http://php.net/manual/es/intro.pdo.php

Conexiones y su administración

Las conexiones se establecen creando instancias de la clase base PDO. No importa el controlador que se utilice; siempre se usará el nombre de la clase PDO. El constructor acepta parámetros para especificar el origen de la base de datos (conocido como DSN) y, opcionalmente, el nombre de usuario y la contraseña (si la hubiera).

Ejemplo #1 Conectarse a MySQL

```
<?php
$mbd = new PDO('mysql:host=localhost;dbname=prueba', $usuario, $contra
seña);
?>
```

Si hubieran errores de conexión, se lanzará un objeto *PDOException*. Se puede capturar la excepción si fuera necesario manejar la condición del error, o se podría optar por dejarla en manos de un manejador de excepciones global de aplicación configurado mediante set_exception_handler().

Ejemplo #2 Manejo de errores de conexión

```
<?php
try {
    $mbd = new PDO('mysql:host=localhost;dbname=prueba', $usuario, $co
ntraseña);
    foreach($mbd->query('SELECT * from FOO') as $fila) {
        print_r($fila);
    }
    $mbd = null;
} catch (PDOException $e) {
    print "¡Error!: " . $e->getMessage() . "<br/>";
    die();
}
?>
```

Advertencia

Si la aplicación no captura la excepción lanzada por el constructor de PDO, la acción predeterminada que toma el motor zend es la de finalizar el script y mostrar información de seguimiento. Esta información probablemente revelará todos los detalles de la conexión a la base de datos, incluyendo el nombre de usuario y la contraseña. Es su responsabilidad capturar esta excepción, ya sea explícitamente (con una sentencia *catch*) o implícitamente por medio de set_exception_handler().

Una vez realizada con éxito una conexión a la base de datos, será devuelta una instancia de la clase PDO al script. La conexión permanecerá activa durante el tiempo de vida del objeto PDO. Para cerrar la conexión, es necesario destruir el objeto asegurándose de que todas las referencias a él existentes sean eliminadas (esto se puede hacer asignando **NULL** a la variable que contiene el objeto). Si no se realiza explícitamente, PHP cerrará automáticamente la conexión cuando el script finalice.

Ejemplo #3 Cerrar una conexión

```
<?php
$mbd = new PDO('mysql:host=localhost;dbname=prueba', $usuario, $contra
seña);
// Utilizar la conexión aquí


// Ya se ha terminado; se cierra
$mbd = null;
?>
```

Muchas aplicaciones web se beneficiarán del uso de conexiones persistentes a servidores de bases de datos. Las conexiones persistentes no son cerradas al final del script, sino que son almacenadas en caché y reutilizadas cuando otro script solicite una conexión que use las mismas credenciales. La caché de conexiones persistentes permite evitar la carga adicional de establecer una nueva conexión cada vez que un script necesite comunicarse con la base de datos, dando como resultado una aplicación web más rápida.

Ejemplo #4 Conexiones persistentes

```
<?php
$mbd = new PDO('mysql:host=localhost;dbname=prueba', $usuario, $contra
```

```
seña, array(
    PDO::ATTR_PERSISTENT => true
));
?>
```

Nota:
Para utilizar conexiones persistentes, se deberá establecer **PDO::ATTR_PERSISTENT** en las opciones del array del controlador pasado al constructor de PDO. Si este atributo se establece con PDO::setAttribute() después de la instanciación del objeto, el controlador no utilizará conexiones persistentes.

Nota:
Si se usa el controlador PDO y las bibliotecas ODBC admiten el aprovisionamiento de conexiones ODBC (unixODBC y Windows lo hacen; podrían haber más), se recomienda no utilizar las conexiones persistentes de PDO, y, en su lugar, dejar el almacenamiento en caché de conexiones a la capa del aprovisionamiento de conexiones de ODBC. La provisión de conexiones de ODBC se comparte con otros módulos en el proceso; si se le indica a PDO que almacene en caché la conexión, entonces dicha conexión nunca será devuelta a la provisión de conexiones de ODBC, dando como resultado la creación de conexiones adicionales para servir a los demás módulos.

Transacciones y autoconsigna ("auto-commit")

Una vez realizada una conexión a través de PDO, es necesario comprender cómo PDO gestiona las transacciones antes de comenzar a realizar consultas. Si no se han manejado anteriormente transacciones, estas ofrecen cuatro características principales: Atomicidad, Consistencia, Aislamiento y Durabilidad (ACID por sus siglas en inglés). En términos sencillos, se garantiza que cualquier trabajo llevado a cabo en una transacción, incluso si se hace por etapas, sea aplicado a la base de datos de forma segura, y sin interferencia de otras conexiones, cuando sea consignado. El trabajo transaccional puede también ser deshecho automáticamente bajo petición (siempre y cuando no se haya consignado ya), lo que hace más sencillo el manejo de errores en los scripts.

Las transacciones son implementadas típicamente para hacer que el lote de cambios se apliquen a la vez; esto tiene el buen efecto secundario de mejorar drásticamente la eficiencia de las actualizaciones. En otras palabras, las transacciones pueden hacer los scripts más rápidos y potencialmente más robustos (aún así es necesario utilizarlas correctamente para obtener tal beneficio).

Desafortunadamente, no todas las bases de datos admiten transacciones, por lo que PDO necesita ejecutarse en lo que es conocido como el modo "auto-commit" cuando se abra por primera vez la conexión. El modo auto-commit significa que toda consulta que se ejecute tiene su propia transacción implícita, si la base de datos la soporta, o ninguna transacción si la base de datos no las soporta. Si fuera necesario el uso de transacciones, se debe usar el métodoPDO::beginTransaction() para iniciar una. Si el controlador subyacente no admite transacciones, se lanzará una PDOException (independientemente de la configuración del manejo de errores: esto es siempre una condición de error serio). Una vez dentro de una transacción, se puede utilizar PDO::commit() o PDO::rollBack() para finalizarla, dependiendo del éxito del código que se ejecute durante la transacción.

Advertencia
PDO sólo comprueba la funcionalidad de transacciones a nivel del controlador. Si una cierta condición durante la ejecución implica que las transacciones no estén disponibles, PDO::beginTransaction() seguirá devolviendo `TRUE` sin ningún error si el servidor de bases de datos acepta la solicitud de iniciar una transacción.

Un ejemplo podría ser el intento de utilizar transacciones en tablas MyISAM en una base de datos MySQL.

Cuando el script finaliza o cuando una conexión está a punto de ser cerrada, si existe una transacción pendiente, PDO la revertirá automáticamente. Esto es una medida de seguridad que ayuda a evitar inconsistencia en los casos donde el script finaliza inesperadamente (si se no consignó la transacción, se asume que algo salió mal, con lo cual se realiza la reversión para la seguridad de los datos).

Advertencia
La reversión automática sólo ocurre si se inicia una transacción a través dePDO::beginTransaction(). Si se ejecuta manualmente una consulta que inicie

una transacción, PDO no tiene forma de saber algo sobre la misma y, por tanto, no puede revertirla si algo saliera mal.

Ejemplo #1 Ejecución de un lote en una transacción

En el siguiente ejemplo, se asume que se ha creado un conjunto de entradas para un nuevo empleado, al cual se le ha asignado el número de ID 23. Además de introducir los datos básicos de una persona, también es necesario registrar su sueldo. Es bastante simple realizar dos actualizaciones independientes, pero realizándolas entre las llamadas a PDO::beginTransaction()y PDO::commit(), se garantiza que nadie más será capaz de ver los cambios hasta que se hayan completado. Si algo sale mal, el bloque catch revierte los cambios realizados desde que se creó la transacción, y luego imprime un mensaje de error.

```
<?php
try {
  $mbd = new PDO('odbc:SAMPLE', 'db2inst1', 'ibmdb2',
      array(PDO::ATTR_PERSISTENT => true));
  echo "Conectado\n";
} catch (Exception $e) {
  die("No se pudo conectar: " . $e->getMessage());
}

try {
  $mbd->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

  $mbd->beginTransaction();
  $mbd-
>exec("insert into staff (id, first, last) values (23, 'Joe', 'Bloggs'
)");
  $mbd->exec("insert into salarychange (id, amount, changedate)
      values (23, 50000, NOW())");
  $mbd->commit();

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

```
}
?>
```

No existe límite al realizar actualizaciones en una transacción. También es posible ejecutar consultas complejas para extraer datos, con al posibilidad de utilizar esa información para construir más actualizaciones y consultas; mientras que la transacción esté activa, se garantiza que nadie más pueda realizar cambios mientras se esté en mitad del trabajo. Para más información sobre transacciones, consulte la documentación proporcionada por su servidor de base de datos.

Sentencias preparadas y procedimientos almacenados

Muchas de las bases de datos más maduras admiten el concepto de sentencias preparadas. Estas pueden definirse como un tipo de plantillas compiladas para SQL que las aplicaciones quieren ejecutar, pudiendo ser personalizadas utilizando parámetros variables. Las sentencias preparadas ofrecen dos grandes beneficios:

- La consulta sólo necesita ser analizada (o preparada) una vez, pero puede ser ejecutada muchas veces con los mismos o diferentes parámetros. Cuando la consulta se prepara, la base de datos analizará, compilará y optimizará su plan para ejecutarla. Para consultas complejas, este proceso puede tomar suficiente tiempo como para ralentizar notablemente una aplicación si fuera necesario repetir la misma consulta muchas veces con los mismos parámetros. Mediante el empleo de una sentencia preparada, la aplicación evita repetir el ciclo de análisis/compilación/optimización. Esto significa que las sentencias preparadas utilizan menos recursos y se ejecutan más rápidamente.
- Los parámetros para las sentencias preparadas no necesitan estar entrecomillados; el controlador automáticamente se encarga de esto. Si una aplicación usa exclusivamente sentencias preparadas, el desarrollador puede estar seguro de que no hay cabida para inyecciones de SQL (sin embargo, si otras partes de la consulta se construyen con datos de entrada sin escapar, aún es posible que ocurran ataques de inyecciones de SQL).

Las sentencias preparadas son tan útiles que son la única característica que PDO emulará para los controladores que no las soporten. Esto asegura que

una aplicación sea capaz de emplear el mismo paradigma de acceso a datos independientemente de las capacidades de la base de datos.

Ejemplo #1 Inserciones repetidas utilizando sentencias preparadas

Este ejemplo realiza dos consultas INSERT sustituyendo *name* y *value* por los marcadores correspondientes.

```
<?php
$sentencia = $mbd-
>prepare("INSERT INTO REGISTRY (name, value) VALUES (:name, :value)");
$sentencia->bindParam(':name', $nombre);
$sentencia->bindParam(':value', $valor);

// insertar una fila
$nombre = 'uno';
$valor = 1;
$sentencia->execute();

// insertar otra fila con diferentes valores
$nombre = 'dos';
$valor = 2;
$sentencia->execute();
?>
```

Ejemplo #2 Inserciones repetidas utilizando sentencias preparadas

Este ejemplo realiza dos consultas INSERT sustituyendo *name* y *value* por los marcadores posicionales '?'.

```
<?php
$sentencia = $mbd-
>prepare("INSERT INTO REGISTRY (name, value) VALUES (?, ?)");
$sentencia->bindParam(1, $nombre);
$sentencia->bindParam(2, $valor);

// insertar una fila
$nombre = 'uno';
$valor = 1;
$sentencia->execute();
```

```php
// insertar otra fila con diferentes valores
$nombre = 'dos';
$valor = 2;
$sentencia->execute();
?>
```

Ejemplo #3 Obtener datos empleando sentencias preparadas

Este ejemplo obtiene datos basándose en un valor de clave proporcionado por un formulario. La entrada del usuario es entrecomillada automáticamente, con lo cual no hay riesgo de un ataque por inyección de SQL.

```php
<?php
$sentencia = $mbd->prepare("SELECT * FROM REGISTRY where name = ?");
if ($sentencia->execute(array($_GET['name']))) {
  while ($fila = $sentencia->fetch()) {
    print_r($fila);
  }
}
?>
```

Si el controlador de la base de datos lo admite, una aplicación podría también vincular parámetros para salida y para entrada. Los parámetros de salida se emplean típicamente para recuperar valores de procedimientos almacenados. Los parámetros de salida son ligeramente más complejos de usar que los de entrada, de manera que el desarrollador debe conocer la magnitud de un parámetro dado cuando se vincula. Si el valor resulta ser más grande que el tamaño indicado, se emitirá un error.

Ejemplo #4 Llamar a un procedimiento almacenado con un parámetro de salida

```php
<?php
$sentencia = $mbd->prepare("CALL sp_returns_string(?)");
$sentencia->bindParam(1, $valor_devuleto, PDO::PARAM_STR, 4000);

// llamar al procedimiento almacenado
$sentencia->execute();
```

```
print "El procedimiento devolvió $valor_devuleto\n";
?>
```

Un desarrollador podría también especificar parámetros que contienen valores tanto de entrada como de salida; la sintaxis es similar a la de los parámetros de salida. En el siguiente ejemplo, la cadena 'hola' es pasada al procedimiento almacenado, y cuando éste finaliza, 'hola' es reemplazada con el valor de retorno del procedimiento.

Ejemplo #5 Llamar a un procedimiento almacenado con un parámetro de entrada/salida

```
<?php
$sentencia = $mbd->prepare("CALL sp_takes_string_returns_string(?)");
$valor = 'hola';
$sentencia-
>bindParam(1, $valor, PDO::PARAM_STR|PDO::PARAM_INPUT_OUTPUT, 4000);

// llamar al procedimiento almacenado
$sentencia->execute();

print "El procedimiento devolvió $valor\n";
?>
```

Ejemplo #6 Uso inválido de un marcador de posición

```
<?php
$sentencia = $mbd-
>prepare("SELECT * FROM REGISTRY where name LIKE '%?%'");
$sentencia->execute(array($_GET['name']));

// los marcadores de posición deben emplearse en el lugar del valor co
mpleto
$sentencia = $mbd-
>prepare("SELECT * FROM REGISTRY where name LIKE ?");
$sentencia->execute(array("%$_GET[name]%"));
?>
```

Errores y su manejo

PDO ofrece tres estrategias diferentes de manejar errores para adaptarse a cualquier estilo de desarrollo de aplicaciones.

- **PDO::ERRMODE_SILENT**

  Este es el modo predeterminado. PDO simplemente establecerá él mismo el código de error para su inspección utilizando los métodos PDO::errorCode() y PDO::errorInfo() tanto en objetos de sentencias como de bases de datos. Si el error resultó de una llamada a un objeto de sentencia, se deberá invocar al método PDOStatement::errorCode() oPDOStatement::errorInfo() sobre dicho objeto. Si el error resultó de una llamada a un objeto de bases de datos, se deberá invocar, en su lugar, a los métodos del objeto de bases de datos.

- **PDO::ERRMODE_WARNING**

  Además de establecer el código de error, PDO emitirá un mensaje E_WARNING tradicional. Esta configuración es útil durante la depuración o realización de pruebas para visualizar los problemas que han ocurrido sin interrumpir el flujo de la aplicación.

- **PDO::ERRMODE_EXCEPTION**

  Además de establecer el código de error, PDO lanzará una PDOException y establecerá sus propiedades para reflejar el error y la información del mismo. Esta configuración también es útil durante la depuración, ya que, de hecho, señalará el lugar del error del script, apuntando a áreas pontencialmente problemáticas del código (recuerde: las transacciones son automáticamente revertidas si la excepción causa la finalización del script).

  El modo Exception también es útil porque se puede estructurar el manejo de errores con más claridad que con el estilo tradicional de advertencias de PHP, y con menos código/anidación que con la ejecución en modo silencioso, comprobando así explícitamente el valor devuelto de cada llamada a la base de datos.

  Véase la referencia de Excepciones para más información sobre excepciones en PHP.

PDO utiliza las cadenas de código de error SQLSTATE del estándar SQL-92; cada controlador de PDO es responsable de la correspondencia de su códigos nativos con los códigos SQLSTATE apropiados. El método PDO::errorCode() devuelve un único código SQLSTATE. Si fuera necesaria más información específica sobre el error, PDO también ofrece el método PDO::errorInfo(), que devuelve un array que contiene el código SQLSTATE, el código de error específico del controlador, y la cadena de error específica.

Ejemplo #1 Crear una instancia de PDO y establecer el modo de error

```php
<?php
$dsn = 'mysql:dbname=prueba;host=127.0.0.1';
$usuario = 'usuario';
$contraseña = 'contraseña';

try {
    $mbd = new PDO($dsn, $usuario, $contraseña);
    $mbd->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch (PDOException $e) {
    echo 'Falló la conexión: ' . $e->getMessage();
}

?>
```

Nota:
PDO::__construct() siempre lanzará una PDOException si la conexión falla, independientemente de que **PDO::ATTR_ERRMODE** esté esteblecido. Las excepciones no capturadas son fatales.

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

```php
<?php
$dsn = 'mysql:dbname=prueba;host=127.0.0.1';
$usuario = 'usuario';
$contraseña = 'contraseña';

/*
    El empleo de try/catch en el constructor sigue siendo válido aunqu
```

```
e se establezca ERRMODE a WARNING,
    ya que PDO::__construct siempre lanzará una PDOException si la con
exión falla.
*/
try {
    $mbd = new PDO($dsn, $usuario, $contraseña, array(PDO::ATTR_ERRMOD
E => PDO::ERRMODE_WARNING));
} catch (PDOException $e) {
    echo 'Error de conexión: ' . $e->getMessage();
    exit;
}

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

El resultado del ejemplo sería:

```
Warning: PDO::query(): SQLSTATE[42S02]: Base table or view not found:
1146 Table 'prueba.tabla_incorrecta' doesn't exist in
/tmp/prueba_pdo.php on line 18
```

Objetos grandes (LOB)

En algún momento, una aplicación podría necesitar almacenar datos "grandes" en la base de datos. Grande típicamente significa "alrededor de 4kb o más", aunque algunas bases de datos pueden manejar fácilmente hasta 32kb antes de que los datos se consideren "grandes". Los objetos grandes, o LOB (acrónimo en inglés de 'Large OBjects'), pueden ser de texto o binarios. PDO permite trabajar con este tipo de datos grandes utilizando el código del tipo **PDO::PARAM_LOB**en llamadas a PDOStatement::bindParam() o PDOStatement::bindColumn(). **PDO::PARAM_LOB**indica a PDO que haga corresponder los datos como un flujo, pudiendo manipularlos así utilizando la API de flujos de PHP.

Ejemplo #1 Mostrar una imagen desde una base de datos

Este ejemplo vincula un LOB a una variable llamada $lob, y luego lo envía al navegador usandofpassthru(). Ya que el LOB está representado como un flujo, se pueden emplear funciones tales
como fgets(), fread() y stream_get_contents() para manejarlo.

```
<?php
$bd = new PDO('odbc:SAMPLE', 'db2inst1', 'ibmdb2');
$sentencia = $bd-
>prepare("select contenttype, imagedata from images where id=?");
$sentencia->execute(array($_GET['id']));
$sentencia->bindColumn(1, $tipo, PDO::PARAM_STR, 256);
$sentencia->bindColumn(2, $lob, PDO::PARAM_LOB);
$sentencia->fetch(PDO::FETCH_BOUND);

header("Content-Type: $tipo");
fpassthru($lob);
?>
```

Ejemplo #2 Insertar una imagen en una base de datos

Este ejemplo abre un fichero y pasa el manejador del fichero a PDO para insertarlo como un LOB. PDO hará todo lo posible para enviar el contenido del fichero a la base de datos de la manera más eficiente.

```
<?php
$bd = new PDO('odbc:SAMPLE', 'db2inst1', 'ibmdb2');
$sentencia = $bd-
>prepare("insert into images (id, contenttype, imagedata) values (?, ?
, ?)");
$id = get_new_id(); // alguna función para asignar un nuevo ID

// Se asume que se está ejecutando como parte de un formulario de subi
da de ficheros
// Se puede encontrar más información en la documentación de PHP

$fp = fopen($_FILES['file']['tmp_name'], 'rb');

$sentencia->bindParam(1, $id);
$sentencia->bindParam(2, $_FILES['file']['type']);
```

```
$sentencia->bindParam(3, $fp, PDO::PARAM_LOB);

$bd->beginTransaction();
$sentencia->execute();
$bd->commit();
?>
```

Ejemplo #3 Insertar una imagen en una base de datos: Oracle

Oracle requiere una sintaxis ligeramente diferente para insertar un LOB desde un fichero. También es esencial que se realice la inserción bajo una transacción, si no, el LOB recién insertado será consignado con una longitud cero como parte de la consigna implícita que ocurre cuando la consulta se ejecuta:

```
<?php
$bd = new PDO('oci:', 'scott', 'tiger');
$sentencia = $bd-
>prepare("insert into images (id, contenttype, imagedata) " .
"VALUES (?, ?, EMPTY_BLOB()) RETURNING imagedata INTO ?");
$id = get_new_id(); // alguna función para asignar un nuevo ID

// Se asume que se está ejecutando como parte de un formulario de subi
da de ficheros
// Se puede encontrar más información en la documentación de PHP

$fp = fopen($ FILES['file']['tmp name'], 'rb');

$sentencia->bindParam(1, $id);
$sentencia->bindParam(2, $ FILES['file']['type']);
$sentencia->bindParam(3, $fp, PDO::PARAM_LOB);

$sentencia->beginTransaction();
$sentencia->execute();
$bd->commit();
?>
```

La clase PDO

(PHP 5 >= 5.1.0, PECL pdo >= 0.1.0)

Introducción

Representa una conexión entre PHP y un servidor de bases de datos.

Sinopsis de la Clase

```
PDO {
public __construct ( string $dsn [, string $username [, string $password
[, array$options ]]] )

public bool beginTransaction ( void )

public bool commit ( void )

public mixed errorCode ( void )

public array errorInfo ( void )

public int exec ( string $statement )

public mixed getAttribute ( int $attribute )

public static array getAvailableDrivers ( void )

public bool inTransaction ( void )

public string lastInsertId ([ string $name = NULL ] )

public PDOStatement prepare ( string $statement [, array $driver_option
s = array()] )

public PDOStatement query ( string $statement )

public string quote ( string $string [, int $parameter_type =
PDO::PARAM_STR ] )

public bool rollBack ( void )

public bool setAttribute ( int $attribute , mixed $value )

}
```

Tabla de contenidos

- PDO::commit — Consigna una transacción
- PDO::__construct — Crea una instancia de PDO que representa una conexión a una base de datos
- PDO::errorCode — Obtiene un SQLSTATE asociado con la última operación en el manejador de la base de datos
- PDO::errorInfo — Obtiene información extendida del error asociado con la última operación del manejador de la base de datos
- PDO::exec — Ejecuta una sentencia SQL y devuelve el número de filas afectadas
- PDO::getAttribute — Devuelve un atributo de la conexión a la base de datos
- PDO::getAvailableDrivers — Devuelve un array con los controladores de PDO disponibles
- PDO::inTransaction — Comprueba si una transacción está activa
- PDO::lastInsertId — Devuelve el ID de la última fila o secuencia insertada
- PDO::prepare — Prepara una sentencia para su ejecución y devuelve un objeto sentencia
- PDO::query — Ejecuta una sentencia SQL, devolviendo un conjunto de resultados como un objeto PDOStatement
- PDO::quote — Entrecomilla una cadena de caracteres para usarla en una consulta
- PDO::rollBack — Revierte una transacción
- PDO::setAttribute — Establece un atributo

La clase PDOStatement

(PHP 5 >= 5.1.0, PECL pdo >= 1.0.0)

Introducción

Representa una sentencia preparada y, después de la ejecución de la instrucción, un conjuntos de resultados asociado.

Sinopsis de la Clase

```
PDOStatement implements Traversable {

/* Propiedades */

readonly string $queryString;

/* Métodos */
```

```
public bool bindColumn ( mixed $column , mixed &$param [, int $type [, int
$maxlen [, mixed $driverdata ]]] )

public bool bindParam ( mixed $parameter , mixed &$variable [, int $data_
type = PDO::PARAM_STR [, int $length [, mixed $driver_options ]]] )

public bool bindValue ( mixed $parameter , mixed $value [, int $data_type
= PDO::PARAM_STR ] )

public bool closeCursor ( void )

public int columnCount ( void )

public void debugDumpParams ( void )

public string errorCode ( void )

public array errorInfo ( void )

public bool execute ([ array $input_parameters ] )

public mixed fetch ([ int $fetch_style [, int $cursor_orientation =
PDO::FETCH_ORI_NEXT [, int $cursor_offset = 0 ]]] )

public array fetchAll ([ int $fetch_style [, mixed $fetch_argument [, arr
ay$ctor_args = array() ]]] )

public mixed fetchColumn ([ int $column_number = 0 ] )

public mixed fetchObject ([ string $class_name =
"stdClass" [, array $ctor_args ]] )

public mixed getAttribute ( int $attribute )

public array getColumnMeta ( int $column )

public bool nextRowset ( void )

public int rowCount ( void )

public bool setAttribute ( int $attribute , mixed $value )

public bool setFetchMode ( int $mode )

}
```

Propiedades

queryString

La cadena de consulta utilizada.

Tabla de contenidos

```php
<?php

$host = "yourHost";
$user = "yourUser";
$pass = "yourPass";
$db = "yourDB";

$cursor = "cr_123456";

try
{
  $dbh =
new PDO("pgsql:host=$host;port=5432;dbname=$db;user=$user;password=$pas
s");
   echo "Connected<p>";
}
catch (Exception $e)
{
   echo "Unable to connect: " . $e->getMessage() ."<p>";
}

$dbh->beginTransaction();

$query = "SELECT yourFunction(0::smallint,'2013-08-01 00:00','2013-09-01
00:00',1::smallint,'$cursor')";

$dbh->query($query);

$query = "FETCH ALL IN \"$cursor\"";

echo "begin data<p>";

foreach ($dbh->query($query) as $row)
{
    echo "$row[0] $row[1] $row[2] <br>";
}

echo "end data";

?>
```

La clase PDOException

(PHP 5 >= 5.1.0)

Introducción

Representa un error generado por PDO. No se debería lanzar una PDOException desde el código. Véase Excepciones para más información acerca de las excepciones en PHP.

Sinopsis de la Clase

```
PDOException extends RuntimeException {

/* Propiedades */

public array $errorInfo ;

protected string $code ;

/* Propiedades heredadas */

protected string $message ;

protected int $code ;

protected string $file ;

protected int $line ;

/* Métodos heredados */

final public string Exception::getMessage ( void )

final public Exception Exception::getPrevious ( void )

final public mixed Exception::getCode ( void )

final public string Exception::getFile ( void )

final public int Exception::getLine ( void )

final public array Exception::getTrace ( void )

final public string Exception::getTraceAsString ( void )

public string Exception::__toString ( void )

final private void Exception::__clone ( void )

}
```

Propiedades

errorInfo

Corresponde a PDO::errorInfo() o PDOStatement::errorInfo()

code

Código de error de *SQLSTATE*. Utilice Exception::getCode() para acceder a él.

Controladores de PDO

Tabla de contenidos

Los siguientes controladores implementan actualmente la interfaz PDO:

Nombre del controlador	Bases de datos admitidas
PDO_CUBRID	Cubrid
PDO_DBLIB	FreeTDS / Microsoft SQL Server / Sybase
PDO_FIREBIRD	Firebird
PDO_IBM	IBM DB2
PDO_INFORMIX	IBM Informix Dynamic Server
PDO_MYSQL	MySQL 3.x/4.x/5.x
PDO_OCI	Oracle Call Interface
PDO_ODBC	ODBC v3 (IBM DB2, unixODBC and win32 ODBC)
PDO_PGSQL	PostgreSQL
PDO_SQLITE	SQLite 3 and SQLite 2
PDO_SQLSRV	Microsoft SQL Server / SQL Azure

Nombre del controlador	Bases de datos admitidas
PDO_4D	4D

Si el desarrollo de tus aplicaciones lo haces con PHP usando como base de datos MySql y no utilizas ningún tipo de framework, seguramente las consultas a la base de datos las realizarás con el **API de mysql**. Un ejemplo del uso de este API es el siguiente:

```
1
2   $con = mysql_connect('localhost', 'user', pass');
3
4   mysql_select_db('nombreBaseDatos');
5   $sql = 'SELECT * FROM tabla';
6
7   $res = mysql_query($sql);
8
9   while( $row = mysql_fetch_array($res) )
10    echo $row['titulo'];
11
12  mysql_free_result($res);
    mysql_close($con);
```

Si este es tu caso, deberías saber que estás usando una API obsoleta y desaconsejada por el equipo de PHP. En su lugar deberías usar el **API de mysqli** o mejor aún **PDO** (PHP Data Objects). Utilizando PDO podemos solventar muchas dificultades que surgen al utilizar la API de mysql. Un par de ventajas que se obtienen con PDO es que el proceso de escapado de los parámetros es sumamente sencillo y sin la necesidad de estar atentos a utilizar en todos los casos funciones para este cometido como *mysql_real_escape_string()*. Otra ventaja es que PDO es una API flexible y nos permite trabajar con cualquier tipo de base de datos y no estar restringidos a utilizar MySql.
Veamos como podemos utilizar PDO en una aplicación para recuperar datos de una base de datos MySql y ver lo sencillo que es usar esta API. Lo primero que tenemos que hacer es realizar la conexión a la base de datos. La conexión la realizaremos con el **constructor de la clase** de la siguiente forma:

```
1   $con = new PDO('mysql:host=localhost;dbname=nombreBaseDatos', 'user', 'pass');
```

Con la llamada anterior ya tenemos creada la conexión a la base de datos. Antes de continuar voy a explicar como tratar los posibles errores con PDO. Por defecto PDO viene configurado para no mostrar ningún error. Es decir que para saber si se ha producido un error, tendríamos que estar comprobando los métodos **errorCode()** y **errorInfo()**. Para facilitarnos la tarea vamos a habilitar las excepciones. De esta forma cada vez que ocurra un error saltará una excepción que capturaremos y podremos tratar correctamente para mostrarle un mensaje al usuario. Para realizar esta tarea utilizaremos la función **setAttribute()** de la siguiente forma:

```
1   $con = new PDO('mysql:host=localhost;dbname=nombreBaseDatos', 'user', 'pass');
2   $conn->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

Los posibles valores que se le podría asignar a *ATTR_ERRMODE* son:

- **PDO::ERRMODE_SILENT** es el valor por defecto y como he mencionado antes no lanza ningún tipo de error ni excepción, es tarea del programador comprobar si ha ocurrido algún error después de cada operación con la base de datos.

- **PDO::ERRMODE_WARNING** genera un error E_WARNING de PHP si ocurre algún error. Este error es el mismo que se muestra usando la API de mysql mostrando por pantalla una descripción del error que ha ocurrido.
- **PDO::ERRMODE_EXCEPTION** es el que acabamos de explicar que genera y lanza una excepción si ocurre algún tipo de error.

Como acabamos de hacer que se lancen excepciones cuando se produzca algún error, el paso que tenemos que dar a continuación es capturarlas por si se producen, para ello realizamos lo siguiente:

```
1
2  try
3  {
4    $con = new PDO('mysql:host=localhost;dbname=nombreBaseDatos', 'user', 'pass');
5    $con->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
6  }
7  catch(PDOException $e)
8  {
9    echo 'Error conectando con la base de datos: ' . $e->getMessage();
   }
```

Ahora que sabemos como conectarnos a la base de datos, vamos a crear una sentencia para poder recuperar datos. Para ejecutar sentencias podemos utilizar la llamada a **query()** o bien la llamada a **prepare()**. Aunque tenemos disponibles las dos llamadas es mucho más seguro utilizar la llamada a **prepare()** ya que esta se encarga de escapar por nosotros los parámetros y nos asegura que no sufriremos problemas de SQL Injection. La función **query()** se suele utilizar cuando la sentencia que vamos a ejecutar no contiene parámetros que ha enviado el usuario. Veamos un ejemplo utilizando la función **query()**:

```
1
2   try
3   {
4     $con = new PDO('mysql:host=localhost;dbname=personal', 'user', 'pass');
5     $con->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
6
7     $datos = $con->query('SELECT nombre FROM personal');
8     foreach($datos as $row)
9       echo $row[0] . '<br/>';
10  }
11  catch(PDOException $e)
12  {
13    echo 'Error conectando con la base de datos: ' . $e->getMessage();
    }
```

Si a pesar de las advertencias aun quieres ejecutar sentencias con **query()** pasándole parámetros de usuarios, la forma correcta de hacerlo sería escapando esos parámetros con la función **quote()**como se muestra a continuación:

```
1   $ape = 'Hernandez';
2
3   try
4   {
5     $con = new PDO('mysql:host=localhost;dbname=personal', 'user', 'pass');
6     $con->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

      $datos = $con->query('SELECT nombre FROM personal WHERE apellidos like ' . $con-
```

```
7   >quote($ape));
8     foreach($datos as $row)
9       echo $row[0] . '<br/>';
10  }
11  catch(PDOException $e)
12  {
13    echo 'Error conectando con la base de datos: ' . $e->getMessage();
14  }
15
```

La forma de utilizar la función **prepare()** que es la más recomendada es la siguiente:

?

```
1
2   try
3   {
4     $con = new PDO('mysql:host=localhost;dbname=personal', 'user', 'pass');
5     $con->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
6
7     $stmt = $con->prepare('SELECT nombre FROM personal');
8     $stmt->execute();
9
10    while( $datos = $stmt->fetch() )
11      echo $datos[0] . '<br/>';
12  }
13  catch(PDOException $e)
14  {
15    echo 'Error: ' . $e->getMessage();
16  }
17
```

Como se ve, es realmente simple ejecutar consultas. Simplemente tenemos que indicarle a la función **prepare()** la sentencia sql que queremos ejecutar. Esta función nos devolverá un **PDOStatement** sobre el cual ejecutaremos la función **fetch()** para poder mostrar su valor.

Si necesitamos pasarle valores a la sentencia sql, utilizaríamos los parámetros. Los parámetros los indicamos en la misma sentencia sql y los podemos escribir de dos formas distintas. Mediante el signo**?** o mediante un nombre de variable precedido por el simbolo : **:nombreParam**. La segunda forma nos permite una identificación más fácil de los parámetros, pero cualquiera de las dos formas es correcta. Veamos un ejemplo:

?

```
1   $ape = 'Hernandez';
2
3   try
4   {
5     $con = new PDO('mysql:host=localhost;dbname=personal', 'user', 'pass');
6     $con->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
7
8     $stmt = $con->prepare('SELECT nombre FROM personal WHERE apellidos like
    :apellidos');
9     $stmt->execute(array(':apellidos' => $ape ));
10
11    while( $datos = $stmt->fetch() )
```

```
11        echo $datos[0] . '<br />';
12    }
13    catch(PDOException $e)
14    {
15      echo 'Error: ' . $e->getMessage();
16    }
17
```

Como se ve hemos llamado al parámetro *:apellidos* y posteriormente en la llamada a la función**execute()** indicamos con un array asociativo el nombre del parámetro y su valor. Otra forma de indicar los parámetros es utilizando la función **bindParam**.

```
1
2     $ape = 'Hernandez';
3
4     try
5     {
6       $con = new PDO('mysql:host=localhost;dbname=personal', 'user', 'pass');
7       $con->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
8
9       $stmt = $con->prepare('SELECT nombre FROM personal WHERE apellidos like
    :apellidos');
10      $stmt->bindParam(':apellidos', $ape, PDO::PARAM_INT);
11      $stmt->execute();
12
13      while( $datos = $stmt->fetch() )
14        echo $datos[0] . '<br />';
15    }
16    catch(PDOException $e)
17    {
18      echo 'Error: ' . $e->getMessage();
19    }
20
```

A la función **bindParam()** le pasamos el nombre del parámetro, su valor y finalmente el tipo que es. Los tipos de parámetros que le podemos pasar los podemos ver en las **constantes predefinidas de PDO** y son:

- PDO::PARAM_BOOL
- PDO::PARAM_NULL
- PDO::PARAM_INT
- PDO::PARAM_STR
- PDO::PARAM_LOB

Al igual que las sentencias select, podemos utilizar las funciones **query()** y **prepare()** para ejecutar inserts, updates y deletes. La forma de hacerlo es igual que lo que hemos estado viendo hasta ahora:

Ejemplo de insert:

```
1     $nom = 'Jose';
2     $ape = 'Hernandez';
3
4     try
5     {
6       $con = new PDO('mysql:host=localhost;dbname=personal', 'user', 'pass');
7       $con->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
8
```

```
8       $stmt = $con->prepare('INSERT INTO personal (nombre, apellidos) VALUES (:nombre
9     :apellidos)');
10      $rows = $stmt->execute( array( ':nombre'  => $nom,
11                                     ':apellidos' => $ape));
12
13      if( $rows == 1 )
14        echo 'Inserción correcta';
15    }
16    catch(PDOException $e)
17    {
18      echo 'Error: ' . $e->getMessage();
19    }
```

Ejemplo de update:

```
?
1
2     $nom = 'Jose';
3     $ape = 'Hernandez';
4
5     try
6     {
7       $con = new PDO('mysql:host=localhost;dbname=personal', 'user', 'pass');
8       $con->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
9
10      $stmt = $con->prepare('UPDATE personal SET apellidos = :apellidos WHERE nombre
11    :nombre');
12      $rows = $stmt->execute( array( ':nombre'  => $nom,
13                                     ':apellidos' => $ape));
14      if( $rows > 0 )
15        echo 'Actualización correcta';
16    }
17    catch(PDOException $e)
18    {
19      echo 'Error: ' . $e->getMessage();
20    }
```

Ejemplo de delete:

```
?
1     $ape = 'Hernandez';
2
3     try
4     {
5       $con = new PDO('mysql:host=localhost;dbname=personal', 'user', 'pass');
6       $con->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
7
8       $stmt = $con->prepare('DELETE FROM personal WHERE apellidos = :apellidos');
9       $rows = $stmt->execute( array( ':apellidos' => $ape));
10
11      if( $rows > 0 )
12        echo 'Borrado correcto';
13    }
14    catch(PDOException $e)
15    {
```

```
14      echo 'Error: ' . $e->getMessage();
15    }
16
17
```

Para acabar con esta entrada sobre PDO vamos a ver otra de las funcionalidades que nos aporta y que puede ser muy útil. PDO nos permite realizar consultas y mapear los resultados en objetos de nuestro modelo. Para ello primero tenemos que crearnos una clase con nuestro modelo de datos.

?

```
1
2     class Usuario
3     {
4       private $nombre;
5       private $apellidos;
6
7       public function nombreApellidos()
8       {
9         return $this->nombre . ' ' . $this->apellidos;
10      }
      }
```

Hay que tener en cuenta que para que funcione correctamente, el nombre de los atributos en nuestra clase tienen que ser iguales que los que tienen las columnas en nuestra tabla de la base de datos. Con esto claro vamos a realizar la consulta.

?

```
1
2     try
3     {
4       $con = new PDO('mysql:host=localhost;dbname=personal', 'user', 'pass');
5       $con->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
6
7      $stmt= $con->prepare('SELECT nombre, apellidos FROM personal');
8       $stmt->execute();
9
10      $stmt->setFetchMode(PDO::FETCH_CLASS, 'Usuario');
11
12      while($usuario = $stmt->fetch())
13        echo $usuario->nombreApellidos() . '<br>';
14
15    }
16    catch(PDOException $e)
17    {
18      echo 'Error: ' . $e->getMessage();
      }
```

La novedad que podemos ver en este script es la llamada al método **setFetchMode()** pasándole como primer argumento la constante PDO::FETCH_CLASS que le indica que haga un mapeado en la clase que le indicamos como segundo argumento, en este caso la clase Usuario que hemos creado anteriormente. Después al recorrer los elementos con *fetch* los resultados en vez de en un vector los obtendremos en el objeto indicado.

Después de todo esto solo me queda decir que si eres de los que todavía sigues usando la antigua API de mysql este es un buen momento para empezar a cambiar y a usar una nueva API más moderna y con mejores prestaciones.