

# Tema 05 – php y MySQL

---

2º DAW – Desarrollo Web Entorno Servidor

**Profesor Juan Carlos Moreno**

**CURSO 2024/2025**

## Tabla de Contenido

6	Php y MySQL .....	3
6.1	Introducción .....	3
6.2	Extensiones PHP para interactuar con Base de Datos .....	3
6.2.1	Extensión mysql .....	3
6.2.2	Extensión mysqli .....	3
6.2.3	Extensión PDO .....	5
6.2.4	Comparación de Métodos .....	6
6.3	Extensión mysqli .....	6
6.3.1	Conexión con la Base de Datos .....	6
6.3.2	Ejecución de sentencias SQL .....	8
6.3.3	Clase mysqli_result .....	9
6.3.4	Cierre de conexiones .....	11
6.3.5	Ejemplo completo .....	12
6.4	Sentencias Preparadas .....	12
6.5	Manejo de Excepciones .....	16
6.5.1	Crear y capturar una excepción .....	16
6.5.2	Control de Excepciones con MySQL .....	19
6.6	Transacciones .....	20
6.6.1	Las instrucciones SQL relacionadas con las transacciones son las siguientes: .....	20
6.6.2	Ejemplo 1. Transacciones con PHP .....	20
6.6.3	Ejemplo 2. Inserción en Tablas Maestro – Detalles .....	21
6.7	PDO – PHP Data Objects .....	22
6.7.1	Conectar una Base de Datos con PDO .....	23
6.7.2	Excepciones y opciones con PDO .....	23
6.7.3	Registrar Datos con PDO .....	24
6.7.4	Diferencia entre bindParam() y bindValue() .....	26
6.7.5	Consultar Datos con PDO .....	27
6.7.6	Diferencia entre query() y prepare()/execute() .....	29
6.7.7	Otras funciones de utilidad .....	29
6.7.8	Transacciones con PDO .....	30

## 6 Php y MySQL

### 6.1 Introducción

En este capítulo vamos a ver cómo gestionar la comunicación de estas aplicaciones web con una base de datos, la cual vamos a utilizar para almacenar toda la información. Esto conlleva aprender la sintaxis necesaria para conectarnos con la base de datos y las sentencias SQL de inserción, actualización, borrado y selección.

También veremos cómo utilizar la información recuperada de la base de datos, como realizar transacciones y cómo acceder a la base de datos de manera serializable para que la base de datos no se quede en ningún momento inconsistente. Esto puede ocurrir debido a que la base de datos con frecuencia deberá soportar el acceso de varias personas a la vez.

Por último, explicaremos cómo obtener la información de otro tipo de fuentes que no sean la propia base de datos.

### 6.2 Extensiones PHP para interactuar con Base de Datos

A la hora de interactuar con una base de datos lo primero que tenemos que hacer es establecer la conexión desde nuestra aplicación web para que posteriormente podamos ejecutar las sentencias SQL (Structured Query Language) que necesitemos de entre todo el abanico de sentencias que nos ofrece este lenguaje.

PHP ofrece diferentes formas de conectarse a bases de datos MySQL y ejecutar comandos SQL dependiendo de la versión que se use. A continuación, se detallan las funciones y clases principales disponibles en distintas versiones:

#### 6.2.1 Extensión mysql

(Deprecada desde PHP 5.5.0, eliminada en PHP 7.0.0)

- Proporciona funciones simples para conectarse y trabajar con bases de datos MySQL.
- Funciones principales:
  - `mysql_connect()`: Abre una conexión a un servidor MySQL.
  - `mysql_select_db()`: Selecciona una base de datos MySQL.
  - `mysql_query()`: Ejecuta una consulta SQL.
  - `mysql_fetch_assoc()`: Obtiene una fila como un array asociativo.
  - `mysql_close()`: Cierra la conexión con el servidor.

**Nota:** Esta extensión ya no debe usarse debido a problemas de seguridad y rendimiento.

#### 6.2.2 Extensión mysqli

(MySQL Improved, PHP 5.x y versiones posteriores)

Esta extensión mejora las capacidades de la anterior, con soporte para Programación Orientada a Objetos (POO) y procedimientos.

## Modo Orientado a Procedimientos

### ■ Funciones principales:

- `mysqli_connect()`: Establece una conexión a un servidor MySQL.
- `mysqli_select_db()`: Selecciona la base de datos.
- `mysqli_query()`: Ejecuta una consulta SQL.
- `mysqli_fetch_assoc()`: Obtiene una fila como array asociativo.
- `mysqli_close()`: Cierra la conexión.

### Ejemplo

```
php Copiar código  
  
<?php  
// Datos de conexión  
$conexion = mysqli_connect("localhost", "root", "", "mi_base");  
  
// Verificar conexión  
if (!$conexion) {  
    die("Error de conexión: " . mysqli_connect_error());  
}  
  
// Ejecutar consulta  
$resultado = mysqli_query($conexion, "SELECT * FROM tabla");  
  
while ($fila = mysqli_fetch_assoc($resultado)) {  
    echo $fila['id'] . " - " . $fila['nombre'] . "<br>";  
}  
  
// Cerrar conexión  
mysqli_close($conexion);  
?>
```

## Modo Orientado a Objetos

### ■ Clases principales:

- `mysqli`: Clase principal para la conexión.
- Métodos comunes:
  - `$mysqli->connect()`: Establece una conexión.
  - `$mysqli->query()`: Ejecuta una consulta SQL.
  - `$mysqli->close()`: Cierra la conexión.
- `mysqli_stmt`: Para sentencias preparadas.
- `mysqli_result`: Para manejar resultados de consultas.

### Ejemplo

```
php Copiar código

$mysqli = new mysqli("localhost", "usuario", "contraseña", "base_datos");

if ($mysqli->connect_error) {
    die("Error de conexión: " . $mysqli->connect_error);
}

$resultado = $mysqli->query("SELECT * FROM tabla");
while ($fila = $resultado->fetch_assoc()) {
    echo $fila['columna'];
}

$mysqli->close();
```

### 6.2.3 Extensión PDO

(PHP Data Objects, PHP 5.1.0 y posteriores)

Es una interfaz unificada para trabajar con múltiples bases de datos, incluyendo MySQL.

Clases principales:

- PDO: Clase principal para la conexión.
- PDOStatement: Clase para manejar resultados.

Métodos principales:

- new PDO(): Establece la conexión.
- query(): Ejecuta una consulta SQL.
- prepare(): Prepara una consulta para ejecutar con parámetros.
- execute(): Ejecuta una consulta preparada.
- fetch(), fetchAll(): Recupera datos del resultado.

Ejemplo

```

php
try {
    $pdo = new PDO("mysql:host=localhost;dbname=base_datos", "usuario", "contraseña");
    $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    $stmt = $pdo->query("SELECT * FROM tabla");
    while ($fila = $stmt->fetch(PDO::FETCH_ASSOC)) {
        echo $fila['columna'];
    }
} catch (PDOException $e) {
    echo "Error: " . $e->getMessage();
}

```

#### 6.2.4 Comparación de Métodos

Características	mysql	mysqli	PDO
Soporte POO	No	Sí	Sí
Consultas preparadas	No	Sí	Sí
Múltiples bbdd	No (MySQL)	No (MySQL)	Sí
Deprecada/obsoleto	SI	No	No

#### Recomendación

- Usa **PDO** si necesitas trabajar con múltiples tipos de bases de datos o quieres escribir código más flexible y seguro.
- Usa **mysqli** si necesitas funcionalidades específicas de MySQL y no planeas cambiar de base de datos.
- Evita **mysql**, ya que está completamente eliminado en versiones modernas de PHP.

### 6.3 Extensión mysqli

La extensión mysqli (MySQL Improved) es una herramienta de PHP para interactuar con bases de datos MySQL. Sustituye a la antigua extensión mysql (ya obsoleta) y ofrece soporte para programación orientada a objetos, sentencias preparadas, y mayor seguridad y rendimiento.

Como hemos visto en el apartado anterior mysqli puede utilizarse de dos formas principales:

- Modo Procedural: Más parecido al antiguo mysql, con funciones.
- Modo Orientado a Objetos (POO): Usando clases y métodos.

#### 6.3.1 Conexión con la Base de Datos

Para conectarnos con la base de datos tenemos que facilitar:

- \$servidor. El servidor donde se encuentra la base de datos. El servidor se puede especificar facilitando el nombre del servidor, seguido del puerto (nombre\_ host :puerto) o mediante una ruta para un servidor local (: ruta)
- \$usuario. El usuario
- \$contraseña. La contraseña de acceso que tiene \$user a la base de datos.
- \$base\_datos. Base de datos a la que nos queremos conectar

#### Modo Procedural

```
php Copiar código  
  
<?php  
// Datos de conexión  
$servidor = "localhost"; // Dirección del servidor (puede ser '127.0.0.1' o un dominio)  
$usuario = "root"; // Usuario de la base de datos  
$contrasena = ""; // Contraseña del usuario  
$base_datos = "mi_base"; // Nombre de la base de datos  
  
// Establecer conexión  
$conexion = mysqli_connect($servidor, $usuario, $contrasena, $base_datos);  
  
// Verificar conexión  
if (!$conexion) {  
    die("Error al conectar con la base de datos: " . mysqli_connect_error());  
}  
echo "Conexión exitosa a la base de datos."  
  
// Consulta a la base de datos  
$sql = "SELECT * FROM tabla_ejemplo";  
$resultado = mysqli_query($conexion, $sql);  
  
if ($resultado) {  
    // Mostrar resultados  
    while ($fila = mysqli_fetch_assoc($resultado)) {  
        echo "ID: " . $fila['id'] . " - Nombre: " . $fila['nombre'] . "<br>";  
    }  
} else {  
    echo "Error en la consulta: " . mysqli_error($conexion);  
}  
  
// Cerrar la conexión  
mysqli_close($conexion);  
?>
```

#### Modo POO

```
php Copiar código  
  
<?php  
$conexion = new mysqli("localhost", "root", "", "mi_base");  
  
if ($conexion->connect_error) {  
    die("Error de conexión: " . $conexion->connect_error);  
}  
echo "Conexión exitosa";  
$conexion->close();  
?>
```

En este apartado vemos dos propiedades muy importantes que nos pueden ser muy útiles en nuestro desarrollo web:

- **connect\_errno:** Nos devuelve un número de error al conectarnos a la base de datos. También se podría haber usado la función `mysqli_connect_errno()`.
- **connect\_error:** Nos devuelve la descripción del error. También se podría usar la función `mysqli_connect_error()`.

### 6.3.2 Ejecución de sentencias SQL

Una vez que hemos logrado conectar la base de datos a nuestra aplicación, ya estamos preparados para ejecutar cualquier tipo de sentencia SQL. Más tarde aprenderemos a utilizar los resultados obtenidos tras la ejecución de estas sentencias.

Dentro de las sentencias SQL que podremos ejecutar existen tres tipos:

- Sentencias de definición de datos (DDL)
- Sentencias de manipulación de datos (DML)
- Sentencias de control (DCL)

En PHP existe el método **query()** o la función equivalente **mysqli\_query()** que ejecuta las sentencias SQL sobre la base de datos conectada y devuelve un valor booleano o un objeto según use la función o el método.

Devolverá un **valor booleano** en el caso de ejecutar sentencias SQL que no devuelvan información al usuario, como, por ejemplo: **la inserción**, el **borrado de datos**, **actualización**, etc. En estos casos el valor booleano informa si la ejecución de la instrucción SQL se ha realizado de forma satisfactoria (true) o por el contrario se ha producido algún error (falsa).

Por otro lado, este método cuando se ejecuta una sentencia SQL de tipo SELECT devuelve un objeto de la clase **mysqli\_result** que contiene el resultado de ejecutar dicha sentencia.

#### Ejemplo



```
php Copiar código  
  
<?php  
$conexion = new mysqli("localhost", "root", "", "mi_base");  
  
$sql = "SELECT * FROM usuarios";  
$resultado = $conexion->query($sql);  
  
while ($fila = $resultado->fetch_assoc()) {  
    echo $fila['id'] . " - " . $fila['nombre'] . "<br>";  
}  
  
$conexion->close();  
?>
```

Debemos tener claro que **\$resultado** es un objeto de la clase **mysqli\_result**.

### 6.3.3 Clase mysqli\_result

Un objeto `mysqli_result` se genera automáticamente cuando se ejecuta una consulta `SELECT` con `mysqli_query()` o con un método como `$mysqli->query()`.

```
php Copiar código  
  
<?php  
$conexion = new mysqli("localhost", "root", "", "mi_base");  
  
$resultado = $conexion->query("SELECT * FROM usuarios");  
if ($resultado instanceof mysqli_result) {  
    echo "Consulta exitosa. Total de filas: " . $resultado->num_rows;  
}  
$conexion->close();  
?>
```


#### Propiedades

- `$resultado->num_rows`. Devuelve el número de filas
- `$resultado->fetch_assoc`. Devuelve del conjunto de resultados como array asociativo
- `$resultado->fetch_row`. Devuelve una fila como array indexado numérico.
- `$resultado->fetch_obj`. Devuelve una fila como objeto de la clase estándar.
- `$resultado->data_seek`. Establece el puntero en una fila específica del conjunto de resultado

#### Método num\_rows()

Devuelve el número de filas en el conjunto de resultados.

php

 Copiar código

```
<?php
$conexion = new mysqli("localhost", "root", "", "mi_base");
$resultado = $conexion->query("SELECT * FROM usuarios");


echo "Número de filas: " . $resultado->num_rows;

$resultado->free(); // Liberar el objeto mysqli_result
$conexion->close();
?>
```

### Método fetch\_assoc()

Devuelve una fila del conjunto de resultados como un array asociativo.

php

 Copiar código

```
<?php
$conexion = new mysqli("localhost", "root", "", "mi_base");
$resultado = $conexion->query("SELECT * FROM usuarios");


while ($fila = $resultado->fetch_assoc()) {
    echo "ID: " . $fila['id'] . " - Nombre: " . $fila['nombre'] . "<br>";
}

$resultado->free(); // Liberar memoria
$conexion->close();
?>
```

### Método fetch\_row()

Devuelve una fila como un array indexado numéricamente.

php

 Copiar código

```
<?php
$conexion = new mysqli("localhost", "root", "", "mi_base");
$resultado = $conexion->query("SELECT * FROM usuarios");

while ($fila = $resultado->fetch_row()) {
    echo "ID: " . $fila[0] . " - Nombre: " . $fila[1] . "<br>";
}

$resultado->free();
$conexion->close();
?>
```

### Método fetch\_object()

Devuelve una fila como un objeto de la clase estándar.

```
php Copiar código  
  
<?php  
$conexion = new mysqli("localhost", "root", "", "mi_base");  
$resultado = $conexion->query("SELECT * FROM usuarios");  
  
while ($fila = $resultado->fetch_object()) {  
    echo "ID: " . $fila->id . " - Nombre: " . $fila->nombre . "<br>";  
}  
  
$resultado->free();  
$conexion->close();  
?>
```

### Método data\_seek()


Establece el puntero en una fila específica del conjunto de resultados.

```
php Copiar código  
  
<?php  
$conexion = new mysqli("localhost", "root", "", "mi_base");  
$resultado = $conexion->query("SELECT * FROM usuarios");  
  
// Mover el puntero a la segunda fila  
$resultado->data_seek(1);  
  
$fila = $resultado->fetch_assoc();  
echo "Fila actual: ID: " . $fila['id'] . " - Nombre: " . $fila['nombre'];  
  
$resultado->free();  
$conexion->close();  
?>
```

#### 6.3.4 Cierre de conexiones

Es importante liberar los recursos asociados con un objeto `mysqli_result` una vez que se ha terminado de usar. Esto se hace con el método `free()`. Por otro lado Una vez que ya hemos terminado de trabajar con la base de datos es preciso cerrar la conexión con el método `close()`

php

 Copiar código

```
<?php
$conexion = new mysqli("localhost", "root", "", "mi_base");
$resultado = $conexion->query("SELECT * FROM usuarios");

// Procesar resultados...
$resultado->free(); // Liberar memoria

$conexion->close();
?>
```

### 6.3.5 Ejemplo completo

php

 Copiar código

```
<?php
$conexion = new mysqli("localhost", "root", "", "mi_base");

if ($conexion->connect_error) {
    die("Error de conexión: " . $conexion->connect_error);
}

$sql = "SELECT id, nombre FROM usuarios";
$resultado = $conexion->query($sql);

if ($resultado instanceof mysqli_result) {
    echo "Número de filas: " . $resultado->num_rows . "<br>";

    // Mostrar todas las filas
    while ($fila = $resultado->fetch_assoc()) {
        echo "ID: " . $fila['id'] . " - Nombre: " . $fila['nombre'] . "<br>";
    }

    // Liberar memoria del resultado
    $resultado->free();
} else {
    echo "Error en la consulta: " . $conexion->error;
}

$conexion->close();
?>
```



## 6.4 Sentencias Preparadas

Una declaración preparada es una función que se utiliza para ejecutar las mismas (o similares) sentencias SQL repetidamente con una alta eficiencia.

Las declaraciones preparadas básicamente funcionan como sigue:

1. **Prepare.** Una plantilla de instrucción SQL se crea y se envía a la base de datos. Algunos valores no se especifican llamado parámetros (con la etiqueta "?"). Ejemplo: INSERT INTO Clientes VALUES (?, ?, ?)
2. **Análisis y optimización.** La base de datos analiza, compila, y lleva a cabo la optimización de consultas en la plantilla de instrucción SQL, y almacena el resultado sin ejecutarlo.
3. **Ejecutar:** En un momento posterior, la aplicación se une a los valores a los parámetros, y la base de datos ejecuta la instrucción. La aplicación puede ejecutar la instrucción tantas veces como se quiera con diferentes valores.

En comparación con la ejecución de las sentencias SQL directamente, las declaraciones preparadas tienen las siguientes ventajas:

- **Reduce tiempo análisis.** Declaraciones preparadas reducen el tiempo de análisis ya que la preparación de la consulta se realiza sólo una vez (aunque la sentencia se ejecuta varias veces)
- **Minimiza ancho banda.** Se minimiza el ancho de banda al servidor ya que se necesite enviar sólo los parámetros cada vez, y no toda la consulta.
- **Mayor seguridad.** Declaraciones preparadas son muy útiles contra inyecciones SQL, porque los valores de los parámetros, que se transmiten más tarde usando un protocolo diferente, no necesitan ser escapados. Si la plantilla de declaración original no se deriva de la entrada externa, no puede ocurrir la inyección de SQL.

Vamos a ver un ejemplo de Sentencias Preparadas en PHP con MySQLi.

```
php Copiar código

<?php
// Configuración de La conexión
$server = "localhost";
$user = "usuario";
$password = "password";
$dbname = "ejemplo";

// 1. Conexión a La base de datos
$db = new mysqli($server, $user, $password, $dbname);

// Verificar conexión
if ($db->connect_error) {
    die("La conexión ha fallado, error número " . $db->connect_errno . ": " . $db->connect_error);
}

// 2. Preparar La sentencia
$stmt = $db->prepare("INSERT INTO Clientes (nombre, ciudad, contacto) VALUES (?, ?, ?)");
if (!$stmt) {
    die("Error al preparar la consulta: " . $db->error());
}

// Vincular parámetros ('ssi' = string, string, integer)
$stmt->bind_param('ssi', $nombre, $ciudad, $contacto);

// 3. Establecer parámetros y ejecutar varias inserciones
$nombre = "Donald Trump";
$ciudad = "Madrid";
$contacto = 4124124;
$stmt->execute(); // Primera ejecución

$nombre = "Hillary Clinton";
$ciudad = "Barcelona";
$contacto = 4665767;
$stmt->execute(); // Segunda ejecución

// 4. Mensaje de éxito
echo "Se han creado las entradas exitosamente";

// 5. Cerrar La sentencia y La conexión
$stmt->close();
$db->close();
?>
```

Incluimos un interrogante donde queremos sustituir un integer, un string, un double o un blob en la función prepare(). Después usamos la función bind\_param():

```
$stmt->bind_param('ssi', $nombre, $ciudad, $contacto);
```

La función enlaza los parámetros con la consulta SQL y le dice a la base de datos que parámetros son. El argumento "ssi" especifica el tipo de dato que se espera que sea el parámetro. Pueden ser de cuatro tipos:

- Letra i - número entero
- Letra d - doble
- Letra s - string
- Letra b - BLOB

Se debe especificar uno por cada parámetro.

### Método `Bind_result()`.

Este método se usa para vincular las columnas del resultado a variables específicas, lo que permite acceder a los valores de las filas devueltas por una consulta.

Veamos el siguiente ejemplo:

```
// 1. Establecer conexión
$db = new mysqli($server, $user, $password, $dbname);

// Verificar conexión
if ($db->connect_error) {
    die("Error de conexión: " . $db->connect_errno . " - " . $db->connect_error);
}

// 2. Preparar la consulta
$stmt = $db->prepare("SELECT id, nombre, ciudad FROM Clientes WHERE ciudad = ?");
if (!$stmt) {
    die("Error al preparar la consulta: " . $db->error);
}

// 3. Vincular parámetros para la consulta (en este caso, ciudad)
$ciudad_buscar = "Madrid";
$stmt->bind_param("s", $ciudad_buscar);

// 4. Ejecutar la consulta
$stmt->execute();

// 5. Vincular los resultados a variables
$stmt->bind_result($id, $nombre, $ciudad);

// 6. Recuperar y procesar los resultados
echo "Resultados para la ciudad: $ciudad_buscar<br>";
while ($stmt->fetch()) {
    echo "ID: $id - Nombre: $nombre - Ciudad: $ciudad<br>";
}

// 7. Cerrar la sentencia y la conexión
$stmt->close();
$db->close();
?>
```



Este método es útil cuando quieres procesar los resultados directamente en variables específicas, evitando la necesidad de usar métodos como `fetch_assoc()` o `fetch_row()`.

## 6.5 Manejo de Excepciones

Una excepción es un objeto derivado de la clase *Exception* de PHP que se encarga de mantener e informar de cualquier error producido. Por lo tanto, su uso principal es para detener la ejecución del programa, notificar de errores y ayudar a depurar información.

Una excepción se creará ante una situación anómala en la ejecución del programa que provoca que este no pueda continuar con normalidad.

La clase *Exception* recibe en su constructor dos argumentos opcionales: mensaje y código de error a mostrar.

Algunos de los métodos más importantes de la clase *Exception* y que debemos de conocer son los siguientes:

- **getMessage()**: Devuelve la cadena de error que le llega al constructor de la clase *Exception*.
- **getCodeError()**: Devuelve el entero que representa el código de error que llega al constructor de la clase *Exception*.
- **getFile()**: Devuelve el fichero donde se ha producido la excepción.
- **getLine()**: Devuelve la línea donde se produjo la excepción.
- **getTrace()**: Devuelve un array multidimensional con información del fichero y línea donde se produce la excepción, además de la función/método donde se produce la misma y los argumentos de entrada.
- **getTraceAsString()**: Devuelve la información de la función anterior pero en formato de cadena.

### 6.5.1 Crear y capturar una excepción

El manejo de excepciones en PHP permite capturar y gestionar errores de manera más estructurada y controlada. A partir de PHP 8.1, la clase *mysqli* lanza excepciones (*mysqli\_sql\_exception*) automáticamente para errores en lugar de devolver valores false. Esto simplifica el manejo de errores al usar try-catch.

Crear una excepción es tan sencillo como utilizar la siguiente sintaxis en el lugar que nos parezca adecuado:

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

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.

#### Estructura básica try-catch



La estructura try-catch permite capturar excepciones y ejecutar un código específico para manejarlas.

```
php Copiar código  
  
try {  
    // Código que puede generar una excepción  
} catch (Exception $e) {  
    // Código para manejar la excepción  
    echo "Error: " . $e->getMessage();  
}
```

De esta forma dividiremos el manejo de excepciones en dos partes:

- 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.
- Dentro de 'catch' introduciremos el código que controlará la excepción. Mostrando el error y aplicando la funcionalidad necesaria para actuar ante el error.

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 sí que se ejecutará el código tras el bloque try/catch. Por lo que si queremos detener totalmente el programa

Por lo tanto, el uso de las excepciones se basa en dos etapas:

- Cuando ocurra algún error, una excepción será lanzada en un punto del script.
- Para mostrar el error y actuar en consecuencia, es necesario capturar dicha excepción.

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

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

La ejecución daría FATAL ERROR.

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. Por lo que vamos a mejorar el ejemplo para captura dicha excepción.

```
class Prueba {  
    protected $id;  
  
    public function __construct($id) {  
        try {  
            if ($this->validId($id))  
                $this->id = $id;  
            else  
                throw new Exception('Identificiador no es un entero');  
        } catch (Exception $e) {  
            echo $e->getMessage();  
            print_r($e->getTrace());  
            exit();  
        }  
    }  
  
    public function validId($id) {  
        if (!is_int($id))  
            return false;  
        else  
            return true;  
    }  
}  
  
$p = new Prueba('prueba');
```

En el ejemplo anterior se ha mostrado como lanzar y capturar una excepción en un mismo método. Pero también es habitual que los métodos lancen las excepciones y estas se capturen al utilizar el objeto instanciado de la clase.

```
class Prueba {  
    protected $id;  
  
    public function __construct($id) {  
        if ($this->validId($id))  
            $this->id = $id;  
        else  
            throw new Exception('Identificiador no es un entero');  
    }  
  
    public function validId($id) {
```

```
        if (!is_int($id))
            return false;
        return true;
    }
}

try{
    $p = new Prueba('prueba');
} catch (Exception $e){
    echo $e->getMessage() . "<br/>";
    echo $e->getTraceAsString();
    exit();
}
```

### 6.5.2 Control de Excepciones con MySQL

Podemos controlar si hay algún problema con la conexión a la base de datos. En estos casos el bloque 'catch' no espera un objeto Exception, sino que espera un objeto DBException. Que es una instancia de nuestra excepción.

```
class Test {
    const servidor = "localhost";
    Const usuario_db = "usuario";
    Const pwd_db = "1234";
    Const nombre_db = "demo";
    private $conn = NULL;

    public function __construct() {
        $this->conectarDB();
    }

    public function conectarDB() {
        try {
            $this->conn = new mysqli(self::servidor,
                self::usuario_db, self::pwd_db, self::nombre_db);

            if ($this->conn->connect_error) {
                throw new DBException('Fallo en la conexion con la BD: ' .
                    $this->conn->connect_error);
            }
        } catch (DBException $e) {
            echo $e->getError();
            exit();
        }
    }
}

$t = new Test();
```

## 6.6 Transacciones

Una transacción es un conjunto de operaciones SQL que se ejecutan como un único bloque, es decir, si falla una operación fallan todas. Es una unidad única de trabajo. Si una transacción tiene éxito, todas las modificaciones de los datos realizadas durante la transacción se confirman y se convierten en una parte permanente de la base de datos. Si una transacción encuentra errores y debe cancelarse o revertirse, se borran todas las modificaciones de los datos.

Un ejemplo de transacción puede ser una transferencia bancaria. Quitamos saldo a una cuenta y lo añadimos en otra. Si no somos capaces de abonar el dinero en la cuenta de destino, no podemos quitarlo de la cuenta de origen. Otro ejemplo puede ser escribir datos en dos tablas donde una de ellas es maestro y la otra es detalle. Si tenemos que escribir una factura y los movimientos de la factura, primero escribimos la cabecera de la factura y después cada una de las líneas de esa factura. Si al insertar una de las líneas de la factura hay un error, lo suyo es que se elimine la cabecera de la factura también. Esto lo veremos en el siguiente punto.

En principio php no realiza ninguna función dentro de las transacciones. La única cuestión real que tiene importancia dentro de tu código php es que realice las queries de manera ordenada hacia MySQL.

Para trabajar con transacciones con php y mysql es importante tener en cuenta:

1. Que tu base de datos tenga soporte para tablasinnodb.
2. Las transacciones sólo son soportadas para las tablas tipo innodb, así que necesitamos crear las tablas o modificarlas con éste tipo. ¿Cómo me aseguro que mis tablas sean innodb? Ejecutando la siguiente query en tu base de datos : ALTER TABLE tutable ENGINE = INNODB;

### 6.6.1 Las instrucciones SQL relacionadas con las transacciones son las siguientes:

- `mysql::query("START TRANSACTION")`: Si devuelve true, es que se han activado las transacciones y false en caso contrario. Se usar por debajo de php 5.5
- `mysql::begin_transaction()`: Se introduce a partir de php 5.5. Si devuelve true, es que se han activado las transacciones y false en caso contrario.
- `mysql::commit()`: envía todas las operaciones a la base de datos
- `mysql::rollback()`: aborta todas las operaciones.

### 6.6.2 Ejemplo 1. Transacciones con PHP

```
$mysql= @new mysqli("localhost", "root", "", "fct_ubrique");
if ($mysql->connect_errno){
    echo "Error al conectar con la base de datos ".$mysql->connect_error;
}
$mysql->set_charset("utf8");
//activamos las transacciones. Si tenemos php 5.5 o superior usar $mysql-
>begin_transaction();
$mysql->autocommit("START TRANSACTION")or die("Error al activar las
transacciones");
//Ejecutamos los inserts.
```

```
try{
    if (!$mysql->query("insertinto profesor(nombre, verificar, email)
values('pp1', 'S', 'm1')" ))
        throw new Exception("Error al insertar");
    if (!$mysql->query("insertinto profesor(nombre, verificar, email)
values('pp2', 'S', 'm2')" ))
        throw new Exception("Error al insertar");
    //si no hay error, entonces enviamos todas las inserciones a la base de
datos.
    $mysql->commit();
}catch(Exception $e){
    $mysql->rollback(); //en caso contrario, abortamos todas las
inserciones.
    echo $e->getMessage();
}
//mostramos el resultado.
mostrar($mysql);
$mysql->close();
```

Donde mostrar() sería

```
Function mostrar($mysql)
{
    $res = $mysql->query("select * from profesor");
    echo "<table>";
    while ( $fila=$res->fetch_array() )
    {
        printf("<tr><td>%s</td><td>%s</td><td>%s</td></tr>", $fila["id"],
        $fila["nombre"], $fila["email"]);
    }
    echo "</table>";
}
```

### 6.6.3 Ejemplo 2. Inserción en Tablas Maestro – Detalles

En este ejemplo, tenemos dos tablas para almacenar una factura. Una factura consta de un encabezamiento, donde se ponen datos tales como el importe de la factura, su fecha, el nombre del cliente y su dirección, etc... En la tabla donde se almacenan los detalles se inserta un registro por cada artículo o servicio que va en la factura y se almacenan datos como el identificador de la cabecera, el código del artículo o servicio, las unidades de ese artículo, el precio por unidad y el importe total de ese artículo o servicio. Es importante que cuando se cree o se modifique una factura se haga en bloque, ya que un error en inserción, un corte de luz por ejemplo, podría ocasionar en situaciones como estas:

- Se ha creado el registro para la cabecera padre, pero no se han creado los detalles.
- Se han guardado parte de los detalles pero no todos.
- Se han creado los hijos, pero no se ha creado el padre por un error indeterminado en la base de datos.

La solución para que esto no ocurra es lanzar, todas las inserciones, propias de una factura, en un bloque. En ese bloque se inserta y se dan por buenas las transacciones, con el commit.

Fuera del bloque se pone un rollback y se le llama si hay algún error, con lo cual no se aborta toda la transacción completa. El siguiente ejemplo implementa lo dicho:

```
/*conectar con la base de datos*/
$mysql=new mysqli("localhost", "root", "", "test");

if ($mysql->connect_errno){
    die( "Error al conectar con la base de datos. Activar el servidor");
}

/*La inserción de maestro y detalle se guardan en un bloque try*/
try{
    $mysql->autocommit("START TRANSACTION");

    /*abrir transacción, por debajo de php 5.5,
    A partir de 5.5 usarmysql->begin_transaction()*/

    /*insertamos la cabecera*/
    if (!$mysql->query("insertintofactura_cab(numero, titular, total)
values('0005', 'Miguel Angel', 300)"))
        throw new Exception("Error al insertar el maestro");

    /*obtener el id*/
    $id=$mysql->insert_id;

    /*insertamos los detalles, ponemos dos registros como ejemplo*/
    if (!$mysql->query("insertintofactura_det(id_cabecera, articulo, precio)
values($id, 'casco',
200)")) throw new Exception("Error al insertar los detalles");
    if (!$mysql->query("insertintofactura_det(id_cabecera, articulo, precio)
values($id,
'guantes', 100)")) throw new Exception("Error al insertar los detalles");
    /*Si llegamos aquí es que todo ha ido bien, validamos la transacción*/
    $mysql->commit();
}catch( Exception $e){ /*aquí se entra si hay error*/
    echo $e->getMessage(), "\n";
    $mysql->rollback();
}
/*cerrar la conexión*/
$mysql->close();
```

## 6.7 PDO – PHP Data Objects

PDO significa PHP Data Objects, Objetos de Datos de PHP, una extensión para acceder a bases de datos. PDO permite acceder a diferentes sistemas de bases de datos con un controlador específico (MySQL, SQLite, Oracle...) mediante el cual se conecta. Independientemente del sistema utilizado, se emplearán siempre los mismos métodos, lo que hace que cambiar de uno a otro resulte más sencillo.

El sistema PDO se fundamenta en 3 clases: PDO, PDOStatement y PDOException. La clase PDO se encarga de mantener la conexión a la base de datos y otro tipo de conexiones específicas como transacciones, además de crear instancias de la clase PDOStatement. Es ésta clase, PDOStatement, la que maneja las sentencias SQL y devuelve los resultados. La clase PDOException se utiliza para manejar los errores.

### 6.7.1 Conectar una Base de Datos con PDO

El primer argumento de la clase PDO es el DSN, Data SourceName, 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). Diferentes sistemas de bases de datos tienen distintos métodos para conectarse. La mayoría se conectan de forma parecida a como se conecta a MySQL:

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

DBH significa DatabaseHandle, y es el nombre de variable que se suele utilizar para el objeto PDO.

Para cerrar una conexión:

```
$dbh = null;
```

### 6.7.2 Excepciones y opciones con 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:

- `$dbh->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_SILENT);`
- `$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::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 un 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();  
}
```

### 6.7.3 Registrar Datos con PDO

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 nombres específicos o variables especiales.

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 lazy, 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 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");
$stmt = $dbh->prepare("INSERT INTO Clientes (nombre, ciudad) VALUES
(:nombre, :ciudad)");
$stmt->execute((array) $cliente);
};
```

#### 6.7.4 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 Clientes (nombre) VALUES
(:nombre)");
$nombre = "Morgan";

// Se enlazan los parámetros a las variables
$stmt->bindParam(':nombre', $nombre); // Se enlaza a la variable

// 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).

### 6.7.5 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_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:

```
// FETCH_ASSOC
$stmt = $dbh->prepare("SELECT * FROM Clientes");

// Especificamos el fetchmode 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 fetchmode cuando llamamos a fetch:
while($row = $stmt->fetch(PDO::FETCH_OBJ)){
    echo "Nombre: " . $row->nombre . "<br>";
    echo "Ciudad: " . $row->ciudad . "<br>";
}
```

Con `FETCH_BOUND` debemos emplear el método `bindColumn()`:

```
// Preparamos
$stmt = $dbh->prepare("SELECT nombre, ciudad FROM Clientes");

// Ejecutamos
$stmt->execute();

// bindColumn
$stmt->bindColumn(1, $nombre);
$stmt->bindColumn('ciudad', $ciudad);
while ($row = $stmt->fetch(PDO::FETCH_BOUND)) {
    $cliente = $nombre . ": " . $ciudad;
    echo $cliente . "<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++;
}
```

Finalmente, 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>";
}
```

### 6.7.6 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 escaparlos. `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.

Ya hemos visto cómo funcionan las sentencias preparadas con `prepare()` y `execute()`, vamos a ver un 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`.

### 6.7.7 Otras funciones de utilidad

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();
```

### 6.7.8 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->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    $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 ('Latrishawagar', '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 (PDOException $e){  
        echo "Ha habido algún error";  
        $dbh->rollback();  
    }
```

Enlaces y recursos.

<http://www.solvetic.com/tutoriales/article/1528-clase-para-gestionar-bases-de-datos-mysql-con-mysqli-y-php/>

<https://efunctions.wordpress.com/2011/12/13/uso-de-mysqli-en-php/>