



# UD5 – Acceso a base de datos

2º CFGS  
Desarrollo de Aplicaciones Web  
2022-23

# 1.- Acceso a bases de datos en PHP

Hoy en día la mayoría de aplicaciones web son **dinámicas**.

El "apellido" dinámicas hace referencia a que las páginas cambian dependiendo de diferentes factores:

- Momento en el que se accede.
- Estar registrado/logueado.
- Conexiones con otros usuarios de la aplicación.
- ...

Para poder mostrar información diferente a cada usuario lo más habitual es **acceder a una base de datos** la cual irá cambiando con el tiempo.

# 1.- Acceso a bases de datos en PHP

PHP soporta **más de 15 SGBS** (**S**istemas **G**estores de **B**ases de **D**atos).

Históricamente el acceso a una base de datos desde PHP requería el uso de **extensiones nativas específicas** para cada SGBD.

Esto significaba que si se iba a trabajar con una base de datos PostgreSQL se debía instalar dicha extensión en el servidor.

Cada extensión tenía unas funciones concretas y no existía compatibilidad entre extensiones por lo que si se cambiaba el SGBD había que rehacer la aplicación.

# 1.- Acceso a bases de datos en PHP

A partir de **PHP 5.1** se añadió al lenguaje una funcionalidad que permite acceder de un mismo modo a distintos SGBD haciendo uso de la POO.

Esta funcionalidad es **PDO (PHP Data Objects)** y gracias a ella se puede usar la misma sintaxis aunque se cambie el SGBD.

Elegir una extensión nativa y PDO dependerá de las necesidades de la aplicación:

- Las extensiones nativas ofrecen más potencia.
- Las extensiones nativas ofrecen en algunos casos más velocidad.
- PDO ofrece funciones comunes para el acceso a bases de datos.
- Con PDO se puede cambiar el SGBD en cualquier momento.

# 1.- Acceso a bases de datos en PHP

Se use una **extensión específica** o se use **PDO** se pueden realizar acciones como:

- Establecer conexiones a la base de datos.
- Ejecutar sentencias SQL.
- Obtener los registros devueltos por una sentencia SQL.
- Emplear transacciones.
- Ejecutar procedimientos almacenados.
- Gestionar los errores producidos durante la conexión a la base de datos.

# 1.- Acceso a bases de datos en PHP

Hoy en día muchas aplicaciones web siguen usando extensiones específicas para el SGBD.

Aún así, PDO ofrece mayor compatibilidad y está desarrollada de forma nativa en POO.

Por ello durante el curso se estudiará el uso de PDO como herramienta de acceso a bases de datos PHP.

---

# Debate

## Extensión nativa

## VS

## PDO

## 2.- MySQL – MariaDB

### MySQL/MariaDB

MySQL es un sistema gestor de bases de datos relacionales de código abierto.

También tiene una licencia comercial para desarrollar aplicaciones de código propietario.

Cuando Sun Microsystems compró MySQL se creó el forl MariaDB para asegurar su continuidad como código abierto.

El conjunto MySQL/MariaDB es el más usado con aplicaciones web PHP.

La extensión nativa para MySQL/MaríaDB se llama [MySQLi](#).



# 3.- phpMyAdmin

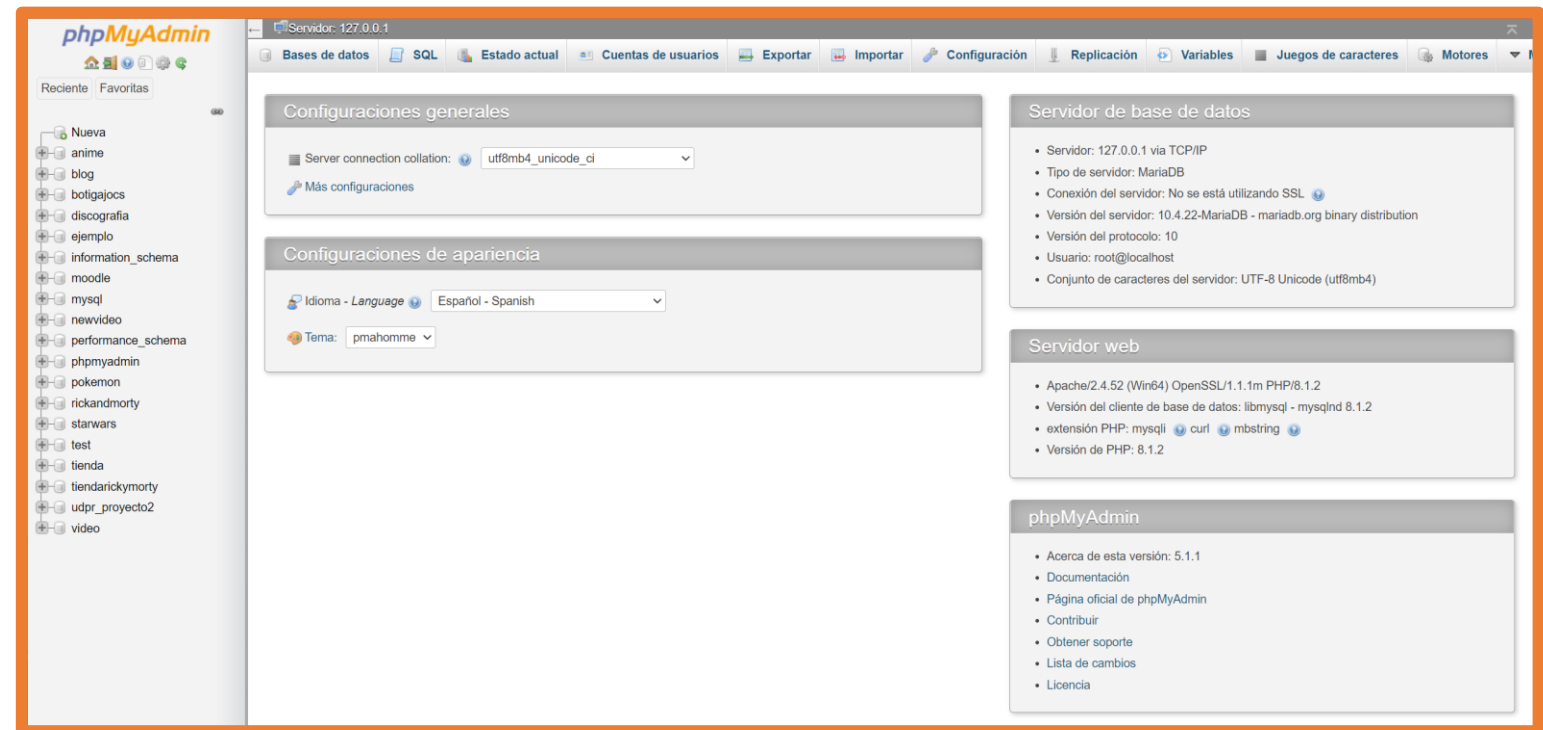
**phpMyAdmin** es una herramienta web que permite la administración del SGBD MySQL/MariaDB.

Se incluye en XAMPP y similares.

<http://localhost/phpmyadmin>

<http://localhost:8080/phpmyadmin>

Veámoslo →



## 4.- Permisos sobre la base de datos

Los datos son muy importantes en una aplicación web.

En las bases de datos de las aplicaciones web se almacenan todo tipos de datos, desde **datos que son públicos** hasta **datos privados de los usuarios**.

Por ello es importante que las bases de datos estén protegidas y que solo puedan acceder a ellas los usuarios con privilegios.

Como en todos los SGBD, en MySQL/MariaDB también se pueden configurar los permisos de los usuarios a diferentes niveles:

- sobre toda la base de datos
- sobre algunas tablas
- algunas acciones únicamente
- ...

## 4.- Permisos sobre la base de datos

Una manera de proteger los datos es crear diferentes usuarios para la base de datos:

- Usuario con permisos SELECT sobre los datos que se muestran muy habitualmente. Por ejemplo, la tabla de entradas de un blog.
- Usuario con permisos INSERT/UPDATE para los scripts que insertan o modifican datos de la base de datos.
- Usuario con permisos SELECT sobre la tabla de usuarios para hacer las comprobaciones de autenticación.
- ...

## 5.- Juego de caracteres

En el mundo existen muchos idiomas diferentes con sus letras y símbolos particulares.

Además cuando se almacena la información en un ordenador se puede optar por codificaciones diferentes.

Por ello es importante seleccionar bien el tipo de codificación que se usará en la base de datos y sus tablas para asegurarse que los símbolos específicos de cada idioma se muestren correctamente.

A la codificación elegida se le llama **juego de caracteres**.

## 5.- Juego de caracteres

Además de todos los caracteres de los idiomas del mundo, desde hace unos años se usan los **emoji** gracias a las aplicaciones de mensajería instantánea y redes sociales.



Para poder representar toda la información que se usa actualmente es importante usar el juego de caracteres **UTF-8**.

Tanto MySQL como MariaDB ofrecen la codificación utf8 pero esta codificación es antigua y no corresponde con UTF-8.

Se recomienda usar la codificación moderna **utf8mb4\_unicode\_ci** que sí corresponde a UTF-8.

# Práctica

## **Actividad 1:** Visita turística.

## 6.- PHP Data Objects → PDO

PDO es una clase que ofrece un conjunto de propiedades y métodos para realizar conexiones a la base de datos.

Un objeto de la clase PDO representa una conexión entre PHP y un servidor de bases de datos.

PDO ofrece una capa de abstracción de **acceso a datos** que permite usar los mismos mecanismos para realizar consultas independientemente de la base de datos usada.

## 6.- PHP Data Objects → PDO

### Estableciendo la conexión

Para establecer la conexión a la base de datos se debe instanciar un objeto de la clase PDO usando su constructor.

El constructor admite los siguientes parámetros:

- Origen de datos (**DSN** – **Data Source Name**): cadena de texto que indica el controlador a usar y parámetros específicos de dicho controlador.
- Usuario con permisos para las acciones que se van a realizar en la BBDD.
- Contraseña del usuario.
- Array asociativo con opciones de conexión.

```
$conexion = new PDO('mysql:host=localhost;dbname=starwars', 'Han', 'Solo');
```



## 6.- PHP Data Objects → PDO

### Estableciendo la conexión

El único **argumento obligatorio** en el constructor de PDO es la **cadena DSN**.

Para usar el controlador MySQL/MariaDB los parámetros de la cadena en orden de uso son:

- host: nombre o IP del servidor.
- port: puerto de escucha en el servidor.
- dbname: nombre de la base de datos.
- unix\_socket: socket Unix de MySQL (no debe ponerse si se usa host o port).
- charset: conjunto de caracteres.



```
$conexion = new PDO('mysql:host=localhost;dbname=starwars', 'Han', 'Solo');
```

## 6.- PHP Data Objects → PDO

### Estableciendo la conexión

Algunos ejemplos de DSN:

```
mysql:host=localhost;dbname=testdb
```

```
mysql:host=localhost;port=3307;dbname=testdb
```

```
mysql:unix_socket=/tmp/mysql.sock;dbname=testdb
```

## 6.- PHP Data Objects → PDO

### Estableciendo la conexión

Como parte de la aplicación, durante la conexión a la base de datos también es importante controlar el juego de caracteres.

Para indicar correctamente el juego de caracteres en la conexión se debe añadir en el array de opciones:

```
$dsn = 'mysql:host=localhost;dbname=starwars';  
$opciones = array(PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8');  
$conexion = new PDO($dsn, 'Han', 'Solo', $opciones);
```

# Importante

Es importante destacar que el usuario de la base de datos y su contraseña son independientes a los posibles usuarios de la aplicación web.

- Usuario y contraseña de la base de datos:  
Se configura en el servidor y se escribe directamente en código PHP.  
Estos datos los deben conocer los programadores de la aplicación web.
- Usuarios de la aplicación web y sus contraseñas :  
Se almacenan en la base de datos.  
La contraseña de un usuario solo la conoce ese usuario.  
La contraseña se debe de almacenar encriptada.

## 7.- Gestión de errores al trabajar con PDO

El constructor de PDO lanzará una **excepción** en caso de error:

```
$dsn = 'mysql:host=localhost;dbname=starwars';  
$user = 'Han';  
$pass = 'Solo';  
$opciones = array(PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8');  
try {  
    $conexion = new PDO($dsn, $user, $pass, $opciones);  
} catch (PDOException $e) {  
    echo 'Fallo durante la conexión: ' . $e->getMessage();  
}
```

En una aplicación web con acceso a base de datos es importante el tratamiento de errores ya que de ello dependerá qué se visualiza en cada página web.

## 8.- Objeto instancia de PDO

Una vez establecida la conexión, se dispone de un objeto mediante el cuál se podrán realizar sobre la base de datos las acciones que se requieran.

Los métodos disponibles para un objeto PDO son:

- [PDO::beginTransaction](#) — Inicia una transacción
- [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

## 8.- Objeto instancia de PDO

Por ejemplo:

Con el método **getAttribute** se puede obtener información del estado de la conexión

```
$version = $conexion->getAttribute(PDO::ATTR_SERVER_VERSION);
```

Con el método **setAttribute** se pueden modificar opciones de la conexión

```
$conexion->setAttribute(PDO::ATTR_CASE, PDO::CASE_UPPER);
```

La instrucción anterior establece que el nombre de los campos se reciban en mayúscula.

## 9.- Consultas que SÍ devuelven datos → SELECT

Para ejecutar consultas **SELECT** se usa el método **query**.

```
$resultado = $conexion->query('SELECT name, birth_year, gender FROM characters;');
```

La variable **\$resultado** contiene una instancia de la clase **PDOStatement** con la información obtenida mediante la consulta.



## 9.- Consultas que SÍ devuelven datos → SELECT

### PDOStatement

Esta clase dispone de varios métodos para trabajar con los resultados obtenidos.

- **rowCount**: devuelve el número de filas obtenidas con la consulta.
- **fetch**: obtiene un array con la siguiente fila de los resultados de la consulta.
- **fetchObject**: obtiene un objeto con la siguiente fila de los resultados de la consulta. Las propiedades del objeto serán los campos seleccionados en la consulta.
- **fetchAll**: obtiene un array multidimensional que contiene todas las filas de los resultados obtenidos con la consulta.

## 9.- Consultas que SÍ devuelven datos → SELECT

Ejemplo de uso de **rowCount**:

```
$resultado = $conexion->query('SELECT name, birth_year FROM characters;');  
  
echo 'Se han obtenido '. $resultado->rowCount() .' personajes.';
```

## 9.- Consultas que SÍ devuelven datos → SELECT

**fetch**: obtiene un array con la siguiente fila de los resultados de la consulta.

Acepta un parámetro que indica el modo con el que se obtiene la fila, los modos más usados son los siguientes.

- **PDO::FETCH\_BOTH** → obtiene un array mixto: numérico y asociativo. (defecto).
- **PDO::FETCH\_ASSOC** → obtiene un array asociativo.
- **PDO::FETCH\_NUM** → obtiene un array numérico.
- **PDO::FETCH\_OBJECT** → obtiene un objeto (igual que el método **fetchObject**).

## 9.- Consultas que SÍ devuelven datos → SELECT

Ejemplo de uso de **fetch**:

```
$resultado = $conexion->query('SELECT name, birth_year FROM characters;');

while ($registro = $resultado->fetch()) {
    echo 'Personaje: '. $registro['name'];
    // Si el campo es null (vacío en la base de datos)
    if($registro['birth_year'] != null) {
        echo ' ('. $registro['birth_year'] .')';
    }
    echo '<br>';
}
```

## 9.- Consultas que SÍ devuelven datos → SELECT

Ejemplo de uso de **fetch**:

```
$resultado = $conexion->query('SELECT name, birth_year FROM characters;');

while ($registro = $resultado->fetch()) {
    echo 'Personaje: '. $registro['name'];
    // Si el campo es null (vacío en la base de datos)
    if($registro['birth_year'] != null) {
        echo ' ('. $registro['birth_year'] .')';
    }
    echo '<br>';
}
```

**fetch** con opciones:

```
$registro = $resultado->fetch(PDO::FETCH_ASSOC);
$registro = $resultado->fetch(PDO::FETCH_NUM);
$personaje = $resultado->fetch(PDO::FETCH_OBJ);
```

## 9.- Consultas que SÍ devuelven datos → SELECT

Ejemplo de uso de **fetchObject**:

```
$resultado = $conexion->query('SELECT name, birth_year FROM characters;');

while ($registro = $resultado->fetchObject()) {
    echo 'Personaje: '. $registro->name;
    // Si el campo es null (vacío en la base de datos)
    if($registro->birth_year != null) {
        echo ' ('. $registro->birth_year .')';
    }
    echo '<br>';
}
```

## 9.- Consultas que SÍ devuelven datos → SELECT

Ejemplo de uso de **fetchAll**:

```
$resultado = $conexion->query('SELECT name, birth_year FROM characters;');

foreach ($resultado->fetchAll() as $registro) {
    echo 'Personaje: '. $registro['name'];
    // Si el campo es null (vacío en la base de datos)
    if($registro['birth_year'] != null) {
        echo ' ('. $registro['birth_year'] .')';
    }
    echo '<br>';
}
```

## 10.- Cerrar conexión a la base de datos

Al acabar la ejecución de un script PHP se liberan todos los recursos usados por el script incluidas las conexiones a bases de datos.

Aún así es importante cerrar las conexiones a la base de datos para ahorrar recursos.

Para ello se deben eliminar las instancias de PDO y PDOStatement.

```
// Se elimina el objeto PDOStatement  
unset($result);  
// Se elimina el objeto PDO  
unset($conexion);
```



## 11.- Consultas preparadas

Las consultas preparadas son consultas que se almacenan en el servidor de base de datos mientras la conexión a la misma sigue activa.

De esta manera la ejecución de la consulta es más rápida cuando se ejecuta la misma consulta con diferentes valores, por ejemplo al insertar de golpe varios registros en la base de datos.

No siempre son recomendables ya que **pueden sobrecargar al servidor.**

## 11.- Consultas preparadas

Una característica importante de las consultas preparadas es que ayudan al programador a **prevenir ataques [SQL injection](#)**.

Los ataques **SQL injection** consisten en intentar ejecutar consultas dañinas o que permiten obtener datos sensibles de una base de datos usando los formularios de la aplicación web.

En resumen, el comportamiento de las consultas preparadas usadas de esta manera consiste en decidir qué parte de la consulta es '**código ejecutable**' y qué parte del texto no lo es.

# 11.- Consultas preparadas

Así, se programará la aplicación web de la siguiente manera:

- Consulta normal: en la consulta el usuario no introduce datos.  
Por ejemplo: al mostrar una lista de noticias, al mostrar una lista de publicaciones, al mostrar una lista de entradas... si el usuario no aplica filtros.
- Consulta preparada: en la consulta el usuario SÍ introduce datos.  
Por ejemplo: en el formulario de registro, en el formulario de login, al guardar un comentario, al realizar una búsqueda...

# 11.- Consultas preparadas

Para crear consultas preparadas en PHP se utilizan los métodos de PDO:

- **prepare:** se indica la consulta con parámetros, que se va a ejecutar.
- **bindParam:** se indica el valor de un parámetro.
- **execute:** ejecuta la consulta preparada con los valores asignados.  
Devuelve un objeto PDOStatement

Los parámetros se pueden indicar por **posición** o por **nombre**, estas dos técnicas no se pueden mezclar en una misma consulta preparada.

Los parámetros serán los valores que ha introducido el usuario y de esta manera es SGBD sabe que los parámetros no son consultas por sí mismos.

## 12.- Consultas preparadas → SELECT

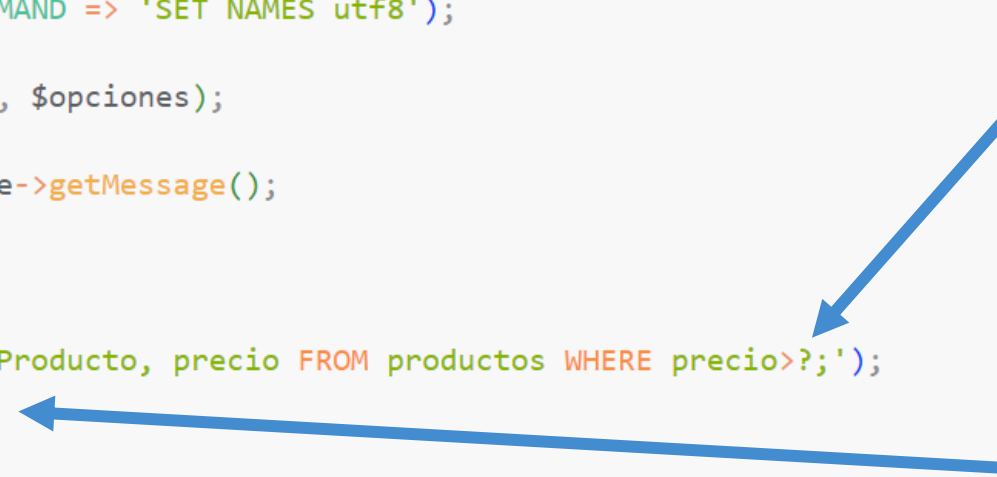
Ejemplo de uso de **consulta preparada SELECT**:

```
$dsn = 'mysql:host=localhost;dbname=tienda';
$user = 'Apu';
$pass = 'badulaque';
$opciones = array(PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8');
try {
    $conexion = new PDO($dsn, $user, $pass, $opciones);
} catch (PDOException $e) {
    echo 'Fallo durante la conexión: ' . $e->getMessage();
}

// Opción 1
$consulta = $conexion->prepare('SELECT nomProducto, precio FROM productos WHERE precio>?;');
$consulta->bindParam(1, $_POST['precio']);

// Opción 2
// $consulta = $conexion->prepare('SELECT nomProducto, precio FROM productos WHERE precio>:prec;');
// $consulta->bindParam(':prec', $_POST['precio']);

$consulta->execute(); // Devuelve true/false según se ejecute con éxito o no
foreach ($consulta->fetchAll(PDO::FETCH_ASSOC) as $producto) {
    echo $producto['nomProducto'].' (pvp: ' . $producto['precio'] . ' €)<br>';
}
```



## 12.- Consultas preparadas → SELECT

En la consulta se utiliza una variable cuyo valor lo ha introducido el usuario en un formulario.

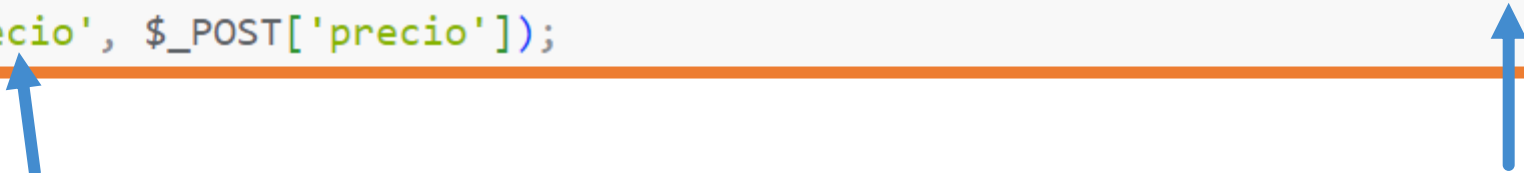
Usando parámetros por posición:

```
$consulta = $conexion->prepare('SELECT nomProducto, precio FROM productos WHERE precio>?;');  
$consulta->bindParam(1, $_POST['precio']);
```



Usando parámetros por nombre:

```
$consulta = $conexion->prepare('SELECT nomProducto, precio FROM productos WHERE precio>:precio;');  
$consulta->bindParam(':precio', $_POST['precio']);
```

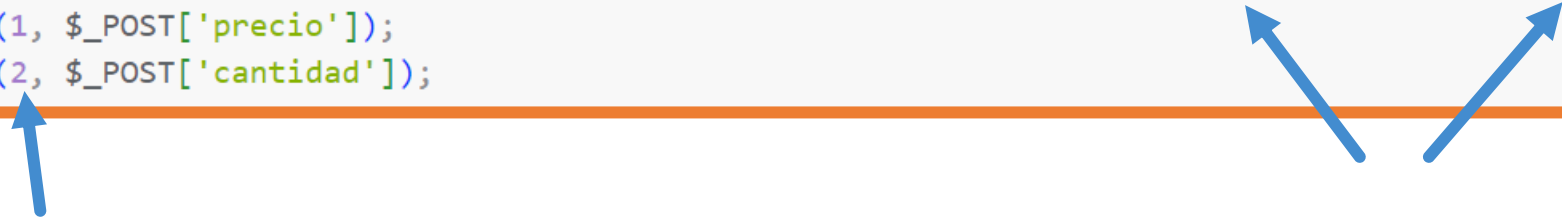


## 12.- Consultas preparadas → SELECT

Consultas preparadas con varios parámetros

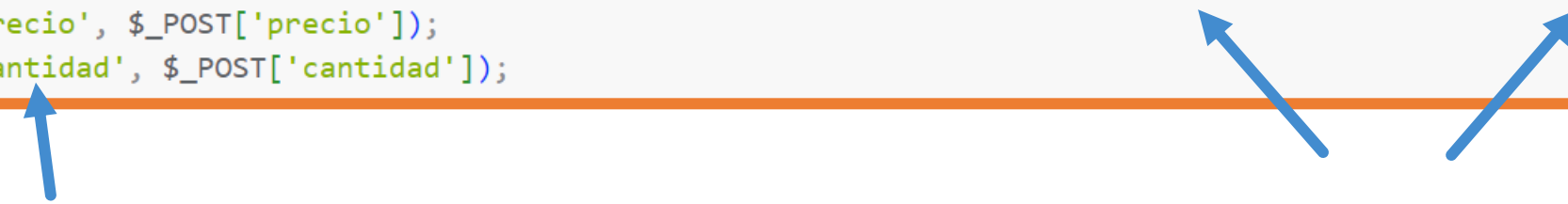
Usando parámetros por posición:

```
$consulta = $conexion->prepare('SELECT nomProducto, precio FROM productos WHERE precio>? AND existencias<?;');  
$consulta->bindParam(1, $_POST['precio']);  
$consulta->bindParam(2, $_POST['cantidad']);
```



Usando parámetros por nombre:

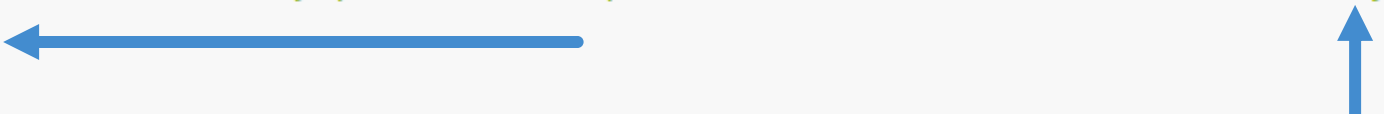
```
$consulta = $conexion->prepare('SELECT nomProducto, precio FROM productos WHERE precio>:precio AND existencias<:cantidad;');  
$consulta->bindParam(':precio', $_POST['precio']);  
$consulta->bindParam(':cantidad', $_POST['cantidad']);
```



## 12.- Consultas preparadas → SELECT

Si en la consulta preparada se usa LIKE con comodines se debe realizar de la siguiente manera:

```
$consulta = $conexion->prepare('SELECT nomProducto, precio FROM productos WHERE nomProducto LIKE ?;');  
$nombre = '%' . $_POST['nombre'] . '%';  
$consulta->bindParam(1, $nombre);
```

A diagram illustrating the binding process. A blue arrow points from the variable `$nombre` in the `bindParam` method call to the question mark placeholder `?` in the SQL statement. Another blue arrow points from the `$_POST['nombre']` part of the `$nombre` variable assignment to the same question mark placeholder, showing how the user input is used to construct the value for the placeholder.

Se debe crear una variable extra ya que en el método **bindParam** no acepta literales.



# Práctica

## **Actividad 2:**

Dungeons & Dragons: partida de rol.

## **Actividad 3:**

Discografía.

## 13.- Consultas que NO devuelven datos

### INSERT, DELETE, UPDATE

En PDO se debe tener en cuenta si la consulta devuelve datos o no.

Para ejecutar consultas **INSERT**, **DELETE** y **UPDATE** se usa el método **exec**.

Este método devuelve el número de registros afectados con la consulta.

```
$registros = $conexion->exec('DELETE FROM productos WHERE existencias>200;');  
echo 'Se han eliminado '. $registros . ' productos.';
```

La variable **\$registros** contiene la cantidad de registros eliminados.

## 14.- Consultas preparadas → INSERT, DELETE, UPDATE

En las consultas **INSERT**, **DELETE** y **UPDATE** es muy importante el uso de consultas preparadas porque generalmente es donde intervienen datos introducidos por el usuario en un formulario.

Usando parámetros por **posición**:

```
$consulta = $conexion->prepare('INSERT INTO productos
                                (NomProducto, Proveedor, Categoria, Precio, Existencias)
                                VALUES (?, ?, ?, ?, ?);');

$consulta->bindParam(1, $_POST['nombre']);
$consulta->bindParam(2, $_POST['proveedor']);
$consulta->bindParam(3, $_POST['categoria']);
$consulta->bindParam(4, $_POST['precio']);
$consulta->bindParam(5, $_POST['existencias']);

$consulta->execute();
```

## 14.- Consultas preparadas → INSERT, DELETE, UPDATE

En las consultas **INSERT**, **DELETE** y **UPDATE** es muy importante el uso de consultas preparadas porque generalmente es donde intervienen datos introducidos por el usuario en un formulario.

Usando parámetros por **nombre**:

```
$consulta = $conexion->prepare('INSERT INTO productos
                                (NomProducto, Proveedor, Categoria, Precio, Existencias)
                                VALUES (:nombre, :proveedor, :categoria, :precio, :existencias);');
$consulta->bindParam(':nombre', $_POST['nombre']);
$consulta->bindParam(':proveedor', $_POST['proveedor']);
$consulta->bindParam(':categoria', $_POST['categoria']);
$consulta->bindParam(':precio', $_POST['precio']);
$consulta->bindParam(':existencias', $_POST['existencias']);
```

## 14.- Consultas preparadas → INSERT, DELETE, UPDATE

Como ya se ha comentado anteriormente, el uso de consultas preparadas también ayuda a la hora de ejecutar una consulta varias veces.

```
$consulta = $conexion->prepare('INSERT INTO productos
                                (NomProducto, Proveedor, Categoria, Precio, Existencias)
                                VALUES (:nombre, :proveedor, :categoria, :precio, :existencias);');

$consulta->bindParam(':nombre', $nombre);
$consulta->bindParam(':proveedor', $proveedor);
$consulta->bindParam(':categoria', $categoria);
$consulta->bindParam(':precio', $precio);
$consulta->bindParam(':existencias', $existencias);

$nombre = 'Nintendo Switch';
$proveedor = 12;
$categoria = 2;
$precio = 299;
$existencias = 58;
$consulta->execute();

$nombre = 'Sony PS5';
$proveedor = 12;
$categoria = 2;
$precio = 549;
$existencias = 12;
$consulta->execute();
```

## 15.- Transacciones

Una transacción es un conjunto de consultas que se deben realizar en bloque.

Si alguna de las consultas de la transacción falla, entonces se debe volver al estado inicial deshaciendo aquellas consultas de la transacción que sí se habían ejecutado.

**O todas o ninguna.**

MySQL/MaríaDB por defecto ya configura las tablas de la base de datos para que se puedan realizar transacciones al configurar el motor de almacenamiento como **InnoDB**.

Si el motor de almacenamiento es **MyISAM** no se podrán realizar transacciones.

## 15.- Transacciones

En MySQL/MaríaDB está configurada por defecto la opción **autocommit**.

Con **autocommit** las transacciones están desactivadas lo que significa que cada consulta que se ejecute se convierte en una transacción.

De esta manera si la consulta falla no se aplica ningún cambio a la base de datos.

Si se quiere iniciar una transacción se debe desactivar el modo **autocommit** realizando una llamada al método **beginTransaction** sobre el **objeto PDO** de la conexión.

Este método devuelve true|false para indicar si se ha cambiado o no el modo.

```
$conexion->beginTransaction();
```

## 15.- Transacciones

Una vez desactivado el autocommit se deben ejecutar las consultas.

Si todas las consultas se ejecutan correctamente se deberá ejecutar un commit:

```
$conexion->commit();
```

Si una consulta falla se debe ejecutar un rollback y no seguir ejecutando consultas:

```
$conexion->rollback();
```

Al ejecutar **commit** o **rollback** MySQL/MariaDB vuelven al modo **autocommit**.



# 15.- Transacciones

En una transacción se pueden usar cualquier tipo de consulta:

- SELECT
- INSERT
- UPDATE
- DELETE
- Normal
- Preparada

## 15.- Transacciones

### **Inconveniente:**

Desde PHP no existe un método automático para saber si alguna consulta ha fallado.

Debe ser el programador el que lo compruebe y realice el commit o el rollback según funcionen todas las consultas o no.

A continuación, se muestra un ejemplo para realizar esta comprobación.

# 15.- Transacciones

Primera solución:

```
$conexion->beginTransaction();  
if($conexion->exec($consulta1) == 0)  
    $ok = false;  
if($conexion->exec($consulta2) == 0)  
    $ok = false;  
// ...  
if($conexion->exec($consultaN) == 0)  
    $ok = false;  
  
if ($ok)  
    $conexion->commit();    // Si todo fue bien, confirma los cambios  
else  
    $conexion->rollback(); // Si alguna consulta falló se revierten los cambios
```

# 15.- Transacciones

## Segunda solución: usando excepciones

```
$conexion->beginTransaction();
try {
    if($conexion->exec($consulta1) == 0)
        throw new Exception('Error update', 1);
    if($conexion->exec($consulta2) == 0)
        throw new Exception('Error update', 1);

    $conexion->commit();
    echo 'Consultas ejecutadas correctamente.';
}
catch (Exception $e) {
    $conexion->rollback();
    echo 'Se ha producido un error.';
}
```

# Práctica

## **Actividad 4:**

Aumentando la discografía.