelve el número de error o null si no se produce ningún error.
- connect\_error (o la función MySQLi\_connect\_error ) devuelve el mensaje de error o null si no se produce ningún error.

El siguiente código comprueba el establecimiento de una conexión con la base de datos "productos" y finaliza la ejecución si se produce algún error:

```
<?php
$host='localhost'; $usuario='productos'; $pass='productos2021'; $bd='productos';
@ $conexion = new MySQLi($host, $usuario, $pass, $bd);
$error = $conexion->connect_errno;
if ($error != null) {
    echo "<p>Error $error conectando a la base de datos: $conexion->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:

<https://www.php.net/manual/es/language.operators.errorcontrol.php>

Si una vez establecida la conexión, se quiere cambiar la base de datos se puede usar el método select\_db (o la función MySQLi\_select\_db de forma equivalente) para indicar el nombre de la nueva.

```
// utilizando el método connect
$conexion->select_db('otra_bd');
```

Cuando terminemos de trabajar con la Base de datos deberías utilizar el método close:

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

### Ejecución de Consultas

La forma más inmediata de ejecutar una consulta, si utilizas esta extensión, es el método query, equivalente a la función MySQLi\_query. Si se ejecuta una consulta de acción que no devuelve datos (como una 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 con la función MySQLi\_affected\_rows ).

```
$host='localhost'; $usuario='productos'; $pass='productos2021'; $bd='productos';
$conexion = new MySQLi($host, $usuario, $pass, $bd);
$error = $conexion->connect_errno;
if ($error == null) {
    $resultado = $conexion->query('DELETE FROM producto WHERE id=1');
    if ($resultado) {
        print "<p>Se han borrado $conexion->affected_rows registros.</p>\n";
        echo "<p>Número de registros afectados: ", $conexion->affected_rows, "</p>\n";
    }
    $conexion->close();
}
```

En el caso de ejecutar una sentencia SQL que sí devuelva datos (como un SELECT ), éstos se devuelven en forma de un objeto resultado (de la clase MySQLi\_result ). El método query tiene un parámetro opcional que afecta a cómo se obtienen internamente los resultados, pero no a la

forma de utilizarlos posteriormente. En la opción por defecto, MySQLi\_STORE\_RESULT, los resultados se recuperan todos juntos de la base de datos y se almacenan de forma local. Si cambiamos esta opción por el valor MySQLi\_USE\_RESULT, los datos se van recuperando del servidor según se vayan necesitando. Cuando se prevea que las consultas que vayamos a realizar van a devolver una gran cantidad de datos deberemos utilizar MySQLi\_USE\_RESULT,

```
$resultado = $conexion->query('SELECT * FROM producto', MySQLi_USE_RESULT);
```

Otra forma que se puede utilizar para ejecutar una consulta es el método `real_query` (o la función `MySQLi_real_query`), que siempre devuelve true o false según se haya ejecutado correctamente o no. Si la consulta devuelve un conjunto de resultados, se podrán recuperar de forma completa utilizando el método `store_result`, o según vaya siendo necesario gracias al método `use_result`:

<https://www.php.net/manual/es/MySQLi.real-query.php> (Functionally, using the MySQLi\_query function is identical to calling MySQLi\_real\_query() followed either by MySQLi\_use\_result() or MySQLi\_store\_result()).

Es importante tener en cuenta que los resultados obtenidos se almacenarán en memoria mientras los estés usando. Cuando ya no los necesites, los puedes liberar con el método `free` de la clase `MySQLi_result` (o con la función `MySQLi_free_result`):

```
$resultado->free();
```

### Transacciones

Para poder usar transacciones debemos asegurarnos de que estén soportadas por el motor de almacenamiento de la base de datos que vayamos a usar. La opción por defecto es que cada consulta individual se incluye dentro de su propia transacción. Se puede cambiar este comportamiento con el método **`autocommit`** (o con la función equivalente `MySQLi_autocommit`).

**`$conexion->autocommit(false);` //quedan deshabilitadas las transacciones automáticas**

Al deshabilitar las transacciones automáticas, las operaciones sobre la base de datos iniciarán una transacción que deberás finalizar utilizando:

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

```
// Inicia una transacción
$conexion->query('DELETE FROM producto WHERE id=2');
$conexion->query('UPDATE producto SET precio=245 WHERE id=3');
// Confirma los cambios
$conexion->commit();
```

```
<?php
//Ejemplo de transacción, transferir una unidad de un producto de una tienda a otra
//Deben de ejecutarse las 2 consultas o ninguna
@$conexion = new MySQLi("localhost", "tienda", "tienda2020", "tienda", 3316);
$error = $conexion->connect_errno;
if ($error != null) {
    print "<p>Se ha producido el error: $conexion->connect_error.</p>";
    exit();
}
// Definimos una variable para comprobar la ejecución de las consultas
$todo_bien = true;
// Iniciamos la transacción
$conexion->autocommit(false);
```



```

$sql = "UPDATE stock SET unidades=unidades-1 WHERE producto='3DSNG' AND tienda=1";
if ($conexion->query($sql) != true) $todo_bien = false;
$sql = "INSERT INTO stock (producto, tienda, unidades) VALUES ('3DSNG', 3, 1)";
if ($conexion->query($sql) != true) $todo_bien = false;
// Si todo fue bien, confirmamos los cambios
// y en caso contrario los deshacemos
if ($todo_bien == true) {
    $conexion->commit();
    print "<p>Los cambios se han realizado correctamente.</p>";
}else {
    $conexion->rollback();
    print "<p>No se han podido realizar los cambios.</p>";
}
$conexion->close();
unset($conexion);
?>

```

Con la opción por defecto, transacciones automáticas, no es posible revertir los cambios que se produzcan por la ejecución de las consultas.

### Realización de consultas que devuelven un conjunto de resultados

Al ejecutar una consulta que devuelve datos obtienes un objeto de la clase **MySQLi\_result**. Esta clase sigue los criterios de ofrecer un interface de programación dual, es decir, una función por cada método con la misma funcionalidad que éste. Para trabajar con los datos obtenidos del servidor, tienes 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.

Por ejemplo, para acceder al primer campo devuelto, podemos utilizar como clave el número 0 o su nombre indistintamente. A continuación se muestra un ejemplo:

```

$host='localhost'; $usuario='productos'; $pass='productos2021'; $bd='productos';
$conexion=new MySQLi($host, $usuario, $pass, $bd);
if ($conexion->errno==null) {
    //$conexion->select_db("productos"); esto no hace falta al indicar en la propia consulta la base de datos (productos)
    $resultado=$conexion->query('SELECT * FROM productos.producto');
    if ($resultado) {
        $producto=$resultado->fetch_array(); //obtenemos el primer reg
        $prod=$producto['id']; //podríamos usar tb. $producto[0]
        $precio=$producto['precio']; // tb. $producto[3]
        echo "El precio del producto id: $prod, nombre: ",$producto['nombre'], " es de: $precio.";
    } else {
        echo "Error en la consulta.";
    }
    $conexion->close();
}

```

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.

\* **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, podemos 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. Ejemplo:

```
$conexion=new MySQLi('localhost','productos','productos2021','productos');
if ($conexion->connect_errno!=null) {
    echo "Error conectando a la base de datos de productos: ",$conexion->connect_error;
    exit();
}
$conexion->set_charset("utf8");
$resultado=$conexion->query('SELECT * FROM producto');
if ($resultado==null) {
    echo "Error realizando consulta a la base de datos de productos";
    exit();
}
echo "<table border='1'>\n";
echo "<tr><th>ID</th><th>DESCRIPCIÓN</th><th>NOMBRE</th><th>PRECIO</th><th>IMAGEN</th></tr>";
while($producto=$resultado->fetch_assoc()) {
    echo "<tr>";
    foreach($producto as $campo) {
        echo "<td>$campo</td>";
    }
    echo "</tr>\n";
}
echo "</table>\n";
$conexion->close();
```

**Ejercicio 1.** Realizar un **CRUD** para el mantenimiento de la tabla de productos de una base de datos de una tienda *online*. De un producto se almacenan **id**, **descripción**, **nombre**, **precio** e **imagen**. Se aporta script de creación de la tabla de productos.

## 5.- PDO

Si se va a programar una aplicación que utilice como sistema gestor de bases de datos MySQL, la extensión MySQLi que se acaba de ver 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 tienes que cambiar el SGBD por otro distinto, tendrás que volver a programar gran parte del código de la misma. Por eso, antes de comenzar el desarrollo, es muy importante revisar las características específicas del proyecto. En el caso de que exista la posibilidad, presente o futura, de utilizar otro servidor como almacenamiento, deberás adoptar una capa de abstracción para el acceso a los datos. Existen varias alternativas como ODBC, pero sin duda la opción más recomendable en la actualidad es PDO.

El objetivo es que si llegado el momento necesitas cambiar el servidor de base de datos, las modificaciones que debas realizar en tu código sean mínimas. Incluso es posible desarrollar aplicaciones preparadas para utilizar un almacenamiento u otro según se indique en el momento de la ejecución, pero éste no es el objetivo principal de PDO. 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. Si esto fuera necesario, habría que programar una capa

de abstracción completa.

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

#### Establecimiento de conexiones

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

- Origen de datos (DSN - Data Source Name). Es una cadena de texto que indica qué controlador se va a utilizar y a continuación, separadas por el carácter dos puntos, los parámetros específicos necesarios para el controlador, como por ejemplo 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.

Por ejemplo, siguiendo con el ejemplo del punto anterior, podríamos establecer la conexión con el servidor mysql local y la base de datos de productos usando PDO, de la siguiente forma:

```
[$host,$usuario,$passwd,$bd]=[ 'localhost', 'productos', 'productos2021', 'productos' ];  
$conexion=new PDO("mysql:host=$host;dbname=$bd;charset=utf8",$usuario,$passwd);
```

Si usamos MySQL los parámetros específicos para utilizar en la cadena DSN van separados por ";" separados del prefijo **mysql:** y pueden ser 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.  
charset. Codificación a utilizar en la conexión  
unix\_socket. Socket de MySQL en sistemas Unix.

En el manual de PHP puedes consultar más información sobre los controladores existentes, los parámetros de las cadenas DSN y las opciones de conexión particulares de cada uno:

<http://es.php.net/manual/es/pdo.drivers.php>

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

Por ejemplo, para obtener la versión del servidor puedes hacer:

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

Y si quieres por ejemplo que te devuelva todos los nombres de columnas en mayúsculas:

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

#### Establecimiento de conexiones

Para ejecutar consultas SQL utilizando PDO, hay que diferenciar entre las sentencias que devuelven un conjunto de datos (SELECT) y las que no (INSERT, UPDATE, DELETE).

En el caso de las sentencias que no devuelven un conjunto de datos, el método **exec**

devuelve el número de registros afectados por la consulta. Por ejemplo:

```
$numRegs=$conexion->exec('DELETE FROM productos.producto WHERE id=1');
```

En el ejemplo anterior la variable `$numRegs` tomará el valor 1, en caso de que exista el producto con id 1 (clave primaria), indicando que se ha borrado dicho registro, o el valor 0, si no existe dicho registro.

En el caso de sentencias que generan un conjunto de resultados, utilizaremos el método **query**, que devuelve un objeto de la clase **PDOStatement**. Por ejemplo:

```
$conexion=new PDO("mysql:host=localhost;dbname=productos;charset=utf8","productos","productos2021");  
$resultado=$conexion->query("SELECT * FROM productos.producto")
```

PDO trabaja por defecto en modo "autocommit". Podemos gestionar las transacciones haciendo uso de 3 métodos:

- **beginTransaction**. Deshabilita el modo "autocommit" y comienza una nueva transacción, que finalizará cuando ejecutes uno de los dos métodos siguientes.

- **commit**. Confirma la transacción actual.

- **rollback**. Revierte los cambios llevados a cabo en la transacción actual.

Una vez que se ha hecho un *commit* o un *rollback* se volverá al modo de confirmación automática.

Ejemplo:

```
$todoBien=true;  
$conexion->beginTransaction();  
if ($conexion->exec('DELETE ...') == 0) { $todoBien=false; }  
if ($conexion->exec('UPDATE ...') == 0) { $todoBien=false; }  
.....  
if ($todoBien) {  
    $conexion->commit();  
} else {  
    $conexion->rollback();  
}
```

Hay que tener en cuenta si el motor de bases de datos que estamos usando soporta o no las transacciones, por ejemplo el motor *MyISAM* de MySQL, no las soporta. En este caso PDO ejecutará el método *beginTransaction* sin errores pero no será capaz de revertir los cambios si fuera necesario hacer un *rollback*.

A Continuación se muestra el mismo ejemplo que antes se mostró usando *MySQLi*, usando esta vez *PDO*:

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

```
// Definimos una variable para comprobar la ejecución de las consultas  
$todoBien = true;  
// Iniciamos la transacción  
$conexion->beginTransaction();  
$sql = 'UPDATE stock SET unidades=unidades-1 WHERE producto="3DSNG" AND tienda=1';  
if ($conexion->exec($sql) == 0) { $todoBien = false; }  
$sql = 'INSERT INTO stock (producto, tienda, unidades) VALUES ("3DSNG", 3, 1)';  
if ($conexion->exec($sql) == 0) { $todoBien = false; }  
// Si todo fue bien, confirmamos los cambios  
// y en caso contrario los deshacemos  
if ($todoBien) {  
    $conexion->commit();  
    print "<p>Los cambios se han realizado correctamente.</p>";  
} else {  
    $conexion->rollback();  
    print "<p>No se han podido realizar los cambios.</p>";  
}  
$conexion->close();  
?>
```

## Sentencias SELECT

Al igual que ocurría con *MySQLi*, con *PDO* tenemos varias posibilidades a la hora de acceder a los resultados de una consulta *SELECT*. Haremos uso de los métodos ***query***, para realizar la consulta y de ***fetch*** para acceder a los resultados. Por ejemplo:

```
[ $host,$usuario,$passwd,$db]=[ 'localhost','productos','productos2021','productos' ];
$conexion=new PDO("mysql:host=$host;dbname=$db;charset=utf8",$usuario,$passwd);
$resultado=$conexion->query("SELECT * FROM producto");
while($producto=$resultado->fetch()) {
    print_r($producto);
}
```

La conexión permanecerá abierta mientras exista el objeto de la clase *PDO*, lo cual quiere decir que si no destruimos el objeto (p.ej. haciendo uso de ***unset(\$conexion)***), la conexión se cerrará al terminar de ejecutarse nuestro script y por lo tanto destruirse el objeto. La clase *PDO* no tiene un método ***close*** para cerrar la conexión.

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.
- ***PDO::FETCH\_OBJ***. Devuelve un objeto cuyas propiedades se corresponden con los campos del registro.
- ***PDO::FETCH\_LAZY***. Devuelve tanto el objeto como el array con clave dual anterior. El objeto devuelto puede indexarse numéricamente, o accederse como array asociativo mediante clave, o con el operador *->* mediante variable miembro.
- ***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. Ejemplo:

```
[ $host,$usuario,$passwd,$db]=[ 'localhost','productos','productos2021','productos' ];
try {
    $conexion=new PDO("mysql:host=$host;dbname=$db;charset=utf8",$usuario,$passwd);
    $sql="SELECT * FROM producto";
    $resultado=$conexion->query($sql);
    if ($resultado==null) {
        exit("Error en consulta: $sql");
    }
} catch (PDOException $ex) {
    exit("<br/>Error: ".$ex->getMessage()."<br/>");
}
$resultado->bindColumn(1,$id); $resultado->bindColumn(2,$descripcion);
$resultado->bindColumn(3,$nombre); $resultado->bindColumn(4,$precio);
$resultado->bindColumn(5,$imagen);
echo "<table border='1'>";
echo "<thead>";
echo "<tr><th>ID</th><th>Descripción</th><th>Nombre</th><th>Precio</th><th>Imagen</th></tr>";
echo "</thead>";
echo "<tbody>";
while ($registro=$resultado->fetch(PDO::FETCH_BOUND)) {
    echo "<tr><td>$id</td><td>$descripcion</td><td>$nombre</td><td>$precio</td><td>$imagen</td></tr>";
}
echo "</tbody>";
echo "</table>";
```

## 6.- Excepciones

Cuando estamos escribiendo código con ayuda de un IDE como *Eclipse* o *Netbeans*, conforme lo vamos escribiendo se nos va avisando de errores de sintaxis, del uso de variables no definidas o de la asignación de variables que luego no son usadas. Pero hay errores que se pueden producir en tiempo de ejecución.

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. Por ejemplo, la constante *E\_NOTICE* hace referencia a avisos que pueden indicar un error al ejecutar el script, y la constante *E\_ERROR* engloba errores fatales que provocan que se interrumpa forzosamente la ejecución.

La lista completa de constantes la puedes consultar en el manual de PHP, donde también se describe el tipo de errores que representa: <http://es.php.net/manual/es/errorfunc.constants.php>.

La configuración inicial de cómo se va a tratar cada error según su nivel se realiza en *php.ini* el fichero de configuración de PHP. Entre los principales parámetros que se pueden ajustar están:

- ***error\_reporting*** . Indica qué tipos de errores se notificarán. Su valor se forma utilizando los operadores a nivel de bit para combinar 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 no se usan para desarrollo sino para producción.

Existen otros parámetros que podemos utilizar en *php.ini* para ajustar el comportamiento de PHP cuando se produce un error: <http://es.php.net/manual/es/errorfunc.configuration.php>.

Desde el propio código, se puede usar la función ***error\_reporting*** con las constantes anteriores para establecer el nivel de notificación en un momento determinado. Por ejemplo, si en algún lugar de tu código figura una división en la que exista la posibilidad de que el divisor sea cero, cuando esto ocurra obtendrás un mensaje de error en el navegador. Para evitarlo, puedes 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);
```

Usando la función ***error\_reporting*** solo se controla qué tipo de errores va a notificar PHP. Existe también la posibilidad de reemplazar la gestión de los mismos por una propia. Se puede especificar 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 como mínimo dos parámetros obligatorios (el nivel del error y el mensaje descriptivo) y hasta otros tres opcionales con información adicional sobre el error (el nombre del fichero en que se produce, el número de línea, y un volcado del estado de las variables en ese momento).

Por Ejemplo, al ejecutar el siguiente código:

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

Obtenemos en el navegador el siguiente resultado:

*Error de tipo no especificado: Undefined variable: dividendo.*

*Error de tipo no especificado: Undefined variable: divisor.*

*Error de tipo WARNING: Division by zero.*

La función ***restore\_error\_handler*** restaura el manejador de errores original de PHP (más concretamente, el que se estaba usando antes de la llamada a ***set\_error\_handler*** ).

### Excepciones

A partir de la versión 5 de PHP se introdujo un modelo de excepciones similar al existente en otros lenguajes de programación:

- 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.

Ver ejemplo al final del punto anterior.

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 puedes 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. En este caso, lo más común es que la extensión defina sus propios manejadores de errores heredando de la clase ***Exception***.

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. Es el 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 .

Es decir, que si quieres utilizar excepciones con la extensión PDO, debes configurar la conexión haciendo:

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

Por ejemplo, al ejecutar el siguiente código:

```
[$host,$usuario,$passwd,$bd]=['localhost','productos','productos2021','productos'];
$conexion=new PDO("mysql:host=$host;dbname=$bd;charset=utf8",$usuario,$passwd);
$conexion->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
try {
    $sql = "SELECT * FROM produc";
    $result = $conexion->query($sql);
} catch (PDOException $ex) {
    echo "Error ". $ex->getMessage(). "<br />";
}
```

Obtenemos en el navegador:

*Error SQLSTATE[42S02]: Base table or view not found: 1146 Table 'productos.produc' doesn't exist.*

Sin embargo si quitamos la sentencia:

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

No se nos muestra ningún mensaje de error.



## NOTAS

1) Para convertir bases de datos **mysql** a **sqlite** y viceversa:

```
sudo apt install python3-pip
sudo pip3 install mysql-to-sqlite3
sudo pip3 install sqlite3-to-mysql
sudo apt install php-sqlite3
sudo systemctl restart apache2
#convertir la base de datos de mysql productos a sqlite3
mysql2sqlite -f productos.sqlite -d productos -u productos --mysql-password productos2021
-f fichero que vamos a generar
-d base de datos de mysql a convertir
-u usuario con permisos sobre esa base de datos
--mysql-password password del usuario referido anteriormente
```

2) Ejemplo de acceso a una base de datos sqlite3 ([https://www.youtube.com/watch?v=Jib2AmRb\\_rk&feature=youtu.be](https://www.youtube.com/watch?v=Jib2AmRb_rk&feature=youtu.be)) usando PDO:


























```
function listaProductosPDOsqlite3() {
    $conexion=new PDO("sqlite:/var/www/phpdata/productos.sqlite");
    $resultado=$conexion->query("select * from producto");
    $resultado->bindColumn(1,$id); $resultado->bindColumn(2,$descripcion); $resultado->bindColumn(3,$nombre);
    $resultado->bindColumn(4,$precio); $resultado->bindColumn(5,$imagen);
    echo "<table border='1'>";
    echo "<thead>";
    echo "<tr><th>ID</th><th>Descripción</th><th>Nombre</th><th>Precio</th><th>Imagen</th></tr>";
    echo "</thead>";
    echo "<tbody>";
    while ($registro=$resultado->fetch(PDO::FETCH_BOUND)) {
        echo "<tr><td>$id</td><td>$descripcion</td><td>$nombre</td><td>$precio</td><td>$imagen</td></tr>";
    }
    echo "</tbody>";
    echo "</table>";
}
```

**Ejercicio 2.** Rehacer el Ejercicio 1 utilizando PDO e intentar que sea "fácilmente" sustituible el SGBD por sqlite3, es decir que si cambiamos de SGBD de mysql a sqlite3 o viceversa tengamos que hacer los mínimos cambios en el código.

**Ejercicio 3.** Crea el programa GESTISIMAL (GESTIÓN SIMplificada de Almacén) para llevar el control de los artículos de un almacén. De cada artículo se debe saber el código, la descripción, el precio de compra, el precio de venta y el stock (número de unidades). La entrada y salida de mercancía supone respectivamente el incremento y decremento de stock de un determinado artículo. Hay que controlar que no se pueda sacar más mercancía de la que hay en el almacén. El programa debe tener, al menos, las siguientes funcionalidades: listado, alta, baja, modificación, entrada de mercancía y salida de mercancía.

- Comprueba la existencia del código en el alta de artículos para evitar errores.
- El aspecto debería ser parecido al siguiente:

## GESTISIMAL

Código	Descripción	Precio de compra	Precio de venta	Margen	Stock				
h020	Barra acero 16mm. longitud 6m.	35.30	45.40	10.1	50	 Eliminar	 Modificar	 Entrada	 Salida
h007	barra para cortina 2,00 m.	10.30	22.33	12.03	5	 Eliminar	 Modificar	 Entrada	 Salida
h005	Caja tuercas 16mm.	21.00	25.05	4.05	20	 Eliminar	 Modificar	 Entrada	 Salida
h006	chapa galvanizada	10.50	20.55	10.05	3	 Eliminar	 Modificar	 Entrada	 Salida
m001	Estantería para pared.	25.30	30.60	5.3	5	 Eliminar	 Modificar	 Entrada	 Salida
Página 1 de 3		 Primera	 Anterior	Siguiente 	Última 				
Código	Descripción	Precio de compra	Precio de venta	Stock					
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	 Nuevo artículo			

### Ejercicio 4.

Modifica el programa anterior añadiendo las siguientes mejoras:

- Añade una nueva opción “Venta” que permita hacer una venta de varios artículos y emitir la factura correspondiente. Se debe preguntar por los códigos y las cantidades de cada artículo que se quiere comprar. Aplica un 21% de IVA.