
new PDO
PHP

Sumario

PHP Data Objects (PDO)

1. INTRODUCCIÓN A LAS BASES DE DATOS.

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. Como ya vimos, es la letra "M" que figura en los acrónimos AMP y XAMPP.

En esta unidad vas a ver cómo acceder desde PHP a bases de datos MySQL utilizando PDO.

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 (Community Edition), 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 (*característica de las bases de datos que permite crear relaciones válidas entre dos registros de la misma o de diferentes tablas, y definir las operaciones necesarias para mantener la validez de las relaciones cuando se borra o modifica alguno de los registros*) 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.

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 (*revisión del estándar ANSI SQL del año 1999, que agrega a la revisión anterior (SQL2 o SQL 92) disparadores, expresiones regulares, y algunas características de orientación a objetos*), y añade además algunas extensiones propias.

En Linux, la instalación de MySQL se divide básicamente en dos paquetes que puedes instalar de forma individual según tus necesidades:

- **mysql-server**. Es el servidor en sí. Necesitas instalar este paquete para gestionar las bases de datos y permitir conexiones desde el equipo local o a través de la red.
- **mysql-client**. Son los programas cliente, necesarios para conectarse a un servidor MySQL. Solo necesitas instalarlos en aquel o aquellos equipos que se vayan a conectar al SGBD (en nuestro caso, las conexiones se realizarán normalmente desde el mismo equipo en el que se ejecuta el servidor).

Una vez instalado, puedes gestionar la ejecución del servicio de la misma forma que cualquier otro servicio del sistema:

```
sudo service mysql status // también start, stop, restart
```

En una instalación típica, el usuario *root* no tiene por defecto contraseña de acceso al servidor. Es importante asignarle una por razones de seguridad:

```
mysqladmin -u root password nueva-contraseña
```

El servidor se ejecuta por defecto en el Puerto TCP 3306. Esto lo debes tener en cuenta para permitir el acceso a través del cortafuegos en configuraciones en red.

2.1.- Herramientas de administración.

Existen muchas herramientas que permiten establecer una conexión con un servidor MySQL para realizar tareas de administración. Algunas herramientas se ejecutan en la línea de comandos, otras presentan un interface gráfico basado en web o propio del sistema operativo en que se ejecuten. Unas se incluyen con el propio servidor, y otras es necesario obtenerlas e instalarlas de forma independiente. Las hay que están orientadas a algún propósito concreto y también que permiten realizar varias funciones de administración.

Repasemos lo básico. Con el servidor MySQL se incluyen algunas herramientas de administración en línea de comandos, entre las que debes conocer:

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

Estas herramientas comparten unas cuantas opciones relativas al establecimiento de la conexión con el servidor. Muchas de estas opciones tienen también una forma abreviada:

`--user=nombre_usuario (-u nombre_usuario)`. Indica un nombre de usuario con permisos para establecer la conexión. Si no se especifica se usará el nombre de usuario actual del sistema operativo.

`--password=contraseña (-p contraseña)`. Contraseña asociada al nombre de usuario anterior. Si se utiliza la opción abreviada, debe figurar justo a continuación de la letra p, sin espacios intermedios. Si es necesario introducir una contraseña y no se indica ninguna, se pedirá para establecer la conexión.

`--host=equipo_servidor (-h equipo servidor):` Si no se indica pondrá el que tenga por defecto en my.conf

Por ejemplo, para establecer una conexión al servidor local con la herrami-enta mysql, podemos hacer:

```
mysql –u root –p
```

Conviene no indicar nunca la contraseña en la misma línea de comandos. En caso de que la cuenta esté convenientemente protegida por una contraseña, es mejor utilizar solo la opción -p como en el ejemplo anterior. De esta forma, la herramienta solicita la introducción de la contraseña y ésta no queda almacenada en ningún registro como puede ser el historial de comandos del sistema.

De entre el resto de herramientas de administración independientes que podemos utilizar con MySQL, podemos destacar tres:

- **MySQL Workbench** es una herramienta genérica con interface gráfico nativo que permite administrar tanto el servidor como las bases de datos que éste gestiona. Ha sido desarrollada por los creadores de MySQL y se ofrece en dos ediciones, una de ellas de código abierto bajo licencia GPL. Muy chulo muy guay, hace de todo pero un autentico tanque: lento y pesado
  http://dev.mysql.com/doc/workbench/en/index.html

- **phpMyAdmin** es una aplicación web muy popular para la administración de servidores. MySQL. Presenta un interface web de administración programado en PHP bajo licencia GPL. Su objetivo principal es la administración de las bases de datos y la gestión de la información que maneja el servidor. Ventaja: viene en todas la instalaciones y es más ligero que el anterior.
  Desventaja: es una autentica ….
  http://www.phpmyadmin.net/

- HeidiSQL: cliente mysql, ligero, rápido, sencillo, el que recomiendo usar. Además tiene versión portable. Desventaja: Sólo para windows aunque con wine funciona.
  https://www.heidisql.com/

2.2. mysql y mysqladmin.

La forma más habitual de utilizar la herramienta mysql es en modo interactivo. Una vez te conectas al servidor MySQL, te presenta una línea de órdenes. En esa línea de órdenes puedes introducir sentencias SQL, que se ejecutarán sobre la base de datos seleccionada, y algunos comandos especiales. Las sentencias SQL deben terminar en el carácter "`;`". Entre los comandos especiales que puedes usar están:

- **connect**: Establece una conexión con un servidor MySQL.
  - **use**: Permite seleccionar una base de datos.
- **exit** o **quit**. Termina la sesión interactiva con mysql.
- **help**:. Muestra una pantalla de ayuda con la lista de comandos disponibles.

Por ejemplo, si cuando estás utilizando la herramienta quieres seleccionar la base de datos "dwes", debes hacer:
mysql> use dwes

Las sentencias SQL que teclees a partir de ese instante se ejecutarán sobre la base de datos "dwes".

Para comprobar la sintaxis de las sentencias SQL admitidas por MySQL, puedes consultar la documentación en línea.
http://dev.mysql.com/doc/refman/5.0/es/sql-syntax.html

También puedes usar mysql en modo de procesamiento por lotes (*ejecución de un conjunto de tareas repetitivas o de forma consecutiva, sin la supervisión directa del usuario*), para ejecutar sobre un servidor MySQL todas las sentencias almacenadas en un fichero de texto (normalmente con extensión .sql). Por ejemplo:
mysql –u root –pabc123. < crear_bd_dwes.sql

Qué buenos tiempos en primero, ¿verdad?
Pero vamos a lo importante.

3. PHP Data Objects (PDO).

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

Sin embargo, **si en el futuro 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 (PHP Data Objects) permite acceder a distintas bases de datos utilizando las misma funciones, lo que facilita la portabilidad. En PHP 5 existen drivers para acceder a las bases de datos más populares (MySQL, Oracle, MS SQL Server, PostgreSQL, SQLite, Firebird, DB2, Informix, etc).
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 ex-

tensión tienes que emplear los objetos que ofrece, con sus métodos y propiedades. No existen funciones alternativas.

En esta lección se explica el acceso a MySQL y SQLite mediante PDO. La extensión PDO no evalúa la corrección de las consultas SQL. Aunque tampoco lo hace mysqli hasta que no ejecute
Resumen de Obtención de la información:

---

4. Antes de utilizar PDO

Para poder acceder a MySQL mediante PDO, debe estar activada la extensión php_pdo_mysql en el archivo de configuración php.ini

Para cada base de datos existe un manejador (driver) específico, que debe estar habilitado en el archivo de configuración de PHP (el archivo *php.ini*). Los manejadores se administran mediante extensiones de PHP, las cuales tienen nombres finalizando con *dll* en Windows y con *so* en Unix.

```
extension=php_pdo.dll
extension=php_pdo_firebird.dll
extension=php_pdo_informix.dll
extension=php_pdo_mssql.dll
extension=php_pdo_mysql.dll
extension=php_pdo_oci.dll
extension=php_pdo_oci8.dll
extension=php_pdo_odbc.dll
extension=php_pdo_pgsql.dll
extension=php_pdo_sqlite.dll
```

Todas estas extensiones deben existir en el directorio de *extensiones* de PHP. Generalmente las extensiones *php_pdo* y *php_pdo_sqlite* estarán habilitadas por omisión. De todas formas, revíselas.

5. Conexión con la base de datos

Para conectar con la base de datos hay que crear una instancia de la clase PDO (una instancia de conexión), que se utiliza en todas las consultas posteriores. En cada página php que incluya consultas a la base de datos es necesario conectar primero con la base de datos.
Si no se puede establecer la conexión con la base de datos, puede deberse a que la base de datos no esté funcionando, a que los datos de usuario no sean correctos, a que no esté activada la extensión pdo o (en el caso de SQLite) que no exista el camino donde se guarda la base de datos.

En el caso de MySQL, la creación de la clase PDO, al igual que en mysqli, incluye el nombre del servidor, el nombre de usuario y la contraseña.

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 (mysql: pgsql: oci:...), 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 'conexion' creada anteriormente de la siguiente forma:

```
$conexion = new PDO('mysql:host=localhost;dbname=base_datos', 'root',
'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.
- **db_name:** Nombre de la base de datos.
- **unix_socket:** Socket de MySQL en sistemas Unix.

En otras palabras el primer parámetro sera una cadena de texto (SIEMPRE SEA CUAL SEA EL TIPO DE CONEXION) y el resto usuario y contraseña.
Por tanto en dicha cadena tendremos que poner, al menos: 'mysql:host=IP' y el resto (db_name, port, unix_socket) serán opcionales.

Observa el siguiente ejemplo, de como se debería hacer una conexión en mysql

```
try {
    $dsn = "mysql:host=localhost;dbname=$dbname";
    $conexion = new PDO($dsn, $user, $password);
      return $conexion;
} catch (PDOException $e){
    echo $e->getMessage();
}
```

¿No ves algo llamativo? Sí, el try catch

6.Try catch en conexiones con la base de datos en PDO

PDO maneja los errores en forma de excepciones, por lo que la conexión siempre ha de ir encerrada en un bloque try/catch. Se puede (y se debe) especificar el modo de error estableciendo el atributo **error mode**:

```
$conexion->setAttribute(PDO::ATTRR_ERRMODE, PDO::ERRMODE_SILENT);
$conexion->setAttribute(PDO::ATTRR_ERRMODE, PDO::ERRMODE_WARNING);
$conexion->setAttribute(PDO::ATTRR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

No importa el modo de error, si existe un fallo en la conexión siempre producirá una excepción, por eso siempre se conecta con try/catch.

- **PDO::ERRMODE_SILENT**. Es el **modo de error por defecto**. Si se deja así habrá que comprobar los errores de forma parecida a como se hace con **mysqli**. Se tendrían que emplear PDO::errorCode() y PDO::errorInfo() o su versión en PDOStatement PDOStatement::errorCode() y PDOStatement::errorInfo().
- **PDO::ERRMODE_WARNING**. Además de establecer el código de error, PDO emitirá un mensaje E_WARNING. Modo empleado para depurar o hacer pruebas para ver errores sin interrumpir el flujo de la aplicación.
- **PDO::ERRMODE_EXCEPTION**. Además de establecer el código de error, PDO lanzará una excepción PDOException y establecerá sus propiedades para luego poder reflejar el error y su información. Este modo se emplea en la mayoría de situaciones, ya que permite manejar los errores y a la vez esconder datos que podrían ayudar a alguien a atacar tu aplicación. Este es el que nosotros usaremos.

El **modo de error** se puede aplicar con el método **PDO::setAttribute** o mediante un **array de opciones al instanciar PDO**:

```
// Con un array de opciones
$user='root';
$password='1234';
try {
    $dsn = "mysql:host=localhost;dbname=$dbname";
    $options = array(
      PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION
    );
    $conexion = new PDO($dsn, $user, $password, $options);
      return $conexion;
} catch (PDOException $e){
    echo  $e->getMessage();
    die('Error, no puedo conectarme a la base de datos');

}
// Con el método PDO::setAttribute
try {
    $dsn = "mysql:host=localhost;dbname=$dbname";
    $conexion = new PDO($dsn, $user, $password);
    $conexion->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    return $conexion;
} catch (PDOException $e){
    echo $e->getMessage();
    die('Error, no puedo conectarme a la base de datos');
}
```

7.Otras opciones de conexión

Por ejemplo, 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",
            PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION);
$conexion = new PDO('mysql:host=localhost;dbname=basedato', 'root',
            'abc123.', $opciones);
```

Existen más opciones aparte de el **modo de error ATTR_ERRMODE**, algunas de ellas son:

- **ATTR_CASE**. Fuerza a los nombres de las columnas a mayúsculas o minúsculas (CASE_LOWER, CASE_UPPER).
- **ATTR_TIMEOUT**. Especifica el tiempo de espera en segundos.
- **ATTR_STRINGIFY_FETCHES**. Convierte los valores numéricos en cadenas.

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);
 echo "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);
```

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.
http://es.php.net/manual/es/pdo.getattribute.php
http://es.php.net/manual/es/pdo.setattribute.php

```
// FUNCIÓN DE CONEXIÓN CON LA BASE DE DATOS MYSQL

function conectar()
{
    try {
        $opciones = array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8",
            PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION);
        $conexion = new PDO("mysql:host=localhost;dbname=BBDD", "root",
            "1234",$opciones);
        return($conexion);
    } catch (PDOException $e) {
        echo("Error grave");
        print "<p>Error: No puede conectarse con la base de datos.</p>\n";
        print "<p>Error: " . $e->getMessage() . "</p>\n";
        die();
    }
}

// EJEMPLO DE USO DE LA FUNCIÓN ANTERIOR
// La conexión se debe realizar en cada página que acceda a la base de datos
$conexion = conectar();
```

8. Desconexión con la base de datos

Para desconectar con la base de datos hay que destruir el objeto PDO. Si no se destruye el objeto PDO, PHP lo destruye al terminar la página.

```
$conexion = null;
```

---

9. Ejecución de Consultas en PDO.

Una vez realizada la conexión a la base de datos, las operaciones se realizan a través de consultas.

Para ello se puede usar el método query, el cual ejecuta consultas a la base de datos.
Por ejemplo:

```
<?php
include_once "base_datos.php";
$conexion=conectar();
$sentencia = $conexion->query ("CREATE TABLE A(a0 int);");
echo ($sentencia)?"Tabla bien creada":"Error, al crear la tabla";

$nombre="Juan"; $apellidos="Lopez";
$sentencia = $conexion->query ("INSERT INTO personas(nombre, apellidos)
            values ($nombre, $apellidos) ")
echo ($sentencia)?"Insercion Correcta":"Error, insercion incorrecta";

$sentencia = $conexion->query("SELECT * FROM personas;");
$personas = $sentencia->fetchAll(PDO::FETCH_OBJ);
var_dump($personas);
?>
```

Sencillo. Eso sí tened en cuenta, que:

- Consultas de definición de estructura (CREATE, ALTER...) devuelve true si se realiza correctamente y false si no.
- Para las operaciones INSERT, DELETE, y UPDATE, query devuelve true si se realiza correctamente y false si no.
- Si embargo, en el caso del SELECT, SHOW o DESCRIBE, que son las consultas devolución de datos, devuelve false si falla y un manejador de resultados con los datos de la consulta.

10. Consultas de recuperación de datos (SELECT)

Para recuperar datos, primero debo ejecutar una instrucción SELECT, y utilizar los métodos:
fetch y **fetchAll**.
Veamos un ejemplo de fetch.

```
<?php
include_once "base_datos.php";
$conexion=conectar();
$sentencia = $conexion->query("SELECT nombre, apellidos FROM personas;");

while ($fila = $resultado->fetch()) {
    echo "Nombre: ".$fila['nombre']." Apellido: ".$fila['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 unos 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 genérico cuyas propiedades se corresponden con los campos del registro.
PDO::FETCH_LAZY. Devuelve tanto del objeto como el array con clave dual anterior.
PDO::FETCH_BOUND. Devuelve true y asigna los valores del registro a variables según se indique en bind_param.
PDO::FETCH_CLASS. Parecido al _OBJ, pues devuelve las propiedades en un objeto, pero en este caso no es genérico. Lo veremos en profuNdidad más adelante.

Ejemplo FETCH_OBJ

```
$conexion = new PDO("mysql:host=localhost;dbname=basedatos", $usuario,
$password);
$resultado = $conexion->query("SELECT producto, unidades FROM stock");

while ($fila = $resultado->fetch(PDO::FETCH_OBJ)) {
       echo "Producto ".$fila->producto.": ".$fila->unidades."<br />";
}
```

Vamos a ver que diferencia hay entre las distintas opciones:
Si no hay ninguna línea que leer en el resultado, estas funciones devuelven NULL.

Consulta	**SELECT * FROM artículos**		
Columnas	**id**	**articulo**	**precio**
1a línea del resultado	1	Albaricoques	35.5

Resultado de un fetch (lectura con diferentes funciones)

PDO::FETCH_NUM		**PDO::FETCH_ASSOC:**		**PDO::FETCH_BOTH**	
Clave	**Valor**	**Clave**	**Valor**	**Clave**	**Valor**
0	1	**id**	1	**id**	1
1	Albaricoques	**articulo**	Albaricoques	**0**	1
2	35.5	**precio**	35.5	**texto**	Albaricoques
				1	Albaricoques
				precio	35.5
				2	35.5

Tanto fetch como fetchAll devuelve un objetos, con un atributo por columna, el nombre del atributo corresponde al nombre de la columna. Esta función ofrece parámetros adicionales que permiten precisar el nombre de la clase que se va a instanciar y pasar parámetros al constructor de la clase (ver la documentación).

En caso de utilización de un alias de columna en la consulta SELECT, por ejemplo:
SELECT AVG(precio-sin-iva) precio_medio FROM colección, es el alias de columna el que se utiliza como clave o nombre de atributo. En el caso anterior precio_medio.

Ejemplo con la base de datos proveedores

```
<?php
// Inclusión del archivo que contiene la definición de
// la función 'mostrar_matriz'.
include 'base_datos.php';
$conexion=conectar();
// Ejecución de una consulta SELECT.
$sql = 'SELECT * FROM pieza';
$consulta = $conexion->query($sql);

// Primer fetch con PDO::FETCH_NUM
$linea = $consulta->fetch(PDO::FETCH_NUM);
echo "<pre>";
echo "<br>Con fetch PDO::FETCH_NUM";
print_r($linea);

// Segundo fetch con FETCH_ASSOC.
$linea = $consulta->fetch(PDO::FETCH_ASSOC);
echo "<br>Con fetch_ASSOC";
print_r($linea);

// Tercer fetch con PDO::FETCH_BOTH
$linea = $consulta->fetch(PDO::FETCH_BOTH);
echo "<br>Con fetch_BOTH";
print_r($linea);

// Cuarto fetch con FETCH_OBJ.
$linea = $consulta->fetch(PDO::FETCH_OBJ);
echo "<br><p /><b> CON FETCH_OBJ</b><br />";
echo "\$linea->NUMPIEZA = $linea->NUMPIEZA<br />";
echo "\$linea->NOMPIEZA = $linea->NOMPIEZA<br />";
echo "\$linea->PRECIOVENT = $linea->PRECIOVENT<br />";
echo "</pre>";
// Desconexion.
$conexion=null;
?>
```

Al ejecutar el código anterior se vería lo siguiente

```
Con fetch_row:
Array(
    [0] => 2
    [1] => Teclado ErgonOmico
    [2] => 15000
)

Con fetch_assoc
Array(
    [NUMPIEZA] => 3
    [NOMPIEZA] => Teclado ps/2
    [PRECIOVENT] => 2570
)

Con fetch_array
Array(
    [0] => 4
    [NUMPIEZA] => 4
    [1] => RatOn ps/2
    [NOMPIEZA] => RatOn ps/2
    [2] => 1500
    [PRECIOVENT] => 1500
```

```
)
Con fetch_OBJ
$linea->NUMPIEZA = A-1001-L
$linea->NOMPIEZA = RATON NEGRO 3BOTONES
$linea->PRECIOVENT = 300
```

En este ejemplo permite observar:

- Los diferentes modos de recuperación de una línea de resultado.
- El hecho de que a cada fetch, el puntero interno avance, y que el fetch siguiente devuelva, por tanto, la siguiente línea, hasta haber pasado por todas las líneas.

Ejemplo de código para la lectura de una línea

Un primer tipo de lectura consiste a menudo en leer una sola línea de información en una matriz o varias matrices con uniones: obtención de información sobre el usuario que acaba de conectarse, ficha de descripción de un artículo…

Para 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 = $dwes->query('SELECT producto, unidades FROM stock WHERE
unidades<2');
//CON EL NULL
while(($stock = $resultado->fetch(PDO::FETCH_OBJ)!=null){
      print "<p>Producto $stock->producto: $stock->unidades unidades.</p>";
}
//OMITIENDO EL NULL
while($stock = $resultado->fetch(PDO::FETCH_OBJ)){
      print "<p>Producto $stock->producto: $stock->unidades unidades.</p>";
}
```

Método $resultado->fetchAll

El método fetch_all lee todas las líneas del resultado en una matriz.

Sintaxis

```
$resultado->fetchAll([entero tipo])
```

Siendo el tipo las diversas opciones tipo fetch:
PDO::FETCH_ASSOC:, PDO::FETCH_NUM … ya sabes, las de antes.

El método fetchAll()l devuelve una matriz multidimensional. La matriz principal es una matriz con índices enteros que contiene una línea para cada línea del resultado. La matriz secundaria contiene los valores de las columnas; el tipo de esta matriz depende del segundo parámetro:

¿Qué método utilizar?

Para leer una sola línea, el método fetch es válido desde el punto de vista del rendimiento. Las opciones FETCH_BOTH, FETCH_ASSOC y FETCH_OBJ permiten utilizar el nombre de las columnas de la consulta y hacer el código más legible.

Para leer todas las líneas, se puede utilizar el método fetchAll si desea pasar por una matriz intermedia y la estructura de la matriz le sirve. Pero si no es el caso, puede utiliza el fetch y un bucle.

Otra manera de sacar los datos es usando el métodos bindColumn, (no recomendado) aquí tenéis otro ejemplo:

```
$conexion = new PDO("mysql:host=localhost;dbname=basedatos", "root",
"abc123.");
$resultado = $conexion->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 />";
}
```

11. Sentencias preparadas PDO

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.
Una consulta preparada se utiliza mucho, ya que tiene varias ventajas. En primer lugar, evita la inyección SQL, ya que no se pueden ejecutar los datos de la consulta. En segundo lugar, si quiere ejecutar varias veces seguidas una consulta de tipo INSERT con distintos valores, no necesita reconstruir la consulta cada vez. Simplemente debe unir los nuevos valores y ejecutarla de nuevo. Por lo tanto, una consulta preparada es más segura y a veces más rápida que una consulta no preparada.
Para evitar ataques de inyección SQL (en la lección Inyecciones SQL se comentan los ataques más elementales), se recomienda el uso de sentencias preparadas, en las que PHP se encarga de "desinfectar" los datos en caso necesario.

Sentencias preparadas

El método para efectuar consultas

1. Preparar la consulta con PDO->prepare($consulta)
2. Ejecutarla con PDO->execute(array(parámetros)), que devuelve el resultado de la consulta.
3. Dependiendo del tipo de consulta, el dato devuelto debe tratarse de formas distintas, como se ha explicado en el apartado anterior.

La **clase PDOStatement** es la que trata las **sentencias SQL**. Una **instancia de PDOStatement** se crea cuando se llama a *PDO->prepare()*, y con ese objeto creado se llama a métodos como *bindParam()* para pasar valores o *execute()* para ejecutar sentencias. PDO facilita el uso de sentencias preparadas en PHP, que mejoran el rendimiento y la seguridad de la aplicación. Cuando se **obtienen, insertan o actualizan datos**, el esquema es:

PREPARE -> [BIND] -> EXECUTE.

Se pueden indicar los parámetros en la sentencia con un interrogante "**?**" o mediante un **nombre específico**.

Si la consulta incluye datos introducidos por el usuario, los datos pueden incluirse directamente en la consulta, pero en ese caso, PHP no realiza ninguna "desinfección" de los datos, por lo que estaríamos corriendo riesgos de ataques:

```
$nombre = $_REQUEST["nombre"];
$apellidos = $_REQUEST["apellidos"];

$consulta = "SELECT COUNT(*) FROM Personas
    WHERE nombre=$nombre AND apellidos=$apellidos";
 // DESACONSEJADO: PHP NO DESINFECTA LOS DATOS
$result = $conexion->prepare($consulta);
$result->execute();
if (!$result) {
```

```
    print "<p>Error en la consulta.</p>\n";
 ...
```

Para que PHP desinfecte los datos, estos deben enviarse al ejecutar la consulta, no al prepararla. Para ello es necesario indicar en la consulta la posición de los datos.

Esto se puede hacer:

- mediantes interrogantes (?) (a evitar)

En este caso la matriz debe incluir los valores que sustituyen a los interrogantes en el mismo orden en que aparecen en la consulta, como muestra el siguiente ejemplo:

```
$nombre    = $_REQUEST["nombre"];
$apellidos = $_REQUEST["apellidos"];

$consulta = "SELECT COUNT(*) FROM $dbTabla
    WHERE nombre=?
    AND apellidos=?";
$result = $conexion->prepare($consulta);
$result->execute(array($nombre, $apellidos));
if (!$result) {
    print "<p>Error en la consulta.</p>\n";
 …
```

Ejemplo más completo (con bindparam)

```
// Prepare
try{
      $opciones = array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES
utf8",PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION);
      $conexion = new PDO("mysql:host=localhost;dbname=basedatos", "root",
"abc123.", $opciones);
} catch (Exception $e) {
  die("Unable to connect: " . $e->getMessage());
}
try{
$consulta_prep = $conexion->prepare("INSERT INTO Vendedores (nombre, ciudad)
VALUES (?, ?)");
// Bind
$nombre = "Peter";
$ciudad = "Madrid";
$consulta_prep->bindParam(1, $nombre);
$consulta_prep->bindParam(2, $ciudad);
// Excecute
$consulta_prep->execute();
// Bind
$nombre = "Martha";
$ciudad = "Cáceres";
$consulta_prep->bindParam(1, $nombre);
$consulta_prep->bindParam(2, $ciudad);
// Execute
$consulta_prep->execute();
}
catch (Exception $e) {
  echo "Failed: " . $e->getMessage();
}
```

Observa que usamos el try catch que es la estructura que quiero que utilicéis.

Mas ejemplos: Este con el array para los parámetros.

```
<?php
try {
  $conexion = new PDO('sqlite:test.db'); //OTRO TIPO DE CONEXION
  echo "Connected\n";
} catch (Exception $e) {
  die("Unable to connect: " . $e->getMessage());
}

try {
  $con_prep = $conexion->prepare("INSERT INTO countries (name, area,
population, density)VALUES (?, ?, ?, ?)");

  $conexion->beginTransaction();
  $con_prep->execute(array('Nicaragua', 129494, 602800, 46.55));
  $con_prep->execute(array('Panama', 78200, 3652000, 46.70));
  $conexion->commit();

} catch (Exception $e) {
  $conexion->rollBack();
  echo "Failed: " . $e->getMessage();
}
?>
```

En este ejemplo usamos arrays, me parecen más cómodo. Además fijate que hay una instrucción beginTransaction(), esto lo veremos más adelante, pero imagina para que sirve.

- mediante parámetros (:parámetro) En este caso podemos definir dos posibili-dades:

1º ASIGNACIÓN MEDIANTE ARRAYS

En este caso la matriz debe incluir los nombres de los parámetros y los valores que sustituyen a los parámetros (el orden no es importante), como muestra el siguiente ejemplo:

```
$nombre    = $_REQUEST["nombre"];
$apellidos = $_REQUEST["apellidos"];

$consulta = "SELECT COUNT(*) FROM PERSONAS
    WHERE nombre=:nombre
    AND apellidos=:apellidos";
$result = $conexion->prepare($consulta);
$result->execute(array(":nombre" => $nombre, ":apellidos" => $apellidos));
if (!$result) {
    print "<p>Error en la consulta.</p>\n";
 ...
```

Mas ejemplos:

```
// Prepare:
$consulta_prep = $conexion->prepare("INSERT INTO Clientes (nombre, ciudad)
VALUES (:nombre, :ciudad)");
$nombre = "Luis";
$ciudad = "Barcelona";
// Bind y execute:
if($consulta_prep->execute(array(':nombre'=>$nombre, ':ciudad'=>$ciudad))) {
    echo "Se ha creado el nuevo registro!";
```

Ejemplo más completo

```
<?php
include 'base_datos.php';

try {
  $con_prep = $conexion->prepare("INSERT INTO countries (name, area,
population, density)VALUES (:name, :area, :population, :density)");

  $conexion->beginTransaction();
  $name = 'Nicaragua'; $area = 129494; $population = 602800; $density =
46.55;
  $con_prep->execute(array(':name'=> $name, ':area'=> $area, ':population'=>
      $population, ':density'=> $density ));
  $name = 'Panama'; $area = 78200; $population = 3652000; $density = 46.70;
  $con_prep->execute(array(':name'=> $name, ':area'=> $area, ':population'=>
      $population, ':density'=> $density ));
  $conexion->commit();

} catch (Exception $e) {
  $conexion->rollBack();
  echo "Failed: " . $e->getMessage();
}
?>
```

Se aconseja el uso de parámetros, ya que reduce la posibilidad de error.
Aunque no vayan a causar problemas en las consultas, sigue siendo conveniente tratar los datos recibidos para eliminar los espacios en blanco iniciales y finales, tratar los caracteres especiales del html, funciones de limpieza de inyecciones y caracteres especiales, etc....

2º ASIGNACIÓN MEDIANTE BINDPARAM

En este caso en vez de usar una matriz para asignar los valores, lo que hacemos es utilizar los métodos bindParam para asignar los parámetros a las "marcas" previamente creadas. No existe ninguna ventaja en este técnica, nada más que es más similar a la que se usa en msyql para consultas preparadas. Mejor usar la anterior.

```
// Prepare
$consulta_prep = $conexion->prepare("INSERT INTO Vendedores (nombre, ciudad)
VALUES (:nombre, :ciudad)");
// Bind
$nombre = "Charles";
$ciudad = "Valladolid";
$consulta_prep->bindParam(':nombre', $nombre);
$consulta_prep->bindParam(':ciudad', $ciudad);
// Excecute
$consulta_prep->execute();
// Bind
$nombre = "Anne";
$ciudad = "Lugo";
$consulta_prep->bindParam(':nombre', $nombre);
$consulta_prep->bindParam(':ciudad', $ciudad);
// Execute
$consulta_prep->execute();
```

Mas ejemplos:

```
<?php
try {
  $conexion = new PDO('sqlite:test.db');//otra tipo de base de datos
  echo "Connected\n";
} catch (Exception $e) {
  die("Unable to connect: " . $e->getMessage());
}

try {
  $con_prep = $conexion->prepare("INSERT INTO countries (name, area,
population, density) VALUES (:name, :area, :population, :density)");
  $con_prep->bindParam(':name', $name);
  $con_prep->bindParam(':area', $area);
  $con_prep->bindParam(':population', $population);
  $con_prep->bindParam(':density', $density);

  $conexion->beginTransaction();
  $name = 'Nicaragua'; $area = 129494; $population = 602800; $density = 46.55;
  $con_prep->execute();
  $name = 'Panama'; $area = 78200; $population = 3652000; $density = 46.70;
  $con_prep->execute();
  $conexion->commit();

} catch (Exception $e) {
  $conexion->rollBack();
  echo "Failed: " . $e->getMessage();
}
?>
```

Una característica importante cuando se utilizan variables para pasar los valores es que **se pueden insertar objetos directamente en la base de datos.**

```
class Clientes
{
    public $nombre;
    public $ciudad;
    public function __construct($nombre, $ciudad){
        $this->nombre = $nombre;
        $this->ciudad = $ciudad;
    }
    // ....Código de la clase....
}
$cliente = new Clientes("Jennifer", "Málaga");
$consulta_prep = $conexion->prepare("INSERT INTO Clientes (nombre, ciudad)
VALUES (:nombre, :ciudad)");
if($consulta_prep->execute((array) $cliente)){
    echo "Se ha creado un nuevo registro!";
      }
```

Para poder hacer esto ten en cuenta que:

- el objeto debe tener el mismo número de atributos que elementos vaya a guardar en el INSERT
- que los atributos se deben llamar igual que los :campos, sin el :. Es decir $this->nombre se debe poner como :nombre y $this->ciudad: se debe poner en el prepare como :ciudad.
- Los atributos han de ser públicos.

12. Diferencia entre bindParam() y bindValue()

Existen dos métodos para enlazar valores: *bindParam()* y *bindValue()*:

- Con *bindParam()* la variable es enlazada como una referencia y **sólo será evaluada cuando se llame a** *execute()*:

```
// Prepare:
$consulta_prep = $conexion->prepare("INSERT INTO Vendedores (nombre) VALUES
(:nombre)");
$nombre = "Morgan";
// Bind
$consulta_prep->bindParam(':nombre', $nombre); // Se enlaza a la variable
$nombre
// Si ahora cambiamos el valor de $nombre:
$nombre = "John";
$consulta_prep->execute(); // Se insertará el cliente con el nombre John
```

- Con *bindValue()* se enlaza el valor de la variable y **permanece hasta execute()**:

```
// Prepare:
$consulta_prep = $conexion->prepare("INSERT INTO Vendedores (nombre) VALUES
(:nombre)");
$nombre = "Morgan";
// Bind
$consulta_prep->bindValue(':nombre', $nombre); // Se enlaza al valor Morgan
// Si ahora cambiamos el valor de $nombre:
$nombre = "John";
$consulta_prep->execute(); // Se insertará el cliente con el nombre Morgan
```

En la práctica *bindValue()* se suele usar cuando se tienen que insertar datos sólo una vez, y *bindParam()* cuando se tienen que pasar datos múltiples (desde un array por ejemplo).
Ambas funciones aceptan un **tercer parámetro**, que define el tipo de dato que se espera. Los **data types** más utilizados son: PDO::PARAM_BOOL (*booleano*), PDO::PARAM_NULL (*null*), PDO::PARAM_INT (*integer*) y PDO::PARAM_STR (*string*).

Pero también es cierto, que si usamos el array para enlazar los elementos todo esto es innecesario

13. Consultar datos con PDO con consultas preparadas

La **consulta de datos** se realiza mediante **PDOStatement::fetch**, que **obtiene la siguiente fila de un conjunto de resultados**. Antes de llamar a fetch (o durante) hay que especificar como se quieren devolver los datos:

- **PDO::FETCH_ASSOC**: devuelve un array indexado cuyos índices son el **nombre de las columnas**.
- **PDO::FETCH_NUM**: devuelve un array indexado cuyos índices son **números,** estos números correspondes a la posición de la columna.
- **PDO::FETCH_BOTH**: valor por defecto. Devuelve un array indexado cuyos índices son tanto el **nombre de las columnas** como **números**.
- **PDO::FETCH_BOUND**: asigna los valores de las columnas a las variables establecidas con el método PDOStatement::bindColumn.
- **PDO::FETCH_CLASS**: asigna los valores de las columnas a propiedades de una clase. Creará las propiedades si éstas no existen.
- **PDO::FETCH_INTO**: actualiza una instancia existente de una clase.
- **PDO::FETCH_OBJ**: devuelve un objeto anónimo con nombres de propiedades que corresponden a las columnas.
- **PDO::FETCH_LAZY**: combina **PDO::FETCH_BOTH** y **PDO::FETCH_OBJ**, creando los nombres de las propiedades del objeto tal como se accedieron.

- Los más utilizados son **FETCH_ASSOC**, **FETCH_OBJ, FETCH_BOUND** y **FETCH_CLASS.**

Vamos a poner un ejemplo de los dos primeros:

```
<?php
include 'base_datos.php';
$conexion=conectar();
// Ejecución de una consulta SELECT.
// FETCH_ASSOC
$consulta_prep = $conexion->prepare("SELECT * FROM VENDEDOR WHERE
provincia=:provincia ");
// Especificamos el fetch mode antes de llamar a fetch()
$consulta_prep->setFetchMode(PDO::FETCH_ASSOC);
//esto sirve para no poner el tipo de recoger los datos
// de poner el fetch pues lo he definido antes.

// Ejecutamos
$consulta_prep->execute(array(':provincia'=>'Alicante'));
// Mostramos los resultados
//IGUAL QUE SIN CONSULTAS PREPARADAS SALVO el EXECUTE
// TAMBIÉN PODRIAMOS PONER $fila = $consulta_prep->fetch(PDO::FETCH_ASSOC)
// pero como lo he definido antes con setFetchMode no es necesarios
while ($fila = $consulta_prep->fetch()){
    echo "Nombre: ".$fila['NOMVEND']." <br>";
    echo "Ciudad: ".$fila['CIUDAD']." <br>";
}
// FETCH_OBJ
$consulta_prep = $conexion->prepare("SELECT * FROM VENDEDOR WHERE
provincia=:provincia ");
// Ejecutamos
$consulta_prep->execute(array(':provincia'=>'Alicante'));
// Ahora vamos a indicar el fetch mode cuando llamamos a fetch:
while($fila = $consulta_prep->fetch(PDO::FETCH_OBJ)){
    echo "Nombre: " . $fila->NOMVEND . "<br>";
    echo "Ciudad: " . $fila->CIUDAD . "<br>";
}
```

Con **FETCH_BOUND** debemos emplear el método *bindColumn()*: NO LO USÉIS

```
include 'base_datos.php';
$conexion=conectar();
// Preparamos
$consulta_prep = $conexion->prepare("SELECT nomvend, ciudad FROM Vendedor");
// Ejecutamos
$consulta_prep->execute();
// bindColumn
$consulta_prep->bindColumn(1, $nombre);
$consulta_prep->bindColumn('ciudad', $ciudad);
while ($fila = $consulta_prep->fetch(PDO::FETCH_BOUND)) {
    $cliente = $nombre . ": " . $ciudad;
    echo $cliente . "<br>";
}
```

y el resto igual que con el método query, salvo el FETCH_CLASS, que veremos más adelante.

Finalmente, para la consulta de datos también se puede emplear directamente **PDOStatement::fetchAll()**, que devuelve un **array con todas las filas devuel-tas por la base de datos** con las que poder iterar. También acepta estilos de devolución:

```
<?php
include 'base_datos.php';
$conexion=conectar();
// fetchAll() con PDO::FETCH_ASSOC
$consulta_prep = $conexion->prepare("SELECT * FROM Vendedor");
$consulta_prep->execute();
$vendedores = $consulta_prep->fetchAll(PDO::FETCH_ASSOC);
foreach($vendedores as $vendedor){
    echo $vendedor['NOMVEND'] . "<br>";
}
// fetchAll() con PDO::FETCH_OBJ
$consulta_prep = $conexion->prepare("SELECT * FROM Vendedor");
$consulta_prep->execute();
$vendedores = $consulta_prep->fetchAll(PDO::FETCH_OBJ);
foreach ($vendedores as $vendedor){
    echo $vendedor->NOMVEND . "<br>";
}

//mas de FETCH ALL
    try {
        $consulta_prep = $conexion->prepare("SELECT * FROM PIEZA");
        $consulta_prep->execute();
        echo "<table border=1><tr>";
        $result = $consulta_prep->fetchAll(PDO::FETCH_ASSOC);
        $keys = array_keys($result[0]);//con esto cojo los nombre de las
                                       //columnas
        foreach ($keys as $key)
          echo "<th>".$key."</th>";
        echo "</tr>";
        foreach ($result as $row) {
          echo "<tr>";
          foreach ($keys as $key)
              echo "<td>".$row[$key]."</td>";
          echo "</tr>";
        }
        echo "</table>";
    } catch (Exception $e) {
      echo "Failed: " . $e->getMessage();
    }
```

14. Consultar datos con PDO con preparadas y objetos FETCH_CLASS

Aquí hacemos un apunte aparte y nos centramos en el estilo de devolver los datos **FETCH_CLASS,** el cual es algo más complejo: **devuelve los datos directamente a una clase**. Las propiedades del objeto se establecen ANTES de llamar al **constructor**. Si hay nombres de columnas que no tienen una propiedad creada para cada una, se crean como **public**. **Si los datos necesitan una transformación antes de que salgan de la base de datos**, se puede hacer automáticamente cada vez que se crea un objeto
Veamos un ejemplo completo:

Base de Datos a utilizar

```
CREATE DATABASE IF NOT EXISTS pruebas;
USE pruebas;
CREATE TABLE personas(
      id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
      nombre VARCHAR(255) NOT NULL,
      apellidos VARCHAR(255) NOT NULL,
      sexo CHAR NOT NULL,
      PRIMARY KEY(id)
);

INSERT INTO personas (nombre,apellidos, sexo) VALUES ('Luis','Cabrera', 'M');
INSERT INTO personas (nombre,apellidos, sexo) VALUES ('Rosa','Martin', 'F');
INSERT INTO personas (nombre,apellidos, sexo) VALUES ('Ramon','Lopez', 'M');
INSERT INTO personas (nombre,apellidos, sexo) VALUES ('Pedro','Perez', 'M');
```

Codigo

```
<?php
include 'base_datos.php';
class Personas
{
    public $nombre;
    public $apellidos;
    private $sexo;
    private $campoNoTabla;
//observa que no está el ID sin embargo en la base de datos SÍ
    public function __construct($otros = 'otro'){
        $this->nombre = strtoupper($this->nombre);
        $this->apellidos = substr($this->apellidos, 0, 5);
        $this->otros=$otros;
    }
    public function getsexo(){
      return $this->sexo;
    }
    // .Más Código de la clase
}
$conexion=conectar();
$consulta_prep = $conexion->prepare("SELECT * FROM PERSONAS");
$consulta_prep->setFetchMode(PDO::FETCH_CLASS, 'Personas');
$consulta_prep->execute();

while ($objeto = $consulta_prep->fetch()){
    echo $objeto->nombre . " -> ";
    echo $objeto->apellidos . "->". $objeto->getsexo() ."<br>";
      var_dump($objeto) ;
   }
```

Veamos que es lo que ocurre:

- **Campos nombre y apellidos:** Se enlazan correctamente con $nombre y $apellidos. Son públicos por que está definidos públicos.
  - **$this->nombre:** será en mayúscula pues al llamar al constructor en el momento que recibe los datos, ejecuta el código strtoupper.
  - **$this->apellidos:** da igual el tamaño que tenga en la tabla, como en el caso de $nombre, como llama al constructor en el momento de recibir los datos deja el apellido en un tamaño de 5.
- **Campo sexo:** Se enlaza correctamente con $this->sexo, pero, eso sí, es privado. Es privado por que así lo definí en la clase. Como se llaman igual se enlazan perfectamente.
- **Campo id:** está en la consulta, pero no está definido en la clase, sin embargo si me fijo, **SÍ existe** en el $objeto->id. Por tanto, todos los campos que devuelva la consulta y no estén definidos en la clase, se generan en el objeto, pero SIEMPRE como públicos.
- **Campo otros:** No existe en la tabla, entonces $this->otros es un atributo que creo en el constructor de la clase. No hay problema por añadir a la clase campos que no estén en el resultado de la consulta. Eso sí su valor será 'otro' pues es el valor por defecto que le pongo.
- **Campo campoNoTabla:** No existe en la tabla, pero sí está definido $campoNotabla, que es un atributo de la clase. No hay problema por tener en la clase campos que no estén en el resultado de la consulta. Evidentemente, al crear el objeto estos campos no lo rellena sino que los pondrá a null, o como en el caso anterior, al valor por defecto que le pongas en el constructor **.**

Si lo que quieres es **llamar al constructor ANTES de que se asignen los datos**, se hace lo siguiente:

```
$consulta_prep->setFetchMode(PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE,
'Personas';
```

Si en el ejemplo anterior añadimos PDO::FETCH_PROPS_LATE, el nombre y apellidos se mostrarán como aparecen en la base de datos, no se pasan a mayúscula ni se “recorta” el apellido.

También se pueden **pasar argumentos al constructor** cuando se quieren devolver datos en objetos con PDO:

```
$consulta_prep->setFetchMode(PDO::FETCH_CLASS, 'Personas',array('masdatos');
```

Con lo anterior $otros pasa a valer 'masdatos'.

O incluso **datos diferentes para cada objeto**:

```
$i = 0;
while ($fila = $consulta_prep->fetch(PDO::FETCH_CLASS, 'Personas', array($i)))
{
    // Código para hacer algo
    $i++;
}
```

En el caso anterior, otros toma en el primer objeto el valor de 0, en el segundo otros vale 1, en el tercero 2…

RESUMIENDO: El o **FETCH_CLASS** se usa y se usa mucho si lo que queremos es “mapear” una tabla y meterla directamente en una clase ya definida por nosotros. La principal diferencia entre **FETCH_CLASS y FETCH_OBJ** es que con **FETCH_CLASS** puedo usar la clase que yo desee con sus método y **FETCH_OBJ**, crea un objeto estándar sin métodos.

15. Diferencia entre query() y prepare()/execute()

En los ejemplos anteriores para las **sentencias en PDO**, no se ha introducido el método *query()*. Este método ejecuta la sentencia directamente y necesita que se escapen los datos adecuadamente para evitar ataques SQL Injection y otros problemas.
execute() ejecuta una sentencia preparada lo que permite enlazar parámetros y evitar tener que escapar los parámetros. *execute()* también tiene mejor rendimiento si se repite una sentencia múltiples veces, ya que se compila en el servidor de bases de datos sólo una vez.

16. Otras funciones de utilidad

Existen otras **funciones en PDO** que pueden ser de utilidad:

a) PDO::exec(). Ejecuta una sentencia SQL y devuelve el número de filas afectadas. Devuelve el número de filas **modificadas o borradas**, no devuelve resultados de una secuencia SELECT:

```
// Si lo siguiente devuelve 1, es que se ha eliminado correctamente:
echo $conexion->exec("DELETE FROM Vendedores WHERE nomvend='Luis'");
// No devuelve el número de filas con SELECT, devuelve 0
echo $conexion->exec("SELECT * FROM Vendedores");
```

b) PDO::lastInsertId(). Este método devuelve el id del último registro en esa conexión. Muy útil para campos autoincrementados.

```
$consulta_prep = $conexion->prepare("INSERT INTO Vendedores (numvend,
        nomvend) VALUES (:numvend, :nombre)");
$nombre = "Angelina"; $numero=66;
$consulta_prep->execute(array (':numvend'=>$numero, 'nombre'=>$nomvend));
echo $conexion->lastInsertId();
```

c) PDOStatement::fetchColumn(). Devuelve una única columna de la siguiente fila de un conjunto de resultados. La columna se indica con un integer, empezando desde cero. Si no se proporciona valor, obtiene la primera columna.

```
$consulta_prep = $conexion->prepare("SELECT * FROM Vendedores");
$consulta_prep->execute();
while ($fila = $consulta_prep->fetchColumn(1)){
    echo "Nombre Vendedor : $fila <br>";//como la segunda es el nomvend
}
```

d) PDOStatement::rowCount(). Devuelve el **número de filas afectadas por la última sentencia SQL.** Muy útil.

```
$consulta_prep = $conexion->prepare("SELECT * FROM Vendedores");
$consulta_prep->execute();
echo $consulta_prep->rowCount();
```

17. Transacciones en PDO

Recordamos: Una transacción en un Sistema de Gestión de Bases de Datos es un conjunto de órdenes que se ejecutan formando una unidad de trabajo, es decir, en forma indivisible o atómica.
Un SGBD se dice transaccional si es capaz de mantener la integridad de datos, haciendo que estas transacciones no puedan finalizar en un estado intermedio. Cuando por alguna causa el sistema debe cancelar la transacción, empieza a deshacer las órdenes ejecutadas hasta dejar la base de datos en su estado inicial (llamado punto de integridad), como si la orden de la transacción nunca se hubiese realizado. Una transacción debe contar con ACID (un acrónimo inglés) que quiere decir: Atomicidad, consistencia, aislamiento y durabilidad.
En un sistema ideal, las transacciones deberían garantizar todas las propiedades ACID; en la práctica, a veces alguna de estas propiedades se simplifica o debilita con vistas a obtener un mejor rendimiento.
Las transacciones deben cumplir cuatro propiedades ACID:

1. Atomicidad (**A**tomicity):es la propiedad que asegura que la operación se ha realizado o no, y por lo tanto ante un fallo del sistema no puede quedar a medias.
2. Consistencia (**C**onsistency): es la propiedad que asegura que sólo se empieza aquello que se puede acabar. Por lo tanto, se ejecutan aquellas operaciones que no van a romper la reglas y directrices de integridad de la base de datos.
3. Aislamiento (**I**solation): es la propiedad que asegura que una operación no puede afectar a otras. Esto asegura que la realización de dos transacciones sobre la misma información nunca generará ningún tipo de error.
4. Permanencia (**D**urability): es la propiedad que asegura que una vez realizada la operación, ésta persistirá y no se podrá deshacer aunque falle el sistema.

La forma algorítmica que suelen tener las transacciones es la siguiente:

```
iniciar transacción (lista de recursos a bloquear)
ejecución de las operaciones individuales.
if (todo_ok) {
  ''aplicar_cambios''
}
else {
  ''cancelar_cambios''
}
```

En cualquier momento, el programa podría decidir que es necesario hacer *fallar* la transacción, con lo que el sistema deberá revertir todos los cambios hechos por las operaciones ya hechas. En el lenguaje SQL se denomina *COMMIT* a *aplicar_cambios* y *ROLLBACK* a *cancelar_cambios*.
Para esto, el lenguaje de consulta de datos SQL (*Structured Query Language*), provee los mecanismos para especificar que un conjunto de acciones deben constituir una transacción.

- BEGIN TRAN: Especifica que va a empezar una transacción.
- COMMIT TRAN: Le indica al motor que puede considerar la transacción completada con éxito.
- ROLLBACK TRAN: Indica que se ha alcanzado un fallo y que debe restablecer la base al punto de integridad.

Bien, ten en cuenta que nosotros usamos mysql con PDO. En realidad la capacidad transaccional depende de la SGBD, en este caso mysql (PDO soportará transacciones si la base de datos lo soporta). ¿Es mysql es transaccional? Depende.
Muy gallega la respuesta.

Primero de todo aclarar el tipo de engine que se está utilizando en MySql. Normalmente hay dos engines clásicos que podemos utilizar en MySql: MyISAM e InnoDB. Inicialmente el engine por defecto era MyISAM, pero la tendencia es utilizar InnoDB por la posibilidad de utilizar transacciones (MyISAM no tiene), bloqueo de registros e integridad referencial.
En otra palabras, si la base de datos en mysql es InnoDB, soportaremos transacciones y si no NO.

Por fin empezamos.

Por defecto PDO trabaja en modo "**autocommit**", esto es, confirma de forma automática **cada sentencia que ejecuta el servidor.** Por tanto si lo que quiero es ejecutar varias operaciones dependientes entre sí (una transacción) debo pornerlo en modo autocommit = false.

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

Todas las sentencias SQL son atómicas por si solas. Es decir si no utilizamos transacciones después de un update o insert, este viene implícito con un commit.
Una vez ejecutado un commit o un rollback, se volverá al modo de confirmación automática.

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

Debido a que no todas las bases de datos soportan transacciones, PHP corre en el modo de *auto-commit* que ejecuta cada instrucción individual en forma implícita. Si se desea usar transacciones, y no se desea utilizar el modo de *auto-commit*, es necesario invocar el método *PDO::beginTransaction()* al inicio de la transacción. Si el manejador de la base de datos no permite el uso de transacciones se producirá una excepción (*PDOException*). Cuando se acabe de especificar la transacción se pueden utilizar los métodos *PDO::Commit* para aplicar dichas instrucciones, o bien, *PDO::rollBack* para abortar dicha transacción.

Si una transacción no fue terminada con la instrucción *commit* y el programa finaliza, la base de datos abortará la transacción automáticamente.

Ejemplo de funcionamiento con TRANSACCIONES

Lo que vamos hacer con el siguiente ejemplo es el típico ejemplo de transacción bancaria de donde sacamos dinero de la cuenta "12345" y la transferimos a la cuenta "12346".
Tenemos la tabla "cuentas" con los siguientes valores.

```
+----+------------+--------+
| id | num_cuenta | saldo |
+----+------------+--------+
|  1 |      12345 |  1200 |
|  2 |      12346 |     0 |
+----+------------+--------+
```

```
CREATE TABLE cuentas(
id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT,
num_cuenta VARCHAR(255) NOT NULL,
saldo bigint NOT NULL DEFAULT 0,
PRIMARY KEY(id)
);
```

Si ejecutamos el siguiente código PHP:

```
<?php
include "base_datos.php";

$cuenta_origen  = 12345;
$cuenta_destino = 12346;
$importe_mover  = 1000;

$conexion=conectar();

$sql = "UPDATE cuentas SET saldo=saldo-:importe_mover
        WHERE num_cuenta=:cuenta_origen";
$sentencia=$conexion->prepare($sql);
$resultado=$sentencia->execute([":cuenta_origen"=>$cuenta_origen,
            ":importe_mover"=>$importe_mover]);
if (!$resultado) {
             die('Fallo en la consulta');
}
$sql = "UPDATE cuentas SET saldo=saldo+:importe_mover
        WHERE num_cuenta=:cuenta_destino";

$sentencia=$conexion->prepare($sql);
$resultado=$sentencia->execute([":cuenta_destino"=>$cuenta_destino,
            ":importe_mover"=>$importe_mover]);

if (!$resultado) {
             die('Fallo en la consulta');
}

$conexion=null;//cierro la conexionecho "Fin...";
?>
```

Lo que hemos hecho es quitar 1000€ de la cuenta "12345" y transferirla a la cuenta "12346". Si miramos el resultado de la tabla cuentas tenemos:

```
+----+------------+--------+
| id | num_cuenta | saldo |
+----+------------+--------+
|  1 |      12345 |   200 |
|  2 |      12346 |  1000 |
+----+------------+--------+
```

Que pasaría si entre un UPDATE y el otro UPDATE nuestro servidor tiene un problema y se reinicia sólo (un corte de luz o un reinicio del admin). Volvamos a dejar la tabla de cuentas como estaba y provocamos un "corte de luz" entre un UPDATE y otro UPDATE.

```
<?php
include "base_datos.php";

$cuenta_origen  = 12345;
$cuenta_destino = 12346;
$importe_mover  = 1000;

$conexion=conectar();

$sql = "UPDATE cuentas SET saldo=saldo-:importe_mover
        WHERE num_cuenta=:cuenta_origen";
$sentencia=$conexion->prepare($sql);
$resultado=$sentencia->execute([":cuenta_origen"=>$cuenta_origen,
            ":importe_mover"=>$importe_mover]);
if (!$resultado) {
             die('Fallo en la consulta');
}
echo "Opss! se fue la luz...";
exit();
$sql = "UPDATE cuentas SET saldo=saldo+:importe_mover
        WHERE num_cuenta=:cuenta_destino";

$sentencia=$conexion->prepare($sql);
$resultado=$sentencia->execute([":cuenta_destino"=>$cuenta_destino,
            ":importe_mover"=>$importe_mover]);

if (!$resultado) {
             die('Fallo en la consulta');
}

$conexion=null;
echo "Fin...";
?>
```

Si ejecutamos ahora el script y miramos el contenido de la tabla vemos como hemos quitado 1000€ de una cuenta, pero la otra sigue estando a 0€.
Hemos hecho desaparecer 1000€. Explica tú ahora al banco que has cometido un error en la programación y que has hecho desaparecer 1000€ de 1000 usuarios diferentes.

```
+----+------------+--------+
| id | num_cuenta | saldo |
+----+------------+--------+
|  1 |      12345 |   200 |
|  2 |      12346 |     0 |
+----+------------+--------+
```

Para hacer correcta la serie de instrucciones debemos enmarcar los dos updates dentro de una transacción. Si utilizamos transacciones aseguraremos las 4 propiedades básicas: atomicidad, consistencia, aislamiento y durabilidad.

- atomicidad: o se realiza todo o nada.
- consistencia: si se viola alguna regla de integridad no se realizará la operación.
- aislamiento: las operaciones de una transacción no influyen a la operaciones de otra transacción aunque estén trabajando con los mismos datos.
- durabilidad: una vez hecho el commit los datos persistirán en la base de datos.

Para arrancar en MySql una transacción deshabilitamos el autocommit por defecto, en PDO, empezamos una transacción con beginTransaction(), para finalizarla utilizaremos un "COMMIT" si queremos guardar

los cambios o un "ROLLBACK" si queremos cancelar toda la transacción. Recuerda que la idea es "o se ejecuta todo o no se ejecuta nada".

```
<?php
include "base_datos.php";

$cuenta_origen  = 12345;
$cuenta_destino = 12346;
$importe_mover  = 1000;

$conexion=conectar();
$conexion->beginTransaction();
$sql = "UPDATE cuentas SET saldo=saldo-:importe_mover
        WHERE num_cuenta=:cuenta_origen";
$sentencia=$conexion->prepare($sql);
$resultado=$sentencia->execute([":cuenta_origen"=>$cuenta_origen,
           ":importe_mover"=>$importe_mover]);

if (!$resultado) {
      $conexion->rollback();
      die('Fallo en la consulta');
}

$sql = "UPDATE cuentas SET saldo=saldo+:importe_mover
        WHERE num_cuenta=:cuenta_destino";
$sentencia=$conexion->prepare($sql);
$resultado=$sentencia->execute([":cuenta_destino"=>$cuenta_destino,
      ":importe_mover"=>$importe_mover]);

if (!$resultado) {
      $conexion->rollback();
      die('Fallo en la consulta');
}
$conexion->commit();
$conexion=null;
echo "Fin...";
?>
```

Si miramos el resultado veremos como al igual que en el primer ejemplo movemos 1000€ de una cuenta a otra:

```
+----+------------+-------+
| id | num_cuenta | saldo |
+----+------------+-------+
|  1 |      12345 |   200 |
|  2 |      12346 |  1000 |
+----+------------+-------+
```

Provocamos ahora un "corte de luz", pero utilizando transacciones, tenemos:

```
<?php
include "base_datos.php";

$cuenta_origen  = 12345;
$cuenta_destino = 12346;
$importe_mover  = 1000;

$conexion=conectar();
$conexion->beginTransaction();
$sql = "UPDATE cuentas SET saldo=saldo-:importe_mover
        WHERE num_cuenta=:cuenta_origen";
$sentencia=$conexion->prepare($sql);
$resultado=$sentencia->execute([":cuenta_origen"=>$cuenta_origen,
            ":importe_mover"=>$importe_mover]);

if (!$resultado) {
      $conexion->rollback();
      die('Fallo en la consulta');
}

echo "Opss! se fue la luz...";
exit();

$sql = "UPDATE cuentas SET saldo=saldo+:importe_mover
        WHERE num_cuenta=:cuenta_destino";
$sentencia=$conexion->prepare($sql);
$resultado=$sentencia->execute([":cuenta_destino"=>$cuenta_destino,
      ":importe_mover"=>$importe_mover]);

if (!$resultado) {
      $conexion->rollback();
      die('Fallo en la consulta');
}
$conexion->commit();
$conexion=null;
echo "Fin...";
?>
```

Si miramos ahora el contenido de la tabla vemos como no se ha producido ningún cambio. A pesar de que “se ha ido la luz” el primer UPDATE no se ha producido porque no se ha finalizado la transacción. La transacción finaliza cuando se hace el COMMIT. ¿Fácil no?. Pues a mover millones de una cuenta a otra.

id	num_cuenta	saldo
1	12345	1200
2	12346	0

Ejemplo sencillos pero con try/catch de transacciones que es como se hace

```
try {
  $conexion->beginTransaction();
  $conexion->exec("INSERT INTO countries (name, area, population, density)
                           values ('Belice',22966,334000,14.54)");
  $conexion->exec("INSERT INTO countries (name, area, population, density)
                           values ('Costa Rica',51100,4726000,92.49)");
  $conexion->exec("INSERT INTO countries (name, area, population, density)
                           values ('El Salvador',21041,6108000,290.29)");
  $conexion->exec("INSERT INTO countries (name, area, population, density)
                           values ('Guatemala',108894,15284000,140.36)");
  $conexion->exec("INSERT INTO countries (name, area, population, density)
                           values ('Honduras',112492,8447000,75.09)");
  $conexion->commit();

} catch (Exception $e) {
  $conexion->rollBack();
  echo "Failed: " . $e->getMessage();
}
```

NOTA DEL EJEMPLO:

public int **PDO::exec** (string $instruccion_sql)

PDO::exec() ejecuta una instrucción simple sql y devuelve la cantidad de filas afectadas por la consulta. (Muy util si no funciona el co), por ejemplo

```
<?php
$con=new PDO('odbc:sample','db2inst1','ibmdb2');//otra forma de conexion
//a otra base de datos

/*Delete all rows from the FRUIT table*/
$count= $con->exec("DELETE FROM fruit WHERE colour='red'");

/*Return number of rows that were deleted*/
echo("Filas borradas:$count.<br>");
?>
```

PDO::exec() Ojo, esta función no devuelve resultados para SELECT, por tanto no USAR con instrucciones SELECT. Para instrucciones SELECT utilizaremos PDO::query(). Para consultas preparadas y múltiples consultas, como ya sabemos utilizaremos PDO::prepare() combinado con PDOStatement::execute().

Mismo ejemplo con query

```
try {
    $conexion->beginTransaction();
    $conexion->query("INSERT INTO Clientes (nombre, ciudad) VALUES ('Leila
Birdsall', 'Madrid')");
    $conexion->query("INSERT INTO Clientes (nombre, ciudad) VALUES ('Brice
Osterberg', 'Teruel')");
    $conexion->query("INSERT INTO Clientes (nombre, ciudad) VALUES ('Latrisha
Wagar', 'Valencia')");
    $conexion->query("INSERT INTO Clientes (nombre, ciudad) VALUES ('Hui
Riojas', 'Madrid')");
    $conexion->query("INSERT INTO Clientes (nombre, ciudad) VALUES ('Frank
Scarpa', 'Barcelona')");
    $conexion->commit();
    echo "Se han introducido los nuevos clientes";
} catch (Exception $e){
    echo "Ha habido algún error";
    $conexion->rollback();
}
```

18.- Errores y manejo de excepciones.

En sus primeros pasos en la programación en lenguaje PHP, ***Carlos se ha encontrado frecuentemente con pequeños errores en el código****. En la mayoría de las ocasiones los errores estaban causados por fallos de programación. Pero en las recientes pruebas llevadas a cabo con bases de datos, se ha dado cuenta de que* ***en algunas ocasiones se pueden producir fallos ajenos al programa.***
Por ejemplo, puede obtener un error porque ***no esté disponible el servidor de bases de datos*** *con el que se ha de conectar la aplicación, o porque* ***se acaba de borrar de la base de datos un registro al que estaba accediendo****. En éstos, y muchos otros, casos es necesario que la aplicación que desarrollen se comporte de forma sólida y coherente. Es necesario estudiar a fondo las* ***posibilidades que ofrece PHP para la gestión de errores.***

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 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,

- constante guión hace referencia a avisos que pueden indicar un error al ejecutar
- constante la ejecución: engloba errores fatales que provocan que se interrumpa forzosamente.

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 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.
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 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).

19.- 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**.

Tras haber lanzado la excepción, necesitamos un mecanismo para capturar esta excepción y poder mostrar la información que queramos del error y actuar en consecuencia al error. Para ello existe una estructura de control llamada try/catch.
De esta forma dividiremos el manejo de excepciones en dos partes:

1. Dentro del bloque try se insertará el código que puede provocar una excepción. Ya que este bloque es el encargado de parar la ejecución del script y pasar el control al bloque 'catch' cuando se produzca una excepción.
2. Dentro de 'catch' introduciremos el código que controlará la excepción. Mostrando el error y aplicando la funcionalidad necesaria para actuar ante el error.

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.
PHP cuenta con dos tipos de excepciones predefinidas,**Exception** y **ErrorException**, y un conjunto de excepciones definidas en la **SPL** (standard PHP library).
Podemos lanzar excepciones manualmente utilizando la palabra clave **throw**:

```
throw new Exception('Mensaje');
```

El constructor de la clase **Exception** adopta una serie de parámetros opcionales, un mensaje y un código de error.
Para manejar las excepciones que lanzamos utilizamos la estructura de control **try/catch.**Todo el código que pueda lanzar una excepción debe incluirse dentro del bloque try ya que este bloque es el encargado de parar la ejecución del script y pasar el control al bloque catch cuando se produzca una excepción.

```
try
{
   // Codigo que puede lanzar excepciones
}
catch ( Exception $excepcion )
{
    // Codigo para controlar la excepcion
}
```

Puede haber más de un bloque catch asociado al mismo bloque try siempre que cada bloque catch espere capturar un tipo de excepción distinta. El objeto que se pasa al bloque catch es el objeto que se ha lanzado con la instrucción throw.

Es importante tener en cuenta que dentro de un bloque 'try', cuando una excepción es lanzada, el código siguiente a dicha excepción no será ejecutado. Pero si que se ejecutará el código tras el bloque try/catch. Por lo que si queremos detener totalmente el programa deberemos usar la función exit() tras mostrar el error, en un bloque 'catch'.

Vamos a ver un ejemplo para entender mejor el mecanismo de control de las excepciones:

```
<?php
try{
   throw new Exception('Se ha producido un error muy grave.');
}
catch(Exception $excepcion){
    echo $excepcion->getMessage();
}
//esto se ejecuta SIEMPRE
echo 'El bloque catch no finaliza la ejecución total del script.';
?>
```

En este ejemplo lo que estamos haciendo es lanzar directamente una excepción de la clase Exception informando que se ha producido un error muy grave. Una vez lanzada la excepción, el bloque catch la cap-

tura y pasa a procesarla mostrando al usuario el mensaje que hemos utilizado en el constructor del objeto excepción. Para ello utilizamos el método **getMessage()**. La clase Exception cuenta con un conjunto de métodos que muestran información sobre la excepción que ha ocurrido:

- **getCode():**Devuelve el código tal cual se haya pasado al constructor. Si no pasamos ninguno por defecto devuelve el código de error cero.
- **getMessage():**Devuelve el mensaje tal cual se haya pasado al constructor. En nuestro caso devolvería la cadena 'Se ha producido un error muy grave.'.
- **getFile():**Devuelve la ruta completa al archivo de código que haya lanzado la excepción.
- **getLine()**: Devuelve el número de la línea del archivo de código en el que se ha producido la excepción.
- **getTrace():**Devuelve un array con la traza que indica dónde se ha producido la excepción.
- **getTraceAsString():**Devuelve la misma información que el anterior método pero con formato de cadena.
- También podemos utilizar el método mágico **__toString()** para customizar la información que mostramos si imprimimos directamente el objeto excepción.

Es importante tener en cuenta que dentro de un bloque 'try', cuando una excepción es lanzada, el código siguiente a dicha excepción no será ejecutado. Pero si que se ejecutará el código tras el bloque try/catch. Por lo que si queremos detener totalmente el programa deberemos usar la función exit()/die() tras mostrar el error, en un bloque 'catch'.

Cabe destacar que realmente no deberíamos mostrar está información al usuario en un entorno de producción debido que muestra información sensible de como está desarrollado nuestra aplicación. Una solución efectiva es mostrar siempre una plantilla de error básica que informe un poco del error ocurrido y darle la opción al usuario de volver al inicio o a la página anterior.

Vamos a ver que pasaría si solo se lanza una excepción pero no se captura:

```
class Prueba{
  protected $id;
  function __construct($id) {
    if($this->validId($id)) $this->id= $id;
    else throw new Exception('Identificiador no es un entero');
  }
  private function validId($id){
    if(!is_int($id)) return false;
    return true;
  }
}
$p = new Prueba('prueba');
```

```
Fatal error: Uncaught exception 'Exception' with message 'Identificiador no es un entero' in /prueba/excepciones.php:8
Stack trace: #0 /prueba/excepciones.php(17): Prueba->__construct('prueba')
 #1 {main} thrown in /prueba/excepciones.php on line 8
```

Se acaba produciendo un error fatal. Lo que ratifica lo comentado anteriormente. PHP intentará encontrar el primer bloque 'catch' coincidente con la excepción lanzada. Si lanzamos una excepción estamos obligados a capturarla.

20 EXCEPCIONES CON PDO

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:

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

Por ejemplo, el siguiente código:

```
$conexion = new PDO("mysql:host=localhost; dbname=basedatos", "root",
"abc123.");
$conexion->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
try {
      $sql = "SELECT * FROM TABLA_NO_EXISTE";
      $result = $conexion->query($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
'conexion.stox' doesn't exist
```

Puedes observar que en el ejemplo anterior hemos usado **PDOException** en vez de la clase **Exception**, en realidad la clase **PDOException** es una clase que hereda de **Exception** por tanto tiene todas la características de la anterior, pero amplia la información de la misma para base de datos.

Más ejemplos

Ejemplo #2 Crear una instancia de PDO y establecer el modo de error desde el constructor

```
<?php
$dsn= 'mysql:dbname=prueba;host=127.0.0.1';
$usuario= 'usuario';
$contraseña= 'pass';
/*El empleo de try/catch en el constructor sigue siendo válido aunque se
establezca ERRMODE a WARNING,ya que PDO::__construct siempre lanzará una
PDOException si la conexión falla.*/
try {
   $conexion = new PDO($dsn, $usuario, $contraseña,
            array(PDO::ATTR_ERRMODE => PDO::ERRMODE_WARNING));
} catch (PDOException $e) {
   echo 'Error de conexión: ' .$e->getMessage();
 die();
}

/* Esto hará que PDO lance un error de nivel E_WARNING en lugar de una
excepción  (cuando la tabla no exista)*/
$conexion->query("SELECT columna_incorrecta FROM tabla_incorrecta");
?>
```

En este caso no ponemos la instrucción SELECT entre try catch pues lo hemos puesto en modo warning, por tanto genera un warning y no una excepción.

Ejemplo #3 Mismo ejemplo pero todo con excepciones. Crear una instancia de PDO y establecer el modo de error desde el constructor

```
<?php
$dsn= 'mysql:dbname=prueba;host=127.0.0.1';
$usuario= 'usuario';
$contraseña= 'pass';
/*El empleo de try/catch en el constructor sigue siendo válido aunque se
establezca ERRMODE a WARNING,ya que PDO::__construct siempre lanzará una
PDOException si la conexión falla.*/
try {
   $conexion = new PDO($dsn, $usuario, $contraseña,
            array(PDO::ATTR_ERRMODE => PDO::ERRMODE_WARNING));
} catch (PDOException $e) {
   echo 'Error de conexión: ' .$e->getMessage();
 die();
}
//COMENZAMOS OTRO TRY CATCH distinto
try {
  $result=$conexion->query("SELECT columna_incorrecta FROM tabla_incorrecta");
    }
catch (PDOException $p) {
      echo "Error ".$p->getMessage()."<br />";
   }
?>
```

21. EXCEPCIONES CON MYSQLI

Mysqli (de mysql ni hablamos pues es anterior a php 5 y con el actual php 7 se considera "decrepada", "maldita" y a extinguir) no disponía de un sistema propio de excepciones. Esto quiere decir que cuando se produce un error, NO generaba un excepción.
Pero claro, hablo en pasado. De hecho SÍ LAS GENERA, aunque hasta ahora no los hubiésemos utilizados. Debemos asegurarnos que :

`MYSQLI_REPORT_ERROR` excepciones en los errores de SQL, pero deberá habilitar `MYSQLI_REPORT_ERROR` si aún no lo está ...

```
      mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT)
```

`mysqli_query()` debería ahora lanzar excepciones en caso de error.

```
    try {
         resultado = $conexion->query( $sql);
    } catch (mysqli_sql_exception $e) {
        throw new MySQLiQueryException($sql, $e->getMessage(), $e-
>getCode());
    }
```