Introducción al lenguaje PHP

José Luís Sanchez

Contenidos

1. Introducción al lenguaje PHP

1.1. El lenguaje PHP

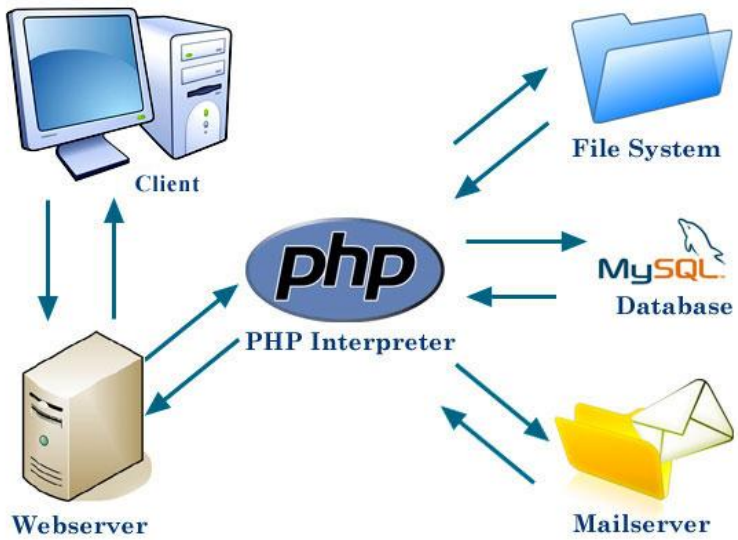
PHP es el acrónimo de "PHP: Hypertext Preprocessor", uno de los más extendidos lenguajes de programación web de los últimos años. Nacido en 1994, se trata de un lenguaje de creación relativamente reciente, aunque con la rapidez con la que evoluciona Internet parezca que ha existido toda la vida. Es un lenguaje que ha tenido una gran aceptación en la comunidad de desarrolladores, debido a la potencia y simplicidad que lo caracterizan, así como al soporte generalizado en la mayoría de los servidores de hosting.

PHP nos permite embeber pequeños fragmentos de código dentro de la página HTML y realizar determinadas acciones de una forma fácil y eficaz, combinando lo que ya sabemos del desarrollo HTML. Es decir, con PHP escribimos scripts dentro del código HTML, con el que se supone que ya estamos familiarizados. Por otra parte, y es aquí donde reside su mayor interés con respecto a los lenguajes pensados para los CGI, PHP ofrece un sinfín de funciones para la explotación de bases de datos de una manera llana y sin complicaciones.

A lo largo de los últimos años se ha tendido a comparar PHP con ASP, otro lenguaje orientado a web de similares características, sin embargo a estas alturas ambos lenguajes han evolucionado de maneras distintas. Mientras que ASP se ha estancado y han salido productos nuevos como .NET para sustituirlo, PHP ha ido mejorando mucho con los años y actualmente su potencia y posibilidades son totalmente distintas, con lo que ha dejado muy atrás la competencia con ASP. Por eso ya no tiene mucho sentido comparar ambos lenguajes.

Qué podemos hacer con PHP

Cualquier cosa. PHP está enfocado principalmente a la programación de scripts del lado del servidor, por lo que se puede hacer cualquier cosa que pueda hacer otro programa CGI, como recopilar datos de formularios, generar páginas con contenidos dinámicos, o enviar y recibir cookies. Aunque PHP puede hacer mucho más.



Existen principalmente tres campos principales donde se usan scripts de PHP.

- Scripts del lado del servidor. Este es el campo más tradicional y el foco principal. Son necesarias tres cosas para que esto funcione. El analizador de PHP (módulo CGI o servidor), un servidor web y un navegador web. Es necesario ejecutar el servidor con una instalación de PHP conectada. Se puede acceder al resultado del programa de PHP con un navegador, viendo la página de PHP a través del servidor. Todo esto se puede ejecutar en su máquina si está experimentado con la programación de PHP.
- Scripts en línea de comandos. Se puede crear un script de PHP y ejecutarlo sin necesidad de un servidor o navegador. Solamente es necesario el analizador de PHP para utilizarlo de esta manera. Este tipo de uso es ideal para scripts que se ejecuten con regularidad empleando cron (en Unix o Linux) o el Planificador de tareas (en Windows). Estos scripts también pueden usarse para tareas simples de procesamiento de texto. Véase la sección Uso de PHP en la línea de comandos para más información.
- Escribir aplicaciones de escritorio. Probablemente PHP no sea el lenguaje más apropiado para crear aplicaciones de escritorio con una interfaz gráfica de usuario, pero si se conoce bien PHP y se quisiera utilizar algunas características avanzadas en aplicaciones del lado del cliente se puede utilizar PHP-GTK para escribir dichos programas. También es posible de esta manera escribir aplicaciones independientes de una plataforma. PHP-GTK es una extensión de PHP no incluida en la distribución principal. Si está interesado en PHP-GTK, puede visitar su propio sitio web.

PHP puede emplearse en todos los sistemas operativos principales, incluyendo Linux, muchas variantes de Unix (incluyendo HP-UX, Solaris y OpenBSD), Microsoft Windows, Mac OS X, RISC OS y probablemente otros más. PHP admite la mayoría de servidores web de hoy en día, incluyendo Apache, IIS, y muchos otros. Esto incluye cualquier servidor web que pueda utilizar el binario de PHP FastCGI, como lighttpd y nginx. PHP funciona tanto como módulo como procesador de CGI. De modo que con PHP se tiene la libertad de elegir el sistema operativo y el servidor web. Además, se tiene la posibilidad de utilizar programación por procedimientos o programación orientada a objetos (POO), o una mezcla de ambas.

Con PHP no se está limitado a generar HTML. Entre las capacidades de PHP se incluyen la creación de imágenes, ficheros PDF e incluso películas Flash (usando libswf y Ming) generadas sobre la marcha. También se puede generar fácilmente cualquier tipo de texto así como XHTML y cualquier otro tipo de fichero XML. PHP puede autogenerar estos ficheros y guardarlos en el sistema de ficheros en vez de imprimirlos en pantalla, creando una caché en servidor para contenido dinámico.

Una de las características más potentes y destacables de PHP es su soporte para un amplio abanico de bases de datos. Escribir una página web con acceso a una base de datos es increíblemente simple utilizando una de las extensiones específicas de bases de datos (p.ej., para mysql), o utilizar una capa de abstracción como PDO, o conectarse a cualquier base de datos que admita el estándar de Conexión Abierta a Bases de Datos por medio de la extensión ODBC. Otras bases de datos podrían utilizar cURL o sockets, como lo hace CouchDB.

PHP también cuenta con soporte para comunicarse con otros servicios usando protocolos tales como LDAP, IMAP, SNMP, NNTP, POP3, HTTP, COM (en Windows) y muchos otros. También se pueden crear sockets de red puros e interactuar usando cualquier otro protocolo. PHP tiene soporte para el intercambio de datos complejos de WDDX entre virtualmente todos los lenguajes de programación web. Y hablando de interconexión, PHP tiene soporte para la instalación de objetos de Java y emplearlos de forma transparente como objetos de PHP.

PHP tiene útiles características de procesamiento de texto, las cuales incluyen las expresiones regulares compatibles con Perl (PCRE), y muchas extensiones y herramientas para el acceso y análisis de documentos XML. PHP estandariza todas las extensiones XML sobre el fundamento sólido de libxml2, y amplía este conjunto de características añadiendo soporte para SimpleXML, XMLReader y XMLWriter.

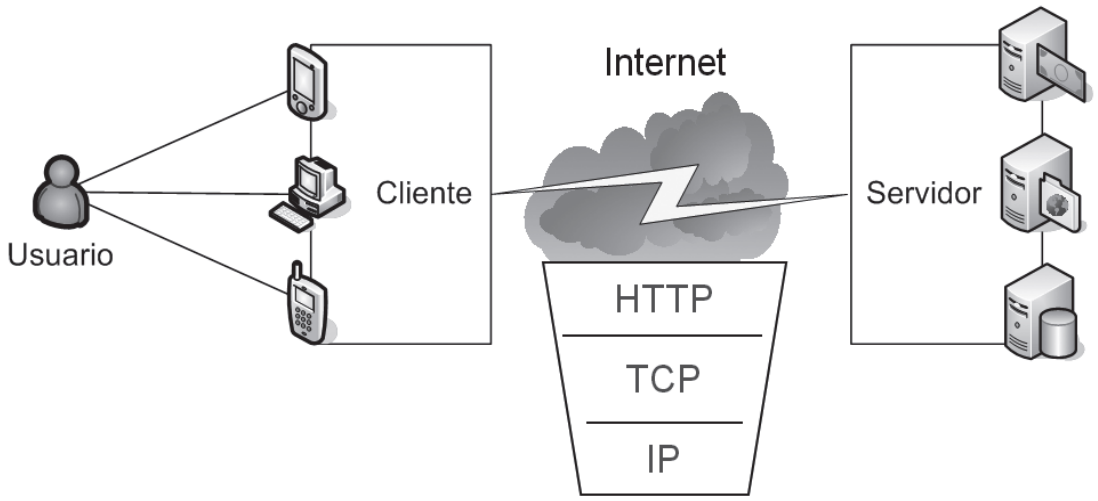
Como se puede apreciar, esta introducción no es suficiente para enumerar todas las características y beneficios que ofrece PHP. Consulte las secciones Instalación de PHP y Referencia de las funciones para una explicación de las extensiones mencionadas aquí.

1.2. Internet y la arquitectura cliente-servidor

La World Wide Web (o la Web, como se conoce comúnmente) representa un universo de información accesible globalmente a través de Internet. Está formada por un conjunto de recursos interconectados que conforman el conocimiento humano actual. El funcionamiento de la Web es posible debido a la coexistencia de una serie de componentes software y hardware. Estos elementos abarcan desde los componentes físicos de Internet (hubs, repetidores, puentes, pasarelas, encaminadores, etc.) y los protocolos de comunicaciones (TCP, IP, HTTP, FTP, SMTP, etc.) hasta la utilización del sistema de nombres de dominio (DNS) para la búsqueda y recuperación de recursos o la utilización de software específico para proveer y consumir dichos recursos.

Modelo de programación en entornos cliente/servidor

En este contexto, el desarrollo en entornos web debe tener en cuenta la distribución de los elementos y la función que tiene cada uno de ellos. La configuración arquitectónica más habitual se basa en el modelo denominado Cliente/Servidor, basado en la idea de servicio, en el que el cliente es un componente consumidor de servicios y el servidor es un proceso proveedor de servicios. Además, esta relación está robustamente cimentada en el intercambio de mensajes como el único elemento de acoplamiento entre ambos.



El agente que solicita la información se denomina cliente, mientras que el componente software que responde a esa solicitud es el que se conoce como servidor. En un proceso habitual el cliente es el que inicia el intercambio de información, solicitando datos al servidor, que responde enviando uno o más flujos de datos al cliente. Además de la transferencia de datos real, este intercambio puede requerir información adicional, como la autenticación del usuario o la identificación del archivo de datos que vayamos a transferir.

Las funcionalidades en los entornos cliente/servidor de la Web suelen estar agrupadas en diferentes capas, cada una centrada en la gestión de un aspecto determinado del sistema web. Si bien es posible encontrarse con divisiones más o menos especializadas, tradicionalmente se

identifican tres tipos de capas fundamentales: capa de presentación, capa de la lógica de negocio y capa de persistencia (almacenamiento de datos). Los modelos arquitectónicos de programación se pueden clasificar en función de en qué lugar, si en el cliente o en el servidor, se sitúan cada una de estas capas. La decisión de dónde se sitúa cada una de estas capas dependerá del entorno de ejecución y, por tanto, de las tecnologías y lenguajes utilizados.

- **Capa de presentación**: esta capa es la que ve el usuario. Le presenta una interfaz gráfica del recurso solicitado y sirve para recoger su interacción. Suele estar situada en el cliente. La programación de esta capa se centra en el formateo de la información enviada por el servidor y la captura de las acciones realizadas por el cliente. En esta capa suelen ubicarse lenguajes como **HTML**, **CSS** o **JavaScript**, dado que suelen ejecutarse en el cliente si bien residen o se crean desde el servidor.
- **Capa de negocio**: es la capa que conoce y gestiona las funcionalidades que esperamos del sistema o aplicación web (lógica o reglas de negocio). Habitualmente es donde se reciben las peticiones del usuario y desde donde se envían las respuestas apropiadas tras el procesamiento de la información proporcionada por el cliente. Al contrario que la capa de presentación, la lógica de negocio puede ser programada tanto en el entorno cliente como en el entorno servidor. Tecnologías como **PHP**, **ASP** e incluso **Ajax** suelen ser las encargadas de esta parte. A menudo es PHP el lenguaje empleado para acceder a la BBDD de la web y proporcionar respuestas al cliente de forma dinámica en forma de contenidos.
- **Capa de persistencia o de datos:** es la capa donde residen los datos y la encargada de acceder a los mismos. Normalmente, está formada por uno o más gestores de bases de datos que realizan todo el proceso de administración de datos, reciben solicitudes de almacenamiento o recuperación de información desde la capa de negocio. En este caso estaríamos hablando de las bases de datos, a menudo en servidores como **MySQL** o **IIS**. El lenguaje de la capa de negocio que reside en el servidor es el que se encarga de interactuar con los datos de la BBDD a petición del cliente.

Lenguajes de programación en entorno servidor

Se entiende por lenguaje de programación en entorno servidor a aquel cuyo código, bien sea como objeto precompilado o bien como código interpretado, es ejecutado por un software específico en el componente que actúa como servidor.

Existen múltiples alternativas a la hora de ejecutar código en el servidor. Una de ellas es utilizar lenguajes de scripting (SSI, LiveWire, ASP, PHP, etc.), cuya característica principal es que el código es interpretado y que se intercala con una plantilla de código HTML con la estructura básica de la página que se envía al cliente. Otra alternativa es el uso de enlaces a programas y componentes ejecutables (CGI, JSP, EJB, etc.), de tal forma que el código ejecutado por el servidor está almacenado en unidades precompiladas y ejecutadas de forma independiente, que generan las páginas enviadas al cliente.

Otras técnicas de programación en el lado del servidor incluyen el uso de estrategias “híbridas” basadas en técnicas de respaldo (denominadas code-behind), como ASP.Net de Microsoft, en el que algunos elementos de código se intercalan con la lógica de presentación mientras que se mantiene la funcionalidad de dichos elementos en ficheros o librerías independientes en el servidor.

1.3. PHP, Apache y MySQL

Para el desarrollo de aplicaciones web y sitios web (scripting del lado del servidor) con PHP se necesitan tres cosas: PHP, un servidor web y un navegador web. Seguramente ya disponga del navegador web y, dependiendo de la configuración del sistema operativo, quizá ya tenga un servidor web (p.ej. Apache en Linux y MacOS X; IIS en Windows). También puede alquilar espacio web en una empresa. De esta forma, no se necesita instalar nada, solo tiene que escribir los scripts de PHP, subirlos al servidor que alquile y ver los resultados en su navegador.

Xampp

Para poder desarrollar código PHP en local, es decir, en nuestra propia máquina, la mayoría de los desarrolladores tienden a hacerlo sobre servidores de tipo Apache y contra bases de datos como MySQL dada su gratuidad y su facilidad para ser instalado y usado. Una de las soluciones más sencillas a la hora de instalar todas estas herramientas en nuestros equipos es mediante los paquetes XAMPP, un servidor independiente de plataforma, software libre, que consiste principalmente en la base de datos MySQL, el servidor web Apache y los intérpretes para lenguajes de script: PHP y Perl. El nombre proviene del acrónimo de X (para cualquiera de los diferentes sistemas operativos), Apache, MySQL, PHP, Perl. También incluye otros módulos como OpenSSL y phpMyAdmin así como un servidor para aplicaciones de Java Tomcat, un servidor de correo Mercury y un servidor FTP entre otras herramientas.

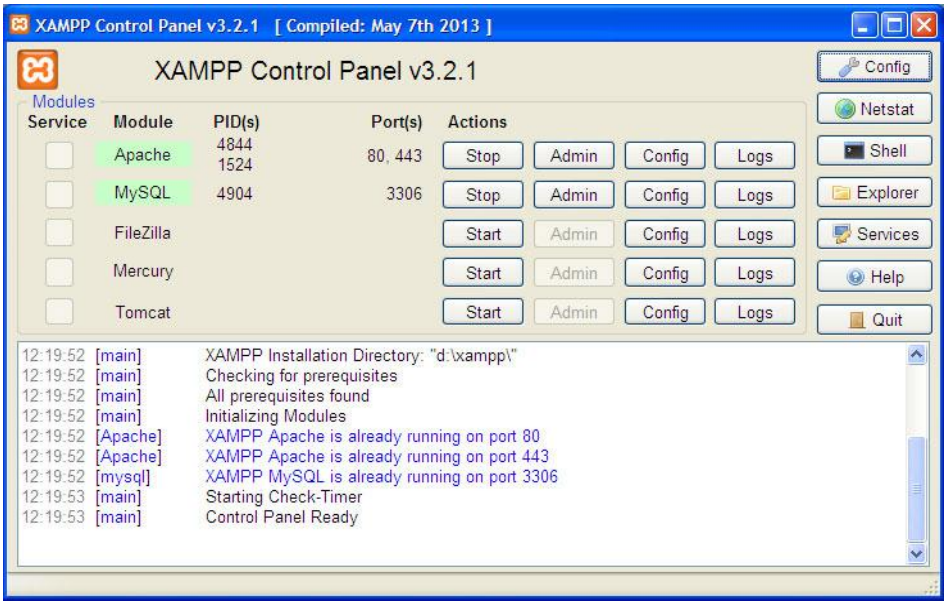
El programa está liberado bajo la licencia GNU y actúa como un servidor web libre, fácil de usar y capaz de interpretar páginas dinámicas. Actualmente XAMPP está disponible para Microsoft Windows, GNU/Linux, Solaris y Mac OS X.

Instalación de Xampp

El proceso de instalación de Xampp no podría ser más sencillo, basta con acceder a la web https://www.apachefriends.org/es y seleccionar la versión que más se ajuste a nuestro sistema operativo (Windows, Linux o MacOS).

Una vez descargada seguimos los pasos para instalarla en nuestro sistema, los cuales son muy sencillos (ver Anexo 1: instalación y configuración de XAMPP en Windows).

Dentro de la carpeta de Xampp o desde el menú de inicio podremos encontrar un acceso directo al panel de control de Xampp, que, dependiendo de la versión, presentará un aspecto similar al siguiente:



Desde aquí podremos activar o desactivar algunos de los servicios más característicos como el servidor MySQL. Para trabajar con PHP sólo tendremos que poner en marcha el servidor Apache, quien será el encargado de interpretar el código PHP.

Si todo ha ido bien la forma más cómoda y rápida de comprobar si nuestro servidor web está funcional será accediendo a la dirección http://localhost desde nuestro navegador web, donde deberíamos encontrar una página similar a la siguiente:

Desde esta web podremos configurar diversas opciones de nuestro servidor, comprobar las versiones instaladas de nuestro PHP y Perl o incluso crear o gestionar nuestras BBDD de tipo MySQL desde el gestor online PHPMySQL.

Una de las páginas más interesantes es la llamada phpinfo(), que podemos encontrar a la izquierda, que no es si no un acceso directo a una función simple de PHP que nos devuelve todo tipo de información acerca de nuestro servidor (módulos instalados, versiones, tipo de servidor, etc).

Toda la información que vemos al acceder a localhostes el resultado de múltiples páginas publicadas en nuestro servidor local, lo que se traduce a archivos que se ubican en la carpeta **xampp\htdocs**. Aunque esta ruta puede ser cambiada, en principio será también esta la carpeta donde colocaremos todas aquellas páginas desarrolladas en PHP (o en otros lenguajes) que deseemos ver funcionando. Veremos un ejemplo más práctico en el siguiente apartado.

En el caso que nos ocupa, PHP, es también el servidor web (Apache) quien tiene la capacidad de entender e interpretar nuesto código PHP gracias a contar con el correspondiente módulo instalado, de manera que el código PHP escrito en páginas que se muestren a través de nuestro servidor será ejecutado por este y su resulta presentado al cliente.

1.4. Uso de PHP con HTML

Como comentamos al principio de este capítulo, con frecuencia el código PHP se incluye dentro de código HTML. El intérprete de PHP simplemente analiza el texto del archivo y lo presenta tal cual lo lee, como si lo ignorase,hasta que encuentra uno de los caracteres especiales que delimitan el inicio de código PHP, que a menudo es <?php. Entonces el intérprete ejecutará todo el código que encuentre hasta localizarla etiqueta de fin de código, ?>, momento en el que el intérprete sigue ignorando el código siguiente. Cuando hacemos esto el archivo deberá usar la extensión PHP en lugar de HTML, aunque esto se podría recalibrar en la configuración del servidor web.

Veamos un sencillo ejemplo:

```
<!DOCTYPE HTML PUBLIC ”-//W3C//DTD HTML 4.01 Transitional//EN”
    ”http://www.w3.org/TR/html4/loose.dtd”>
<html>
    <head>
        <title>Ejemplo</title>
    </head>
    <body>

        <?php
            echo ”¡Hola mundo, soy un script de PHP!”;
        ?>

    </body>
</html>
```

En este ejemplo podemos ver la estructura básica de una página HTML dentro de cuya zona body encontramos código PHP, concretamente la instrucción echo. Esta instrucción se limita a presentar por pantalla el mensaje entrecomillado. Volveremos a utilizarla más adelante.

Para poder ver cómo se ejecuta deberemos guardar este archivo con extensión php dentro del directorio htdocs que hemos mencionado anteriormente. Al hacerlo de esta manera y accediendo desde la dirección local veríamos una página con el mensaje:

¡Hola mundo, soy un script de PHP!

Si mirásemos el código fuente de este ejemplo desde el navegador no encontraríamos rastro alguno de PHP en él, simplemente veríamos el HTML y, dentro de la zona body, la mencionada frase. Esto sucede porque el intérprete de PHP de nuestro servidor hace precisamente eso, interpretar el código, por lo que en lugar de mostrarlo tal cual se ve arriba, lo ejecuta, presentando el resultado de las instrucciones que encuentra.

Si por el contrario abriésemos nuestro archivo en el navegador como si fuera cualquier documento, no a través del servidor, nuestro navegador nos mostraría el siguiente mensaje en pantalla:

<?php echo ”¡Hola mundo, soy un script de PHP!”; ?>

Es decir, lo muestra literalmente porque los navegadores web no saben interpretar el PHP que encuentra dentro. Igualmente si mirásemos el código fuente de esa página desde el navegador lo veríamos tal cual se ha escrito en el ejemplo de arriba.

Por otra parte también podemos crear archivos con sólo código PHP, sin HTML y hacer uso de ellos de múltiples maneras. Más adelante estudiaremos los conceptos de inclusión y requerimiento y en el capítulo final de este manual veremos también el uso de plantillas, lo que confiere a la combinación HTML + PHP una nueva y muy flexible dimensión.

Comentarios en PHP

A menudo es necesario incluir comentarios en nuestro código fuente. PHP dispone para ello de dos formas que, además, son bastante populares entre otros lenguajes:

```
// Para comentarios de una sola línea
/* Para descripciones y comentarios de más de una línea */
```

Sin embargo las etiquetas para comentarios que se usan en HTML (<!-- /-->) no son aplicables al código PHP si este se incluye dentro de este tipo de documentos.

2. Sintaxis básica

En este capítulo abordaremos las herramientas básicas para programar con PHP, especialmente en lo referente a almacenamiento y tratamiento de datos en tiempo de ejecución con cosas como las variables, las constantes y los arrays, el uso de los operadores para realizar comparaciones u operaciones con dichos datos y veremos también algunas instrucciones y funciones básicas para poder trabajar con estos elementos antes de adentrarnos en las estructuras de control.

Comenzaremos este capítulo con algunas instrucciones básicas para que podamos probar lo que vayamos viendo en nuestras propias páginas.

2.1. Instrucciones básicas para impresión por pantalla

Posiblemente el conjunto de instrucciones más importante para comenzar sean aquellas que nos permiten imprimir mensajes por pantalla. Tal es el caso de las instrucciones echo y print.

Echo

La instrucción **echo** permite mostrar por pantalla cadenas de texto, expresiones matemáticas, variables o una combinación de todo junto. En PHP las cadenas de texto se definen entre comillas (simples o dobles).

```
echo “Hola Mundo”;

echo ”Esto abarca
múltiple líneas. Los saltos de línea también
se mostrarán”;
```

La instrucción echo permite concatenar varios parámetros, es decir, unir unos con otros para presentar un mensaje con todos ellos:

```
echo “Hola”.” Mundo”; // mostrará Hola Mundo
```

Aunque la instrucción echo se puede usar también acompañada por paréntesis, hay que tener en cuenta que no se trata de una función y por lo tanto no se puede comportar como tal.

En cuanto al uso de las comillas dobles o simples, podemos hacer un uso indistinto entre ellas si bien conviene tener en cuenta que existe cierta distinción. Principalmente haremos uso de uno u otro tipo en el caso de que queremos imprimir una de ellas por pantalla:

```
Echo ‘Este texto viene “entrecomillado”’;
```

Más adelante veremos también que su uso dependerá a la hora de trabajar por ejemplo con variables.

Print

La instrucción **print** es muy similar a echo, de hecho mucha gente la usa de idéntica manera, pero aporta además la posibilidad de usarla como si fuera una función, algo que exploraremos más adelante.

```
print(“Hola mundo”);

print “print() también funciona sin paréntesis.”;

print “ Esto abarca
múltiple líneas. Los saltos de línea también
se mostrarán”;
```

Al igual que echo, la instrucción print también puede usarse con paréntesis sin ser una función, sin embargo a diferencia de echo, print sí puede usarse como si lo fuera:

```
$var = print “Hola mundo”; // a la variable $var se le asigna la
instrucción print junto con el texto que deberá imprimir
```

Ya que echo no se comporta como una función, el siguiente código no es válido:

```
$variable ? echo 'verdadero' : echo 'falso';
```

Sin embargo, el siguiente código sí funcionará:

```
$variable ? print 'verdadero' : print 'falso'; // esta línea de código
se comporta de forma similar a como lo hace la instrucción if:
comprobará el valor de $variable  e imprimirá verdadero si es true o
falso en caso contrario


Echo $variable ? 'verdadero' : 'falso'; // esta alternativa también
sería posible; lo que hace es un echo del resultado de la comparación
ternaria
```

Al igual que en el caso de la instrucción echo, podremos hacer uso de las comillas simples o dobles basándonos en el mismo principio.

Diferencias:

- print imprime una cadena, echo puede imprimir más de una separadas por coma
- print devuelve un valor int que según la documentación siempre es 1, por lo que puede ser utilizado en expresiones mientras que echo es tipo void, no hay valor devuelto y no puede ser utilizado en expresiones.
- Según algunas fuentes, como w3schools, echo es ligeramente más rápido que print.

2.2. Variables y constantes

Como en todos los lenguajes de programación PHP permite almacenar datos de distintos tipos en memoria mediante lo que llamaremos variables y constantes. La diferencia estriba en el uso que se le pretende dar a los valores que almacenan cada uno de estos elementos. Mientras que el valor de una constante se plantea como precisamente algo constante, que no cambiará a lo largo de todo el flujo de nuestra aplicación, el de las variables será un valor que en principio sí podrá hacerlo.

Variables

En el caso de PHP las variables se declaran precedidas siempre por el símbolo dólar ($), son sensibles a las mayúsculas y no pueden comenzar con un número. También vale la pena tener en cuenta que en PHP no es necesario declarar las variables con antelación a su uso. Para asignar un valor a una variable podemos usar el operador = tal como podemos ver en los siguientes ejemplos:

```
$nombre= "Luis"; //crea una variable llamada $nombre que contiene el
valor de texto Luis

$edad=35; //crea una variable llamada $edad que contiene el valor
entero 35
```

Las variables tienen un ámbito local (estudiaremos esto con más detenimiento a la hora de trabajar con funciones), aunque también pueden ser accedidas de forma global (con ámbito general a todo nuestro código) si anteponemos a su llamada el término **global**.

En el próximo apartado veremos los tipos de datos que podemos almacenar en nuestras variables.

Constantes

PHP cuenta también con lo que se conoce como constantes, un elemento similar a las variables pero que siempre contienen el mismo valor a lo largo de toda la ejecución del programa y que suelen usarse con un ámbito global aunque también podemos hacerlo en otros ámbitos:

Así podemos definir una constante y asignarle un valor:

```
define("IVA", "21%"); //nos permite definir una constante global
llamada IVA con el valor 21%


define("SALUDO", "¡Bienvenidos a PHP!");
echo SALUDO;

const VALOR_MINIMO = 0.0; //nos permite definir una constante de
ámbito local a la clase en la que se ubique llamada VALOR_MINIMO, no
global
```

PHP posee además una serie de valores constantes propios, conocidos como constantes predefinidas, capaces de mostrar valores específicos propios de la configuración de nuestro sistema. Algunos ejemplos:

```
PHP_VERSION // Versión del intérprete de PHP de nuestro servidor
PHP_OS      // Nos revela el sistema operativo de nuestro servidor
PHP_LIBDIR  // Ruta en la que se almacenan las librerías de PHP
```

PHP también cuenta con lo que se conoce como constantes mágicas, constantes capaces de mostrar valores dependiendo de dónde se ejecuten. Su sintaxis es algo diferente, ya que se acompañan por la izquierda y por la derecha de dos guiones bajos seguidos:

```
__LINE__    // nos indica el número de línea actual
__FILE__    // nos presenta el nombre y la ruta completa del archivo
__CLASS__   // nos muestra el nombre de la clase en la que nos
encontramos
```

Variables Superglobales

Además de las variables y las constantes, PHP incorpora un elemento mixto llamado variables superglobales. Se trata de un conjunto de variables internas del sistema de tipo array asociativo, cuyo nombre y estructura ya vienen definidos desde PHP, capaces de contener una gran cantidad de información útil para trabajar y que están disponibles siempre en todos los ámbitos, por lo que no es necesario emplear **global $variable;** para acceder a ellas dentro de las funciones o métodos. Las variables superglobales disponibles en PHP son las siguientes:

- **$GLOBALS**: hace referencia a todas las variables disponibles en el ámbito global
- **$_SERVER**: concentra información relacionada con el entorno del servidor y de ejecución[1]
- **$_GET**: relacionadas con la recepción de información desde otras páginas enviadas mediante el método GET
- **$_POST**: relacionadas con la recepción de información desde otras páginas enviadas mediante el método POST
- **$_FILES**: contiene información acerca de los archivos cargados a una página
- **$_COOKIE**: contiene información acerca de las cookies
- **$_SESSION**: contiene información acerca de las sesiones de usuario abiertas en la página
- **$_REQUEST**: contiene el contenido de *$_GET*, *$_POST* y *$_COOKIE*.
- **$_ENV**: contiene información acerca de las variables definidas en el entorno bajo el que está siendo ejecutado el intérprete PHP (definidas directamente sobre el sistema)

[1] Para información sobre sus índices: http://php.net/manual/es/reserved.variables.server.php

2.3. Tipos de datos

En PHP no es necesario definir el tipo de una variable porque ya se define de forma automática al asignársele un valor. No obstante es interesante distinguir los tipos de datos con los que este lenguaje puede trabajar:

Boolean

Expresa un valor de verdad y se usa principalmente para la evaluación de condiciones. Puede ser TRUE o FALSE.

```
$bool = TRUE; // un valor booleano con valor verdadero
```

Integer

Define un número entero (no decimal) que va desde -2.147.483.648 y +2.147.483.647. Requiere como mínimo un dígito y no puede contener espacios en blanco. Permite, además, ser especificado en formato decimal (base 10), hexadecimal (base 16 con prefijo 0x) u octal (base 8 con prefijo 0) y ser precedido opcionalmente por los símbolos + ó -.

```
$num = 1234; // numero entero
$num = -123; // un numero negativo
$num = 0123; // numero octal (equivalente al 83 decimal)
$num = 0x1A; // numero hexadecimal (equivalente al 26 decimal)
$a=5;
$b=10;
print $a+$b // imprime 15 por pantalla
```

Float

Los números de punto flotante (también conocidos como “flotantes”, “dobles” o “números reales”) representan cualquier tipo de número decimal y pueden ser especificados usando cualquiera de las siguientes sintaxis:

```
$a = 1.234;
$b = 1.2e3;
$c = 7E-10;
```

El tamaño de un flotante depende de la plataforma, aunque un valor común consiste en un máximo de ~1.8e308 con una precisión de aproximadamente 14 dígitos decimales (lo que es un valor de 64 bits en formato IEEE).

Se pueden encontrar algunas referencias al tipo “double”, pero debe considerarse como el mismo que float. Su nombre existe solo por razones históricas.

String

El tipo string representa tanto para representar caracteres simples como cadenas de caracteres sin tamaño definido y de carácter alfanumérico. PHP entenderá que una variable es de tipo string siempre que su valor venga entrecomillado. Para ello podemos utilizar indistintamente tanto comillas simples como dobles, lo cual nos permite utilizar unas dentro de las otras sin cerrar el string. Sin embargo PHP interpreta de manera distinta el uso de las comillas dobles y las simples: los strings con comillas dobles pueden sustituir ciertos símbolos por variables, mientras que las comillas simples muestra el contenido de forma literal.

```
$texto1 = "Hola mundo";
$texto2 = ' Hola mundo ';
$texto3 = "Dentro de un string podemos utilizar 'texto entrecomillado' sin problemas";
$nombre= "Jose Luis";
Echo "mi nombre es $nombre"; //muestra la frase seguida del nombre almacenado en la variable $nombre
Echo 'mi nombre es $nombre'; //muestra mi nombre es $nombre
```

Dentro de un string con entrecomillado doble podemos utilizar determinados símbolos de escape para representar ciertos elementos no visuales como retornos de carro, fin de línea o tabulaciones. Para se suele usar la barra invertida (\) como elemento de escape:

\n	nueva línea
\r	retorno de carro
\t	tabulación
\\	barra invertida
\$	símbolo del dólar, para evitar confusión con variables

La barra invertida también nos servirá en el caso de las comillas si necesitamos usar las mismas que definen un string:

```
$enlace= "<a href=\"http://www.google.es\">Link</a>";
Echo "<p>Puedes accede a Google a través del siguiente $enlace</p>";
echo "esto no es una variable, sólo el símbolo del dólar \$";
```

Índices de string

Si pensamos en las cadenas como una sucesión de caracteres en un orden determinado, podemos querer acceder a parte de los caracteres. Esto es posible mediante el uso de los corchetes ([ ]) y un índice numérico correspondiente a la posición del carácter buscado dentro del string.

```
$cadena= "hola mundo";
Echo "la primera letra de mi variable es $cadena[0]";
```

Heredoc

Otra forma de delimitar un string es mediante la sintaxis **heredoc**: <<<. Después de este operador, se deberá proporcionar un identificador y justo después una nueva línea. A continuación va el propio string, y para cerrar la notación se pone el mismo identificador.

El identificador de cierre debe empezar en la primera columna de la nueva línea. Asimismo, el identificador debe seguir las mismas reglas de nomenclatura de las etiquetas en PHP: debe contener solo caracteres alfanuméricos y guiones bajos, y debe empezar con un carácter alfabético o un guion bajo.

```
<?php
$str = <<<EOD
Ejemplo de una cadena
expandida en varias líneas
empleando la sintaxis heredoc.
EOD;

/* Un ejemplo más complejo con variables. */
class foo
{
    var $foo;
    var $bar;

    function foo()
    {
        $this->foo = 'Foo';
        $this->bar = array('Bar1', 'Bar2', 'Bar3');
    }
}

$foo = new foo();
$nombre = 'MiNombre';

echo <<<EOT
Mi nombre es "$nombre". Estoy escribiendo un poco de $foo->foo.
Ahora, estoy escribiendo un poco de {$foo->bar[1]}.
Esto debería mostrar una 'A' mayúscula: \x41
EOT;

?>
```

Es muy importante señalar que la línea con el identificador de cierre no debe contener ningún otro carácter, excepto un punto y coma (;). Esto, en especial, significa que el identificador *no debe estar sangrado*, y que no debe existir ningún espacio ni tabulación antes o después del punto y coma.

Array

Un **array** en PHP es en realidad un mapa ordenado. Un mapa es un tipo de datos que asocia valores con claves. Por su flexibilidad este tipo resulta especialmente óptimo para representar distintas estructuras de datos, por lo que puede ser usado como una matriz real, una lista (vector), una tabla asociativa (caso particular de implementación de un mapa), diccionarios,

colecciones, pilas, colas y probablemente más. Dado que PHP admite crear arrays con otros arrays como valor resulta fácil simular incluso estructuras de árboles.

Un array puede ser creado usando el constructor del lenguaje array():

```
$dias= array (“lunes”, “martes”, “miércoles”);
Echo “Hoy es $dias[0]”; // Hoy es lunes
$vector= array (1, 2, 3, 4, 5);
```

En el caso de un array asociativo éste toma un cierto número de parejas clave =>valor como argumentos:

```
$array = array(
    ”foo” => ”bar”,
    ”bar” => ”foo”,
);

// a partir de PHP 5.4 también se puede usar [] en lugar de ()
$array = [
    ”foo” => ”bar”,
    ”bar” => ”foo”,
];
```

List

List () es un una instrucción especialmente útil para trabajar con arrays, dado que está orientada a asociar valores y variables contenidos en otros array en una sola sentencia. Veámos su uso con un sencillo ejemplo:

```
$info = array('café', 'marrón', 'cafeína');

// Listando todas las variables
list($drink, $color, $power) = $info;
echo ”El $drink es $color y la $power lo hace especial.\n”;

// Listando algunas variables
list($drink, , $power) = $info;
echo ”El $drink tiene $power.\n”;

// Mostramos sólo la tercera
list( , , $power) = $info;
echo ”Necesito $power!\n”;
```

Volveremos a los array más adelante para estudiar su uso con distintas estructuras de datos y sus funciones más importantes.

Object

Un objeto es un tipo de dato capaz de almacenar tanto datos como los procedimientos necesarios para trabajar con dichos datos. Los objetos deben ser declarados de forma explícita, lo cual hacemos mediante lo que se conoce como una clase del objeto. Una clase es una estructura que contiene propiedades y métodos para el adecuado uso de los objetos. A partir de ella instanciamos objetos con esas mismas propiedades y métodos. Veamos un ejemplo de declaración de un objeto:

```
class Coche {
      var $marca;
      function Coche() {
          $this->marca = "SEAT"; //define la marca del coche
      }
}
$panda = new Coche();// crear un objeto

echo $panda->marca;// imprime el valor de la propiedad marca: SEAT
```

Dado su enorme importancia, estudiaremos este tipo de dato y la programación orientada objetos con mayor detenimiento más adelante.

NULL

El tipo de datos NULL sirve para otorgar a una variable sólo el valor de nulo o NULL, lo que representa que esa variable no tiene asignado ningún valor. Este tipo de dato es frecuentemente utilizado para vaciar de forma rápida una variable.

Resource

Las variables de tipo resource no son en realidad un tipo de dato si no que son variables especiales que contiene una referencia a un recurso externo. Los recursos son creados y usados por funciones especiales. Veremos más sobre este tipo de dato cuando hablemos de bases de datos o acceso a ficheros.

2.4. Expresiones y operadores

Las expresiones son las piedras de construcción más importantes de PHP. En PHP casi todo lo que se escribe es una expresión. La manera más simple y acertada de definir lo que es una expresión es “cualquier cosa que tiene un valor”.

Asignación

Las formas más básicas de expresiones son las constantes y las variables. Cuando se escribe “$a = 5”, se está asignando '5' a $a. '5', obviamente, tiene el valor 5, o en otras palabras, '5' es una expresión con el valor de 5 (en este caso, '5' es una constante entera).

Después de esta asignación, se espera que el valor de $a sea 5 también, por lo que si se escribe $b = $a, se espera que esto se comporte tal como si se escribiera $b = 5. En otras palabras, $a es también una expresión con el valor 5. Si todo funciona bien, esto es exactamente lo que sucederá.

```
$a= 5;
$b= $a; // Ahora $b también vale 5
$a = ($b = 4) + 5; // $a es igual a 9 y $b ha sido definido a 4.
```

Un ejemplo de expresiones algo más complejo son las funciones. Por ejemplo, considere la siguiente función:

```
function foo ()
{
      return 5;
}
```

Asumiendo para este ejemplo que está familiarizado con el concepto de función (si no tranquilo, lo estudiaremos más adelante), entenderemos que escribir *$c = foo()* es esencialmente igual que escribir *$c = 5*. Y está en lo cierto. Las funciones son expresiones con el valor de sus valores devueltos. Ya que *foo()* devuelve 5, el valor de la expresión '*foo()*' es 5. Normalmente las funciones no sólo devuelven un valor estático, sino que computan algo.

Como hemos visto en el apartado anterior, PHP soporta cuatro tipos de valores escalares: valores enteros (integer), valores de coma (punto) flotante (float), valores de cadena (string) y valores booleanos (boolean) - (valores escalares son aquellos que no se pueden descomponer en piezas más pequeñas, a diferencia de las matrices, por ejemplo). PHP también soporta dos tipos compuestos (no escalares): matrices (arrays) y objetos. Cada uno de estos tipos de valores puede ser asignado a variables o devueltos desde funciones.

En conjunto con el operador básico de asignación, existen “operadores combinados” para todas las operaciones de aritmética binaria y de cadenas, que le permiten usar un valor en una expresión y luego definir su valor como el resultado de esa expresión. Por ejemplo:

```
$a = 3;
$a += 5; // define $a como 8, equivalente a hacer: $a = $a + 5;
$b = "¡Hola ";
```

```
$b .= "a todos!"; // define $b como "¡Hola a todos!", tal como $b = $b
. "a todos!";
```

Asignación múltiple

PHP lleva las expresiones mucho más allá, de la misma manera que lo hacen otros lenguajes. PHP es un lenguaje orientado a expresiones, en el sentido de que casi todo es una expresión. Considerando el ejemplo anterior de $a= 5 y $b=5, en PHP también podríamos escribir *$b= $a=* 5, ya que las asignaciones se analizan de derecha a izquierda.

Pre y post incremento

Otro buen ejemplo de orientación a expresiones es el pre- y post-incremento y decremento. Los usuarios de PHP y de otros muchos lenguajes pueden estar familiarizados con la notación *variable++* y *variable*--. Éstos son los operadores de incremento y decremento. En PHP, al igual que en C, hay dos tipos de incrementos - pre-incremento y post-incremento.

Ejemplo	Nombre	Efecto
`++$a`	Pre-incremento	Incrementa $a en uno, y luego devuelve $a
`$a++`	Post-incremento	Devuelve $a, y luego incrementa $a en uno
`--$a`	Pre-decremento	Decrementa $a en uno, luego devuelve $a
`$a--`	Post-decremento	Devuelve $a, luego decrementa $a en uno

Ambas alternativas esencialmente incrementan o decrementan la variable, y el efecto sobre la variable es idéntico. La diferencia estriba en cuándo se evalúa la expresión: con *++$variable*, se incrementa la variable antes de leer su valor, de ahí el nombre de 'pre-incremento). El post-incremento, escrito *$variable*++ evalúa el valor original de $variable antes de que sea incrementado (PHP incrementa la variable después de leer su valor).

Operadores

Un operador es un elemento a lo que se entrega uno o más valores (operandos que pueden, a su vez ser otras expresiones) y produce otro valor (de modo que la construcción misma se convierte en una expresión). Bajo esta definición encontraríamos tanto los operadores matemáticos que nos permiten operar con números o texto como muchos que veremos a continuación.

Precedencia de Operadores

La precedencia de un operador indica en qué orden deberán realizarse las operaciones que representan en una expresión. Por ejemplo, en la expresión 1 + 5 * 3, la respuesta es 16 y no 18, ya que el operador de multiplicación (*) tiene una mayor precedencia que el operador de adición (+). Los paréntesis pueden ser usados para marcar la precedencia, si resulta necesario. Por ejemplo: (1 + 5) * 3 evalúa a 18. Si la precedencia de los operadores es la misma, se utiliza una asociación de izquierda a derecha.

Operadores aritméticos

Los operadores aritméticos son los que nos van a permitir desarrollar expresiones matemáticas entre valores y variables. Encontraremos los siguientes:

Ejemplo	Nombre	Resultado
`-$a`	Negación	El opuesto de $a.
`$a + $b`	Adición	Suma de $a y $b.
`$a - $b`	Substracción	Diferencia entre $a y $b.
`$a * $b`	Multiplicación	Producto de $a y $b.
`$a / $b`	División	Cociente de $a y $b.
`$a % $b`	Módulo	Resto de $a dividido por $b.

El operador de división (“/”) devuelve un valor flotante en todos los casos, incluso si los dos operandos son enteros (o cadenas que son convertidas a enteros).

Nota: El resto de $a % $b es negativo para valores negativos de $a.

Operadores de texto

PHP también cuenta, al igual que en otro lenguajes, con operadores específicamente diseñados para trabajar con texto. Al igual que en Java y JavaScript podemos utilizar el + para unir o concatenar cadenas de texto, aunque en PHP se suele usar más el punto (.).

```
$cadena1= “Hola”;
$cadena2= “mundo”;
$cadenafinal= $cadena1. “ “ . $cadena2;
echo $cadenafinal; //imprime Hola mundo
```

También podemos hacer uso del operador (.=) que sirve para añadir texto a una cadena existente:

```
$cadena1= "Hola";
$cadena2= "mundo";
$cadena1= $cadena1. $cadena2; // Hola mundo
$cadena1.=$cadena2;// sería equivalente a la operación anterior
```

Operadores de comparación

Los operadores de comparación, como su nombre indica, permiten ser usados en expresiones para comparar dos valores. Estas expresiones evalúan si algo es **FALSE** (falso) o **TRUE** (verdadero).

Ejemplo	Nombre	Resultado
`$a == $b`	Igual	**TRUE** si $a es igual a $b.
`$a === $b`	Idéntico	**TRUE** si $a es igual a $b, y son del mismo tipo. (A partir de PHP 4)
`$a != $b`	Diferente	**TRUE** si $a no es igual a $b.
`$a <> $b`	Diferente	**TRUE** si $a no es igual a $b.
`$a !== $b`	No idénticos	**TRUE** si $a no es igual a $b, o si no son del mismo tipo. (A partir de PHP 4)
`$a < $b`	Menor que	**TRUE** si $a es estrictamente menor que $b.
`$a > $b`	Mayor que	**TRUE** si $a es estrictamente mayor que $b.
`$a <= $b`	Menor o igual que	**TRUE** si $a es menor o igual que $b.
`$a >= $b`	Mayor o igual que	**TRUE** si $a es mayor o igual que $b.

Si se compara un entero con una cadena, la cadena es convertida a un número. Si se comparan dos cadenas numéricas, éstas serán comparadas como enteros.

Los operadores de comparación, como hemos dicho, se usan mayormente dentro de ejecuciones condicionales, tales como las sentencias ***if*** *o* ***switch*** *que también estudiaremos más adelante* y también en lo que se conoce como el operador condicional ternario que ya nos habíamos encontrado en un ejemplo anterior:

```
$variable ? print 'verdadero' : print 'falso'; // esta línea de código se comporta de forma similar a como lo hace la instrucción if que se ve a continuación:

If ($variable) {   // es equivalente a ($variable==TRUE)
      Print 'verdadero'
}else{
      Print 'falso'
} //Esta operación comprobará el valor de $variable e imprimirá verdadero si es true o falso en caso contrario
```

Operadores lógicos

Los operadores lógicos se usan principalmente para evaluar o comparar no solo valores sin también expresiones completas. En PHP existen los siguientes operadores:

Ejemplo	Nombre	Resultado
`$a and $b`	And (y)	TRUE si tanto $a como $b son TRUE.
`$a or $b`	Or (o inclusivo)	TRUE si cualquiera de $a o $b es TRUE.
`$a xor $b`	Xor (o exclusivo)	TRUE si $a o $b es TRUE, pero no ambos.
`! $a`	Not (no)	TRUE si $a no es TRUE.
`$a && $b`	And (y)	TRUE si tanto $a como $b son TRUE.
`$a || $b`	Or (o inclusivo)	TRUE si cualquiera de $a o $b es TRUE.

Operadores para arrays

Los arrays cuentan con un conjunto de operadores específicos para operar con sus elementos:

Ejemplo	Nombre	Resultado
`$a + $b`	Unión	Unión de $a y $b (anexa al final de $a)
`$a == $b`	Igualdad	TRUE si $a y $b tienen las mismas parejas llave/valor.
`$a === $b`	Identidad	TRUE si $a y $b tienen las mismas parejas llave/valor en el mismo orden y de los mismos tipos.
`$a != $b`	No-igualdad	TRUE si $a no es igual a $b.
`$a <> $b`	No-igualdad	TRUE si $a no es igual a $b.
`$a !== $b`	No-identidad	TRUE si $a no es idéntico a $b.

El operador + adiciona la matriz del lado derecho a la del lado izquierdo, al mismo tiempo que cualquier llave duplicada NO es sobrescrita.

```
$a = array("manzana", "banana");
$b = array("banana", "cereza", "limón");

$a + $b; // "manzana", "banana", "cereza", "limón"
```

Los elementos de las matrices son considerados equivalentes en la comparación si éstos tienen la misma clave y valor.

```
$a = array("manzana", "banana");
$b = array(1 =>"banana", "0" =>"manzana");

var_dump($a == $b); // bool(true)
var_dump($a === $b); // bool(false)
```

Los siguientes casting de tipos están permitidos:

- (int), (integer) - forzado a integer
- (bool), (boolean) - forzado a boolean
- (float), (double), (real) - forzado a float
- (string) - forzado a string
- (array) - forzado a array
- (object) - forzado a object
- (unset) - forzado a NULL (PHP 5)

2.5. Funciones relacionadas con el tratamiento de variables

En PHP existen una serie de funciones específicamente diseñadas para tratar con las variables, sus valores y sus tipos. En este apartado estudiaremos algunas de las más importantes.

Var_dump ()

Esta función muestra información estructurada sobre una o más expresiones incluyendo su tipo y valor. Las matrices y los objetos son explorados recursivamente con valores sangrados para mostrar su estructura.

```
$b = 3.1;
$c = true;
var_dump($b, $c);
```

El resultado del ejemplo sería:

```
Float(3.1)
Bool(true)
```

Print_r ()

Funciona de manera similar a como lo hace var_dump, imprimiendo información legible para humanos sobre una variable.

```
<?php
$a = array ('a' => 'manzana', 'b' => 'banana', 'c' => array ('x', 'y',
 'z'));
print_r ($a);
?>
```

El resultado del ejemplo sería:

```
Array
(
    [a] => manzana
    [b] => banana
    [c] => Array
        (
            [0] => x
            [1] => y
            [2] => z
        )
)
```

Isset ()

Determina si una variable está definida y no es NULL. Isset() devolverá FALSE si prueba una variable que ha sido definida como NULL. Si son pasados varios parámetros, entonces isset() devolverá TRUE únicamente si todos los parámetros están definidos. La evaluación se realiza de izquierda a derecha y se detiene tan pronto como se encuentre una variable no definida.

```
$var = '';

// Esto evaluará a TRUE así que el texto se imprimirá.
if (isset($var)) {
    echo ”Esta variable está definida, así que se imprimirá”;
}

// En los siguientes ejemplo usaremos var_dump para imprimir
// el valor devuelto por isset().

$a = ”prueba”;
$b = ”otraprueba”;

var_dump(isset($a));      // TRUE
var_dump(isset($a, $b)); // TRUE

unset ($a); //Libera la variable

var_dump(isset($a));     // FALSE
var_dump(isset($a, $b)); // FALSE

$foo = NULL;
var_dump(isset($foo));   // FALSE
```

Unset ()

unset() destruye las variables especificadas liberando así la memoria que ocupaban. El comportamiento de unset() dentro de una función puede variar dependiendo de qué tipo de variable se está tratando de destruir. Tal como se ha podido observar en el ejemplo anterior, al destruir una variable con unset(), esta arrojará un valor de false si se evalúa con iseet().

```
$a = ”prueba”;
unset ($a); //Libera la variable
var_dump(isset($a));     // FALSE
```

Gettype ()

Esta función permite averiguar el tipo de dato almacenado en una variable. Puede devolver los siguientes valores: integer, double, Boolean, string, array, object, resource, NULL y unknown type.

```
$data = array(1, 1., NULL, new stdClass, 'foo');

foreach ($data as $value) {
    echo gettype($value), ”\n”;
}
```

El resultado del ejemplo sería algo similar a:

```
Integer
Double
NULL
Object
String
```

Settype ()

Permite convertir el tipo de una variable al especificado en la función de entre los siguientes: integer, double, Boolean, string, array u object. A partir de la versión de PHP 4.2 también se pueden usar los tipos NULL, float en lugar de double, int en lugar de integer y bool en lugar de boolean.

```
$foo = ”5bar”; // cadena
$bar = true;   // booleano

settype($foo, ”integer”); // $foo es ahora 5   (entero)
settype($bar, ”string”);  // $bar es ahora ”1” (cadena)
```

Empty ()

Determina si una variable es considerada vacía. Una variable se considera vacía si no existe o si su valor es igual a FALSE. Empty() no genera una advertencia si la variable no existe. A partir de la versión de PHP 5.5 esta función soporta expresiones en vez de únicamente variables.

```
$var = 0;

// Se evalúa a true ya que $var está vacia
if (empty($var)) {
    echo '$var es o bien 0, vacía, o no se encuentra definida en absol
uto';
}

// Se evalúa como true ya que $var está definida
if (isset($var)) {
    echo '$var está definida a pesar que está vacía';
}
```

Is_integer (), is_double (), is_string (), intval(), strval ()

Todas estas funciones operan de idéntica forma: devuelven true si la variable evaluada coincide con el tipo que invoca la función.

```
$numero_enter0= 0;
If (is_integer() ($numero_entero)) {
      Echo “numero_entero es del tipo integer”;
}
```

Intval(), doubleval(), strval()

De forma similar a como opera settype(), estas funciones convierten el valor de una variable al tipo indicado en la función, aunque estas funciones no permiten la conversión a tipos object o array.

```
$cadena="232";
Echo "el tipo de la variable cadena es ".gettype ($cadena)."<br>";
$numero= intval ($cadena);
Echo ("el número es $numero");
```

3. Estructuras de control

Todo script PHP está construido en base a una serie de sentencias. Una sentencia puede ser una asignación, una llamada de función, un ciclo, una sentencia condicional o incluso una sentencia que no hace nada (una sentencia vacía). Las sentencias generalmente finalizan con un punto y coma. Adicionalmente, las sentencias pueden agruparse en un conjunto de sentencias, encapsulándolas entre corchetes. Un grupo de sentencias es una sentencia por sí misma también. Sin embargo se hace difícil imaginar un programa mínimamente funcional basado sólo en la ejecución secuencial de sentencias simples, es decir, sin contener algún tipo de estructura de control en su interior.

En casi todos los lenguajes de programación existen estas estructuras de control, que no son más que una serie de instrucciones orientadas a controlar el flujo de ejecución de los programas y que se basan en conceptos algorítmicos de gran funcionalidad. Tal es su importancia que, independientemente de lo complejo que sea el programa, siempre encontraremos alguna de estas instrucciones funcionando en el núcleo de su código fuente.

En este apartado estudiaremos las más importantes divididas por categorías según su funcionalidad.

3.1. Estructuras de selección

Las estructuras de selección son un conjunto de instrucciones que nos van a permitir ejecutar un grupo u otro de sentencias dependiendo de una o más condiciones.

If

La instrucción **if** es una de las características más importantes de muchos lenguajes, incluido PHP. Permite la ejecución condicional de fragmentos de código. PHP dispone de una estructura *if* que es similar a la de C, JavaScript y otros muchos lenguajes:

if (expr) {

sentencia }

Como se describe en la sección sobre expresiones, la expresión es evaluada a su valor booleano. Si la expresión se evalúa como TRUE, PHP ejecutará la sentencia y si se evalúa como FALSE la ignorará.

El siguiente ejemplo mostraría a es mayor que b si *$a* es mayor que *$b*:

```
if ($a > $b) {
  echo "a es mayor que b";
}
```

A menudo se desea tener más de una sentencia para ser ejecutada condicionalmente. Por supuesto, no hay necesidad de envolver cada sentencia con una cláusula *if*. En cambio, se pueden agrupar varias sentencias en un grupo de sentencias. Por ejemplo, este código mostraría a es mayor que b si $a es mayor que $b y entonces asignaría el valor de $a a $b:

```
if ($a > $b) {
  echo "a es mayor que b";
  $b = $a;
}
```

Las condiciones que es capaz de evaluar la instrucción if puede ser cualquier tipo de expresión, lo que significa que podría contener varias igualdades o desigualdades e incluso el resultado devuelto por alguna función.

```
if (($a > $b)&& ($b> $c)) {
    echo "a es mayor que b y que c";
}

if (isset($var)) {
    echo '$var está definida';
}
```

Las sentencias *if* pueden anidarse dentro de otra sentencias *if* infinitamente, lo cual provee completa flexibilidad para la ejecución condicional de diferentes partes del programa.

La instrucción if cuenta con una sintaxis alternativa en sustitución de las llaves que, para según qué ocasión, puede resultar realmente útil:

If (condicion):

Sentencias;

Endif;

Este tipo de sintaxis es altamente útil en situaciones en las que deseamos combinar PHP con otros lenguajes para, por ejemplo, ejecutar un fragmento de HTML u otro en caso de que se cumpla una condición:

```
<?php if (condition): ?>

      <p>Se cumple la condición</p>

<?php else: ?>

      <p>No se cumple la condición</p>

<?php endif ?>
```

Else

Con frecuencia se desea ejecutar una sentencia si una determinada condición se cumple y una sentencia diferente si la condición no se cumple. Esto es para lo que sirve *else*. El *else* extiende una sentencia *if* para ejecutar una sentencia en caso que la expresión en la sentencia *if* se evalúe como FALSE. Por ejemplo, el siguiente código deberá mostrar a es mayor que b si *$a* es mayor que *$b* y a NO es mayor que b en el caso contrario:

```
if ($a > $b) {
  echo "a es mayor que b";
} else {
  echo "a NO es mayor que b";
}
```

La sentencia *else* sólo es ejecutada si la expresión *if* es evaluada como FALSE. Para un mayor nivel de ramificación en la toma de decisiones podríamos hablar de anidamiento, es decir, incluir varias instrucciones if-else unas dentro de otras:

```
if ($a > $b) {
      echo "a es mayor que b";
} else {
      if ($a < $b)  {
            echo "a  es menor que b";
      }else {
            echo "a y b son iguales";
}
```

En casos como este podríamos usar también la instrucción elseif para simplificar la longitud del código.

Elseif

elseif, como su nombre sugiere, es una combinación de if y else. Del mismo modo que else, extiende una sentencia if para ejecutar una sentencia diferente en caso que la expresión if original se evalúe como FALSE. Sin embargo, a diferencia de else, esa expresión alternativa sólo se ejecutará si la expresión condicional del elseif se evalúa como TRUE. Por ejemplo, el siguiente código debe mostrar a es mayor que b, a es igual que b o a es menor que b:

```
if ($a > $b) {
    echo "a es mayor que b";
} elseif ($a == $b) {
    echo "a es igual que b";
} else {
    echo "a es menor que b";
}
```

Puede haber varios elseif dentro de la misma sentencia if. La primera expresión elseif (si hay alguna) que se evalúe como TRUE sería ejecutada. En PHP también se puede escribir 'else if'

(con dos palabras) y el comportamiento sería idéntico al de 'elseif' (con una sola palabra). El significado sintáctico es ligeramente diferente (si se está familiarizado con C, este es el mismo comportamiento) pero la conclusión es que ambos resultarían tener exactamente el mismo comportamiento.

La sentencia elseif es ejecutada solamente si la expresión if precedente y cualquiera de las expresiones elseif precedentes son evaluadas como FALSE, y la expresión elseif actual se evalúa como TRUE.

Nota: Tenga en cuenta que elseif y else if serán considerados exactamente iguales solamente cuando se utilizan llaves como en el ejemplo anterior.

Switch

La sentencia *switch* es similar a una serie de sentencias if-elseif en la misma expresión. En muchas ocasiones, es posible que se quiera comparar la misma variable (o expresión) con muchos valores diferentes, y ejecutar una parte de código distinta dependiendo de a qué valor es igual. Para esto es exactamente la expresión *switch*.

Los dos ejemplos siguientes son dos formas diferentes de escribir lo mismo, uno con una serie de sentencias *if* y *elseif*, y el otro usando la sentencia *switch*:

```
if ($i == 0) {
    echo "i es igual a 0";
} elseif ($i == 1) {
    echo "i es igual a 1";
} elseif ($i == 2) {
    echo "i es igual a 2";
}

switch ($i) {
    case 0:
        echo "i es igual a 0";
        break;
    case 1:
        echo "i es igual a 1";
        break;
    case 2:
        echo "i es igual a 2";
        break;
}
```

La instrucción **break**, que veremos más adelante, tiene por misión finalizar el flujo de la secuencia, es decir, dar por terminada la ejecución de la instrucción switch.

Ejemplo 2: la estructura switch permite el uso de Strings:

```
switch ($i) {
    case "manzana":
        echo "i es una manzana";
        break;
    case "barra":
        echo "i es una barra";
        break;
    case "pastel":
        echo "i es un pastel";
        break;
}
```

Es importante entender cómo la sentencia switch es ejecutada con el fin de evitar errores. La sentencia switch ejecuta línea por línea (en realidad, sentencia por sentencia). Al principio, ningún código es ejecutado. Sólo cuando una sentencia case es encontrada con un valor que coincide con el valor de la sentencia switch, PHP comienza a ejecutar la sentencias. PHP continúa ejecutando las sentencias hasta el final del bloque switch, o hasta la primera vez que vea una sentencia break. Si no se escribe una sentencia break al final de la lista de sentencias de un caso, PHP seguirá ejecutando las sentencias del caso siguiente.

También existe la instrucción **default**, que se utiliza para el caso en el que la expresión evaluada no coincida con ninguna de las sentencias case. Por ejemplo:

```
switch ($i) {
    case 0:
        echo "i es igual a 0";
        break;
    case 1:
        echo "i es igual a 1";
        break;
    case 2:
        echo "i es igual a 2";
        break;
    default:
       echo "i no es igual a 0, 1 ni 2";
}
```

La expresión case puede ser cualquier expresión que se evalúa como un tipo simple, es decir, entero o números de punto flotante y strings. Los arrays u objetos no se pueden utilizar aquí a menos que sean desreferenciados a un tipo simple.

Con Switch también podemos definir intervalos:

```
$a=45;

switch ($a) {
      case ($a>40):
            echo "es >40";
            break;
      case ($a<40):
```

```
            echo "es <40";
            break;
      default:
            echo "es 40";
}
```

3.2. Estructuras de recorrido y bucles

El siguiente conjunto de instrucciones permite repetir la ejecución de un grupo de sentencias dependiendo de una o más condiciones. A la ejecución repetitiva de sentencias se le llama comúnmente bucle.

Aunque con sutiles diferencias sintácticas, en casi todos los lenguajes se suele distinguir entre las siguientes estructuras de bucle dependiendo de cómo o en qué momento se evalúa la condición:

- Ejecutar un grupo de sentencias mientras se cumpla una condición (While)
- Ejecutar un grupo de sentencias hasta que se cumpla una condición (Do-While)
- Ejecutar un grupo de sentencias un número determinado de veces (For-Next)

Es importante tener siempre bajo control la condición a evaluar para evitar caer en lo que se conoce como un bucle infinito, es decir, una ejecución de sentencias repetida sin condición de parada alguna.

While

Los bucles while son el tipo más sencillo de bucle en PHP. Se comportan igual que su contrapartida en C. La forma básica de una sentencia while es:

while (expr) {

sentencias;

}

El significado de una sentencia while es simple. Le dice a PHP que ejecute las sentencias anidadas, tanto como la expresión while se evalúe como TRUE. El valor de la expresión es verificado cada vez al inicio del bucle, por lo que incluso si este valor cambia durante la ejecución de las sentencias anidadas, la ejecución no se detendrá hasta el final de la iteración (cada vez que PHP ejecuta las sentencias contenidas en el bucle es una iteración). A veces, si la expresión while se evalúa como FALSE desde el principio, las sentencias anidadas no se ejecutarán ni siquiera una vez.

```
$i = 1;
while ($i <= 10) {
    echo $i++; /* el valor presentado sería
                  $i antes del incremento
```

```
                    (post-incremento) */
}
```

La sentencia While cuenta, al igual que con la sentencia if, una sintaxis alternativa para sustituir el uso de las llaves:

while (expr):

sentencias;

endwhile;

Un ejemplo, idéntico al anterior, con este tipo de sintaxis alternativa:

```
$i = 1;
 while ($i <= 10):
     print $i;
$i++;
 endwhile;
```

Do while

Los bucles do-while son muy similares a los bucles while, excepto que la expresión verdadera es verificada al final de cada iteración en lugar que al principio. La diferencia principal con los bucles while es que está garantizado que corra la primera iteración de un bucle do-while (la expresión verdadera sólo es verificada al final de la iteración), mientras que no necesariamente va a correr con un bucle while regular (la expresión verdadera es verificada al principio de cada iteración, si se evalúa como FALSE justo desde el comienzo, la ejecución del bucle terminaría inmediatamente).

Hay una sola sintaxis para bucles do-while:

```
$i = 0;
do {
    echo $i;
} while ($i > 0);
```

El bucle de arriba se ejecutaría exactamente una sola vez, ya que después de la primera iteración, cuando la expresión verdadera es verificada, se evalúa como **FALSE** (*$i* no es mayor que 0) y termina la ejecución del bucle.

For

Los bucles for son los más complejos en PHP. Esta instrucción es capaz de recorrer estructuras de forma repetitiva desde un valor inicial hasta otro final según un incremento que también podemos definir. La sintaxis de un bucle for es:

for (expr1; expr2; expr3) {

Sentencias;

}

La primera expresión (expr1) es evaluada (ejecutada) una vez incondicionalmente al comienzo del bucle. Normalmente es la expresión utilizada para indicar el valor inicial del recorrido.

En el comienzo de cada iteración, se evalúa *expr2*, es decir, la expresión que definirá el final del bucle. Si se evalúa como **TRUE**, el bucle continúa, repitiéndose una vez más, y se ejecutan las sentencias anidadas. Si se evalúa como **FALSE**, finaliza la ejecución del bucle.

Al final de cada iteración, se evalúa (ejecuta) *expr3*, que es la expresión que definirá el incremento (o decremento) que permite una nueva iteración en el bucle.

Cada una de las expresiones puede estar vacía o contener múltiples expresiones separadas por comas. En *expr2*, todas las expresiones separadas por una coma son evaluadas, pero el resultado se toma de la última parte. Que *expr2* esté vacía significa que el bucle debería ser corrido indefinidamente (PHP implícitamente lo considera como **TRUE**, como en C). Esto puede no ser tan inútil como se pudiera pensar, ya que muchas veces se debe terminar el bucle usando una sentencia condicional **break** en lugar de utilizar la expresión verdadera del *for*.

Considere los siguientes ejemplos. Todos ellos muestran los números del 1 al 10:

```
/* ejemplo 1 */

for ($i = 1; $i <= 10; $i++) {
    echo $i;
}

/* ejemplo 2 */

for ($i = 1; ; $i++) {
    if ($i > 10) {
        break;
    }
    echo $i;
}

/* ejemplo 3 */

$i = 1;
for (; ; ) {
    if ($i > 10) {
        break;
    }
    echo $i;
    $i++;
}

/* ejemplo 4 */

for ($i = 1, $j = 0; $i <= 10; $j += $i, print $i, $i++);
```

Por supuesto, el primer ejemplo parece ser el más claro, pero se puede observar que la posibilidad de usar expresiones vacías en los bucles *for* resulta útil en muchas ocasiones.

En lo que respecta a la tercera expresión de la instrucción, la que marca el incremento, también puede incrementarse de forma no unitaria:

```
/* ejemplo 5: incrementa de 10 en 10 presentado en una tabla HTML*/

$precio = 5;

echo "<table border=\"1\" align=\"center\">";
echo "<tr><th>Cantidad</th>";
echo "<th>Precio</th></tr>";
for ( $counter = 10; $counter <= 100; $counter += 10) {
        echo "<tr><td>";
        echo $counter;
        echo "</td><td>";
        echo $brush_price * $counter;
        echo "</td></tr>";
}
echo "</table>";
```

E incluso también podemos realizar bucles basados en decrementos:

```
/* ejemplo 6: mostrar los números del 10 al 1 */

for ($i = 10; $i >0; $i--) {
    echo $i;
}
```

PHP también admite la sintaxis alternativa de los dos puntos para bucles for.

for (expr1; expr2; expr3):

sentencias;

endfor;

También es bastante común usar for para iterar arrays como en el siguiente ejemplo, si bien la instrucción foreach resulta más óptima para dicho cometido.

```
/* Ejemplo 7: array con datos a modificar al recorrer el bucle */

$gente = array(
    array('nombre' => 'Kalle', DNI => 856412),
    array('nombre' => 'Pierre', 'DNI' => 215863)
);

for($i = 0; $i < count($gente); $i++) {
    $people[$i]['DNI'] = mt_rand(000000, 999999);
}
```

El código anterior puede ser lento, debido a que el tamaño del array se capta en cada iteración mediante la llamada a la función count() que cuenta la cantidad de elementos del array principal.

Foreach

La instrucción foreach proporciona un modo sencillo de iterar sobre arrays. Foreach funciona sólo sobre arrays y objetos, y emitirá un error al intentar usarlo con una variable de un tipo diferente de datos o una variable no inicializada. Pese a recordar a la instrucción for por su nombre, su comportamiento no es en absoluto idéntico, dado que no requiere ni expresión de finalización ni de incremento ya que foreach se limita a recorrer los elementos del array de uno en uno hasta el último (a menos que indiquemos otro fin dentro del bucle).

Existen dos sintaxis:

foreach (expresión_array as $valor) {

sentencias;}

foreach (expresión_array as $clave => $valor) {

sentencias;}

La primera forma recorre el array dado por expresión_array. En cada iteración, el valor del elemento actual se asigna a $valor y el puntero interno del array avanza una posición (así en la próxima iteración se estará observando el siguiente elemento).

La segunda forma además asigna la clave del elemento actual a la variable $clave en cada iteración.

```
/* ejemplo 1: sólo valor*/
$vector = array(1, 2, 3, 17);

foreach($vector as $valor) {
   print "Valor actual de \$vector: $valor.\n";
}

/*Hacer esto con un bucle for */

for($i==0;$i<=count($vector);$i++) {
   print "Valor actual de \$vector: $vector[$i].\n";
}
```

```
/* ejemplo 2: valor (con clave impresa) */
$a = array(1, 2, 3, 17);

$i = 0; /* sólo para propósitos demostrativos */
```

```
foreach($a as $v) {
   print "\$a[$i] => $v.\n";
   $i++;
}


/* ejemplo 3: clave y valor */
$a = array(
   "uno" =>1,
   "dos" =>2,
   "tres" =>3,
   "diecisiete" =>17
);


foreach($a as $k =>$v) {
   print "\$a[$k] => $v.\n";
}

/* ejemplo 4: matriz multi-dimensional */

$a[0][0] = "a";
$a[0][1] = "b";
$a[1][0] = "y";
$a[1][1] = "z";

foreach($a as $v1) {
   foreach ($v1 as $v2) {
       print "$v2\n";
   }
}
```

Cuando foreach inicia su ejecución, el puntero interno del array se pone automáticamente en el primer elemento del array. Esto significa que no es necesario llamar la función reset() antes de un bucle foreach. Ya que foreach depende el puntero de array interno, cambiar éste dentro del bucle puede conducir a un comportamiento inesperado.

Utilizando arrays anidados con list()

PHP 5.5 añade la posibilidad de recorrer un array de arrays y utilizar el array interior en las variables del bucle proporcionando **list()** como el valor.

Por ejemplo:

```
$array = [
    [1, 2],
    [3, 4],
];

foreach ($array as list($a, $b)) {// $a contiene el primer elemento
```

```
del array interior y $b contiene el segundo elemento.
    echo "A: $a; B: $b\n";
}
```

El resultado del ejemplo sería:

```
A: 1; B: 2
A: 3; B: 4
```

```
//Otro ejemplo del uso de list:

$info = array('café', 'marrón', 'cafeína');

// Enumerar todas las variables
list($bebida, $color, $energía) = $info;
echo "El $bebida es $color y la $energía lo hace especial.\n";


// list() no funciona con cadenas
list($bar) = "abcde";
var_dump($bar); // NULL
```

Advertencia: en PHP 5, list() asigna los valores empezando desde el parámetro más a la derecha. En PHP 7, list() empieza desde el parámetro más a la izquierda.

3.3. Instrucciones para ruptura, finalización, salto y retorno

En PHP existen varias instrucciones pertenecientes a la categoría de estructuras de control especializadas no tanto en repetir o recorrer variables si no en lo contrario, permitir terminar o escapar de una iteración y devolver valores o el control de flujo a otra parte del código. Vemos algunas de ellas.

Break

Break es una instrucción diseñada para escapar de un proceso iterativo, tal como vimos al estudiar la instrucción switch, aunque es aplicable a cualquier otra estructura iterativa como for, foreach, while y do-while.

```
Vector = array (1, 2, 3, NULL, 4, 5);


foreach($vector as $valor) {
      if ($valor==NULL){
            break;
      }
      print "Valor actual: $valor.\n";
}
/*Recorre $vector hasta que encuentra el valor nulo, o sea, 1, 2, 3*/
```

Break además acepta un argumento numérico opcional el cual indica de cuantas estructuras anidadas encerradas se debe salir:

```
/* Usando el argumento opcional. */

$i = 0;
while (++$i) {
    switch ($i) {
    case 5:
        echo "En 5<br />\n";
        break 1; /* Sólo sale del switch. */
    case 10:
        echo "En 10; saliendo<br />\n";
        break 2;  /* Sale del switch y del while */
    default:
        break;
    }
}
```

Continue

continue se utiliza dentro de las estructuras iterativas para saltar el resto de la iteración actual del bucle (por ejemplo un paso) y continuar la ejecución en la evaluación de la condición, para luego comenzar la siguiente iteración.

```
$vector = array (1, 2, 3, NULL, 4, 5);


foreach($vector as $valor) {
      if ($valor==NULL){
            continue;
      }
   print "Valor actual: $valor.\n";
}
/*Recorre $vector hasta que encuentra el valor nulo, cuya impresión se
salta o sea que imprime: 1, 2, 3, 4, 5*/
```

Continue acepta un argumento numérico opcional, que indica a cuántos niveles de bucles encerrados se ha de saltar al final. El valor por omisión es *1*, por lo que salta al final del bucle actual.

En PHP, la sentencia switch se considera una estructura iterativa para los propósitos de *continue*. C*ontinue* se comporta igual que *break* (cuando no se proporcionan argumentos). Si un *switch* está dentro de un bucle, *continue 3* continuará con la siguiente iteración del bucle externo.

```
while (list($clave, $valor) = each($arr)) {
    if (!($clave % 2)) { // saltar los miembros impares
        continue;
    }
    hacer_algo($valor);
}

$i = 0;
while ($i++ < 5) {
    echo "Exterior<br />\n";
    while (1) {
        echo "Medio<br />\n";
        while (1) {
            echo "Interior<br />\n";
            continue 3;
        }
        echo "Esto nunca se imprimirá.<br />\n";
    }
    echo "Ni esto tampoco.<br />\n";
}
```

Goto

El operador *goto* puede ser usado para saltar a otra sección en el programa. El punto de destino es especificado mediante una etiqueta seguida de dos puntos y la instrucción es dada como *goto* seguida de la etiqueta del destino deseado.

```
goto a;
echo 'Foo';

a:
echo 'Bar';

/* El resultado sería Bar */
```

Dado el peligro que puede suponer contar con una instrucción capaz de saltar a cualquier sitio de nuestro código, algo que favorece la mala estructuración de un programa, esta instrucción no está exenta de restricciones. La etiqueta de destino debe estar dentro del mismo fichero y contexto, lo que significa que no se puede saltar fuera de una función o método, ni se puede saltar dentro de uno. Tampoco se puede saltar dentro de cualquier clase de estructura de bucle o switch. Se puede saltar fuera de estos y un uso común es utilizar un *goto* en lugar de un *break* multi-nivel.

```
for($i=0,$j=50; $i<100; $i++) {
  while($j--) {
    if($j==17) goto end;
  }
}
echo "i = $i";
end:
echo 'j alcanzó 17';

/*El resultado sería j alcanzó 17 */
```

Return

return devuelve el control del programa a el modulo que lo invoca. La ejecución vuelve a la siguiente declaración después del módulo que lo invoca.

Si se llama desde una función, la sentencia return termina inmediatamente la ejecución de la función actual y devuelve su argumento como el valor de la llamada a la función. return también termina la ejecución de una sentencia eval() o un archivo de script.

Si se llama desde el ámbito global, entonces la ejecución del script actual se termina. Si el archivo script actual fue incluido o requerido con include o require, entonces el control es pasado de regreso al archivo que hizo la llamada. Además, si el archivo script actual fue incluido con include, entonces el valor dado a return será retornado como el valor de la llamada include. Si return es llamado desde dentro del fichero del script principal, entonces termina la ejecución del script.

Cabe señalar que dado que return es un constructor del lenguaje y no una función, los paréntesis que rodean sus argumentos no son necesarios. Es común no utilizarlos, y en realidad se debería hacer así a fin de que PHP tenga menos trabajo que hacer en este caso. Si no se suministra un parámetro, entonces el paréntesis debe omitirse y NULL será retornado. Llamadas a return con paréntesis pero sin argumentos resultarán en un error del intérprete.

Nunca se deben usar paréntesis alrededor de la variable de retorno cuando se retorna por referencia, ya que esto no funcionará. Sólo se pueden retornar variables por referencia, no el resultado de una sentencia. Si se utiliza return ($a); entonces no se está retornando una variable, sino el resultado de la expresión ($a) (el cual es, por supuesto, el valor de $a).

Veremos actuar a la instrucción return con frecuencia cuando usemos funciones en el próximo capítulo.

Exit y Die

Ambas instrucciones son equivalentes y funcionan de forma idéntica: finalizan la ejecución del script sin devolver salida o valor algunos, a diferencia de return, salvo si se le adjunta un valor de status con el fin de informar del motivo de la finalización del script (por ejemplo en caso de error).

```
exit ($status)
```

Si `status` es un valor integer, ese valor será usado también como el status de salida y no se mostrará. Los status de salida deben estar en el rango 0 a 254, el status de salida 255 es reservado por PHP y no debe ser usado. El status 0 es usado para finalizar el programa de forma satisfactoria. Lo habitual, en cualquier caso, es devolver alguna cadena que explique la razón de la finalización de forma más clara.

```
$nombre_archivo = '/ruta/hacia/archivo-datos';
$archivo = fopen($nombre_archivo, 'r')
    or exit("no se pudo abrir el archivo ($nombre_archivo)");

/*En este ejemplo se intenta abrir un archivo; si resulta imposible
por la razón que sea, el script finalizará e imprimirá el mensaje
devuelto por exit*/
```

Sin embargo se considera poco profesional declinar la responsabilidad de un error en cualquiera de estas dos instrucciones, es decir, no siendo debidamente capturadas, al menos de cara al usuario final. Hablaremos de esto en un capítulo posterior.

4. Funciones

Las funciones son una de las herramientas más importantes en todo buen lenguaje de programación. Se trata de agrupaciones de instrucciones con una funcionalidad concreta a las que podemos invocar en momentos muy específicos de nuestro código. El uso de funciones ofrece, además, un mayor grado de estructuración en el desarrollo de nuestros programas dado que nos permite modularizar determinadas funcionalidades, algunas de las cuales pueden necesitar ser reutilizadas en más de una ocasión. Así, una función será una estructura desarrollada en una parte de nuestro código mientras que esta función será llamada o invocada desde donde se la necesite.

Sintaxis general de una función:

```
Function nombre (parámetro1, parámetro2, ..., parámetroN) {
      Instrucciones;
}
```

Junto al concepto de función debemos también definir el de argumento. Los argumentos son valores que podemos pasar a las funciones y que estas deberán procesar en su interior. Tras dicho proceso, las variables normalmente devolverán un resultado en forma de uno o más valores el cual puede ser utilizado de varias formas a su regreso. En los siguientes ejemplos invocamos funciones desde la instrucción echo que ya conocemos, lo que provocará que se impriman los resultados de esas funciones, operaciones realizadas sobre sus argumentos:

```
Echo sqrt(9); //imprime la raíz cuadrada de 9, que será 3
Echo rand(10,20); // imprime un numero aleatorio entre 10 y 20
```

Existen otras formas de invocar a una función, como por ejemplo pasando su resultado a una variable (u otra función), en una condición de un bucle o, en caso de no devolver resultado valor, simplemente llamándola y dejando que haga su trabajo.

Al igual que en la mayoría de los lenguajes de alto nivel, PHP cuenta con dos tipos de funciones: las creadas por los usuarios y las incluidas en el propio sistema. Con el primer grupo disponemos de las herramientas necesarias para que construir nuestras propias funciones personalizadas; con el segundo grupo contamos con una inmensa cantidad de herramientas útiles, a modo de instrucciones del lenguaje, capaces de ampliar notablemente la potencia del lenguaje.

En este capítulo estudiaremos primero las funciones definidas por el usuario y sus características más importantes y después revisaremos algunas de las más interesantes que incluye PHP, especialmente para el tratamiento de datos tan comunes como el texto, las expresiones regulares y los arrays.

Otra práctica común en el mundo de la programación consiste en la agrupación de funciones útiles en lo que se conoce como librerías, las cuales serán después incluidas en nuestros

programas mediante el uso de las instrucciones include y require que también estudiaremos más adelante.

4.1. Funciones definidas por el usuario

Como hemos dicho en la introducción, las funciones creadas por los usuarios resultan de lo más versátiles para programar, especialmente en caso de agrupar acciones de uso más o menos sistemático.

En el siguiente ejemplo podemos ver una función que, llamada al comienzo de nuestro script, construye el encabezado de nuestro documento HTML y coloca el título que queremos a la página:

```
Function creaCabecera($titulo) {
      return "<html><head><title>$titulo</title></head>";
}
```

Esta sería su llamada:

```
<?php

Echo creaCabecera("mi web de ejemplo"); //invocamos a la función
llamada creaCabecera y le pasamos un string para formar el título

¿>
```

El resultado será el siguiente:

```
<html><head><title>mi web de ejemplo</title></head>
```

Los nombres de las funciones siguen las mismas reglas que otras etiquetas de PHP. Un nombre de función válido comienza con una letra o guion bajo, seguido de cualquier número de letras, números, o guiones bajos. Los nombres de funciones se pueden llamar con mayúsculas o minúsculas, aunque es una buena costumbre llamar a las funciones tal y como aparecen en su definición. Una práctica bastante común a la hora de bautizar a una función (y también más adelante a los métodos) será mediante el estilo camelCase, es decir, juntar varias palabras que definen su funcionalidad con la primera letra de cada una en mayúsculas salvo la primera (crearCabecera, calcularFactura, etc).

Devolver valores

Una función puede devolver un resultado mediante la instrucción return () o no hacerlo porque su cometido se lleva a cabo dentro de la propia función. Una variante de la función anterior podría ser así:

```
Function creaCabecera ($titulo) {
      echo"<html><head><title>$titulo</title></head>";
}
```

Y su llamada sería simplemente:

```
creaCabecera("mi web de ejemplo");
```

De esta manera la función, en lugar de devolver un valor que se deba usar en el código de nuestro programa principal se encargará directamente de llevar a cabo su tarea, tras lo cual el flujo de control volverá a la instrucción siguiente a su invocación.

Como decíamos en la introducción también podemos pasar el resultado de una función a una variable para ser usada más tarde, operar entre resultados de funciones o incluso hacer uso de una función como parte de una condición de un bucle:

```
$total= calcularPrecioFinal ($precio, $cantidad);
$paginas= totalMensajes($mensajes)/mensajesPorPagina($resolucion);
For (i==0; i<=longitud ($vector); i++) {…}
```

La instrucción return está diseñada para que devuelva sólo un valor, si bien podría devolver más de uno si los encapsulamos dentro de un dato estructurado como un array o un objeto.

```
<?php

$frutas = array("limón", "naranja", "banana", "albaricoque");
sort($frutas);
foreach ($frutas as $clave => $valor) {
    echo "frutas[" . $clave . "] = " . $valor . "\n";
}

?>
```

La función sort(), incluida en PHP, permite ordenar los valores un array por orden alfabético. El resultado del ejemplo sería:

```
frutas[0] = albaricoque
frutas[1] = banana
frutas[2] = limón
frutas[3] = naranja
```

Las funciones no necesitan ser definidas antes de que se referencien y dentro de una función puede aparecer cualquier código PHP válido, incluso otras funciones. PHP no soporta la sobrecarga de funciones, ni es posible redefinir funciones previamente declaradas, salvo en casos en los que tenemos una función dentro de otra.

Funciones dentro de funciones

Aunque ya sabemos que una función puede ser invocada desde dentro de otra, en ocasiones puede darse el caso de que necesitemos tener una función definida dentro de otra. Esto no es un problema en absoluto si se hace adecuadamente:

```
function calcularPrecioFinal ($precio, $cantidad) {
```

```
        function PrecioIVAInc ($precio) {
                return ($precio +$precio*IVA/100); //IVA es constante
        }
        return(PrecioIVAInc($precio)*$cantidad);
}
```

Argumentos de funciones

En PHP los argumentos o parámetros de las funciones se pasan por valor, por lo que si el valor del argumento dentro de la función se cambia, no se cambia fuera de la función. Para permitir a una función modificar sus argumentos, éstos deben pasarse por referencia.

Para hacer que un argumento a una función sea siempre pasado por referencia hay que poner delante del nombre del argumento el signo 'ampersand' (&) en la definición de la función:

```
function añadir_algo(&$cadena)
{
    $cadena .= 'y algo más.';
}

$cad = 'Esto es una cadena, ';
añadir_algo($cad);
echo $cad;    // imprime 'Esto es una cadena, y algo más.'
```

Argumentos predeterminados

Una función puede definir valores predeterminados al estilo C++ para argumentos escalares como sigue:

```
function hacercafé($tipo = "capuchino")
{
    return "Hacer una taza de $tipo.\n";
}
echo hacercafé();
echo hacercafé(null);
echo hacercafé("espresso");
```

El resultado del ejemplo sería:

```
Hacer una taza de capuchino.
Hacer una taza de .
Hacer una taza de espresso.
```

El valor predeterminado debe ser una expresión constante, no (por ejemplo) una variable, un miembro de una clase o una llamada a una función.

Observe que cuando se usan argumentos predeterminados, cualquiera de ellos debería estar a la derecha de los argumentos no predeterminados; si no, las cosas no funcionarán como se esperaba.

4.2. Ámbito de las variables

Todas las funciones y las clases de PHP tienen ámbito global - pueden ser llamadas fuera de una función incluso si fueron definidas dentro, y viceversa. Sin embargo, a las variables actúan justamente al contrario; el ámbito de una variable es el contexto dentro del que la variable está definida. La mayor parte de las variables PHP sólo tienen un ámbito local.

Variables locales

Entendemos como variables de ámbito local a aquellas que se encuentran definidas en un entorno cerrado, como por ejemplo dentro de una función. Fuera de dicho ámbito tanto el valor como la variable dejan de existir de forma automática.

Por ejemplo:

```
<?php
Function creaNumero() {
      $num=rand ();
      Echo $num;
}
creaNumero();
Echo $num;
¿>
```

El primer echo imprimirá un número entero aleatorio. El segundo echo presentará el valor NULL porque la variable $num simplemente no existe fuera de la función creaNumero dado que es local a esta función.

Si quisiéramos extraer ese valor deberíamos devolverlo desde dentro de la función mediante la instrucción return () y entonces asignarlo a otra variable o imprimirlo directamente con echo:

```
<?php
Function creaNumero() {
      $num=rand ();
      Echo $num;
      Return ($num);
}
Echo creaNumero();
¿>
```

Así se imprimirán dos veces seguidas el mismo valor calculado aleatoriamente, una con el echo de dentro de la función y la otra por el echo externo que imprime lo que la función devuelve, que es el mismo valor.

Variables globales

Se conoce como variable global a aquella que está definida en un entorno abierto a nuestras funciones, es decir, fuera de ellas. Sin embargo las funciones no podrán hacer uso de una variable global a menos que se especifique:

```
<?php
$num=8;
Function creaNumero() {
      $num=24;
      Echo $num;
}
creaNumero();
Echo $num;
¿>
```

El resultado de este código sería 24 y 8. Pese a haberse definido la variable $num=8 a nivel global, la función encuentra otra variable dentro con un valor=24, por lo que el primer echo, al estar dentro de la función, mostrará 24, el valor local de la variable $num. EL Echo que encontramos fuera de la función mostrará 8 porque encuentra la variable global al principio del código.

Si quisiéramos manipular el valor de una variable global desde dentro de una función definirla precediendo el nombre de dicha variable de la palabra clave **global**:

```
<?php
$a = 1;
$b = 2;

function Suma()
{
    global $a, $b;
    $b = $a + $b;
}
Suma();
echo $b;
?>
```

El script anterior producirá la salida *3*. Al declarar *$a* y *$b* globales dentro de la función, todas las referencias a tales variables se referirán a la versión global. No hay límite al número de variables globales que se pueden manipular dentro de una función.

Un segundo método para acceder a las variables desde un ámbito global es usando el array ***$*GLOBALS**. El ejemplo anterior se puede reescribir así:

```
<?php
$a = 1;
$b = 2;

function Suma()
{
    $GLOBALS['b'] = $GLOBALS['a'] + $GLOBALS['b'];
}
```

```
Suma();
echo $b;
?>
```

El array **$GLOBALS** es un array asociativo con el nombre de la variable global como clave y los contenidos de dicha variable como el valor del elemento del array.

Variables estáticas

Otra característica importante del ámbito de las variables es la variable *estática*. Una variable estática existe sólo en el ámbito local de la función, pero no pierde su valor cuando la ejecución del programa abandona este ámbito. Consideremos el siguiente ejemplo:

```
<?php
function test()
{
    $a = 0;
    echo $a;
    $a++;
}
?>
```

Esta función tiene poca utilidad ya que cada vez que es llamada asigna a *$a* el valor *0* e imprime un *0*. La sentencia *$a*++, que incrementa la variable, no sirve para nada, ya que en cuanto la función finaliza, la variable *$a* desaparecerá. Para hacer una función útil para contar, que no pierda la pista del valor actual del conteo, la variable *$a* debe declararse como estática:

```
?php
function test()
{
    static $a = 0;
    echo $a;
    $a++;
}
?>
```

Ahora, $a se inicializa únicamente en la primera llamada a la función, y cada vez que la función test() es llamada, imprimirá el valor de $a y lo incrementa.

Las variables estáticas también proporcionan una forma de manejar funciones recursivas. Una función recursiva es la que se llama a sí misma. Se debe tener cuidado al escribir una función recursiva, ya que puede ocurrir que se llame a sí misma indefinidamente. Hay que asegurarse de implementar una forma adecuada de terminar la recursión.

4.3. Recursividad

En el ámbito de la programación, un **algoritmo recursivo** es un algoritmo que expresa la solución de un problema en términos de una llamada a sí mismo. La llamada a sí mismo se conoce como llamada recursiva o recurrente. A nivel de programación esto resulta especialmente fácil de llevar a cabo mediante el uso de funciones, dado que no existe ninguna regla que lo impida.

Ejemplo de función recursiva:

```
function recursividad($a)
{
    if ($a < 20) {
        echo "$a\n";
        recursividad($a + 1);
    }else {
      echo "$a es mayor o igual a 20";
}
```

En ocasiones la recursividad es la mejor forma de solucionar determinados problemas, como por ejemplo para el cálculo de un número factorial (*):

```
function factorial ($num) {
      if ($num<2) {
            return (1)
      }else{
            return = $num * factorial ($num-1);
      }
}
```

(*) El factorial de un entero positivo n, el factorial de n o n factorial se define en principio como el producto de todos los números enteros positivos desde 1 (es decir, los números naturales) hasta n. Por ejemplo, 

En PHP es posible llamar a funciones recursivas. Sin embargo, evite las llamadas a funciones/métodos recursivos con más de 100-200 niveles de recursividad ya que pueden agotar la pila y causar la terminación del script actual.

Una posible solución para evitar caer en bucles infinitos es usando, como hemos mencionado antes, variables estáticas. La siguiente función cuenta recursivamente hasta 10, usando la variable estática *$count* para saber cuándo parar:

```
function test()
{
    static $count = 0;
    $count++;
    echo $count;
    if ($count < 10) {
        test();
    }
    $count--;
}
```

4.4. Usar librerías de funciones

A medida que un proyecto gana en complejidad, el número de funciones tiendo a multiplicarse y conviene tener en cuenta algún tipo de medida en aras de la estructuración. También sucederá que de un proyecto a otro nos interese reutilizar determinadas funciones desarrolladas por nosotros (para conexión a las BBDD, acceso a ficheros, etc) o incluso hacer uso de herramientas previamente desarrolladas por otros programadores. La solución para este tipo de cuestiones pasará por el uso de librerías.

Entendemos por librería (o también biblioteca) a un conjunto razonablemente ordenado de funciones u objetos que poseen algo en común y una interface bien definida que a menudo se localizan en un archivo externo a nuestro proyecto. Se trata de una práctica muy común en el mundo de la programación, no solo por el hecho de mejorar la estructuración de nuestro código y modularizar nuestras herramientas sino también por permitir que ciertos proyectos se ayuden del trabajo de otros programadores si el lenguaje en cuestión cuenta con una extensa comunidad.

Para crear una librería no necesitamos nada que no sepamos usar ya: programar el código de forma normal con sus correspondientes etiquetas de apertura y cierre y guardar el archivo con extensión php o inc. Después simplemente debemos usar desde la página principal las instrucciones **include** o **require**.

```
<?php
include “lib/MisFunciones.php”;
include “bbdd.inc”;
require “seguridad.php”;
¿>
```

En PHP apenas existe diferencia entre una instrucción y la otra salvo por el hecho de que la instrucción include es más tolerante a fallos que require y permitirá continuar con la ejecución del programa aunque se encuentre un fallo. Típicamente se hace uso de require para aplicaciones que necesiten obligatoriamente algún archivo crítico.

En ambos casos los archivos son incluidos con base en la ruta de acceso dada o, si ninguna es dada, el **include_path** especificado. Si el archivo no se encuentra en el include_path, include finalmente verificará en el propio directorio del script que hace el llamado y en el directorio de trabajo actual, antes de fallar.

En proyectos de alta complejidad puede darse la situación de que incluyamos muchas librerías, lo que puede provocar que, por error, incluyamos el mismo archivo en más de una ocasión, generándose a menudo errores de difícil detección. Para evitar este tipo de situaciones podemos usar las variantes **include_once y require_once**

Cuando se incluye un archivo, el código que contiene hereda el ámbito de las variables de la línea en la cual ocurre la inclusión. Cualquier variable disponible en esa línea del archivo que hace el llamado, estará disponible en el archivo llamado, desde ese punto en adelante. Sin embargo, todas las funciones y clases definidas en el archivo incluido tienen el ámbito global.

vars.php
```
<?php

$color = 'verde';
$fruta = 'manzana';
?>
```

test.php
```
<?php

echo "Una $fruta $color"; // Una

include 'vars.php';

echo "Una $fruta $color"; // Una manzana verde
?>
```

El desarrollo de páginas web a menudo también se vale de las instrucciones include y su funcionalidad para construir páginas de forma modular, algo especialmente útil en caso de que existan secciones que se deben repetir. Es fácil encontrarse, por ejemplo, con webs cuyas primeras líneas de código (encabezado y los primeros elementos como logotipo, menús, etc) residen en un archivo y el cuerpo con los contenidos principales en otro archivo distinto que será el que incluya los demás módulos:

```
<html>
<head>
<?php include (“header.php”);?>
</head>

<body>
<?php include (“menus.php”); ?>

<div id=”cuerpo”>
      ...
</div>

<?php
      include (“sideNav.php”);
      include (“bottom.php”);
¿>
</body>
</html>
```

4.5. Funciones para tratamiento de texto y arrays

PHP incluye una nutrida colección de funciones propias en su núcleo que vale la pena conocer. De entre todas ellas en este apartado destacaremos algunas de las más importantes para el tratamiento de cadenas de texto, expresiones regulares y matrices.

La mayoría de las funciones orientadas al tratamiento de cadenas de texto están pensadas para tomar una cadena como argumento y devolver algún dato (longitud, posición de un carácter, etc), otra cadena procesada o incluso un array. Presentamos aquí una pequeña lista de las más populares, aunque hay muchas más.

Funciones para el tratamiento de cadenas de texto

strlen ($cadena): proporciona la longitud de una cadena

```
$longitud= strlen ($cadena);
```

strpos ($cadena, $letra): encuentra la posición de la primera ocurrencia de un carácter dentro de una cadena; insensible a mayúsculas y minúsculas

```
$mystring = 'abc';
$findme   = 'b';
$pos = strpos($mystring, $findme); //devolverá la posición 1, pues
empieza por 0 como un array
```

strcmp ($cadena1, $cadena2): comparación de cadenas bit a bit; si devuelve 0 es que ambas cadenas son iguales

```
if (strcmp ($cadena1, $cadena2)==0) {
      echo "ambas cadenas son iguales";
}
```

strstr($cadena, "texto"): encuentra la primera aparición de una cadena dentro de otra cadena; no devuelve una posición sino el string a partir del punto en el que aparece el texto buscado

```
$email  = 'name@example.com';
$domain = strstr($email, '@');
echo $domain; // mostrará @example.com
```

substr ($cadena, $inicio, $longitud): devuelve parte de una cadena desde $inicio y con longitud $longitud. Si $inicio es un entero negativo empezará por el final de la cadena. Si $longitud no se indica la función devolverá todo el texto desde $incio:

```
$resto = substr("abcdef", 1, 1);    // devuelve "b"
$resto = substr("abcdef", 2);    // devuelve "cdef"
$resto = substr("abcdef", -1);    // devuelve "f"
$resto = substr("abcdef", -2);    // devuelve "ef"
$resto = substr("abcdef", -3, 1); // devuelve "d"
```

str_replace ($texto1, $texto2, $cadena): busca todas las apariciones del texto $texto1 en $cadena y lo sustituye por $texto2:

```
$cadena=”Esta frase contiene muchas palabras. Esta otra frase también tiene muchas palabras”;
$cadena= str_replace (“frase”, “oración”, $cadena);
// Mostrará “Esta oración contiene muchas palabras. Esta otra oración también tiene muchas palabras”
```

strrev ($cadena): invierte el contenido una cadena

```
$cadena = 'abcdef';
$revertida = strrev($cadena);

echo $revertida; // Esto imprime algo como: fedcba
```

str_shuffle ($cadena): baraja o reordena aleatoriamente el contenido de la cadena de entrada y devuelve una cadena de la misma longitud pero con sus caracteres reordenados.

```
$cadena = 'abcdef';
$desordenada = str_shuffle($cadena);

echo $desordenada; // Esto imprime algo como: bfdaec
```

Htmlspecialchars ($cadena): esta función se suele usar especialmente a la hora de recoger datos desde un campo de formulario para garantiza que cualquier carácter que sea especial en html se codifique adecuadamente, de manera que nadie pueda inyectar etiquetas HTML o Javascript en nuestras páginas y provocar errores.

```
$textolimpio= htmlspecialchars ($_POST['campo34']);
```

rtrim ($cadena): retira los espacios en blanco (u otros caracteres) del final de un string; esta función también es conocida como chop

```
echo rtrim(“Texto con espacios al final     “); // imprime “Texto con espacios al final”
echo rtrim(“Texto con punto final.“,  “.”); // imprime “Texto con espacios al final”
```

ltrim($cadena):de forma similar a la función anterior, retira espacios en blanco (u otros caracteres) del inicio de un string

```
echo ltrim(“Texto con espacios al final“); // imprime “Texto con espacios al final”
echo ltrim(“Texto con punto final.“,  “T”); // imprime “exto con espacios al final.”
```

trim($cadena):de forma similar a las funciones anteriores, trim es capaz de retira espacios en blanco (u otros caracteres) de un string por la izquierda y por la derecha simultáneamente

```
echo trim("Texto con espacios al principio y al final "); // imprime
"Texto con espacios al final"
```

strtolower ($cadena) y **strtoupper ($cadena)**: convierten el contenido de un texto todo a minúsculas o toda a mayúsculas respectivamente.

Funciones para el tratamiento de expresiones regulares

Las expresiones regulares son patrones de búsqueda dentro de cadenas. Estos patrones se utilizan muy a menudo para comprobar valores de entrada como una dirección de correo bien formada, un DNI o un número de tarjeta con la codificación adecuada, etc Estos patrones se construyen mediante caracteres especiales que cumplen unas reglas determinadas. Listamos aquí algunos de los más importantes:

Patrón	Significado
c	carácter c
.	cualquier carácter
^c	empezar por el carácter c
c$	terminar por el carácter c
c+	1 o más caracteres c
c*	0 o más caracteres c
c?	0 o 1 caracteres c
\n	nueva línea
\t	tabulador
\	Escape; para escribir delante de caracteres especiales: ^ . [ ] % ( ) | * ? { } \
[a-z]	cualquier letra minúscula
[A-Z]	cualquier letra mayúscula
[0-9]	cualquier dígito
[cde]	cualquiera de los caracteres c, d o e
[c-f]	cualquier letra entre c y f (es decir, c, d, e o f)
[^c]	que no esté el carácter c
(ab)	conjunto ab (caracteres a y b agrupados)
(ab|cd|ef)	conjunto ab o cd o ef

Ereg (“exp”, $cadena): es una función capaz de comprobar si una cadena se corresponde con el patrón que se pasa como fragmento.

```
//si evaluamos distintas direcciones de email contra el siguiente
condicional
If (ereg (“[a-z|\.]+@[a-z|\.]+\.(org|com|net)$”, $correo)) {
      Echo “el correo se acepta”;
}else{
      Echo “el correo no se acepta”;
}
// $correo= “joseluis.sanchez@audiogil.es”; → el correo se acepta
// $correo= “joseluis@sanchez@audiogil.es”; → el correo no se acepta
// $correo= “JOSELUIS.sanchez@audiogil.es”;→ el correo no se acepta
```

También existe la función eregi () que es idéntica a ereg () pero no es case sensitive.

Ereg_replace($patron, $reemplazo, $cadena) y eregi_replace($patron, $reemplazo, $cadena): permite reemplazar una expresión regular en una cadena de texto. Esta función busca en $cadena coincidencias de $patron, después reemplaza el texto coincidido con $reemplazo.

```
$número = '4';
$cadena = "Esta cadena tiene cuatro palabras.";
$cadena = ereg_replace('cuatro', $número, $cadena);
echo $cadena;   /* Salida: 'Esta cadena tiene 4 palabras.' */

$texto = ereg_replace("[[:alpha:]]+://[^<>[:space:]]+[[:alnum:]/]",
"<a href=\"\0\">\0</a>", $texto); // sustituye una URL por un vínculo
```

Funciones para el tratamiento de cadenas de arrays

Dada la importancia de la estructura de datos array, el conjunto de funciones orientadas a su tratamiento es numeroso y muy variado en PHP. Por ello nos limitaremos a comentar solo algunas de las más populares y funcionales.

In_array: comprueba si un valor existe en un array. Devuelve True si encuentra el valor en el array y False en caso contrario. Su comprobación es case sensitive.

```
if (in_array("Irix", $os)) {
    echo "Existe Irix";
}
if (in_array("mac", $os)) {
    echo "Existe mac";
}
```

Count: cuenta los elementos de una matriz

```
$a[0] = 1;
$a[1] = 3;
$a[2] = 5;
$resultado = count($a);
// $resultado == 3
```

Unset: al igual que con las variables, podemos usar esta función para eliminar elementos de un array.

```
$vector = array(0, 10, 20, 30);
unset($vector [2]);
var_dump($vector);
/* vector (3) {
  [0]=>  int(0)
  [1]=>  int(10)
  [3]=>  int(30)
} */
```

Hay que tener presente que para poder eliminar un elemento en concreto deberemos conocer su índice. De lo contrario, si llamamos a la función sin índices hacemos vaciaremos todo el array y obtendremos un error si intentamos acceder a él:

```
unset ($vector);
```

Sort: esta función ordena un array. Los elementos estarán ordenados de menor a mayor cuando la función haya terminado.

```
$frutas = array("limón", "naranja", "banana", "albaricoque");
sort($frutas);
foreach ($frutas as $clave => $valor) {
    echo "frutas[" . $clave . "] = " . $valor . "\n";
}
```

El resultado del ejemplo sería:

```
frutas[0] = albaricoque
frutas[1] = banana
frutas[2] = limón
frutas[3] = naranja
```

Rsort: esta función ordena un array en orden inverso (mayor a menor).

```
$fruits = array("limón", "naranja", "plátano", "manzana");
rsort($fruits);
foreach ($fruits as $key => $val) {
    echo "$key = $val\n";
}
```

El resultado del ejemplo sería:

```
0 = plátano
1 = naranja
2 = manzana
3 = limón
```

Las frutas han sido ordenadas alfabéticamente pero en orden inverso.

Array_reverse: toma un valor `array` y devuelve un nuevo array con el orden de los elementos invertido.

```
$fruits = array("limón", "naranja", "plátano", "manzana");
echo (array_reverse($input));
```

El resultado del ejemplo sería:

```
manzana
plátano
naranja
limón
```

Shuffle: esta función mezcla un array (crea un orden aleatorio de sus elementos). En realidad, en lugar de reordenar los elementos o las claves, esta función asigna nuevas clave a los elementos del `array`, eliminando cualquier clave existente que haya sido asignada de entrada.

```
$números = range(1, 20);
shuffle($números);
foreach ($números as $número) {
    echo "$número ";
}
```

Current, next, prev y reset: devuelve un valor del array según donde se encuentre el puntero de este.

current() devuelve el elemento actual en un array. Cada array tiene un puntero interno a su elemento "actual", que es iniciado desde el primer elemento insertado en el array.

next() se comporta como current(), con una diferencia. Avanza el puntero interno un lugar a delante antes de devolver el valor del elemento. Esto significa que devuelve el siguiente valor del array y avanza el puntero interno del array un lugar.

prev() se comporta como next(), a excepción de que rebobina el puntero interno del array una posición en lugar de avanzar.

reset() rebobina el puntero interno de un `array` al primer elemento y devuelve el valor del primer elemento del array.

```
$transporte = array('pie', 'bici', 'coche', 'avión');
$modo = current($transport); // $modo = 'pie';
$modo = next($transport);    // $modo = 'bici';
$modo = current($transport); // $modo = 'bici';
$modo = prev($transport);    // $modo = 'pie';
$modo = reset($transport);    // $modo = 'pie';
```

Key: obtiene una clave de un array. La función key() simplemente devuelve la clave del elemento del array que está apuntando actualmente el puntero interno. No desplaza el puntero de ninguna manera. Si el puntero interno señala más allá del final de la lista de elementos o el array está vacío, key() devuelve NULL.

```
$array = array(
    'fruta1' => 'manzana',
    'fruta2' => 'naranja',
    'fruta3' => 'uva',
    'fruta4' => 'manzana',
    'fruta5' => 'manzana');

// Muestra las claves del array donde el valor equivale a "manzana"
while ($nombre_fruta = current($array)) {
    if ($nombre_fruta == 'manzana') {
        echo key($array).'<br />';
    }
    next($array);
}
```

array_merge: combina los elementos de uno o más arrays juntándolos de modo que los valores de uno se anexan al final del anterior. Retorna el array resultante.

Si los arrays de entrada tienen las mismas claves de tipo string, el último valor para esa clave sobrescribirá al anterior. Sin embargo, los arrays que contengan claves numéricas, el último valor *no* sobrescribirá el valor original, sino que será añadido al final. Los valores del array de entrada con claves numéricas serán renumeradas con claves incrementales en el array resultante, comenzando desde cero.

```
$array1 = array("color" => "red", 2, 4);
$array2 = array("a", "b", "color" => "green", "shape" => "trapezoid",
4);
$resultado = array merge($array1, $array2);
print_r($resultado);
```

El resultado del ejemplo sería:

```
Array
(
    [color] => green
    [0] => 2
    [1] => 4
    [2] => a
    [3] => b
    [shape] => trapezoid
    [4] => 4
}
```

array_push: inserta uno o más elementos al final de un array. Esta función trata array como si fuera una pila y coloca la variable que se le proporciona al final del array. El tamaño del array será incrementado por el número de variables insertados.

```
$pila = array("naranja", "plátano");
array_push($pila, "manzana", "arándano");
print($pila);
```

El resultado del ejemplo sería:

```
Array
(
    [0] => naranja
    [1] => plátano
    [2] => manzana
    [3] => arándano
)
```

Nota: Si se utiliza **array_push()** para añadir un solo elemento al array, es mejor utilizar *$array[]* = ya que de esta forma no existe la sobrecarga de llamar a una función.

array_pop: extrae y devuelve el último valor del array, acortando el array con un elemento menos. Esta función además ejecutará un reset() en el puntero de array del array de entrada después de su uso.

```
<?php
$stack = array("naranja", "plátano", "manzana", "frambuesa");
$fruit = array_pop($stack);
print_r($stack);
?>
```

Después de hacer esto, $stack solo tendrá 3 elementos:

```
Array
(
```

```
    [0] => naranja
    [1] => plátano
    [2] => manzana
)
```

Array_pop() junto con array_push() se suelen utilizar para emular el comportamiento de una pila.

Array_shift(): quita el primer valor del array y lo devuelve, acortando el array un elemento y corriendo el array hacia abajo. Todas las claves del array numéricas serán modificadas para que empiece contando desde cero mientras que los arrays con claves literales no serán modificados.

```
<?php
$stack = array("naranja", "plátano", "manzana", "frambuesa");
$fruit = array_shift($stack);
print_r($stack);
?>
```

El resultado del ejemplo sería:

```
Array
(
    [0] => plátano
    [1] => manzana
    [2] => frambuesa
)
```

Esta función, junto con array_push() se suelen utilizar para emular el comportamiento de una cola.

5. Paso de información entre documentos PHP

Si tenemos en cuenta cómo se construyen las webs hoy en día, una de las funcionalidades más importantes de un lenguaje como PHP es la de pasar información de una página a otra a medida que un usuario las visita. Podemos además entender que este proceso implica una comunicación entre el cliente y el servidor en el sentido de que uno envía información al otro, información que debe permanecer activa o accesible a medida que el usuario hace uso de la web.

Aunque no todas cumplen el mismo cometido, PHP proporciona varias herramientas de probada eficacia como son el pase de información a través de formularios y la cabecera http, el uso de cookies y el tratamiento de sesiones. Dada su popularidad y utilidad, en este tema estudiaremos todos estos mecanismos.

5.1 Paso de información con formularios mediante GET y POST

Una de las fórmulas más empleadas para transmitir información de una página a otra es mediante el uso de formularios y el protocolo HTTP. Según el típico escenario, un usuario pasa de una página a otra al pulsar sobre un enlace o botón que le hará viajar a la siguiente página o realizar algún tipo de consulta al servidor. En estos casos, se suele aducir que al hacer click se está enviando información al servidor, la cual será procesada y provocará una respuesta de algún tipo. Otro típico escenario será al rellenar un formulario, dado que la información incluida deberá enviarse a algún lugar para, de nuevo, ser tratada. Para este tipo de operaciones PHP se vale de dos métodos, los conocidos como **GET** y **POST**.

En apariencia ambos métodos funcionan de forma similar, haciendo uso de unas variables superglobales de tipo array asociativo (**$_GET** y **$_POST**) a las que podremos acceder en todo momento, sin embargo cada método aporta una funcionalidad específica a tener en cuenta. Veamos su funcionamiento con sencillo un ejemplo a partir del siguiente formulario:

```
<form action="accion.php" method="get">
<p>Su nombre: <input type="text" name="nombre" /></p>
<p>Su edad: <input type="text" name="edad" /></p>
<p><input type="submit" /></p>
</form>
```

Al pulsar el botón de submit, este formulario HTML enviará los valores introducidos en los campos input llamados “nombre” y “edad” al archivo accion.php mediante un el método GET. Por su parte, el archivo accion.php procesará la información de la siguiente manera:

```
Hola <?php echo htmlspecialchars($_GET['nombre']); ?>.
Usted tiene <?php echo (int)$_GET['edad']; ?> años.
```

Y mostrará el siguiente resultado por pantalla:

```
Hola Jose. Usted tiene 22 años.
```

Lo que se ha sucedido aquí es que al hacer submit se ha creado un array asociativo con los pares índice-valor con los campos del formulario correspondientes al nombre y al valor y lo ha almacenado en un array llamado $_GET. Después, el documento accion.php ha accedido a los valores de la variable $_GET a través de sus índices "nombre" y "edad" y los ha impreso.

Adicionalmente, en el caso del nombre, se hace uso de la función **htmlspecialchars()** para garantiza que cualquier carácter que sea especial en html se codifique adecuadamente, de manera que nadie pueda inyectar etiquetas HTML o Javascript en la página. El campo edad, ya que sabemos que es un número, podemos convertirlo a un valor de tipo integer que automáticamente se deshará de cualquier carácter no numérico.

Esto son medidas bastante habituales a la hora de trabajar con entrada de datos por formulario para filtrar la información recibida y evitar así usos malintencionados que puedan provocar fallos o robos de datos en nuestras aplicaciones web.

Este mismo ejemplo podría ser usado con el método POST en el formulario y recogido igualmente desde accion.php mediante la variable superglobal $_POST.

Diferencias entre los métodos GET y POST

La principal diferencia entre hacer uso de GET o POST es que al usar GET todos los pares campo-valor son enviados a través de la URL en su llamada al archivo accion.php de una forma similar a esta: *http://localhost/php/accion.php?nombre=Jose&edad=22*

Estas llamadas son visible por cualquiera en tiempo de navegación y también pueden ser cacheadas (historial del navegador), indexadas por buscadores e incluso podemos agregar los enlaces a nuestros favoritos o hasta pasar una url completa a otra persona para que directamente ingrese a esa página. Con el método POST sin embargo no se puede hacer esto.

Es por ello que se tiende a pensar que el envío de información mediante el método GET es menos seguro que usando el método POST y que por lo tanto GET no debería ser usado. Sin embargo esto no es del todo cierto. Las solicitudes GET son sencillas, y no necesitan de un formulario, ya que este mecanismo nos permite crear vínculos que envíen directamente parámetros de una página a otra desde dentro del código. Estas son especialmente recomendables cuando la información enviada no servirá para modificar datos, como por ejemplo en el caso de un buscador

Con el método POST, el navegador hace una conexión al servidor y se encarga de mandar los datos de manera separada, de esta forma los datos se mantienen hasta cierto punto privados, y tenemos la ventaja de que podemos enviar una mayor cantidad de datos en la solicitud.

El método POST debe ser utilizado en lugar de GET cuando deseamos transmitir mucha información, modificar datos en el servidor o en el caso de transmitir información sensible, como por ejemplo autenticar un usuario con contraseña, validar una compra, actualizar datos en una BBDD…

```
<form action="acceso.php" method="post">
<p>Nombre de usuario: <input type="text" name="user" /></p>
<p>Contraseña: <input type="password" name="pass" /></p>
<p><input type="submit" /></p>
</form>
```

Acceso.php

```
<?php
$usuario="Pepe";
$contrasenya="hola1";

If (($usuario== htmlspecialchars($_POST['user'])) && ($contrasenya==
htmlspecialchars($_POST['pass']))) {
Echo "Bienvenido ".htmlspecialchars($_POST['user']);
}else{
Echo "nombre de usuario o contraseña incorrectos";
}
?>
```

Tanto $_GET como $_POST son variables superglobales, lo que significa que son variables que están disponibles en cualquier parte del script. No hace falta hacer global $variable; para acceder ellas desde funciones o métodos.

5.2 Paso de información con cookies y su tratamiento

PHP soporta el tratamiento de cookies HTTP para almacenar datos en el navegador del cliente y poder así monitorizar o identificar a usuarios que vuelven al sitio web.

Una cookie es un fragmento de información que un navegador web almacena en el disco duro del visitante a una página web. La información se almacena a petición del servidor web, ya sea directamente desde la propia página web con JavaScript o desde el servidor web mediante las cabeceras HTTP, que pueden ser generadas desde un lenguaje de web scripting como PHP. La información almacenada en una cookie puede ser recuperada por el servidor web en posteriores visitas a la misma página web.

Las cookies resuelven un grave problema del protocolo HTTP: al ser un protocolo de comunicación "sin estado" (stateless), no es capaz de mantener información persistente entre diferentes peticiones. Gracias a las cookies se puede compartir información entre distintas páginas de un sitio web o incluso en la misma página web pero en diferentes instantes de tiempo.

HTML 5

La llegada del estandar HTML 5 ha propiciado la aparición de nuevas funcionalidades, algunas de las cuales resuelven particularmente la situación de almacenamiento local de información., proporcionando herramientas de base como local storage. Esta alternativa es más rápida, más segura y ofrece una capacidad de almacenamiento mucho mayor (entorno a los 5 Mb). Sin embargo esta funcionalidad está diseñada para que la información no sea transferida al servidor, por lo que su acceso desde PHP es imposible a menos que se haga uso de herramientas como Javascript y más concretamente Ajax. El objeto en cuestión se llama window.localStorage, el cual permite almacenar información sin fecha de caducidad. Existe también otro objeto llamado window.sessionStorage que almacenará la información sólo durante la sesión actual (hasta que se cierre la pestaña o ventana del navegador).

- Más información sobre local storage y sesión storage en HTML 5
- Más información sobre el tratamiento y acceso de PHP a la caché del cliente

Guardar una cookie

En PHP se emplea la función **setcookie**() para asignar valor a una cookie. El prototipo de esta función es:

bool setcookie(string $name [, string $value [, int $expire [, string $path [,

string $domain [, bool $secure]]]]])

Todos los argumentos con excepción de nombre son opcionales. Es posible también reemplazar un argumento con una cadena vacía ("") para evitar ese argumento. Dado que el argumento expire es entero, no puede saltarse con una cadena vacía, se deberá usar un cero (0) en su lugar., aunque a menudo se usa junto con la función time() que proporciona la hora

actual. Por otra parte no se puede hacer una cookie que no expire, pero sí usar una fecha tan distante que parezca no caducar. Por ejemplo 10 años: time() + (10 * 365 * 24 * 60 * 60).

```
$valor = 'algo desde algún lugar';

setcookie("CookieDePrueba", $valor);
setcookie("CookieDePrueba", $valor, time()+3600); // expira en 1 hora

CookieDePrueba = "usuario";
$valor= "John Doe";
setcookie($CookieDePrueba , $valor, time() + (86400 * 30), "/"); //
86400 = 1 día, o sea, expira en un mes
```

Las cookies son parte de la cabecera HTTP, por lo que setcookie() debe ser invocada antes de que cualquier otra salida sea enviada al navegador. Esto requiere que coloque las llamadas a esta función antes de cualquier salida, incluyendo las etiquetas <html> y <head> así como cualquier espacio en blanco. Si existe salida antes de llamar esta función, setcookie() fallará y devolverá FALSE. Si setcookie() se ejecuta con éxito, devolverá TRUE.

En la siguiente tabla se explican los parámetros de setcookie con mayor detalle:

Parámetro	Descripción	Ejemplos
name	El nombre de la cookie.	'nombre_cookie' es llamada como *$_COOKIE['nombre_cookie']*
value	El valor de la cookie. Este valor es almacenado en el equipo del cliente; no almacene información sensible.	Asumiendo que *nombre* es 'nombre_cookie', este valor es recuperado por medio de *$_COOKIE['nombre_cookie']*
expire	La hora en la que expira la cookie. Este valor es una marca de tiempo Unix así que es el número de segundos recorridos desde el epoch. Se puede definir con la función **time()** más el número de segundos antes de que usted quiera que expire.	time()+3600 define que la cookie expirará dentro de una hora a partir de su creación. time()+60*60*24*30 definirá que la cookie expire en 30 días. Si no se define, la cookie expirará al final de la sesión (cuando el navegador sea cerrado).
path	La ruta en el servidor en la que estará disponible la cookie.	Si se define como *'/'*, la cookie estará disponible en el *dominio* completo. Si se define como *'/foo/'*, la cookie será disponible únicamente al interior del directorio */foo/* y todos sus

		subdirectorios. El valor predeterminado es el directorio actual en el que se define la cookie.
domain	El dominio en el que la cookie está disponible.	Para lograr que la cookie esté disponible en todos los subdominios de example.com será necesario definir este valor como *'.example.com'*.
secure	Indica que la cookie debería ser transmitida únicamente sobre una conexión HTTPS segura. Cuando su valor es **TRUE**, la cookie será definida únicamente si existe una conexión segura. El valor predeterminado es **FALSE**.	*0* ó *1*

Si se desea asignar múltiples valores a una única cookie, solo se debe agregar [] al nombre de la cookie.

```
//Definimos una única cookie de tipo array con tres valores

setcookie("cookie[tres]", "cookietres");
setcookie("cookie[dos]", "cookiedos");
setcookie("cookie[uno]", "cookieuno");
```

Setcookie () también nos permite actualizar el valor de una cookie si su nombre ya existía, aunque para ello deberemos volver a poner todos los parámetros que se habían usado al crearla. Esto permite, por ejemplo, desarrollar sencillos contadores de visitas.

Leer una cookie

Para recuperar el valor de una cookie se emplea la variable superglobal de tipo array **$_COOKIE** con el nombre de la cookie como índice.

```
if(!isset($_COOKIE[$cookie])) {
    echo "La cookie llamada '" . $cookie . "' no existe!";
} else {
    echo "La cookie '" . $cookie . "' existe!<br>";
    echo "Su valor es: " . $_COOKIE[$cookie];
}
```

En caso de que nuestra cookie sea de tipo array, leerla implicará recorrerla con ayuda de la instrucción foreach junto con sus índices:

```
if (isset($_COOKIE['cookie'])) {
   foreach ($_COOKIE['cookie'] as $nombre => $valor) {
       echo "$nombre : $valor <br />\n";
   }
}
```

Borrar una cookie

Para borrar una cookie, podemos hacer uso de la función unset() o asignando a la cookie una fecha de caducidad (expire) en el pasado, es decir, una fecha anterior a la actual.

```
unset($_COOKIE["micookie"]);

setcookie("micookie","valor",time()-1);
/*con esta fecha expira, ergo se borra*/
```

Comprobar si las cookies están habilitadas en el navegador del cliente

En la actualidad es posible, por seguridad, que un usuario desactive el uso de cookies en su navegador, si bien es algo que carece de sentido en la mayor parte de los sitios webs existentes. No obstante si nuestro sitio web requiere usar cookies para su correcto funcionamiento, deberíamos poder comprobarlo. Una forma fácil es probando a crear una cookie y acto seguido comprobar si podemos acceder a ella o, incluso de forma más sencilla, comprobando si la variable $_COOKIE contiene algún valor, algo que podemos averiguar fácilmente mediante la función count():

```
if(count($_COOKIE) > 0) {
    echo "Cookies activadas.";
} else {
    echo "Cookies no activadas.";
}
```

5.3 Paso de información con sesiones y su tratamiento

El soporte de sesiones en PHP consiste en una manera de guardar ciertos datos a través de diferentes accesos web. Esto permite crear aplicaciones más personalizadas y mejorar las características del sitio web.

Una sesión es un mecanismo de programación de las tecnologías de web scripting que permite conservar información sobre un usuario al pasar de una página a otra. A diferencia de una cookie, los datos asociados a una sesión se almacenan en el servidor y nunca en el cliente, si bien PHP necesitará almacenar al menos una cookie en el cliente que contendrá un valor que identifique al usuario en el servidor web, lo que se conoce como PHPSESSID. En el servidor web estarán almacenados todos los datos de la sesión y se accede a ellos cada vez que se pasa de página gracias al identificador almacenado en la cookie.

Las sesiones siguen un flujo de trabajo sencillo. Cuando una sesión se inicia, PHP recuperará una sesión existente usando el ID pasado (normalmente desde una cookie de sesión) o, si no se pasa una sesión, se creará una sesión nueva. PHP rellenará la variable superglobal **$_SESSION** con cualquier dato de la sesión iniciada. Cuando PHP se cierra, automáticamente toma el contenido de $_SESSION, lo serializa, y lo envía para almacenarlo en el servidor.

Crear una sesión

Las sesiones se puede iniciar manualmente usando la función **session_start()**, si la directiva **session.auto_start** se establece a 1, una sesión se iniciará automáticamente ante cualquier petición de arranque.

Las sesiones normalmente se cierran se forma automática cuando PHP termina de ejecutar un script, lo que serializa y envía su contenido, pero también se pueden cerrar manualmente usando la función **session_write_close()**.

Lo habitual es iniciar la sesión manualmente en cada página que se desee y permitir que se serialice y envíe automáticamente, ya que iniciar las sesiones automáticamente implica hacer algunos cambios en la configuración de nuestro PHP y forzar, cada vez, el cierre de la sesión, lo que implica serializar y enviar los datos al final de cada página manualmente.

Ejemplo #1 - demo1.php registra datos en una sesión

```
<?php
session_start(); // importante que esta sea la primera línea de código
?>
<!DOCTYPE html>
<html>
<body>
<?php
   //Guarda los datos en la sesión
   $_SESSION["favcolor"] = "azul";
   $_SESSION["favanimal"] = "gato";
   echo "Las variables de sesión se han creado.";
   echo "<a href=\"demo2.php\">Ir a demo2.php</a>";
```

```
?>
</body>
</html>
```

Leer variables de sesión

El acceso a las variables de sesión es muy similar a como lo hacíamos con las cookies, es decir, accediendo a su variable superglobal $_SESSION junto con el índice de cada variable. Además, en la página donde hagamos dicho acceso deberemos también iniciar la sesión para comprobar que existe y es la que esperamos que sea:

Ejemplo #3 – demo2.php accede a los datos de la sesión y los muestra

```
<?php
session_start();
?>


<!DOCTYPE html>
<html>
<body>

<?php
  //Imprime los valores de la sesión guardados en demo1.php
  echo "Mi color favorito es " . $_SESSION["favcolor"] . ".<br>";
  echo "Mi animal favorito es " . $_SESSION["favanimal"] . ".";
?>

</body>
</html>
```

Otra forma de mostrar los valores de una sesión es mediante la instrucción print_r:

```
<?php
print_r($_SESSION);
?>
```

Para ver si una variable de sesión ha sido creada se suele utilizar la función isset() pasándole como parámetro S_SESSION junto con la variable que deseamos saber si ha sido creada.

Ejemplo #3 Registrar una variable con $_SESSION

```
<?php
session_start();
if (!isset($_SESSION['usuario'])) {
  echo “Estás identificado como $_SESSION['usuario'];
} else {
  echo “No estás identificado. <a href=\”login.php\”>Pincha aquí para
hacer login</a>”;
}
?>
```

Para controlar el acceso a una página web de una zona privada se suele emplear una variable de sesión que se inicializa con cierto valor en la página de control de acceso; en las páginas donde se quiere controlar si el usuario tiene permiso para acceder se consulta el valor de la variable de sesión para ver si tiene el valor esperado. Si no contiene el valor esperado, lo normal es mostrar una página con un mensaje de error o redirigir al usuario a la página principal del sitio web.

ID de una sesión

Como hemos dicho, cuando se crea una sesión se genera un ID automático de 32 caracteres que se guardará tanto en la sesión como en una cookie del cliente. Así, cada vez que accedemos a una página en la cual nuestra sesión debe estar activa (por ejemplo un usuario identificado), lo que hace PHP es comprobar internamente si el ID de la cookie coincide con el de la sesión.

Dado que todo esto se realiza de forma automática y transparente para el usuario, quizá el desarrollador sí desee acceder en algún momento a dicha información. Esto se consigue mediante la función session_id(). Asimismo también podemos acceder fácilmente a dicho valor en la cookie mediante el índice PHPSESSID. Veamos un ejemplo en el que podemos contrastar ambos valores:

```
<?php
session_start();
?>
<!DOCTYPE html>
<html>
<body>
<?php
   $_SESSION["nombre"] = "Luis";
   echo $_SESSION["nombre"]."<br>";
   echo "id en session: ".session_id()."<br>";
   echo "id en cookie: ".$_COOKIE["PHPSESSID"]."<br>";
   echo SID;
?>

</body>
</html>
```

La cookie se gestiona de forma automática, de manera que no es necesario crearla, renovarla o borrarla ya que todos estos procesos se realizarán de forma automática junto con la sesión correspondiente, incluido su borrado si la sesión es destruida, cosa que sucederá en el momento en que el usuario abandone la web.

No obstante sí puede interesarnos alterar el tiempo de vida de la cookie asociada a la sesión PHPSESSIONID para que, por ejemplo, la sesión caduque con mayor rapidez, como en el caso de por ejemplo una web bancaria o para que tarde semanas o meses, como puede ser el caso de uncarro de la compra de una tienda online.

Podemos hacer esto mediante la instrucción ini_set de la siguiete manera:

```
  ini_set('session.cookie_lifetime', 30*60); // 30 minutos
```

También podemos establecer este tipo de valor para todas nuestras páginas si lo definimos en el archivo de configuración de nuestro intérprete de PHP en el servidor: php.ini

Eliminar variables de sesión

Para eliminar una variable de una sesión podemos hacer uso de la función **unset()**:

Ejemplo #4 Destruir una variable con $_SESSION

```
<?php
   session_start();
   unset($_SESSION['usuario']);
?>
```

Para destruir una sesión, es decir, eliminar sus datos así como la cookie asociada a esta con el session_id podemos hacer uso de las instrucciones **session_unset()** y **session_destroy()**:

```
<?php
session_unset();// borra todas las variables de sesión

session_destroy();// destruye la sesión (y su cookie asociada)
?>
```

6. Tratamiento de ficheros

Trabajar con archivos es una funcionalidad muy importante en cualquier lenguaje de programación y el caso de PHP no es una excepción. Dado que PHP funciona en un servidor, el tratamiento de archivos se llevará a cabo en este, no en el cliente, si bien podemos generar archivos que después sean descargables para el navegante de la web. Al tiempo que hablamos de archivos también estudiaremos cómo crear y gestionar carpetas y cómo movernos por estructuras de ficheros del servidor.

Otro de los factores importantes a estudiar aquí será justo lo contrario, cómo abordar el proceso de subir archivos desde el cliente al servidor y poder almacenarlos adecuadamente.

6.1 Funciones para lectura y escritura de ficheros

Independientemente de lo que hagamos con un fichero, su tratamiento en PHP se traduce en la realización de una secuencia de pasos muy sencilla:

1. Apertura del fichero
2. Manipulación del fichero (lectura o escritura)
3. Cierre del fichero

Para realizar cada uno de estos pasos haremos uso de funciones específicas que estudiaremos a continuación con mayor detenimiento.

Apertura de un fichero

Para cualquier tipo de operación contra un fichero el primer paso que debemos realizar es su apertura. Para ello utilizaremos la función **fopen():**

> *resource fopen (string $filename , string $mode [, bool $use_include_path = false [, resource $context]])*

fopen() asocia un recurso con nombre, especificado por filename, a un flujo con un modo de acceso específico. Un flujo es un objeto de tipo resource que exhibe un comportamiento similar al de un flujo. Esto es, puede ser leído o escrito de una manera lineal, y puede usar la función **fseek()** para buscar posiciones arbitrarias dentro del flujo.

filename

El argumento filename hará referencia a un archivo o a una URL. Si filename está en la forma "esquema://...", se asume que será un URL y PHP buscará un gestor de protocolos (también conocido como envoltura) para ese protocolo.

Si PHP ha decidido que filename especifica un fichero local, intentará abrir un flujo para ese fichero. El fichero debe ser accesible para PHP, por lo que es necesario asegurarse de que los permisos de acceso del fichero permiten este acceso.

Si PHP ha decidido que filename especifica un protocolo registrado, y ese protocolo está registrado como un URL de red, PHP se asegurará de que **allow_url_fopen** está habilitado. Si es desactivado, PHP emitirá un aviso y la llamada a fopen fallará.

En la plataforma Windows, debemos asegurarnos de escapar cualquier barra invertida usada en la ruta de fichero, o use barras hacia delante.

mode

El parámetro mode especifica el tipo de acceso que se necesita para el flujo. Puede ser cualquiera de los siguientes:

mode	Descripción
'r'	Apertura para sólo lectura; coloca el puntero al fichero al principio del fichero.
'r+'	Apertura para lectura y escritura; coloca el puntero al fichero al principio del fichero.
'w'	Apertura para sólo escritura; coloca el puntero al fichero al principio del fichero y trunca el fichero a longitud cero. Si el fichero no existe se intenta crear.
'w+'	Apertura para lectura y escritura; coloca el puntero al fichero al principio del fichero y trunca el fichero a longitud cero. Si el fichero no existe se intenta crear.
'a'	Apertura para sólo escritura; coloca el puntero del fichero al final del mismo. Si el fichero no existe, se intenta crear. En este modo, fseek() solamente afecta a la posición de lectura; las lecturas siempre son pospuestas.
'a+'	Apertura para lectura y escritura; coloca el puntero del fichero al final del mismo. Si el fichero no existe, se intenta crear. En este modo, fseek() solamente afecta a la posición de lectura; las lecturas siempre son pospuestas.
'x'	Creación y apertura para sólo escritura; coloca el puntero del fichero al principio del mismo. Si el fichero ya existe, la llamada a fopen() fallará devolviendo FALSE y generando un error de nivel E_WARNING. Si el fichero no existe se intenta crear.

En términos generales usaremos los modos r para sólo leer, w para sólo escribir, w+ para actualizar y a+ para añadir al final del archivo con posibilidad de leer y buscar información mediante fseek().

```
$gestor = fopen("/test/fichero.txt", "r"); //abre fichero.txt ubicado
en el directorio test para solo lectura

$gestor = fopen("/test/fichero2.txt", "w"); //abre fichero.txt ubicado
en el directorio test para escritura; si el fichero no existe se
intenta crear

$gestor = fopen("http://www.audiogil.com/texto.txt", "r"); //abre el
fichero texto.txt ubicado en una URL para su lectura
```

```
$gestor = fopen("ftp://ftp.audiogil.es/texto.txt", "a+"); //
suponiendo que se dispone de los permisos adecuados, se abre el
fichero texto.txt ubicado en un servidor en modo de escritura para
añadir algo al final
```

Nota: diferentes familias de sistemas operativos tienen diferentes convenciones para el final de línea. Cuando escribe un fichero de texto y quiere insertar un salto de línea, necesita usar el carácter o caracteres correctos de final de línea para su sistema operativo. Los sistemas basados en Unix usan \n como el carácter de final de línea, los sistemas basados en Windows usan \r\n como caracteres de final de línea y los sistemas basados en Macintosh usan \r como carácter de final de línea.

Si se usan los caracteres de final de línea erróneos cuando escribe ficheros, puede suceder que otras aplicaciones que abran esos ficheros presenten su contenido de forma rara e inconexa.

Se puede usar 'b' para forzar el modo de escritura o lectura binario, lo cual no traducirá su información. Para usar estas banderas, especifique 'b' como el último carácter del parámetro mode.

```
$gestor = fopen("/home/archivo.gif", "rb"); //lectura en modo binario
```

Cierre de un fichero

Cuando se termina de operar con un fichero, lo correcto es cerrarlo para impedir errores e inconsistencias en el sistema de ficheros. Para este proceso usaremos la función **fclose()**.

*bool **fclose** (resource `$gestor`)*

Esta función devolverá TRUE si el archivo se ha cerrado correctamente o FALSE en caso contrario. $gestor deberá ser un puntero a un fichero abierto.

```
$gestor = fopen('archivo.txt', 'r');
fclose($gestor); // Cierra el archivo
```

Lectura de un fichero

Para leer información de un fichero usaremos la función fread();

string fread (resource $gestor , int $length)

fread() lee hasta length bytes desde el puntero al fichero referenciado por gestor. La lectura termina tan pronto como se encuentre una de las siguientes condiciones:

- length bytes han sido leídos
- EOF (fin de fichero) es alcanzado
- Un paquete se encuentra disponible o el tiempo límite del socket se agota (para flujos de red)

Ejemplo 1 - poner el contenido de un fichero en una cadena

```
<?php
  $nombre_fichero = "/usr/local/algo.txt";
  $gestor = fopen($nombre_fichero, "r");
  $contenido = fread($gestor, filesize($nombre_fichero));
  fclose($gestor);
?>
```

La función **filesize()** empleada en el ejemplo anterior resulta especialmente útil para calcular en tiempo de ejecución la longitud de un fichero.

Nota: **fread()** lee desde la poción actual del puntero al fichero. Use **fseek()** para localizar una posición, **ftell()** para encontrar la posición actual del puntero o **rewind()** para rebobinar la posición del puntero.

Advertencia: en sistemas en los que se diferencia entre archivos binarios y de texto (esto es, Windows) el fichero debe ser abierto con 'b' incluida en el parámetro modo de fopen().

Ejemplo 2 - lectura binaria de una imagen con fread()

```
<?php
  $filename = "c:\\files\\imagen.gif";
  $gestor = fopen($filename, "rb");
  $contenido = fread($gestor, filesize($filename));
  fclose($gestor);
?>
```

Advertencia: cuando se lee desde algo que no es un fichero local normal, como los flujos devueltos cuando se leen ficheros remotos o desde fopen() y fsockopen(), la lectura se detendrá después de que esté disponible un paquete. Esto significa que debería reunir la información en trozos como se muestra en los ejemplos de abajo.

Ejemplo 3 - Ejemplos de lectura remota con fread()

```
<?php
  $gestor = fopen("http://www.example.com/", "rb");
  $contenido = '';
  while (!feof($gestor)) {
     $contenido .= fread($gestor, 8192);
  }
  fclose($gestor);
?>
```

En el ejemplo anterior hacemos uso de la función **feof()**, función que os permite comprobar si el puntero está al final del archivo.

El ejemplo siguiente sería una alternativa equivalente al anterior:

```
<?php
  // Para PHP 5 y superior
  $gestor = fopen("http://www.example.com/", "rb");
```

```
  $contenido = stream_get_contents($gestor);
  fclose($gestor);
?>
```

Además de la función fread() existen otras alternativas dependiendo de cómo necesitemos leer el archivo. Si lo que se necesita es el contenido del archivo, se puede usar la función **file_get_contents().** Si queremos cada una de las líneas del archivo en una matriz (array) se puede usar el comando **file().**

Ejemplo 4 – lectura directa del contenido

```
<?php
$contenido = file_get_contents('datos.txt');
echo $contenido;
?>
```

file_get_contents() es la manera preferida de transmitir el contenido de un fichero a una cadena y lo hace de forma directa de principio a fin. Usa técnicas de mapeado de memoria, si está soportado por su sistema operativo, para mejorar el rendimiento.

Ejemplo 5 – lectura línea a línea

```
<?php
  $lineas = file('datos.txt');
  foreach ($lineas as $numero => $linea) {
      $numero_de_linea = $numero + 1;
      echo "Línea $numero_de_linea: $linea";
}
?>
```

El resultado es:

```
Línea 1: Soy la primera línea de un archivo
Línea 2: Soy la segunda línea de un archivo
Línea 3: Si dijera que soy la cuarta línea del archivo, estaría
mintiendo
```

Esta función lee el archivo completo como un array. Cada item en la matriz corresponde a una línea en el archivo.

También existe al función **fgets()** que es capaz de leer una única línea de nuestro fichero.

Moverse dentro de un fichero

PHP provee de varias herramientas para permitir que nos desplacemos dentro de un fichero, aquí veremos un par de ellas:

Rewind(): básicamente nos permite rebobinar, es decir posicionar el punto al principio de un archivo. Resulta especialmente útil para reescribir al principio de un archivo o para iniciar su lectura.

Ejemplo #1 Ejemplo de sobrescritura con rewind()

```
<?php
   $gestor = fopen('salida.txt', 'r+');

   fwrite($gestor, 'Una sentencia realmente larga.');
   rewind($gestor);
   fwrite($gestor, 'Foo');
   rewind($gestor);

   echo fread($gestor, filesize('salida.txt'));

   fclose($gestor);
?>
```

El resultado del ejemplo sería algo similar a:

```
Una sentencia realmente larga.
```

Lamentablemente esta instrucción sólo nos permite ir al principio del fichero, no desplazarnos en una cierta cantidad de bytes. Para eso se suele utilizar la función fseek().

fseek(): nos permite desplazar el puntero del fichero en una dirección u otra según cierta cantidad de bytes:

```
<?php

$fp = fopen('fichero.txt', 'r');

// leer alguna información
$data = fgets($fp, 4096);

// volver al principio del fichero igual que rewind($fp);
fseek($fp, 0);

?>
```

Nota: si se ha abierto un fichero en modo de adición (*a* o *a+*), cualquier información que se escriba en el fichero será siempre añadida, sin importar la posición, y el resutado de llamar a **fseek()** será indefinido.

Para movernos adecuadamente dentro de un fichero PHP proporciona funciones como **ftell()** que nos dice la posición actual del puntero y **feof()**, que nos permite comprobar si el puntero está al final del archivo. También la función **filesize()**, capaz de decirnos el tamaño en bytes del archivo, puede resultarnos útil para una búsqueda más eficiente.

Escritura de un fichero

La escritura de ficheros en PHP se realiza principalmente con la función **fwrite(),** a quien en algunos libros y referencias también se la puede encontrar como fputs(), que es un alias de la misma con idéntico funcionamiento.

La función **fwrite()** proporciona la funcionalidad de escritura de un archivo en modo binario seguro.

*int **fwrite** (resource `$gestor` , string `$string` [, int `$length`])*

fwrite() escribe el contenido de `string` al flujo de archivo apuntado por `gestor`. **fwrite()** devuelve el número de bytes escritos, o **`FALSE`** si se produjo un error.

```
<?php
   $file = fopen("test.txt","w");
   echo fwrite($file,"Hello World. Testing!");
   fclose($file);
?>
```

El resultado será 21.

En sistemas en los que se diferencia entre archivos binarios y de texto (esto es, Windows) el archivo debe ser abierto con 'b' incluida en el parámetro modo de **fopen()**.

Ejemplo 2 – Añadir contenido a un fichero existente

```
<?php
$nombre_archivo = 'prueba.txt';
$contenido = "Agregar esto al archivo\n";
$gestor = fopen($nombre_archivo, 'a');

// En nuestro ejemplo estamos abriendo $nombre_archivo en modo de
adición, de manera que el puntero se encuentra al final del archivo,
por lo que es ahí es donde se ubicará $contenido

   // Escribir $contenido a nuestro archivo abierto.
   if (fwrite($gestor, $contenido) === FALSE) {
       echo "No se puede escribir al archivo ($nombre_archivo)";
       exit;
   }else {
      echo "La información se añadió con éxito";
   }

fclose($gestor);

?>
```

Borrar un fichero

La función para eliminar un archivo es **unlink**() si bien también existe su alias delete():

bool unlink (string $filename)

La función unlink() Elimina filename. Devuelve TRUE en caso de éxito o FALSE en caso de error.

6.2 Gestión de sistemas de ficheros y directorios

PHP provee también de algunas herramientas para poder trabajar con archivos y directorios dentro de sistema de ficheros del servidor. Enumeraremos algunas de las más importantes:

mkdir ($pathname) — Crea un directorio especificado por pathname. El modo predeterminado es 0777, lo que significa el acceso más amplio posible. Si se desea otro modo, este puede especificarse como un segundo argumento.

```
mkdir("/ruta/a/mi/directorio", 0700);
```

rmdir ($dirname) — Intenta eliminar el directorio nombrado por `dirname`. El directorio debe estar vacío, y los permisos relevantes deben permitirlo. Un error de nivel **`E_WARNING`** será generado si se produce un error.

chown ($filename, $user)— Intenta cambiar el propietario del fichero `filename` por el usuario `user`. Sólo el superusuario puede cambiar el propietario de un fichero.

copy ($source, $dest) — Copia un fichero

Realiza una copia del fichero `source` a `dest`.

```
$fichero = 'ejemplo.txt';
$nuevo_fichero = 'ejemplo.txt.bak';

if (!copy($fichero, $nuevo_fichero)) {
    echo "Error al copiar $fichero...\n";
}
```

rename ($oldname, $newname) — Intenta renombrar `oldname` a `newname`, moviéndolo a otro directorio si fuera necesario. Si `newname` existe, lo sobrescribirá. Esta función a menudo es también usada para mover archivos de un directorio a otro:

```
rename("/tmp/archivo_tmp.txt", "/home/user/login/docs/archivo.txt");
```

move_uploaded_file ($filename, $destination) — Mueve un archivo subido a una nueva ubicación. Esta función intenta asegurarse de que el archivo designado por `filename` es un archivo subido válido (lo que significa que fue subido mediante el mecanismo de subida HTTP POST de PHP). Si el archivo es válido, será movido al nombre de archivo dado por `destination`.

Devuelve TRUE en caso de éxito.

Si filename no es un archivo válido subido, no sucederá ninguna acción, y move_uploaded_file() devolverá FALSE.

Si filename es un archivo subido válido, pero no puede ser movido por algunas razones, no sucederá ninguna acción, y move_uploaded_file() devolverá FALSE. Adicionalmente, se emitirá un aviso.

is_dir — Indica si el nombre de archivo es un directorio

is_file — Indica si el nombre de fichero es un fichero normal

is_executable — Indica si el nombre de archivo es ejecutable

is_readable — Indica si un fichero existe y es legible

7. Programación orienta a objetos con PHP

Aunque hasta ahora no lo hemos abordado de esta forma, PHP es un lenguaje orientado a objetos. Esto significa que posee un potente mecanismo para desarrollar nuestro código y usar el de otros de una forma más óptima y cercana a los mecanismos de programación contemporáneos con los que muchos programadores pueden estar ya familiarizados.

Para aquellos desarrolladores que no tengan una noción muy clara sobre lo que implica la POO comenzaremos explicando los fundamentos principales de este tipo de programación tan extendido en el desarrollo de aplicaciones, tanto web como de cualquier otro tipo.

7.1 Fundamento de la programación orientada a objetos

La programación orientada a objetos o POO (OOP según sus siglas en inglés) es un paradigma de programación centrada precisamente en el uso de objetos para diseñar aplicaciones y programas informáticos. La POO es una manera especial de pensar, en ocasiones de forma muy similar a cómo lo haríamos en la vida real, lo que implica que a veces resulte subjetiva respecto a quien programa, de manera que la forma de hacer las cosas puede ser diferente según el programador.

LA POO implica **abstraer** el problema que debemos abordar de manera que podamos aislar de forma **atómica** los protagonistas de este en forma de **objetos**. Estos contarán con una serie de **propiedades** que los definan y sobre los que podremos realizar determinado tipo de operaciones conocidas como **métodos**. Estas definiciones y funcionalidades tienden a definirse en lo que se conoce como **clases**, es decir, una plantilla que engloba (encapsula) todas las particularidades de un objeto (tanto propiedades como métodos) y a partir de las cuales se **instanciarán** estos. Entenderemos por instanciar al proceso de crear un objeto que responde a las características definidas en una clase, lo que tradicionalmente haremos mediante un método especial llamado **constructor**. Esto facilita la posibilidad de poder hacer uso de más de un objeto simultáneamente en nuestra aplicación sin que interfieran unos con otros.

La POO proporciona, además, funcionalidades adicionales como la **encapsulación** (acceso restringido a la definición de una clase), **herencia** (la posibilidad de desarrollar clases y objetos a partir de otros ya existentes) y el **polimorfismo** (la posibilidad de dotar de adaptabilidad a nuestros objetos pudiendo comportarse de una forma u otra frente a distintas situaciones). Estas funcionalidades pueden resultar de una enorme ayuda para la reutilización de código y la optimización de recursos de todo tipo.

En los próximos apartados procederemos a estudiar con más calma la mayoría de los conceptos aquí expuestos desde el punto de vista de PHP. A partir de PHP 5, el modelo de objetos ha sido reescrito para mejorar el rendimiento y permitir mayor funcionalidad. Este fue un cambio importante a partir de PHP 4. PHP 5 tiene un modelo de objetos completo que incluye funcionalidades como la visibilidad, los constructores, los destructores, las clases y métodos abstractos y finales, los interfaces, la clonación y la determinación de tipos.

7.2 Clases, métodos y propiedades

Podemos entender a un objeto como un elemento con un tipo de dato especial y complejo que se corresponderá con uno de los agentes implicados en nuestra aplicación. Así, en sentido estricto, podemos decir que una clase es un tipo de dato que contiene a modo de estructura un conjunto de variables y funciones asociadas todas ellas a ese objeto en cuestión. Desde este punto de vista, una clase nos permitiría crear tantos objetos como necesitemos en nuestra aplicación, lo que se conoce como instanciar.

En PHP podemos definir una clase así:

```
Class PaginaWeb {
      public $titulo;
      Function getTitulo () {
            Return $this->titulo;
      }
}
```

Mediante la palabra reservada class definimos una clase completa. Dentro se definen las propiedades y los métodos de los objetos, cosa que haremos mediante variables y funciones respectivamente. El ´termino public que antecede a la propiedad $titulo hace referencia al nivel de visibilidad de las propiedades y también de los métodos, algo que estudiaremos más adelante. Por otra parte siendo el nivel por defento podemos no incluirlo si no queremos, aunque se recomienda hacerlo como buena práctica. En los siguientes ejemplos, en pro de la claridad de exposición y por facilitar la lectura evitaremos incluirlo en los métodos hasta estudiarlos con detenimiento en el apartado correspondiente.

La función getTitulo responde a la definición de un método específico para los objetos de esta clase que devolverá el título de la página web que estemos creando a partir de esta clase. La pseudovariable *$this* está disponible cuando un método es invocado dentro del contexto de un objeto. *$this* es una referencia al objeto invocador (usualmente el objeto al cual el método pertenece). Por ello, este método, invocado desde un objeto de clase PaginaWeb, devolverá la propiedad título de la página, es decir, del objeto que llama al método.

Los métodos no tienen por qué devolver siempre algún tipo de valor, en ocasiones servirán para cambiar ciertas propiedades de los objetos. Veámoslo a modo de ejemplo con una versión extendida de la clase anterior:

```
Class PaginaWeb {
      public $titulo;

      Function setTitulo ($titulo= "título por defecto") {
            $this->titulo= $titulo;
      }

      Function getTitulo () {
            Return $this->titulo;
```

```
	}

	Function __construct ($titulo) {
		$this->setTitulo ($titulo);
	}

	Function cabecera () {
		Echo ("<html><head><title>");
		Echo $this->titulo;
		Echo ("</title></head><body>");
	}

	Function cuerpo () {
		Echo ("<p>Este es el cuerpo de la web</p>");
	}

	Function pie() {
		Echo ("</body></html>");
	}

	Function mostrarPagina () {
		$this->cabecera();
		$this->cuerpo();
		$this->pie();
	}
}
```

Nuestra clase, más desarrollada que la versión anterior, incorpora varios métodos nuevos encargados de desarrollar las distintas páginas de una web. Estas funciones (cabecera, cuerpo y pie) serán llamadas desde el método mostrarPagina(), por lo que este será el método principal que usaremos con nuestros objetos para poder crear una web completa.

¡Ojo!: El uso de $this es obligatorio aunque estemos dentro de la propia clase

Instancia de clase

Instanciar un objeto es esencialmente crear un objeto perteneciente a esta clase y será la vía para que nuestro código funcione. Esto lo haremos utilizando la instrucción new:

```
$pagina = new PaginaWeb();
```

De esta manera hemos creado un objeto llamado $pagina perteneciente a la clase PaginaWeb, de manera que este objeto tendrá acceso a las propiedades y métodos definidos en dicha clase. Podremos acceder a ellos así:

```
$pagina->setTitulo("Mi nueva web");
$pagina->mostrarPagina();
```

Aunque de esta manera ya podríamos trabajar sin problemas con este objeto, para tener una definición más correcta y robusta lo ideal es definir, además, un método constructor.

Método constructor

Un constructor es un método especial que se desarrolla con la intención de inicializar todas las cosas que necesitamos para empezar a usar un objeto en condiciones y evitar así tener que hacer uso de llamadas a otros métodos. PHP, al igual que la mayoría de los demás lenguajes con soporte para objetos proporciona un constructor por defecto cuando se instancia a un objeto mediante la instrucción new, pero siempre tenemos la posibilidad de crear el nuestro el cual, de existir, sustituirá al que incorpora el lenguaje. Los constructores se deben crear como una función con el siguiente nombre: **__construct()**.

```
Function __construct ($titulo) {
      $this->setTitulo ($titulo);
}
```

En nuestro ejemplo el constructor se encargará, mediante el paso del parámetro $titulo, de insertar directamente el título de la página cuando creamos el objeto, por lo que nos ahorramos tener que invocar al método setTitulo como hicimos antes. De esta manera, la creación del objeto quedaría finalmente así:

```
$pagina= new PaginaWeb ("Mi nueva web");
$pagina->mostrarPagina();
```

¡Ojo!: El uso del nombre de la propia clase como constructor se considera obsoleto en PHP 7

Método destructor

Por supuesto, de forma similar a como lo hacen otros lenguajes, PHP también provee al desarrollador la posibilidad de crear un destructor mediante el método **__destruct().** Un destructor es un método que se encargará de realizar las operaciones oportunas justo antes de que el objeto sea destruido. El método destructor será llamado tan pronto como no haya otras referencias a un objeto determinado o en cualquier otra circunstancia de finalización, por ello definirlo en lugar de dejarlo en manos del propio sistema puede proporcionarnos más de una ventaja. Como ejemplo pensemos en una clase para controlar una base de datos, en cuyo caso nos puede interesar que al eliminar el objeto se cierre la conexión con esta.

```
function __destruct() {
      unset($this->titulo); //eliminamos la propiedad $titulo
}
```

7.3 Herencia

La herencia en la programación orientada a objetos es un concepto que permite desarrollar una clase a partir de otra de manera que la clase descendiente sea capaz de reutilizar código previamente desarrollado y verificado, heredando así parte de la funcionalidad de la clase padre. La herencia facilita la creación de objetos a partir de otros ya existentes e implica que una subclase obtiene todo el comportamiento (métodos) y eventualmente los atributos (variables) de su superclase además de poder incorporar los suyos propios. Así pues una clase hija o descendiente adquiere las siguientes propiedades:

- Todas las variables miembro de la clase padre
- Todas las funciones miembro de la clase padre con idéntico funcionamiento
- Posibilidad de definir nuevas variables y funciones (eventualmente incluso redefinir algunas de las heredadas)

Para desarrollar una clase descendiente de otra utilizamos la palabra **extends** a continuación del nombre de la nueva clase y seguida del nombre de la clase padre. Como ejemplo vamos a crear una nueva clase a partir de PaginaWeb que permita desarrollar páginas con un formulario:

```
Class PaginaWebForm extends PaginaWeb
{
	Function formInicio ($accion) {
		Echo "<form action\"=$accion\">";
	}

	Function formFin() {
		Echo "</form>";
	}

	Function formCajaTexto ($nombre) {
		Echo "$nombre <input type=\"text\" name=\"$nombre\"/>";
	}

	Function formEnvio () {
		Echo "<input type=\"submit\" name=\"submit\"/>";
	}

	Function mostrarPagina () {
		$this->cabecera (); //heredada de PaginaWeb
		$this->formInicio("procesar.php");
		$this->formCajaTexto("nombre");
		$this->formEnvio();
		$this->formFin();
		$this->pie(); //heredada de PaginaWeb
	}
}
```

Como se puede observar en este ejemplo, la nueva clase PaginaWebForm implementa nuevos métodos para alcanzar su objetivo pero deja otros sin desarrollar como cabecera() y pie() sencillamente porque no hace falta, dado que ya existen en la clase padre y por ello puede usarlas directamente. Lo mismo sucede con los métodos getTitulo o setTitulo (que en esta clase no se usan de forma explícita) y con el método constructor, que al ser igual, no es necesario desarrollarlo, ya que tomará automáticamente el constructor de su padre al instanciar objetos de esta clase:

```
$formulario = new PaginaWebForm ("Web con formulario");
$formulario->mostrarPagina();
```

Los constructores del padre no son llamados implícitamente si la clase descendiente se define un constructor. Para ejecutar un constructor parent, se requiere invocar a parent::__construct() desde el constructor de la clase descendiente. Si el descendiente no define un constructor, entonces se puede heredar de la clase padre como un método de clase normal (si no fue declarada como privada).

Operador de resolución de ámbito (::)

A veces es útil hacer rereferencia a variables o funciones en clases base, o referenciar funciones en clases que aún no tienen instancias. El Operador de Resolución (::) también conocido como Paamayim Nekudotayim (significa doble-dos-puntos en Hebreo) se usa para ello.

- self: Cuando queramos acceder a una constante o método estático desde dentro de la clase.
- parent: Cuando queramos acceder a una constante o método de una clase padre.

```php
<?php
class OtherClass extends MyClass
{
    public static $my_static = 'variable estática';

    public static function doubleColon() {
        echo parent::CONST_VALUE . "\n";
        echo self::$my_static . "\n";
    }
}

$classname = 'OtherClass';
$classname::doubleColon(); // A partir de PHP 5.3.0

OtherClass::doubleColon();
?>
```

$this vs self::

Se usa $this para hacer referencia al objeto (instancia) actual, y se utiliza self:: para referenciar a la clase actual. Se utiliza $this->nombre para nombres no estáticos y se utiliza self::nombres para nombres estáticos.

Final

PHP 5 introduce la nueva palabra clave final, que impide que las clases hijas sobrescriban un método, antecediendo su definición con final. Si la propia clase se define como final, entonces no se podrá heredar de ella. Aplicado a nuestro ejemplo, si el método mostrarPagina() de la clase PaginaWeb estuviera definido como final, la llamada a este método realizada desde el objeto $formulario de la clase descendiente PaginaWebForm devolvería un error.

Herencia múltiple

PHP no ofrece herencia múltiple de forma directa, pero sí permite hacer uso de lo que se conoce como herencia encadenada, es decir, se puede conseguir que una clase herede de dos clases padres siempre que una de las clases padre herede de la anterior.

```
Class ClaseA () {…}
Class ClaseB extends ClaseA () {…}
Class ClaseC extends ClaseB () {…}
```

Así conseguiríamos que ClaseC sea descendiente tanto de ClaseA como de ClaseB.

7.4 Visibilidad

El término visibilidad hace referencia a cómo pueden ser accedidos los métodos y las propiedades de una clase. Mientras no se especifique otra cosa, tanto los métodos como las propiedades son siempre públicos, es decir, podrán ser accedidos desde fuera del objeto. En nuestro ejemplo esto implica que podríamos ver la propiedad $título de la siguiente manera:

```
$tituloWeb= $pagina->titulo;
```

Esto no es muy elegante y, hasta cierto punto, se considera una mala práctica de programación dado que provoca vulnerabilidades en nuestro código. Para consultar el título de nuestra web lo correcto es hacerlo mediante el método creado exclusivamente para ello:

```
$tituloWeb=$pagina->getTitulo();
```

Como norma general las propiedades de los objetos deben estar ocultas al entorno exterior y tomar sus valores desde funciones definidas dentro de la clase (métodos del objeto). Por esto existen varios niveles de visibilidad que nos permiten dotar a nuestras clases de cierta protección.

La visibilidad de una propiedad o método se puede definir anteponiendo una de las palabras claves **public**, **protected** o **private** en la declaración. Los miembros de clases declarados como public pueden ser accedidos desde cualquier lado. Los miembros declarados como protected, sólo desde la misma clase, desde las clases que hereden de ella y desde las clases padre. Aquellos miembros definidos como private, únicamente pueden ser accedidos desde la clase que los definió. Estudiemos estos niveles con más detenimiento.

Métodos privados

El nivel de visibilidad privado es el más restrictivo de todos. Asume que **un método o propiedad privados sólo podrán ser accedidos desde la clase que los define**. Las clases que hereden de una clase con métodos privados no tendrán acceso a esos métodos y tampoco será posible hacer llamadas a ellos desde fuera del objeto. Para definir un método o propiedad como privado bastará con anteponer la palabra private en su declaración:

```
Class PaginaWeb
{
      private $titulo;
      Function __construct ($titulo) {
            $this->setTitulo ($titulo);
      }
      private function setTitulo ($titulo= "título por defecto") {
            $this->titulo= $titulo;
      }
}
```

Así tanto nuestra propiedad título como el método setTitulo() serán privados de manera que ni siquiera la clase PaginaWebForm() que hereda de PaginaWeb() podría acceder al método setTitulo() porque obtendríamos un error de acceso ilegal salvo a través del constructor.

Métodos protegidos

Los métodos protegidos son menos restrictivos que los privados, dado que **sí permiten el acceso tanto a clases heredadas como a clases padre**. El ejemplo anterior podría modificarse de la siguiente manera:

```
Class PaginaWeb
{
      private $titulo;
      protected Function __construct ($titulo) {
            $this->setTitulo ($titulo);
      }
      protected function setTitulo ($titulo= "título por defecto") {
            $this->titulo= $titulo;
      }
}
```

De esta forma sí tendríamos acceso a setTitulo desde PaginaWebForm()

Métodos públicos

Como hemos mencionado antes**, los métodos y propiedades públicos pueden ser accedidos desde el exterior**, lo cual no siempre es recomendable. Como buena práctica de programación, se recomienda que las propiedades se definan siempre como privadas o protegidas, para que no puedan ser accedidas desde fuera del objeto. Además deben llevar asociados los métodos de acceso a ellas, normalmente setVariable() y getVariable(), como públicos (o cuando menos protegidos) para establecer y leer dicha propiedad desde otros ámbitos.

```
Class PaginaWeb
{
      private $titulo;

      function __construct ($titulo) {
            $this->setTitulo ($titulo);
      }
      public function setTitulo ($titulo= "título por defecto") {
            $this->titulo= $titulo;
      }
      public function getTitulo () {
            Return $this->titulo;
      }
}
```

Los métodos y propiedades definidos por defecto, es decir, sin visibilidad especificada o mediante la palabra clave var son considerados como públicos.

7.5 Interfaces y clases abstractas

PHP dispone tanto de clases abstractas como de interfaces para facilitar y enriquecer el desarrollo de clases. La idea subyacente en ambos casos es la de poder desarrollar algo similar a una plantilla o base para el posterior desarrollo de clases.

Interface

Un interface no es más que una clase con sus propiedades y métodos definidos (declarados) pero no desarrollados. Su funcionalidad es la de determinar el funcionamiento de una clase, es decir, funciona como un molde o como una plantilla, pero no incluye el código de los métodos desarrollado aunque sí sus argumentos. La idea es que un interface presente una clase que después cualquiera podría desarrollar (de una forma u otra), lo que resulta de una gran ayuda en el desarrollo de proyectos en grupo o colaborativos. Todos los métodos declarados en una interfaz deben ser públicos, ya que ésta es la naturaleza de una interfaz.

Los interfaces son definidos utilizando la palabra clave **interface**, de la misma forma que con clases estándar, pero sin métodos que tengan su contenido definido.

```
interface IPaginaWeb {
      Var $titulo;
      Public function setTitulo ($titulo= "titulo por defecto");
      Public function getTitulo ();
      Public function cabecera ();
      Public function cuerpo ();
      Public function pie();
      Public function mostrarPagina ();
}
```

La clase que implemente una interfaz debe utilizar exactamente las mismas estructuras de métodos que fueron definidos en la interfaz. De no cumplir con esta regla, se generará un error fatal. Una clase que desarrolla los métodos y propiedades de un interface debe incluir el operador **implements** tras su nombre seguido del de la clase interface aludida:

```
Class PaginaWeb implements IPaginaWeb {…}
```

Todos los métodos en una interfaz deben ser implementados dentro de la clase; el no cumplir con esta regla resultará en un error fatal. Las clases pueden implementar más de una interfaz si se deseara, separándolas cada una por una coma.

Clase abstracta

La idea de una clase abstracta es la de definir y en ocasiones desarrollar determinadas propiedades y métodos que van a ser utilizados, principalmente, por clases descendientes que hereden sus definiciones.

Las clases definidas como abstractas no están pensadas para proporcionar objetos, por lo que no se pueden instanciar y cualquier clase que contiene al menos un método abstracto debe ser

definida como tal. Los métodos definidos como abstractos simplemente declaran la firma del método, pero no pueden definir la implementación.

```
abstract class animales {
      abstract function nacer();
}

class mamiferos extends animals {
      public function nacer(){
            echo "los mamíferos nacen del parto de la madre";
      }
}
class oviparos extends animals {
      public function nacer(){
            echo "los ovíparos nacen de los huevos";
      }
}
```

En el ejemplo anterior podemos ver que la clase animales sólo incluye una definición, de forma similar a como lo haría un interface aunque, a diferencia de estos, una clase abstracta también podría incluir métodos implementados si fuera necesario. Hemos definido la clase animales como abstracta porque no vamos a crear objetos de tipo animales, aunque sí lo haremos de sus descendientes, mamíferos y ovíparos.

SI intentásemos instanciar un objeto de la clase animales obtendríamos un error fatal:

```
$animales = new animales();
Fatal error: Cannot instantiate abstract class animales
```

Cuando se hereda de una clase abstracta, todos los métodos definidos como abstractos en la declaración de la clase madre deben ser definidos en la clase hija, de lo contrario obtendríamos un error. Por otra parte, si en una clase abstracta incluimos métodos no abstractos e implementados, estos podrían ser usados directamente por sus descendientes sin necesidad de ser definidos de nuevo en esta, como sucedería con cualquier otro caso de herencia.

Los métodos abstractos deben ser definidos en las clases descendientes con la misma (o con una menos restrictiva) visibilidad. Por ejemplo, si el método abstracto está definido como protegido, la implementación de la función debe ser definida como protegida o pública, pero nunca como privada. Por otra parte, las firmas de los métodos tienen que coincidir, es decir, la implicación de tipos y el número de argumentos requeridos deben ser los mismos.

7.6 Funciones para manejo de clases

Incluimos aquí un breve listado de algunas funciones de utilidad para la obtención de información acerca de clases, herencia, métodos y propiedades:

Get_class ($objeto) – devuelve el nombre de la clase a la que pertenece el objeto pasado como parámetro

Get_parent_class ($objeto) – devuelve el nombre de la clase padre

class_exists ($clase) — Verifica si la clase ha sido definida

interface_exists ($nombre_interface) — Comprueba si una interfaz ha sido definida

get_declared_classes () — Devuelve una matriz con los nombres de las clases definidas

get_declared_interfaces () — Devuelve un array con todas las interfaces declaradas

is_a ($objeto, $clase) — Comprueba si un objeto es de una clase o tiene esta clase como uno de sus padres

is_subclass_of ($objeto, $clase) — Verifica si el objeto tiene esta clase como uno de sus padres

method_exists($objeto, $nombremetodo) — Comprueba si existe un método de una clase

property_exists($objeto, $nombrepropiedad) — Comprueba si el objeto o la clase tienen una propiedad

8. PHP y las bases de datos

PHP ofrece una gran variedad de herramientas y controladores para manejarse con bases de datos de casi cualquier tipo. Esto comprende bibliotecas y extensiones completas con multitud de mecanismos de acceso, control y manejo de sistemas que, en su mayoría se basan en el lenguaje de consultas estructurado SQL. Sobre esta sintaxis funcionan la práctica totalidad de los motores implementados en PHP: mysql, mysqli, sqlite, PDO, etc.

En este capítulo nos centraremos exclusivamente en el tratamiento de bases de datos de tipo relacional basados en SQL, concretamente en el popular sistema gestor **MySQL**, con el cual PHP es capaz de comunicarse con gran soltura y precisión dado que sus desarrollos parecen haber fluido en paralelo a lo largo de los últimos años. Así, estudiaremos de establecer conexiones con bases de datos y realizar consultas, actualizaciones, inserciones y borrado de sus datos desde un entorno web.

A la hora de desarrollar un sitio web en PHP que se apoya en una base de datos relacional hospedada en un servidor MySQL, buena parte del trabajo iniciar tiende a llevarse a cabo desde el entorno **phpMyAdmin**. Cosas como por ejemplo el diseño o el despliegue de la base de datos (si esta ha sido creada en otro entorno y se desea importar después), la creación de usuarios con acceso a la misma y la gestión de sus permisos resultará más fácil llevarlo a cabo desde este entorno, dado que nos evitará desarrollar centenares de líneas de código para un uso muy exclusivo y limitado. Es decir que en la mayoría de los casos el desarrollador de PHP dejará buena parte de ese trabajo inicial en manos del administrador de la base de datos, mientras que este creará usuarios para que accedan a la base de datos desde la web a desarrollar.

Llegados a este punto se asume que el alumno tiene un mínimo conocimiento del lenguaje SQL al igual que sobre el funcionamiento y estructura de las bases de datos relacionales.

8.1 Introducción a MySQL, phpMyAdmin y mysqli

MySQL es un sistema de gestión de bases de datos relacional, multihilo y multiusuario con más de seis millones de instalaciones. Por un lado se ofrece bajo la GNU GPL para cualquier uso compatible con esta licencia, pero para aquellas empresas que quieran incorporarlo en productos privativos deben comprar a la empresa una licencia específica que les permita este uso.

MySQL es una base de datos muy rápida en la lectura cuando utiliza el motor no transaccional MyISAM, pero puede provocar problemas de integridad en entornos de alta concurrencia en la modificación. En aplicaciones web hay baja concurrencia en la modificación de datos y en cambio el entorno es intensivo en lectura de datos, lo que hace a MySQL ideal para este tipo de aplicaciones. Por ello MySQL es muy utilizado en aplicaciones web de todo tipo, por lo cual se incluye junto con el servidor web Apache en distribuciones de servidor libre como XAMPP y similares sobre distintas plataformas.

phpMyAdmin

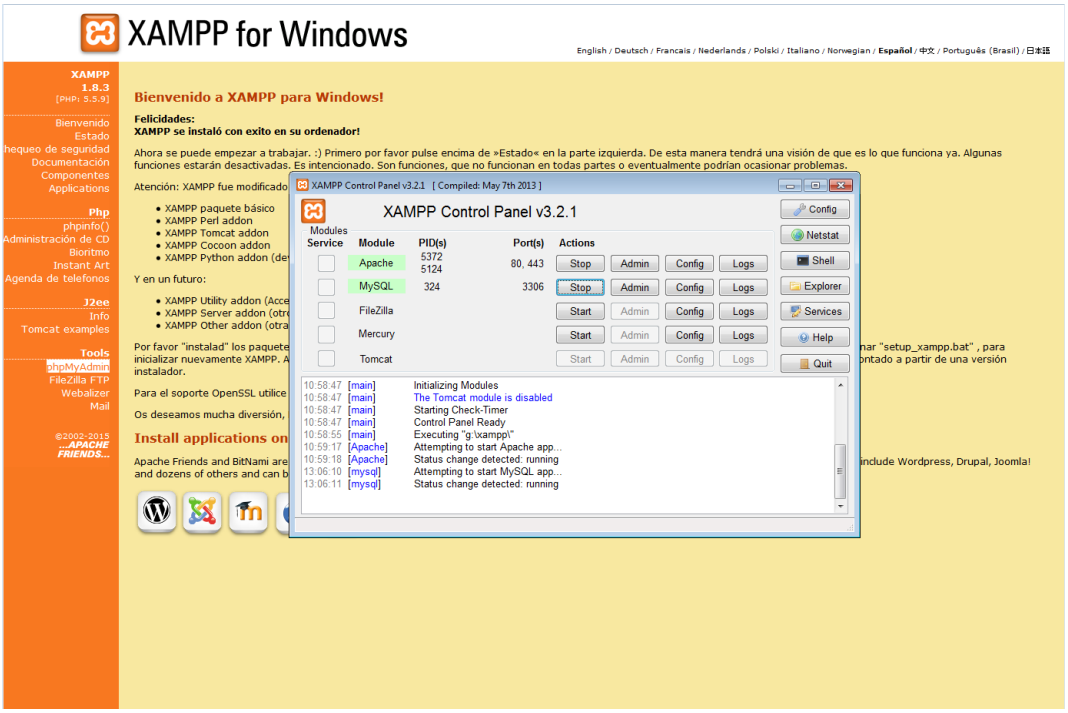
phpMyAdmin es un entorno web desarrollado en PHP que permite la administración de un servidor MySQL y gestión completa de cualquier base de datos relacional desplegada sobre dicho servidor.

Actualmente puede crear y eliminar Bases de Datos, crear, eliminar y alterar tablas, borrar, editar y añadir campos, ejecutar cualquier sentencia SQL, administrar claves en campos, administrar privilegios, exportar datos en varios formatos y está disponible en 62 idiomas. Se encuentra disponible bajo la licencia GPL Versión 2.

phpMyAdmin es también una herramienta muy apreciada por ser capaz de importar datos desde CSV y SQL y también exportar a varios formatos: CSV, SQL, XML, PDF, OpenDocument Text y Spreadsheet, Word, Excel, LaTeX y otros.

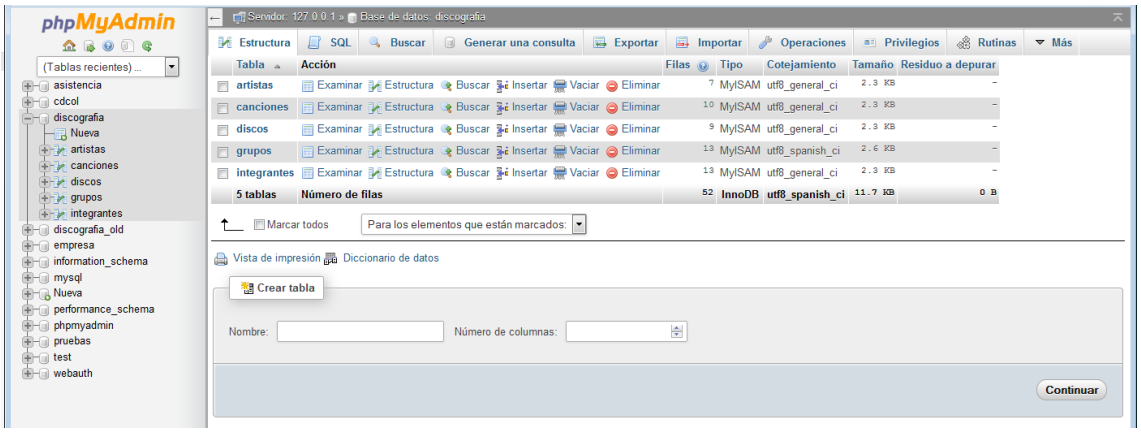
Como decíamos en la introducción de este tema, buena parte del trabajo inicial como el despliegue de la BBDD, la creación de usuarios, permisos y contraseñas tradicionalmente se desarrollará desde esta plataforma de manera que nuestra aplicación web acceda a la base de datos para consultar, actualizar, insertar o borrar datos.

En nuestra instalación local podremos acceder a phpMyAdmin siempre que tengamos en marcha el servicio MySQL:



Para un adecuado estudio de los métodos que veremos a continuación deberemos tener previamente desplegada una BBDD en nuestro servidor así como un usuario para el acceso vía web con su correspondiente contraseña. Dado que trabajaremos en local, casi todas las referencias al servidor se realizarán contra localhost.

Para la mayor parte de los ejemplos desarrollados a lo largo de este tema haremos uso de una base de datos previamente creada que aborda la información en torno a grupos musicales, sus integrantes y sus discografías, tal como se puede apreciar en la siguiente imagen:



Como se puede observar nuestra base de datos, de nombre **Discografía**, se compone de cinco tablas: artistas, discos, canciones, grupos e integrantes según la siguiente estructura:

- **Grupos** (**idgrupo**, nombre, nacionalidad, idioma, descripcion, website)
- **Artistas** (**idartista**, nombre, apellidos, nacionalidad, instrumento, biografia, website)
- **Integrantes** (**idgrupo, idartista**, rol)
- **Discos** (**iddisco**, titulo, anyo, sello_discografico, descripcion, cubierta, idgrupo)

- **Canciones** (**idcanción**, título, duración, autor, fecha_grabacion, lugar_grabacion, letra, notas, iddisco)

Adicionalmente para poder acceder a dicha base de datos desde nuestro código PHP usaremos dos usuarios distintos, uno sólo para realizar consultas llamado **discouser** con contraseña **discouser** y otro usuario con permiso para consultar, insertar, actualizar y borrar datos usuario **discoadmin** y la contraseña **discopass**.

Mysqli

Hasta la versión de PHP 5.5 el acceso a nuestra BBDD desde PHP se realizaba con el conjunto de instrucciones pertenecientes a la API mysql, pero esta extensión está obsoleta a partir de esa versión, por lo que no está recomendada para escribir código nuevo, ya que será eliminada en el futuro. En su lugar, se debería utilizar la extensión mysqli o PDO. En este tema estudiaremos la primera dado que su sintaxis y funcionamiento son prácticamente idénticos a los de la mencionada mysql por ser una versión mejorada de esta.

La extensión mysqli ofrece una interfaz dual. Soporta el paradigma de programación procedimental y también el orientado a objetos, cosa que la extensión mysql no ofrecía originalmente y por lo que mysqli resulta más potente y robusta en muchos sentidos.

Los usuarios que migren desde la extensión mysql antigua pueden preferir la interfaz procedimental. Esta interfaz es similar a la de la extensión antigua de mysql. En la mayoría de los casos, los nombres de funciones difieren únicamente por el prefijo. Algunas funciones de mysqli toman como primer argumento un gestor de conexión, mientras que las funciones similares de la antigua interfaz de mysql lo toman como el último argumento opcional.

8.2 Conexión a MySQL

Conectar con la base de datos de nuestro SGBD MySQL será el primer paso a realizar cuando deseemos tener acceso a nuestra BBDD. El proceso para esto pasa por conocer los datos relacionados con la ubicación del servidor, la base de datos a la que deseamos acceder y el nombre de usuario y contraseña con los que queremos conectarnos. Todo esto podremos hacerlo con la siguiente instrucción:

```
$mysqli = mysqli_connect("ejemplo.com", "user", "pass", "basedatos");
```

En su versión orientada a objetos esta operación se llevaría a cabo de la siguiente manera:

```
$mysqli = new mysqli ("ejemplo.com", "user", "pass", "basedatos");
```

Control de errores de conexión

A menudo el proceso de conexión debe ir adecuadamente vigilado para tener constancia en caso de que se produzca un error. Cuando se trata de mysqli la gestión de errores se realiza a dos niveles. Los errores de conexión utilizan recursos de diagnóstico distintos de otros errores de utilización de MySQL. El código de error se identifica en el caso de la programación por procesos mediante la función: **mysqli_connect_errno()** y la descripción de ese error se objetiene mediante **mysqli_connect_error()**:

```
$mysqli = new mysqli ("ejemplo.com", "user", "pass", "basedatos");

if (mysqli_connect_errno($mysqli)) {
    echo "Fallo al conectar: ". mysqli_connect_error();
}
```

En caso de optar por la versión orientada a objetos, los códigos de error de conexión y sus descripciones los obtendríamos de las propiedades **$objeto->mysqli_connect_errno** y **$objeto->mysqli_connect_error**:

```
$mysqli = new mysqli("localhost", "user", "pass", "basedatos");

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

Conviene observar que, a diferencia del caso de programación por procesos, aquí errorno y error no van seguidos de paréntesis. Esto se debe a que son propiedades del objeto $mysqli, no funciones encargadas de devolver un valor.

Otra propiedad interesante a tener en cuenta a la hora de presentar un posible fallo de conexión es **host_info,** el cual puede mostrarnos información sobre el estado de la conexión en sí:

```
echo $mysqli->host_info . "\n";
```

El resultado del ejemplo sería:

```
Localhost via UNIX socket
127.0.0.1 via TCP/IP
```

Dado que el establecimiento de la conexión con la BBDD se suele llevar a cabo con frecuencia desde distintas páginas, resulta una buena práctica almacenar este proceso en un archivo independiente que será invocado desde todas las páginas que lo requieran mediante un include, si bien en este caso suele ser más recomendado usar require, puedo que será preferable detener la ejecución de nuestro código si la conexión falla.

Cierre de la conexión

Otra operación importante relacionada con la conexión es la desconexión, algo que deberemos hacer al terminar nuestras operaciones:

```
mysqli_close ($mysqli);
```

O de nuevo en su versión OOP:

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

El proceso de conexión y cierre se debe realizar en cada página en la que accedamos a la BBDD, por lo que una práctica muy común consiste en agrupar parte de esta información (nombre de usuario, contraseña, BD, etc) en un archivo e incluirlo cada vez.

Otro proceso que en ocasiones también puede resultar práctico es el de realizar conexiones persistentes, es decir, mantener cada conexión abierta y lista de forma permanente durante la sesión de uso del visitante. Mysqli proporciona, a diferencia de otras API de acceso a BBDD, un control de conexión persistente gracias a lo que se conoce como una caché de conexiones.

Caché de conexiones

La extensión mysqli soporta conexiones persistentes a bases de datos, las cuales son un tipo especial de conexiones almacenadas en caché. Por defecto, cada conexión a una base de datos abierta por un script es cerrada explícitamente por el usuario durante el tiempo de ejecución o liberada automáticamente al finalizar el script. Una conexión persistente no. En su lugar, se coloca en una caché para su reutilización posterior, si una conexión es abierta al mismo servidor usando el mismo nombre de usuario, contraseña, socket, puerto y base de datos predeterminada. La reutilización ahorra gastos de conexión.

Cada procesos de PHP utiliza su propia caché de conexiones mysqli. Dependiendo de modelo de distribución del servidor web, un proceso PHP puede servir una o múltiples peticiones. Por lo tanto, una conexión almacenada en caché puede ser utilizada posteriormente por uno o más scripts.

Conexiones persistentes

El uso de conexiones persistentes se puede habilitar y deshabilitar usando la directiva de PHP **mysqli.allow_persistent**. El número total de conexiones abiertas por un script puede ser limitado con **mysqli.max_links**. El número máximo de conexiones persistentes por proceso de PHP puede restringirse con **mysqli.max_persistent**.

Sin embargo una queja común sobre las conexiones persistentes es que su estado no es reiniciado antes de su uso, lo que puede verse como un efecto secundario no deseado capaz de afectar a la seguridad de nuestra aplicación. Corresponde al usuario elegir entre comportamiento seguro o mejor rendimiento.

8.3 Recorrido y lectura de datos

Lo habitual cuando se desea leer información de una base de datos es enviar, en primer lugar, una sentencia SQL que realice la consulta deseada a la BBDD. En la mayoría de los casos esto se limita a crear la sentencia SQL como un string y enviársela al motor de la BBDD a través de nuestro objeto (en el caso de OOP) de forma similar a como se puede observar en el siguiente ejemplo:

```
$query=”SELECT * FROM tabla”;
$resultado = $mysqli->query($query);
```

Sin embargo lo habitual es que se ejecute junto a alguna alternativa de notificación en caso de que se produzca un error:

```
$resultado = $mysqli->query($query) or die('Consulta fallida: ' .
$mysqli-> mysqli_connect_error);
```

En ambos casos la variable $resultado albergará el resultado de la consulta. Esta tendrá un valor FALSE en caso de error; si se trata de una consulta del tipo SELECT, SHOW, DESCRIBE o EXPLAIN y resulta exitosa, mysqli_query() retornará un objeto **mysqli_result**. Para otras consultas exitosas de mysqli_query() retornará TRUE.

Lectura de resultados

Al margen de los posibles resultados booleanos la ejecución de una consulta no devuelve los resultados en un formato legible. Los valores devueltos requieren ser convertidos a un formato que sea interpretable por PHP o, en caso de envío directo al cliente, por JavaScript. Mysqli ofrece un conjunto de alternativas para proporcionar esa traducción del objeto mysqli_result dependiendo de cómo necesitemos tratar las respuestas. Una forma cómoda de hacer esto es interpretando cada fila del resultado como un array asociativo en el que los índices serán los nombres de los campos del resultado. Esto, por sí mismo, ya nos proporciona la posibilidad de imprimir datos por pantalla o realizar cálculos o búsquedas de cualquier tipo, pero también reconvertir la información en un formato JSON para el envío a JavaScript si fuera necesario.

Otra cuestión importante a tener en cuenta es que la respuesta de una consulta rara vez se limitará a dar una línea como resultado. Lo habitual es que una consulta devuelva varías filas, lo que implicará que el resultado deba ser recorrido de alguna manera plausible. Normalmente haremos esto ayudado de alguna estructura de repetición usando instrucciones como while, for e incluso foreach.

mysqli_result::fetch_row — Obtener una fila de resultados como un array enumerado. Obtiene una fila de datos del conjunto de resultados y la devuelve como un array enumerado, donde cada columna es almacenada en un índice del array comenzando por 0 (cero). Cada llamada subsiguiente a esta función devolverá la siguiente fila del conjunto de resultados, o NULL si no hay más filas.

```
if ($resultado = $mysqli->query($consulta)) {

    while ($fila = $resultado->fetch_row()) {
        echo "<p>$fila[1] - $fila[8] </p>";
    }
```

Este método no se suele usar con demasiada frecuencia dado que normalmente no resulta demasiado práctico trabajar con un array enumerado.

mysqli_result::fetch_assoc — Obtener una fila de resultado como un array asociativo. Devuelve un array asociativo de strings que representa a la fila obtenida del conjunto de resultados, donde cada clave del array representa el nombre de una de las columnas de éste; o NULL si no hubieran más filas en dicho conjunto de resultados.

```
if ($resultado = $mysqli->query($query)) {

    while ($fila = $resultado->fetch_assoc()) {
       echo "<p>".$fila["nombre"]." - ".$fila["titulo"]."</p>";
    }
}
```

También podríamos hacer uso de la instrucción foreach para recorrer los resultados, aunque para ello deberemos antes crear una variable de tipo array donde guardar todos los resultados:

```
$salida= array();
$salida= $resultado->fetch_all(MYSQLI_ASSOC);

foreach ($salida as $fila) {
      echo "<p>".$fila["nombre"]." - ".$fila["titulo"]."</p>";
}
```

mysqli_result::fetch_array — Obtiene una fila de resultados como un array asociativo, numérico, o ambos. Es una versión extendida de la función mysqli_fetch_row(). Además de guardar la información en los índices numéricos del array resultante, la función mysqli_fetch_array() también puede guardar la información en índices asociativos, utilizando los nombres de los campos del resultado como claves para su recorrido. Los nombres de los campos devueltos por esta función son sensibles a mayúsculas y minúsculas.

El método fetch_array acepta un argumento que define el formato en el que devolverá los datos. Este parámetro opcional es una constante que indica qué tipo de array debiera generarse con la información de la fila actual. Los valores posibles para este parámetro son las constantes **MYSQLI_ASSOC**, **MYSQLI_NUM** o **MYSQLI_BOTH**.

Al emplear la constante MYSQLI_ASSOC esta función se comportará de manera idéntica a mysqli_fetch_assoc(), mientras que con MYSQLI_NUM se comportará exactamente igual que la

función mysqli_fetch_row(). La última opción MYSQLI_BOTH creará un único array con los atributos de ambas dos.

```
if ($resultado = $mysqli->query($query)) {

    while ($fila = $resultado->fetch_array(MYSQLI_ASSOC)) {
       echo "<p>".$fila["nombre"]." - ".$fila["titulo"]."</p>";
    }
}
```

mysqli_result::fetch_object — Devuelve la fila actual de un conjunto de resultados como un objeto, donde los atributos del objeto representan los nombres de los campos encontrados en el conjunto de resultados.

```
if ($resultado = $mysqli->query($query)) {

      while ($obj = $resultado->fetch_object()) {
            echo "<p>$obj->nombre - $obj->titulo</p>";
      }
}
```

Otros resultados de interés que devuelven un solo valor:

mysqli_result::$num_rows — Obtiene el número de filas de un resultado

```
echo "<p>número de resultados: $resultado->num_rows</p>";
```

mysqli_result::$field_count — Obtiene el número de campos de un resultado

```
echo "<p>número de campos: $resultado->field_count</p>";
```

mysqli::$affected_rows — Obtiene el número de filas afectadas en la última operación SQL. Resulta especialmente interesante cuando se usan sentencias de tipo INSERT, UPDATE, REPLACE o DELETE, dado que así podemos observar el número de registros sobre los que se ha aplicado la operación.

```
echo "<p>Registros afectados: $resultado->affected_rows</p>";
```

mysqli_stmt::$sqlstate — Devuelve el error SQLSTATE de la operación de sentencia previa. El código de error consiste en cinco caracteres. '00000' significa sin error. Los valores están especificados por ANSI SQL y ODBC. Para una lista completa de los valores posibles, véase » http://dev.mysql.com/doc/mysql/en/error-handling.html.

```
if ($resultado = $mysqli->query($query)) {
   echo "Error en la consulta: $resultado->sqlstate";
}
```

Búsquedas dentro de una tabla

La realización de búsquedas en una tabla pasa por hacer uso de la sentencia de SQL LIKE y operar con ella como si de cualquier otra consulta se tratara:

*SELECT * FROM usuario WHERE nombre LIKE 'Jose%'*

Esta sentencia buscará en nuestra tabla usuarios a aquellos cuyo nombre empiece por Jose.

Si lo que debemos hacer es un buscador que admita el campo a buscar a través de un formulario deberemos almacenar previamente el contenido del campo en una variable y anexarla a continuación en nuestra sentencia si bien habría que tener cuidado de tratar adecuadamente los datos de entrada para evitar errores (veremos este aspecto en el próximo apartado):

```
If (isset ($_GET["busca"])){
      $cadena= $mysqli->real_escape_string($_GET["busca"]);
      $sql= "SELECT * FROM usuario WHERE nombre LIKE '%$cadena%'";
$mysqli->query($sql);
}
```

8.4 Manipulación e inserción de datos

El proceso para la manipulación de datos de una base de datos será muy similar al de la consulta de los mismos, es decir, haremos uso, principalmente, de sentencias SQL capaces de añadir, borrar o actualizar datos además del comentado proceso de conexión de desconexión. No obstante, dada la delicadeza de estas instrucciones, deberemos contemplar ciertas precauciones para evitar la pérdida o manipulación indebida de información cuando esta se deba producir desde el cliente. Las instrucciones de SQL asociadas a estas operaciones son INSERT, UPDATE y DELETE.

Ejemplo de inserción de un registro en una tabla:

```
$mysqli = new mysqli ("localhost", "user", "pass", "discografia");

$grupo= "Rolling Stones";
if ($mysqli->query("INSERT into Grupos(Nombre) VALUES ('$grupo')")) {
    echo "$mysqli->affected_rows filas insertadas";
}

$mysqli->close();
```

Ejemplo de actualización de un registro:

```
$sql = "UPDATE Agenda SET apellido='Doe' WHERE id=2";

if ($resultado=$mysqli ->query($sql)) {
    echo "Registro actualizado correctamente";
} else {
    echo "Se ha producido un error durante la actualización: ".
$resultado->sqlstate;
}
```

Ejemplo de borrado de un registro:

```
$sql = "DELETE FROM Grupos WHERE id=3";
 if ($mysqli->query($sql)) {
    echo "Error al borrar: $mysqli->sqlstate";
}
```

Uno de los procesos más arriesgados es el de la inserción de datos, operación que en el mundo web se suele llevar a cabo a partir de un formulario, lo que podría permitir la entrada no deseada de código, es decir, las famosas inyecciones SQL. Una buena forma de protegernos ante este tipo de intrusión es tratar la entrada antes de ejecutar la consulta, para lo cual PHP ofrece algunas herramientas interesantes. Otra forma de asegurar adecuada la entrada de datos consistirá en preparar las consultas, lo cual también contribuye a una ejecución más eficiente de las consultas, especialmente si se tienen que realizar de forma repetida.

Filtrando datos de entrada

PHP proporciona distintas herramientas, normalmente en forma de funciones o métodos, cuando se trata de sanear la entrada de datos por parte de usuarios. Resulta altamente recomendable usar algunas de las siguientes sentencias al trabajar con formularios y en especial cuando dicha entrada va a afectar a una base de datos.

stripslashes — Quita las barras de un string con comillas escapadas. Un ejemplo de uso de stripslashes() es cuando la directiva de PHP magic_quotes_gpc es on (estaba activado por defecto antes de PHP 5.4) y no se están insertando estos datos en un lugar (como una base de datos) que requiera escapado. Por ejemplo, si simplemente se le da salida a los datos directamente desde un formulario HTML. Devuelve un string con las barras invertidas retiradas. (\' se convierte en ' y así sucesivamente.) Barras invertidas dobles (\\) se convierten en una sencilla (\).

```
$str = "Is your name O\'reilly?";

// Salida: Is your name O'reilly?
echo stripslashes($str);
```

htmlspecialchars — Convierte los caracteres especiales más comunes en entidades HTML, lo que ayuda a limpiar código HTML fraudulento en una entrada de formulario. Si se desease después reconvertir el código de entrada en HTML, este se podría decodificar de vuelta se puede usar la función **htmlspecialchars_decode().**

```
$nuevo = htmlspecialchars("<a href='test'>Test</a>", ENT_QUOTES);
echo $nuevo; // &lt;a href=&#039;test&#039;&gt;Test&lt;/a&gt;
```

htmlentities — Convierte todos los caracteres aplicables a entidades HTML. Esta función es idéntica a htmlspecialchars() en todos los aspectos, excepto que con htmlentities(), todos los caracteres que tienen su equivalente HTML son convertidos a estas entidades.

```
$str = "Esta 'frase' esta en <b>negrita</b>";


echo htmlentities($str);
// Devuelve: Esta 'frase' esta en &lt;b&gt;negrita&lt;/b&gt;
```

Mysqli ofrece también un método realmente útil para escapar caracteres especiales que además ha sido especialmente diseñado para cuando estos deben enfrentarse a un motor de base de datos, que es justo lo que nos interesa en este apartado:

mysqli::real_escape_string — Escapa los caracteres especiales de una cadena para usarla en una sentencia SQL, tomando en cuenta el conjunto de caracteres actual de la conexión. Esta función se usa para crear una cadena SQL legal que se puede usar en una sentencia SQL. La cadena dada es codificada a una cadena SQL escapada, tomando en cuenta el conjunto de caracteres actual de la conexión.

```
/* esta consulta fallará debido a que no escapa $ciudad */

if (!$mysqli->query
("INSERT into miCiudad (Name) VALUES ('$ciudad')")) {
    printf("Error: %s\n", $mysqli->sqlstate);
}

$ciudad = $mysqli->real_escape_string($ciudad);

/* esta consulta con $ciudad escapada funcionará */
if ($mysqli-
>query("INSERT into miCiudad (Name) VALUES ('$ciudad')")) {
    printf("%d fila insertada.\n", $mysqli->affected_rows);
}
```

Además de tratar los datos de entrada con estos procedimientos, una de las formas más seguras de enviar sentencias SQL al servidor es mediante el uso de sentencias preparadas, funcionalidad de especial interés que veremos más adelante.

Ejemplo de inyección de borrado: en este ejemplo nos llega una posible inyección maliciosa a través del método GET, el cual espera recibir un nombre de usuario:

```
$_GET['username'] = "'; DELETE FROM users; /*"

// escape manual con mysqli_real_escape_string ()
$username = mysqli_real_escape_string($_GET['username']);

$mysqli->query ("SELECT * FROM users WHERE username = '$username'");

// La consulta da error, pero se evita el borrado de usuarios que se
intentaba inyectar
```

8.5 Creación, manipulación y borrado de tablas y bases de datos

Aunque, como decíamos al principio, este tipo de operaciones no se realizan habitualmente desde un entorno web no especializado, conviene al menos revisar un par de ejemplos sobre cómo se realizar estas operaciones desde PHP junto con mysqli. Para la correcta realización de este tipo de operaciones es altamente recomendable repasar con detenimiento nuestros conocimientos sobre SQL.

Ejemplo - Crear una tabla:

En el siguiente ejemplo crearemos una table llamada Agenda con los siguientes campos: id, nombre, apellidos, email y fecha_creacion. La sentencia SQL sería como sigue:

```
CREATE TABLE Agenda (
 id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
 nombre VARCHAR(30) NOT NULL,
 apellidos VARCHAR(60) NOT NULL,
 email VARCHAR(50),
 reg_date TIMESTAMP
)
```

Código php completo:

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

// Crear conexión
$conn = new mysqli($servername, $username, $password, $dbname);
// Comprobar conexión
if ($conn->connect_error) {
    die("Fallo de conexion: " . $conn->connect_error);
}

// sql para la creación de la tabla
$sql = "CREATE TABLE Agenda (
   id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
   nombre VARCHAR(30) NOT NULL,
   apellidos VARCHAR(60) NOT NULL,
   email VARCHAR(50),
   reg_date TIMESTAMP
)";

if ($conn->query($sql) === TRUE) {
    echo "Tabla Agenda creada con exito!";
} else {
    echo "Error creando la tabla: " . $conn->error;
}

$conn->close();
?>
```

Como se puede observar la creación de una tabla se llevará a cabo de forma idéntica a como hemos hecho con la realización de consultas, dado que la forma más simple es delegar toda la carga no en instrucciones de PHP dedicadas si no en el propio motor de la base de datos. Por supuesto el proceso de alteración y borrado se realizará de idéntica forma:

Ejemplo – Borrar una tabla

```
$sql="DROP TABLE Agenda";
$conn->query($sql);
$conn->close();
```

Crear una base de datos

En cuando a la creación de una base datos, hay que tener en cuenta que el proceso de conexión deberá ser ligeramente distinto, es decir, dado que la BBDD no existirá antes de su creación deberemos obviar este campo durante el proceso de conexión:

```
$servername = "localhost";
$username = "username";
$password = "password";

// Crear conexión
$conn = new mysqli($servername, $username, $password);
// Comprobar conexión
if ($conn->connect_error) {
    die("Fallo de conexion: " . $conn->connect_error);
}

$sql= "CREATE DATABASE nombreDB";
$conn->query($sql);
```

Si después de crear una BBDD deseamos poder operar con ella dentro de la misma página, deberemos seleccionarla una vez creada. Podremos hacer esto mediante el método **select_db**:

```
$conn->select_db("nombreDB");
```

En cuanto al borrado de una BBDD, deberemos tener en cuenta que una vez realizado procederá la desconexión o, al menos, la selección de otra BBDD dado que esta ya no existirá.

8.6 Sentencias preparadas

Las bases de datos MySQL soportan sentencias preparadas. Una sentencia preparada o una sentencia parametrizada se usa para ejecutar la misma sentencia repetidamente con gran eficiencia pero también para evaluar su sintaxis antes de llevarla a cabo, por lo que podemos ahorrar tiempo de proceso y comunicación con la BBDD. Hacer uso de consultas preparadas está considerado como una buena práctica, especialmente en lo concerniente a seguridad.

Flujo de trabajo básico

La ejecución de sentencias preparadas consiste en dos etapas: la preparación y la ejecución. En la etapa de preparación se envía una plantilla de sentencia al servidor de bases de datos. El servidor realiza una comprobación de sintaxis e inicializa los recursos internos del servidor para su uso posterior.

Preparación:

```
 $sentencia = $mysqli->prepare($SQL);
```

Ejecución:

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

Existe además un posible tercer paso conocido como vinculación que se ubica entre los dos pasos previamente mencionados, pero que no siempre es necesario aunque sí recomendable. El servidor de MySQL soporta el uso de parámetros de sustitución posicionales anónimos con el símbolo ?. Esto permite tratar previamente los valores de entrada, por ejemplo en variables independientes, para luego vincularlas a la sentencia SQL, la cual incluirá interrogantes en lugar de valores en la fase de preparación. Para ello usamos el método bind_param:

bool mysqli_stmt::bind_param (string $types , mixed &$var1 [, mixed &$...])

Enlaza variables para los marcadores de parámetros en la instrucción SQL que se envía a $mysqli->prepare(). Principalmente consta de dos parámetros, tipo, que define el tipo de las variables que se le pasan y un número de variables var1, var2... cuyo número debe coincidir con la longitud de la cadena tipo .

El tipo se refleja de la siguiente manera:

- i la variable correspondiente es de tipo entero
- d la variable correspondiente es de tipo double
- s la variable correspondiente es de tipo string
- b la variable correspondiente es un blob y se envía en paquetes

Ejemplo 1: preparación de una consulta simple que lista los datos de un usuario:

```
$query = $mysqli->prepare('SELECT * FROM users WHERE username = ?');
$query->bind_param('s', $_GET['username']);
$query->execute();
```

Ejemplo 2: preparación de una consulta de inserción de un valor

```
$mysqli->prepare("INSERT INTO test(id) VALUES (?)");
$query->bind_param("i", $variable_entera);
$query->execute();
```

Ejemplo 3: preparación de una consulta de inserción que deberá insertar varios valores

```
$mysqli->prepare("INSERT INTO test(id) VALUES (?, ?, ?, ?)");
$query->bind_param("iiss", $variable_entera1, $variable_entera2,
$variable_string1, $variable_string2)
$query->execute();
```

Ejemplo completo de sentencia preparada #1 - Primera etapa: preparación

```
$mysqli = new mysqli("ejemplo.com", "usuario", "contraseña", "basedato
s");

if ($mysqli->connect_errno) {
    echo "Falló la conexión a MySQL: (" . $mysqli->connect_errno . ")
" . $mysqli->connect_error;
}

/* Sentencia preparada, etapa 1: preparación */
if (!($sentencia = $mysqli->prepare("INSERT INTO Alumno VALUES (?, ?,
?, ?)"))) {
    echo "Falló la preparación: (" . $mysqli->errno . ") " . $mysqli-
>error;
}
```

La preparación es seguida de la ejecución. Durante la ejecución el cliente vincula los valores de los parámetros y los envía al servidor. El servidor crea una sentencia desde la plantilla de la sentencia y los valores vinculados para ejecutarla usando los recursos internos previamente creados.

Ejemplo completo de sentencia preparada #2 - Segunda etapa: vincular y ejecutar

```
/* Sentencia preparada, etapa 2: vincular y ejecutar */

$id = 1;
if (!$sentencia->bind_param("issi", $id, $nombre, $apellidos,
$telefono)) {
    echo "Falló la vinculación de parámetros: (" . $sentencia-
>errno . ") " . $sentencia->error;
```

```
}

if (!$sentencia->execute()) {
    echo "Falló la ejecución: (" . $sentencia->errno . ") " .
$sentencia->error;}
```

Ejecución repetida

Una sentencia preparada se puede ejecutar repetidamente. En cada ejecución el valor actual de la variable vinculada se evalúa y se envía al servidor. La sentencia no se analiza de nuevo y la plantilla de la sentencia no es transferida otra vez al servidor. Este tipo de operación es ideal cuando necesitamos implementar un mecanismo según el cual se introducen varios elementos idénticos en una base de datos desde código o a través de un formulario desde nuestra web, por ejemplo dar de alta un grupo de alumnos o varios productos con similares características.

En el siguiente ejemplo insertamos 5 registros en nuestra tabla test con solo el campo ID:

```
<?php
$mysqli = new mysqli("ejemplo.com", "usuario", "contraseña", "basedato
s");
if ($mysqli->connect_errno) {
    echo "Falló la conexión a MySQL: (" . $mysqli-
>connect_errno . ") " . $mysqli->connect_error;
}


/* Sentencia preparada, etapa 1: preparación */
if (!($sentencia = $mysqli-
>prepare("INSERT INTO test(id) VALUES (?)"))) {
     echo "Falló la preparación: (" . $mysqli->errno . ") " . $mysqli-
>error;
}

/* Sentencia preparada, etapa 2: vinculación y ejecución */
$id = 1;
if (!$sentencia->bind_param("i", $id)) {
    echo "Falló la vinculación de parámetros: (" . $sentencia-
>errno . ") " . $sentencia->error;
}

if (!$sentencia->execute()) {
    echo "Falló la ejecución: (" . $sentencia-
>errno . ") " . $sentencia->error;
}

/* Sentencia preparada: ejecución repetida, sólo datos transferidos de
sde el cliente al servidor */
for ($id = 2; $id < 5; $id++) {
    if (!$sentencia->execute()) {
        echo "Falló la ejecución: (" . $sentencia-
>errno . ") " . $sentencia->error;
    }
}

/* se recomienda el cierre explícito */
$sentencia->close();
```

```
?>
```

Cada sentencia preparada ocupa recursos del servidor. Las sentencias deberían cerrarse explícitamente inmediatamente después de su uso. Si no se realiza el cierre explícito la sentencia será cerrada cuando el gestor de la sentencia sea liberado por PHP.

Usar una sentencia preparada no es siempre la manera más eficiente de ejecutar una sentencia. Una sentencia preparada ejecutada una sola vez causa más viajes de ida y vuelta desde el cliente al servidor que una sentencia no preparada.

8.7 Transacciones

Una **transacción** es un procedimiento según el cual se engloban una serie de operaciones dentro de un único proceso con la intención de atomizar su ejecución en un solo paso que no pueda ser interrumpido hasta que se hayan finalizado todas las operaciones. Este proceso suele ser de vital importancia en determinadas operaciones en las BBDD de carácter crítico como por ejemplo una transferencia de fondos monetarios entre dos cuentas o un proceso de pago o compra online.

El servidor MySQL soporta transacciones dependiendo de del motor de almacenamiento usado. Desde MySQL 5.5, el motor de almacenamiento predeterminado es InnoDB. InnoDB tiene soporte completo para transacciones ACID mientras que las tablas de tipo MyISAM no soportan la reversión (ROLLBACK) de las transacciones fallidas.

Las transacciones se pueden controlar usando SQL o llamadas a la API (en nuestro caso mysqli). Se suele recomendar usar llamadas a la API para habilitar y deshabilitar el modo de consignación automático (auto commit) y para consignar y restaurar transacciones ya que así se libera de carga al motor de MySQL y se tiene un control más temprano al ser previo a ninguna operación en la BBDD.

Las transacciones proporcionan, además, un mecanismo según el cual podemos deshacer las operaciones realizadas si esta no se consigue completar en su totalidad (**COMMIT**). A esto lo llamamos **ROLLBACK** y PHP también los soporta mediante mysqli.

Fases de una transacción

1. Iniciar transacción - Autocommit

Por defecto, MySQL se ejecuta en modo autocommit. Tener activo el modo de ejecución automática (autocommit) significa que cada consulta se tratará como una transacción individual. Es importante desactivar autocommit cuando se va a llevar a cabo una transacción porque de lo contrario la llamada al método query del objeto mysqli se ejecutaría de inmediato (a menos que trabajásemos en un escenario de sentencias preparadas):

```
$mysqli->autocommit(false);
```

2. Realización de las consultas

Una vez se ha desactivado la opción de autocommit, ya podemos proceder a la realización de las consultas de forma normal, aunque estas no se realizarán si no que quedarán a la espera de una última instrucción (similar al execute de las sentencias preparadas) para llevar a cabo todas de golpe o deshacer el proceso.

3. Finalizar o revertir transacción

Para este último paso contamos con los métodos commit y rollback para terminar y ejecutar la transacción en el primer caso o revertir el proceso en el segundo:

```
if ($flag) {//si no ha habido error en las consultas
     $mysqli->commit();
     echo "Consultas ejecutadas correctamente";
 } else {
     $mysqli->rollback();
     echo "Todas las consultas han sido revertidas";
 }
```

Es muy importante saber que tanto si se confirma una transacción como si se revierte, la conexión de base de datos se devuelve automáticamente a modalidad autocommit hasta que la siguiente llamada que inicia una nueva transacción.

También hay que tener cuidado con rollback() ya que no todas las sentencias sql pueden ser revertidas dentro de una transacción. Ya que hay un número determinado de sentencias que llevan un commit implícito, como las de creación o borrado de una tabla o una BBDD entre otras.

Ejemplo completo de transacción:

```
<?php
$mysqli = new mysqli("ejemplo.com", "usuario", "contraseña", "basedato
s");


if ($mysqli->connect_errno) {
    echo "Falló la conexión a MySQL: (" . $mysqli-
>connect_errno . ") " . $mysqli->connect_error;
}

// Fase 1 - desactivar autocommit
$mysqli->autocommit(false);



// Fase 2 - realización de consultas

$mysqli->query("UPDATE cuentas SET balance = balance - 500 WHERE
cod_cliente=cc1");

$mysqli->query("UPDATE cuentas SET balance = balance + 500 WHERE
cod_cliente=cc2");


// Fase 3 - Finalización o reversión de la transacción

if (!$mysqli->commit()) {
    print("Falló la consignación de la transacción\n");

    $mysqli->rollback();

}
```

```
// Fase 4 – cierre de conexión
$mysqli->close();


}
?>
```

Uso de Rollback con savepoints

La posibilidad de revertir una transacción mediante la instrucción rollback puede ser más flexible si se usa en conjunción con el método savepoint que también incluye la API mysqli. Este método permite establecer un punto de guardado entre instrucciones al que podemos regresar en caso error mediante un rollback dirigido a ese punto.

El método se puede invocar en cualquier punto de nuestro código de la siguiente manera:

```
$mysqli->savepoint('guardado-1');
```

Esto genera un punto de guardado al que podemos regresar mediante el rollback en caso necesario:

```
$mysqli->rollback('guardado-1');
```

Ejemplo:

```
$mysqli->autocommit(false);


$flag=true; // variable booleana de control

//Vamos a realizar cuatro inserciones de entre las cuales fallará la
última; al hacerlo cambiará el valor de nuestra variable de control
$flag, lo que disparará un rollback específico


$mysqli->query("INSERT INTO myCity (id) VALUES (100)") ? null :
$flag=false;

$mysqli->query("INSERT INTO myCity (id) VALUES (200)") ? null :
$flag=false;

$mysqli->savepoint('control');

$mysqli->query("INSERT INTO myCity (id) VALUES (300)") ? null :
$flag=false;

$mysqli->query("INSERT INTO myCity (id) VALUES (100)") ? null :
$all_query_ok=false; //clave primaria duplicada

//evaluamos nuestra variable de control
$flag? $mysqli->commit() : $mysqli->rollback('control');

$mysqli->close();
```

De esta forma si falla alguna de las dos últimas instrucciones, el rollback (‘control’) nos devolvería a al punto de control, llevándose a cabo las primeras dos instrucciones pero no las dos últimas.

8.8 PDO

La extensión *Objetos de Datos de PHP* (PDO por sus siglás en inglés) define una interfaz ligera para poder acceder a bases de datos en PHP. Cada controlador de bases de datos que implemente la interfaz PDO puede exponer características específicas de la base de datos, como las funciones habituales de la extensión.

PDO proporciona una capa de abstracción de *acceso a datos*, lo que significa que, independientemente de la base de datos que se esté utilizando, se emplean las mismas funciones para realizar consultas y obtener datos.

PDO viene con PHP 5.1 y proporciona un funcionamiento basado en Orientación a Objetos, al igual que MySQLi y, a diferencia de este que se centra exclusivamente en MySQLi, PDO ofrece soporte para numerosos motores de bases de datos entre los que destaca Oracle, MS SQL Server, ODBC, DB2, PostgreSQL, SQLite, 4D, Informix, Firebird y desde luego MySQL. Aunque en última instancia MySQLi es ligeramente más rápido que PDO (aproximadamente un 6,5%), su enorme soporte y su alto grado de uso en la actualizad lo convierten en una extensión de alto interés que se recomienda conocer y explorar.

Ejemplo de conexión (orientado a objetos):

```
abstract class Conection {
      // Conexion a la base de datos
    public static function con() {
        $con = new PDO("mysql:host=localhost;dbname=discografia",
"MiUsuario", "MiPassword");
        $con->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        return $con;
    }
}
```

Ejemplo de realización de consulta contra la tabla Canciones:

```
$sql = "SELECT * FROM canciones ORDER BY idcancion DESC";
/* Construimos la consulta en el string $sql, la preparamos y la
enviamos al servidor dentro de un try-catch para garantizar su
protección ante errores  */
try {
      $db = Conection::con();
      $stmt = $db->prepare($sql);
      $stmt->execute();
      $canciones = $stmt->fetchAll();
      $stmt = null;
      $db = null;
      $data = array("status" => "success", "data"=> $canciones, "code"
=> 200);
}
```

```
catch (PDOException $e){
      $data = array("status"=> "error", "msg" => "Se ha producido un
error con los datos. Por favor recargue la página", "code" => 404);
}

//$data contiene el resultado de la consulta el cual devolvemos al
cliente mediante JSON

echo json_encode($data);
}
```

Otra alternativa más sencilla y en la línea de los ejemplos anteriores:

```
<div id="container">
<?php

  $servername = "localhost";
  $username = "dam";
  $password = "dam";
  $dbname = "discografia";

  $sql = "SELECT c.idcancion, c.titulo AS cancion, d.titulo AS disco,
  g.nombre FROM ((canciones c INNER JOIN cancionesdisco cd ON
  c.idcancion=cd.idcancion) INNER JOIN discos d ON
  cd.iddisco=d.iddisco) INNER JOIN grupos g ON g.idgrupo=d.idgrupo
  ORDER BY c.idcancion ASC";

  /* Construimos la consulta en el string $sql, la preparamos y la
  enviamos al servidor dentro de un try-catch para garantizar su
  protección ante errores */

  try {
      $connexion = new PDO ("mysql:host=$servername;dbname=$dbname",
  $username, $password);
      $connexion->setAttribute(PDO::ATTR_ERRMODE,
  PDO::ERRMODE_EXCEPTION);

      $stmt = $connexion->prepare($sql);
      $stmt->execute();
      $canciones = $stmt->fetchAll();
      $connexion = null;
      $db = null;
  }
  catch (PDOException $e){
      echo "Connection failed: " . $e->getMessage();
  }
?>
```

```
<div class="tg-wrap">
  <h3>Títulos de canciones, nombre del grupo que la interpreta y disco
  en el que aparece</h3>
  <table class="tg">
  <th>ID</th><th>Canción</th><th>Grupo</th><th>Disco</th>

  <?php
      foreach($canciones as $row) {
        echo "<tr><td>".$row['idcancion']."</td>
        <td>".$row['cancion']."</td><td>".$row['nombre']."</td><td>
        ".$row['disco']."</td></tr>";
      }
  ?>

  </table>
</div>
</div>
```

Explorar con mayor profuncidad PDO queda fuera de este manual si bien mediante el anterior ejemplo queda demostrada la gran semejanza entre el uso esta extensión y MySQLi. Para un mayor estudio del PDO dejamos aquí algunos completos manuales:

- Tutorial de uso de PDO en castellano: https://diego.com.es/tutorial-de-pdo
- Tutorial completo en inglés: https://phpdelusions.net/pdo
- Manual oficial de PHP PDO: http://php.net/manual/es/book.pdo.php

9. Uso de plantillas en PHP

Cuando se empieza a desarrollar con PHP existe la tendencia de mezclar nuestro código PHP con HTML/CSS en los mismos ficheros. Esto, aun siendo cómodo en algunas ocasiones, no deja de ser una mala práctica ya que mezcla la lógica de nuestro proyecto con la presentación de la misma manera que si en HTML insertamos formato que de otra manera se definiría vía CSS en la cabecera o en un fichero aparte.

El lenguaje PHP es completamente abierto, en el sentido de que no requiere desarrollar en alguna arquitectura concreta, sino que un fichero puede contener un millón de líneas con código PHP y código HTML intercalados, lo cual tiene severas desventajas y problemas:

- Un diseñador tendrá muy difícil manejar el código HTML/CSS de esa página.
- La escalabilidad es prácticamente nula.
- Mantener una aplicación así se complica progresivamente.
- La depuración de ese tipo de aplicaciones se hace muy complicada.

La de idea de usar plantillas parte precisamente de esta premisa, proponiendo un mayor grado de independencia entre nuestro código PHP y el resto de la web y resolviendo así todos esos problemas, y ofreciéndonos además multitud de ventajas adicionales.

A lo largo de este curso ya hemos visto más de una forma de trabajar con plantillas por nuestra cuenta, ayudándonos de includes o basándonos en nuestra propia estructura de objetos. Aunque son métodos razonablemente fáciles y funcionales no son capaces de aportar la potencia y flexibilidad de un mecanismo consolidado de los que existen actualmente en el mercado. De entre todos aquí visitaremos con Smarty, uno de los más extendidos y veteranos.

9.1 Smarty

Aunque existen muchos mecanismos para la el manejo de plantillas en PHP, para este tema nos decantaremos por el uso de Smarty, un motor de plantillas ya veterano en el mercado. Estas son algunas de las principales ventajas que ofrece Smarty frente a sus competidores:

- Es extremadamente rápido.
- Plantillas limpias fáciles de usar por los diseñadores.
- Escalabilidad.
- Mantenimiento más sencillo (al igual que la escalabilidad, únicamente con la separación de código y presentación no se consigue un mantenimiento más sencillo, también se requerirá de una buena codificación).
- Depuración óptima del código, al tener ficheros pequeños únicamente con código PHP.
- No analiza gramaticalmente cada template, si no que trabaja con una versión compilada del mismo y que sólo recompilará si encuentra cambios.
- Posibilidad de introducir comentarios dentro de las plantillas que no se enviarán al servidor. Ejemplo: {* comentario smarty *} en lugar de <!-- comentario HTML -->
- Funciones integradas que facilitan el tratamiento de variables. Ejemplos: {foreach}{/foreach}, {if}{else}{/if}.

- Funciones asistentes para generación de código HTML. Ejemplo: {html_image file=”banner.jpg”} generaría
- Expandir Smarty con más funcionalidades mediante plugins.

9.2 Instalación de Smarty

El proceso de instalación de Smarty es razonablemente sencillo. Empecemos por descargar la última versión desde su website: http://www.smarty.net/download o directamente desde su repositorio de GitHub: https://github.com/smarty-php/smarty/releases

Lo único que tenemos que hacer a continuación es incluir el directorio lib en la carpeta anterior a la de nuestro proyecto si queremos hacer uso de Smarty en más de un proyecto o en algún sitio localizable dentro de la carpeta de nuestra web si sólo lo pensamos usar en este sitio. A continuación deberemos poder acceder al archivo **Smarty.class.php** desde las páginas de nuestra web que no requieran de forma similar a como sigue:

```
<?php
require('libs/Smarty.class.php'); //la carpeta libs se encontrará en
la misma carpeta que este script
$smarty = new Smarty;
?>
```

Otra alternativa podría ser la de definir una constante llamada SMARTY_DIR que apunte a la carpeta exacta donde se ubican los archivos de smarty de forma similar a la siguiente:

```
<?php
define('SMARTY_DIR', '/usr/local/lib/php/Smarty/');
require(SMARTY_DIR . 'Smarty.class.php');
$smarty = new Smarty;
?>
```

Una vez que la librería de archivos esta accesible desde nuestro sitio web deberemos configurar los directorios que necesita Smarty para su funcionamiento. Smarty require por defecto los siguientes cuatro directorios: 'templates/', 'templates_c/', 'configs/' y 'cache/'.

Cada uno de estos directorios deberá ser definido en las propiedades de la clase de Smarty. **$template_dir, $compile_dir, $config_dir**, y **$cache_dir** respectivamente:

```
$smarty->template_dir = 'smarty/templates/';
$smarty->compile_dir = 'smarty/templates_c/';
$smarty->config_dir = 'smarty/configs/';
$smarty->cache_dir = 'smarty/cache/';
```

Independientemente de si ubicamos la carpeta libs dentro de nuestro proyecto o fuera de él a nivel de nuestro servidor, se recomienda encarecidamente configurar estas carpetas de forma independiente para cada una de nuestras aplicaciones web y no dejar que compartan estos recursos, dado que podríamos provocar problemas de sobreescritura entre archivos de nuestras webs.

Adicionalmente Smarty necesitará permisos de escritura para las carpetas templates_c y cache para poder guardar dentro las plantillas compiladas.

9.3 Usando Smarty

En este capítulo repasaremos el uso básico de Smarty recorriendo parte de su sintaxis y algunas de sus funcionalidades más importantes. Para ello nos valdremos de numerosos ejemplos puntuales y uno más completo que expondremos al final del tema.

Smarty hace un extenso uso de las llaves { } en su sintaxis. Normalmente haremos uso de las llaves para invocar código de Smarty como instrucciones o definiciones de variables. Por otra parte usar las llaves acompañadas de asteriscos nos permitirá crear comentarios.

Veamos un ejemplo de uso básico. Vamos a crear un documento llamado index.tpl que ubicaremos dentro de la carpeta templates y que contendrá el siguiente código:

```
{* Smarty *}
<p>Hola, <b>{$name}</b>! Este es tu primer documento con Smarty</p>
```

Lo que hemos hecho es crear una variable llamada $nombre y la hemos invocado dentro del párrafo. Por otra parte en la parte superior hemos incluido un comentario. No es obligatorio pero sí es una buena práctica comenzar nuestras plantillas con un comentario introductorio sobre la misma. Nótese que la extensión del archivo será tpl (de template) y que en su interior podremos ubicar código HTML, mientras que no debemos incluir las etiquetas de apertura y cierre típicas de PHP.

Lo siguiente que vamos a hacer es crear nuestro documento index.php conteniendo el siguiente código:

```
<?php
require('libs/Smarty.class.php');
$smarty = new Smarty;

$smarty->template_dir = 'smarty/templates/';
$smarty->compile_dir = 'smarty/templates_c/';
$smarty->config_dir = 'smarty/configs/';
$smarty->cache_dir = 'smarty/cache/';

$smarty->assign('nombre','Pepito');

$smarty->display('index.tpl');
?>
```

El método **assign** nos permite enlazar la variable nombre que habíamos definido en el tpl con un valor, en este caso Pepito.

El método **display** se encargará de mostrar nuestra plantilla realizando todas las sustituciones correspondientes, en este caso solamente el nombre. El resultado será:

*Hola, **Pepito**! Este es tu primer documento con Smarty*

Usando varias plantillas

Extendamos el ejemplo anterior para ver cómo podríamos hacer uso de varios archivos tpl para construir una página a partir de ellos de forma similar a como hicimos con nuestra clase PaginaWeb en el tema de Orientación a Objetos:

index.php

```
<?php
require('libs/Smarty.class.php');
$smarty = new Smarty;

$smarty->template_dir = 'smarty/templates/';
$smarty->compile_dir = 'smarty/templates_c/';
$smarty->config_dir = 'smarty/configs/';
$smarty->cache_dir = 'smarty/cache/';

$respuesta= (3*4+2*4)*2+128/(32*2);
$smarty->assign('titulo','Mi web con Smarty');
$smarty->assign('pie','Texto para el pie de página');
$smarty->assign('answer',$respuesta);

$smarty->display('cabecera.tpl');
$smarty->display('cuerpo.tpl');
$smarty->display('pie.tpl');
?>
```

Cabecera.tpl

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title> {$titulo}</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<link rel="stylesheet" type="text/css" href="style/estilo.css">
</head>
<body>
<div id="container">
```

Cuerpo.tpl

```
<h1>Ejemplo de web con Smarty</h1>
<div id="contenidos">
<p>Esto es una página web creada a base de plantillas con PHP y
Smarty. Entre las ventajas de crear esta web con plantillas cabe
destacar las siguientes:</p>
<ol>
      <li>Separacion entre c&oacute;digo y dise&ntilde;o</li>
      <li> P&aacute;ginas f&aacute;cilmente escalables</li>
      <li> Rapidez de carga</li>
      <li> Mantenimiento sencillo</li>
</ol>
<p>La respuesta a todo en este universo es: <b>{$answer}</b></p>
</div>
```

Pie.tpl

```
<div id="pie">
      <p>{$pie}</p>
</div>
</div>
</body>
</html>
```

Nótese que en la zona head del archivo cabecera.tpl hemos incluido una llamada a una hoja de estilos CSS llamada estilo.css que reside en la carpeta style que hay en el directorio raíz del proyecto. Smarty no tiene problemas en relacionarse con CSS siempre y cuando este resida en un archivo externo, que por otro parte es lo habitual en proyectos de cierto tamaño. Sin embargo, en caso de que necesitáramos incluir código CSS dentro de un plantilla seguramente tendríamos problemas a causa del uso de las llaves, que en CSS también empleamos con profusión.

Para evitar este tipo de problemas Smarty provee de una solución tan sencilla como usar la etiqueta **{literal}** antes y después del código CSS como en el ejemplo siguiente:

```
<style>
{literal}
      p {
      font-size: 14px;
      }
{/literal}
</style>
```

9.4 Modificadores de variables

Los modificadores son pequeñas operaciones u alteraciones que podemos aplicar a una variable, una función o una cadena de texto. Para utilizar un modificador debemos incluirlo junto con, por ejemplo, una variable separado por el símbolo pipe | de la siguiente forma:

```
<title>{$titulo|upper}</title>
{* El título de la web se mostrará en mayúsculas *}
```

También existen modificadores capaces de contar caracteres, palabras, párrafos o incluso reemplazar expresiones regulares. A cierto nivel, este tipo de modificadores servirían para evitar ciertas funciones propias de PHP mientras que otras se centran en cambios estéticos (upper, lower, indent…). Hay quienes consideran que el uso de modificadores debería limitarse o incluso evitarse, ya que en parte implicarían una forma de intrusión en el estilo (pudiendo hacerse desde CSS o HTML) casi equivalente a la que se pretende evitar entre PHP, HTML y CSS.

Algunos ejemplos de modificadores:

Ejemplo de count_paragraphs

```
<?php
$smarty->assign('articleTitle',
                 "War Dims Hope for Peace. Child's Death Ruins
Couple's Holiday.\n\n
                 Man is Fatally Slain. Death Causes Loneliness,
Feeling of Isolation."
               );
?>
```

Donde el template es:

```
{$articleTitle}
{$articleTitle|count_paragraphs}
```

Esta es la salida:

```
War Dims Hope for Peace. Child's Death Ruins Couple's Holiday.

Man is Fatally Slain. Death Causes Loneliness, Feeling of Isolation.
2
```

Ejemplo de date_format

```
<?php
      $smarty->assign('yesterday', strtotime('-1 day'));
?>
```

El template podría ser así (usando $smarty.now):

```
{$smarty.now|date_format}
```

```
{$smarty.now|date_format:"%D"}
{$smarty.now|date_format:"%I:%M %p"}
{$yesterday|date_format}
{$yesterday|date_format:"%A, %B %e, %Y"}
{$yesterday|date_format:"%H:%M:%S"}
```

La salida sería:

```
Feb 6, 2001
02/06/01
02:33 pm
Feb 5, 2001
Monday, February 5, 2001
14:33:00
```

Otros modificadores populares de Smarty:

- capitalize – pone en l mayúsculas las primeras letras de una cadena
- cat – concatena cadenas de texto
- count_characters – cuenta los caracteres
- count_sentences – cuenta las frases de un texto (no los párrafos)
- count_words – cuenta las palabras de un texto
- default – permite definir un texto por defecto para una variable si esta no recibe otro valor
- escape – sirve para escapar símbolos de html, comillas o código javascript de un texto
- indent – indenta un texto en una cantidad de espacio
- lower – pasa un texto a minúsculas
- nl2br - convierte los saltos de línea a etiquetas
- regex_replace - localiza una expresión regular y la remplaza en la variable
- replace – sustituye una cadena de texto por otro
- spacify – inserta un espacio o un carácter entre cada letra de una cadena

Combinar modificadores

Smarty permite combinar más de un modificador para una misma variable simplemente poniendo uno a continuación de otro separados por el símbolo pipeline. Smarty aplicará dichos modificadores en el orden en el que fueron definidos.

```
<?php
$smarty->assign('articleTitle', 'Smokers are Productive, but Death
Cuts Efficiency.');
?>
```

Donde el template es:

```
{$articleTitle}
{$articleTitle|upper|spacify}
```

La salida se parecerá a:

```
Smokers are Productive, but Death Cuts Efficiency.
```

```
S M O K E R S   A R E   P R O D U C T I V E , B U T   D E A T H   C U
T S   E F F I C I E N C Y .
```

9.5 Funciones más comunes

Además de las ya conocidas **assign** y **display**, Smarty incluye un respetable número de funciones e instrucciones casi comparable al de un lenguaje completo de manera que es posible flexibilizar el uso que de hace de las plantillas hasta un punto en el que el uso de PHP puede incluso resultan innecesario para algunas partes. Repasemos algunas de las más interesantes:

{capture} es usado para recolectar toda la salida de un template en una variable en lugar de mostrarla. Cualquier contenido entre **{capture name="foo"}** y **{/capture}** es recolevtado en una variable especificada por el atributo name. El contenido capturado puede ser usado en el template a partir de la variable especial $smarty.capture.foo en donde foo es el valor pasado para el atributo name.

{config_load} Esta función es usada para cargar las #variables# de un archivo de configuración dentro de un template.

Ejemplo.conf (creado por nosotros)

```
#this is config file comment

# global variables
pageTitle = "Main Menu"
bodyBgColor = #000000
tableBgColor = #000000
rowBgColor = #00ff00
tableBorderSize= 1
```

y el template

```
 {config_load file="example.conf"}
{* cargamos el archive conf y sus variables quedan definidas y listas
para ser usadas *}

 <html>
      <title>{#pageTitle}</title>
      <body bgcolor="{#bodyBgColor#}">
     <table border="{#tableBorderSize#}" bgcolor="{#tableBgColor#}">
         <tr bgcolor="{#rowBgColor#}">
             <td>First</td>
             <td>Last</td>
             <td>Address</td>
         </tr>
     </table>
     </body>
 </html>
```

{section} Las secciones de un template son usadas para realizar iteraciones (loops) de un conjunto de datos de tipo array de forma similar a como lo haríamos en otros lenguajes con instrucciones de repetición como for, foreach o while. Todas las etiquetas section deben tener su par /section. Los parámetros requeridos son name y loop.

```
<?php

$data = array(1000,1001,1002);
$smarty->assign('custid',$data);

?>
```

Template

```
{* este ejemplo imprimirá todos los valores del array $custid *}
{section name=customer loop=$custid}
  id: {$custid[customer]}<br />
{/section}

{* aqui los imprimimos en orden inverso *}
{section name=foo loop=$custid step=-1}
  {$custid[foo]}<br />
{/section}
```

{foreach} Los bucles de tipo foreach son una alternativa a las secciones para la realización de bucles aunque con una sintaxis más simple que section, pero tiene una desventaja de que solo puede ser usada en una única matriz. La etiqueta foreach debe tener su par /foreach. Los parámetros requeridos son from e item. Los bucles foreach pueden ser anidados aunque para ello los nombres de los arrays implicados deben ser diferentes. La variable from (normalmente una matriz de valores) determina el número de veces del bucle.

```
<?php
$arr = array( 1001,1002,1003);
$smarty->assign('custid', $arr);
?>
```

```
{* este ejemplo muestra todos los valores de la matriz $custid *}
{foreach from=$custid item=curr_id}
  id: {$curr_id}<br />
{/foreach}
```

Esta es la salida del ejemplo de arriba:

```
id: 1000<br />
id: 1001<br />
id: 1002<br />
```

Foreach también es capaz de iterar entre elementos devueltos por la consulta a una base de datos.

{if}, {elseif}, {else} Los comandos {if} de Smarty tiene mucho de la flexibilidad del comando if de php, con algunas adaptaciones al tratamiento de las plantillas. Todo {if} debe tener su etiqueta de cierre {/if}. Las instrucciones {else} y {elseif} también son permitidas. Toda las condicionales de PHP son reconocidas por Smarty, tal como ||, or, &&, and, etc.

Ejemplos

```
{if $name == "Fred"}
      <p>Welcome Sir</p>
{elseif $name == "Wilma"}
      <p>Welcome Ma'am</p>
{else}
      <p>Welcome, whatever you are</p>
{/if}

{* Un ejemplo con "or" logico *}
{if $name eq "Fred" or $name eq "Wilma"}
      ...
{/if}

{* El mismo que arriba *}
{if $name == "Fred" || $name == "Wilma"}
      ...
{/if}

{* Expresiones como $name=="Fred" serían inváidas porque los operandos
requieren espacios alrededor*}

{* comprueba si el valor es par o impar *}
{if $var is even}
      ...
{/if}
{if $var is odd}
      ...
{/if}

{* comprueba si la variable var es divisible por 4 *}
{if $var is div by 4}
      ...
{/if}
```

```
{* También se pueden usar funciones *}
{if count($var) > 0}
      ...
{/if}
```

{fetch} Es usado para obtener archivos de sistema local, http o ftp, y mostrar el contenido. Si el nombre del archivo comienza con "http://", la página de la web cargada y mostrada. Si el nombre del archivo comienza con "ftp://", el archivo será obtenido del servidor ftp y mostrado. Para archivos locales, se debe proporcionar la ruta completa del sistema de archivos o una ruta relativa al script php a ejecutar.

Ejemplos:

```
{* cargando javascript en nuestra plantilla *}
{fetch file="scripts/navbar.js"}

{* incrustando información meteorological en formato de texto en la
plantilla desde otro sitio web *}
{fetch file="http://www.myweather.com/68502/"}

{* cargando un fichero de texto via ftp *}
{fetch
file="ftp://user:password@ftp.example.com/path/to/currentheadlines.txt
"}

{* asignando los contenidos obtenidos con fetch a una variable del
template *}
{fetch file="http://www.myweather.com/68502/" assign="weather"}
{if $weather ne ""}
  <b>{$weather}</b>
{/if}
```

{html_table} Es una función que transforma un array en una tabla HTML. El tamaño de la tabla se puede definir o dejar que Smarty lo decida de forma razonablemente eficiente a partir del número de elementos proporcionado. Si no, mediante el atributo *cols* se puede determinar el número de columnas que tendrá la tabla, dividiendo la información del array entre las celdas correspondientes y dejando vacías las sobrantes si las hubiera. Los valores *table_attr*, *tr_attr* y *td_attr* determinan los atributos dados para las etiquetas tabla, tr y td. Estos atributos pueden ser iterados a modo de bucle si se le proporcionan valores en formato array.

Ejemplo (php)

```
<?php
$smarty->assign('data',array(1,2,3,4,5,6,7,8,9));
$smarty->assign('tr',array('bgcolor="#eeeeee"','bgcolor="#dddddd"'));
$smarty->display('index.tpl');
```

```
?>
```

Template:

```
{html_table loop=$data}
{html_table loop=$data cols=4 table_attr='border="0"'}
{html_table loop=$data cols=4 tr_attr=$tr}
```

La salida de los tres casos:

```
<table border="1">
<tr><td>1</td><td>2</td><td>3</td></tr>
<tr><td>4</td><td>5</td><td>6</td></tr>
<tr><td>7</td><td>8</td><td>9</td></tr>
</table>

<table border="0">
<tr><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>9</td><td> </td><td> </td><td> </td></tr>
</table>

<table border="1">
<tr bgcolor="#eeeeee"><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr bgcolor="#dddddd"><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr
bgcolor="#eeeeee"><td>9</td><td> </td><td> </td><td> </
td></tr>
</table>
```

{html_image} Es una función de las más habituales. Se encarga de generar automáticamente una etiqueta HTML para una imagen.Entre sus parámetros cabe destacar:

- file: nombre del fichero
- height: altura proporcionada que sustituiría a la real de la imagen (que es la que se toma or defecto si este valor no se proporciona)
- width: anchura proporcionada que sustituiría a la real de la imagen (que es la que se toma or defecto si este valor no se proporciona)
- alt: texto alternativo que mostrará la imagen al poner el ratón encima o que reproducirá en caso de fallo de carga de la imagen o con navegadores con reproducción aural.
- href: enlace que incluirá la imagen. Si se proporciona este parámetro se generará una etiqueta <a href="LINKVALUE"><a> alrededor de la imagen.

Ejemplos

```
{html_image file="pumpkin.jpg"}
{html_image file="images/pumpkin.jpg" href="http://www.pumpking.com"}
```

Generarán las siguientes salidas:

```
<img src="pumpkin.jpg" alt="" width="44" height="68" />
<a href=http://www.pumpking.com> <img src="images/pumpkin.jpg" alt=""
width="44" height="68" /></a>
```

{html_image} requiere un acceso a disco para leer la imagen y calcular su altura y anchura. Si en nuestro proyecto web no hemos habilitado el uso de cache en la generación de las plantillas será mejor evitar su uso y optar por las etiquetas de imagen HTML para un funcionamiento más óptimo.

Elementos de formulario

Además de {htmlimage} existe toda una colección de instrucciones similares para generar etiquetas HTML de distinto tipo, especialmente para crear elementos de formulario:

- {html_checkboxes} – Crea casillas de verificación
- {html_options} – Crea elementos de selección desplegables, especialmente útiles para crear fechas calendarios, listados de poblaciones, etc
- {html_radios} – Para construir opciones exclusivas (radio buttons)
- {html_select_date} – Construye selectores de fecha
- {html_select_time} – Construye selectores de hora

10. PHP y JSON

10.1 Introducción a JSON

JSON, acrónimo de JavaScript Object Notation, es un formato ligero para el intercambio de datos. La simplicidad de JSON ha dado lugar a la generalización de su uso, especialmente como alternativa a XML en AJAX pero también de cara al intercambio de general de información y desde bases de datos. Una de las supuestas ventajas de JSON sobre XML como formato de intercambio de datos en este contexto es que es mucho más sencillo escribir un analizador sintáctico (parser) de JSON. En JavaScript, un texto JSON se puede analizar fácilmente usando la función eval(), lo cual ha sido fundamental para que JSON haya sido aceptado por parte de la comunidad de desarrolladores AJAX, debido a la ubicuidad de JavaScript en casi cualquier navegador web. Sin embargo dado el potencial peligro de dejar expuesto en nuestor código una función tan potente como Eval, las más recientes versiones de Javascript así como la mayor parte de los frameworks incorporan funciones específicas para su tratamiento, codificación y decodificación. Lo mismo sucede con otros lenguajes que pueden hacer uso de JSON, como es el caso de PHP, quien por su presencia en los servidores posee una posición privilegiada para su uso en el intercambio de información.

10.2 Sintaxis de JSON

Un objeto JSON está encerrado entre llaves {} y contiene propiedades separadas por comas. Cada propiedad tiene un valor separado por dos puntos y cada una de ellas se le asigna un valor que puede ser un string literal, un número, un array, una función u otro objeto JSON. Veamos un ejemplo:

```
{
        Nombre: "Juan",
        Apellidos: "Nadie",
        Edad: 25
}
```

Aunque el estándar estricto dicta que tanto propiedades como valores deber ir entrecomillados, lo habitual es que no se haga, lo que crea semejanzas con lenguajes como CSS (salvo por el uso de coma en lugar de punto y coma) o incluso con la notación de los objetos de JavaScript y otros lenguajes.

Ejemplo con vectores y otro objeto JSON:

```
{
        Nombre: "Juan",
        Apellidos: "Nadie",
        Edad: 25,
        Hijos: ["Mary", "Sean"],
        Pareja: {
                Nombre: "Eva",
                Apellidos: "Alguien"}
}
```

La principal ventaja de usar JSON es que podemos acceder con facilidad a los valores de las propiedades de forma similar a como accedemos a las propiedades de un objeto:

```
Var cliente = {
        Nombre: "Juan",
        Apellidos: "Nadie",
        Edad: 25,
        Hijos: ["Mary, "Sean"],
        Pareja: {
                Nombre: "Eva",
                Apellidos: "Alguien"
                }
        };

Alert ("el cliente "+ cliente.nombre + "tiene "+ cliente.hijos.lenght
+ " hijos y su pareja se llama "+ cliente.pareja.nombre);
```

10.3 Uso de JSON en Javascript

Uno de los principales usos de JSON es para transportar información de una forma estructurada en forma de objetos a través de AJAX como alternativa a XML. Sin embargo para que JavaScript interprete la información como un objeto en lugar de como un string, debe parsear (interpretar) esta información, para lo cual se vale de la función eval:

```
var persona = eval ("(" + cliente+ ")");
```

Alternativamente las versiones más recientes de JavaScript proporcionan una nueva función específica que resulta mucho más segura: JSON.parse (text):

```
var persona = JSON.parse(cliente);
```

La función JSON.parse (text), incluida en la mayoría de los navegadores modernos resulta mucho más segura porque es capaz de interpretar sólo código JSON, al contrario que la función eval, capaz de interpretar código JavaScript y ser propensa a la ejecución de código malicioso.

Adicionalmente tenemos a nuestra disposición funciones específicas que utilizan expresiones regulares para evitar código inapropiado. Librerías como jQuery o Mootols tienen sus propios métodos seguros.

Qué es y cómo funciona eval

Según la especificación ECMA262, eval es una propiedad del Objeto Global (Global Object) que evalúa el argumento facilitado como un programa Javascript. O lo que es lo mismo, esta función recoge una cadena que convierte en un código válido que después ejecuta.

Debido precisamente a su capacidad de ejecutar cadenas, eval puede comprometer el nivel de seguridad de nuestra aplicación: un error de arquitectura o validación puede facilitar la inyección de código malicioso que comprometa el sistema. Esta ha sido la principal razón por la que esta función cayó en desgracia.

Sin embargo, la aparición del formato de intercambios de datos JSON devolvió a eval todo el protagonismo perdido. Javascript necesita evaluar una cadena JSON para convertirla en un objeto puro. Esto ha llevado a la aparición de funciones específicas, primero ofrecidas por librerías de terceros y después presentes en la propia especificación Javascript, que realizan un eval seguro evitando los riesgos mencionados anteriormente.

Ejemplo de uso de eval

```
var string1 = "foo";
var string2 = "bar";
var funcName = string1 + string2;

function foobar(){
      alert( 'Hello World' );
}

eval(funcName + '()' );  // Hello World
```

10.3 Uso de JSON en PHP

PHP dispone de varias funciones para hacer distintos tratamientos con notación de objetos JSON, que permite convertir un objeto PHP, o cualquier otro tipo de variable, a un string con notación JSON, así como crear un objeto PHP a partir de un string codificado con JSON.

En PHP, como decíamos, es posible producir y consumir datos cargados con notación JSON, por medio de unas funciones de las que dispone el lenguaje, que existen de manera predeterminada en los servidores modernos de PHP y que se pueden utilizar también en instalaciones antiguas de PHP, aunque con algún trabajo de instalación adicional.

A partir de PHP 5.2 las funciones JSON están disponibles siempre, pero si utilizamos por ejemplo PHP 4 tendríamos que instalarlas manualmente.

Funciones de JSON:

json_decode — Decodifica un string codificado en JSON a una variable de PHP.

Ejemplo de json_decode()

```
<?php
$json = '{"a":1,"b":2,"c":3,"d":4,"e":5}';

var dump(json decode($json));
var_dump(json_decode($json, true));

?>
```

El resultado del ejemplo sería:

```
object(stdClass)#1 (5) {
    ["a"] => int(1)
    ["b"] => int(2)
    ["c"] => int(3)
    ["d"] => int(4)
    ["e"] => int(5)
}
```

Cuando el valor booleano (assoc) es **TRUE**, los objetos devueltos serán convertidos a array asociativos.

```
array(5) {
    ["a"] => int(1)
    ["b"] => int(2)
    ["c"] => int(3)
    ["d"] => int(4)
    ["e"] => int(5)
}
```

json_encode — Devuelve la representación JSON del valor dado, que originalmente puede ser un objeto o array asociativo. Los datos de tipo string deben estar codificados con UTF-8. Devuelve un string JSON codificado en caso de éxito o **FALSE** en caso de error.

Ejemplo #1: ejemplo sencillo de json_encode() con un array:

```
<?php
$arr = array('a' => 1, 'b' => 2, 'c' => 3, 'd' => 4, 'e' => 5);

echo json_encode($arr);
?>
```

El resultado del ejemplo sería:

```
{"a":1,"b":2,"c":3,"d":4,"e":5}
```

Ejemplo#2: uso e json_encode() sobre un objeto:

```
$myObj->name = "John";
$myObj->age = 30;
$myObj->city = "New York";
```

```
$myJSON = json_encode($myObj);

echo $myJSON;
```

El resultado sería:

```
{"name":"John","age":30,"city":"New York"}
```

Ejemplo#3: devolución de resultados al cliente de una consulta realizada a una base de datos con mysqli:

```
$mysqli = new mysqli($host, $user, $pass, $db);

// Realizar una consulta MySQL
$query = 'SELECT * FROM discos, grupos WHERE
discos.idgrupo=grupos.idgrupo ORDER BY año ';

$resultado = $mysqli->query($query) or die('Consulta fallida: ' .
mysql_error());

$salida= array();
$salida= $resultado->fetch_all(MYSQLI_ASSOC); //convierte el resultado
de la consulta en un arrays asociativo

echo json_encode($salida); //convierte el array en json y lo devuelve
al cliente para ser tratado con JavaScript
```

Ejemplo#4 – Uso de json_decode() y json_encode() para la realización deo cnsultas en una BBDD donde se envía desde el cliente los nombres de las tablas a consultar:

```
/* Se asume que al siguiente archivo le llega un objeto json como el
siguiente */

obj = { "table":"grupos", "limit":10 };
```

El envío, realizado a través de Ajax desde JavaScript, pasaría por enviarlo mediante el método POST convirtiendo a string el objeto JSON de la siguiente manera:

```
dbParam = JSON.stringify(obj);

/* Procedimiento para conectar con el servidor vía Ajax */

xmlhttp.open("POST", "json_demo_db_post.php", true);
xmlhttp.setRequestHeader("Content-type", "application/x-www-form-
urlencoded");  //construcción de la cabecera a enviar mediante POST

xmlhttp.send("x=" + dbParam); //Envío de
```

El archivo PHP que deberá recibir la información tendría el siguiente código:

```
<?php
header("Content-Type: application/json; charset=UTF-8");

$obj = json_decode($_POST["x"], false);

$resultado = $ mysqli ->query("SELECT nombre FROM ".$obj->table."
LIMIT ".$obj->limit);

$salida= array();
$salida= $resultado->fetch_all(MYSQLI_ASSOC);

echo json_encode($salida);
?>
```

Otras funciones de interés:

json_last_error_msg — Devuelve el string con el error de la última llamada a json_encode() o a json_decode()

json_last_error — Devuelve el último error que ocurrió