

ACCESO A BASES DE DATOS MEDIANTE PHP

OBJETIVOS

- Analizar las ventajas que ofrece conectar con sistemas gestores de bases de datos en una aplicación web
- Determinar los elementos necesarios para utilizar bases de datos desde una aplicación web
- Capturar errores de bases de datos adecuadamente
- Ejecutar instrucciones SQL desde la aplicación web
- Transformar datos procedentes de la base de datos a una forma más humana y visual
- Crear estructuras relacionales apropiadas para almacenar la información de una aplicación web
- Crear aplicaciones web que requieran manipular bases de datos

CONTENIDOS

6.1 BASES DE DATOS

- 6.1.1 VENTAJAS DE LAS BASES DE DATOS
- 6.1.2 PROCESO DE ACCESO A UN SISTEMA DE BASES DE DATOS DESDE PHP
- 6.1.3 PROCESO DE ACCESO A LAS BASES DE DATOS DESDE PHP

6.2 GESTIÓN DE ERRORES

- 6.2.1 IMPORTANCIA DE LA GESTIÓN DE ERRORES
- 6.2.2 CONFIGURACIÓN DE REPORTE DE ERRORES

6.3 USAR MYSQL DESDE PHP

- 6.3.1 CONEXIÓN A MYSQL DESDE PHP
- 6.3.2 USO DE MYSQLI. ¿FUNCIONES U OBJETOS?

6.4 ESTABLECER CONEXIÓN CON MYSQL DESDE PHP

- 6.4.1 PERSISTENCIA DE LAS CONEXIONES
- 6.4.2 CONTROL DE ERRORES EN LA CONEXIÓN
- 6.4.3 CERRAR LA CONEXIÓN CON LA BASE DE DATOS

6.5 SELECCIONAR BASES DE DATOS

6.6 EJECUCIÓN DE INSTRUCCIONES SQL

- 6.6.1 SQL GENÉRICO DE MYSQL

6.6.2 OBTENER EL NÚMERO DE FILAS MODIFICADAS EN INSTRUCCIONES DML

6.6.3 GESTIÓN DE ERRORES AL EJECUTAR INSTRUCCIONES SQL

6.7 OBTENER INFORMACIÓN MEDIANTE INSTRUCCIONES SELECT

- 6.7.1 USO DE LA SENTENCIA QUERY PARA EJECUTAR INSTRUCCIONES SELECT
- 6.7.2 RECOGIDA DE LOS RESULTADOS
- 6.7.3 FUNCIONES INTERESANTES DE LOS CONJUNTOS DE RESULTADOS
- 6.7.4 CODIFICACIÓN DE CARÁCTERES
- 6.7.5 PROBLEMAS DE SEGURIDAD. INYECCIONES DE SQL

6.8 SOPORTE DE TRANSACCIONES

- 6.8.1 TRANSACCIONES EN MYSQL
- 6.8.2 AUTOCOMMIT
- 6.8.3 CONFIRMAR Y ANULAR TRANSACCIONES

6.9 PRÁCTICAS RESUELTA

6.10 PRÁCTICAS PROPUESTAS

6.11 RESUMEN DE LA UNIDAD

6.12 TEST DE REPASO

6.1 BASES DE DATOS

Nota: Se da por hecho en este tema que se tienen conocimientos al menos básicos sobre el lenguaje SQL y el funcionamiento de las bases de datos relacionales tales como: creación de tablas, uso de transacciones, consultas, modificación de datos, etc.

6.1.1 VENTAJAS DE LAS BASES DE DATOS

El uso de bases de datos es imprescindible en cualquier aplicación web de hoy en día. Algunas razones para utilizarlas son:

- **Proporcionan almacenamiento permanente.** Es decir, los datos que se almacenan, en principio, lo harán de forma permanente.
- **Añaden una capa más de seguridad.** Los datos están fuera del alcance directo del usuario, ya que no se almacenan en las peticiones http. Los datos se encuentran en un servidor de base de datos con el que se comunica el servidor web. Esta comunicación es totalmente opaca al usuario, el servidor PHP presenta los resultados de interaccionar con el servidor de bases de datos, pero no el código ni las claves para abrir la base de datos.

Aunque hay técnicas para intentar acceder a la base de datos usando trucos, lo cierto es que las bases de datos son un software más preparado para la seguridad que si nosotros gestionáramos directamente la información.

- **Proporcionan herramientas avanzadas de gestión de datos y usuarios.** Los Sistemas Gestores de Bases de Datos son un software especializado en la gestión de datos, permite hacer consultas avanzadas, crear diferentes vistas y permitir su uso a ciertos usuarios. También aportan facilidad para hacer copias de seguridad de los datos, gestión de transacciones, validación avanzada de datos, etc.
- **Uso del lenguaje SQL.** Las bases de datos relacionales (todavía hoy en día son las mayoritarias) poseen el lenguaje estándar SQL para manipular los datos. Es un lenguaje muy conocido por todos los profesionales de la información y proporciona una poderosa sintaxis para gestionar la información.

Hay que tener en cuenta que también hay bases de datos NoSQL, es decir, bases de datos no relacionales que utilizan otros lenguajes para manipular sus datos como es el caso de **MongoDB**, **Cassandra**, **CouchDB** o **HBase**. La mayoría de ellas también se pueden utilizar con PHP, aunque ciertamente no es lo habitual.

6.1.2 PROCESO DE ACCESO A UN SISTEMA DE BASES DE DATOS DESDE PHP

Para que podamos acceder a sistemas de bases de datos desde PHP, necesitamos instalar una interfaz software que comunique a ambos servidores. Cualquier lenguaje de servidor (aunque no

sea PHP) requiere lo mismo, un software (o incluso varios) que conecten al servidor web con el de la base de datos.

El proceso de conexión de un servidor PHP con un servidor de bases de datos suele conllevar estos pasos:

- [1] Instalar un software de acceso al sistema gestor de bases de datos en el servidor web. No siempre este paso es obligatorio. Por ejemplo para acceder a servidores de bases de datos **Oracle** deberemos instalar el software conocido como **Oracle Instant Client**.
- [2] Colocar la librería PHP que contiene el **API (Applications Programming Interface)**, interfaz de programación de aplicaciones) de acceso al servidor de bases de datos, en el directorio de extensiones de PHP, que es donde se guardan las librerías. Un API no es más que el conjunto de funciones que permiten acceder y manipular las bases de datos del servidor correspondiente.
- [3] Modificar el archivo **php.ini** para asegurar que se carga la librería que hemos instalado.
- [4] Reiniciar el servidor Apache y comprobar que disponemos del API de acceso a la base de datos.

Realmente, cada base de datos supone su propio proceso de instalación, las más populares poseen documentación oficial en la dirección:

<http://es1.php.net/manual/es/refs.database.vendors.php>

En el caso de MySQL es muy fácil ya que dispone de una integración excelente con PHP. La instalación de MySQL se ha explicado en el Apartado 2.6 "Instalación Y Configuración de MySQL", en la página 77.

6.1.3 PROCESO DE ACCESO A LAS BASES DE DATOS DESDE PHP

Suponiendo que tenemos correctamente instalado el acceso al servidor de bases de datos deseado desde el servidor de PHP, una aplicación que desee utilizar información de una base de datos necesita realizar los siguientes pasos:

- [1] **Conectarse a la base de datos.** Lo que suele implicar una función a la que se le pasa la llamada **cadena de conexión** con la base de datos. Esa cadena suele tener estos elementos:
 - **Dirección IP o nombre del servidor de la base de datos.** En entornos de producción, el servidor de base de datos suele ser físicamente diferente del servidor PHP, por lo que necesitamos indicar cuál es.
 - **Puerto de acceso.** La conexión requiere indicar con qué puerto debemos comunicar con la base de datos. Si no se indica, se tomará el puerto por defecto. MySQL utiliza por defecto el 3306.

- **Usuario y contraseña.** Evidentemente, si el sistema de bases de datos tiene un mínimo de seguridad, el acceso estará restringido a aquellas personas que tengan permiso. Para verificar este derecho, se requiere autenticar a dicha persona.
- **Nombre de la base de datos.** Esto difiere según la base de datos. En MySQL no es obligatorio para conectar, pero sí para acceder a los datos, ya que están organizados en bases de datos. Oracle, por ejemplo, tiene otro tipo de organización en la que cada usuario posee su propio esquema de datos.

- [2] **Preparar la instrucción SQL que deseamos lanzar sobre la base de datos.** Esto supone que el servidor prepare los recursos necesarios para el acceso a sus datos. A veces, también se realizan tareas de seguridad, como por ejemplo evitar ataques por inyección de SQL.
- [3] **Ejecutar la instrucción SQL.**
- [4] **Recoger los resultados de la instrucción SQL y mostrarlos al usuario.**
- [5] A veces se requiere volver al paso 4 ya que los datos se van recuperando poco a poco.
- [6] **Cerrar la instrucción.** No es obligatorio en todas las bases de datos, pero en las que lo permiten, se liberan los recursos que el sistema ha utilizado para ejecutar la instrucción.
- [7] Podemos ejecutar más instrucciones, para lo cual volvemos al paso 2.
- [8] **Cerrar la conexión y liberar recursos.**

6.2 GESTIÓN DE ERRORES

6.2.1 IMPORTANCIA DE LA GESTIÓN DE ERRORES

En los pasos de proceso de bases de datos comentados en el punto anterior, faltarían aquellos que permiten gestionar los errores.

Al utilizar bases de datos hay muchos posibles errores que pueden ocurrir. Ejemplos de errores son:

- Fallo al realizar la conexión: porque el servidor de bases de datos no está en funcionamiento, por fallo en la red, por indicar mal los datos de conexión, etc.
- Fallo en la ejecución de una instrucción SQL: por mala sintaxis, porque la instrucción incumple reglas de validación de datos, etc.
- Fallo al recoger la información de la instrucción SQL.
- Fallo al cerrar la conexión.
- Fallo por interrupción de la comunicación con la base de datos.

No gestionar estos errores significa depender de la gestión que el servidor PHP haga de ellos. Sin gestión y por defecto, el servidor muestra los errores como un mensaje Warning que aparece

en la página web. Esto es un error gravísimo, ya que esa información no ayuda en nada al usuario, el cual se asustará por un mensaje que no entiende en absoluto; además ese tipo de mensajes puede dar pistas a aquellas personas que intenten atacar nuestro sistema porque sabrán qué servidores estamos utilizando y, sabiendo sus vulnerabilidades, utilizar esa información para romper el sistema.

Todas las API de acceso a la base de datos poseen funciones con las que podemos capturar los errores y obrar de forma más apropiada con ellos. Es muy importante conocer esas funciones.

6.2.2 CONFIGURACIÓN DE REPORTE DE ERRORES

Dentro de la configuración de PHP (mediante el archivo `php.ini`) podemos especificar qué errores se mostrarán al usuario. La directiva `error_reporting` es la encargada de decidir qué errores se muestran y cuáles se ignoran, según su nivel de gravedad.

Inicialmente cuando estamos en proceso de depuración de nuestro código, nos interesa mostrar todo lo que ocurre, pero cuando pasamos a producción, solo lo más grave debería mostrarse. Por ello debemos calibrar bien la directiva `error_reporting` a nuestras necesidades. Sus posibles valores son:

- **E_ERROR**: Solo se mostrarán los errores fatales (errores irrecuperables, graves).
- **E_WARNING**: Se muestran las advertencias(*warnings*) que son errores más leves.
- **E_NOTICE**: Avisos en tiempo de ejecución (más leves que las anteriores).
- **E_CORE_ERROR**: Igual que **E_ERROR** pero solo muestra los errores procedentes del núcleo de PHP.
- **E_CORE_WARNING**: Igual que **E_WARNING** pero solo muestra los errores procedentes del núcleo de PHP.
- **E_COMPILE_WARNING**: Avisos de compilación de motores como por ejemplo Zend.
- **E_USER_ERROR**: Errores generados por el propio programador.
- **E_USER_WARNING**: Advertencias generadas por el propio programador.
- **E_USER_NOTICE**: Avisos generados por el propio programador.
- **E_STRICT**: Avisa del código utilizado por el programador que, aunque funciona, no es correcto (porque podría tener problemas en versiones futuras de PHP).
- **E_RECOVERABLE_ERROR**: Error fatal, pero que no impidió la ejecución del motor PHP al no considerarse inestable tras el fallo.
- **E_DEPRECATED**: Avisos sobre código obsoleto.
- **E_USER_DEPRECATED**: Avisos sobre código obsoleto lanzados por el propio programador.

- **E_ALL**. Muestra todos los errores (salvo los **E_STRICT** hasta la versión 5.4 de PHP).

Inicialmente el valor de **error_reporting** en el archivo `php.ini` suele estar establecido de esta forma:

```
error_reporting = E_ALL | E_STRICT
```

Por lo tanto se muestran todos los errores. Si queremos mostrar solo los errores graves podemos utilizar:

```
error_reporting = E_ERROR
```

En lugar de cambiar la directiva **error_reporting** en el archivo de configuración de PHP, podemos utilizar la función **error_reporting**, que permite modificar el nivel de errores que se muestran, pero en tiempo de ejecución. Es decir, desde el momento en el que se invoca a la función decidiremos qué errores se mostrarán independientemente de la configuración de `php.ini`.

Ejemplo:

```
error_reporting(E_ERROR);
```

6.3 USAR MYSQL DESDE PHP

6.3.1 CONEXIÓN A MYSQL DESDE PHP

Suponemos ya instalado y funcionando MySQL. MySQL dispone de tres API para PHP:

- La API clásica **mysql**.
- La API mejorada **mysqli** (disponible desde la versión 5.0).
- La api **PDO** (disponible desde la versión 5.1).

Se consideran activas las dos últimas. La recomendada actualmente es **mysqli**. Tanto **PDO** como **mysqli** utilizan objetos, pero **mysqli** permite también el acceso mediante funciones sin usar objetos, aunque la mayor parte de programadores utilizan objetos por ser más versátiles.

La conexión entre PHP y MySQL mediante **mysqli** requiere que se cargue con PHP la librería **php_mysqli** (en Windows `php_mysql.dll` y en Linux `php_myso.so`). Para hacerlo hoy en día basta con quitar el comentario (si lo hubiera) referido a esa librería en el archivo de configuración `php.ini`, o bien añadir a mano esta línea (Windows):

```
extension=php_mysqli.dll
```

Por supuesto, debemos tener disponible esa librería en el directorio de extensiones. Lo normal es que en la mayoría de instalaciones no haga falta tocar nada, bien porque las instalaciones por paquetes o las de software completo, como XAMPP, vienen ya preparadas con **mysqli**. En la instalación en Linux mediante código fuente, se suele compilar el código PHP con la opción

--with-mysqli (véase la Unidad 2 para más información), lo que asegura la instalación correcta de la librería

6.3.2 USO DE MYSQLI. ¿FUNCIONES U OBJETOS?

Los usuarios de la librería mysql clásica de acceso a MySQL no suelen estar cómodos con el uso de objetos. Por ello se permite utilizar con mysqli la forma clásica que no implica el uso de objetos.

Sin embargo, los amantes de la programación orientada a objetos se encontrarán más cómodos trabajando con esta librería usando sus capacidades con los objetos. Para diferenciar las dos formas veamos este ejemplo:

```
$mysql = mysqli_connect("127.0.0.1", "root", "12345");
mysqli_select_db("prueba");
$resultado = mysqli_query("SELECT * FROM alumnos");
```

Esta es la forma procedimental de acceder, muy parecido a la librería clásica de acceso a MySQL. La forma orientada a objetos sería:

```
$mysqli = new mysqli("127.0.0.1", "root", "12345", "prueba");
$resultado = $mysqli->query("SELECT * FROM alumnos");
```

En este manual se ha optado por la segunda opción al ser la recomendada actualmente. A pesar de no haber explicado en este manual cómo funcionan los objetos, el uso de la librería es muy sencillo y entendible. También hay que observar que para acceder a una base de datos en el servidor local, se utiliza la dirección IP de máquina local: **127.0.0.1**, en lugar del nombre **localhost**, ya que la dirección es más seguro que funcione en todos los sistemas.

6.4 ESTABLECER CONEXIÓN CON MYSQL DESDE PHP

Si ya hemos instalado correctamente MySQL, establecer la conexión significa crear un nuevo objeto que representará la propia conexión con la base de datos. La sintaxis para crear una nueva conexión es:

```
objetoMySQL = new mysqli([servidor [,usuario [,contraseña
[, baseDatos [, puerto [, socket]]]]]);
```

Los parámetros son todos opcionales. Se explican a continuación:

- **servidor**. Nombre y puerto de la máquina a la que nos conectamos (es decir, el servidor MySQL). Se indica su nombre o IP y, opcionalmente el puerto. Si no se indica su valor se recoge de la directiva PHP (presente en el archivo de configuración **php.ini**) **mysql.default_host** que, normalmente, vale **localhost_3306** (es decir, conexión a servidor local MySQL mediante el puerto 3306, el puerto habitual de comunicación de MySQL).

- **usuario.** Nombre del usuario de la base de datos con el que nos conectamos (y que, por lo tanto, al menos tendrá permiso de conexión). De no indicarlo, se recoge de la directiva PHP **mysql.default_user**.
- **contraseña.** Contraseña del usuario con el que nos conectamos. Si no se indica se recoge de la directiva **mysql.default_pw** (por defecto vale nulo).
- **base de datos.** Indica el nombre de la base de datos con la que se trabajará en MySQL. Si no se elige, lo deberemos de hacer luego porque no se pueden ejecutar la mayoría de instrucciones SQL en MySQL sin seleccionar una base de datos.
- **puerto.** Puerto de conexión con MySQL. Por defecto es el 3306 (puerto habitual de MySQL) que es el establecido en el parámetro **mysql.default_port**.
- **socket.** Indica el nombre del socket a utilizar. Si no, se usa el establecido en **mysql.default_socket**, que suele tomar el valor **MySQL**.

Ejemplo:

```
$mysqli=new mysqli ("127.0.0.1","andrei","12345","prueba");
```

En el ejemplo conectamos con el servidor local de MySQL utilizando el usuario **andrei** con contraseña **12345** y conectamos usando la base de datos **prueba**.

Si la conexión es correcta, la función devuelve como resultado el objeto de conexión (el enlace a datos) que se suele asignar a una variable (en el ejemplo anterior se los queda **\$con**). En el caso de que falle deberemos comprobar los errores como se explica en el Apartado 6.4.2 Control de errores en la conexión.

6.4.1 PERSISTENCIA DE LAS CONEXIONES

La librería **mysqli** admite que las conexiones sean persistentes. Sin persistencia, una conexión se cierra cuando se ha ejecutado el código PHP de una página. Sin embargo, una conexión persistente se mantiene abierta en diferentes páginas. Para ello, la conexión a la base de datos en las páginas se debe crear usando el mismo servidor, usuario y base de datos. De no ser así se creará más de una conexión.

Las conexiones persistentes ahorran recursos en el servidor de base de datos pero requiere de una mayor dedicación a cada conexión. Que se use o no persistencia depende de estas directivas del archivo de configuración (php.ini) de PHP:

- **mysqli.allow_persistent.** Puede valer 1 (se permite la persistencia) o 0 (no se admite la persistencia).
- **mysqli.max_persistent.** Máximo número de conexiones persistentes que se pueden crear. Con valor 0 son ilimitadas.
- **mysqli.max_links.** Máximo número de conexiones que se permite hacer.

6.4.2 CONTROL DE ERRORES EN LA CONEXIÓN

Hay tres métodos fundamentales en la comprobación de errores:

- **connect_error**. Recoge el error si lo ha habido (de otro modo valdrá falso).
- **connect_errno**. Devuelve el número del error
- **error_list**. Muestra la lista de todos los errores que han ocurrido desde que se realizó la conexión

Ejemplo:

```
$mysqli=new mysqli ('localhost','andrei','12345','prueba');

if ($mysqli->connect_error) {
    echo "Error al conectar: ".$mysqli->connect_errno .".
        $mysqli->connect_error;
}
```

Hasta la versión 5.3 de PHP no se podía utilizar la versión orientada a objetos de estas dos funciones por lo que se debía hacer así:

```
$mysqli=new mysqli ('localhost','andrei','12345','prueba');

if (mysqli_connect_error()){
    error "Error al conectar: ".mysqli_connect_errno()." ".
        mysqli_connect_error();
}
```

Por último, el método **error_list**, que es accesible mediante **mysqli_error_list()** o con **\$mysqli->error_list**) devuelve la lista de todos los errores provocados desde la conexión.

6.4.3 CERRAR LA CONEXIÓN CON LA BASE DE DATOS

El método **close** se encarga de cerrar la conexión. No lo debemos utilizar si deseamos que la conexión sea persistente, pero siempre es conveniente cerrar las conexiones cuando no prevenimos utilizarla; de no hacerlo, estaremos malgastando recursos. MySQL con el tiempo cerrará todas las conexiones que se hayan abierto y no se utilicen, pero si dejamos que lo haga MySQL podría ocurrir que no tuviera tiempo de cerrar si abrimos demasiadas conexiones (y no cerramos ninguna).

Ejemplo (si **\$mysqli** es el objeto creado para conectar):

```
$mysqli->close();
```

6.5 SELECCIONAR BASES DE DATOS

En MySQL las tablas pertenecen a una base de datos, por lo que es imperativo elegir la base de datos en la que vamos a trabajar. Se puede elegir durante la conexión, pero también después mediante el método `select_db()`, al que se le pasa el nombre de la base de datos.

Ejemplo:

```
$mysqli->select_db("almacenes");
```

El método devuelve `true` si no han ocurrido errores al elegir la base de datos y `false` en caso contrario.

6.6 EJECUCIÓN DE INSTRUCCIONES SQL

6.6.1 SQL GENÉRICO DE MYSQL

La función más versátil de PHP para utilizar bases de datos es la función `query`. Dicha función, recibe una instrucción SQL en forma de string y simplemente se la envía a MySQL utilizando la conexión y base de datos seleccionada. Ejemplo:

```
$mysqli->query(  
    "CREATE TABLE personas(id_persona INTEGER,nombre VARCHAR(25))");
```

Esta instrucción crea, si el usuario con el que se ha conectado tiene permisos, una tabla en la base de datos en uso actualmente, de MySQL. Para comunicar con MySQL usa la variable de conexión `$mysqli` que, se supone, se creó anteriormente.

Si la instrucción no funciona, entonces `query` devuelve `false`. Si la instrucción es exitosa los resultados que devuelve dependen del tipo de instrucción SQL:

- Si es una instrucción de tipo `SELECT`, `SHOW`, `DESCRIBE` o `EXPLAIN`, dado que son instrucciones que devuelven un conjunto de resultados, retornan un objeto de tipo `mysqli_result`, lo que comúnmente se conoce como un `result set` u objeto de `conjunto de resultados`. Si la instrucción falla, el método `query` devuelve false.
- Con cualquier otro tipo de instrucción, devuelve `true` si ha sido exitosa y `false` de no ser así.

6.6.2 OBTENER EL NÚMERO DE FILAS MODIFICADAS EN INSTRUCCIONES DML

Ya se ha explicado en el apartado anterior que la función `query` es la que se utiliza para enviar instrucciones SQL a una base de datos MySQL. De esta forma las instrucciones `INSERT`, `DELETE` o `UPDATE`. Se pueden ejecutar directamente con la instrucción anterior.

Ejemplo:

```
$sql= "UPDATE personas SET salario=salario*1.1 WHERE cod_depart=9";
if($mysqli->query($sql)){
    echo "Se ha actualizado la tabla de personas";
} else{
    echo "Ha fallado la instrucción";
}
```

En estas instrucciones es normal querer saber cuántas filas ha modificado la instrucción. La librería **mysqli** aporta un método para obtener esta información. Se trata de **affected_rows** que devuelve el número de filas que se han modificado. Así, podríamos mejorar el código anterior de esta forma:

```
$sql= "UPDATE personas SET salario=salario*1.1 WHERE cod_depart=9";
if($mysqli->query($sql)){
    echo "Se han modificado $mysqli->affected_rows personas";
} else{
    echo "Ha fallado la instrucción";
}
```

Ahora indicamos cuántas personas han modificado la instrucción.

6.6.3 GESTIÓN DE ERRORES AL EJECUTAR INSTRUCCIONES SQL

Cuando falla una instrucción SQL, deberemos gestionar los errores para obrar de la forma más apropiada. Los métodos que gestionan este tipo de errores son:

- **error**. Contiene el mensaje de error procedente de MySQL.
- **errno**. Número de error.

Ejemplo:

```
$sql="CREATE TABLE notas(cod_persona INT(11),nota DECIMAL(5,2))";
if($mysqli->query($sql)){
    echo "Tabla creada";
}
else{
    echo "Fallo al crear la tabla: ";
    $mysqli->error." nº ".$mysqli->errno;
}
```

Generalmente no debemos mostrar los mensajes de error de MySQL, más bien detectar qué error ha ocurrido (sea por el mensaje o por el número) y mostrar información que sea más entendible por nuestros usuarios.

6.7 OBTENER INFORMACIÓN MEDIANTE INSTRUCCIONES SELECT

6.7.1 USO DE LA SENTENCIA QUERY PARA EJECUTAR INSTRUCCIONES SELECT

Como se ha explicado en el Apartado 6.6.1, en las instrucciones de tipo **SELECT** (también en las instrucciones no estándar **DESCRIBE**, **SHOW** o **EXPLAIN**, propias de MySQL), se devuelve un **conjunto de resultados** (un *result set*).

Un conjunto de resultados es una estructura que almacena resultados que tienen forma de tabla (filas y columnas). De forma práctica podemos decir que es una variable capaz de recoger el resultado de una consulta SQL. Un ejemplo sería:

```
$sql="SELECT nombre,apellido1,telefono FROM clientes";
$res=$mysqli->query($sql);
```

\$res es la variable que recogerá el conjunto de resultados (valdrá **false**, si falla la instrucción SQL).

Realmente la función **query** tiene un segundo parámetro. Se trata de un número entero que indica la forma en la que vamos a recoger los datos de la consulta. Se usa con dos posibles constantes:

- **MYSQL_STORE_RESULT**. Es el valor por defecto y hace que en la variable se almacenen **todos** los resultados de la consulta. Esta forma de funcionar requiere bastante memoria y si hemos ejecutado una consulta con numerosos resultados, podemos provocar un error de **desbordamiento de memoria** y detener la ejecución de la aplicación. Si no son muchos los resultados devueltos, **MYSQL_STORE_DEFAULT** es la forma más conveniente de trabajar ya que es más rápida.
- **MYSQL_USE_RESULT**. Representa la forma de trabajo contraria. Los resultados se almacenan hasta llenar un búfer de resultados. El resto de resultados no se vuelcan y cuando los necesitemos, tendremos que hacer una nueva petición de datos a MySQL. La ventaja es que no gasta tanta memoria y la desventaja es que supone más peticiones a la base de datos.

Por lo tanto, si la instrucción anterior requiere almacenar mucha información, lo correcto sería:

```
$sql="SELECT nombre,apellido1,telefono FROM clientes";
$res=$mysqli->query($sql,MYSQL_USE_RESULT);
```

6.7.2 RECOGIDA DE LOS RESULTADOS

Los conjuntos de resultados son un paso previo para realmente obtener los resultados de una consulta SELECT. Hay varios métodos pertenecientes a los objetos de tipo **result set** que permiten recoger los resultados. El más interesante es **fetch_assoc**. Este método permite recorrer fila a fila los resultados de un conjunto de resultados. Cada vez que invocamos a **fetch_assoc** nos devuelve un array asociativo que contiene los datos de la fila de la consulta en la que nos encontramos actualmente; las claves de ese array son el nombre de cada columna.

Es decir, la forma habitual de recorrer los resultados de un SELECT desde PHP es:

- [1] Usar **query** para lanzar la instrucción SELECT. Su resultado será un objeto de tipo **result set** al que le asignaremos una variable.
- [2] Usar el método **fetch_assoc** de esa variable, el cual devuelve un array asociativo en el que los índices son los nombres de las columnas y los valores son los valores de la primera fila de la consulta resultado de la instrucción SELECT. Si no hay resultados, devuelve **false**.
- [3] Si se vuelve a invocar a **fetch_assoc**, se cambia la posición a la fila siguiente del resultado, de la cual se devuelven los valores. Así seguiremos invocando a **fetch_assoc** hasta que devuelva **false**, indicando entonces que no hay más resultados.

Durante el proceso de lectura no se pueden ejecutar otras sentencias sobre esa conexión. Por lo que, tras leer los datos deseados, tenemos que invocar al método **close** del objeto de resultados, y así liberar los recursos de esa instrucción y, en definitiva, cerrarla.

En resumen, el proceso de lectura de datos de una consulta SELECT es el siguiente:

- [1] Ejecutar la instrucción **SELECT** utilizando el método **query** y asignar el resultado a una variable. Si va a haber muchos resultados, utilizar como segundo parámetro de **query**, la constante **MYSQL_USE_RESULT**.
- [2] Comprobar si la variable no tiene el valor **false**, que indicaría que ha fallado la instrucción, y por lo tanto, gestionar los errores (se explica más adelante como controlar los errores de la función **query**).
- [3] Leer la primera fila de resultados utilizando la función **fetch_assoc** perteneciente a la variable en la que hemos almacenado el conjunto de resultados y almacenar ese resultado (un array) en una variable.
- [4] Comprobar que la variable que almacena la fila no tiene valor **false**. Si vale **false** es que ya no hay más valores que leer.
- [5] Mostrar o manipular la forma deseada los datos de la fila actual.
- [6] Leer la siguiente fila invocando de nuevo a **fetch_assoc**.
- [7] Volver al paso 4.
- [8] Si hemos salido del bucle invocar al método **close** de la variable que almacena el conjunto de resultados, para cerrar y liberar la instrucción.

Veamos, por ejemplo, como mostrar los nombres y apellidos de una tabla de personas en forma de tabla:

```

<!doctype html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
</head>
<body>
<?php
$mysqli = new mysqli("127.0.0.1", "root", "345Gtfa");
if ($mysqli->connect_error) {
    echo "Error al realizar la conexión";
}
else {
    $mysqli->select_db("trabajo");
    $sql= "SELECT nombre,apellido1 FROM personas";
    $res=$mysqli->query($sql,MYSQLI_USE_RESULT);
    if($res) {
        echo "<table><tr><th>Nombre</th><th>Apellido</th></tr>";
        $fila=$res->fetch_assoc();
        while($fila){
            echo "<tr><td>{$fila["nombre"]}</td>";
            echo "<td>{$fila["apellido1"]}</td></tr>";
            $fila=$res->fetch_assoc();
        }
        echo "</table>";
        $res->close(); //cierre de la instrucción
    }
    else{
        echo "Fallo al obtener la lista de personas ";
    }
    $mysqli->close(); //cierre de la conexión
}
?>
</body>
</html>
```

6.7.3 FUNCIONES INTERESANTES DE LOS CONJUNTOS DE RESULTADOS

- **num_rows.** Propiedad de los conjuntos de resultados, que devuelve el número de filas de la instrucción SELECT. Si la instrucción se generó con la constante **MYSQL_STORE_RESULT**, es decir, sin usar búfer de resultados, entonces, esta función podría fallar al

indicar las filas, ya que el cálculo exacto depende de que se hayan cargado todas las filas. En todo caso es más recomendable utilizar búferes.

- **data_seek.** Función que permite colocarlos en el número de fila que indiquemos. Por ejemplo, si `$res` es la variable que representa al conjunto de resultados, para colocarnos en la ultima fila de los resultados, utilizaríamos la expresión:

```
$res->data_seek($res->num_rows-1);
```

6.7.4 CODIFICACIÓN DE CARACTERES

Uno de los problemas más habituales en la lectura de datos procedentes de MySQL es la cuestión de que la página web muestre la información usando una codificación de texto (por ejemplo Unicode UTF-8) y la información se recibe de MySQL usando otra modificación (por ejemplo en formato Latín 1).

Lo normal hoy en día es que las páginas web se codifiquen en formato **UTF-8 de Unicode**. Para conseguir mostrar texto de otras codificaciones en ese formato, podemos utilizar la función **utf8_encode**. Si, por ejemplo, en la lectura de datos del apartado anterior tuviéramos problemas con la codificación. Podríamos cambiar el bucle **while** de esta forma:

```
while($fila){
    echo "<tr><td>".utf8_encode($fila["nombre"])."</td>";
    echo "<td>".utf8_encode($fila["apellido1"])."</td></tr>";
    $fila=$res->fetch_assoc();
}
```

El método funciona, pero resulta un tanto pesado. Para evitar tener que invocar a esta función continuamente, lo lógico es hacer que todos los datos estén en formato utf-8 en MySQL y además que la comunicación con MySQL utilice ese formato para codificar el texto.

Esto último implica modificar el archivo de configuración de MySQL (**my.ini** o **my.cnf** normalmente). Hoy en día MySQL suele tener un apartado referido a UTF-8 en ese archivo, pero lo tiene entre comentarios. En todo caso para asegurar que la comunicación realmente es en UTF-8 en el apartado **[mysqld]** del archivo de configuración deberán aparecer estas líneas:

```
[mysqld]
...
## UTF 8 Settings
init-connect='SET NAMES utf8'
default-character-set=utf8
collation_server=utf8_unicode_ci
character_set_server=utf8
skip-character-set-client-handshake
```

Si nuestra página PHP utiliza Unicode UTF-8 (es lo recomendable hoy en día), y estamos teniendo siempre la precaución de que dentro de MySQL trabajamos en Unicode, los datos se verán siempre de forma correcta.

6.7.5 PROBLEMAS DE SEGURIDAD. INYECCIONES DE SQL

Puesto que muchas veces los datos que se envían a través de SQL al servidor utilizan formularios que rellenan los usuarios, los hackers podrían tener un resquicio de acceso a la base de datos mediante técnicas de **inyección SQL**. Estas técnicas consisten en añadir de manera camouflada instrucciones SQL para obtener información de nuestra base de datos.

Como ejemplo sencillo y básico para hacernos idea de la importancia de tener en cuenta las inyecciones SQL, supongamos que hemos realizado una página de autentificación de usuarios a través de este formulario:

```
<!doctype html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Formulario de acceso</title>
</head>
<body>
<h1>Datos de acceso</h1>
<form action="acceso.php" method="POST">
    <label for="usuario">Usuario</label>
    <input type="text" name="usuario" id="usuario"/><br/>
    <label for="pass">Contraseña</label>
    <input type="text" name="pass" id="pass"/><br/>
    <button>Enviar</button>
</form>
</body>
</html>
```

El formulario envía el usuario a través del parámetro POST de nombre **usuario** y la contraseña con el nombre **pass**. La página **acceso.php** hará una consulta a la tabla de usuarios y si existe un usuario con ese nombre y contraseña dirá que el usuario es correcto; pero si no, nos dirá que es incorrecto. Supongamos que hemos creado este código para la página **acceso.php**:

```
<!doctype html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Página de usuario</title>
</head>
<body>
<?php
if(isset($_POST["usuario"])) && isset($_POST["pass"])) {
    $usuario=$_POST["usuario"];
```

```

$pass=$_POST["pass"];
$mysqli = new mysqli("127.0.0.1", "root", "s1D43We7");
if($mysqli->connect_error) echo "Error al realizar la conexión";
else {
    $mysqli->select_db("prueba");
    $sql="SELECT nombre,pass FROM usuarios WHERE ".
        "nombre='$usuario' and pass='$pass'";
    $res=$mysqli->query($sql);
    if($res){
        $fila=$res->fetch_assoc();
        if($fila)
            echo "<p>Usuario y contraseña correcta</p>";
        else
            echo "<p>Usuario o contraseña incorrecta</p>";
    }
    else echo "<p>Fallo al ejecutar la instrucción</p>";
}
}
else header("Location:formulario.html");
?>
</body>
</html>

```

En principio todo va bien, cuando el usuario pone los datos correctos, entra correctamente y si pone una mala contraseña, no entra. Sin embargo, en cuanto en la contraseña colocáramos, por ejemplo, una comilla, el mensaje sería el de **"Fallo al ejecutar la instrucción"**. Ese mensaje no es el habitual y da la pista de que algo raro está pasando. Y así, si el usuario (si es conocedor de las técnicas de inyección de SQL) escribe estos datos en el formulario:

Datos de acceso

Usuario	<input type="text" value="jorge"/>
Contraseña	<input type="text" value="12345 or '1'='1"/>
<input type="button" value="Enviar"/>	

Figura 6.1: Ejemplo de inyección SQL

Entonces el usuario entra, sin haber introducido contraseña alguna. La razón es que esta entrada genera la instrucción SQL:

```

SELECT nombre,pass FROM usuarios WHERE nombre='jorge' AND pass='12345
OR '1'='1'

```

El WHERE de esta instrucción siempre es verdadero porque la condición '1'='1' siempre lo es. Luego este código es muy problemático y está a expensas del ataque de terceras personas. La cuestión es cómo evitar este problema.

Estas medidas pueden ayudar:

- Utilizar la función **real_escape_string** que recibe un texto y le devuelve a su formato seguro. Lo que hará es que los caracteres peligrosos (como las comillas, saltos de línea, etc.) se pasan a su forma **escapada**. Por ejemplo, la comilla simple ('') se convierte en (\'), con lo cual no se pueden delimitar nuevas instrucciones o elementos y estaremos más protegidos ante una inyección.

```
$sqlSeguro=$mysqli->real_escape_string($sql);
$mysqli->query($sqlSeguro);
```

- No utilizar la función **multi_query**. Esa instrucción es una variable de **query**, que admite ejecutar varias instrucciones SQL a la vez si se separa cada una de ellas con un punto y coma. Ante una eventual inyección de código SQL estaríamos en serio peligro porque si podemos ejecutar varias instrucciones a la vez, entonces, podemos añadir una instrucción SQL completa y malévolas.
- Conectar, si es posible, con un usuario que tenga los mínimos privilegios posibles. Es muy mala idea conectar con el usuario root. Lo lógico en una web que accede a una base de datos solo para examinar la información, es crear un usuario con privilegios de lectura sobre las tablas que deseamos utilizar.
- Verificar o convertir los datos que se esperan sean numéricos. Teniendo en cuenta que éstos en SQL no utilizan comillas, la función **real_scape_string** no nos ayudaría. Por lo que antes de incrustar ese dato en nuestra instrucción SQL realmente habría que comprobar que es válido.
- Cifrar contraseñas y otros datos, tanto al almacenar la información como al enviarla durante la comunicación entre la base de datos y el servidor PHP. Para más información sobre cifrado de datos, véase el Apartado 4.5 "Cifrado", en la página 206.

6.8 SOPORTE DE TRANSACCIONES

6.8.1 TRANSACCIONES EN MYSQL

En SQL estándar existen dos instrucciones que permiten anular o confirmar transacciones: son **ROLLBACK** (anular) y **COMMIT** (confirmar). Ambas trabajan con la transacción en curso. Una transacción comienza cuando se ejecuta una instrucción DML (**INSERT**, **DELETE** o **UPDATE**) y finaliza cuando se acepta o confirma (también puede finalizar por otras razones como cerrar la conexión o ejecutar una instrucción DDL por ejemplo).

MySQL soporta transacciones solo en bases de datos y tablas gestionadas por motores compatibles con el uso de transacciones. El motor actual, **InnoDB**, soporta transacciones de forma completa, lo que se conoce como soporte **ACID** de transacciones. ACID son las siglas referentes a:

- **Atomicidad**. Asegura que una instrucción no se pueda ejecutar a medias, o se ejecuta del todo o no hace ninguna labor.

- **Consistencia.** Asegura que el manejo de una transacción nunca deje a la base de datos en un estado incoherente.
- **Aislamiento (*Isolation*).** Asegura que las operaciones de una transacción no afectan a otras operaciones que se estén produciendo en la base de datos, permanecen independientes.
- **Persistencia (*Durability*).** Asegura que cuando confirmemos o anulemos una transacción, ese estado sea realmente definitivo y no temporal.

6.8.2 AUTOCOMMIT

Por defecto en PHP todas las instrucciones DML se confirman (**COMMIT**) al instante de forma automática, por lo que no pueden ser revocadas. Este estado es el habitual ya que es el más cómodo para evitar que una transacción quede abierta demasiado tiempo, ya que las transacciones son costosas de mantener por parte del servidor de bases de datos.

Pero el objeto de conexión (normalmente le llamamos **\$mysqli**) posee un método llamado **autocommit** que recibe como parámetro un valor booleano: true significaría que se confirma automáticamente cada instrucción DML y false que se gestionan transacciones, por lo que las instrucciones serán definitivas si se usa el método COMMIT.

6.8.3 CONFIRMAR Y ANULAR TRANSACCIONES

La forma de anular y confirmar transacciones es a través de dos métodos del objeto de conexión (**\$mysqli**):

- **commit.** Confirma la transacción actual.
- **rollback.** Anula la transacción actual.

Ejemplo de funcionamiento:

```
<?php
$mysqli = new mysqli("miservidor.com", "alice", "45FdA", "almacenes");
$mysqli->autocommit(false); //se activa la gestión de transacciones

$mysqli->query("INSERT INTO ventas VALUES(12,300,'Samsung galaxy S4')");
//...
//supongamos que hemos detectado que hay que anular esa inserción:
$mysqli->rollback(); //anula la transacción

//nuevas instrucciones
$mysqli->query("INSERT INTO ventas VALUES(12,350,'Samsung galaxy S4')");
$mysqli->query("UPDATE productos SET precio=precio*1.1");
//...
$mysqli->commit(); //confirmamos las dos instrucciones anteriores
?>
```

6.11 RESUMEN DE LA UNIDAD

- Las bases de datos aportan importantes ventajas a la creación de aplicaciones web, fundamentalmente: almacenamiento permanente, una capa extra de seguridad, herramientas avanzadas de gestión de datos y el uso de un lenguaje especializado en datos (normalmente SQL).
- El acceso a las bases de datos desde PHP requiere cargar un módulo que contiene la API de acceso a la base de datos concreta (por ejemplo *mysqli*) al arrancar PHP. En ocasiones se requiere también instalar un software conector en el servidor (por ejemplo el *Oracle Instant Client* para conectar con servidores *Oracle Database*).
- Los datos necesarios para conectar con una base de datos son: la IP o nombre del servidor de base de datos, el puerto que utiliza, el nombre de la base de datos a utilizar, un usuario y su contraseña de acceso.
- Para acceder a MySQL las librerías recomendadas son **mysqli** y **pdo**. La primera es la que más se utiliza actualmente.
- El proceso de acceso a una base de datos MySQL con la librería **mysqli** es:
 - [1] Crear el objeto de conexión (normalmente se llama **\$mysqli**).
 - [2] Ejecutar la instrucción SQL con ayuda del método **query**.
 - [3] Recoger los datos de la instrucción si es de consulta con ayuda del método **fetch_assoc**. Si no es de consulta basta con comprobar si se ha ejecutado bien.
 - [4] Cerrar la conexión.
- Un problema de seguridad habitual es la inyección de SQL, que puede permitir un acceso no deseado a la base de datos. Para evitar este (y otros problemas) hay que tomar las medidas pertinentes.
- Es posible utilizar instrucciones de transacción con MySQL, siempre y cuando el motor de la base de datos lo permita.