50 ANIVERSARIO

IES ANTONIO SEQUEROS ALMORADÍ

TEMA

6

Base de Datos

2021/2022

Versión: 220110.1841

IES Antonio Sequeros

ÍNDICE

1. BASES DE DATOS SQL

1.1. PHP y bases de datos

Cuando una aplicación web necesita conservar información de forma permanente para recuperarla posteriormente, suele ser conveniente recurrir a un sistema gestor de bases de datos (SGBD).

Normalmente los SGBD son aplicaciones externas que se instalan y administran de forma separada. PHP permite utilizar la mayoría de los SGBD más conocidos, libres (MySQL, PostgreSQL, MariaDB, Firebird, etc) o comerciales (Oracle, MS SQL Server, Actian X (antes llamada Ingres), etc) como se puede consultar en el apartado sobre acceso a bases de datos del manual de PHP.

Un caso particular es SQLite, que no es una aplicación externa sino una biblioteca en C que implementa un motor de bases de datos SQL. Eso quiere decir que PHP puede gestionar bases de datos directamente, sin necesidad de recurrir a SGBD externos.

Aunque PHP siempre ha permitido utilizar numerosos SGBD, la forma de hacerlo ha ido variando con el tiempo.

1.2. Acceso mediante bibliotecas específicas

En las primeras versiones de PHP, la única manera de acceder a un SGBD era a través de una biblioteca específica, que contenía las funciones necesarias. A esas bibliotecas se les suelen llamar también extensiones. En algunos casos esas extensiones se incluían en las distribuciones oficiales de PHP y para poder utilizarlas era suficiente con incluir la directiva correspondiente en el archivo de configuración php.ini. En otros casos esas extensiones no estaban incluidas en las distribuciones oficiales de PHP, pero estaban incluidas en PEAR o PECL y para poder utilizarlas era necesario instalarlas por separado.

Con el paso del tiempo, este enfoque fue mostrando sus limitaciones. El principal inconveniente de este enfoque es que cada extensión está estrechamente vinculada a cada SGBD y si se quiere cambiar de SGBD es necesario reescribir completamente la aplicación (los nombres de cada función son diferentes, el orden de los argumentos distinto, las funcionalidades disponibles son diferentes, etc.). Además en algunos casos, las extensiones no están mantenidas adecuadamente (bugs de seguridad, etc.).

Actualmente (enero de 2022), PHP se sigue distribuyendo con muchas de estas extensiones, pero en su lugar se recomienda utilizar la biblioteca PDO que se comenta a continuación.

1.3. Acceso mediante PDO

Para poder escribir programas independientes del SGBD elegido, es necesario utilizar una **capa de abstracción** que permita acceder de una forma común a cualquier SGBD. Esta necesidad no es exclusiva de PHP, sino que afecta a cualquier aplicación que necesita trabajar con un SGBD y no tiene solución completa pues las diferencias entre los SGBD no permiten olvidarse de cuál estamos utilizando realmente.

Una de estas capas de abstracción es ODBC (Open DataBase Connectivity), una API independiente del sistema operativo, lenguaje de programación o SGBD utilizada que se empezó a desarrollar en 1992. En PHP existe una extensión llamada también ODBC que permite conexiones ODBC (creo que esta extensión se incorporó en PHP 3.0, publicado en junio de 1998, en cualquier caso estaba activada en PHP 4.0, publicado en mayo de 2000). El inconveniente de ODBC es que introduce un elemento más en la cadena, el controlador ODBC del sistema operativo, lo que puede ralentizar el rendimiento.

La capa de abstracción propia de PHP se llama PDO.

PDO, biblioteca orientada a objetos, se podía utilizar como extensión PECL de PHP 5.0, publicado en agosto de 2004 y está incluida en PHP desde PHP 5.1, publicado en noviembre de 2005. PDO incluye controladores para acceder a las bases de datos más populares y la lista se va ampliando con el tiempo (aunque no incluye todos los SGBD que disponen de extensiones específicas). Utilizando PDO no podemos olvidarnos completamente del SGBD utilizado, pero la mayor parte del código es independiente del SGBD y sólo en algunas partes del programa (en la conexión con el SGBD o en la creación de tablas, por ejemplo) el código es específico del SGBD.

Actualmente (enero de 2022), la extensión PDO es la biblioteca recomendada para acceder a SGBD desde PHP.

1.4. PHP y MySQL

PHP ha tenido tres extensiones para acceder a la base de datos MySQL:

- Extensión mysql: Esta extensión fue la primera extensión para MySQL incluida en PHP (creo que esta extensión se incorporó en PHP 3.0, publicado en junio de 1998, en cualquier caso estaba activada en PHP 4.0, publicado en mayo de 2000). En diciembre de 2012 se decidió eliminar la extensión. Desde PHP 5.5 (publicado en junio de 2013) la extensión estuvo desaconsejada y su uso generaba un aviso E_DEPRECATED. Desde PHP 7.0 (publicada en diciembre de 2015), ya no está incluida en PHP.
- Extensión mysqli (improved mysql): Esta extensión se incluyó por primera vez en PHP 5.0 (publicado en julio de 2004). Las principales ventajas de esta extensión son la posibilidad de utilizar una sintaxis orientada a objetos, sentencias preparadas, transacciones y otras.
- Extensión PDO: esta extensión, que permite trabajar con diferentes bases de datos, incluye un controlador para MySQL desde que fue incluida en PHP 5.1 (publicado en noviembre de 2005).

Aunque la extensión mysqli se sigue incluyendo en PHP, en su lugar se recomienda acceder a MySQL mediante la extensión PDO.

Internamente, estas bibliotecas utilizan un controlador para acceder a MySQL. Antiguamente el controlador se llamaba libmysqlclient y había sido desarrollado por la empresa MySQL AB, pero a partir de PHP 5.3 (publicado en junio de 2009), el controlador es mysqlnd (MySQL Native Driver), desarrollado por el equipo de PHP. Nuestros programas no pueden utilizar directamente estos controladores

2. PHP DATA OBJECTS (PDO)

La extensión PDO (PHP Data Objects) permite acceder a distintas bases de datos utilizando las misma funciones, lo que facilita la portabilidad. En PHP 5 existen drivers para acceder a las bases de datos más populares (MySQL, Oracle, MS SQL Server, PostgreSQL, SQLite, Firebird, DB2, Informix, etc). La extensión PDO no evalúa la corrección de las consultas SQL, aunque sí implementa algunas medidas de seguridad mediante las consultas preparadas.

2.1. Conexión con la base de datos

Para conectar con la base de datos hay que crear una instancia de la clase PDO, que se utiliza en todas las consultas posteriores. En cada página php que incluya consultas a la base de datos es necesario conectar primero con la base de datos.

Si no se puede establecer la conexión con la base de datos, puede deberse:

- a que la base de datos no esté funcionando
- a que los datos de usuario no sean correctos
- a que no esté activada la extensión pdo o (en el caso de SQLite) que no exista el camino donde se guarda la base de datos.

2.2. Conexión con MySQL

En el caso de MySQL, para crear el objeto PDO se necesita proporcionar el nombre del servidor, el nombre de usuario y la contraseña. En el ejemplo siguiente esos datos se proporcionan como constantes que deberían definirse en el programa.

Para poder acceder a MySQL mediante PDO, debe estar activada la extensión php_pdo_mysql en el archivo de configuración php.ini (véase el apartado extensión pdo_mysql en la lección de configuración de Apache y PHP).

```
// FUNCIÓN DE CONEXIÓN CON LA BASE DE DATOS MYSQL
function conectaDb()
{
    try {
        $tmp = new PDO(MYSQL_HOST, MYSQL_USER, MYSQL_PASSWORD);
        $tmp->setAttribute(PDO::MYSQL_ATTR_USE_BUFFERED_QUERY, true);
        $tmp->exec("set names utf8mb4");
        return($tmp);
    } catch(PDOException $e) {
        print "    <p>Error: No puede conectarse con la base de datos.</p>\n";
        print "\n";
        print "    <p>Error: " . $e->getMessage() . "</p>\n";
        exit();
    }
}
// EJEMPLO DE USO DE LA FUNCIÓN conectaDB()
// La conexión se debe realizar en cada página que acceda a la base de datos
$db = conectaDB();
```

2.3. Conexión con SQLite 3

En SQLite, no se hace una conexión a un servidor, sino que simplemente se indica el archivo que va a contener la base de datos. En SQLite no hay un servidor que gestiona todas las bases de datos, sino que cada base de datos es un archivo independiente (que debe estar situado en un directorio que exista y en el que el servidor web tenga permisos de escritura).

Para poder utilizar SQLite mediante PDO, debe estar activada la extensión php_pdo_sqlite en el archivo de configuración php.ini

```
// FUNCIÓN DE CONEXIÓN CON LA BASE DE DATOS SQLITE
function conectaDb() {
    global $dbDb;
    try {
        $tmp = new PDO("sqlite:" . $dbDb);
        return($tmp);
    } catch(PDOException $e) {
        print "    <p>Error: No puede conectarse con la base de datos.</p>\n";
        print "\n";
        print "    <p>Error: " . $e->getMessage() . "</p>\n";
        exit();
    }
}
// EJEMPLO DE USO DE LA FUNCIÓN conectaDb()
// La conexión se debe realizar en cada página que acceda a la base de datos
$db = conectaDB();
```

2.4. Conexión configurable

Si se incluyen ambas conexiones en el mismo programa, cada usuario puede elegir la base de datos más conveniente en cada caso.

En los ejercicios en este curso se propone al alumno organizar los programas de manera que puedan trabajar tanto con SQLite como con MySQL y hacerlo de forma organizada, para que se puedan añadir fácilmente otras bases de datos.

Para ello se crearán dos bibliotecas, una dedicada a MySQL y otra a SQLite, que contengan las funciones específicas de cada base de datos. Además habrá una biblioteca general en la que se pueda seleccionar la biblioteca a utilizar (MySQL o SQLite). Así, cada página llamará a la biblioteca general y esta llamará a la biblioteca específica.

Por ejemplo, para el caso de la función de conexión, el resultado sería:

- en cualquier fichero que quiera conectar con la base de datos se llamaría a la biblioteca general y se llamaría a la función genérica conectaDB():

```
// EJEMPLO DE USO DE CONEXIÓN CONFIGURABLE
// La conexión se debe realizar en cada página que acceda a la base de datos
require_once "biblioteca.php";
$db = conectaDB();
```

- biblioteca.php: en ella se selecciona la biblioteca específica a cargar:

```
// biblioteca.php
define("MYSQL",          "MySQL");         // Base de datos MySQL
define("SQLITE",         "SQLite");        // Base de datos SQLITE
$dbMotor = SQLITE;                         // Base de datos empleada (MYSQL o SQLITE)
if ($dbMotor == MYSQL) {
    require_once "biblioteca-mysql.php";
} elseif ($dbMotor == SQLITE) {
    require_once "biblioteca-sqlite.php";
}
```

- biblioteca-mysql.php: contiene la definición de la función conectaDB() específica para trabajar con MySQL

```
// biblioteca-mysql.php
function conectaDb() {
    try {
        $tmp = new PDO(MYSQL_HOST, MYSQL_USER, MYSQL_PASSWORD);
        $tmp->setAttribute(PDO::MYSQL_ATTR_USE_BUFFERED_QUERY, true);
        $tmp->exec("set names utf8mb4");
        return($tmp);
    } catch(PDOException $e) {
        print "    <p>Error: No puede conectarse con la base de datos.</p>\n";
        print "\n";
        print "    <p>Error: " . $e->getMessage() . "</p>\n";
        exit();
    }
}
```

- biblioteca-sqlite.php: contiene la definición de la función conectaDB() específica para trabajar con SQLite

```
// biblioteca-sqlite.php
function conectaDb() {
    global $dbDb;
    try {
        $tmp = new PDO("sqlite:" . $dbDb);
        return($tmp);
    } catch(PDOException $e) {
        print "    <p>Error: No puede conectarse con la base de datos.</p>\n";
        print "\n";
        print "    <p>Error: " . $e->getMessage() . "</p>\n";
        exit();
    }
}
```

2.5. Desconexión con la base de datos

Para desconectar con la base de datos hay que destruir el objeto PDO. Si no se destruye el objeto PDO, PHP lo destruye al terminar la página.

```
require_once "biblioteca.php";
$db = conectaDB();
// ...
$db = null;
```

2.6. Consultas a la base de datos

Una vez realizada la conexión a la base de datos, las operaciones se realizan a través de consultas.

El método para efectuar consultas es PDO->query($consulta), que devuelve el resultado de la consulta. Dependiendo del tipo de consulta, el dato devuelto debe tratarse de formas distintas.

- Si es una consulta que no devuelve registros, sino que simplemente realiza una acción que puede tener éxito o no (por ejemplo, insertar un registro), el método devuelve true o false. No es necesario guardar el resultado de la consulta en ninguna variable, pero se puede utilizar para sacar un mensaje diciendo que todo ha ido bien (o no). Por ejemplo,

```
// EJEMPLO DE CONSULTA DE INSERCIÓN DE REGISTRO
require_once "biblioteca.php";
$db = conectaDB();
$consulta = "INSERT INTO $dbTabla (nombre, apellidos) VALUES ("$nombre", "$apellidos")";
if ($db->query($consulta)) {
    print "    <p>Registro creado correctamente.</p>\n";
} else {
    print "    <p>Error al crear el registro.<p>\n";
}
$db = null;
```

- Pero si la consulta devuelve registros, el método devuelve los registros correspondientes o false. En ese caso sí que es conveniente guardar lo que devuelve el método en una variable para procesarla posteriormente. Si contiene registros, la variable es de un tipo especial llamado recurso que no se puede acceder directamente, pero que se puede recorrer con un bucle foreach(),

```
// EJEMPLO DE CONSULTA DE SELECCIÓN DE REGISTROS
require_once "biblioteca.php";
$db = conectaDB();
$consulta = "SELECT * FROM $dbTabla";
$result = $db->query($consulta);
if (!$result) {
    print "    <p>Error en la consulta.</p>\n";
} else {
    foreach ($result as $valor) {
        print "    <p>$valor[nombre] $valor[apellidos]</p>\n";
    }
}
```

```
$db = null;
```

En los ejemplos, se define una variable $consulta que contiene la consulta y a continuación se ejecuta la consulta, pero podría estar en una sola:

```
// En dos líneas
$consulta = "SELECT * FROM $dbTabla";
$result = $db->query($consulta);
// En una sola línea
$result = $db->query("SELECT * FROM $dbTabla");
```

Se recomienda utilizar la primera versión, que permite por ejemplo imprimir la consulta mientras se está programando para comprobar que no tiene errores:

```
$consulta = "SELECT * FROM $dbTabla";
print "<p>Consulta: $consulta</p>\n";
$result = $db->query($consulta);
```

2.7. Seguridad en las consultas: consultas preparadas

Para evitar ataques de inyección SQL (en la lección Inyecciones SQL se comentan los ataques más elementales), se recomienda el uso de sentencias preparadas, en las que PHP se encarga de "desinfectar" los datos en caso necesario.

En general, cualquier consulta que incluya **datos introducidos por el usuario** debe realizarse mediante consultas preparadas.

2.8. Consultas preparadas

El método para efectuar consultas es primero preparar la consulta con PDO->prepare($consulta) y después ejecutarla con PDO->execute([parámetros]), que devuelve el resultado de la consulta.

```
// Consulta preparada
$consulta = "SELECT * FROM $dbTabla";
$result = $db->prepare($consulta);
$result->execute();
```

Dependiendo del tipo de consulta, el dato devuelto debe tratarse de formas distintas, como se ha explicado en el apartado anterior.

Si la consulta incluye datos introducidos por el usuario, los datos pueden incluirse directamente en la consulta, pero en ese caso, PHP no realiza ninguna "desinfección" de los datos, por lo que estaríamos corriendo riesgos de ataques:

```
$nombre    = $_REQUEST["nombre"];
$apellidos = $_REQUEST["apellidos"];
$consulta = "SELECT COUNT(*) FROM $dbTabla
    WHERE nombre=$nombre
    AND apellidos=$apellidos";     // DESACONSEJADO: PHP NO DESINFECTA LOS DATOS
$result = $db->prepare($consulta);
$result->execute();
if (!$result) {
    print "    <p>Error en la consulta.</p>\n";
 ...
```

Para que PHP desinfecte los datos, estos deben enviarse al ejecutar la consulta, no al prepararla. Para ello es necesario indicar en la consulta la posición de los datos. Esto se puede hacer de dos maneras, mediante parámetros o mediante interrogantes, aunque se aconseja la utilización de parámetros:

- mediante parámetros (:parametro)

  En este caso la matriz debe incluir los nombres de los parámetros y los valores que sustituyen a los parámetros (el orden no es importante), como muestra el siguiente ejemplo:

```
$nombre    = $_REQUEST["nombre"];
$apellidos = $_REQUEST["apellidos"];
$consulta = "SELECT COUNT(*) FROM $dbTabla
    WHERE nombre=:nombre
    AND apellidos=:apellidos";
$result = $db->prepare($consulta);
$result->execute([":nombre" => $nombre, ":apellidos" => $apellidos]);
if (!$result) {
    print "    <p>Error en la consulta.</p>\n";
 ...
```

- mediante interrogantes (?)

  En este caso la matriz debe incluir los valores que sustituyen a los interrogantes en el mismo orden en que aparecen en la consulta, como muestra el siguiente ejemplo:

```
$nombre    = $_REQUEST["nombre"];
$apellidos = $_REQUEST["apellidos"];
$consulta = "SELECT COUNT(*) FROM $dbTabla
    WHERE nombre=?
    AND apellidos=?";
$result = $db->prepare($consulta);
$result->execute([$nombre, $apellidos]);
if (!$result) {
    print "    <p>Error en la consulta.</p>\n";
 ...
```

Aunque no vayan a causar problemas en las consultas, sigue siendo conveniente tratar los datos recibidos para eliminar los espacios en blanco iniciales y finales, tratar los caracteres especiales del html, etc., como se comenta en esta lección de Recogida de datos.

Restricciones en los parámetros de consultas preparadas

Debido a que las consultas preparadas se idearon para optimizar el rendimiento de las consultas, el uso de parámetros tiene algunas restricciones. Por ejemplo

- los identificadores (nombres de tablas, nombres de columnas, etc) no pueden sustituirse por parámetros
- los dos elementos de una igualdad no pueden sustituirse por parámetros
- en general no pueden utilizarse parámetros en las consultas DDL (lenguaje de definición de datos) (nombre y tamaño de las columnas, etc.)

Si no podemos usar parámetros, no queda más remedio que incluir los datos en la consulta. Como en ese caso PHP no hace ninguna desinfección de los datos, la tenemos que hacer nosotros previamente.

Como en estos casos los valores introducidos por el usuario suelen tener unos valores restringidos (por ejemplo, si el usuario puede elegir una columna de una tabla, los nombres de las columnas están determinadas y el usuario sólo puede elegir uno de ellos).

Podemos crear una función de recogida de datos específica que impida cualquier tipo de ataque de inyección por parte del usuario, como muestra el siguiente ejemplo:

```
// FUNCIÓN DE RECOGIDA DE UN DATO QUE SÓLO PUEDE TOMAR DETERMINADOS VALORES
$columnas = [
    "nombre",
    "apellidos"
];
function recogeValores($var, $valoresValidos, $valorPredeterminado)
{
    foreach ($valoresValidos as $valorValido) {
        if (isset($_REQUEST[$var]) && $_REQUEST[$var] == $valorValido) {
            return $valorValido;
        }
    }
    return $valorPredeterminado;
}

// EJEMPLO DE USO DE LA FUNCIÓN ANTERIOR
$columna = recogeValores("columna", $columnas, "apellidos");
$nombre  = $_REQUEST["nombre"];
$consulta = "SELECT * FROM $dbTabla
    WHERE nombre=:nombre
    ORDER BY $columna ASC";
$result = $db->prepare($consulta);
$result->execute([":nombre" => $nombre]);
if (!$result) {
    print "<p>Error en la consulta.</p>\n";
 ...
```

3. EJEMPLOS DE CONSULTAS

En los ejemplos de este apartado, se han utilizado sentencias preparadas en los casos en los que las consultas incluyen datos proporcionados por el usuario y consultas no preparadas cuando no incluyan datos proporcionados por el usuario. En la mayoría de los casos se podrían haber utilizado sentencias preparadas aunque no haya datos proporcionados por el usuario.

3.1. Consultas CREATE DATABASE, DROP DATABASE, CREATE TABLE

Estas consultas no son iguales en MySQL y SQLite. En los ejercicios propuestos para que se pueda utilizar una u otra base de datos, estas consultas se incluyen en las bibliotecas específicas.

- En el caso de utilizar SQLite, no tiene sentido crear o borrar la base de datos ya que con SQLite cada base de datos es un fichero distinto y al conectar con la base de datos ya se dice con qué archivo se va a trabajar y se crea en caso necesario.
- Para crear una tabla, se utiliza la consulta CREATE TABLE. Las consultas de creación de tabla suelen ser específicas de cada base de datos. Los ejemplos no utilizan sentencias preparadas (en caso de utilizarse sentencias preparadas, las variables no podrían ir como parámetros por tratarse de sentencias DDL).

Consultas en MySQL

Crear Base de Datos

Para crear una base de datos, se utiliza la consulta CREATE DATABASE.

```
// EJEMPLO DE CONSULTA DE CREACIÓN DE BASE DE DATOS EN MYSQL
$consultaCreaDb = "CREATE DATABASE $dbDb
    CHARACTER SET utf8mb4
    COLLATE utf8mb4_unicode_ci";
if ($db->query($consulta)) {
    print "    <p>Base de datos creada correctamente.</p>\n";
    print "\n";
} else {
    print "    <p>Error al crear la base de datos.</p>\n";
    print "\n";
}
```

Nota: El juego de caracteres utilizado en este curso es **UTF-8**, por lo que en la base de datos MySQL se utiliza el juego de caracteres **utf8mb4** (que permite almacenar cualquier carácter Unicode) y el cotejamiento **utf8mb4_unicode_ci** (que implementa todos los criterios de ordenación de Unicode). Para una explicación más detallada se puede consultar el blog de Mathias Bynens.

Borrar Base de Datos

Para borrar una base de datos, se utiliza la consulta DROP DATABASE.

```
// EJEMPLO DE CONSULTA DE BORRADO DE BASE DE DATOS EN MYSQL
$consulta = "DROP DATABASE $dbDb";
if ($db->query($consulta)) {
    print "    <p>Base de datos borrada correctamente.</p>\n";
    print "\n";
} else {
    print "    <p>Error al borrar la base de datos.</p>\n";
    print "\n";
}
```

Crear Tabla

Para crear una tabla, se utiliza la consulta CREATE TABLE. Las consultas de creación de tabla suelen ser específicas de cada base de datos. El ejemplo no utiliza sentencias preparadas (en caso de utilizarse sentencias preparadas, las variables no podrían ir como parámetros por tratarse de sentencias DDL).

```
// EJEMPLO DE CONSULTA DE CREACIÓN DE TABLA EN MYSQL
$consulta = "CREATE TABLE $dbTabla (
    id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
    nombre VARCHAR($tamNombre),
    apellidos VARCHAR($tamApellidos),
    PRIMARY KEY(id)
    )";
if ($db->query($consulta)) {
    print "    <p>Tabla creada correctamente.</p>\n";
    print "\n";
} else {
    print "    <p>Error al crear la tabla.</p>\n";
    print "\n";
}
```

Consultas en SQLite

En el caso de utiliza SQLite, no tiene sentido crear o borrar la base de datos ya que con SQLite cada base de datos es un fichero distinto y al conectar con la base de datos ya se dice con qué archivo se va a trabajar y se crea en caso necesario. Es suficiente borrar y crear las tablas.

Borra Tabla

```
// EJEMPLO DE CONSULTA DE BORRADO DE TABLA EN SQLITE
$consulta = "DROP TABLE $dbTabla";
if ($db->query($consulta)) {
    print "    <p>Tabla borrada correctamente.</p>\n";
    print "\n";
} else {
    print "    <p>Error al borrar la tabla.</p>\n";
    print "\n";
}
```

Crear Tabla

Para crear una tabla, se utiliza la consulta CREATE TABLE. Las consultas de creación de tabla suelen ser específicas de cada base de datos. El ejemplo no utiliza sentencias preparadas (en caso de utilizarse sentencias preparadas, las variables no podrían ir como parámetros por tratarse de sentencias DDL).

```
// EJEMPLO DE CONSULTA DE CREACIÓN DE TABLA EN SQLite
$consulta = "CREATE TABLE $dbTabla (
    id INTEGER PRIMARY KEY,
    nombre VARCHAR($tamNombre),
    apellidos VARCHAR($tamApellidos)
```

```
    )";
if ($db->query($consulta)) {
    print "    <p>Tabla creada correctamente.</p>\n";
    print "\n";
} else {
    print "    <p>Error al crear la tabla.</p>\n";
    print "\n";
}
```

Solución configurable

El resultado sería

- en el fichero que quiera reiniciar la base de datos se llamaría a la biblioteca general y se llamaría a la función genérica borraTodo():

```
// EJEMPLO DE USO DE CONEXIÓN CONFIGURABLE
// La conexión se debe realizar en cada página que acceda a la base de datos
require_once "biblioteca.php";
$db = conectaDb();
borraTodo($db);
$db = null;
```

- biblioteca-mysql.php: contiene la definición de la función borraTodo() específica para trabajar con MySQL y que borra la base de datos, la crea y crea la tabla:

```
// biblioteca-mysql.php
$consultaCreaTabla = "CREATE TABLE $dbTabla (
    id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,
    nombre VARCHAR($tamNombre),
    apellidos VARCHAR($tamApellidos),
    PRIMARY KEY(id)
    )";
function borraTodo($db) {
    global $dbDb, $consultaCreaTabla;
    $consulta = "DROP DATABASE $dbDb";
    if ($db->query($consulta)) {
        print "    <p>Base de datos borrada correctamente.</p>\n";
        print "\n";
    } else {
        print "    <p>Error al borrar la base de datos.</p>\n";
        print "\n";
    }
    $consulta = "CREATE DATABASE $dbDb";
    if ($db->query($consulta)) {
        print "    <p>Base de datos creada correctamente.</p>\n";
        print "\n";
        $consulta = $consultaCreaTabla;
```

```
            if ($db->query($consulta)) {
                print "    <p>Tabla creada correctamente.</p>\n";
            } else {
                print "    <p>Error al crear la tabla.</p>\n";
            }
        } else {
            print "    <p>Error al crear la base de datos.</p>\n";
        }
    }
```

- biblioteca-sqlite.php: contiene la definición de la función borraTodo() específica para trabajar con SQLite y que borra la tabla y la crea:

```
    // biblioteca-sqlite.php
    $consultaCreaTabla = "CREATE TABLE $dbTabla (
        id INTEGER PRIMARY KEY,
        nombre VARCHAR($tamNombre),
        apellidos VARCHAR($tamApellidos)
        )";
    function borraTodo($db)
    {
        global $dbTabla, $consultaCreaTabla;
        $consulta = "DROP TABLE $dbTabla";
        if ($db->query($consulta)) {
            print "    <p>Tabla borrada correctamente.</p>\n";
            print "\n";
        } else {
            print "    <p>Error al borrar la tabla.</p>\n";
            print "\n";
        }
        $consulta = $consultaCreaTabla;
        if ($db->query($consulta)) {
            print "    <p>Tabla creada correctamente.</p>\n";
        } else {
            print "    <p>Error al crear la tabla.</p>\n";
            print "\n";
        }
    }
```

3.2. Consultas DROP TABLE, INSERT INTO, UPDATE, DELETE FROM

Borrar Tabla

Para borrar una tabla, se utiliza la consulta DROP TABLE.

```
// EJEMPLO DE CONSULTA DE BORRADO DE TABLA
$consulta = "DROP TABLE $dbTabla";
if ($db->query($consulta)) {
    print "    <p>Tabla borrada correctamente.</p>\n";
    print "\n";
} else {
    print "    <p>Error al borrar la tabla.</p>\n";
    print "\n";
}
```

Añadir Registro

Para añadir un registro a una tabla, se utiliza la consulta INSERT INTO.

```
// EJEMPLO DE CONSULTA DE INSERCIÓN DE REGISTRO
$nombre    = recoge("nombre");
$apellidos = recoge("apellidos");
$consulta = "INSERT INTO $dbTabla
    (nombre, apellidos)
    VALUES (:nombre, :apellidos)";
$result = $db->prepare($consulta);
if ($result->execute([":nombre" => $nombre, ":apellidos" => $apellidos])) {
    print "    <p>Registro creado correctamente.</p>\n";
    print "\n";
} else {
    print "    <p>Error al crear el registro.</p>\n";
    print "\n";
}
```

Modificar Registro

Para modificar un registro a una tabla, se utiliza la consulta UPDATE.

```
// EJEMPLO DE CONSULTA DE MODIFICACIÓN DE REGISTRO
$nombre    = recoge("nombre");
$apellidos = recoge("apellidos");
$id        = recoge("id");
$consulta = "UPDATE $dbTabla
    SET nombre=:nombre, apellidos=:apellidos
    WHERE id=:id";
$result = $db->prepare($consulta);
```

```
if ($result->execute([":nombre" => $nombre, ":apellidos" => $apellidos, ":id" => $id])) {
    print "    <p>Registro modificado correctamente.</p>\n";
    print "\n";
} else {
    print "    <p>Error al modificar el registro.</p>\n";
    print "\n";
}
```

Borrar Registro

Para borrar un registro de una tabla, se utiliza la consulta DELETE FROM.

Nota: En el ejemplo, los registros a borrar se reciben en forma de matriz y se recorre la matriz borrando un elemento en cada iteración.

```
// EJEMPLO DE CONSULTA DE BORRADO DE REGISTRO
$id = recogeMatriz("id");
foreach ($id as $indice => $valor) {
    $consulta = "DELETE FROM $dbTabla
        WHERE id=:indice";
    $result = $db->prepare($consulta);
    if ($result->execute([":indice" => $indice])) {
        print "    <p>Registro borrado correctamente.</p>\n";
        print "\n";
    } else {
        print "    <p>Error al borrar el registro.</p>\n";
        print "\n";
    }
}
```

3.3. Consulta SELECT

Para obtener registros que cumplan determinados criterios se utiliza una consulta SELECT.

- Si se produce un error en la consulta, la consulta devuelve el valor false .

```
// EJEMPLO DE CONSULTA DE SELECCIÓN DE REGISTROS
       $consulta = "SELECT * FROM $dbTabla";
       $result = $db->query($consulta);
       if (!$result) {
           print "    <p>Error en la consulta.</p>\n";
           print "\n";
       } else {
           print "    <p>Consulta ejecutada.</p>\n";
           print "\n";
       }
```

- Si la consulta devuelve un único registro se puede utilizar la función PDOStatement->fetchColumn() para recuperar la primera columna.

```
// EJEMPLO DE CONSULTA DE SELECCIÓN DE REGISTROS
$consulta = "SELECT COUNT(*) FROM $dbTabla";
$result = $db->query($consulta);
if (!$result) {
    print "    <p>Error en la consulta.</p>\n";
    print "\n";
} else {
    $encontrados = $result->fetchColumn();
    print "    <p>Se han encontrado $encontrados registros.</p>\n";
    print "\n";
}
```

- Si la consulta se ejecuta correctamente, la consulta devuelve los registros correspondientes.
  - Para acceder a los registros devueltos por la consulta, se puede utilizar un bucle foreach.

```
// EJEMPLO DE CONSULTA DE SELECCIÓN DE REGISTROS
$consulta = "SELECT * FROM $dbTabla";
$result = $db->query($consulta);
if (!$result) {
    print "    <p>Error en la consulta.</p>\n";
    print "\n";
} else {
    foreach ($result as $valor) {
        print "    <p>Nombre: $valor[nombre] - Apellidos: $valor[apellidos]</p>\n";
        print "\n";
    }
}
```

- O también se puede utilizar la función PDOStatement->fetch().

```
// EJEMPLO DE CONSULTA DE SELECCIÓN DE REGISTROS
$consulta = "SELECT * FROM $dbTabla";
$result = $db->query($consulta);
if (!$result) {
    print "    <p>Error en la consulta.</p>\n";
    print "\n";
} else {
    while ($valor = $result->fetch()) {
        print "    <p>Nombre: $valor[nombre] - Apellidos: $valor[apellidos]</p>\n";
        print "\n";
    }
}
```

- Si la consulta no devuelve ningún registro, los dos bucles anteriores (foreach o fetch) no escribirían nada. Por ello se recomienda hacer primero una consulta que cuente el número de resultados de la consulta y, si es mayor que cero, hacer la consulta.

- El ejemplo siguiente utiliza la función PDOStatement->fetchColumn(), que devuelve la primera columna del primer resultado (que en este caso contiene el número de registros de la consulta).

```
// EJEMPLO DE CONSULTA DE SELECCIÓN DE REGISTROS
$consulta = "SELECT COUNT(*) FROM $dbTabla";
$result = $db->query($consulta);
if (!$result) {
    print "    <p>Error en la consulta.</p>\n";
    print "\n";
} elseif ($result->fetchColumn() == 0) {
    print "    <p>No se ha creado todavía ningún registro en la tabla.</p>\n";
    print "\n";
} else {
    $consulta = "SELECT * FROM $dbTabla";
    $result = $db->query($consulta);
    if (!$result) {
        print "    <p>Error en la consulta.</p>\n";
        print "\n";
    } else {
        foreach ($result as $valor) {
            print "    <p>Nombre: $valor[nombre] - Apellidos: $valor[apellidos]</p>\n";
            print "\n";
        }
    }
}
```

3.4. Consulta SELECT LIKE

La consulta SELECT permite efectuar búsquedas en cadenas utilizando el condicional LIKE o NOT LIKE y los comodines _ (cualquier carácter) o % (cualquier número de caracteres).

Ejemplos de consultas:

- Registros en los que el apellido empieza por la cadena recibida:

```
// EJEMPLO DE CONSULTA DE SELECCIÓN DE REGISTROS
$apellidos = recoge("apellidos");
$consulta = "SELECT COUNT(*) FROM $dbTabla
    WHERE apellidos LIKE :apellidos";
$result = $db->prepare($consulta);
$result->execute([":apellidos" => "$apellidos%"]);
if (!$result) {
    print "    <p>Error en la consulta.</p>\n";
```

```
            print "\n";
        } else {
            print "    <p>Se han encontrado " . $result->fetchColumn() . " registros.</p>\n";
            print "\n";
        }
```

- Registros en los que el apellido contiene la cadena recibida:

```
        // EJEMPLO DE CONSULTA DE SELECCIÓN DE REGISTROS
        $apellidos = recoge("apellidos");
        $consulta = "SELECT COUNT(*) FROM $dbTabla
            WHERE apellidos LIKE :apellidos";
        $result = $db->prepare($consulta);
        $result->execute([":apellidos" => "%$apellidos%"]);
        if (!$result) {
            print "    <p>Error en la consulta.</p>\n";
            print "\n";
        } else {
            print "    <p>Se han encontrado " . $result->fetchColumn() . " registros.</p>\n";
            print "\n";
        }
```

3.5. Consultas de unión de tablas

Se pueden también realizar consultas de unión entre varias tablas (el ejemplo está sacado del ejercicio de Biblioteca):

Nota: Escribir como consulta preparada.

```
// EJEMPLO DE CONSULTA DE UNIÓN DE TABLAS
$consulta = "SELECT $dbPrestamos.id AS id, $dbUsuarios.nombre as nombre,
    $dbUsuarios.apellidos as apellidos, $dbObras.titulo as titulo,
    $dbPrestamos.prestado as prestado, $dbPrestamos.devuelto as devuelto
    FROM $dbPrestamos, $dbUsuarios, $dbObras
    WHERE $dbPrestamos.id_usuario=$dbUsuarios.id AND
    $dbPrestamos.id_obra=$dbObras.id and $dbPrestamos.devuelto='0000-00-00'
    ORDER BY $columna $orden";
$result = $db->query($consulta);
if (!$result) {
    print "    <p>Error en la consulta / {$pdo->errorInfo()[2]}</p>\n";
    print "\n";
} else {
   ...
```