DESARROLLO WEB EN ENTORNO SERVIDOR

2ºDAW(13/14) PHP_04

U.D.4. Utilización de Técnicas de Acceso a Datos

En el capítulo que vamos a ver, vamos a ver cómo gestionar la comunicación de las aplicaciones web con una base de datos, la cual vamos a utilizar para almacenar toda la información. Esto conlleva aprender la sintaxis necesaria para conectarnos con la base de datos y la de las sentencias SQL de inserción, actualización, borrado y selección. También veremos como utilizar la información recuperada de la base de datos, cómo realizar transacciones y cómo acceder a la base se de datos de manera serializable para que la base de datos no se quede en ningún momento inconsistente. Esto puede ocurrir debido a que la base de datos con frecuencia deberá soportar el acceso de varias personas a la vez. Por último explicaremos cómo obtener la información de otro tipo de fuentes que no sean la propia base de 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, vas a aprender a utilizar 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.

En esta unidad vas a ver cómo acceder desde PHP a bases de datos MySQL utilizando tanto PDO como la extensión nativa MySQLi. Previamente verás una pequeña introducción al manejo de MySQL, aunque para el seguimiento de esta unidad se supone que conoces el lenguaje SQL utilizado en la gestión de bases de datos relacionales.

Además, para el acceso a las funcionalidades de ambas extensiones deberás utilizar objetos. Aunque más adelante verás todas las características que nos ofrece PHP para crear programas orientados a objetos, debemos suponer también en este punto un cierto conocimiento de programación orientada a objetos. Básicamente, debes saber cómo crear y utilizar objetos.

En PHP se utiliza la palabra new para crear un nuevo objeto instanciando una clase:

```
$a = new A();
```

Y para acceder a los miembros de un objeto, debes utilizar el operador flecha →:

```
$a->fecha();
```

Es importante conocer las características más básicas de la utilización objetos en PHP. Características más básicas.

2. MySQL

MySQL es un sistema gestor de bases de datos (SGBD) relacionales. Es un programa de código abierto que se ofrece bajo licencia GNU GPL, aunque también ofrece una licencia comercial en caso de que quieras utilizarlo para desarrollar aplicaciones de código propietario. En las últimas versiones (a partir de la 5.1), se ofrecen, de hecho, varios productos distintos: uno de código libre (CommunityEdition), y otro u otros comerciales (Standard Edition, Enterprise Edition).

Incorpora múltiples motores de almacenamiento, cada uno con características propias: unos son más veloces, otros, aportan mayor seguridad o mejores capacidades de búsqueda. Cuando crees una base de datos, puedes elegir el motor en función de las características propias de la aplicación. Si no lo cambias, el motor que se utiliza por defecto se llama **MyISAM**, que es muy rápido pero a cambio no contempla integridad referencial ni tablas transaccionales. El motor **InnoDB**es un poco más lento pero sí soporta tanto integridad referencial como tablas transaccionales.

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 99, y añade además algunas extensiones propias.

En el siguiente enlace puedes consultar el manual en línea de MySQL.Manual en línea de MySQL.

Ejercicio 1:

Crea la BD **"dwes"** utilizando el fichero crear_BD_dwes.sql. Usuario "dwes" con contraseña "abc123."

Ejercicio 2:

Utiliza phpMyAdmin para ejecutar las consultas del siguiente fichero, que rellenan con datos las tablas de la base de datos "dwes". Esta información la utilizaremos en los próximos ejercicios.

3. Utilización de bases de datos MySQL en 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, podrás 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**.

3.1. 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:

```
// utilizando constructores y métodos de la 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);
```

En ambos casos, la variable $conexion es de tipo objeto. La utilización de los métodos y propiedades que aporta la clase **mysqli** normalmente produce un código más corto y legible que si utilizas llamadas a funciones.

Entre las mejoras que aporta a 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. En este fichero hay una sección específica para las opciones de configuración propias de cada extensión. Entre las opciones que puedes 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.

3.1.1. 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 número del puerto en que se ejecuta el servidor MySQL.

- ✓ El socket o la tubería con nombre (named pipe) a usar.

Si utilizas el constructor de la clase, para conectarte a la base de datos "dwes" puedes hacer:

```
// utilizando el constructor de la clase
$dwes = new mysqli('localhost', 'dwes', 'abc123.',
                   'dwes');
```

Aunque también tienes la opción de primero crear la instancia, y después utilizar el método connect para establecer la conexión con el servidor:

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

Por el contrario, utilizando el interface procedimental de la extensión:

```
// utilizando llamadas a funciones
$dwes = mysqli_connect('localhost', 'dwes', 'abc123.',
'dwes');
```

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.

Por ejemplo, el siguiente código comprueba el establecimiento de una conexión con la base de datos "dwes" y finaliza la ejecución si se produce algún error:

```
$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, como veremos posteriormente con más detalle, puedes anteponer a cualquier expresión el operador de control de errores @ para que se ignore cualquier posible error que pueda producirse al ejecutarla. Operador de control de errores @.

Si una vez establecida la conexión, quieres cambiar la base de datos puedes 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
$dwes->select_db('otra_bd');
```

La función **mysqli_select_db**, toma dos parámetros: el nombre de la base de datos, y opcionalmente, el nombre de la conexión de la base de datos. Si no se especifica este último, se emplea por defecto la última conexión que empleó **mysqli_connect.**

Una vez finalizadas las tareas con la base de datos, 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();
```

Ejercicio 3:

Crea un archivo **db_acceso.php**, que te servirá como plantilla para futuros aplicaciones web, en el que se establezca la información necesaria para el acceso a una base de datos.

Ejercicio 4:

Crea un archivo **db_conexion.php**, que te servirá como plantilla para futuros aplicaciones web, en el que se establezca la información necesaria para el acceso a una base de datos.

Ejercicio 5:

Una vez hayas establecido la conexión con la base de datos **test**, realiza la selección de ésta.

3.1.2. 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).

```
@ $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 sí devuelva datos (como un SELECT), éstos se devuelven en forma de un objeto resultado (de la clase mysqli_result). En el punto siguiente verás cómo se pueden manejar los resultados obtenidos.

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.

```
$resultado = $dwes->query('SELECT producto, unidades FROM
                           stock', MYSQLI_USE_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();
```

Para poder obtener las filas a partir del conjunto de resultados, se utiliza **mysqli_fetch_row**, que toma como parámetro el resultado de **mysqli_result**. Devuelve una fila a la vez desde la consulta, hasta que no hay más filas, momento en el cual muestra FALSE. Lo que se debe hacer por tanto, es ejecutar un bucle sobre el resultado de **mysqli_fech_row**

3.1.3. Transacciones

Una transacción es una secuencia de operaciones realizadas como una sola unidad lógica de trabajo. Una unidad lógica de trabajo debe exhibir cuatro propiedades, conocidas como propiedades de atomicidad, coherencia, aislamiento y durabilidad (ACID), para ser calificada como transacción.

Si necesitas utilizar transacciones deberás asegurarte de que estén soportadas por el motor de almacenamiento que gestiona tus tablas en MySQL. Si utilizas InnoDB, por defecto cada consulta individual se incluye dentro de su propia transacción. Puedes gestionar este comportamiento con el método autocommit (función mysqli_autocommit).

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

Al deshabilitar las transacciones automáticas, las siguientes 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.

```
...
$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.

3.1.4. Obtención y utilización de conjuntos de resultados

Ya sabes que 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.

```
$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.

- ✓ **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, puedes 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();
}
```

Ejercicio 6:

Crea una página web en la que se muestre el stock existente de un determinado producto en cada una de las tiendas. Para seleccionar el producto concreto utiliza un cuadro de selección dentro de un formulario en esa misma página. Puedes usar como base los ficheros plantillas que encontrarás en la intranet.

3.1.5. Consultas preparadas

Cada vez que se envía una consulta al servidor, éste debe analizarla antes de ejecutarla. Algunas sentencias SQL, como las que insertan valores en una tabla, 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 la extensión MySQLi de PHP, debes utilizar la clase **mysqli_stmt**. Utilizando el método **stmt_init** de la clase **mysqli** (o la función **mysqli_stmt_init**) obtienes un objeto de dicha clase.

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

Los pasos que debes seguir para ejecutar una consulta preparada son:

- ✓ Preparar la consulta en el servidor MySQL utilizando el método **prepare** (función **mysqli_stmt_prepare**).
- ✓ Ejecutar la consulta, tantas veces como sea necesario, con el método **execute** (función **mysqli_stmt_execute**).
- ✓ Una vez que ya no se necesita más, se debe ejecutar el método **close** (función **mysqli_stmt_close**).
- ✓ Por ejemplo, para 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();
```

El problema que ya habrás observado, es que de poco sirve preparar una consulta de inserción de datos como la anterior, si los valores que inserta son siempre los mismos. Por este motivo las consultas preparadas admiten parámetros. Para preparar una consulta con parámetros, en lugar de poner los valores debes indicar con un signo de interrogación su posición dentro de la sentencia SQL.

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

```
VALUES (?, ?)');
```

Y antes de ejecutar la consulta tienes 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 almacenas los valores a insertar en sendas variables, puedes hacer:

```
$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();
```

Cuando uses **bind_param** para enlazar los parámetros de una consulta preparada con sus respectivos valores, deberás usar siempre variables como en el ejemplo anterior. Si intentas utilizar literales, por ejemplo:

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

Obtendrás un error. El motivo es que los parámetros del método bind_param se pasan por referencia. Aprenderás a usar paso de parámetros por referencia en una unidad posterior.

El método **bind_param** permite tener una consulta preparada en el servidor MySQL y ejecutarla tantas veces como quieras 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. Observa el siguiente código:

```
$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();
```

En el manual de PHP tienes más información sobre consultas preparadas y la clase **mysqli_stmt**. Consultas preparas y la clase **mysqli_stmt**. (http://es.php.net/manual/es/class.mysqli-stmt.php)

Ejercicio 7:

A partir de la página web obtenida en el ejercicio anterior, añade la opción de modificar el número de unidades del producto en cada una de las tiendas.

Utiliza una consulta preparada para la actualización de registros en la tabla stock. No es necesario tener en cuenta las tareas de inserción (no existían unidades anteriormente) y borrado (si el número final de unidades es cero).

En esta ocasión es necesario crear un nuevo formulario en la página, en la sección donde se muestra el número de unidades por tienda. Cuando se envía ese formulario, hay que preparar la consulta y ejecutarla una vez por cada registro de la tabla stock (una vez por cada tienda en la que exista stock de ese producto).

Ejercicio 8:

Según la información que figura en la tabla stock de la base de datos dwes, la tienda 1 (CENTRAL) tiene 2 unidades del producto de código 3DSNG y la tienda 3 (SUCURSAL2) ninguno. Suponiendo que los datos son esos (no hace falta que los compruebes en el código), utiliza una transacción para mover una unidad de ese producto de la tienda 1 a la tienda 3.

Deberás hacer una consulta de actualización (para poner unidades=1 en la tienda 1) y otra de inserción (pues no existe ningún registro previo para la tienda 3). Observa el código de la solución. Comprueba que se ejecuta bien solo la primera vez, pues en ejecuciones posteriores ya no es posible insertar la misma fila en la tabla.

3.2. PHP Data Objects (PDO).

Si vas a programar una aplicación que utilice como sistema gestor de bases de datos MySQL, la extensión **MySQLi** que acabas 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.

La extensión PDO debe utilizar un driver o controlador específico para el tipo de base de datos que se utilice. Para consultar los controladores disponibles en tu instalación de PHP, puedes utilizar la información que proporciona la función phpinfo.

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.

3.2.1. 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). 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 por 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, podemos establecer una conexión con la base de datos 'dwes' creada anteriormente de la siguiente forma:

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

Si como en el ejemplo, se utiliza el controlador para MySQL, los parámetros específicos para utilizar en la cadena DSN (separadas unas de otras por el carácter 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.

Si quisieras indicar al servidor MySQL utilice codificación UTF-8 para los datos que se transmitan, puedes 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);
```

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

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

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

En el manual de PHP, las páginas de las funciones getAttribute y setAttribute te permiten consultar los posibles parámetros que se aplican a cada una.

getAttribute (http://es.php.net/manual/es/pdo.getattribute.php)

setAttribute (http://es.php.net/manual/es/pdo.setattribute.php)

3.2.2. Ejecución de consultas

Para ejecutar una consulta SQL utilizando PDO, debes 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, debes utilizar el método **query**, que devuelve un objeto de la clase **PDOStatement**.

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

Por defecto PDO trabaja en modo "autocommit", esto es, 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 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 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
```

Ten en cuenta que no todos los motores no soportan transacciones. Tal es el caso, como ya viste, del motor MyISAM de MySQL. En este caso concreto, PDO ejecutará el método beginTransaction sin errores, pero naturalmente no será capaz de revertir los cambios si fuera necesario ejecutar un **rollback**.

Ejercicio 9:

De una forma similar al anterior ejercicio de transacciones, utiliza PDO para repartir entre las tiendas las tres unidades que figuran en stock del producto con código PAPYRE62GB.

En esta ocasión, para comprobar si los cambios se hacen correctamente en la base de datos y confirmamos la transacción, se revisa el número de registros afectados por la ejecución de las consultas. Revisa el código de la página y comprueba que la segunda vez que intentas ejecutarlo no actualizará los datos, tal y como sucedía en el ejercicio equivalente de la extensión MySQLi.

3.2.3. Obtención y utilización de conjuntos de resultados.

Al igual que con la extensión MySQLi, en PDO tienes varias posibilidades para tratar con el conjunto de resultados devuelto por el método **query**. La más utilizada es el método **fetch** de la clase PDOStatement. 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.
- ✓ **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 = $resultado->fetch(PDO::FETCH_OBJ)) {
    echo "Producto ".$producto.": ".$unidades."<br />";
}
```

Ejercicio 10:

Modifica la página web que muestra el stock de un producto en las distintas tiendas, obtenida en un ejercicio anterior utilizando MySQLi, para que use PDO. Omite la gestión de errores, que veremos en el último punto de este tema.

3.2.4. Consultas preparadas

Al igual que con MySQLi, también utilizando PDO podemos preparar consultas parametrizadas en el servidor para ejecutarlas de forma repetida. El procedimiento es similar e incluso los métodos a ejecutar tienen prácticamente los mismos nombres.

Para preparar la consulta en el servidor MySQL, deberás 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 como en el caso anterior.

```
$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 un valor a los parámetros utilizando el método bindParam de la clase PDOStatement.. Si utilizas signos de interrogación para marcar los parámetros, el procedimiento es equivalente al método bindColumn que acabamos de ver.

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

Si utilizas parámetros con nombre, debes indicar ese nombre en la llamada a bindParam.

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

Tal y como sucedía con la extensión MySQLi, cuando uses **bindParam** para asignar los parámetros de una consulta preparada, deberás usar siempre variables como en el ejemplo anterior.

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 hayas indicado los parámetros) en la llamada a **execute**.

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

Puedes consultar la información sobre la utilización en PDO de consultas preparadas y la clase PDOStatement en el manual de PHP. Consultas preparadas y la clase PDOStatement. (http://es.php.net/manual/es/class.pdostatement.php)

Ejercicio 11:

Modifica el ejercicio sobre consultas preparadas que realizaste con la extensión MySQLi, el que modificaba el número de unidades de un producto en las distintas tiendas, para que utilice ahora la extensión PDO.

4. Errores y manejo de excepciones

A buen seguro que, conforme has ido resolviendo ejercicios o simplemente probando código, te has encontrado con errores de programación. Algunos son reconocidos por el entorno de desarrollo (NetBeans), y puedes corregirlos antes de ejecutar. Otros aparecen en el navegador en forma de mensaje de error al ejecutar el guión. 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 guión, 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. Lista completa de constantes en PHP.

(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 puedes 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. Parámetros que podemos ajustar en php.ini. (http://es.php.net/manual/es/errorfunc.configuration.php)

Desde código, puedes 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);
```

Al usar la función **error_reporting** solo controlas qué tipo de errores va a notificar PHP. A veces puede ser suficiente, pero para obtener más control sobre el proceso existe también la posibilidad de reemplazar la gestión de los mismos por la que tú definas. Es decir, puedes 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 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).

```
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 (más concretamente, el que se estaba usando antes de la llamada a **set_error_handler**).

4.1. Excepciones

A partir de la versión 5 se introdujo en PHP 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.

Por ejemplo, para lanzar una excepción cuando se produce una división por cero podrías 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();
}
```

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 (como en el ejemplo anterior) 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** (veremos cómo utilizar la herencia en una unidad posterior).

Utilizando la clase ErrorException es posible traducir los errores a excepciones.

Clase ErrorException. (http://es.php.net/manual/es/class.errorexception.php)

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. 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:

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

Por ejemplo, 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
```

Ejercicio 12:

Agrega control de excepciones para controlar los posibles errores de conexión que se puedan producir en el último ejercicio, mostrando en la parte inferior de la pantalla los errores que se produzcan.