

## USAR PDO EN PHP

En esta guía vamos a ver los conceptos básicos de PDO (PHP Data Object). PDO es una extensión para acceder a bases de datos. PDO tiene sus propias funciones y métodos que facilitan el acceso a diferentes sistemas de bases de datos con un controlador mediante el cual se conecta. Independientemente del sistema utilizado se emplearán siempre los mismos métodos, lo que hace que cambiar de uno a otro sistema de base de datos resulte más sencillo.

### QUE ES PDO

Los Objetos de Datos de PHP, o PDO, son una capa de abstracción de bases de datos específicas para aplicaciones PHP. PDO ofrece una API homogénea para las aplicaciones PHP, independientemente del tipo de servidor de bases de datos con el que se vaya a conectar la aplicación. En teoría, si se utiliza la API PDO, se podría cambiar el servidor de bases de datos en uso, por ejemplo de PostgreSQL a MySQL, y sólo se necesitarían algunos cambios menores en el código PHP.

A pesar de que PDO tiene sus ventajas, tales como una API limpia, sencilla y portable, su mayor inconveniente es que no permite utilizar todas las funcionalidades avanzadas en la versión del servidor MySQL. Por ejemplo, PDO no permite hacer uso de las Declaraciones Múltiples de MySQL.

PDO está desarrollado utilizando el framework de extensiones de PHP. Su código fuente se ubica en el directorio ext/pdo.

### PORQUE USAR PDO?

Uno de los principales motivos para usar PDO es la seguridad, porque, como se verá a continuación, se utilizan parámetros por nombre y “sentencias preparadas” que reducen la inyección de SQL, además PDO es orientado a objetos e incorpora ventajas de optimización, transacciones, pooling de conexiones y varias cosas más, lo cual le da un mejor rendimiento.

### CONEXIÓN MYSQL CON PDO

El primer argumento de la clase PDO es el DSN, Data Source Name, en el cual se han de especificar el tipo de base de datos (mysql), el host (localhost) y el nombre de la base de datos (se puede especificar también el puerto). Se pueden realizar conexiones a otras bases de datos como Oracle, sqlite, etc. Cada sistema de bases de datos tienen distintos métodos para conectarse, aunque la mayoría se conectan de forma parecida a como se conecta a MySQL.

Para ver los controladores (drivers) disponibles en tu servidor, puedes emplear el método `getAvailableDrivers()`:

```
$dbPDO = new PDO("mysql:host=localhost;dbname=database_name",$dbuser,$dbpass);
```

La conexión se debe incluir siempre dentro de un bloque Try/Catch para el manejo de excepciones

```
define("DB_DRIVER", "mysql"); // database driver, mysql
define("DB_HOST", "localhost"); // db host
define("DB_NAME", "dbpdo"); // database name
define("DB_USER", "root"); // database username
define("DB_PASS", ""); // database password

try
{
    $dbCon = new PDO(DB_DRIVER . ':host=' . DB_HOST . ';dbname=' . DB_NAME, DB_USER,
DB_PASS);
    $dbCon->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch (PDOException $ex) {
    echo $ex->getMessage();
    die();
}
```

Para cerrar una conexión:

```
$dbCon = null
```

Se puede (y se debe) especificar el modo de error estableciendo el atributo **error mode**:

```
$dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_SILENT);
$dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
$dbh->setAttribute(PDO::ATTR_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.

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
try {
    $dsn = "mysql:host=localhost;dbname=$dbname";
    $options = array(
        PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
    );
    $dbh = new PDO($dsn, $user, $password, $options);
} catch (PDOException $e) {
    echo $e->getMessage();
}

// Con el método PDO::setAttribute
try {
    $dsn = "mysql:host=localhost;dbname=$dbname";
    $dbh = new PDO($dsn, $user, $password);
    $dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch (PDOException $e) {
    echo $e->getMessage();
}
```

Existen más opciones aparte del 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.

## CREAR, ACTUALIZAR, ELIMINAR

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.

- Utilizando interrogantes para los valores

```
// Prepare
$stmt = $dbh->prepare("INSERT INTO Clientes (nombre, ciudad) VALUES (?, ?)");
// Bind
$nombre = "Peter";
$ciudad = "Madrid";
$stmt->bindParam(1, $nombre);
$stmt->bindParam(2, $ciudad);
// Execute
$stmt->execute();
// Bind
$nombre = "Martha";
$ciudad = "Cáceres";
$stmt->bindParam(1, $nombre);
$stmt->bindParam(2, $ciudad);
// Execute
$stmt->execute();
```

- Utilizando variables para los valores

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

También existe un método, que es pasando los valores mediante un array (siempre array, aunque sólo haya un valor) al método execute():

```
// Prepare:
$stmt = $dbh->prepare("INSERT INTO Clientes (nombre, ciudad) VALUES (:nombre, :ciudad)");
$nombre = "Luis";
$ciudad = "Barcelona";
// Bind y execute:
if($stmt->execute(array(':nombre'=>$nombre, ':ciudad'=>$ciudad))) {
```

```
    echo "Se ha creado el nuevo registro!";  
}
```

Es el método *execute()* el que realmente envía los datos a la base de datos. Si no se llama a *execute* no se obtendrán los resultados sino un error.

Una característica importante cuando se utilizan variables para pasar los valores es que se pueden insertar objetos directamente en la base de datos, suponiendo que las propiedades coinciden con los nombres de las variables:

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

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

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

```
// Prepare:
$stmt = $dbh->prepare("INSERT INTO Clientes (nombre) VALUES (:nombre)");
$nombre = "Morgan";
// Bind
$stmt->bindValue(':nombre', $nombre); // Se enlaza al valor Morgan
// Si ahora cambiamos el valor de $nombre:
$nombre = "John";
$stmt->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*).

## Ejemplo CRUD

### Create

```
if(isset($_POST['save']))
{
    $uname = isset($_POST['uname']) ? $_POST['uname'] : "";
    $umail = isset($_POST['umail']) ? $_POST['umail'] : "";

    $stmt = $DBcon->prepare("INSERT INTO tbl_test(username,email) VALUES(:uname,
:umail)");

    $stmt->bindParam(':uname', $uname);
    $stmt->bindParam(':umail', $umail);
    $stmt->execute();
}
```

### Update

```
if(isset($_POST['update']))
{
    $uname = isset($_POST['uname']) ? $_POST['uname'] : "";
    $umail = isset($_POST['umail']) ? $_POST['umail'] : "";
    $id = isset($_GET['edit_id']) ? $_POST['edit_id'] : "";

    $stmt = $DBcon->prepare("UPDATE tbl_test SET username=:uname, email=:umail WHERE
id=:id");
    $stmt->bindParam(':uname', $uname);
```

```

$stmt->bindParam(':uemail', $umail);
$stmt->bindParam(':id', $id);
$stmt->execute();
header("Location: index.php");
}

```

## Delete

```

if(isset($_GET['delete_id']))
{
    $id = isset($_GET['delete_id']) ? $_GET['delete_id'] : -1;
    $stmt = $DBcon->prepare("DELETE FROM tbl_test WHERE id=:id");
    $stmt->execute(array(':id' => $id));
    header("Location: index.php");
}

```

## CONSULTAR DATOS CON PDO

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 keys son el nombre de las columnas.
- PDO::FETCH\_NUM: devuelve un array indexado cuyos keys son números.
- PDO::FETCH\_BOTH: valor por defecto. Devuelve un array indexado cuyos keys 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\_INTOTO: 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. Ejemplo de los dos primeros:

```
// FETCH_ASSOC
$stmt = $dbh->prepare("SELECT * FROM Clientes");
// Especificamos el fetch mode antes de llamar a fetch()
$stmt->setFetchMode(PDO::FETCH_ASSOC);
// Ejecutamos
$stmt->execute();
// Mostramos los resultados
while ($row = $stmt->fetch()) {
    echo "Nombre: {$row["nombre"]} <br>";
    echo "Ciudad: {$row["ciudad"]} <br><br>";
}
// FETCH_OBJ
$stmt = $dbh->prepare("SELECT * FROM Clientes");
// Ejecutamos
$stmt->execute();
// Ahora vamos a indicar el fetch mode cuando llamamos a fetch:
while ($row = $stmt->fetch(PDO::FETCH_OBJ)) {
    echo "Nombre: " . $row->nombre . "<br>";
    echo "Ciudad: " . $row->ciudad . "<br>";
}
```

El estilo de devolver los datos **FETCH\_CLASS** 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:

```
class Clientes
{
    public $nombre;
    public $ciudad;
    public $otros;
    public function __construct($otros = '')
    {
        $this->nombre = strtoupper($this->nombre);
        $this->ciudad = mb_substr($this->ciudad, 0, 3);
        $this->otros = $otros;
    }
    // ....Código de la clase....
}
$stmt = $dbh->prepare("SELECT * FROM Clientes");
$stmt->setFetchMode(PDO::FETCH_CLASS, 'Clientes');
$stmt->execute();
while ($objeto = $stmt->fetch()) {
    echo $objeto->nombre . " -> ";
    echo $objeto->ciudad . "<br>";
}
```



Con lo anterior hemos podido modificar cómo queríamos mostrar nombre y ciudad de cada registro. A nombre lo hemos puesto en mayúsculas y de ciudad sólo hemos mostrado las tres primeras letras.

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

```
$stmt->setFetchMode(PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE, 'Clientes');
```

Si en el ejemplo anterior añadimos `PDO::FETCH_PROPS_LATE`, el nombre y la ciudad se mostrarán como aparecen en la base de datos.

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

```
$stmt->setFetchMode(PDO::FETCH_CLASS, 'Clientes', array('masdatos'));
```

O incluso datos diferentes para cada objeto:

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

Para la consulta de datos también se puede emplear directamente `PDOStatement::fetchAll()`, que devuelve un array con todas las filas devueltas por la base de datos con las que poder iterar. También acepta estilos de devolución:

```
// fetchAll() con PDO::FETCH_ASSOC
$stmt = $dbh->prepare("SELECT * FROM Clientes");
$stmt->execute();
$clientes = $stmt->fetchAll(PDO::FETCH_ASSOC);
foreach ($clientes as $cliente) {
    echo $cliente['nombre'] . "<br>";
}

// fetchAll() con PDO::FETCH_OBJ
$stmt = $dbh->prepare("SELECT * FROM Clientes");
$stmt->execute();
$clientes = $stmt->fetchAll(PDO::FETCH_OBJ);
foreach ($clientes as $cliente) {
    echo $cliente->nombre . "<br>";
}
```

## 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.

### Ejemplo con query():

```
$stmt = $dbh->query("SELECT * FROM Clientes");
$clientes = $stmt->fetchAll(PDO::FETCH_OBJ);
foreach ($clientes as $cliente){
    echo $cliente->nombre . "<br>";
}
```

Se cambia *prepare* por *query* y se quita el *execute*.

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

- 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 $dbh->exec("DELETE FROM Clientes WHERE nombre='Luis'");

// No devuelve el número de filas con SELECT, devuelve 0
echo $dbh->exec("SELECT * FROM Clientes");
```

- PDO::lastInsertId(). Este método devuelve el id autoincrementado del último registro en esa conexión:

```
$stmt = $dbh->prepare("INSERT INTO Clientes (nombre) VALUES (:nombre)");
$nombre = "Angelina";
$stmt->bindValue(':nombre', $nombre);
$stmt->execute();
echo $dbh->lastInsertId();
```

- `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.

```
$stmt = $dbh->prepare("SELECT * FROM Clientes");
$stmt->execute();
while ($row = $stmt->fetchColumn(1))
{
    echo "Ciudad: $row <br>";
}
```

- `PDOStatement::rowCount()`. Devuelve el número de filas afectadas por la última sentencia SQL:

```
$stmt = $dbh->prepare("SELECT * FROM Clientes");
$stmt->execute();
echo $stmt->rowCount();
```

## TRANSACCIONES CON PDO

Cuando tenemos que ejecutar varias sentencias de vez, como INSERT, es preferible utilizar transacciones ya que agrupa todas las acciones y permite revertirlas todas en caso de que haya algún error.

Una transacción en PDO comienza con el método `PDO::beginTransaction()`. Este método desactiva cualquier otro commit o sentencia SQL o consultas que aún no son **committed** hasta que la transacción es **committed** con `PDO::commit()`. Cuando este método es llamado, todas las acciones que estuvieran pendientes se activan y la conexión a la base de datos vuelve de nuevo a su estado por defecto que es **auto-commit**. Con `PDO::rollback()` se revierten los cambios realizados durante la transacción.

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

### **Actividad 01**

Los aprendices deberán realizar los siguientes puntos:

1. Seleccionar 3 interfaces del proyecto formativo con sus respectivos archivos HTML, CSS y JAVASCRIPT
2. Crear una clase conexión con base en la guía la cual retorne el objeto conexión y se pueda utilizar para las diferentes consultas
3. Implementar el CRUD en 3 tablas del proyecto formativo con base en la guía y el apoyo del instructor. Se debe tener un menú general que se incluya en todas las páginas