Conexiones a Bases de Datos MySQL desde PHP

Introducción

En el desarrollo web moderno, la comunicación entre nuestras aplicaciones y bases de datos es fundamental. Casi todas las aplicaciones web necesitan almacenar y recuperar información de manera persistente, y para esto utilizamos las bases de datos.

MySQL es uno de los sistemas de gestión de bases de datos relacionales más populares, especialmente en combinación con PHP. Juntos forman parte del stack LAMP (Linux, Apache, MySQL, PHP), que ha sido la columna vertebral de muchos sitios web durante años.

En esta guía, aprenderemos cómo conectar nuestras aplicaciones PHP con bases de datos MySQL, desde los conceptos básicos hasta técnicas más avanzadas.

¿Qué necesitamos antes de empezar?

Antes de conectarnos a una base de datos MySQL desde PHP, necesitamos tener instalado:

1. **PHP**: Versión 7.0 o superior recomendada
2. **MySQL**: Servidor de base de datos instalado y funcionando
3. **Extensión mysqli o PDO**: Generalmente vienen activadas por defecto en PHP

También necesitaremos conocer los siguientes datos para realizar la conexión:

- Nombre del servidor (host) - generalmente "localhost" si está en la misma máquina
- Nombre de usuario de MySQL
- Contraseña de MySQL
- Nombre de la base de datos

Métodos para conectar PHP con MySQL

Existen dos formas principales de conectar PHP con MySQL:

1. **MySQLi (MySQL Improved)**: Una extensión específica para MySQL
2. **PDO (PHP Data Objects)**: Una interfaz que permite conectar con diferentes tipos de bases de datos

Veremos ambos métodos, pero nos centraremos más en PDO ya que es más versátil y seguro.

Conexión básica usando MySQLi

Veamos primero cómo realizar una conexión simple usando MySQLi:

```php
<?php
// Datos de conexión
$servidor = "localhost";
$usuario = "usuario_mysql";
```

```php
$password = "contraseña_mysql";
$baseDatos = "nombre_base_datos";

// Crear conexión
$conexion = new mysqli($servidor, $usuario, $password, $baseDatos);

// Verificar si hay errores en la conexión
if ($conexion->connect_error) {
    die("La conexión falló: " . $conexion->connect_error);
}

echo "¡Conexión exitosa!";

// Cerrar la conexión cuando hayamos terminado
$conexion->close();
?>
```

Cuando ejecutamos este código, pueden ocurrir dos cosas:

1. Si la conexión es exitosa, veremos el mensaje "¡Conexión exitosa!"
2. Si hay algún error (por ejemplo, datos incorrectos), veremos un mensaje que indica cuál fue el problema

Explicación detallada:

- **die()**: Esta función detiene la ejecución del script y muestra un mensaje. Es útil para detener todo si no podemos conectarnos a la base de datos.
- **connect_error**: Contiene información sobre el error de conexión, si ocurrió alguno.
- **close()**: Siempre debemos cerrar la conexión cuando terminemos de usarla para liberar recursos.

Conexión usando PDO (recomendada)

PDO ofrece una interfaz más consistente y segura para trabajar con bases de datos. Veamos cómo crear una conexión:

```php
<?php
// Datos de conexión
$servidor = "localhost";
$usuario = "usuario_mysql";
$password = "contraseña_mysql";
$baseDatos = "nombre_base_datos";

try {
    // Crear conexión PDO
    $conexion = new PDO(
        "mysql:host=$servidor;dbname=$baseDatos",
        $usuario,
        $password
    );
```

```
    // Configurar PDO para que lance excepciones en caso de errores
    $conexion->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

    echo "¡Conexión exitosa usando PDO!";
} catch(PDOException $e) {
    // Capturar y mostrar cualquier error
    echo "Error de conexión: " . $e->getMessage();
}

// La conexión se cierra automáticamente al finalizar el script
// pero podemos cerrarla explícitamente si queremos
$conexion = null;
?>
```

Explicación detallada:

- **try-catch**: Esta estructura nos permite "intentar" un bloque de código y "capturar" cualquier error que ocurra.
- **PDO::ATTR_ERRMODE**: Configura cómo PDO maneja los errores. PDO::ERRMODE_EXCEPTION hace que lance excepciones que podemos capturar.
- **$conexion = null**: Forma de cerrar explícitamente la conexión en PDO.

Ventajas de usar PDO sobre MySQLi

1. **Soporte para múltiples bases de datos**: PDO puede trabajar con MySQL, PostgreSQL, SQLite y muchas otras.
2. **Seguridad mejorada**: PDO facilita el uso de consultas preparadas, que protegen contra ataques de inyección SQL.
3. **Manejo de errores más robusto**: El sistema de excepciones de PDO es más completo.
4. **Código más limpio**: PDO generalmente resulta en código más legible y mantenible.

Creando un archivo de configuración para la conexión

Es una buena práctica separar los datos de conexión en un archivo aparte para:

1. Reutilizar el código de conexión en diferentes partes de nuestra aplicación
2. Mejorar la seguridad al no tener credenciales esparcidas por todo el código
3. Facilitar cambios en los parámetros de conexión

Veamos cómo implementarlo:

1. Primero, creamos un archivo llamado `config.php`:

```
<?php
// config.php
define('DB_HOST', 'localhost');
define('DB_USER', 'usuario_mysql');
define('DB_PASS', 'contraseña_mysql');
```

```
define('DB_NAME', 'nombre_base_datos');
?>
```

2. Luego, creamos un archivo `conexion.php` que use esta configuración:

```php
<?php
// conexion.php
require_once 'config.php';

function conectar() {
    try {
        $conexion = new PDO(
            "mysql:host=" . DB_HOST . ";dbname=" . DB_NAME,
            DB_USER,
            DB_PASS
        );
        $conexion->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        return $conexion;
    } catch(PDOException $e) {
        echo "Error de conexión: " . $e->getMessage();
        return null;
    }
}
?>
```

3. Finalmente, en cualquier archivo donde necesitemos la conexión:

```php
<?php
// archivo_que_usa_la_bd.php
require_once 'conexion.php';

$conexion = conectar();

if ($conexion) {
    echo "Conectado a la base de datos correctamente";
    // Usar $conexion para realizar operaciones con la base de datos

    // Cerrar conexión
    $conexion = null;
}
?>
```

Realizando operaciones básicas con la base de datos

Una vez establecida la conexión, podemos realizar operaciones básicas en la base de datos. Veamos algunos ejemplos:

1. Consultar datos (SELECT)

```php
<?php
require_once 'conexion.php';
$conexion = conectar();

try {
    // Preparar consulta SELECT
    $consulta = $conexion->prepare("SELECT id, nombre, email FROM usuarios");

    // Ejecutar consulta
    $consulta->execute();

    // Obtener resultados
    $resultados = $consulta->fetchAll(PDO::FETCH_ASSOC);

    // Mostrar resultados
    echo "<h2>Lista de Usuarios</h2>";
    echo "<ul>";
    foreach ($resultados as $fila) {
        echo "<li>Usuario: " . $fila['nombre'] . " - Email: " . $fila['email']
. "</li>";
    }
    echo "</ul>";

} catch(PDOException $e) {
    echo "Error en la consulta: " . $e->getMessage();
}

$conexion = null;
?>
```

Si la tabla `usuarios` contiene datos como:

id	nombre	email
1	María	maria@ejemplo.com
2	Juan	juan@ejemplo.com

El resultado sería:

```
Lista de Usuarios
• Usuario: María - Email: maria@ejemplo.com
• Usuario: Juan - Email: juan@ejemplo.com
```

2. Insertar datos (INSERT)

```php
<?php
require_once 'conexion.php';
$conexion = conectar();

try {
    // Datos a insertar
    $nombre = "Carlos";
    $email = "carlos@ejemplo.com";

    // Preparar consulta INSERT
    $consulta = $conexion->prepare("INSERT INTO usuarios (nombre, email) VALUES
(:nombre, :email)");

    // Vincular parámetros
    $consulta->bindParam(':nombre', $nombre);
    $consulta->bindParam(':email', $email);

    // Ejecutar consulta
    $consulta->execute();

    echo "Usuario agregado correctamente. ID: " . $conexion->lastInsertId();

} catch(PDOException $e) {
    echo "Error al insertar: " . $e->getMessage();
}

$conexion = null;
?>
```

Al ejecutarse, este código insertará un nuevo usuario y mostrará el ID asignado automáticamente.

3. Actualizar datos (UPDATE)

```php
<?php
require_once 'conexion.php';
$conexion = conectar();

try {
    // Datos a actualizar
    $id = 2;
    $nuevoNombre = "Juan Carlos";

    // Preparar consulta UPDATE
    $consulta = $conexion->prepare("UPDATE usuarios SET nombre = :nombre WHERE
id = :id");

    // Vincular parámetros
    $consulta->bindParam(':nombre', $nuevoNombre);
    $consulta->bindParam(':id', $id);
```

```
    // Ejecutar consulta
    $consulta->execute();

    echo "Usuario actualizado correctamente. Filas afectadas: " . $consulta-
>rowCount();

} catch(PDOException $e) {
    echo "Error al actualizar: " . $e->getMessage();
}

$conexion = null;
?>
```

4. Eliminar datos (DELETE)

```
<?php
require_once 'conexion.php';
$conexion = conectar();

try {
    // ID del usuario a eliminar
    $id = 1;

    // Preparar consulta DELETE
    $consulta = $conexion->prepare("DELETE FROM usuarios WHERE id = :id");

    // Vincular parámetro
    $consulta->bindParam(':id', $id);

    // Ejecutar consulta
    $consulta->execute();

    echo "Usuario eliminado correctamente. Filas afectadas: " . $consulta-
>rowCount();

} catch(PDOException $e) {
    echo "Error al eliminar: " . $e->getMessage();
}

$conexion = null;
?>
```

Consultas preparadas y prevención de inyección SQL

Las consultas preparadas son una forma segura de ejecutar operaciones en la base de datos, ya que previenen ataques de inyección SQL. Veamos cómo funcionan:

1. **Preparar**: Creamos la consulta con marcadores de posición (?) o nombres (:nombre)
2. **Vincular**: Asociamos valores a esos marcadores

3. **Ejecutar**: Realizamos la consulta con los valores vinculados

```
<?php
require_once 'conexion.php';
$conexion = conectar();

// Datos de búsqueda (potencialmente inseguros)
$busqueda = "O'Reilly"; // Nombre con apóstrofe que podría causar problemas

try {
    // Forma INSEGURA (no usar esto):
    // $consulta = $conexion->query("SELECT * FROM usuarios WHERE nombre =
'$busqueda'");

    // Forma SEGURA usando consultas preparadas:
    $consulta = $conexion->prepare("SELECT * FROM usuarios WHERE nombre =
:busqueda");
    $consulta->bindParam(':busqueda', $busqueda);
    $consulta->execute();

    $resultados = $consulta->fetchAll(PDO::FETCH_ASSOC);

    foreach ($resultados as $fila) {
        echo "ID: " . $fila['id'] . ", Nombre: " . $fila['nombre'] . "<br>";
    }

} catch(PDOException $e) {
    echo "Error: " . $e->getMessage();
}

$conexion = null;
?>
```

¿Por qué son importantes las consultas preparadas?

Imagina que alguien introduce lo siguiente en un campo de búsqueda:

```
' OR '1'='1
```

En una consulta sin preparar, tendríamos:

```
SELECT * FROM usuarios WHERE nombre = '' OR '1'='1'
```

Esta consulta devolvería TODOS los usuarios porque `'1'='1'` siempre es verdadero. En cambio, con consultas preparadas, el valor se trata como texto plano y no modifica la estructura de la consulta.

Transacciones en MySQL con PHP

Las transacciones permiten ejecutar múltiples consultas como una sola unidad. Si alguna falla, todas se revierten (rollback). Esto es útil para operaciones que deben completarse en su totalidad o no realizarse.

Por ejemplo, al transferir dinero entre cuentas, necesitamos:

1. Restar dinero de una cuenta
2. Añadir dinero a otra cuenta

Ambas operaciones deben completarse correctamente o ninguna debería realizarse.

```php
<?php
require_once 'conexion.php';
$conexion = conectar();

try {
    // Iniciar transacción
    $conexion->beginTransaction();

    // Consulta 1: Restar saldo de la cuenta origen
    $consultaOrigen = $conexion->prepare("UPDATE cuentas SET saldo = saldo -
:monto WHERE id = :idOrigen");
    $consultaOrigen->bindParam(':monto', $monto);
    $consultaOrigen->bindParam(':idOrigen', $idOrigen);
    $consultaOrigen->execute();

    // Consulta 2: Sumar saldo a la cuenta destino
    $consultaDestino = $conexion->prepare("UPDATE cuentas SET saldo = saldo +
:monto WHERE id = :idDestino");
    $consultaDestino->bindParam(':monto', $monto);
    $consultaDestino->bindParam(':idDestino', $idDestino);
    $consultaDestino->execute();

    // Si todo salió bien, confirmar los cambios
    $conexion->commit();
    echo "Transferencia completada correctamente";

} catch(PDOException $e) {
    // Si hubo error, deshacer cambios
    $conexion->rollBack();
    echo "Error en la transferencia: " . $e->getMessage();
}

$conexion = null;
?>
```

Gestión de errores y buenas prácticas

Para desarrollar aplicaciones robustas, es importante manejar adecuadamente los errores y seguir buenas prácticas:

1. Usar bloques try-catch para capturar errores

```
try {
    // Código que puede causar errores
} catch(PDOException $e) {
    // Manejar el error de forma elegante

    // Para desarrollo (mostrar detalles):
    echo "Error: " . $e->getMessage();

    // Para producción (mensaje genérico):
    // echo "Ocurrió un error al conectar con la base de datos. Por favor
intente más tarde.";

    // Registrar el error en un archivo de log:
    error_log("Error de base de datos: " . $e->getMessage(), 0);
}
```

2. Nunca mostrar errores de base de datos al usuario final en producción

Los mensajes de error pueden revelar información sensible. En un entorno de producción, muestra mensajes genéricos al usuario y registra los detalles en logs.

3. Cerrar las conexiones cuando termines

```
$conexion = null; // Cierra la conexión en PDO
```

4. Limitar los permisos del usuario de la base de datos

El usuario de MySQL que uses desde PHP debería tener solo los permisos necesarios (SELECT, INSERT, UPDATE, DELETE) y no permisos administrativos.

Conclusión

Conectar PHP con MySQL es un paso fundamental en el desarrollo web. A través de esta guía, hemos visto:

1. Cómo establecer conexiones usando MySQLi y PDO
2. Las ventajas de usar PDO
3. Cómo organizar el código de conexión
4. Realizar operaciones básicas (CRUD)
5. Prevenir ataques de inyección SQL
6. Utilizar transacciones

7. Gestionar errores correctamente

Con estos conocimientos, puedes comenzar a desarrollar aplicaciones web que interactúen con bases de datos de manera segura y eficiente.

Ejercicios prácticos

1. Crea una base de datos llamada "mi_tienda" con una tabla "productos"
2. Desarrolla un script PHP que se conecte a esta base de datos
3. Crea un formulario que permita añadir nuevos productos
4. Implementa una página que muestre todos los productos en una tabla
5. Añade funcionalidad para editar y eliminar productos

Estos ejercicios pondrán en práctica lo que has aprendido y te ayudarán a consolidar tus conocimientos.