Tema 3: Acceso a DATOS en PHP

Extensión PDO

Existen varias extensiones para conectarse a una BD en PHP, siendo MySQLi y PDO las más utilizadas.

Optaremos por **PDO**, ya que:

- Provee una interfaz uniforme para trabajar con diversos SGBD: Oracle, PostgreSQL, SQLITE, etc
- Es más segura y rápida que otras opciones
- Soporta las consultas parametrizadas
- Posibilita usar transacciones
- Provee de una interfaz orientada a objetos

El sistema PDO se fundamenta en 3 clases:

- **PDO:** se encarga de mantener la conexión a la base de datos, , crear instancias PDOStatement y controlar transacciones
- **PDOStatement:** maneja sentencias SQL y devuelve los resultados.
- **PDOException:** se utiliza para manejar los errores

Conectarse a una base de datos MySQL

Intanciaremos un objeto que representa la conexión a la BD, indicando:

- Data Source
- Usuario
- Password

En el Data Source (1º parámetro) se indicará el driver utilizado, host, base de datos y otras informaciones opcionales como puerto, juego de caracteres,etc

```
try {
   # Conexión a BD MySQL
   $db = new PDO("mysql:host=$host;dbname=$dbname", $user, $pass);
}
catch(PDOException $e) {
    echo $e->getMessage();
}
```

Cerrar conexión

Aunque PHP cierra automáticamente las conexiones a bases de datos al finalizar el script PHP, podemos cerrarlas de modo explícito:

```
$db = null;
unset($db);
```

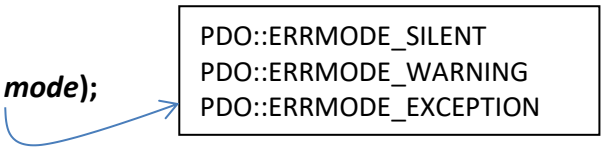
o bien

Excepciones PDO

PDO maneja los errores en forma de excepciones, por ello el bloque try/catch.

Se puede especificar la precisión de manejo de los errores así:

$db ->setAttribute(PDO::ATTRR_ERRMODE, *error mode*);



Consultas de selección

- El método **->query()** ejecuta una sentencia SQL, devolviendo un conjunto de resultados en forma de objeto ***PDOStatement***

- Posteriormente, podemos recorrer el conjunto de resultados mediante los métodos **fetch**:

  - **->fetch()** recupera 1 fila y avanza a la siguiente. Admite varios modos de recuperación de datos:

PDO::FETCH_ASSOC	devuelve cada fila como array asociativo, cuyas claves son los nombres de las columnas en la consulta
PDO::FETCH_NUM	devuelve cada fila como array numérico. El orden de los campos es el de la SELECT
PDO::FETCH_BOTH	valor por defecto, combinación de los 2 anteriores
PDO::FETCH_CLASS	devuelve cada fila como objeto de una clase
PDO::FETCH_OBJ	devuelve cada fila como objeto anónimo con nombres de propiedades que corresponden a las columnas.

  - **->fetchAll()** devuelve un array con todas las filas del conjunto de resultados. También admite personalizar el modo de recuperación

Ejemplos

```
$stmt = $db->query('SELECT nombre, ciudad from clientes');
$stmt->setFetchMode(PDO::FETCH_ASSOC);
while($fila = $stmt ->fetch()) {
   echo $fila['nombre'] . "<br/>";
   echo $fil['ciudad'] . "<br/>";
}
```

```
$stmt = $db->query('SELECT nombre, ciudad from clientes');
$stmt->setFetchMode(PDO::FETCH_OBJ);
while($objeto = $stmt->fetch()) {
   echo $objeto->nombre . "<br/>";
   echo $objeto->ciudad . "<br/>";
}
```

```
class Cliente(){
        public function __construct(….){…….}
        public function metodo(){……}
}
$stmt = $db->query('SELECT nombre, ciudad from clientes');
$stmt ->setFetchMode(PDO::FETCH_CLASS, 'Cliente');
while($obj = $stmt ->fetch()) {
   echo $obj->metodo();
}
```

```
$stmt = $db->query('SELECT nombre, ciudad from clientes');
$stmt->setFetchMode(PDO::FETCH_OBJ);
$personas=$stmt->fetchAll();

foreach($clientes as $cliente) {
   echo $cliente->nombre . "<br/>";
   echo $cliente->ciudad . "<br/>";
}
```

- **->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.
  Util para consultas de agrupados

Ejemplo

```
$imgs = $db->query('SELECT COUNT(id) FROM imagenes');
$this->cantidad = $ imgs ->fetchColumn();
```

Consultas de actualización

- El método **->exec()** ejecuta una consulta de actualización devolviendo el número de filas afectadas

```
$borradas = $db->exec("DELETE FROM frutas WHERE color = 'rojo'");
echo "$borradas filas borradas";

$sql = "UPDATE usuarios SET nombre = '". $_POST['nombre']."' WHERE email = '".$_POST['email']."'";
$actualizadas = $db->exec($sql);
```

- El método **->lastInsertId()** devuelve el valor de la última clave primaria insertada en la BD.
  Útil para claves autonuméricas

```
$conn->exec("INSERT INTO usuarios (nombre, email)  VALUES ('John Doe', 'john@example.com')");
echo "Nueva fila insertada. Ultimo ID insertado: " . $db->lastInsertId();
```

Consultas parametrizadas

En comparación con la ejecución directa de sentencias SQL, las sentencias preparadas tienen varias ventajas:

- Reducen el tiempo de análisis de la consulta, ya que ésta se prepara una sola vez
- Cada vez que se ejecute la consulta, se envian solamente los parámetros (valores) y no toda la consulta
- Evitan las inyecciones de SQL: no es necesario escapar los valores de los parámetros. Estos se transmiten más tarde utilizando un protocolo diferente

Las declaraciones preparadas funcionan en 3 pasos:

1. **Preparar**: se crea una plantilla SQL que se envía a la base de datos. En ella los valores/parámetros no se especifican y quedan marcados por un símbolo especial

   $stmt = $db**->prepare** ("INSERT INTO Clientes (nombre, ciudad) VALUES (**?**, **?**)");
   $stmt = $db**->prepare** ("INSERT INTO Clientes (nombre, ciudad) VALUES (**:nombre**, **:ciudad**)");

2. **Optimizar**:La base de datos analiza, compila y optimiza la consulta

3. **Ejecutar**: se vinculan valores a los parámetros y se obtiene un resultado para dichos valores, tantas veces como sea necesario

   Vinculamos valores: $stmt -> **bindParam** (referencia,valor)
   Ejecutamos: $stmet ->**execute**()

   EJEMPLO: Parámetros referenciados con **INTERROGANTES:**

```
$stmt = $db->prepare("INSERT INTO folks (nombre, edad) values (?, ?, ?)");
$stmt->bindParam(1, $nombre);
$stmt->bindParam(2, $edad);
$stmt->execute();
```

   EJEMPLO: Parámetros referenciados con **NOMBRES:**

```
$stmt =$db->prepare("INSERT INTO folks (nombre, edad) values (:name, :age)");
$stmt->bindParam(':name', $nombre);
$stmt->bindParam(':age', $edad);
$stmt->execute();
```

EJEMPLO: Parámetros referenciados EN ARRAY u OBJETO:

```
$stmt =$db->prepare("INSERT INTO folks (nombre, edad) values (:name, :age)");
$datos = array( 'name' => 'Cathy', 'age' => 9);
$stmt->execute($datos);

$stmt =$db->prepare("INSERT INTO folks (nombre, edad) values (:name, :age)");
$stmt->execute((array)new Persona('Janet',55));
```

EJEMPLO: Select parametrizada

```
$sql = "SELECT id, nombre, precio FROM productos WHERE precio>=:precio";
$stmt = $db->prepare($sql);
$stmt->bindParam(':precio', $precio);
$stmt->execute();
return $stmt->fetchAll(PDO::FETCH_ASSOC);
```

- El método **->rowCount()** también devuelve las filas afectadas por la última consulta de actualización

```
$stmt=$db->prepare("delete from usuarios where edad> (select avg(edad) from usuarios)");
$stmt->execute();
echo $stmt->rowCount() ." usuarios borrados";
```

Transacciones

Una transacción es un conjunto de operaciones atómico, de modo que o se realizan todas (commit) o no se realiza ninguna (rollback)
Las transacciones permiten seguir la pista de los cambios sobre la BD, de manera que se puedan revertir, si es necesario. Durante el tiempo que dura una transacción, los cambios en los datos no son definitivos.

Por ejemplo, deseamos insertar 5 tuplas en una tabla, considerando ésto una acción atómica (o se insertan las 5 o no se inserta ninguna).

-> beginTransaction() : Inicia una transacción, deshabilitando el modo autocommit (que es el modo en el que opera por defecto MySQL)

->commit(): Hace efectiva la transacción y vuelve al modo autocommit

-> rollback(): Deshace la transacción actual y vuelve al modo autocommit

Ejemplo

```
try {
   $db->beginTransaction();
   $dbh->exec("INSERT INTO Clientes (nombre, ciudad) VALUES ('Leila Birdsall', 'Madrid')");
   $dbh-> exec ("INSERT INTO Clientes (nombre, ciudad) VALUES ('Brice Osterberg', 'Teruel')");
   $dbh-> exec ("INSERT INTO Clientes (nombre, ciudad) VALUES ('Latrisha Wagar', 'Valencia')");
   $dbh-> exec ("INSERT INTO Clientes (nombre, ciudad) VALUES ('Hui Riojas', 'Madrid')");
   $dbh-> exec ("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();
}
```

MySQL: Insertar y recuperar campos BLOB

Si deseamos almacenar el archivo subido desde un formulario como campo BLOB (binario) de MySQL

1. Recuperamos el contenido del archivo, por ejemplo, del siguiente modo:

   $contenido =addslashes (file_get_contents($_FILES[‘xxxxxx’][‘tmp_name’]));

   ➔ **file_get_contents** lee y devuelve el contenido total del fichero cuya ruta (nombre) se pasa
   ➔ **addslashes** escapa con barra invertida ‘\’ las comillas simples y dobles

2. y lo asignamos al campo blob en una consulta:

   “update tabla set campo_blob = ‘ “ . $contenido . “ ‘ “;

   El valor del blob va entre comillas, como si fuera un string o una fecha (De ahí addslashes)

Para visualizarlo posteriormente:

```
$imagen=stripslahes($campo_blob);
echo "<img src='".base64_encode($imagen) ." '/>"
```

Funciones **addslashes** y **stripSlahes**

addslashes (string $str):string

Útil cuando trabajamos con BBDD, tomando como partida valores de formularios. **addSlashes()** escapa la comilla simple (‘), la comilla doble (“) y la barra invertida (\): las precede de barra invertida (\ )

Ejemplo:

$consulta = “ update tabla set columna=’ ” . **$_POST[‘comentario’]** . “ ’ where iduser = $id”;

Si el comentario identificado por $_POST[‘comentario’] contiene alguna comilla, ésta podría ser interpretada como comilla de cierre de la consulta, y por lo tanto no funcionaría
(Por ejemplo, para el siguiente comentario: ….Me gustan los libros de O’Reilly……)

La manera de proteger la comilla de O’Reilly para que sea interpretada como un carácter más del comentario (en lugar de cómo carácter de cierre) es precediéndola de la barra invertida

$consulta = “ update tabla set columna=’ ” . **addslashes ($_POST[‘comentario’])** . “ ’ where iduser = $id”;

stripslashes (string $str):string

realiza la función inversa: devuelve una cadena sin barras de escape delante de la comilla simple (‘), la comilla doble (“) y la barra invertida (\)

ATENCION! Si la directiva de php **magic_quotes_gpc** está activada, se aplica automáticamente addslahes a todos los datos GET, POST y COOKIE, y en este caso, un doble escape, haría fallar el código. (Está en Off por defecto)

Funciones **urlencode** y **urldecode**

urlencode (string $str) : str

urlencode (string $str) : str

urlencode codifica los valores de parámetros enviados a través de una URL, (GET)
Devuelve una cadena en la que los espacios se representan con el signo +, las letras, excepto la ñ, y los números se dejan igual, y los demás signos, como <, =,), etcétera, son sustituidos por el signo % seguido de dos dígitos en base hexadecimal.

!	#	$	%	&	'	(	)	*	+	,	/	:	;	=	?	@	[	]
%21	%23	%24	%25	%26	%27	%28	%29	%2A	%2B	%2C	%2F	%3A	%3B	%3D	%3F	%40	%5B	%5D

Posteriormente, cuando se procese la URL, se podrán recuperar los valores originales utilizando **urldecode**

Ejemplo

Supongamos que en la URL se quisieran enviar estas combinaciones de parámetro/valor:

```
<?php
        $web= “https://miwebpersonal.org/”;
        $name=”García Pérez-Argüello”;

         echo “<a href=’http://destino.php?web=$web&name=$name’>Link</a>”;
?>
```

Vemos que existen espacios, tildes, caracteres especiales, algo que nos dará un pequeño dolor de cabeza si no codificamos adecuadamente esta consulta.

Para solucionar lo anterior, habría que enviar los valores de los parámetros GET codificados con **urlencode**, y tras recibirlos, decodificarlos con **urldecode**

```
<?php
        $web== “https://miwebpersonal.org/”;
        $name=”García Pérez-Argüello”;

         echo “<a href=’http://destino.php?web =” . urlencode($web) .” &name=” . urlencode($name).
“’>Link</a>”;
?>
```

Cuando un formulario es enviado via GET, los campos son automáticamente codificados (urlendoded) al enviarse y descodificados (urldecoded) al recuperarse.

Almacenamiento de **contraseñas**

Es fundamental que las contraseñas que se guarden en bases de datos lo hagan con una encriptación segura y eficiente.
El algoritmo de encriptación debe asegurar no sólo la protección frente a ataques, sino también frente a los propios empleados de la aplicación.
Hasta ahora, para crear hashes, era suficiente con algoritmo como md5 o sha1, pero hoy en día, estos algoritmos se pueden crackear fácilmente.
La librería Hash de contraseñas PHP crea passwords complejas, incluyendo la generación de salts aleatorios.

Obtener contraseña encriptada

password_hash (string $password , integer $algoritmo [, array $options]) : string

Esta función hashea (encripta) la contraseña plana que se le pasa como 1º parámetro, con el algoritmo pasado como 2º parámetro (PASSWORD_DEFAULT o PASSWORD_BCRYPT).

Ejemplo

```
$hash = password_hash('micontraseña', PASSWORD_DEFAULT, [15]);
```

** El coste indica cuánto de complejo debe ser el algoritmo y por lo tanto cuánto tardará en generarse el hash. El número se puede considerar como el número de veces que el algoritmo hashea la contraseña.*

Verificar contraseña

password_verify (string $password, string $hash): bool

Ejemplo

```
if (password_verify($_POST['password'], $fila->password)) {
        echo 'Password válida!';
```