TEMA 3

Contenido

Trabajar con bases de datos en PHP.

Caso práctico
Una de las tareas prioritarias que tienen que abordar en el nuevo proyecto de BK Programación es el almacenamiento de la información que utilizará la aplicación web, y el método de acceso que se utilizará para manejarla desde PHP.
*En una reunión de trabajo, **Esteban** les informa que para la gestión de la empresa están utilizando una aplicación de código libre que almacena los datos en un servidor MySQL. Afortunadamente, este servidor es el más utilizado en la programación con lenguaje PHP, por lo que no tendrán problemas en integrar la nueva aplicación web con la ya existente. Solo necesitan conocer la estructura de los datos que se almacenan, y ver qué métodos puede usar para manejar la información.*

1.- Acceso a bases de datos desde PHP.

Caso práctico
***Carlos es nuevo en el mundo de la programación web**. Además, apenas ha trabajado con bases de datos, por lo que se asombra de la gran diversidad de opciones que existen en PHP para trabajar con datos almacenados en servidores de distintos tipos.*
Algunos de los gestores sobre los que lee mientras revisa la documentación de PHP los conoce, otros simplemente le suenan, pero hay muchos de los que ni siquiera conocía su existencia. Sabe que debe centrarse en el servidor MySQL, que es el que usarán para desarrollar la aplicación, pero aun así el volumen de información disponible es tan grande que le cuesta decidirse por dónde empezar.

Una de las aplicaciones más frecuentes de PHP es generar un interface web para acceder y gestionar la información almacenada en una base de datos. Usando PHP podemos mostrar en una página web información extraída de la base de datos, o enviar sentencias al gestor de la base de datos para que elimine o actualice algunos registros.

PHP soporta más de 15 sistemas gestores de bases de datos: SQLite, Oracle, SQL Server, PostgreSQL, IBM DB2, MySQL, etc. Hasta la versión 5 de PHP, el acceso a las bases de datos se hacía principalmente utilizando extensiones específicas para cada sistema gestor de base de datos (extensiones nativas). Es decir, que si queríamos acceder a una base de datos de PostgreSQL, deberíamos instalar y utilizar la extensión de ese gestor en concreto. Las funciones y objetos a utilizar eran distintos para cada extensión.

A partir de la versión 5 de PHP se introdujo en el lenguaje una extensión para acceder de una forma común a distintos sistemas gestores: PDO. La gran ventaja de PDO está clara: podemos seguir utilizando una misma sintaxis aunque cambiemos el motor de nuestra base de datos. Por el contrario, en algunas ocasiones preferiremos seguir usando extensiones nativas en nuestros programas. Mientras PDO ofrece un conjunto común de funciones, las extensiones nativas normalmente ofrecen más potencia (acceso a funciones específicas de cada gestor de base de datos) y en algunos casos también mayor velocidad.

De los distintos SGBD existentes, vas a aprender a utilizar MySQL. MySQL es un gestor de bases de datos relacionales de código abierto bajo licencia GNU GPL. Es el gestor de bases de datos más empleado con el lenguaje PHP. Como ya vimos, es la letra "M" que figura en los acrónimos AMP y XAMPP.

En esta unidad vas a ver cómo acceder desde PHP a bases de datos MySQL utilizando tanto PDO como la extensión nativa MySQLi. Previamente verás una pequeña introducción al manejo de MySQL, aunque para el seguimiento de esta unidad se supone que conoces el lenguaje SQL utilizado en la gestión de bases de datos relacionales.

Además, para el acceso a las funcionalidades de ambas extensiones deberás utilizar objetos. Aunque más adelante verás todas las características que nos ofrece PHP para crear programas orientados a objetos, debemos suponer también en este punto un cierto conocimiento de programación orientada a objetos. Básicamente, debes saber cómo crear y utilizar objetos.

En PHP se utiliza la palabra `new` para crear un nuevo objeto instanciando una clase:

```
$a = new A();
```

Y para acceder a los miembros de un objeto, debes utilizar el operador flecha `->`:

```
$a->fecha();
```

Características básicas de la utilización de objetos en PHP

La programación orientada a objetos es una metodología de programación avanzada y bastante extendida, en la que los sistemas se modelan creando clases, que son un conjunto de datos y funcionalidades. Las clases son definiciones, a partir de las que se crean objetos. Los objetos son ejemplares de una clase determinada y como tal, disponen de los datos y funcionalidades definidos en la clase.

La programación orientada a objetos permite concebir los programas de una manera bastante intuitiva y cercana a la realidad. La tendencia es que un mayor número de lenguajes de programación adopten la programación orientada a objetos como paradigma para modelizar los sistemas. Prueba de ello es la nueva versión de PHP (5), que implanta la programación de objetos como metodología de desarrollo. También Microsoft ha dado un vuelco hacia la programación orientada a objetos, ya que .NET dispone de varios lenguajes para programar y todos orientados a objetos.

Así pues, la programación orientada a objetos es un tema de gran interés, pues es muy utilizada y cada vez resulta más esencial para poder desarrollar en casi cualquier lenguaje moderno. En este artículo vamos ver algunas nociones sobre la programación orientada a objetos en PHP. Aunque es un tema bastante amplio, novedoso para muchos y en un principio, difícil de asimilar, vamos a tratar de explicar la sintaxis básica de PHP para utilizar objetos, sin meternos en mucha teoría de programación orientada a objetos en general.

Las clases: class

Una clase es un conjunto de variables, llamados atributos, y funciones, llamadas métodos, que trabajan sobre esas variables. Las clases son, al fin y al cabo, una definición: una especificación de propiedades y funcionalidades de elementos que van a participar en nuestros programas.

Por ejemplo, la clase "Caja" tendría como atributos características como las dimensiones, color, contenido y cosas semejantes. Las funciones o métodos que podríamos incorporar a la clase "caja" son las funcionalidades que deseamos que realice la caja, como `introduce()`, `muestra contenido()`, `comprueba si cabe()`, `vaciate()`...

Las clases en PHP se definen de la siguiente manera:

```
<?
class Caja{
   var $alto;
   var $ancho;
   var $largo;
   var $contenido;
   var $color;

   function introduce($cosa){
      $this->contenido = $cosa;
   }
```

```
   function muestra_contenido(){
      echo $this->contenido;
   }
}
?>
```

En este ejemplo se ha creado la clase Caja, indicando como atributos el ancho, alto y largo de la caja, así como el color y el contenido. Se han creado, para empezar, un par de métodos, uno para introducir un elemento en la caja y otro para mostrar el contenido.

Si nos fijamos, los atributos se definen declarando unas variables al principio de la clase. Los métodos se definen declarando funciones dentro de la clase. La variable `$this`, utilizada dentro de los métodos la explicaremos un poco más abajo.

Utilizar la clase

Las clases solamente son definiciones. Si queremos utilizar la clase tenemos que crear un ejemplar de dicha clase, lo que corrientemente se le llama instanciar un objeto de una clase.

```
$micaja = new Caja;
```

Con esto hemos creado, o mejor dicho, instanciado, un objeto de la clase Caja llamado `$micaja`.

```
$micaja->introduce("algo");
$micaja->muestra_contenido();
```

Con estas dos sentencias estamos introduciendo "`algo`" en la caja y luego estamos mostrando ese contendido en el texto de la página. Nos fijamos que los métodos de un objeto se llaman utilizando el código "`->`".

```
nombre_del_objeto->nombre_de_metodo()
```

Para acceder a los atributos de una clase también se accede con el código "`->`". De esta forma:

```
nombre_del_objeto->nombre_del_atributo
```

La variable $this

Dentro de un método, la variable `$this` hace referencia al objeto sobre el que invocamos el método. En la invocación `$micaja->introduce("algo")` se está llamando al método introduce sobre el objeto `$micaja`. Cuando se está ejecutando ese método, se vuelca el valor que recibe por parámetro en el atributo contenido. En ese caso `$this->contenido` hace referencia al atributo contenido del objeto `$micaja`, que es sobre el que se invocaba el método.

Constructores

Los constructores son funciones, o métodos, que se encargan de realizar las tareas de inicialización de los objetos al ser instanciados. Es decir, cuando se crean los objetos a partir de las clases, se llama a un constructor que se encarga de inicializar los atributos del objeto y realizar cualquier otra tarea de inicialización que sea necesaria.

No es obligatorio disponer de un constructor, pero resultan muy útiles y su uso es muy habitual. En el ejemplo de la caja, que comentábamos anteriormente, lo normal sería inicializar las variables como color o las relacionadas con las dimensiones y, además, indicar que el contenido de la caja está vacío. Si no hay un constructor no se inicializan ninguno de los atributos de los objetos.

El constructor se define dentro de la propia clase, como si fuera otro método. El único detalle es que el constructor debe tener el mismo nombre que la clase. Atentos a PHP, que diferencia entre mayúsculas y minúsculas. A partir de la versión de PHP 5, para definir métodos constructores, el constructor de la clase puede llamarse __construct(). Se permiten las dos formas por temas de compatibilidad con versiones anteirores. A partir de PHP 7 es obligatorio usar __construct.

Para la clase Caja definida anteriormente, se podría declarar este constructor:

```
// function __construct($alto=1,$ancho=1,$largo=1,$color="negro"){
function Caja($alto=1,$ancho=1,$largo=1,$color="negro"){
   $this->alto=$alto;
   $this->ancho=$ancho;
   $this->largo=$largo;
   $this->color=$color;
   $this->contenido="";
}
```

En este constructor recibimos por parámetro todos los atributos que hay que definir en una caja.
Es muy útil definir unos valores por defecto en los parámetros que recibe el constructor, igualando el parámetro a un valor dentro de la declaración de parámetros de la función constructora, pues así, aunque se llame al constructor sin proporcionar parámetros, se inicializará con los valores por defecto que se hayan definido.

Es importante señalar que en los constructores no se tienen por qué recibir todos los valores para inicializar el objeto. Hay algunos valores que pueden inicializarse a vacío o a cualquier otro valor fijo, como en este caso el contenido de la caja, que inicialmente hemos supuesto que estará vacía.

2.- MySQL.

Caso práctico

***Juan y Carlos deciden comenzar revisando el servidor que van a utilizar, MySQL**. Aunque van a utilizar un servidor que ya está en funcionamiento, deben comprender sus capacidades y las herramientas de las que disponen para poder gestionar tanto el servidor como los datos que almacena.*

***María conoce bien MySQL** y les orienta sobre los pasos necesarios para instalarlo y configurarlo. Con su ayuda y con el permiso de **Esteban**, hacen una copia a algunos de los datos que necesitan, y los replican en un servidor local para poder trabajar con ellos. Por supuesto, se aseguran de no utilizar para las pruebas información sensible como la de los clientes o proveedores, que pueda ocasionarles problema legales.*

MySQL es un sistema gestor de bases de datos (SGBD) relacionales. Es un programa de código abierto que se ofrece bajo licencia GNU GPL, aunque también ofrece una licencia comercial en caso de que quieras utilizarlo para desarrollar aplicaciones de código propietario. En las últimas versiones (a partir de la 5.1), se ofrecen, de hecho, varios productos distintos: uno de código libre (Community Edition), y otro u otros comerciales (Standard Edition, Enterprise Edition).

Incorpora múltiples motores de almacenamiento, cada uno con características propias: unos son más veloces, otros, aportan mayor seguridad o mejores capacidades de búsqueda. Cuando crees una base de datos, puedes elegir el motor en función de las características propias de la aplicación. Si no lo cambias, el motor que se utiliza por defecto se llama `MyISAM`, que es muy rápido pero a cambio no contempla integridad referencial (*característica de las bases de datos que permite crear relaciones válidas entre dos registros de la misma o de diferentes tablas, y definir las operaciones necesarias para mantener la validez de las relaciones cuando se borra o modifica alguno de los registros*) ni tablas transaccionales (*conjunto de operaciones sobre los datos que se han de realizar de forma conjunta, una sola vez, e independientemente del resto de manipulaciones sobre los datos. Toda transacción debe cumplir cuatro propiedades: atomicidad, consistencia, aislamiento y permanencia*). El motor `InnoDB` es un poco más lento pero sí soporta tanto integridad referencial como tablas transaccionales.

MySQL se emplea en múltiples aplicaciones web, ligado en la mayor parte de los casos al lenguaje PHP y al servidor web **Apache**. Utiliza SQL para la gestión, consulta y modificación de la información almacenada. Soporta la mayor parte de las características de ANSI SQL 99 (*revisión del estándar ANSI SQL del año 1999, que agrega a la revisión anterior (SQL2 o SQL 92) disparadores, expresiones regulares, y algunas características de orientación a objetos*), y añade además algunas extensiones propias.

3.- Utilización de bases de datos MySQL en PHP.

Caso práctico
*Entre **María, Juan y Carlos**, han creado una pequeña base de datos con cuatro tablas y unas decenas de registros que usarán en las pruebas de la nueva aplicación web.*
***Juan, que ha tenido cierta experiencia programando aplicaciones en PHP**, se da cuenta que el lenguaje ha evolucionado mucho en los últimos tiempos. Y uno de los aspectos que más ha evolucionado es precisamente el que concierne al acceso a bases de datos MySQL.*
En las aplicaciones que había realizado hace ya algunos años, siempre había utilizado la misma extensión. Y ahora, por lo que ha estado viendo, existen otras maneras más eficientes o más genéricas de llevar a cabo esa tarea.
*Para estar seguro, **busca consejo en algunos programadores amigos** y llega a una conclusión: tendrá que **escoger entre una extensión nativa, MySQLi, y PDO**. Revisa la documentación sobre ambas y realiza un pequeño estudio comparativo. Además, diseña unas pruebas para llevar a cabo con la **ayuda de Carlos** y poder tomar una decisión. Siempre es mejor asegurarse antes de empezar, aunque eso implique alargar algo más los plazos.*

Como ya viste, existen dos formas de comunicarse con una base de datos desde PHP: utilizar una extensión nativa programada para un SGBD concreto, o utilizar una extensión que soporte varios tipos de bases de datos. Tradicionalmente las conexiones se establecían utilizando la extensión nativa **mysql**. Esta extensión se mantiene en la actualidad para dar soporte a las aplicaciones ya existentes que la utilizan, pero no se recomienda utilizarla para desarrollar nuevos programas. Lo más habitual es elegir entre **MySQLi** (extensión nativa) y **PDO**.

Con cualquiera de ambas extensiones, podrás realizar acciones sobre las bases de datos como:
- ✓ Establecer conexiones.
- ✓ Ejecutar sentencias SQL.
- ✓ Obtener los registros afectados o devueltos por una sentencia SQL.
- ✓ Emplear transacciones.
- ✓ Ejecutar procedimientos almacenados.
- ✓ Gestionar los errores que se produzcan durante la conexión o en el establecimiento de la misma.

PDO y MySQLi (y también la antigua extensión mysql) utilizan un **driver de bajo nivel** para comunicarse con el servidor MySQL. Hasta hace poco el único driver disponible para realizar esta función era **libmysql**, que no estaba optimizado para ser utilizado desde PHP. A partir de la versión 5.3, PHP viene preparado para utilizar también un nuevo driver mejorado para realizar esta función, el Driver Nativo de MySQL, **mysqlnd**.

3.1.- Extensión MySQLi.

Esta extensión se desarrolló para aprovechar las ventajas que ofrecen las versiones 4.1.3 y posteriores de MySQL, y viene incluida con PHP a partir de la versión 5. Ofrece un interface de programación dual, pudiendo accederse a las funcionalidades de la extensión utilizando objetos o funciones de forma indiferente. Por ejemplo, para establecer una conexión con un servidor MySQL y consultar su versión, podemos utilizar cualquiera de las siguientes formas:

```
// utilizando constructores y métodos de la programación orientada a objetos
$conexion = new mysqli('localhost', 'usuario', 'contraseña', 'base_de_datos');
print conexion->server info;

// utilizando llamadas a funciones
$conexion = mysqli_connect('localhost', 'usuario', 'contraseña', 'base_de_datos');
print mysqli_get_server_info($conexion);
```

En ambos casos, la variable `$conexion` es de tipo objeto. La utilización de los métodos y propiedades que aporta la clase **mysqli** normalmente produce un código más corto y legible que si utilizas llamadas a funciones.

Toda la información relativa a la instalación y utilización de la extensión, incluyendo las funciones y métodos propios de la extensión, se puede consultar en el manual de PHP.
http://es.php.net/manual/es/book.mysqli.php

Entre las mejoras que aporta a la antigua extensión mysql, figuran:
- ✓ Interface orientado a objetos.
- ✓ Soporte para transacciones.
- ✓ Soporte para consultas preparadas.
- ✓ Mejores opciones de depuración.

Como ya viste en la primera unidad, las opciones de configuración de PHP se almacenan en el fichero `php.ini`. En este fichero hay una sección específica para las opciones de configuración propias de cada extensión. Entre las opciones que puedes configurar para la extensión MySQLi están:
- ✓ **`mysqli.allow_persistent`**. Permite crear conexiones persistentes.
- ✓ **`mysqli.default_port`**. Número de puerto TCP predeterminado a utilizar cuando se conecta al servidor de base de datos.
- ✓ **`mysqli.reconnect`**. Indica si se debe volver a conectar automáticamente en caso de que se pierda la conexión.
- ✓ **`mysqli.default_host`**. Host predeterminado a usar cuando se conecta al servidor de base de datos.
- ✓ **`mysqli.default_user`**. Nombre de usuario predeterminado a usar cuando se conecta al servidor de base de datos.
- ✓ **`mysqli.default_pw`**. Contraseña predeterminada a usar cuando se conecta al servidor de base de datos.

3.1.1.- Establecimiento de conexiones.

Para poder comunicarte desde un programa PHP con un servidor MySQL, el primer paso es establecer una conexión. Toda comunicación posterior que tenga lugar, se hará utilizando esa conexión.
Si utilizas la extensión MySQLi, establecer una conexión con el servidor significa crear una instancia de la **clase mysqli**. El constructor de la clase puede recibir seis parámetros, todos opcionales, aunque lo más habitual es utilizar los cuatro primeros:
- ✓ El nombre o dirección IP del servidor MySQL al que te quieres conectar.
- ✓ Un nombre de usuario con permisos para establecer la conexión.
- ✓ La contraseña del usuario.
- ✓ El nombre de la base de datos a la que conectarse.
- ✓ El número del puerto en que se ejecuta el servidor MySQL.
- ✓ El socket o la tubería con nombre (named pipe) a usar.

Si utilizas el constructor de la clase, para conectarte a la base de datos "dwes" puedes hacer:

```
// utilizando el constructor de la clase
$dwes = new mysqli('localhost', 'dwes', 'abc123.', 'dwes');
```

Aunque también tienes la opción de primero crear la instancia, y después utilizar el método `connect` para establecer la conexión con el servidor:

```
// utilizando el método connect
$dwes = new mysqli();
$dwes->connect('localhost', 'dwes', 'abc123.', 'dwes');
```

Por el contrario, utilizando el interface procedimental de la extensión:

```
// utilizando llamadas a funciones
$dwes = mysqli_connect('localhost', 'dwes', 'abc123.', 'dwes');
```

Es muy importante el control y gestión de los errores devueltos por el servidor de Base de Datos. **El comportamiento predeterminado del manejo de errores la MySQLi ha cambiado a partir de la versión 8.1 de PHP**.

a) En versiones de PHP anteriores a la 8.1, silenciaba los errores de MySQL por pantalla y los almacenaba en las propiedades de de error de la clase (**connect_errno, connect_error, errno y error**)

Por ejemplo, el siguiente código comprueba el establecimiento de una conexión con la base de datos "`dwes`" y finaliza la ejecución si se produce algún error:

```
@ $dwes = new mysqli('localhost', 'dwes', 'abc123.', 'dwes');
$error = $dwes->connect_errno;
if ($error != 0) {
   echo "<p>Error $error conectando a la base de datos: $dwes->connect_error</p>";
   exit();
}
```

b) En versiones de PHP posteriores a la 8.1, MySQLi genera una excepción automáticamente cuando el servidor de BD devuelve un error.

```
try{
        $dwes = new mysqli('localhost', 'dwes', 'abc123.', 'dwes');
} catch (Exception $ex) {
        die($ex->getMessage());
}
```

El modo por defecto a partir de PHP 8.1 está establecido como:

mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);

Para trabajar sin excepciones, hay que cambiar el modo por defecto a:

mysqli_report(MYSQLI_REPORT_OFF);

En PHP, como veremos posteriormente con más detalle, puedes anteponer a cualquier expresión el operador de control de errores @ para que se ignore cualquier posible error que pueda producirse al ejecutarla.

http://es.php.net/manual/es/language.operators.errorcontrol.php

Si una vez establecida la conexión, quieres cambiar la base de datos puedes usar el método **`select_db`** (o la función **`mysqli_select_db`** de forma equivalente) para indicar el nombre de la nueva.

```
// utilizando el método connect
$dwes->select_db('otra_bd');
```

Una vez finalizadas las tareas con la base de datos, utiliza el método **`close`** (o la función **`mysqli_close`**) para cerrar la conexión con la base de datos y liberar los recursos que utiliza.

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

Para asegurarnos que los datos obtenidos desde el servidor tienen la codificación UTF-8, podemos ejecutar después de la conexión: ***$dwes->set_charset('utf8mb4');***

3.1.2.- Ejecución de consultas.

La forma más inmediata de ejecutar una consulta, si utilizas esta extensión, es el método **`query`**, equivalente a la función **`mysqli_query`**. Si se ejecuta una consulta de acción que no devuelve datos (como una sentencia SQL de tipo **`UPDATE`**, **`INSERT`** o **`DELETE`**), la llamada devuelve `true` si se ejecuta correctamente o `false` en caso contrario. El número de registros afectados se puede obtener con la propiedad **`affected_rows`** (o con la función **`mysqli_affected_rows`**).

```
@ $dwes = new mysqli('localhost', 'dwes', 'abc123.', 'dwes');
$error = $dwes->connect_errno;
if ($error == 0) {
   $resultado = $dwes->query('DELETE FROM stock WHERE unidades=0');
   if ($resultado) {
```

```
        print "<p>Se han borrado $dwes->affected_rows registros.</p>";
    }
    $dwes->close();
}
```

En el caso de ejecutar una sentencia SQL que sí devuelva datos (como un `SELECT`), éstos se devuelven en forma de un objeto resultado (de la clase **`mysqli_result`**). En el punto siguiente verás cómo se pueden manejar los resultados obtenidos.

El método query tiene un parámetro opcional que afecta a cómo se obtienen internamente los resultados, pero no a la forma de utilizarlos posteriormente. En la opción por defecto, **`MYSQLI_STORE_RESULT`**, los resultados se recuperan todos juntos de la base de datos y se almacenan de forma local. Hasta que no están todos recuperados no se podrán leer. Si cambiamos esta opción por el valor **`MYSQLI_USE_RESULT`**, los datos se van recuperando del servidor según se vayan necesitando, es decir, el recurso se podrá leer aunque todavía no se haya rellenado entero.

```
$resultado = $dwes->query('SELECT producto, unidades FROM stock', MYSQLI_USE_RESULT);
```

Otra forma que puedes utilizar para ejecutar una consulta es el método **real_query** (o la función **mysqli_real_query**), que siempre devuelve true o false según se haya ejecutado correctamente o no. Si la consulta devuelve un conjunto de resultados, se podrán recuperar de forma completa utilizando el método **store_result**, o según vaya siendo necesario gracias al método **use_result**.

http://es.php.net/manual/es/mysqli.real-query.php

Es importante tener en cuenta que los resultados obtenidos se almacenarán en memoria mientras los estés usando. Cuando ya no los necesites, los puedes liberar con el método **`free`** de la clase **`mysqli_result`** (o con la función **`mysqli_free_result`**):

```
$resultado->free();
```

```
if ($mysqli->query('SELECT Name FROM City ORDER BY ID LIMIT 20, 5')) {
   do {
      if ($result = $mysqli->use_result()) {
         while ($row = $result->fetch_row()) {
            printf("%s\n", $row[0]);
         }
         $result->close();
      }
      if ($mysqli->more_results())
            printf("-----------------\n");
   } while ($mysqli->next_result());
}
```

3.1.3.- Transacciones.

Como ya comentamos, si necesitas utilizar transacciones deberás asegurarte de que estén soportadas por el motor de almacenamiento que gestiona tus tablas en MySQL. Si utilizas X, por defecto cada consulta individual se incluye dentro de su propia transacción. Puedes gestionar este comportamiento con el método **`autocommit`** (función **`mysqli_autocommit`**).

```
$dwes->autocommit(false);   // deshabilitamos el modo transaccional automático
```

Al deshabilitar las transacciones automáticas, las siguientes operaciones sobre la base de datos iniciarán una transacción que deberás finalizar utilizando:

- ✓ **`commit`** (o la función **`mysqli_commit`**). Realizar una operación "`commit`" de la transacción actual, devolviendo `true` si se ha realizado correctamente o `false` en caso contrario.
- ✓ **`rollback`** (o la función **`mysqli_rollback`**). Realizar una operación "`rollback`" de la transacción actual, devolviendo `true` si se ha realizado correctamente o `false` en caso contrario.

```
…
$dwes->query('DELETE FROM stock WHERE unidades=0');  // Inicia una transacción
$dwes->query('UPDATE stock SET unidades=3 WHERE producto="STYLUSSX515W"');
…
$dwes->commit();  // Confirma los cambios
```

Una vez finalizada esa transacción, comenzará otra de forma automática.
Las consultas "query" y el commit deben ir dentro del try, y el rollback en el catch. Si las dos instrucciones están correctas se hace el commit, si alguna de ellas fallas, saltaría la excepción y se ejecutaría el rollback.

Según la información que figura en la tabla stock de la base de datos dwes, la tienda 1 (CENTRAL) tiene 2 unidades del producto de código 3DSNG y la tienda 3 (SUCURSAL2) ninguno. Suponiendo que los datos son esos (no hace falta que los compruebes en el código), utiliza una transacción para mover una unidad de ese producto de la tienda 1 a la tienda 3.
Deberás hacer una consulta de actualización (para poner unidades=1 en la tienda 1) y otra de inserción (pues no existe ningún registro previo para la tienda 3).
Comprueba que se ejecuta bien solo la primera vez, pues en ejecuciones posteriores ya no es posible insertar la misma fila en la tabla.

En el modo de gestión de transacciones que se utiliza por defecto, ¿es posible revertir los cambios que se aplican al ejecutar una consulta de acción?

- No
- Sí

3.1.4.- Obtención y utilización de conjuntos de resultados.

Ya sabes que al ejecutar una consulta que devuelve datos obtienes un objeto de la clase **mysqli_result**. Esta clase sigue los criterios de ofrecer un interface de programación dual, es decir, una función por cada método con la misma funcionalidad que éste.

Para trabajar con los datos obtenidos del servidor, tienes varias posibilidades:
fetch_array (función **mysqli_fetch_array)**. Obtiene un registro completo del conjunto de resultados y lo almacena en un array. Por defecto el array contiene tanto claves numéricas como asociativas.

Por ejemplo, para acceder al primer campo devuelto, podemos utilizar como clave el número 0 o su nombre indistintamente.

```
$resultado = $dwes->query('SELECT producto, unidades FROM stock WHERE unidades<2');
$stock = $resultado->fetch_array();  // Obtenemos el primer registro
$producto = $stock['producto'];  // O también $stock[0];
$unidades = $stock['unidades'];  // O también $stock[1];
print "<p>Producto $producto: $unidades unidades.</p>";
```

Este comportamiento por defecto se puede modificar utilizando un parámetro opcional, que puede tomar los siguientes valores:

- ✓ **MYSQLI_NUM**. Devuelve un array con claves numéricas.
- ✓ **MYSQLI_ASSOC**. Devuelve un array asociativo.
- ✓ **MYSQLI_BOTH**. Es el comportamiento por defecto, en el que devuelve un array con claves numéricas y asociativas.

fetch_assoc (función **mysqli_fetch_assoc)**. Idéntico a fetch_array pasando como parámetro **MYSQLI_ASSOC**.
fetch_row (función **mysqli_fetch_row)**. Idéntico a fetch_array pasando como parámetro **MYSQLI_NUM**.
fetch_object (función **mysqli_fetch_object)**. Similar a los métodos anteriores, pero devuelve un objeto en lugar de un array. Las propiedades del objeto devuelto se corresponden con cada uno de los campos del registro.

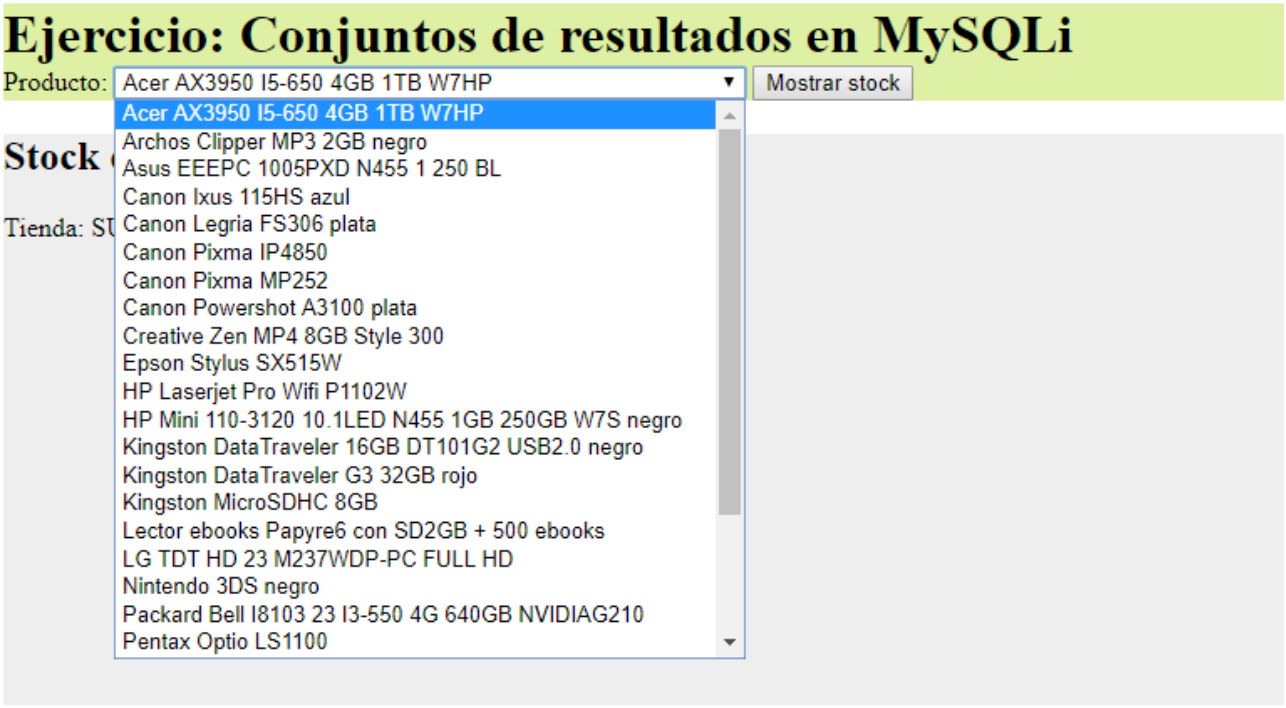
Parar recorrer todos los registros de un array, puedes hacer un bucle teniendo en cuenta que cualquiera de los métodos o funciones anteriores devolverá null cuando no haya más registros en el conjunto de resultados.

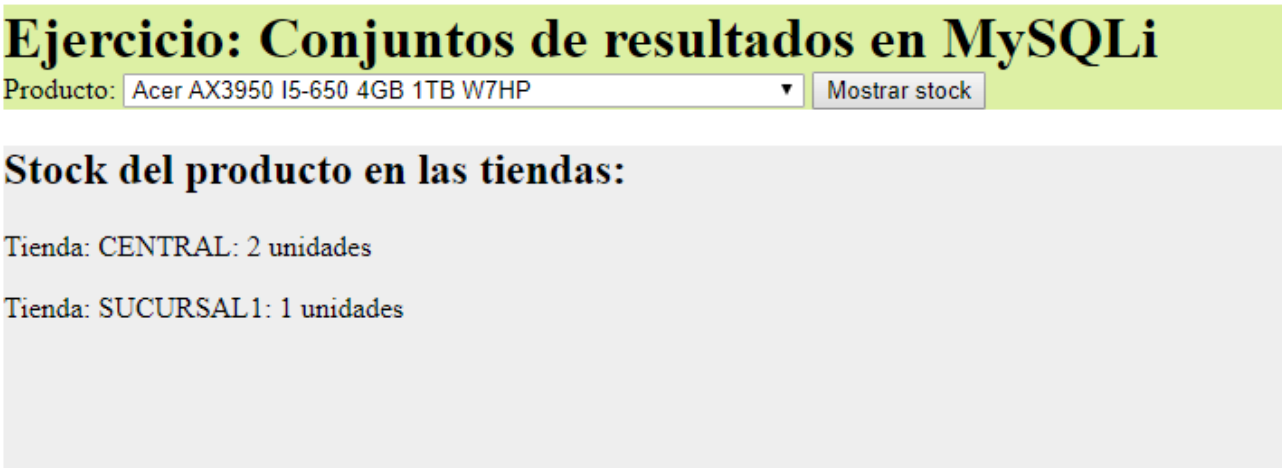
```
$resultado = $dwes->query('SELECT producto, unidades FROM stock WHERE unidades<2');
$stock = $resultado->fetch_object();
while ($stock != null) {      //while($stock=$resultado->fetch_objetct())
   print "<p>Producto $stock->producto: $stock->unidades unidades.</p>";
   $stock = $resultado->fetch object();
}
```

En el manual de PHP tienes más información sobre los métodos y propiedades de la clase mysqli_result.

http://es.php.net/manual/es/class.mysqli-result.php

Crea una página web en la que se muestre el stock existente de un determinado producto en cada una de las tiendas. Para seleccionar el producto concreto utiliza un cuadro de selección dentro de un formulario en esa misma página, en el que se muestre el nombre de los todos los productos que hay.


Ejercicio: Conjuntos de resultados en MySQLi
Producto: Acer AX3950 I5-650 4GB 1TB W7HP
Mostrar stock
Acer AX3950 I5-650 4GB 1TB W7HP
Archos Clipper MP3 2GB negro
Asus EEEPC 1005PXD N455 1 250 BL
Canon Ixus 115HS azul
Canon Legria FS306 plata
Canon Pixma IP4850
Canon Pixma MP252
Canon Powershot A3100 plata
Creative Zen MP4 8GB Style 300
Epson Stylus SX515W
HP Laserjet Pro Wifi P1102W
HP Mini 110-3120 10.1LED N455 1GB 250GB W7S negro
Kingston DataTraveler 16GB DT101G2 USB2.0 negro
Kingston DataTraveler G3 32GB rojo
Kingston MicroSDHC 8GB
Lector ebooks Papyre6 con SD2GB + 500 ebooks
LG TDT HD 23 M237WDP-PC FULL HD
Nintendo 3DS negro
Packard Bell I8103 23 I3-550 4G 640GB NVIDIAG210
Pentax Optio LS1100


Ejercicio: Conjuntos de resultados en MySQLi
Producto: Acer AX3950 I5-650 4GB 1TB W7HP
Mostrar stock
Stock del producto en las tiendas:
Tienda: CENTRAL: 2 unidades
Tienda: SUCURSAL1: 1 unidades

3.1.5.- Consultas preparadas.

Cada vez que se envía una consulta al servidor, éste debe analizarla antes de ejecutarla. Algunas sentencias SQL, como las que insertan valores en una tabla, deben repetirse de forma habitual en un programa. Para acelerar este proceso, MySQL admite consultas preparadas. Estas consultas se almacenan en el servidor listas para ser ejecutadas cuando sea necesario.

Para trabajar con consultas preparadas con la extensión MySQLi de PHP, debes utilizar la clase **mysqli_stmt**. Utilizando el método **stmt_init** de la clase **mysqli** (o la función **mysqli_stmt_init**) obtienes un objeto de dicha clase. También se puede hacer directamente usando el método **prepare(string $query)** de la clase MySQLi que devuelve ya el objeto de la clase **mysqli_stmt**.

```
$dwes = new mysqli('localhost', 'dwes', 'abc123.', 'dwes');
```

Los pasos que debes seguir para ejecutar una consulta preparada son:

- ✓ Preparar la consulta en el servidor MySQL utilizando el método `prepare` (función `mysqli_stmt_prepare`).
- ✓ Ejecutar la consulta, tantas veces como sea necesario, con el método `execute` (función `mysqli_stmt_execute`).
- ✓ Una vez que ya no se necesita más, se debe ejecutar el método `close` (función `mysqli_stmt_close`).

Por ejemplo, para preparar y ejecutar una consulta que inserta un nuevo registro en la tabla familia:

```
$stmt=$dwes->prepare('INSERT INTO familia (cod, nombre) VALUES ("TABLET", "Tablet PC")');

$stmt->execute();
$stmt->close();
$dwes->close();
```

El problema que ya habrás observado, es que de poco sirve preparar una consulta de inserción de datos como la anterior, si los valores que inserta son siempre los mismos. Por este motivo las consultas preparadas admiten parámetros. Para preparar una consulta con parámetros, en lugar de poner los valores debes indicar con un signo de interrogación su posición dentro de la sentencia SQL.

```
$dwes->prepare('INSERT INTO familia (cod, nombre) VALUES (?, ?)');
```

Y antes de ejecutar la consulta tienes que utilizar el método `bind_param` (o la función `mysqli_stmt_bind_param`) para sustituir cada parámetro por su valor. El primer parámetro del método `bind_param` es una cadena de texto en la que cada carácter indica el tipo de un parámetro, según la siguiente tabla.

Caracteres indicativos del tipo de los parámetros en una consulta preparada.	
Carácter.	**Tipo del parámetro.**
I.	Número entero.
D.	Número real (doble precisión).
S.	Cadena de texto.
B.	Contenido en formato binario (BLOB).

En el caso anterior, si almacenas los valores a insertar en sendas variables, puedes hacer:

```
$stmt=$dwes->prepare('INSERT INTO familia (cod, nombre) VALUES (?, ?)');

$cod_producto = "TABLET";
$nombre_producto = "Tablet PC";
$stmt->bind param('ss', $cod producto, $nombre producto);
$stmt->execute();
$stmt->close();
$dwes->close();
```

Cuando uses **bind_param** para enlazar los parámetros de una consulta preparada con sus respectivos valores, deberás usar siempre variables como en el ejemplo anterior. Si intentas utilizar literales, por ejemplo:

```
$->bind_param('ss', 'TABLET', 'Tablet PC');          // Genera un error
```

Obtendrás un error. El motivo es que los parámetros del método bind_param se pasan por referencia. Aprenderás a usar paso de parámetros por referencia en una unidad posterior.

El método **bind_param** permite tener una consulta preparada en el servidor MySQL y ejecutarla tantas veces como quieras cambiando ciertos valores cada vez. Además, en el caso de las consultas que devuelven valores, se puede utilizar el método **bind_result** (función **mysqli_stmt_bind_result**) para asignar a variables los campos que se obtienen tras la ejecución. Utilizando el método **fetch** (**mysqli_stmt_fetch**) se recorren los registros devueltos. Observa el siguiente código:

```
$stmt=$dwes->prepare('SELECT  producto, producto, unidades FROM stock  WHERE unidades<2');
 $stmt->execute();
$stmt->bind_result($producto, $unidades);
while($stmt->fetch()) {
   print "<p>Producto $producto: $unidades unidades.</p>";
}
$stmt->close();
$dwes->close();
```

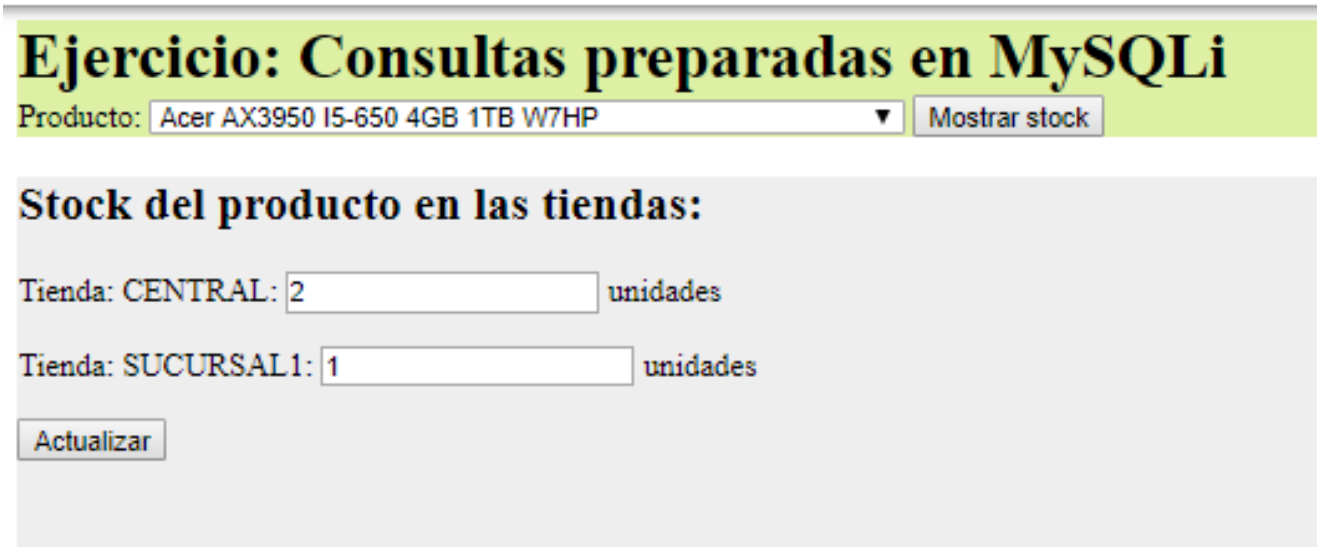
```
stmt=$dwes->prepare('SELECT  producto, producto, unidades FROM stock  WHERE unidades<2');
stmt->execute();
$resultado=$stmt->get_result();
while($fila=$resultado->fetch_object())
   print "<p>Producto $fila->producto: $fila->unidades unidades</p>";
}
$stmt->close();
$dwes->close();
```

En el manual de PHP tienes más información sobre consultas preparadas y la clase **mysqli_stmt**.

http://es.php.net/manual/es/class.mysqli-stmt.php

A partir de la página web obtenida en el ejercicio anterior, añade la opción de modificar el número de unidades del producto en cada una de las tiendas. Utiliza una consulta preparada para la actualización de registros en la tabla stock. No es necesario tener en cuenta las tareas de inserción (no existían unidades anteriormente) y borrado (si el número final de unidades es cero).

En esta ocasión es necesario crear un nuevo formulario en la página, en la sección donde se muestra el número de unidades por tienda. Cuando se envía ese formulario, hay que preparar la consulta y ejecutarla una vez por cada registro de la tabla stock (una vez por cada tienda en la que exista stock de ese producto).



3.2.- PHP Data Objects (PDO).

Si vas a programar una aplicación que utilice como sistema gestor de bases de datos MySQL, la extensión **MySQLi** que acabas de ver es una buena opción. Ofrece acceso a todas las características del motor de base de datos, a la vez que reduce los tiempos de espera en la ejecución de sentencias.

Sin embargo, **si en el futuro tienes que cambiar el SGBD** por otro distinto, tendrás que volver a programar gran parte del código de la misma. Por eso, antes de comenzar el desarrollo, es muy importante revisar las características específicas del proyecto. En el caso de que exista la posibilidad, presente o futura, de utilizar otro servidor como almacenamiento, deberás adoptar una capa de abstracción para el acceso a los datos. Existen varias alternativas como ODBC, pero sin duda la opción más recomendable en la actualidad es **PDO**.

El objetivo es que si llegado el momento necesitas cambiar el servidor de base de datos, las modificaciones que debas realizar en tu código sean mínimas. Incluso es posible desarrollar aplicaciones preparadas para utilizar un almacenamiento u otro según se indique en el momento de la ejecución, pero éste no es el objetivo principal de PDO. PDO no abstrae de forma completa el sistema gestor que se utiliza. Por ejemplo, no modifica las sentencias SQL para adaptarlas a las características específicas de cada servidor. Si esto fuera necesario, habría que programar una capa de abstracción completa.

La extensión PDO debe utilizar un driver o controlador específico para el tipo de base de datos que se utilice. Para consultar los controladores disponibles en tu instalación de PHP, puedes utilizar la información que proporciona la función `phpinfo`.

PDO se basa en las características de orientación a objetos de PHP pero, al contrario que la extensión MySQLi, no ofrece un interface de programación dual. Para acceder a las funcionalidades de la extensión tienes que emplear los objetos que ofrece, con sus métodos y propiedades. No existen funciones alternativas.

3.2.1.- Establecimiento de conexiones.

Para establecer una conexión con una base de datos utilizando PDO, debes instanciar un objeto de la clase PDO pasándole los siguientes parámetros (solo el primero es obligatorio):

- ✓ Origen de datos (DSN). Es una cadena de texto que indica qué controlador se va a utilizar y a continuación, separadas por el carácter dos puntos, los parámetros específicos necesarios por el controlador, como por ejemplo el nombre o dirección IP del servidor y el nombre de la base de datos.
- ✓ Nombre de usuario con permisos para establecer la conexión.
- ✓ Contraseña del usuario.
- ✓ Opciones de conexión, almacenadas en forma de array.

Por ejemplo, podemos establecer una conexión con la base de datos 'dwes' creada anteriormente de la siguiente forma:

```
$dwes = new PDO('mysql:host=localhost;dbname=dwes', 'dwes', 'abc123.');
```

Si como en el ejemplo, se utiliza el controlador para MySQL, los parámetros específicos para utilizar en la cadena DSN (separadas unas de otras por el carácter punto y coma) a continuación del prefijo **mysql:** son los siguientes:

- ✓ **`host`**. Nombre o dirección IP del servidor.
- ✓ **`port`**. Número de puerto TCP en el que escucha el servidor.
- ✓ **`dbname`**. Nombre de la base de datos.
- ✓ **`unix_socket`**. Socket de MySQL en sistemas Unix.

Si quisieras indicar al servidor MySQL utilice **codificación UTF-8** para los datos que se transmitan, puedes usar una opción específica de la conexión:

```
$opciones = array(PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8mb4");
$dwes = new PDO('mysql:host=localhost;dbname=dwes', 'dwes', 'abc123.', $opciones);
```

Otra opción para indicar al servidor MySQL que utilice **codificación UTF-8** para los datos que se transmitan es indicarlo en el primer parámetro de la conexión:

```
$dwes = new PDO('mysql:host=localhost;dbname=dwes;charset=utf8mb4', 'dwes', 'abc123.', $opciones);
```

En el manual de PHP puedes consultar más información sobre los controladores existentes, los parámetros de las cadenas DSN y las opciones de conexión particulares de cada uno.
http://es.php.net/manual/es/pdo.drivers.php

Una vez establecida la conexión, puedes utilizar el método **`getAttribute`** para obtener información del estado de la conexión y **`setAttribute`** para modificar algunos parámetros que afectan a la misma. Por ejemplo, para obtener la versión del servidor puedes hacer:

```
$version = $dwes->getAttribute(PDO::ATTR_SERVER_VERSION);
print "Versión: $version";
```

Y si quieres por ejemplo que te devuelva todos los nombres de columnas en mayúsculas:

```
$version = $dwes->setAttribute(PDO::ATTR_CASE, PDO::CASE_UPPER);
```

En el manual de PHP, las páginas de las funciones getAttribute y setAttribute te permiten consultar los posibles parámetros que se aplican a cada una.

http://es.php.net/manual/es/pdo.getattribute.php
http://es.php.net/manual/es/pdo.setattribute.php

3.2.2.- Ejecución de consultas.

Para ejecutar una consulta SQL utilizando PDO, debes diferenciar aquellas sentencias SQL que no devuelven como resultado un conjunto de datos, de aquellas otras que sí lo devuelven.

En el caso de las consultas de acción, como `INSERT`, `DELETE` o `UPDATE`, el método `exec` devuelve el número de registros afectados.

```
$registros = $dwes->exec('DELETE FROM stock WHERE unidades=0');
print "<p>Se han borrado $registros registros.</p>";
```

Si la consulta genera un conjunto de datos, como es el caso de `SELECT`, debes utilizar el método `query`, que devuelve un objeto de la clase `PDOStatement`.

```
$dwes = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123.");
$resultado = $dwes->query("SELECT producto, unidades FROM stock");
```

Por defecto PDO trabaja en modo "`autocommit`", esto es, confirma de forma automática cada sentencia que ejecuta el servidor. Para trabajar con transacciones, PDO incorpora tres métodos:

- ✓ `beginTransaction`. Deshabilita el modo "`autocommit`" y comienza una nueva transacción, que finalizará cuando ejecutes uno de los dos métodos siguientes.
- ✓ `commit`. Confirma la transacción actual.
- ✓ `rollback`. Revierte los cambios llevados a cabo en la transacción actual.

Una vez ejecutado un `commit` o un `rollback`, se volverá al modo de confirmación automática.

```
$ok = true;
$dwes->beginTransaction();
if($dwes->exec('DELETE …') == 0) $ok = false;
if($dwes->exec('UPDATE …') == 0) $ok = false;
…
if ($ok) $dwes->commit();  // Si todo fue bien confirma los cambios
else $dwes->rollback();   //  y si no, los revierte
```

El método "exec" puede devolver el valor booleano FALSE (cuando se produce un error), pero también puede devolver un valor no booleano que se evalúa como FALSE (Un 0). Hay que tener cuidado y usar el operador === para evitar problemas, ya que si no se ve afectado ningún registro por la sentencia SQL, devolverá 0 que se evalúa como false.

```
if($dwes->exec('UPDATE …') === FALSE) $ok = false;
```

Ten en cuenta que no todos los motores soportan transacciones. Tal es el caso, como ya viste, del motor MyISAM de MySQL. En este caso concreto, PDO ejecutará el método `beginTransaction` sin errores, pero naturalmente no será capaz de revertir los cambios si fuera necesario ejecutar un `rollback`.

Si se usan transacciones, las consultas "query" y el commit deben ir dentro del try, y el rollback en el catch. Si las dos instrucciones están correctas se hace el commit, si alguna de ellas fallas, saltaría la excepción y se ejecutaría el rollback.

De una forma similar al anterior ejercicio de transacciones, utiliza PDO para repartir entre las tiendas las tres unidades que figuran en stock del producto con código PAPYRE62GB.
En esta ocasión, para comprobar si los cambios se hacen correctamente en la base de datos y confirmar la transacción, se revisa el número de registros afectados por la ejecución de las consultas. Comprueba que la segunda vez que intentas ejecutarlo no actualizará los datos, tal y como sucedía en el ejercicio equivalente de la extensión MySQLi.

3.2.3.- Obtención y utilización de conjuntos de resultados.

Al igual que con la extensión MySQLi, en PDO tienes varias posibilidades para tratar con el conjunto de resultados devuelto por el método `query`. La más utilizada es el método `fetch` de la clase `PDOStatement`. Este método devuelve un registro del conjunto de resultados, o `false` si ya no quedan registros por recorrer.

```
$dwes = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123.");
$resultado = $dwes->query("SELECT producto, unidades FROM stock");
while ($registro = $resultado->fetch()) {
   echo "Producto ".$registro['producto']."): ".$registro['unidades']."<br />";
}
```

Por defecto, el método `fetch` genera y devuelve, a partir de cada registro, un array con claves numéricas y asociativas. Para cambiar su comportamiento, admite un parámetro opcional que puede tomar uno de los siguientes valores:

- ✓ `PDO::FETCH_ASSOC`. Devuelve solo un array asociativo.
- ✓ `PDO::FETCH_NUM`. Devuelve solo un array con claves numéricas.
- ✓ `PDO::FETCH_BOTH`. Devuelve un array con claves numéricas y asociativas. Es el comportamiento por defecto.
- ✓ `PDO::FETCH_OBJ`. Devuelve un objeto cuyas propiedades se corresponden con los campos del registro.

```
$dwes = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123.");
$resultado = $dwes->query("SELECT producto, unidades FROM stock");
```

```
while ($registro = $resultado->fetch(PDO::FETCH_OBJ)) {
  echo "Producto ".$registro->producto."): ".$registro->unidades."<br />";
}
```

- ✓ `PDO::FETCH_LAZY`. Devuelve tanto el objeto como el array con clave dual anterior.
- ✓ `PDO::FETCH_BOUND`. Devuelve true y asigna los valores del registro a variables, según se indique con el método `bindColumn`. Este método debe ser llamado una vez por cada columna, indicando en cada llamada el número de columna (empezando en 1) y la variable a asignar.

```
$dwes = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123.");
$resultado = $dwes->query("SELECT producto, unidades FROM stock");
$resultado->bindColumn(1, $producto);
$resultado->bindColumn(2, $unidades);
while ($registro = $resultado->fetch(PDO::FETCH_BOUND)) {
    echo "Producto ".$producto."): ".$unidades."<br />";
}
```

- ✓ También se puede utilizar el método **fetchObject()** que obtiene la siguiente fila y la devuelve como un objeto.

Modifica la página web que muestra el stock de un producto en las distintas tiendas, obtenida en un ejercicio anterior utilizando MySQLi, para que use PDO. Omite la gestión de errores, que veremos en el último punto de este tema.

3.2.4.- Consultas preparadas.

Al igual que con MySQLi, también utilizando PDO podemos preparar consultas parametrizadas en el servidor para ejecutarlas de forma repetida. El procedimiento es similar e incluso los métodos a ejecutar tienen prácticamente los mismos nombres.

Para preparar la consulta en el servidor MySQL, deberás utilizar el método `prepare` de la clase PDO. Este método devuelve un objeto de la clase `PDOStatement`. Los parámetros se pueden marcar utilizando signos de interrogación como en el caso anterior.

```
$dwes = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123.");
$consulta = $dwes->prepare('INSERT INTO familia (cod, nombre) VALUES (?, ?)');
```

O también utilizando parámetros con nombre, precediéndolos por el símbolo de dos puntos.

```
$dwes = new PDO("mysql:host=localhost;dbname=dwes", "dwes", "abc123.");
$consulta = $dwes->prepare('INSERT INTO familia (cod, nombre) VALUES (:cod, :nombre)');
```

Antes de ejecutar la consulta hay que asignar un valor a los parámetros utilizando el método `bindParam` de la clase `PDOStatement`. Si utilizas signos de interrogación para marcar los parámetros, el procedimiento es equivalente al método `bindColumn` que acabamos de ver.

```
$cod_producto = "TABLET";
$nombre_producto = "Tablet PC";
$consulta->bindParam(1, $cod_producto);
$consulta->bindParam(2, $nombre_producto);
```

Si utilizas parámetros con nombre, debes indicar ese nombre en la llamada a `bindParam`.

```
$consulta->bindParam(":cod", $cod_producto);
$consulta->bindParam(":nombre", $nombre_producto);
```

Tal y como sucedía con la extensión MySQLi, cuando uses `bindParam` para asignar los parámetros de una consulta preparada, deberás usar siempre variables como en el ejemplo anterior.

Una vez preparada la consulta y enlazados los parámetros con sus valores, se ejecuta la consulta utilizando el método `execute`.

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

Alternativamente, es posible asignar los valores de los parámetros en el momento de ejecutar la consulta, utilizando un array (asociativo o con claves numéricas dependiendo de la forma en que hayas indicado los parámetros) en la llamada a `execute`.

```
$parametros = array(":cod" => "TABLET", ":nombre" => "Tablet PC");
$consulta->execute($parametros);
```

Si la consulta preparada devuelve valores, estos se pueden extraer ejecutando después de **execute()** los métodos **fech()** o **fetchObject()**.

Puedes consultar la información sobre la utilización en PDO de consultas preparadas y la clase PDOStatement en el manual de PHP.

http://es.php.net/manual/es/class.pdostatement.php

Modifica el ejercicio sobre consultas preparadas que realizaste con la extensión MySQLi, el que modificaba el número de unidades de un producto en las distintas tiendas, para que utilice ahora la extensión PDO.

Como puedes comprobar, para obtener la solución se puede aprovechar la mayoría del código existente en el ejercicio anterior.