Hernán Beati

# PHP

## Creación de páginas Web dinámicas

**2ª edición**

Apoyo en la


Alfaomega

ITMaster
PROFESSIONAL TRAINING

# PHP

## Creación de páginas Web dinámicas

2ª ed.

Hernán Beati

# PHP

## Creación de páginas Web dinámicas

**2ª ed.**

Hernán Beati

Alfaomega

Buenos Aires • Bogotá • México DF • Santiago de Chile

Beati, Hernán
PHP : creación de páginas web dinámicas . - 2a ed. - Ciudad Autónoma de Buenos Aires : Alfaomega Grupo Editor Argentino, 2015.
428 p. ; 23x17 cm.
ISBN 978-987-3832-04-8
1. Programación. I. Título
CDD 004.6



**Edición:** Damián Fernández
**Corrección:** Vanesa García
**Armado de interiores:** Vanesa García
**Diseño de tapa: Diego Linares**
**Revisión de armado:** Vanesa García

Internet: http://www.alfaomega.com.mx


Paraguay 1307, PB, oficina 11
ISBN 978-987-3832-04-8
Queda hecho el depósito que prevé la ley 11.723



**Empresas del grupo:**

**Argentina:** Alfaomega Grupo Editor Argentino, S. A.
Paraguay 1307 P.B. “11”, Buenos Aires, Argentina, C.P. 1057
Tel.: (54-11) 4811-7183/0887
E-mail: ventas@alfaomegaeditor.com.ar

**México:** Alfaomega Grupo Editor, S. A. de C.V.
Pitágoras 1139, Col. Del Valle, México, D.F., México, C.P. 03100
Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420/2490. Sin costo: 01-800-020-4396
E-mail: atencionalcliente@alfaomega.com.mx

**Colombia**: Alfaomega Colombiana S. A.
Calle 62 N° 20-46, Bogotá, Colombia
Tel. (57-1)7460102 - Fax: (57-1) 2100415
E-mail: cliente@alfaomega.com.co

**Chile**: Alfaomega Grupo Editor, S. A.
Av. Providencia 1443, Oficina 24, Santiago de Chile, Chile
Tel.: (56-2) 235-4248/2947-5786 – Fax: (56-2) 235-5786
E-mail: agechile@alfaomega.cl

# Mensaje del editor

Los conocimientos son esenciales en el desempeño profesional, sin ellos es imposible lograr las habilidades para competir laboralmente. La universidad o las instituciones de formación para el trabajo ofrecen la oportunidad de adquirir conocimientos que serán aprovechados más adelante en beneficio propio y de la sociedad; el avance de la ciencia y de la técnica hace necesario actualizar continuamente esos conocimientos. Cuando se toma la decisión de embarcarse en una vida profesional, se adquiere un compromiso de por vida: mantenerse al día en los conocimientos del área u oficio que se ha decidido desempeñar.

Alfaomega tiene por misión ofrecerles a estudiantes y profesionales conocimientos actualizados dentro de lineamientos pedagógicos que faciliten su utilización y permitan desarrollar las competencias requeridas por una profesión determinada. Alfaomega espera ser su compañera profesional en este viaje de por vida por el mundo del conocimiento.

Alfaomega hace uso de los medios impresos tradicionales en combinación con las tecnologías de la información y las comunicaciones (TIC) para facilitar el aprendizaje.

Libros como éste tienen su complemento en una página Web, en donde el alumno y su profesor encontrarán materiales adicionales.

Esta obra contiene numerosos gráficos, cuadros y otros recursos para despertar el interés del estudiante, y facilitarle la comprensión y apropiación del conocimiento. Cada capítulo se desarrolla con argumentos presentados en forma sencilla y estructurada claramente hacia los objetivos y metas propuestas.

Los libros de Alfaomega están diseñados para ser utilizados dentro de los procesos de enseñanza-aprendizaje, y pueden ser usados como textos para diversos cursos o como apoyo para reforzar el desarrollo profesional.

Alfaomega espera contribuir así a la formación y el desarrollo de profesionales exitosos para beneficio de la sociedad.

# Acerca del autor

No nací programador. Por esta razón, estoy convencido de que todas las personas pueden aprender a programar y me especializo en su capacitación.

Al finalizar mi adolescencia, comencé mi formación pedagógica cursando un magisterio de música (que nunca ejercí); pero paralelamente me dediqué profesionalmente al diseño gráfico durante casi una década, y ya hacia fines del siglo XX, me convertí en diseñador Web.

Conocer el lenguaje PHP en el año 2000 fue lo que me hizo estudiar Análisis de Sistemas y la Licenciatura en Informática Educativa y pronto, me especialicé en programación para la Web, realizando numerosas aplicaciones Web, trabajando tanto para empresas como freelance.

Desarrollé mi perfil docente desde el año 2002, cuando fundé el campus virtual de SaberWeb y comencé a enseñar en los institutos ITMaster, Image Campus y Dotzero, en las carreras terciarias de Escuela Da Vinci e IMAGE, y para graduados de las carreras de Diseño en la FADU de la Universidad de Buenos Aires. En la Universidad Tecnológica Nacional (FRBA), dicté el curso Professional Webmaster y fui el creador del curso de Programador Web Avanzado, que aún se sigue dictando.

Desde 2013, soy profesor titular de las materias Programación Multimedial II y III en la Universidad Maimónides, dentro de la Licenciatura en Tecnología Multimedial, donde investigamos sobre la base de PHP y MySQL la creación de servicios Web para Apps móviles y videojuegos, y creamos aplicaciones Web orientadas a objetos.

Sigo capacitándome continuamente: los últimos cursos que hice en esta área fueron sobre PHP orientado a objetos, y backend de aplicaciones móviles y videojuegos.

Siempre me gustó el modelo de desarrollo colaborativo, propio del software libre, del que PHP es un perfecto exponente.

Hernán Beati, julio de 2015

*A mis cuatro mujeres (mi esposa Katty y mis hijas Stephanie, Violeta y Mora),*
*que soportaron pacientemente las horas que les robé*
*para poder escribir este libro.*
*.*

**Hernán Beati**

# Agradecimientos

A Maximiliano Firtman, que siempre confió en mis posibilidades, haciéndome parte de sus equipos de docentes, y ahora brindándome la oportunidad de publicar este libro.

A Damián Fernández, de Alfaomega Grupo Editor, por su seguimiento y estímulo constante.

A todos mis alumnos, porque todo este esfuerzo solo cobra sentido en la medida en que puedan aprovecharlo para su crecimiento professional.

Hernán Beati

# Contenido


Prólogo ..... XIII
Plataforma de contenidos interactivos ..... XVI

CAPÍTULO 1
**MÁS ALLÁ DE HTML Y CSS** ..... 1
¡No más páginas Web: aplicaciones Web ..... 1

CAPÍTULO 2
**EL AMBIENTE PHP** ..... 15
Esas extrañas siglas: LAMP, MAMP, WAMP, xAMP ..... 15

CAPÍTULO 3
**MEZCLANDO PHP Y HTML** ..... 35
El concepto clave: completando las páginas HTML en el acto ..... 35

CAPÍTULO 4
**LOS ALMACENES DE DATOS** ..... 51
Contenedores temporales y permanentes, de pocos y de muchos datos ..... 51
Las variables: pocos datos, provisorios ..... 53
Las constantes: pocos datos que no cambiaremos ..... 65
Las matrices: muchos datos provisorios ..... 69

CAPÍTULO 5
**ENVIANDO DATOS HACIA EL SERVIDOR** ..... 81
Herramientas para enviar datos: enlaces y formularios ..... 81

CAPÍTULO 6
**VALIDACIONES** ..... 91
Validando datos de formularios y enlaces ..... 91
Las condicionales ..... 92
¿Qué hacer si responde que no es verdad? El else y el elseif ..... 94

CAPÍTULO 7
**IDENTIFICACIÓN CON COOKIES Y SESIONES** ..... 123
Cookies: datos que identifican a un navegador ..... 123
Sesiones: datos que identifican a un usuario ..... 141

CAPÍTULO 8
**LOS BUCLES Y LOS ARCHIVOS DE TEXTO** ..... 163
Recorriendo línea por línea la información información almacenada ..... 163
Tipos de bucles: for, while, do while, foreach ..... 164
Condicionales dentro de bucles ..... 179
Los archivos de texto ..... 182
Formas de leer datos desde un archivo de texto ..... 186
Cómo escribir y acumular datos en un archivo de texto ..... 198

CAPÍTULO 9
**CREANDO Y USANDO FUNCIONES** ..... 203
Reutilizando nuestros códigos ..... 203
Planificando nuestros sistemas Web ..... 205
La función: una caja cerrada que procesa datos ..... 207
Declarar una función ..... 210

CAPÍTULO 10
**FUNCIONES INCORPORADAS MÁS USADAS** ..... 229

Manejo de caracteres, fechas y envío de correos ....229
Funciones de manejo de caracteres ....230
Funciones de fecha y hora ....249
Funciones de envío de correos electrónicos....257

CAPÍTULO 11
**CREANDO BASES DE DATOS** ....269
El almacén de datos más potente para nuestros sitios Web....269
Diferencia entre archivos de texto y bases de datos: el lenguaje SQL....270
Conceptos fundamentales: base, tabla, registro y campo....272
Creando bases y tablas con phpMyAdmin.275
Proceso de altas, bajas y modificaciones..289
Los tipos de datos más usados....297
Atributos de los campos....309

CAPÍTULO 12
**LLEVANDO DATOS DE LA BASE A LAS PÁGINAS** ....315
Cómo leer datos desde una base con PHP.315
PHP....315
Complementos de la orden SELECT del lenguaje SQL....324
Funciones propias para mostrar datos ....331

CAPÍTULO 13
**LLEVANDO DATOS DE LAS PÁGINAS A LA BASE** ....345
Cómo escribir datos en una base desde PHP....345
Cómo eliminar datos de una base con PHP....357
Cómo modificar datos de una base con PHP....365
Radiografía de un sistema con back-end y front-end....379

APÉNDICE
**PROGRAMACIÓN ORIENTADA A OBJETOS**....387

APÉNDICE WEB
**ADAPTANDO SOFTWARE LIBRE**
http://libroweb.alfaomega.com.mx

# Prólogo

En 1994, un programador nacido en Groenlandia, llamado Rasmus Lerdorf (http://lerdorf.com), desarrolló un código que le ayudaría a crear su página Web personal de manera más sencilla. Lo llamó Personal Home Page Tools (PHP Tools) o herramientas para páginas iniciales personales. De las primeras tres palabras en inglés surge el nombre del lenguaje que finalmente se liberó al público, gratis, en 1995.

Quince años después, el mundo de la Web ha cambiado drásticamente. La evolución y difusión de PHP en el mundo del desarrollo Web ha ido mucho más allá de lo que Rasmus pudo imaginar; se trata de un mundo del que ahora podrás ser parte.

Dos años más tarde, junto a otras personas, se reescribe parte del código del lenguaje y se lanza la versión de PHP que ha llevado el lenguaje al estrellato: PHP 3. Un lenguaje simple, rápido y dinámico que permite crear páginas Web interactivas con muy poco código.

En ese momento deciden que el nombre Personal Home Page ya le quedaba un poco corto al lenguaje y deciden cambiar el significado de las siglas. Así es que hoy PHP significa "PHP Hypertext Preprocessor". No es un error de imprenta: la "pe" de PHP significa PHP. Es una sigla recursiva (un truco de programadores) y el resto del nombre significa "pre-procesador de hipertexto". Es un pre-procesador porque se ejecuta antes que el navegador y trabaja principalmente sobre hipertexto, que es el concepto subyacente de los documentos HTML.

Con los años igualmente se ha ganado su propio nombre. PHP es, simplemente, PHP. Es tan importante en la Web como lo es HTML. Es un lenguaje fácil de aprender, simple de usar, potente, rápido, gratuito, de código abierto y utilizado en más de la mitad de todos los sitios Web del mundo.

Solo para ejemplificar la potencia del lenguaje, mencionaremos que el sitio más importante y con más visitas hoy en el mundo, Facebook, está desarrollado con PHP.

Tuve la oportunidad de conocer personalmente a Rasmus en Madrid en una conferencia sobre optimización extrema de PHP y me ha quedado muy presente

una anécdota que me gustaría compartir contigo. Rasmus tuvo la oportunidad de analizar el código fuente PHP de las primeras versiones de Facebook. El código era un desastre, pésimamente programado, con errores por todos lados.

Y a pesar del desastre que él detectó que era esa Web desde el lado del código, ¡funcionaba bien! Y miren en lo que Facebook se ha convertido: una empresa billonaria. La moraleja de la anécdota es que la gran ventaja de PHP es su capacidad de funcionar sin problemas en cualquier circunstancia y de poder tener una Web lista en muy poco tiempo.

Rasmus, un poco exagerando para hacer entender el objetivo de su moraleja, comentó que los proyectos más exitosos en la Web no son los mejor programados, los que siguen las mejores prácticas o los que son desarrollados por académicos. Son aquellos que implementan las mejores ideas y lo hacen lo más rápido posible; por eso PHP es el lenguaje ideal para la Web.

Luego de esta introducción, estarás seguro con muchas ganas de empezar a trabajar con PHP. Y qué mejor que hacerlo de la mano de Hernán Beati, un excelente profesional y profesor que conozco hace ya diez años y recomiendo plenamente para tu viaje en el mundo de este apasionante lenguaje.

¡A programar!

Lic. Maximiliano Firtman
Director ITMaster
@firt


Alfaomega e ITMaster Professional Training te dan la posibilidad de que certifiques tus conocimientos y experiencias adquiridos como lector de este libro. Su aprobación te permitirá tener la certificación en Programación PHP Inicial.

Luego de la obtención del certificado, podrás continuar tu formación en la carrera corta de Programador Web, en los másters de especialización Programador Experto PHP y Experto en Mobile Web y en los cursos cortos presenciales y online disponibles en todo el mundo.

Para dar la evaluación de certificación o recibir mayor información sobre este servicio, ingresa en el sitio Web o envíanos un e-mail a la dirección correspondiente a tu país.

España

`http://libros.itmaster.es - info@itmaster.es`

México

`http://libros.itmaster.com.mx - info@itmaster.com.mx`

Argentina y otros países

`http://libros.itmaster.com.ar - info@itmaster.com.ar`

## Plataforma de contenidos interactivos

Para tener acceso al material de la plataforma de contenidos interactivos del libro: *PHP: creación de páginas Web dinámicas*, 2ª edición, siga los siguientes pasos:

1. Ir a la página: http://libroweb.alfaomega.com.mx

2. Ir a la sección *Catálogo* y seleccionar la imagen de la portada del libro, al dar doble clic sobre ella, tendrá acceso al material descargable.

NOTA: Se recomienda respaldar los archivos descargados de las páginas web en un soporte físico.

# Más allá de HTML y CSS

# 1

## ¡No más páginas Web: aplicaciones Web!

*"No es la Programación, son los Negocios". Eso es PHP.*

Dominar el lenguaje PHP amplía nuestros horizontes profesionales como diseñadores o programadores, y nos convierte en creadores de Aplicaciones Web. Nos lleva de la mano a un mundo de comercio electrónico, redes sociales, intranets, portales de noticias y entretenimientos, un mundo "mágico" en el que podemos acceder gratuitamente a miles de sistemas completos prearmados, listos para usar (y para vender a nuestros nuevos clientes).

Nos abre un nuevo mercado, donde los clientes ya no están tan interesados en el diseño (aunque puede aportar su encanto), sino en las funcionalidades, para que, a través de un navegador, las personas puedan hacer alguna tarea concreta en su sitio Web.

### Ganando nuevos mercados a dos competidores: diseñadores gráficos y empresas de sistemas

PHP nos despega de nuestros antiguos competidores (diseñadores gráficos o programadores de aplicaciones de escritorio) y nos lleva a un nuevo mercado, en el que ofrecemos soluciones Web a comercios, empresas de diversos tamaños, profesionales, instituciones educativas, medios de difusión.

Ahora competimos con empresas de sistemas, con la ventaja de la rapidez y economía de nuestras soluciones PHP.

| LENGUAJES | HTML/CSS | PHP/MySQL | Otros (Java, .Net) |
|---|---|---|---|
| COMPETIDORES | Diseñadores gráficos | Programadores Web | Empresas de sistemas |
| TAREA PRINCIPAL | Decorar páginas (no saben programar) | Adaptar sistemas pre-armados rápida y económicamente | Hacer sistemas a medida desde cero (caros y de largo plazo) |
| NIVEL DE PRESUPUESTOS | Cientos | Miles/Decenas de miles | Decenas a centenas de miles |

**Tabla 1-1. Mercado de los sistemas prearmados**

Eso explica por qué se propaga con tanta velocidad el conocimiento de PHP entre diseñadores y programadores de otros lenguajes. Es la clave para llevar a cabo cualquier proyecto que trascienda las páginas Web HTML estáticas.

Y el detalle fundamental: con PHP, nuestros presupuestos pueden llegar a tener uno o incluso dos ceros más que los presupuestos que hacíamos como diseñadores o programadores de otros lenguajes.

Además, PHP es fácil de aprender.

## La lógica de PHP: un amigo invisible

¿Qué es PHP? PHP no se ve. Es "transparente", invisible. Por esta razón, es difícil explicar qué es y cuál su funcionamiento. Sin embargo, lo intentaremos.

Éste es el concepto más abstracto del libro, pero es imprescindible para entender qué hace PHP.

PHP es un acrónimo de *PHP: Hypertext Preprocessor*, es decir, "Preprocesador de Hipertexto marca PHP". El hecho de que sea un preprocesador es lo que marca la diferencia entre el proceso que sufren las páginas Web programadas en PHP del de aquellas páginas Web comunes, escritas solo en lenguaje HTML.

Para llegar a entender qué es un preprocesador, examinaremos primero cuál es la diferencia entre el proceso de una página Web normal (HTML) y el preproceso de una página escrita en lenguaje PHP.

## Proceso de archivos HTML

¿Cuál es el camino que sigue una página Web común (escrita en lenguaje HTML) desde que escribimos su dirección en nuestro navegador hasta que la vemos en nuestra pantalla?

- Comenzamos escribiendo en el navegador la URL deseada y pulsamos **enter** (o pulsamos un enlace con el *mouse*); en ambos casos, la barra de direcciones nos muestra la URL del archivo HTML que nuestro navegador está solicitando:

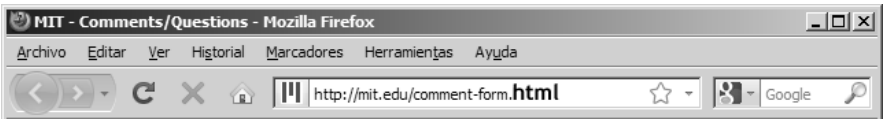


**Figura 1-1. URL de un archivo HTML.**

- En ese momento, el navegador envía una petición que solicita esa página. Ese pedido "viaja" desde nuestro navegador hasta la máquina *hosting* que hospeda el archivo requerido. Pero el navegador no remite únicamente el pedido del archivo que necesita, sino que lo acompaña con un número que nos identifica inequívocamente: nuestra dirección IP.


**Figura 1-2. El navegador solicita un archivo y envía nuestra dirección IP.**

- Podemos compararlo con un *delivery* de pizza a domicilio; para recibir el pedido, le decimos al telefonista dos datos: "cuál gusto de pizza" queremos y "a qué dirección" nos la debe enviar.
- Cuando el pedido llega al *hosting* indicado, un programa denominado servidor Web que está encendido en esa máquina, recibe el pedido y va a buscar el archivo solicitado en el disco rígido.


**Figura 1-3. El servidor Web busca en el disco rígido del *hosting* el archivo solicitado.**

- El rol del servidor Web es similar al del empleado de la pizzería que atiende al teléfono y va a buscar el pedido a la cocina de la pizzería.
- Ese servidor Web, una vez que localizó el archivo solicitado, envía, entrega o "sirve" (de ahí, su nombre: "servidor") el archivo al navegador que se había quedado esperando una respuesta en la dirección IP que lo identifica.


**Figura 1-4. El servidor Web envía el archivo solicitado a la dirección IP del navegador.**

- Equivaldría al viaje de la moto que nos trae la pizza hacia nuestra casa.
- Una vez que llegó el archivo hasta nuestro navegador, éste se encarga de interpretar los contenidos de ese archivo de texto y código HTML, armando cada elemento (textos, tablas, colores) de la página recibida en nuestra pantalla para que la podamos leer.

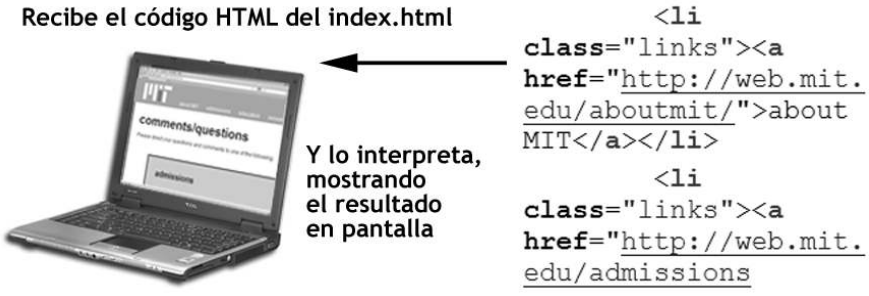


**Figura 1-5. El navegador interpreta el código HTML y muestra el resultado.**

En la metáfora de la pizza, llega el momento de recibir la pizza y servirla en nuestra mesa, lista para comer.

Podemos mencionar algunas conclusiones que identifican el proceso de archivos HTML comunes:

- Nuestro **navegador** tiene la capacidad de pedir archivos a distintos servidores Web, y a su vez, de entender y "descifrar" esos archivos recibidos, cuyo contenido es código HTML. Nuestro navegador es un programa que todo el tiempo realiza pedidos de archivos (peticiones) y recibe un archivo HTML como respuesta, que luego muestra a los seres humanos. Pide archivos y los muestra, pide y muestra...
- El **servidor Web** es un programa instalado en los *hosting*s, que todo el tiempo recibe pedidos de navegadores (se los llama peticiones), y le entrega a esos navegadores el archivo HTML solicitado.
- Este diálogo entre un navegador y un servidor Web sigue las reglas de un **protocolo** (una convención, un estándar) denominado HTTP (*HyperText Transfer Protocol* o Protocolo de Transferencia de Hipertexto).

Todo esto sucede cada vez que queremos ver un archivo HTML común.

## Preproceso de archivos PHP

Pero, ¿cuál es el camino que sigue una página Web cuya extensión es .php desde que escribimos su dirección en nuestro navegador hasta que la vemos?

Cuando la extensión del archivo solicitado es .php, se introduce un elemento diferente en este circuito:

- Hacemos el pedido de ver una página con extensión .php desde nuestro navegador:


**Figura 1-6. URL de un archivo PHP.**

- El programa servidor Web instalado en el *hosting* recibe nuestro pedido y, de inmediato, detecta que el archivo solicitado tiene extensión .php y, por lo tanto, deriva el pedido a otro programa que está encendido en esa misma máquina *hosting*, que se denomina **intérprete de PHP** (es una especie de "ser mágico", cuya presencia es muy difícil intuir, y que debemos acostumbrarnos a imaginar que "está ahí" para poder programar correctamente en PHP).


**Figura 1-7. El servidor Web le pasa el pedido al intérprete de PHP.**

- Este programa intérprete de PHP busca en el disco rígido del *hosting* el archivo .php que fue solicitado, y comienza a leer su código línea por línea, buscando determinadas "marcas" o etiquetas que nosotros, como programadores, hemos dejado escritas y que contienen órdenes destinadas a ese programa intérprete de PHP.

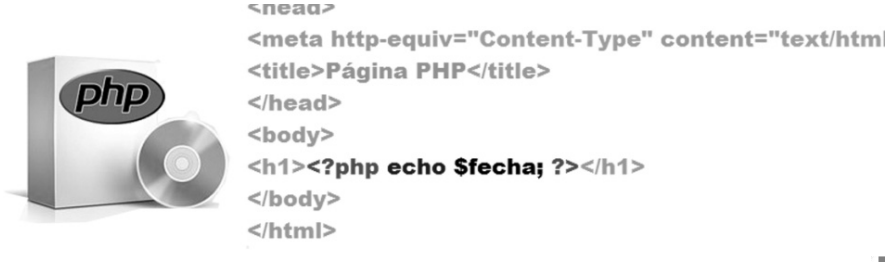


**Figura 1-8. El intérprete de PHP busca las marcas con órdenes para él.**

- Cuando este programa intérprete de lenguaje PHP encuentra estas órdenes, las ejecuta (las procesa) y, a continuación, reemplaza todas las

órdenes que hubiese entre la apertura y el cierre de la etiqueta de PHP por el resultado de procesar esas órdenes. Es decir, **borra** las órdenes del código HTML en el que estaban escritas y, en su lugar, coloca los **datos** obtenidos como consecuencia de la ejecución de esas órdenes.

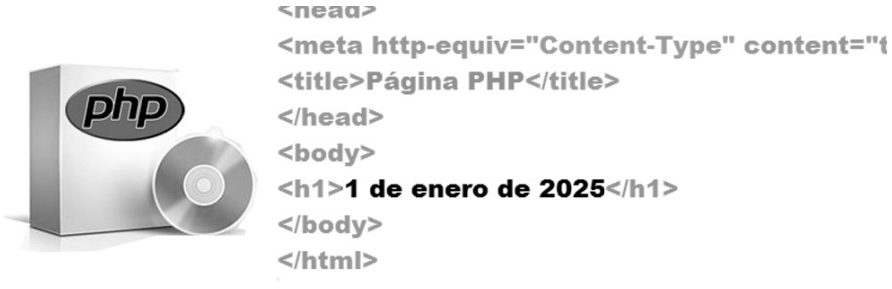


**Figura 1-9. El intérprete de PHP borra cada orden y en su lugar escribe el resultado de procesar esa orden.**

Veamos un ejemplo: supongamos que en una de esas órdenes le dijimos al intérprete de PHP que averigüe la fecha actual y que la escriba en el lugar exacto donde dejamos la orden escrita. Ésta se borrará del código fuente y, en su lugar, **quedará escrita la fecha**.

No se escribe así de simple y en español, pero ése es el concepto. Por esta razón, no debemos ejecutarlo porque no funcionará. Lo mencionamos, simplemente, para ilustrar la idea.

```
...
<body>
<h1>
<?php
Atención, software de PHP!:
Quiero que escribas la fecha actual aquí!
?>
</h1>
</body>
...
```

- El software de PHP ejecuta la orden que le dejamos escrita y, al finalizar, devuelve al software servidor Web **el texto y el código HTML producido**, para que el servidor Web lo entregue **al navegador**, que lo interpreta como si este código HTML, que incluye la fecha, hubiese estado escrito allí desde un principio:

```
...
<body>
<h1>1 de enero de 2025</h1>
</body>
...
```

- En el código fuente que le llega al **navegador**, no vemos ningún rastro de la orden que habíamos escrito para el software de PHP, ya que este software se ocupó de borrarla para que nadie la vea, y en el lugar exacto en el que habíamos escrito esa orden, colocó "el resultado de ejecutar esa orden", es decir, la fecha, que se ocupó de conseguir.

En resumen, el **preproceso** de páginas PHP consiste en esta serie de pasos: dejamos escritas entre medio de nuestras páginas algunas **órdenes** destinadas al **software intérprete de PHP** (órdenes que casi siempre consisten en que el software de PHP obtenga cierta información, como la fecha del ejemplo anterior); luego, colocamos otras órdenes para que el software intérprete de PHP "realice algo" con esa información, típicamente, que la **escriba** dentro del código fuente de la página HTML que se enviará al navegador del usuario.

En la Figura 1-10, observamos algunas órdenes en lenguaje PHP escritas en medio del código HTML de una página:

```
</head>
<body>
<h1><?php echo $fecha; ?></h1>
<h2><?php echo $titulo; ?></h2>
<p><?php echo $noticia; ?></p>
</body>
</html>
```

**Figura 1-10. Ejemplos de órdenes PHP intercaladas en el código HTML.**

## Páginas estáticas

De la diferencia entre los procesos que sufren las páginas HTML comunes y las páginas PHP, podemos concluir que las páginas Web, escritas únicamente en lenguaje HTML, son **estáticas**: es decir, **nunca cambian** su contenido: pase lo que

pase, lo que llegará al navegador del usuario es lo que ha sido escrito en ellas por el diseñador Web, ni más ni menos, siempre lo mismo.

### Páginas dinámicas

Por el contrario, las páginas que incluyen código escrito en lenguaje PHP, nos dan la oportunidad de **personalizar** su contenido sobre la base de ciertas órdenes escritas (como en el ejemplo anterior de la fecha, la página hoy mostrará una fecha y mañana otra, y así sucesivamente; es decir: siempre generará un **contenido distinto**, variable).

El contenido de esas páginas, al menos en partes de ellas, cambiará y no será siempre el mismo, ya que dependerá de la información que obtenga el software de PHP y coloque en ellas. Serán páginas dinámicas.

### Las bases de datos

El concepto de páginas dinámicas que acabamos de esbozar, se complementa a la perfección con las **bases de datos**, ya que éstas se ocupan de almacenar datos y, las páginas dinámicas, de **leerlos** y **mostrarlos** dentro de ellas.

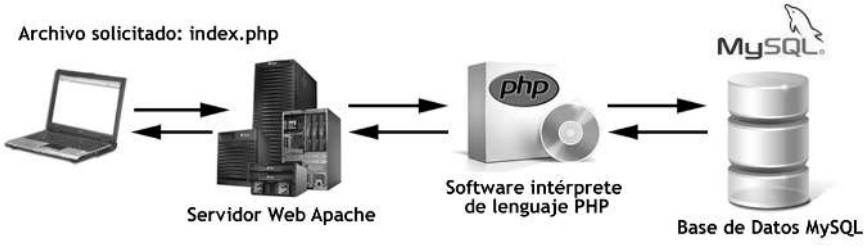


**Figura 1-11. Esquema de interacción entre los programas que participan cuando las páginas utilizan una base de datos.**

### MySQL: la base de los proyectos Web exitosos

Desde la aparición de PHP, la base de datos que siempre estuvo asociada a PHP fue MySQL, no solo gracias a su gran potencia y rapidez, sino, fundamentalmente, a que su licencia no tenía costo para los *hostings* (a diferencia de Oracle, SQL Server y otros competidores), lo que contribuyó a su rápida difusión y a su mejoramiento vertiginoso en sucesivas versiones, que contribuyeron a que se convierta en la base de datos que provee almacenamiento a sistemas de la envergadura de Google, Facebook y millones de otros populares sitios Web en todo el mundo.

Pero, ¿para qué sirve usar una base de datos en un sitio Web? Veamos algunos de sus posibles usos.

### *Un camino de ida: del almacén a las páginas*

Pensemos en un ejemplo: una página que muestra un catálogo de libros:


**Figura 1-12. Página que muestra información de una base de datos.**

Si pretendiéramos hacer este tipo de sitios únicamente con HTML, deberíamos crear esta página manualmente y, en caso de alguna modificación en los datos de alguno de los productos (un nuevo producto, o quitar uno existente), deberíamos modificar el archivo HTML a mano, y volver a colocar en el *hosting* una copia actualizada de esta página HTML mediante FTP. Así, en todos los cambios de cualquiera de los miles de artículos que se muestran.

En cambio, con PHP y MySQL, podemos crear una **base de datos** que contenga información sobre los miles de productos que se mostrarán. Luego, utilizaremos una **página dinámica** escrita en PHP, que incluya la orden de leer esa base de datos y publicar los productos. Si modificamos los datos, éstos se reemplazarán directamente en la base de datos, sin necesidad de modificar ni una línea de las páginas Web que exhiben los productos, ya que la orden sigue siendo exactamente la misma: “leer la base y mostrar todos los productos”.

Otra vez, para graficar la idea, la simplificaremos, pero pronto lo aprenderemos a programar para que funcione:

```
...
<p>
<?php
Atención, software de PHP!:
Quiero que le pidas a la base de datos la lista de
productos y la muestres aquí!
?>
</p>
...
```

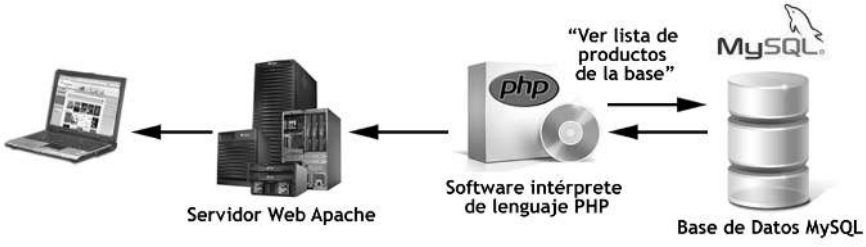


**Figura 1-13. Esquema del circuito de una página que muestra información almacenada en una base de datos.**

### *Un camino de vuelta: recepción de datos de formularios*

Pero aquí no terminan las ventajas de almacenar en una base de datos la información de una página, sino que PHP y MySQL agregan un camino imposible de lograr solamente con HTML: la posibilidad de enviar datos desde el navegador del usuario hacia el servidor, y que estos datos queden almacenados en la base de datos del *hosting*.

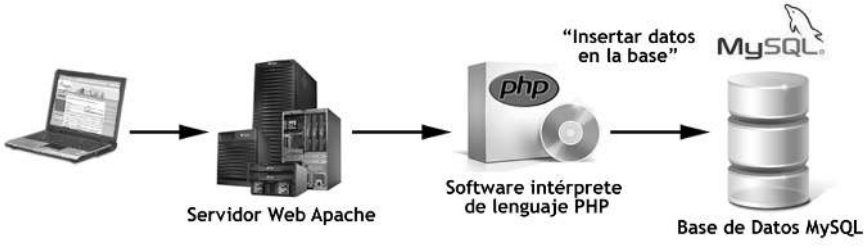


**Figura 1-14. Esquema del circuito de una página que envía datos hacia el servidor.**

¿Posibles usos?: envío de formularios con consultas, currículums, formularios para paneles de administración que actualizan datos de catálogos, envío de mensajes para publicar en foros, envío de fotografías, etc.

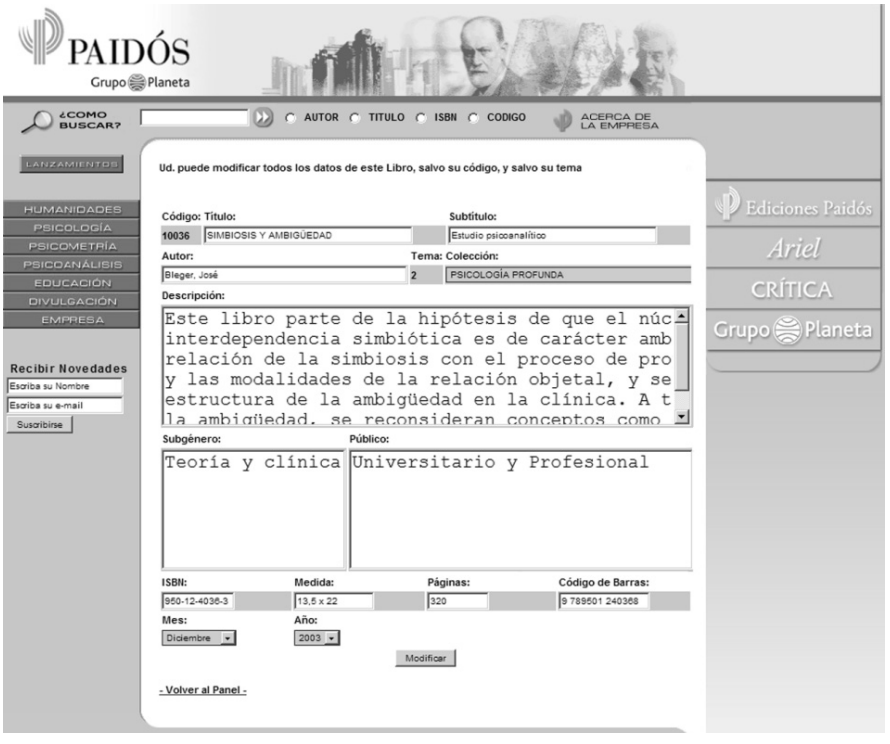


**Figura 1-15. Ejemplo de formulario que envía datos al servidor y los guarda en una base de datos.**

### *Sitios autoadministrables: CMS con Front-end y Back-end*

La combinación de los dos "caminos" anteriores (enviar datos **hacia** la base de datos y, luego, **mostrar** esos datos mediante páginas dinámicas) permiten crear sistemas denominados CMS (*Content Management System* o Sistemas de Administración de Contenidos) que gestionan la publicación de contenidos de portales, sistemas de comercio electrónico, comunidades *on-line*, y muchas otras aplicaciones, siempre sobre la base de esos conceptos básicos: el usuario envía datos hacia el *hosting*, que se almacenan en la base de datos y, luego, esos datos se muestran (a él o a otros usuarios autorizados).

Se denomina *Front-end* a la parte visible por los usuarios, y *Back-end* al panel de administración de acceso restringido solo a los usuarios autorizados (como el de la última imagen).

### *Millares de CMS de uso libre y gratuito*

Existen literalmente miles de CMS con distintas finalidades, listos para usar, con licencia de libre uso y modificación (típicamente, la licencia GPL y similares), creados con PHP y MySQL. Algunos de los más famosos son *Joomla, WordPress, osCommerce, Mambo, Drupal, Moodle, Magento, OpenRealty,* pero existen miles más.

El secreto del negocio está en **saber programar en PHP** lo suficiente como para **entender** y poder realizar algunas **modificaciones** para adaptar estos CMS a las necesidades específicas de cada cliente, y poder ofrecerles soluciones de bajo costo y muy rápida implementación.


**Figura 1-16. Nuestro negocio: tomar código prearmado y modificarlo según las necesidades de nuestros clientes.**

De aprender lo necesario de PHP y MySQL para poder hacer esas modificaciones, es de lo que nos ocuparemos en el resto de este libro.

# El ambiente PHP

# 2

## Esas extrañas siglas: LAMP, MAMP, WAMP, xAMP

Hojeando libros o mirando tutoriales en la Web, encontraremos muchas veces asociada a PHP alguna de estas siglas. ¿A qué se refieren?

Es muy simple: si comenzamos por el final, la "P" corresponde a **PHP**, la "M", a **MySQL** y, la "A", a **Apache**, que es el servidor Web más usado en los *hosting*s que tienen instalado el intérprete de PHP.

¿Y esa primera letra, la "L", "M", "W" y "x"? Son las iniciales de los **sistemas operativos** más usados: "L" de Linux (cualquiera de sus miles de variantes); "M", de Mac; "W", de Windows. Y la "x" se usa como un comodín, cuando se puede prescindir de un sistema operativo en particular y nos referimos a una instalación de Apache, MySQL y PHP genérica, en cualquier sistema operativo.

Esta sigla resume el entorno bajo el cual estamos usando a PHP. Así que, si están usando Windows y, a continuación, instalan Apache, MySQL y PHP, estarán trabajando bajo una plataforma WAMP (Windows con Apache, MySQL y PHP). Por el contrario, la mayoría de los *hosting*s se basan en LAMP (Linux, Apache, MySQL y PHP).

## El hosting

### Eligiendo un buen hosting con PHP y MySQL

Sin lugar a duda, es imprescindible para probar nuestros desarrollos que dispongamos de un *hosting*, que puede funcionar bajo cualquiera de las plataformas recién mencionadas (aunque la más recomendable es **LAMP**, ya que es la que más potencia permite sacarle a PHP). ¡Atención!: esto no significa que nosotros debamos usar Linux en nuestra computadora, sino que el *hosting* usará ese sistema operativo. Nosotros simplemente nos conectaremos mediante algún programa de FTP para colocar en ese servidor nuestros archivos, y podemos hacer esto desde cualquier sistema operativo.

Para contratar algún servicio de *hosting*, debemos tener en cuenta que no son todos iguales, sino que existen distintas versiones de PHP y pueden tener instalada cualquiera de ellas. Lo ideal es conseguir *hostings* que posean la versión de PHP más actualizada que nos resulte posible: podemos consultar cuál es el número de la última versión de PHP si entramos a la Web oficial de PHP, en http://www.php.net

Además de buscar *hostings* con una versión actualizada de PHP, también debemos intentar que posean una versión lo más actualizada posible de MySQL, y del mismo modo que con PHP, el número de versión lo averiguaremos entrando a la Web oficial, en este caso, de MySQL: http://www.mysql.com

Hechas esas recomendaciones, el resto es sentido común: es mejor un *hosting* con soporte 24 horas, que podamos pagarlo en nuestra moneda local sin gastos de transferencia, y que no sea el de moda ni el más barato, porque suelen tener problemas frecuentemente. Suele ser útil que posean teléfono al que podamos llamar al costo de una llamada local para gestionar reclamos con más efectividad que por *e-mail* o ticket de soporte. Un *hosting* promedio, sin demasiados usuarios, suele ser mejor negocio que otro más barato pero saturado en su capacidad de dar soporte (más que un producto, el *hosting* es un servicio y es clave que tengamos acceso a las personas que nos podrán dar ayuda en caso de necesidad, de nada sirven esos soportes técnicos que repiten respuestas mecánicamente).

## El servidor local para pruebas

Si bien siempre probaremos nuestros códigos en el *hosting* que nuestro cliente usará para su proyecto (para luego no tener sorpresas de configuración a la hora de dejar el sistema *on-line*), será muchísimo más práctico probar previamente nuestro código PHP localmente, en nuestra propia computadora, mientras

programamos, sin necesidad de esperar a transferir los archivos por FTP al *hosting* ante cada mínimo cambio que hagamos, ya que resulta muy molesto.

Para trabajar con un **servidor Web local** (dicho en palabras poco técnicas, una especie de simulador de *hosting*), tendremos que colocar nuestros archivos dentro de una **carpeta** en particular, que contendrá todos los archivos que programemos, tanto los ejercicios de este libro como nuestros propios proyectos profesionales. Y para que esos archivos funcionen, tendremos que mantener encendido un programa denominado servidor Web que, justamente, le "servirá" al navegador esos archivos ya procesados.

Ahora vamos a **descargar** e **instalar** ese software que nos permitirá montar nuestro propio servidor Web local.

Y, a continuación, veremos cómo acceder con el navegador a los archivos que serán servidos por ese software que instalemos.

### Cómo descargar un servidor de pruebas

Aunque podríamos instalar todo el software necesario para programar en PHP y MySQL manualmente, es un trabajo bastante complejo y es probable cometer errores de configuración difíciles de solucionar sin ayuda de un Administrador de Sistemas. Por eso, es que existen muchos **instaladores automáticos** de todos los programas necesarios para probar código PHP en nuestra propia computadora: algunos de los nombres de estos instaladores son easyPHP, XAMPP, AppServ, etc. En este libro, utilizaremos el **XAMPP**, que es un instalador automático, que configura en instantes todos estos programas:

1. Un programa **servidor Web** llamado Apache,
2. El programa **intérprete del lenguaje PHP** propiamente dicho,
3. Un **programa gestor de bases de datos** denominado MySQL,
4. Una **interfaz visual** para interactuar con esas bases de datos, cuyo nombre es phpMyAdmin.

Este práctico paquete instalador "todo en uno", se descarga gratuitamente de: http:// www.apachefriends.org. Buscamos la versión correspondiente a nuestro sistema operativo (Windows, Linux, Mac, Solaris) y descargamos el instalador.

### Cómo instalar el servidor de pruebas

Una vez terminada la descarga, haremos doble clic en el instalador que hemos descargado, y aceptaremos todo lo que nos pregunte, con total confianza (es software libre además de gratuito, así que nos permitirá utilizarlo sin limitaciones, de forma totalmente legal).

Una vez elegido el idioma de la instalación (en español), y manifestado nuestro acuerdo con la licencia de los productos, nos advertirá (en inglés) que no se recomienda utilizar este instalador en un *hosting* público real, por lo que seguimos adelante pulsando en **Siguiente** y, aquí, llegamos al momento crítico de la instalación: cambiaremos la **ruta de instalación** por defecto, por otra. Podemos ver que la ruta por defecto que figura en **Destination folder** es "C:\" así que la cambiaremos por esta:

C:\servidor\

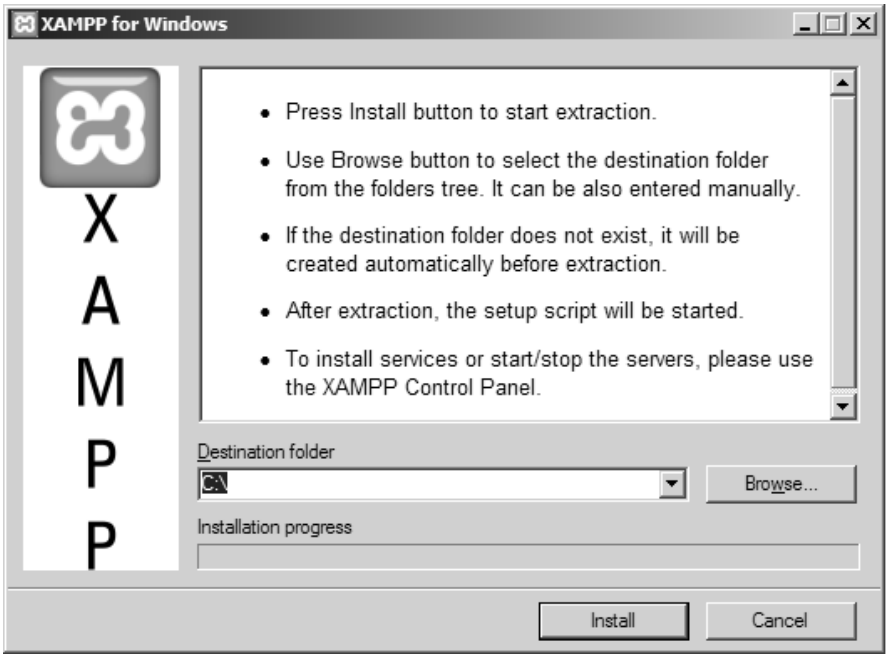


**Figura. 2-1. Ruta por omisión.**

Para cambiarla, simplemente **escribimos desde la última barra en adelante**, y solamente la palabra "servidor\", para que quede de esta manera:

Desde ya que no es obligatorio utilizar exactamente esta misma ruta de instalación, el lector puede elegir libremente cualquier otra carpeta distinta si así lo prefiere: es solo para que coincida con lo que se mencionará en otras partes del libro.

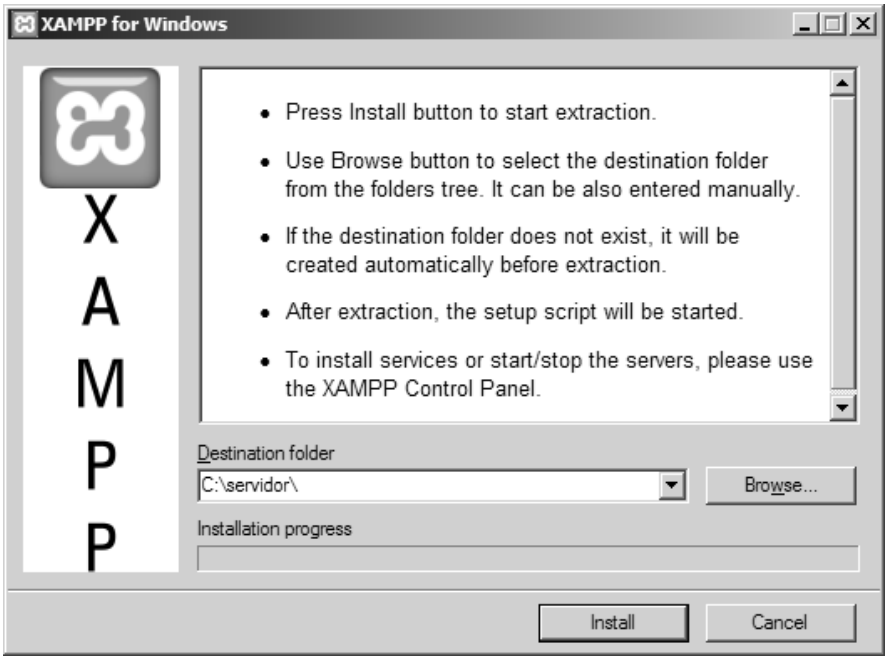


**Figura 2-2. Ruta que utilizaremos en este libro.**

Recomiendo que usemos **esa ruta**, ya que todas las explicaciones del libro están armadas tomando como base esa ruta de instalación.

A continuación, pulsamos el botón **Install**, y esperamos aproximadamente un minuto mientras extrae todos los paquetes necesarios.

Si al momento de leer este libro la versión de este instalador cambió o los pasos y pantallas son otros, esto no debe preocupar al lector; se debe simplemente seguir paso a paso las instrucciones que aparezcan en pantalla.
El único punto importante es tener conciencia de cuál es la carpeta en la que estamos instalando los servidores, el resto de configuraciones no es importante.

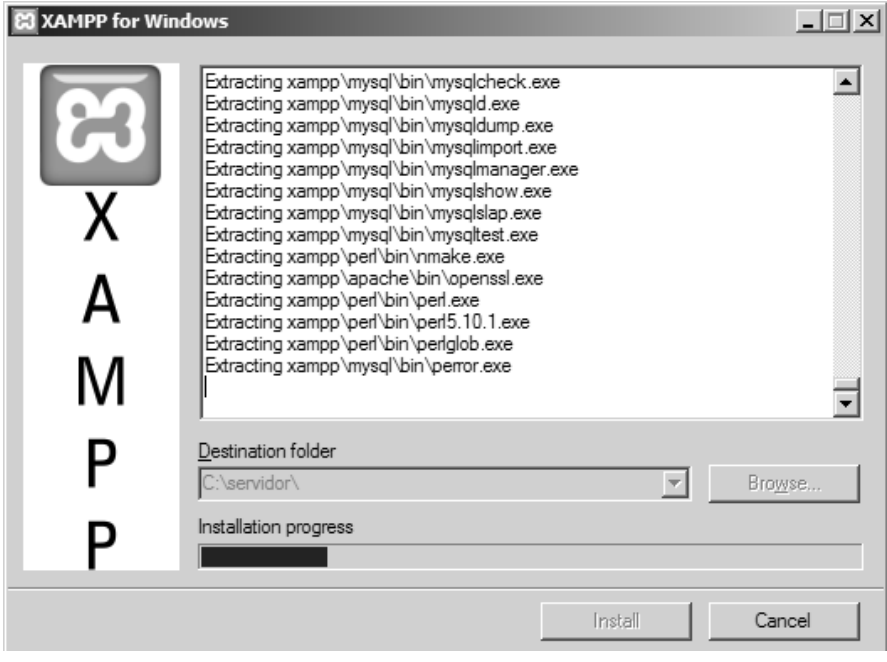


**Figura 2-3. Pantalla que veremos mientras esperamos.**

Por último, se abrirá una ventana de sistema que nos preguntará algunas cosas:

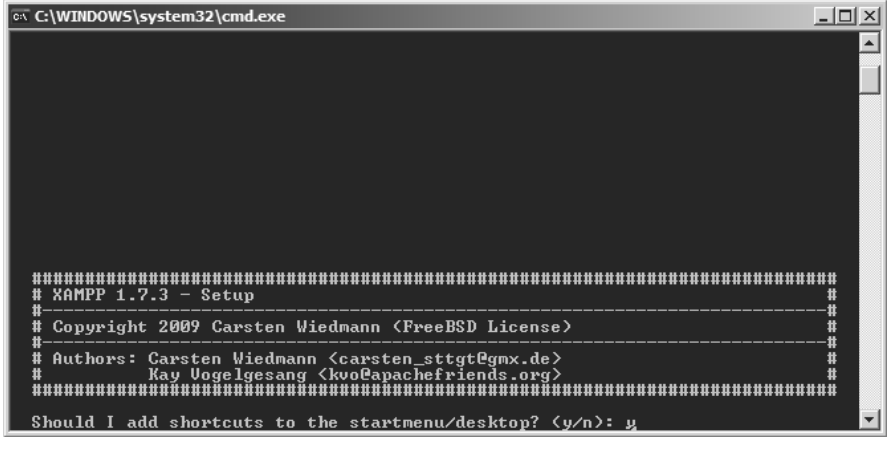


**Figura 2-4. Accesos directos.**

Nos está preguntando si agrega accesos directos al **menú Inicio** y al **Escritorio**, pulsamos una **y** para indicar que sí y, después, pulsamos **Enter**.

Luego, detectará que hemos cambiado la carpeta por omisión y nos pedirá que confirmemos la modificación de las rutas de todos los archivos de configuración (es fundamental que aquí respondamos que sí, pulsando **y** y, a continuación, **Enter**):

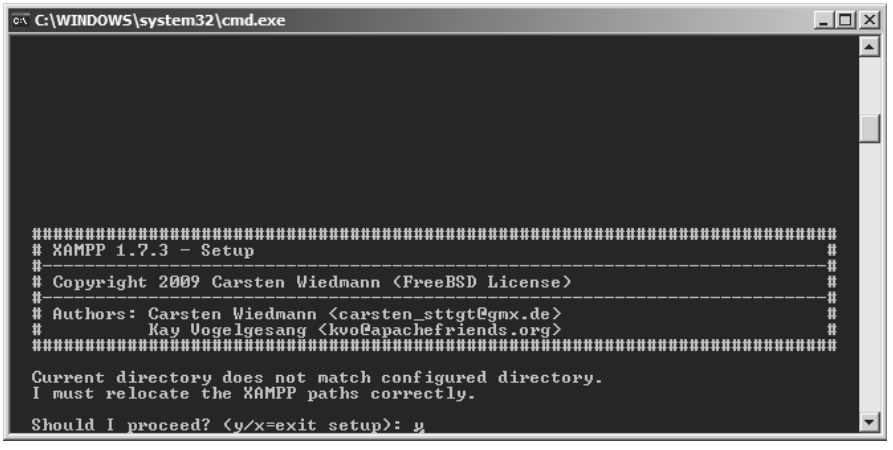


**Figura 2-5. Confirmamos cambio de directorio de instalación.**

Acto seguido, nos preguntará si nos interesaría crear una instalación portátil –para discos portátiles–, pero, como, por el momento, no es nuestra intención, pulsaremos la letra **n** y, luego, **Enter**:

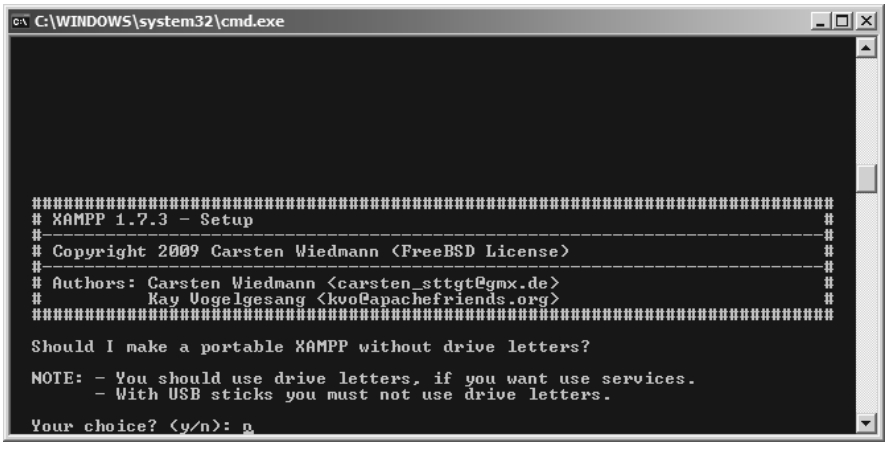


**Figura 2-6. Manifestamos que ahora no estamos interesados en crear una instalación portable.**

Ahora, realizará el cambio de configuración que habíamos autorizado dos pasos atrás, lo que demorará unas decenas de segundos, y nos indicará que pulsemos **Enter**:

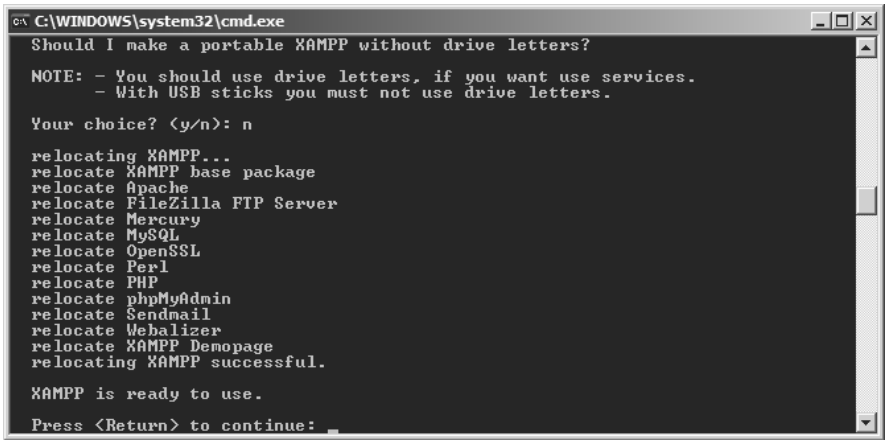


**Figura 2-7. Final de la instalación.**

Por último, configurará la zona horaria según lo detectado en el sistema operativo, y nos pedirá que pulsemos **Enter** otra vez:

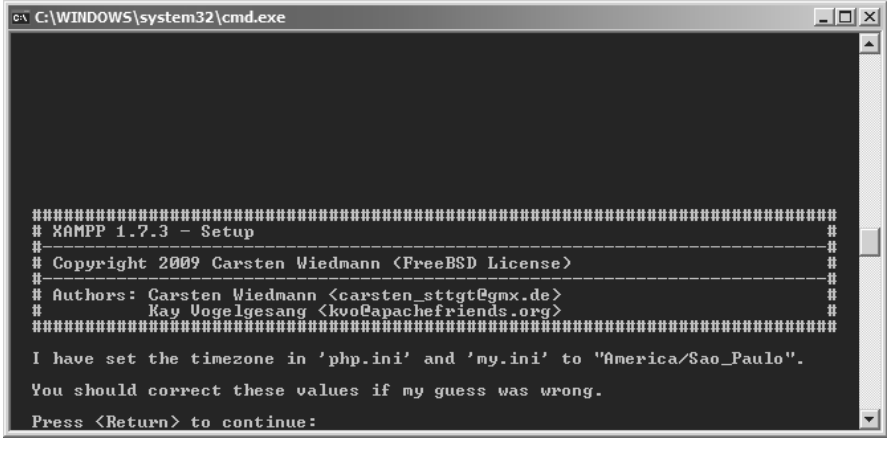


**Figura 2-8. Autodetección de zona horaria.**

Llegado este momento, nos muestra una pantalla con un menú de opciones:

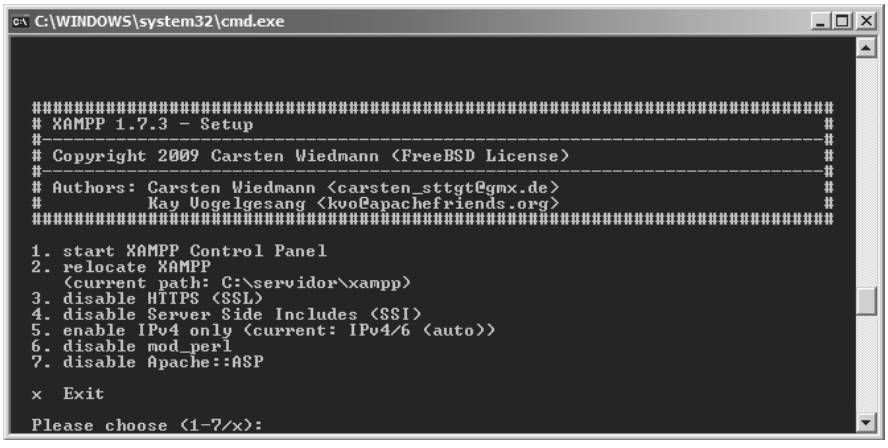


**Figura 2-9. Opciones para comenzar.**

Podemos cerrar esta ventana de sistema escribiendo una **x** y pulsando **Enter**, porque ya tenemos todos los programas necesarios instalados.

## Cómo encender y apagar el servidor de pruebas

1. Cómo encenderlo:

Para "encender" este software, debemos pulsar el ícono que dice **XAMPP Control Panel**, que quedó en nuestro Escritorio, o ingresar al menú:

**Inicio** -> **Todos los programas** -> **XAMPP for Windows** -> **XAMPP Control Panel**

Esto abrirá el siguiente panel que controla el encendido y apagado de todas las aplicaciones que fueron instaladas por el XAMPP. (ver Fig. 2-10)

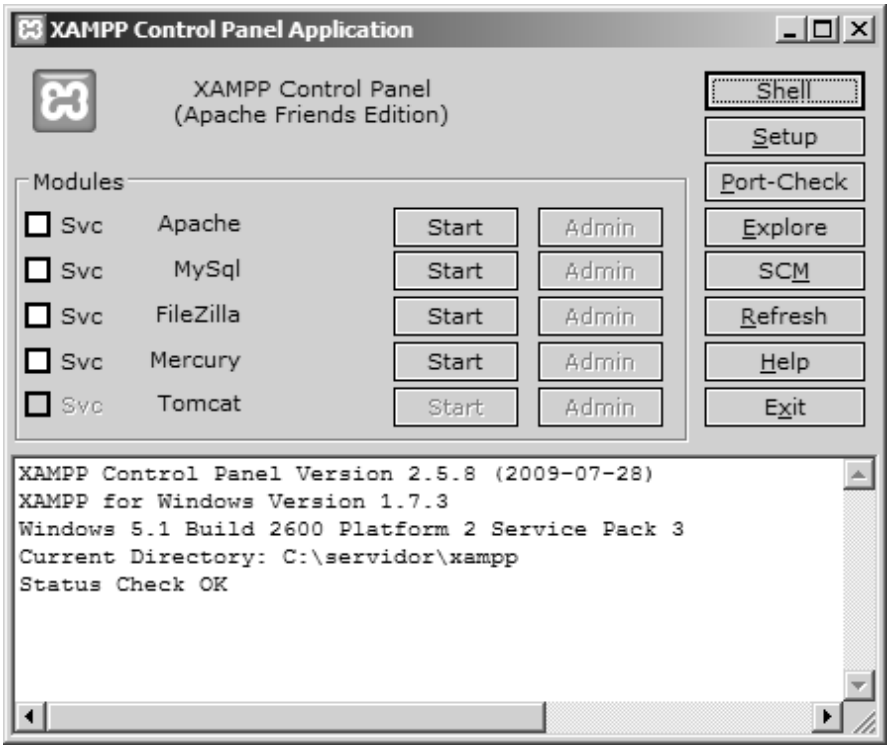


**Figura 2-10. Panel de control de las aplicaciones instaladas por el XAMPP.**

Para encender el servidor Web Apache, que es lo que precisamos por ahora para programar en PHP, pulsamos el botón **Start** que se encuentra a la derecha de la palabra **Apache**: sucederán varias cosas: la primera, ese botón ahora dirá **Stop**; la segunda, a la izquierda del botón que acabamos de apretar aparecerá la palabra **Running** sobre un fondo verde, que indica que el servidor Web Apache ya está encendido; además, en el recuadro blanco en el que aparecen los mensajes de texto, veremos que dice **Apache started**.

Cuando, más adelante en el proceso de aprendizaje, utilicemos el sistema gestor de bases de datos MySQL, lo encenderemos desde este mismo panel. Por ahora, solamente encenderemos el servidor web Apache, que incluye como uno de sus módulos al programa intérprete de lenguaje PHP.

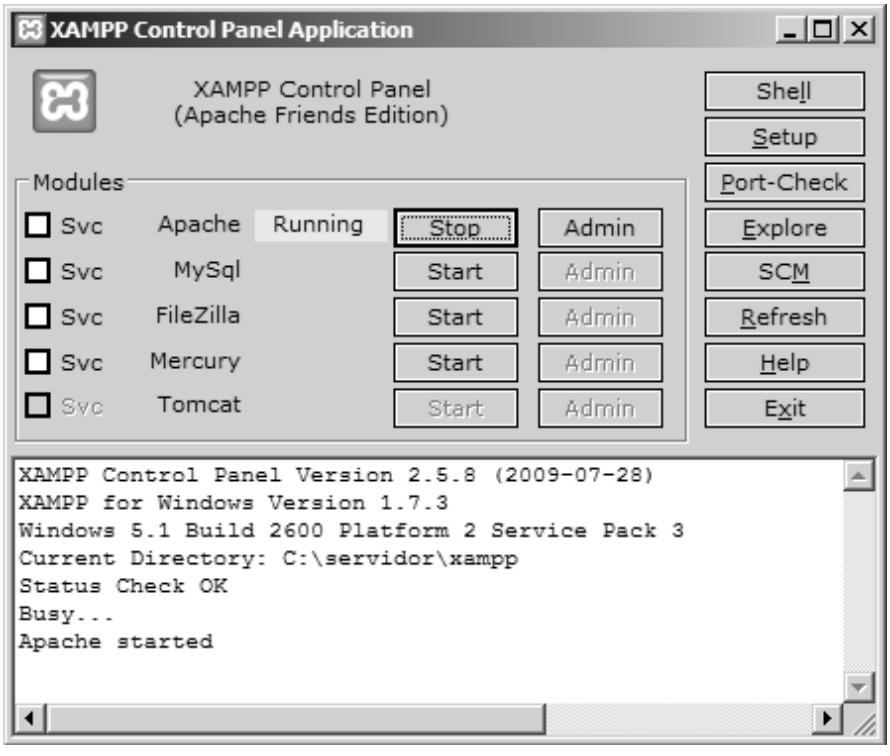


**Figura 2-11. Panel de control con el servidor Web Apache encendido.**

Si es la **primera vez** que lo encendemos, el *firewall* de Windows abrirá este cuadro de diálogo, que nos pedirá permiso para desbloquear el servidor Web que acabamos de encender. Para poder usar el servidor Web local, debemos pulsar en **Desbloquear**.

Según la versión del sistema operativo que estemos utilizando, las pantallas del firewall pueden ser diferentes. Asimismo, si tenemos instalado algún antivirus o programa que incluya protección a nivel firewall, deberemos autorizar la ejecución de nuestro servidor, o de lo contrario no podremos utilizarlo.


**Figura 2-12. Alerta del Firewall de Windows, para que autoricemos el desbloqueo del servidor Web Apache.**

Ahora, podremos minimizar o cerrar el Panel de Control de XAMPP y, aunque lo cerremos, los programas que hemos iniciado seguirán funcionando.

Nos daremos cuenta de que el Panel de Control está abierto, si miramos en la barra de tareas de Windows y encontramos el ícono de XAMPP:


**Figura 2-13. Barra de tareas mostrando el ícono de XAMPP encendido.**

Si le hacemos **clic a ese ícono** en la barra de tareas, **se abrirá nuevamente el Panel de Control**.

2. Cómo apagarlo:

Cuando terminemos de programar en PHP, podremos "apagar" el servidor Web, para eso, abriremos el Panel de Control de XAMPP, y pulsaremos el botón gris que dice **Stop**, a la derecha de **Apache** y de la palabra **Running** sobre fondo verde, y se apagará el servidor Web Apache en pocos segundos.

## Cómo configurar el servidor de pruebas

Para programar localmente en PHP con mayor comodidad, es muy conveniente dedicar un minuto a configurar que el programa intérprete de PHP nos avise cada vez que cometamos un **error de sintaxis** (cosa que en un servidor Web de un

*hosting* no es recomendable, porque puede dar demasiada información a un potencial intruso pero que, en un servidor que solo nosotros usamos, puede ser de mucha utilidad).

Es muy sencillo realizar esa configuración; para ello, abriremos con el bloc de notas el archivo ubicado en:

C:\servidor\xampp\php\php.ini

Este es el archivo de configuraciones del programa intérprete de PHP, que se lee cuando encendemos el servidor Web Apache.

Lo abrimos con el bloc de notas, vamos al menú **Edición → Buscar**, y buscamos **error_reporting** y, cuando lo encuentra por primera vez, vamos a **Buscar siguiente**:

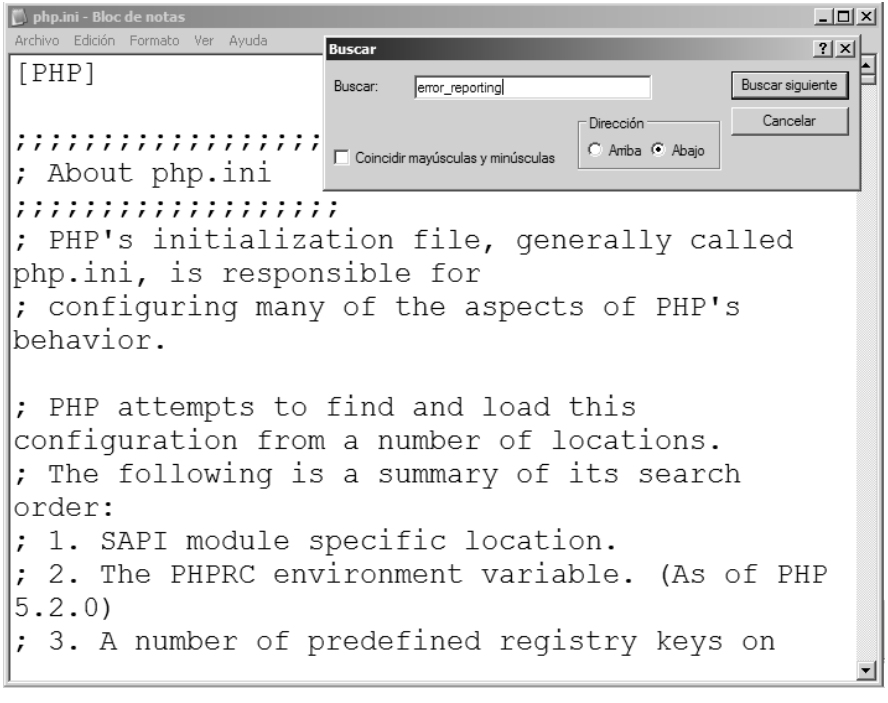


**Figura 2-14. Buscamos error_reporting hasta que comience sin punto y coma el renglón que contenga esa directiva.**

Notaremos que por omisión ese renglón contiene esto:

```
error_reporting = E_ALL & ~E_NOTICE & ~E_DEPRECATED
```

Simplemente, copiaremos lo que dice dentro de ese mismo archivo, tres líneas antes, a la derecha de "Development value", y lo pegamos a la derecha de **error_reporting**, para que quede así:

```
error_reporting = E_ALL | E_STRICT
```

Una vez hecho esto, dentro del bloc de notas en el que estamos, vamos al menú **Archivo → Guardar**. Si teníamos encendido el servidor Web Apache, lo apagamos y volvemos a iniciarlo para que lea esta nueva configuración. Y, con este único cambio, ya tenemos todo preparado como para comenzar a programar en PHP.

Esta es la única vez que vamos a tener que configurar estas cuestiones técnicas, propias de Administradores de Sistemas y de Redes; en la vida real, todo esto ya está funcionando y configurado en los *hosting*s que uno contrata para sus sitios Web.

Como conclusión: al servidor Web Apache lo vamos a prender cuando queramos probar nuestros archivos PHP, es un software que "no se ve" nunca, ya vamos a entender esto al usarlo, muy pronto.

Si, por casualidad, ya tuviéramos otro **servidor Web** instalado en nuestra computadora (típicamente el *Internet Information Server* o algún otro Apache), deberemos apagarlo cada vez que queramos usar este servidor que acabamos de instalar.
No pueden estar encendidos ambos **a la vez** (podrían si se configuraran los puertos de una forma especial, pero es más fácil apagar uno y prender el otro cuando lo necesitemos usar).

## Cómo crear y dónde colocar los archivos

Para comenzar a programar en PHP, usaremos nuestro **editor HTML** favorito (Dreamweaver, Aptana, KomodoEdit, etc.). Definiremos un **nuevo "proyecto"** o **"sitio"** (según como lo llame el editor) con el nombre que queramos, pero lo importante es que la **carpeta raíz** de ese nuevo sitio/proyecto que definimos sea:

C:\servidor\xampp\htdocs\

Recordemos que esta es la ruta donde hemos instalado el servidor local, es decir, C:\servidor
En caso de haber elegido otra carpeta al momento de la instalación, reemplacemos por esa carpeta.

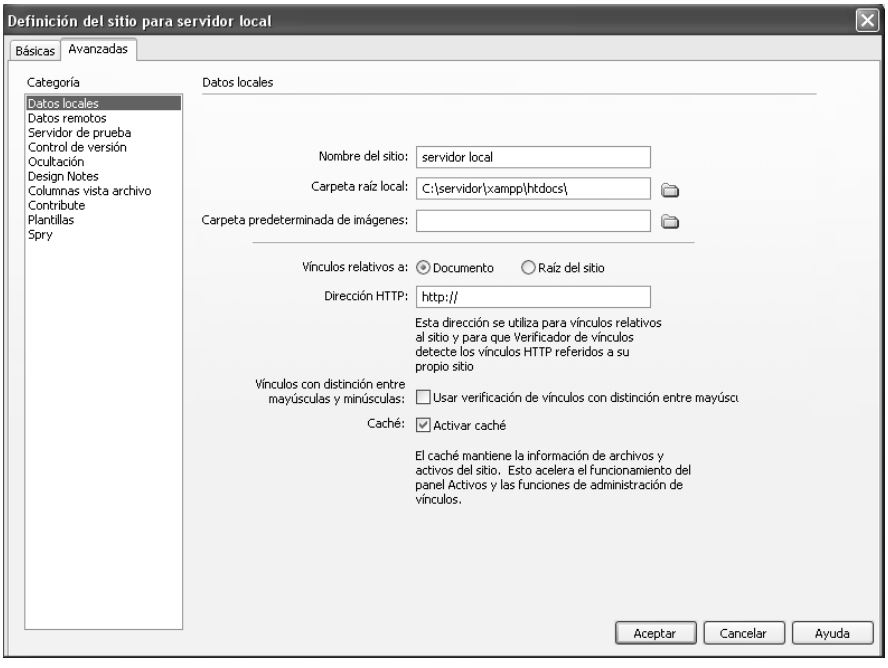


**Figura 2-15. Definiremos un sitio o proyecto en la carpeta raíz del servidor local.**

Por ejemplo, para el primer ejercicio del libro, crearemos una carpeta llamada "ejercicio1", que entonces va a quedar ubicada en:

C:\servidor\xampp\htdocs\ejercicio1\

Para crear los archivos PHP que daremos como ejemplo en este libro, al ir al menú **Archivo → Nuevo** del Dreamweaver, en lugar de elegir **Página en blanco → HTML** vamos a elegir la última opción de esa segunda columna, la que dice **PHP** (esto en el Dreamweaver, en otros editores, simplemente ir a **Nuevo Archivo**):

Si luego de pulsar en **Crear** vemos la pantalla en blanco, es porque estamos trabajando en la **Vista de Diseño** del Dreamweaver, debemos trabajar en **Vista de Código**.

En ese caso, debemos pulsar en el botón **Código** y veremos el **código fuente** del archivo:

Desde ya que podemos utilizar cualquier otro editor de código HTML, aunque al usar Dreamweaver contamos con la facilidad de que posee cargado el listado completo de funciones del lenguaje PHP y sus parámetros, con lo que se vuelve muy fácil programar en PHP.

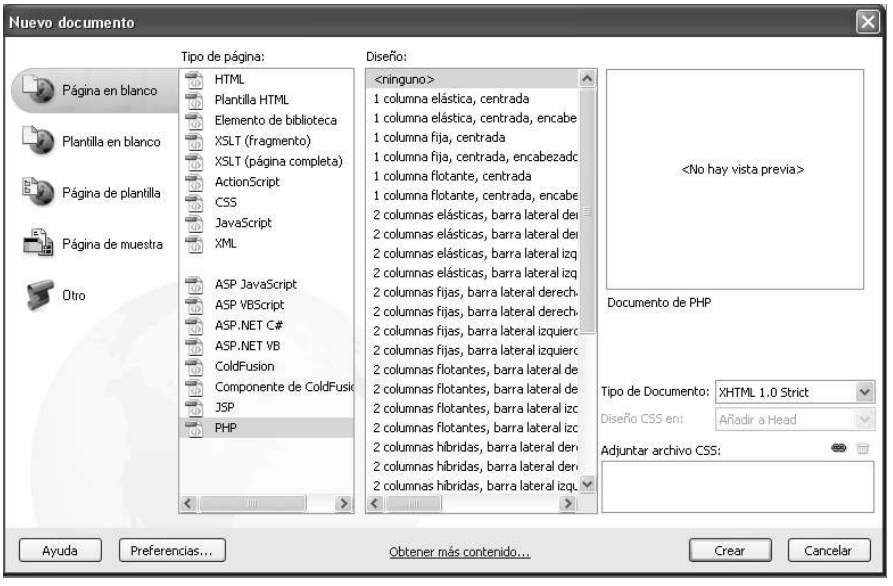


**Figura 2-16. Elegiremos nuevo archivo PHP.**

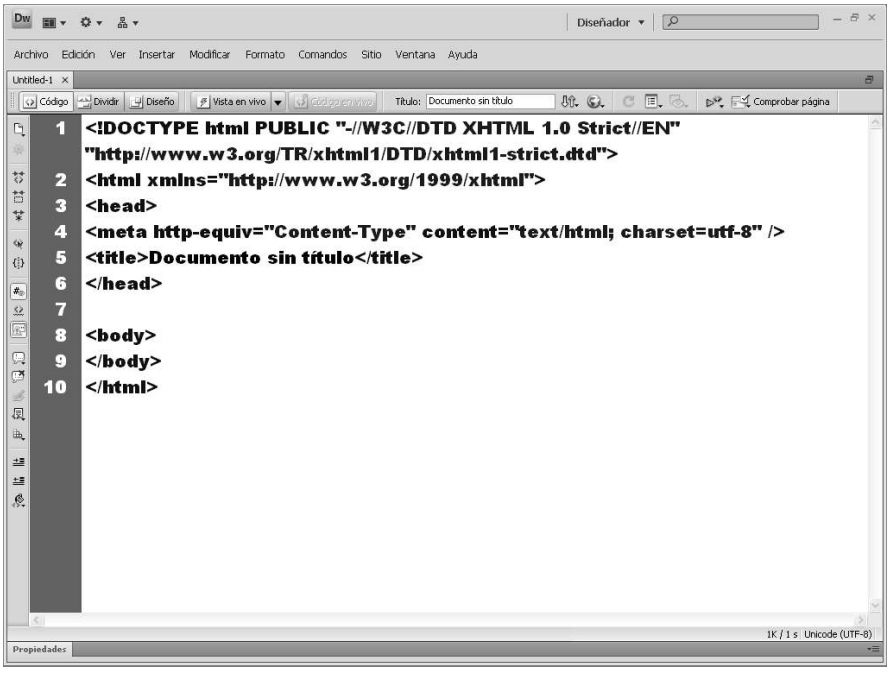


**Figura 2-17. Trabajaremos siempre en la vista Código.**

A continuación, **guardaremos** el archivo, y deberemos colocarle (obligatoriamente) como **extensión .php** en lugar de .html, como hacíamos hasta ahora; ingresamos en el menú **Archivo → Guardar como...**, y lo denominaremos **ejercicio1.php**. Luego, lo colocaremos dentro de una carpeta llamada **ejercicio1**; entonces lo guardaremos aquí:

C:\servidor\xampp\htdocs\ejercicio1\ejercicio1.php

Es importante que comprobemos que haya sido bien escrita la **extensión** del archivo **.php**, ya que, de lo contrario, nuestro servidor Web no lo podrá ejecutar.

Si nuestro editor no nos muestra la extensión de los archivos que creamos, es probable que tengamos que configurar nuestro sistema operativo para que no nos esconda las extensiones.

Para ello, abrimos el programa **Mi PC** y entramos al menú **Herramientas → Opciones de carpeta**, y pulsamos en la solapa **Ver**. Veremos algo como lo que muestra la siguiente figura:

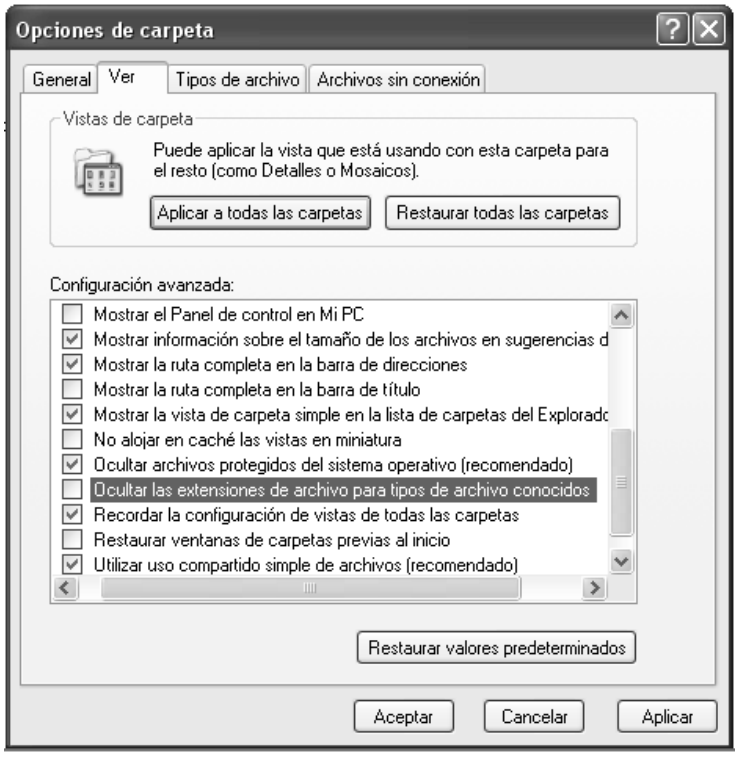


**Figura 2-18. Nos aseguraremos de que nuestro sistema operativo no oculte las extensiones de los archivos.**

Tenemos que asegurarnos que esté **sin marcar** la opción que dice **Ocultar las extensiones de archivo para tipos de archivo conocidos** y que quede vacío el cuadrito de selección. Luego, pulsamos **Aceptar** y veremos las extensiones de los archivos en todos los programas.

Ahora, programaremos una simple línea de código PHP para que esta página deje de estar vacía.

Siempre en **Vista de Código** en nuestro Dreamweaver (o en el editor que usemos), para esta prueba borraremos todo el código fuente HTML que el Dreamweaver genera (DTD, etiquetas html, head, body, etc. hasta que no quede nada):

Luego, dentro de ese código fuente, escribiremos únicamente esto:

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

Guardaremos los cambios que realizamos en el archivo, y... ¡ya hemos creado nuestro primer archivo PHP! ¡Felicitaciones!

### Cómo navegar por archivos PHP usando el servidor de prueba

Si ahora queremos que el servidor Web local procese ese archivo (simulando lo que se vería en un *hosting* verdadero), tenemos que **encender** el servidor Web Apache desde el panel de control del XAMPP, y luego escribiremos en la barra de direcciones de nuestro navegador la URL de nuestro archivo PHP:

http://localhost/ejercicio1/ejercicio1.php

Es decir, todas las carpetas previas a **htdocs** (ésta incluida) cuando vamos al navegador no se escriben, se reemplazan por **http://localhost** y, de ahí en más, sí se escriben los nombres de las **subcarpetas y de los archivos** que queramos ver (en este caso, la carpeta **ejercicio1** y el archivo **ejercicio1.php**). Podemos pensar las partes de esta URL tal como si **localhost** fuera el nombre de un dominio.

Al cargarse esa página con el servidor encendido, veremos esto:

Una vez más, lo que veamos en pantalla puede tener ligeras diferencias con la siguiente captura de pantalla al momento en que lo probemos, ya que se suele cambiar en distintas versiones de PHP. Pero eso no debe preocuparnos.

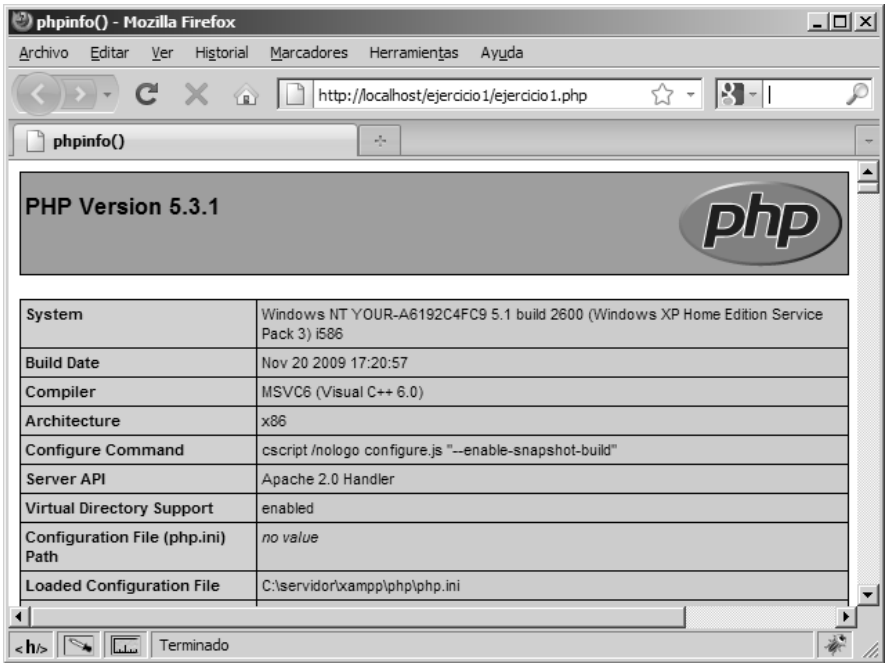


**Figura 2-19. Página con datos, generada por la función "phpinfo()" que ejecutamos.**

Es decir, observaremos una pantalla larga, con colores celeste, azul, gris y blanco. Si la visualizamos es porque hemos configurado todo correctamente, y ya podemos disfrutar de los beneficios de crear nuestra primera página con PHP (fue suficiente apenas una línea de código para generar una página completa llena de datos, lo que no es poco, ¿no es cierto?).

Por supuesto, también deberíamos probar siempre nuestros ejercicios *on-line*, en un *hosting* que soporte PHP y MySQL.

Reiteramos que es sumamente recomendable crear una carpeta por cada ejercicio que desarrollemos, tanto en el servidor local como en el *hosting* que utilicemos (usualmente los ejercicios y proyectos profesionales que realicemos estarán formados por numerosos archivos, y si no los vamos ordenando desde un principio dentro de carpetas, terminaremos mezclando todo).

Con todos los programas instalados, configurados y probados, ¡ya podemos comenzar a programar en PHP!

# Mezclando PHP y HTML

# 3

## El concepto clave: completando las páginas HTML en el acto

Vamos a crear, probar y tratar de entender nuestro primer código PHP.

1. Escribamos el siguiente código dentro de un archivo creado con nuestro editor (si es Dreamweaver, lo haremos en la vista **Código**), asegurándonos de guardarlo con un nombre cuya **extensión** sea **.php**, por ejemplo: **hola.php**. Este será el código de **hola.php**:

```
<!DOCTYPE html>
<html lang="es">
<head>
      <title>Hola</title>
</head>
<body>
<p>Esto estaba escrito en HTML.</p>
<?php
print("<p>Hola mundo! Esto lo escribió el intérprete de
PHP</p>");
?>
</body>
</html>
```

Este archivo deberá guardarse dentro de la **carpeta raíz** del servidor Web que hayamos instalado; por ejemplo, quienes hayan instalado el XAMPP en la ruta que explicamos en el capítulo anterior, podrían crear una subcarpeta dentro de la raíz del servidor Web Apache, y colocarlo en:

C:/servidor/xampp/htdocs/ejercicio1/hola.php

2. Encendemos el software **servidor Web** local (si lo instalamos usando el XAMPP, deberemos ir al menú **Inicio → Todos los Programas → XAMPP for Windows → XAMPP Control Panel**).

El software servidor Web **quedará a la espera** de que le pidamos, a través de un navegador (Explorer, Firefox, Opera), alguno de los archivos que tiene listos para servirnos. En este caso, le pediremos el archivo hola.php.

3. Para ello, abriremos el **navegador** (Explorer, Firefox, Opera, Safari o cualquier otro) y **escribiremos la URL** necesaria para solicitarle al servidor Web el archivo hola.php. Este es un tema clave.

En una PC con un servidor Web local, deberemos escribir en el navegador:

http://localhost/ejercicio1/hola.php o bien usando la dirección IP local: http://127.0.0.1/ejercicio1/hola.php

Atención: **no funcionará ir al menú Archivo → Abrir del navegador**, ya que nuestro archivo PHP necesita ser procesado por el programa intérprete de PHP instalado en el servidor Web, que debe estar encendido.
Para quienes usen Dreamweaver, **tampoco funciona pulsar F12**, por la misma razón.
Ni **tampoco funciona darle doble clic** al archivo PHP.
En todos esos casos erróneos, si miran la URL, no mencionará en ningún lado http://localhost/ejercicio1/hola.php. Por esta razón, el software servidor Web y el intérprete de PHP no procesarán ese archivo.

Una vez solicitada la URL del archivo desde nuestro navegador, y luego de esperar unos segundos, llegará al navegador el código fuente que fue procesado en el servidor. A partir de este momento, en el que el código generado llegó a nuestro **navegador**, podemos acceder al **código fuente que fue entregado** a nuestro navegador (usando el menú: **Ver → Código fuente**) y, en este ejemplo, veremos que le ha llegado lo siguiente:

```
<!DOCTYPE html>
<html lang="es">
<head>
```

```
      <meta charset="utf-8">
      <title>Hola</title>
</head>
<body>
      <p>Esto estaba escrito en HTML.</p>
      <p>Hola mundo! Esto lo escribió el intérprete de
PHP</p>
</body>
</html>
```

Es decir: el intérprete de PHP borró todas las órdenes que encontró entre las marcas **<?php** y **?>**, que son las que indican el **comienzo** y el **fin** –respectivamente– de una zona en la que se va a dar órdenes al intérprete de PHP y, en lugar de lo que había escrito allí, **escribió**, mediante la función **print** (que analizaremos seguidamente), el texto:

```
<p>Hola mundo! Esto lo escribió el intérprete de PHP</p>
```

## Formas de que PHP escriba dentro de HTML

Notemos que es posible la intercalación de órdenes en lenguaje PHP, alternándolas dentro de una página escrita en lenguaje HTML, tantas veces como sea necesario (pueden abrirse y cerrarse los tags de PHP tantas veces como queramos). Por ejemplo:

```
<!DOCTYPE html>
<html lang="es">
<head>
      <meta charset="utf-8">
      <title>Hola</title>
</head>
<body>
      <h1>Esto fue escrito estáticamente, en HTML.</h1>
      <?php
      print("<h2>Hola mundo! Esto lo escribió el intérprete
de PHP</h2>");
      ?>
      <p>Esto ya estaba escrito en el código HTML.</p>
```

```
      <?php
      print("<p>Esto también lo escribió el software
intérprete de PHP</p>");
      ?>
      <p><a href="index.php"><?php print("Volver a la Home
del sitio, escrito por PHP"); ?></a></p>
</body>
</html>
```

### Apertura y cierre de la etiqueta PHP

Notemos que el tag o etiqueta de PHP:

- Puede abrirse y cerrarse en la misma línea en que abrió, o puede cerrarse en otra línea diferente. Es indistinto.
- Puede intercalarse dentro de etiquetas HTML preexistentes.
- Puede generar nuevas etiquetas HTML mediante un echo o print.
- Y puede abrirse y cerrarse muchas veces dentro de una misma página.

Los distintos tipos de tags de apertura y cierre de un bloque escrito en lenguaje PHP que podemos llegar a encontrar, son los siguientes:

1. Apertura y cierre estándar:

```
<?php (aquí van los contenidos) ?>
```

O también:

```
<?php
(aquí van los contenidos)
?>
```

Esta es la única sintaxis universal: funciona siempre. Es la única forma recomendada, y la que vamos a usar, sin importar si abre y cierra en la misma línea o en diferentes.

2. Apertura y cierre corto:

```
<? (aquí van los contenidos) ?>
```

O también:

```
<?
```

```
(aquí van los contenidos)
?>
```

Esta sintaxis se conoce como **short tags** (etiquetas cortas). Fue muy usada en los primeros años de PHP, pero no es estándar. No todas las configuraciones del intérprete de PHP habilitan su uso, por lo que un código que utilice esta sintaxis, puede dejar de funcionar al ser ubicado en un servidor con otra configuración más estricta. Por ese motivo, no la recomendamos.

3. Apertura y cierre mediante etiqueta script:

```
<script language="php"> (aquí van los contenidos)
</script>
```

O también:

```
<script language="php">
(aquí van los contenidos)
</script>
```

Esta sintaxis, si bien todavía se soporta, en PHP7 ya no se soportará, es innecesariamente larga y es rarísimo encontrar algún código que la emplee. Por lo tanto, al no tener ninguna otra ventaja añadida, no recomendamos su uso.

4. Tags estilo ASP:

```
<% (aquí van los contenidos) %>
```

O también:

```
<%
(aquí van los contenidos)
%>
```

Es la sintaxis al estilo del lenguaje de programación ASP de Microsoft: no es estándar, la posibilidad de usarla depende de la configuración del intérprete; y por lo tanto, tampoco se recomienda su utilización.

En los ejemplos anteriores, hemos utilizado la función **print**, que "escribe" en el código fuente de la página que está por enviarse instantes después desde el servidor hacia el navegador del usuario, y que, como vemos, puede escribir no solamente texto, sino también etiquetas HTML como el "<p>" o "<h1>" del ejemplo, etiquetas que luego son interpretadas por el navegador como cualquier otro código HTML que hubiese estado escrito allí originalmente. Veamos cómo trabaja esta función.

### Escribir en el código con la función print()

El lenguaje PHP posee una función que es una de las más utilizadas de todas. Hablamos de la función **print()**, que le indica al software intérprete de PHP que "escriba" –en el código fuente de la página que devolverá al navegador del usuario– aquello que pongamos entre sus paréntesis.

Ya hemos utilizado intuitivamente esta función en los ejemplos anteriores. Si lo que deseamos es que se escriba en el código de la página un texto, literalmente, debemos escribirlo **entre comillas** dentro de sus paréntesis.

Ejemplo:

```
<?php
print("hola");
?>
```

Si solo tuviéramos que escribir textos y nunca código HTML, no tendríamos problemas pero, como debemos encerrar entre comillas el texto a mostrar, se nos planteará un problema a la hora de escribir código HTML que, a su vez, tenga comillas dentro.

En el siguiente ejemplo, veremos por qué:

```
<?php
print("<h1 class="portada">Bienvenidos</h1>");
?>
```

Este ejemplo generará un error, pues la comilla ubicada luego del signo = está cumpliendo, sin querer, la función de **cerrar** la primera de las comillas –la que se abrió al inicio del **print**, luego del paréntesis inicial– y, por lo tanto, el tramo de texto se da por concluido, y al resto de texto que sigue a esa comilla el software intérprete de PHP no sabe cómo tratarlo, y lo advierte mostrando un mensaje de error en la pantalla que dirá: *"Unexpected 'portada'…"*.

Una posible solución al problema de las comillas es desactivar (a esto se lo denomina "escapar") todas las comillas dobles intermedias, una por una, para que no den por concluida la cadena de texto antes de que lleguemos a la última comilla doble que indica el término de la función **print**.

El carácter de escape es la barra invertida **** y sirve para **no ejecutar** el carácter que le sigue inmediatamente como si fuera parte de una orden del lenguaje PHP, sino que lo considera como una letra más que debe ser escrita literalmente.

Por esta razón, el ejemplo anterior podría quedar así:

```
<?php
```

```
print("<h1 class=\"portada\">Bienvenidos</h1>");
?>
```

Esto funciona muy bien en frases cortas, pero el mayor inconveniente o molestia que nos puede causar surge cuando tenemos que imprimir largos bloques de código HTML, ya que es muy probable que esos bloques (tal vez páginas enteras) ya los tengamos escritos de antes, generados por nuestro editor de código HTML, y es casi seguro que poseerán numerosas comillas dobles. En esos casos, estaríamos obligados a la tediosa tarea de encontrar las comillas una por una, y "escaparlas" anteponiéndoles una barra invertida o, en su defecto, podríamos utilizar las herramientas de búsqueda y reemplazo de caracteres de algunos editores HTML para buscar una comilla y reemplazarla por la barra de escape más la comilla. Pero, en ambos casos, sería una larga y aburrida tarea.

Mucho mejor que esto, sería utilizar **comillas simples** para delimitar el inicio y final del bloque de texto a imprimir:

```
<?php
print('<h1 class="portada">Bienvenidos</h1>');
?>
```

¡Y problema solucionado!

### Cómo funciona el comando "echo"

Este comando (no es una función) también puede utilizar optativamente comillas simples o dobles para delimitar lo que va a imprimir, de la misma manera que print. Pero, a diferencia de print, no es habitual envolver entre paréntesis lo que escribirá. Ejemplo:

```
<?php
echo "Hola Mundo entre comillas dobles!";
echo '<html>
      <head>
         <title>Envuelvo entre comillas simples</title>
      </head>
      <body>"Esto tiene comillas dobles, "muchas comillas",
      y no importa"
      </body>
</html>';
?>
```

Notemos de paso que el código que escribirá puede estar dividido en **múltiples líneas** (PHP ignora tanto los saltos de línea como los espacios en blanco), y también señalemos otro detalle al que todavía no habíamos prestado atención: para dar por terminada una sentencia u orden, se agrega un **punto y coma** al final de la línea.

### Grandes bloques: heredoc

Cuando tenemos necesidad de escribir largos bloques de código HTML, incluso con variables intercaladas (que es muy común, según veremos pronto), podemos usar la construcción **heredoc**, que nos permite escribir grandes cantidades de texto, sin necesidad de escapar caracteres en su interior.

Su uso es muy simple. Al inicio del bloque de texto, debemos colocar tres veces el signo "menor que", de esta manera: **<<<** seguido de varios caracteres alfanuméricos (en el ejemplo que sigue, hemos elegido **EOT**, pero pudo ser cualquier otra combinación de letras); luego, pegamos el bloque de texto y código HTML que escribiremos y, para finalizar, repetimos los mismos tres caracteres que indicaron el inicio del bloque:

```
<?php
echo <<<EOT
<p>Este texto puede tener dentro "comillas" sin necesidad
de escaparlas.</p>
<p>También procesa (reemplaza por su valor) las $variables
que hubiera dentro del código, tema que veremos
próximamente.</p>
<p>Esta construcción del lenguaje llamada heredoc es ideal
para incluir largos bloques de código HTML.</p>
EOT;
?>
```

También, podemos almacenarlo dentro de una variable:

```
<?php
$codigo = <<<EOT
<p>Este texto puede tener dentro "comillas" sin necesidad
de escaparlas.</p>
```

```
<p>También procesa (reemplaza por su valor) las $variables
que hubiera dentro del código, tema que veremos
próximamente.</ p>
EOT;
echo $codigo;
?>
```

Si bien los caracteres identificadores pueden formarse con cualquier combinación alfanumérica, por convención, se suele utilizar los caracteres EOT (*end of text* o final de texto). Lo importante es que esos mismos caracteres no estén incluidos (no aparezcan) dentro del texto, ya que de suceder eso, el software intérprete de PHP considerará que allí termina el bloque, y provocará un mensaje de error al no saber qué hacer con el resto de texto.

Otra consideración es que estos caracteres indicadores del inicio y fin del bloque, deben incluirse justo al comienzo del renglón (sin dejar ni un solo espacio en blanco, ni tabuladores, ni indentados de código, ni comentarios), y tampoco deben contener ningún otro carácter al final, salvo el punto y coma que finaliza el echo (y, luego del punto y coma, debe haber un salto de línea).

Por ese motivo, si escribimos este código en un editor de texto bajo Windows, podemos tener problemas al incluir el salto de línea (Enter) al final del renglón, ya que el salto de línea en Windows no es el carácter de nueva línea **\n** que esta construcción espera encontrar, sino que es **\r\n**.

Podemos hacer la prueba y, si vemos que al terminar la orden no se interrumpe la escritura del texto, deberíamos verificar que el renglón final esté completamente vacío de cualquier carácter extraño. Una vez confirmado que nuestro editor esté generando el salto de forma correcta, conviene que subamos al servidor por FTP el archivo empleando el modo *Ascii* (y no el binario), y desactivar la opción de "autodetectar" que la mayoría de programas de FTP contienen.

Además de la facilidad de lectura y pegado en su lugar de los largos bloques de código, esta sintaxis incrementa la velocidad de interpretación del código en el servidor si la comparamos con el uso de varios **echo** o **print** seguidos.

## Copiando y pegando archivos enteros

### Los rompecabezas de include y require

Es normal que varias páginas de un sitio o aplicación Web contengan **elementos en común**, que se repiten una y otra vez a lo largo de decenas de páginas. Por

ejemplo, un mismo encabezado de página, un mismo menú de navegación, o un mismo pie de página.

En esos casos, no sería nada práctico que el código HTML de esos bloques se repitiera idénticamente en cada uno de los numerosos archivos HTML del sitio, ya que cuando llegue el momento de modificar el contenido de alguno de estos elementos (agregar un botón nuevo en el menú, o un simple cambio de teléfono en un pie de página), será necesario hacer el cambio en cada uno de los archivos que muestran ese bloque, y luego estaremos obligados a subir por FTP al servidor quizás decenas, o cientos, o miles de archivos. Para solucionar este problema, PHP posee cuatro "construcciones" (no son funciones) denominadas:

- include,
- require,
- include_once y
- require_once.

La idea al utilizarlas es que colocaremos, en un **archivo aparte**, los contenidos que tengan en común muchas páginas.

Por ejemplo, ubicaremos, en un archivo denominado **menu.php**, el código fuente necesario para que se vea el menú, el código del pie de página lo situaremos en otro archivo aparte llamado **pie.php**, y así sucesivamente con todos los elementos comunes a varias páginas.

Luego, en cada página del sitio donde necesitemos mostrar ese menú o ese pie, le ordenaremos al software intérprete de PHP que incluya el código completo del archivo en cuestión en el lugar exacto en el que lo especifiquemos.

Esta orden realiza automáticamente una tarea similar a la que haríamos manualmente si seleccionáramos el código de esos archivos externos (menu.php, pie.php, etc.), los copiáramos y los pegáramos luego en cada una de las páginas en las que queremos mostrar ese pie o ese menú.

De esta manera, cuando fuera necesario realizar un cambio a alguno de esos archivos, lo haremos en el archivo que contiene exclusivamente el código del menú, o el del pie de página, y terminados los cambios de contenido de esos archivos únicos, los subiremos por FTP, y no tendremos que cambiar absolutamente nada en las otras decenas, o cientos, o miles de páginas que conforman el sitio, que reflejarán los cambios de inmediato, ya que en ellas solo dice "mostrar aquí lo que haya en menu.php" (de una forma más técnica, pero la idea es esa).

Veamos un ejemplo, el archivo **pagina.php**:

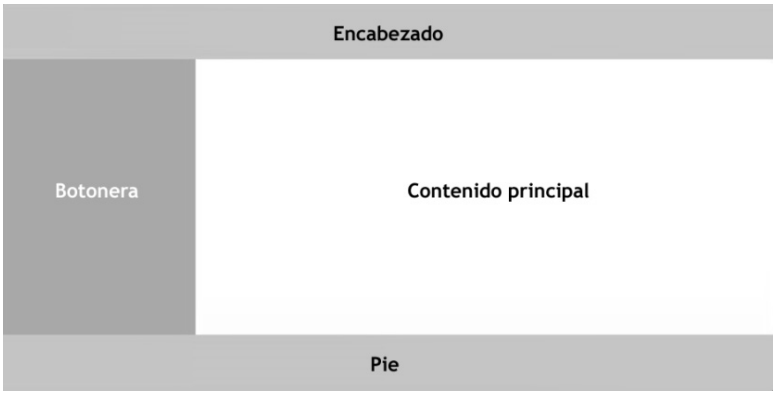


**Figura 3-1. Ejemplo de archivo que integra otros archivos incluyéndolos en él.**

El código de este archivo **pagina.php** –que simula ser una de las tantas páginas estándar del sitio– quedaría de la siguiente manera:

```
<!DOCTYPE html>
<html lang="es">
<head>
      <meta charset="utf-8">
      <title>Página que incluye otros archivos</title>
</head>
<body>
      <div id="contenedor">
         <?php include("encabezado.php"); ?>
         <div id="contenidoPrincipal"> Contenido principal
         </div>
         <?php include("botonera.php"); ?>
         <?php include("pie.php"); ?>
      </div><!-- cierre del contenedor -->
</body>
</html>
```

Podemos descargar el código completo de estos archivos en el sitio Web de este libro, buscándolos bajo el nombre de este capítulo. Allí, se incluye el código CSS

necesario para que la página se visualice idéntica al ejemplo, que era demasiado largo y no específico del tema PHP como para que lo trascribiéramos aquí.

Y, además, tendríamos un archivo por cada bloque de página que deseamos independizar:

El archivo **encabezado.php**:

```
<div id="encabezado">
      Encabezado
</div>
```

El archivo **botonera.php**:

```
<div id="botonera">
      Botonera
</div>
```

El archivo **pie.php**:

```
<div id="pie">
      Pie
</div>
```

Queda claro entonces que include nos brinda un enorme ahorro de tiempo a la hora de realizar tareas de mantenimiento de un sitio de muchas páginas.

Más adelante, también utilizaremos include para colocar en archivos externos declaraciones de variables, funciones, clases y de cualquier otra información que sea necesario mantenerla disponible a lo largo de muchas páginas, pero sin repetirla, declarándola y almacenándola una sola vez en un archivo externo que será incluido por todas aquellas páginas que necesiten la información que estos archivos externos contienen.

### Diferencias entre include, require, include_once y require_once

Las diferencias entre **include** y **require** son mínimas, simplemente se diferencian por el tipo de error que generan si fracasan en su intento de incluir un archivo (por ejemplo, si ese archivo no existe porque lo borramos o renombramos). Un include crea, en ese caso, un *Warning*; esto es: envía una advertencia por pantalla, pero no interrumpe la ejecución del resto del archivo.

En cambio, cuando falla la orden require genera un **Fatal error**, que interrumpe definitivamente en ese punto la ejecución del archivo que estaba haciendo el intento de inclusión.

Por lo tanto, en casos en los que sea absolutamente imprescindible contar con los datos que estaban guardados en el archivo externo que se iba a incluir, es mejor usar require (un caso típico son los archivos externos que contienen usuario y *password* de una base de datos, ya que si no accedemos a esos datos clave, ya no tiene sentido continuar la ejecución de las órdenes que vengan a posteriori; en ese caso, sería aconsejable utilizar **require**).

¿Y cuál es la diferencia entre **require** y **require_once**, o entre **include** y **include_once**? (nota: *once* en inglés quiere decir "una sola vez").

Con require_once o include_once, si el archivo ya ha sido incluido en la misma página con anterioridad, no se volverá a insertar por segunda vez. Esto puede ser útil en los casos en los que un mismo archivo pudiera ser potencialmente incluido más de una vez durante la ejecución de un código, debido a la complejidad del código, y se quiera estar seguro de que se inserta una sola vez, para evitar problemas con redefiniciones de funciones, valores de variables, etc.

A continuación, un ejemplo que utiliza require, include_once y require_once:

```
<!DOCTYPE html>
<html lang="es">
<head>
      <meta charset="utf-8">
      <title>Página que incluye otros archivos</title>
</head>
<body>
      <div id="contenedor">
         <?php require("encabezado.php"); ?>
         <div id="contenidoPrincipal">
            Contenido principal
         </div>
         <?php include_once("botonera.php"); ?>
         <?php require_once("pie.php"); ?>
      </div><!-- cierre del contenedor -->
</body>
</html>
```

## Los comentarios

### *Dejando anotaciones dentro del código*

Realizar comentarios dentro del código PHP es una muy simple pero muy eficaz costumbre, que consiste en ir escribiendo dentro de cada bloque de código aclaraciones que nos sean de utilidad cuando, varios días, semanas o meses después, volvamos a leer el código para hacerle alguna modificación. Recordemos que estos comentarios no llegan al código fuente del navegador, por lo que los usuarios de nuestros sitios no los pueden ver, ya que son para nuestro uso personal y el de otros programadores que tengan acceso a nuestro código.

El objetivo de envolver algo entre comentarios es lograr que el software intérprete de PHP no intente ejecutar las palabras que escribamos, sino que las ignore, y prosiga luego de finalizado el comentario.

PHP permite escribir comentarios de una o varias líneas:

```
<?php
$variable = "valor"; //comentario de una línea
$cifra = 123; #otro comentario de una línea
echo $variable; /* comentario multi
línea
*/
?>
```

Un uso práctico es aclarar cosas que sea útil recordar más adelante, al revisar este código:

```
<?php
$clave = "xyz345"; // avisar al Gerente si la cambiamos
$tolerancia = 3; # cantidad de reintentos
echo $clave; /* recordar que si mostramos esto,
previamente deberíamos informar al usuario,
para que no lo exponga a la vista de otros.
*/
?>
```

Otro uso muy práctico de los comentarios es “desactivar” renglones de código que queremos que no se ejecuten, pero que no tenemos la intención de borrarlos, sino que solo queremos ejecutar el archivo sin que se ejecuten las líneas del código comentadas. Comentamos partes del código, y es como si no

existieran. Esto nos ayuda en la detección de errores, si comentamos alguna línea cercana a la zona en la que está produciéndose un error, para evitar su ejecución y ayudarnos a detectar qué es lo que lo está produciendo, tema que practicaremos más adelante.

Como acabamos de ver, uno de los principales usos de PHP es ayudarnos a completar ciertas partes de nuestras páginas Web, escribiendo textos y código HTML en el acto, unos breves instantes antes de que el servidor Web envíe ese código "terminado" hacia el navegador del usuario.

Esta tarea de escritura –que va desde el agregado de unos pocos caracteres hasta la inclusión de archivos HTML enteros– aunque puede parecer muy simple, es de una enorme potencia que ya iremos comprendiendo a lo largo de este libro, y resulta imprescindible para crear páginas Web "dinámicas", con contenidos que, según la ocasión, pueden ser muy distintos.

El punto clave es que esos textos, esa información que será escrita dentro del código HTML, se obtendrá automáticamente, de muy distintas fuentes, tales como variables, constantes, matrices, funciones, archivos de texto y bases de datos.

En el siguiente capítulo, comenzaremos a familiarizarnos con algunos de estos lugares en los que se almacena la información.

# Los almacenes de datos

# 4

## Contenedores temporales y permanentes, de pocos y de muchos datos

En los capítulos precedentes, le indicamos letra por letra al intérprete de PHP los contenidos que debía escribir dentro de cada orden *print()* o *echo*. Le proporcionamos, además, cada uno de los datos con los que trabajó y le dictamos todo lo que escribió.

Pero, paradójicamente, la utilidad de PHP radica en entregarle a un navegador Web (Chrome, Firefox, Safari, Internet Explorer, Opera, o el que fuere) **diferentes** contenidos, según se vayan desencadenando determinados sucesos previstos por el programador de la página (de acuerdo con la contraseña introducida, el enlace pulsado, un dato que se valida, etc.).

Este comportamiento se denomina **páginas Web dinámicas**; es decir: el navegador, luego de solicitar una URL, recibirá como respuesta una página Web que no fue escrita en su totalidad por el diseñador o programador con su programa editor, sino que su código –todo, o al menos una parte– lo escribirá el intérprete de PHP **cada vez** que un usuario pida ver esa página con su navegador; por lo tanto, la página se construirá "en vivo" en el servidor, unos instantes antes de que la envíe hacia el navegador.

Esto nos lleva a una pregunta obvia: si no somos nosotros (diseñadores, programadores) quienes le dictaremos letra por letra al intérprete de PHP esos textos, ¿de dónde obtendrá esos datos? La respuesta es "de varios lugares posibles".

De acuerdo con cada una de nuestras necesidades específicas, dispondremos de distintas alternativas para almacenar datos en lugares a los que accederemos desde el servidor, donde el intérprete de PHP podrá leerlos por sí mismo y los utilizará para su tarea de fabricar una página HTML. Mencionaremos, a continuación, algunos de esos posibles lugares –los principales–, para que nos vayamos imaginando los recursos con los que trabajaremos:

- Los datos se podrán almacenar en una **variable o en una matriz** (creada por nosotros, o una de las tantas matrices en las que el intérprete de PHP almacena información automáticamente).
- El usuario podrá ingresar los datos y, consciente o inconscientemente, usará su navegador para enviar las variables hacia el servidor, ya sea a través de un **formulario** o mediante las **variables adjuntadas a una URL**, con un tipo de enlace especial. Un envío de "señales" hacia el servidor que nuestra página PHP estará esperando para decidir qué información mostrar.
- Los datos también se podrán obtener como resultado de ejecutar una **función** (de las que vienen incluidas en el lenguaje PHP, o de las funciones que crearemos nosotros mismos).
- Un dato se podrá almacenar en una *cookie* que el navegador del usuario guardará silenciosamente.
- Se podrá leer un dato de una variable de **sesión**, lo que nos permitirá identificar a un usuario en particular en un momento dado.
- Se podrán leer los datos escritos dentro de un **archivo de texto** (txt, XML, etc.) existente en el servidor.
- Se podrán leer los datos almacenados en una **base de datos** (sin dudas, la opción más poderosa para manejar grandes cantidades de datos, como catálogos de productos, mensajes de foros, etc.).

| | TEMPORALES | PERMANENTES (O CASI) |
|---|---|---|
| **Pocos datos:** | Variables (locales, de formularios, URLs), constantes, funciones. | Archivos de texto, *cookies*, sesiones |
| **Muchos datos:** | Matrices | Bases de datos |

**Tabla 4-1. Orígenes de datos.**

Todo el resto de este libro consistirá en aprender a leer y a escribir datos en estos lugares.

Comenzaremos aprendiendo a trabajar con el almacén de datos más simple y más universalmente utilizado: las **variables**.

# Las variables: pocos datos, provisorios

## Las ventajas de declarar y usar variables

Aunque a quienes todavía no son programadores les cueste un poco imaginar la verdadera utilidad de algo tan simple como una variable, continuamente tendremos la necesidad de que el servidor **recuerde** por un momento algún dato, ya sea porque se lo hayamos escrito nosotros mismos dentro del código que programamos, o porque el usuario haya ingresado ese dato en un formulario, o porque sea fruto de una operación realizada por el intérprete de PHP o que sea resultado de la ejecución de una función. Todo el tiempo usaremos variables. Y no es una exageración ya que, prácticamente, no realizaremos ninguna página que no utilice numerosas variables.

Por ejemplo, si queremos mostrar un saludo que utilice el nombre que un usuario acaba de escribir en un formulario, habrá que almacenar en algún lugar ese dato mientras maniobramos con él y le agregamos un texto que lo acompañe. O si queremos multiplicar dos números (una cantidad y un precio de un producto, por ejemplo), deberíamos poder almacenar en algún lado el primer número, el segundo número, y también el resultado de esa operación, para luego mostrarlo. O, si estamos reutilizando una página de **Términos y condiciones de uso** y, entre el sitio original y el nuevo, solo cambia el nombre de la empresa, podríamos haber almacenado el nombre de la empresa en una variable y, simplemente escribiendo el nombre nuevo por única vez en el valor de esa variable, en todos los lugares donde la usábamos aparecerá automáticamente el nombre de la nueva empresa.

Es decir, el uso de las variables será imprescindible para la tarea más común del intérprete de PHP, que es completar parte de los textos o del código HTML de una página Web.

Para todas las tareas que mencionamos –que requieren "recordar momentáneamente un dato"– existen las variables, que no son más que un **espacio en la memoria RAM** del servidor (un servidor es una computadora y tiene memoria RAM como cualquier computadora personal), espacio que será temporalmente reservado para que le escribamos algo adentro, y lo dejemos allí guardado, esperando hasta que lo utilicemos instantes después, como parte del código que el intérprete de PHP está fabricando en ese momento.

Cada uno de los espacios que queramos utilizar de la memoria del servidor, será una variable. Y cada una de ellas **se identificará con un nombre** que

nosotros mismos le inventaremos, nombre mediante el cual nos referiremos a ese dato de allí en más, cada vez que necesitemos disponer del contenido almacenado en ese espacio.

De una forma más gráfica y simplificada, una variable es una especie de «caja etiquetada» que almacena algún dato hasta que necesitemos usarlo.
Usamos variables a diario; por ejemplo, guardamos dinero en un cajón y luego decimos: *"usemos el dinero que hay en el primer cajón".* "Primer cajón" sería el nombre de una variable, que a veces contendrá un valor de «0» (si no hay dinero), y a veces contendrá una cantidad "X" de dinero; es decir, su contenido no será siempre el mismo, a lo largo del tiempo podrá variar, de allí su nombre "variable". Cualquier receta de cocina está llena de variables (la cantidad de cada ingrediente, es una variable).
¡Incluso los seres humanos estamos llenos de variables! Cuando en un formulario nos dicen "Escriba aquí su nombre", *"su nombre"* es una variable, ese casillero contendrá "Hernán", en mi caso, pero cualquier otro nombre en el caso de que alguien no sea tocayo mío. Todo lo que puede describirse cuantitativa o cualitativamente es un dato que puede almacenarse en una variable: el peso, la altura, la edad, el apellido, la nacionalidad, la ocupación, el teléfono, etc.
Variables y más variables. Textos y números: datos.

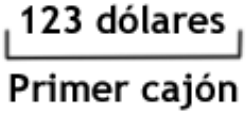


**Figura 4-1. Una variable, con su nombre y su valor.**

Una variable siempre tiene dos elementos: un **nombre** (siempre el mismo, así como "Primer cajón" será siempre "Primer cajón", y gracias a esa invariabilidad podemos referirnos a él sin lugar a confusiones como "Primer cajón"), y un **valor** (el dato que almacenamos dentro de ella) que a diferencia del nombre, sí puede variar:

```
<?php
$nombre = "VALOR";
?>
```

Podemos observar, en este ejemplo, algunos detalles de sintaxis: en el lenguaje PHP las variables se crean anteponiéndole un signo $ al nombre que le

queramos dar a la variable (sin dejar ningún espacio entre el signo y la primera letra del nombre de la variable).

Y se les almacena un valor mediante el signo **igual**, al que técnicamente lo denominaremos **operador de asignación**, ya que, precisamente, sirve para asignarle un valor a una variable.

Veamos otros ejemplos para seguir comprendiendo esta sencilla sintaxis:

```
<?php

$cantidad = 5;
$precio = 3;
$importe = $cantidad * $precio;
print($importe);

?>
```

Esta vez hemos definido tres variables:

- la variable **$cantidad**, a la que le hemos asignado un número 5;
- la variable **$precio**, en la que guardamos un número 3;
- y la variable **$importe**, que almacena el resultado de multiplicar lo que contenía **$cantidad** por lo que contenía **$precio**; es decir, que contendrá un **15** en este caso.

Imaginemos que, en ese momento, la memoria RAM del servidor contiene esto:

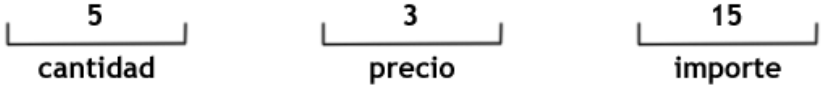


**Figura 4-2. Variables en la memoria del servidor.**

Prestemos atención a los siguientes detalles:

1. Cada sentencia (también la denominamos informalmente una "orden"–vamos a encontrar que muchos autores utilizan estos nombres indistintamente–), se debe terminar con un **punto y coma**.

2. Notemos que a los **valores** de las variables con contenido numérico –en este caso, los números **5** y **3**–, no los hemos puesto entre comillas; el lenguaje PHP distingue entre varios tipos de datos (que veremos pronto) entre los cuales **los números no necesitan estar envueltos entre comillas**, a diferencia de los bloques de texto, que sí precisan estar siempre delimitados en su inicio y en su fin con comillas.

3. Dentro del print(), tampoco hemos puesto entre comillas la variable **$importe**. Esto es así porque si la envolviéramos entre comillas **simples**, considerará que debe escribir literalmente el texto **'$importe'**, y no lo que la variable **$importe** tenía almacenado (en este caso, un **15**). Sin embargo, tendríamos también la posibilidad de envolverla entre comillas **dobles** y, de esa manera, la variable sí se reemplazaría normalmente por su valor, veamos otro ejemplo:

```
<?php
$nombre = "Pepe";
$mensaje = "Hola, señor $nombre";
print($mensaje);
?>
```

En breve, analizaremos con más detenimiento la diferencia de comportamiento entre una variable envuelta entre comillas simples y otra entre comillas dobles.

Como hemos mencionado ya desde el título, las variables son un depósito **provisorio** de datos, "efímero", que se utilizará casi inmediatamente.

Recordemos que todo el trabajo del programa intérprete de PHP tiene lugar en el servidor (hosting), y se realiza durante los breves instantes que median entre la llegada de una petición HTTP ejecutada por un navegador (un pedido de ver una página web) y la emisión de una respuesta por parte del servidor web (el envío del código HTML que el navegador recibirá y procesará, mostrando algo en pantalla al usuario).

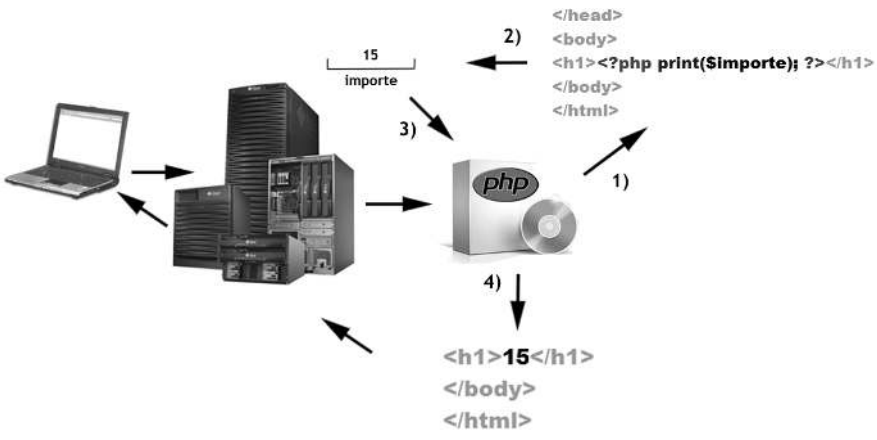


**Figura 4-3. Circuito del reemplazo de las variables por su valor.**

La vida de una variable en PHP es muy breve: dura apenas unas pocas décimas de segundo. En el momento en que el intérprete de PHP lee la declaración de una variable, ésta comienza a existir; acto seguido, la podrá utilizar para alguna cuestión (podría ser una orden escrita que le hayamos dejado al intérprete de PHP para que utilice o muestre esa variable, para que realice alguna operación con ella) y, en cuanto el intérprete de PHP haya terminado la lectura y proceso del código PHP de esa página Web completa (menos de un segundo, habitualmente) y haya entregado el código HTML ya procesado al servidor Web para que lo envíe al navegador que estaba esperando su respuesta, en ese instante, el intérprete de PHP **borrará** de la memoria RAM del servidor Web todas las variables que hubieran sido declaradas durante el breve tiempo que demoró el procesamiento de esa página.

Muy pronto comprenderemos la simplicidad pero, a la vez, la enorme fuerza de este concepto, que deja depositado un dato provisoriamente en la memoria de un servidor.

## Eligiendo una convención para darles nombre

Un muy simple pero importantísimo detalle que nos facilitará la lectura de un código PHP, es seguir alguna regla relativa a los **nombres** que le elegiremos a todos los lugares de almacenamiento de datos que utilicemos; entre ellos, las variables. El objetivo es intentar que el nombre de una variable simbolice lo mejor que sea posible el **contenido** que vaya a almacenar. Pensemos que un código se escribe –de la misma manera que cualquier texto–, para que el lector lo

comprenda; por esta razón, todo lo que contribuya a facilitar la lectura y mejore su rápida comprensión, nos ahorrará mucho tiempo.

Lamentablemente, suele ser muy común que encontremos códigos con nombres de variables escritos con abreviaturas extrañas como, por ejemplo, **$var_67_x_1**. Dentro de unos meses, cuando debamos leer ese mismo código, nos habremos olvidado los datos que contenían cada una de las variables.

Veamos un ejemplo de código con nombres muy difíciles (la versión de este mismo código con nombres claros es la que hemos utilizado en los párrafos anteriores):

```
<?php
$dato_123 = "Pepe";
$dato_46 = "Hola, señor $dato_123";
print($dato_46);
?>
```

Imagínense esto proyectado a cientos o miles de líneas de código (algo que será común en nuestros proyectos). ¡Imposible de entender a simple vista!

Además de los nombres descriptivos, otra recomendación es que no conviene mezclar nombres de variables en **mayúsculas** y en **minúsculas**. Lo mejor es seguir algún criterio, alguna convención, ya que es ésta una de las causas más frecuentes de errores que nos hacen perder tiempo al tratar de descubrir la causa.

La razón es que PHP es un lenguaje *case-sensitive*, o "sensible" a mayúsculas y a minúsculas; es decir, no es lo mismo **$a** –en minúsculas– que **$A** –en mayúsculas–: para PHP, son dos variables distintas. No es la misma variable una que se llame **$nombre** que otra llamada **$Nombre** o que **$nomBRE**. Y si habíamos definido **$Nombre** y después hacemos un *echo* de **$nombre**... ¡no funcionará! Y con justa razón, ya que no existe la variable que intentamos leer, existe otra con una letra de diferencia, y eso la vuelve totalmente distinta.

Por eso, es sumamente recomendable que sigamos alguna regla para nombrar nuestras variables, y que la mantengamos a lo largo de todos nuestros proyectos.

Por ejemplo, algunas **convenciones comunes** relativas a los nombres de las variables, son las siguientes:

1. Todo en minúsculas:

   **$variable**

   **$mivariablelarga**

2. Todo en minúsculas, pero con guiones bajos entre palabras, si el nombre de una variable tiene varios términos:

   **$variable**

   **$mi_variable_larga**

3. En minúsculas, la primera palabra –o si la variable es de una sola palabra–, pero colocando una mayúscula al comienzo de cada nueva palabra intermedia, en el caso de que el nombre de la variable tuviera más de un término. Esta notación se llama *camel case* –porque las subidas y bajadas de las mayúsculas intercaladas entre medio ¡parecen las jorobas de un camello!. Un nombre muy imaginativo... Esta es la sintaxis más extendida, y es la más recomendable:

   **$variable**

   **$miVariableLarga**

4. Cada palabra comienza siempre con mayúscula, aunque la variable sea de una sola palabra:

   **$Variable**

   **$MiVariableLarga**

No se recomienda poner a las variables nombres en mayúsculas; por ejemplo, **$NOMBRE**, ya que da lugar a confusión con otro tipo de depósito de datos llamado constante, cuyos nombres en PHP se escriben con todas sus letras en mayúsculas (aunque sin el signo $ delante, pero igual da lugar a confundirlas, volviendo poco clara la lectura del código).

Más allá de cuál de todas estas convenciones elijamos, es recomendable elegir una sola de estas notaciones –cualquiera de las anteriores– y seguir siempre la misma, eso nos evitará muchísimos errores.

Cuando tengamos códigos de cientos de líneas (es decir, muy pronto, ¡antes de lo que imaginamos!) será muy importante que podamos leer ese código con la velocidad con que leemos un texto cualquiera, y que sea posible entender ese código con una sola lectura superficial; por eso, es preferible que nos acostumbremos desde un principio a utilizar nombres que sigan alguna regla de sintaxis que nos resulte cómoda.

Mencionemos, por último, que en el lenguaje PHP no podemos comenzar el nombre de una variable con un número: las variables deben comenzar por una **letra** o por un **guion bajo**, aunque luego sí pueden incluir números en su nombre, letras de la “a” a la “z” y guiones bajos.

No utilizaremos eñes, ni acentos, ni espacios en blanco, ni guion medio, ni caracteres especiales como asteriscos o cualquier otro signo en los nombres de nuestras variables.

Veamos algunos ejemplos de nombres de variables, correctos e incorrectos:

```
<?php
$lugar1 = "Buenos Aires"; /* Correcto, puede comenzar por
una letra */
$_lugar_2 = "México"; /* Correcto, puede comenzar por un
guion bajo */
$3_lugar = "España"; /* Incorrecto, no puede comenzar por
un número */
?>
```

## El problema de las comillas

### Cuándo usar comillas simples y cuándo comillas dobles

Releamos el código anterior a este título. Allí, se realizan tres operaciones de asignación de valor a tres variables distintas. Si miramos a la derecha del signo =, veremos que el valor de esas variables está envuelto entre comillas. Podemos deducir si observamos los distintos ejemplos anteriores, que los valores **alfanuméricos** deben envolverse entre comillas, ya sean simples o dobles; y que, en cambio, los valores **numéricos** no deben envolverse entre comillas.

Para almacenar un dato **alfanumérico** dentro de una variable, deberemos comprender cómo son afectados esos datos al ser delimitado su inicio y su fin por los dos tipos de comillas existentes: comillas **dobles** y **simples**.

Pero veamos cómo incide esto cuando a esos textos les agregamos variables.

### La interpretación de variables dentro de comillas

Ya hemos visto cómo se comportan el comando echo y la función print (ambos son iguales en esto):

- Cuando deseamos que el intérprete de PHP escriba un **texto literalmente** usaremos **comillas simples** para delimitar el inicio y el final de ese texto. La única limitación es que no podremos incluir comillas simples dentro de ese texto (deberemos escaparlas), y cualquier variable que incluyamos dentro de algo envuelto entre comillas simples, **no será reemplazada** por su valor.

- En cambio, cuando queremos que se interpreten y **reemplacen las variables por su valor**, usaremos **comillas dobles** para delimitar el inicio y el final del bloque de texto.

Veamos un ejemplo:

```
<?php
$comillasDobles = "Texto entre comillas dobles, puede
contener 'comillas simples' dentro sin problemas";

$comillasSimples = 'Texto entre comillas simples, puede
contener "comillas dobles" pero sin variables dentro,
porque usamos comillas simples para delimitar el inicio y
fin del bloque';

$escapeDoble = "Texto con \"comillas\" dobles escapadas";

$escapeSimple = 'Texto con \'comillas\' simples
escapadas';

$variablesDobles = "Texto con variables como $nombre y
$apellido intercaladas entre comillas dobles, que se
reemplazarán por su valor";

$variablesSimples = 'Texto con variables como $nombre
y $apellido intercaladas entre comillas simples, que no se
reemplazarán por su valor, quedará escrito $nombre y
$apellido tal cual';
?>
```

## Concatenación

En el último ejemplo anterior a este título ($variablesSimples), nos hemos encontrado con un problema muy común: tener un texto envuelto entre comillas simples, y necesitar que las variables incluidas dentro se reemplacen por su valor.

Otras veces, al dictarle al intérprete de PHP largos bloques de código HTML (con abundantes comillas dobles incluidas), por comodidad elegiremos delimitar

el inicio y el final entre comillas simples, y eso se convertirá en un obstáculo si queremos incluir datos provenientes de variables entre medio de ese texto.

Veamos por qué:

```
<?php
$sitio = 'PHP';
$concatenacion = '<h1 class="resaltado">Bienvenidos a
$sitio</h1>';
?>
```

Este ejemplo no producirá el resultado deseado, sino esto:

```
<h1 class="resaltado">Bienvenidos a $sitio</h1>
```

Para solucionarlo y lograr que esa variable se reemplace por su valor, usaremos una técnica denominada **concatenación**. Concatenar es unir, es "pegar" elementos que estaban separados. Esos elementos serán, en nuestro caso, los textos, las variables y las constantes que necesitemos procesar. Nos basaremos en la idea sencilla de "sacar afuera" de las comillas las variables y constantes que queramos reemplazar por su valor. Es decir, **interrumpiremos** momentáneamente el bloque de texto delimitado por comillas y, luego de la variable, lo **reiniciaremos**.

Para esta tarea, es decir, para finalizar y recomenzar tramos de texto, usaremos el **operador de concatenación**, que no es más que un simple punto: "**.**"

Ese punto será el "pegamento" que unirá todo aquello que nos convenga mantener a salvo de las comillas del bloque, pero manteniéndolo unido a él.

Veamos algunos ejemplos para entenderlo mejor:

1. Concatenar un texto y una variable:

```
<?php
$nombre = 'Pepe';
$concatenacion = '<p id="saludo">Hola '.$nombre.'</p>';
?>
```

Visualicemos mejor los tres "tramos" que componen esta unión de partes, mirando cada parte por separado:

1. '<p id="saludo">Hola'
2. $nombre

3. '</p>';

Son esos tres tramos los que estamos pidiendo que sean escritos (o mejor dicho, en este caso, que sean almacenados dentro de la variable llamada $concatenacion). Primero, se escribirá la apertura de la etiqueta de párrafo con su identificador que usa comillas dobles para envolver su valor, luego se une esto al valor que tenga la variable $nombre (Pepe, en este caso) y, finalmente, se cierra la etiqueta de párrafo.

Ese código producirá este resultado:

```
<p id="saludo">Hola Pepe</p>
```

que era lo que deseábamos lograr.

2. Concatenar una variable y otra variable:

```
<?php
$nombre = 'Juan';
$apellido = 'Pérez';
$concatenacion = '<p>Su nombre y apellido es
'.$nombre.$apellido.'</p>';
?>
```

Vemos que pueden unirse no solamente un texto a una variable, sino también dos variables entre sí, aunque esto generó un pequeño detalle visual: el nombre y el apellido quedarán “pegados” entre sí, ya que no hemos incluido ningún espacio entre ambas variables:

```
p>Su nombre y apellido es JuanPérez</p>
```

Podemos crear un tramo de texto concatenado, exclusivamente para insertar ese espacio faltante:

```
<?php
$nombre = 'Juan';
$apellido = 'Pérez';
$concatenacion = '<p>Su nombre y apellido es '.$nombre.
''.$apellido.'</p>';
?>
```

Notemos que el primer tramo de texto va desde la apertura de la etiqueta de párrafo hasta el final de la palabra "es"; el segundo tramo es la variable $nombre; luego viene un tramo de texto entrecomillado que es solo para escribir un espacio; a continuación, sigue la variable $apellido y, por último, el cierre del párrafo.

Antes de dar por terminada esta introducción al uso de variables y su combinación con otros textos, mencionemos que una misma variable, en distintos momentos, puede ir cambiando su valor almacenado, tantas veces como sea necesario.

Por ejemplo:

```
<?php
$precio = 100;
$cantidad = 5;
$total = $precio * $cantidad;
/* $total contiene 500 */
$aumento = 2;
$total = $total + $aumento;
/* $total ahora contiene 502 */
?>
```

De paso, observemos que el **orden de ejecución** de los términos de una igualdad (asignación) es siempre de derecha a izquierda.

En esta expresión:

```
$total = $total + $aumento;
```

primero se procesó el término que está a la derecha del signo igual; es decir, el intérprete de PHP realizó el reemplazo de estas variables por sus valores:

```
$total + $aumento;
```

que, en este caso, fue igual a:

```
500 + 2
```

o, lo que es lo mismo:

```
502
```

Una vez terminada esa operación y reducidos los elementos del término de la derecha a un solo valor, ese valor fue asignado como nuevo valor de la variable $total, mediante el signo =, que es el **operador de asignación**; lo que sería lo mismo que simplificar todo el término derecho ya resuelto de esta manera:

```
$total = 502;
```

Así, no solo hemos visto que una misma variable puede reutilizarse, sino que hemos visualizado cómo se va procesando el código PHP, paso a paso, antes de asignarle valor a una variable.

Hemos visto suficientes detalles acerca del uso de **variables**. Ahora, ya estamos preparados para utilizar variables en los siguientes capítulos para cosas más interesantes que las de estos sencillos ejemplos.

# Las constantes: pocos datos que no cambiaremos

## Una variable que no vamos a modificar

Algunas veces necesitaremos que el valor que almacenamos en una variable no se modifique aunque cometamos un error.

Es decir, necesitamos que el valor almacenado **permanezca idéntico**, **constante**, hasta que el intérprete de PHP termine de procesar nuestra página.

En esos casos, en lugar de una variable nos convendría usar una **constante**. A diferencia de las variables, es imposible definirles un valor mediante un operador de asignación (el signo igual), lo que facilita que "ni por error" alteremos su valor durante toda su vida útil, ya que siempre almacenarán el mismo valor.

Las constantes se definen con la función **define()**, que necesita que coloquemos dos elementos separados por una coma: el nombre de la constante y su valor.

Veamos un ejemplo de su sintaxis:

```
<?php
define("PI", 3.1415926);
define("BR", "<br>");
define("LIBRO", "PHP, de Hernán Beati");
print(PI);
print(BR);
print(LIBRO);
?>
```

Muy importante: al momento de mostrar constantes (con echo o print), las constantes **no van entre comillas** (no son una cadena de texto a imprimir literalmente) **ni llevan un signo pesos delante** (no son una variable).

Además, al definirlas, por convención **se escribe su nombre totalmente en MAYÚSCULAS** para poder diferenciarlas a simple vista de las variables y de los textos literales. Recordemos lo importante que es facilitar la lectura de nuestro código.

Utilizaremos constantes muy a menudo en nuestros proyectos, para almacenar datos de conexión a base de datos, usuarios y contraseñas, y cualquier otra información que necesitemos almacenar sin riesgos de que la modifiquemos en otra parte de nuestro código.

## Includes de constantes

Es una muy buena idea definir **todas las constantes** de un sitio web en un **archivo aparte**, y luego incluir ese archivo (mediante **include** o **require**) en todas las páginas que necesiten utilizar esas constantes. Es una técnica muy utilizada en sitios multilingües, que definen constantes para los textos y usan un archivo para definir esos textos en cada idioma.

Veamos un ejemplo de esta técnica: imaginemos un sitio web con versión en español y en inglés. Existirán dos archivos que almacenarán constantes, uno con los textos en español y, otro, con los textos en inglés. Por ejemplo, dentro de la carpeta **lenguajes**, crearemos un archivo llamado **english.php** que contendrá solamente esto (notemos que no incluimos nada de código HTML, ya que este archivo solo está almacenando datos que serán mostrados dentro de otro archivo que ya tiene su propio código HTML completo):

```
<?php
define('NAVBAR_TITLE', 'My Account');
define('HEADING_TITLE', 'My Account Information');
define('OVERVIEW_TITLE', 'Overview');
define('OVERVIEW_SHOW_ALL_ORDERS', '(show all orders)');
define('OVERVIEW_PREVIOUS_ORDERS', 'Previous Orders');
?>
```

Y, dentro de otro archivo denominado **castellano.php**, colocaremos la traducción al español de esas mismas constantes:

```
<?php
define('NAVBAR_TITLE', 'Mi Cuenta');
define('HEADING_TITLE', 'Datos de Mi Cuenta');
define('OVERVIEW_TITLE', 'Resumen');
define('OVERVIEW_SHOW_ALL_ORDERS', '(ver todos mis
pedidos)');
define('OVERVIEW_PREVIOUS_ORDERS', 'Pedidos Anteriores');
?>
```

De esta manera, con indicar dentro de cada página el idioma que se mostrará (típicamente con una variable de sesión, tema que veremos más adelante), podremos incluir el archivo de textos correspondiente al idioma elegido por el usuario.

Por ejemplo, si preparamos una página llamada **cuenta.php** que debe mostrar esos textos en español, su código será similar a esto:

```
<!DOCTYPE html>
<html lang="es">
<?php include('lenguajes/castellano.php'); ?>
<head>
      <title><?php print(NAVBAR_TITLE); ?></title>
</head>
<body>
      <h1><?php print(HEADING_TITLE); ?></h1>
      <h2><?php print(OVERVIEW_TITLE); ?></h2>

      <ul>
         <li><a href="show_all_orders.php">
         <?php print(OVERVIEW_SHOW_ALL_ORDERS); ?>
         </a></li>
         <li><a href="show_previous_orders.php">
         <?php print(OVERVIEW_PREVIOUS_ORDERS); ?>
         </a></li>
      </ul>
</body>
</html>
```

Luego, será otro tema ver cómo podríamos hacer para obtener el dato de cuál idioma prefiere el usuario (mediante una variable de sesión, como dijimos, sería lo ideal) y utilizarlo en lugar de la palabra "castellano" dentro de la ruta del include, pero eso lo dejamos para más adelante.

Notemos que **no hemos incluido constantes dentro de bloques de texto** delimitados con comillas simples o dobles, como hacemos cuando queremos asignar un valor a una variable, ya que esto no funcionaría. Tampoco funcionaría colocarlas dentro de un echo o print junto con un texto envuelto entre comillas.

Veamos por qué:

```
<!DOCTYPE html>
<html lang="es">
<head>
      <title>Constantes</title>
</head>
<body>
      <h1><?php print("Bienvenido a HEADING_TITLE"); ?>
      </h1>
<!-- No se reemplazará por su valor, se escribirá el texto
     HEADING_TITLE tal cual, letra por letra -->

      <h2><?php print('Hola OVERVIEW_TITLE'); ?></h2>
<!-- Tampoco se reemplazará por su valor, se escribirá el
     texto OVERVIEW_TITLE tal cual, letra por letra -->
<?php
$comillasDobles = "Texto entre comillas dobles, no puede
contener constantes como HEADING_TITLE porque no se
reemplazarán por su valor";

$comillasSimples = 'Texto entre comillas simples, tampoco
puede contener constantes como HEADING_TITLE porque no se
reemplazarán por su valor';
?>
<!-- No se reemplazarán las constantes por su valor. -->
</body>
</html>
```

El motivo es que, dentro de esos bloques de texto, al no estar identificado el nombre de la constante con ningún signo especial (como el signo **$** que se utiliza para indicar que algo es una variable), el intérprete de PHP no tiene forma de detectar que se trata de una constante y no de una palabra común que, simplemente, tiene que escribir letra por letra.

Para remediar esta limitación, utilizaremos la técnica denominada **concatenación**, que hemos aprendido recientemente.

Por ejemplo:

```
$concatenado = 'Texto concatenado con una constante' .
HEADING_TITLE . ' que ahora sí se reemplazará por su
valor';
```

Así que siempre que intercalemos constantes, deberemos concatenarlas.

# Las matrices: muchos datos provisorios

## Un paquete de variables

Una matriz es un lugar en el que almacenaremos datos, de la misma manera que en las variables, pero con la posibilidad de almacenar **varios datos (valores) ordenados en distintos compartimientos**, en lugar de un solo dato, como en el caso de una variable.

Podríamos decir que, si una variable era una bicicleta que acarreaba una sola pequeña caja con su carga, las matrices equivaldrían a un camión cargado con decenas, cientos o miles de cajas, cada una guardando un dato diferente, pero todos dentro del mismo contenedor (el camión).
Otra comparación: si una variable era un estante único (y muy pequeño) en el que únicamente podíamos guardar un solo libro, comparativamente, una matriz equivaldría a un gran estante en el que entran decenas de libros o, incluso, una biblioteca completa, con decenas de estantes a la vez. Muchos libros, muchos estantes, pero todos dentro de una misma biblioteca, de un mismo elemento contenedor.

Veamos un par de ejemplos que nos aclararán un poco más la diferencia entre una variable y una matriz.

Este código declara y adjudica valor a una **variable**:

```
<?php
$numero = 514;
print ($numero);
// escribe 514.
?>
```


**Figura 4-4. Una variable.**

El nombre de la variable es **$numero** y el valor almacenado en este momento es el número **514**.

Ya hemos dicho que una variable es como **una caja** con un único compartimiento, donde colocamos datos (letras, números), y les asignamos un nombre para referirnos a "eso" que guarda la variable.

Ahora veamos la diferencia en el caso de definir una **matriz**.

En el siguiente código, declararemos una **matriz** de tres elementos o celdas, a la que denominaremos **$numeros**:

```
<?php
$numeros[0] = 75;

$numeros[1] = 90;

$numeros[2] = 45;

print ($numeros[0]."<br>".$numeros[1]."<br>".$numeros[2]);

// Escribirá: 75<br>90<br>45
?>
```

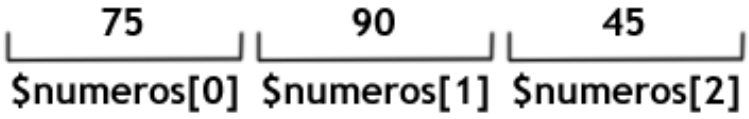


**Figura 4-5. Una matriz.**

En este caso, el nombre de la matriz es **$numeros** y tiene tres subdivisiones internas a las que denominaremos "elementos" o "celdas" (cada elemento de la matriz es una de las tres "cajitas" que observamos en la figura anterior). Notemos que cada elemento (o celda) es un par que consta de un **índice** (identificador) y de un **valor**.

El primer elemento de esta matriz tiene como índice el número **0** y como **valor** un **75**

El segundo elemento, con índice **1**, almacena un **90** y el tercer elemento de índice **2** guarda un número **45** en su interior.

Es decir, a diferencia de una variable, una matriz almacena **varios datos**, cada uno de ellos con el mismo nombre de matriz ($numeros, en este caso), pero con un **nombre de índice siempre diferente, único**.

Sería similar a un estante de una biblioteca, al que denominamos "libros de inglés". Si quiero el primero de los libros de ese estante, lo pediría diciendo "el primer libro del estante de inglés"; si quisiera el segundo, diría "el segundo libro del estante de inglés"; en la programación en PHP, podríamos referirnos a esa situación con esta notación: $libros_de_inglés[1] y, si quisiera que me alcanzaran el segundo de los libros, me referiría a él como $libros_de_inglés[2] y así, sucesivamente.

Otro ejemplo: una matriz es como una caja grande, subdividida en otras "cajitas" más chicas adentro, cada una de las cuales guarda algo. Las metáforas pueden ser realmente muchas...

El único punto que entra en conflicto con la vida real es que los **índices** que identifican a cada celda de una matriz, en PHP, no se numeran a partir de 1 en adelante, como en la vida real (nunca pediríamos "el libro cero del estante", pedimos el "primero", el "1"), sino que en PHP **se numeran a partir de cero**. La primera celda es la [0] (se lee "subcero"). La segunda es la "sub 1", la tercera es la "sub 2" y así, sucesivamente, manteniendo ese "corrimiento" de un dígito.

Veamos un segundo ejemplo del uso de una matriz:

```
<?php
```

```
$paises[0] = "Afganistán";
$paises[1] = "Albania";
$paises[2] = "Alemania";
$paises[3] = "Andorra";
$paises[4] = "Angola";

print($paises[0]."<br>".$paises[1]."<br>".$paises[2]."<br>
".$paises[3]."<br>".$paises[4]);
// Escribirá: Afganistán<br>Albania<br>Alemania<br>
Andorra<br>Angola
?>
```

Será sumamente común el utilizar matrices como almacén provisorio de datos provenientes de un almacén permanente de datos, tal como una base de datos o un archivo de texto plano.

Distintas funciones de PHP que ya veremos en otros capítulos, se encargarán de ubicar dentro de matrices los datos leídos de una base de datos (como la lista de países para mostrarlos en un menú de selección HTML de un formulario, por ejemplo, o el listado de los distintos productos de un sitio de comercio electrónico, incluyendo su nombre, descripción, precio, imagen, etc.), y será muy fácil trabajar con esa matriz como depósito temporal de datos, típicamente utilizando un bucle para recorrerla (algo que veremos muy pronto).

## Índices numéricos

Las matrices pueden utilizar dos tipos de índices: **numéricos** (los que vimos hasta ahora) y **alfanuméricos**.

Comenzaremos a ver cómo se cargan datos en las matrices de índices numéricos.

Hay diferentes maneras de "inicializar" una matriz de índices numéricos (esto es, darle un índice y un valor a cada una de sus celdas; es decir: colocar datos dentro de ella). Podemos realizar esa tarea de forma **explícita**, de forma **implícita**, mezclando **ambas** formas, o usando el constructor **array**. Veremos, a continuación, cada una de estas posibilidades.

### Declaración explícita

En el ejercicio de líneas anteriores, hemos **declarado explícitamente** cada índice de la matriz, colocando **entre corchetes** el número de índice deseado para el valor que inmediatamente le adjudicamos:

```
$paises[0] = "Afganistán";
$paises[1] = "Albania"; // etc.
```

Los números de índice no necesariamente deben comenzar de cero, ni necesariamente ser consecutivos, aunque eso ya lo veremos cuando estudiemos los bucles.

Si luego precisábamos acceder a un dato almacenado, lo hacíamos especificando su índice, de esta manera:

```
print ($paises[0]);
print ($paises[1]); // etc.
```

y así sucesivamente, especificando explícitamente el número de índice que identificará a cada dato.

### Declaración implícita

Pero también podríamos haber realizado el mismo ejemplo con una **declaración implícita** de índices, de la siguiente manera (notemos los corchetes vacíos en el primer renglón):

```
<?php
$paises[] = "Afganistán"; /* Esta es una declaración
implícita, dejando vacíos los corchetes. */

$paises[1] = "Albania";
$paises[2] = "Alemania";
$paises[3] = "Andorra";
$paises[4] = "Angola";

print ($paises[0]."<br>".$paises[1]."<br>".$paises[2].
"<br>".$paises[3]."<br>".$paises[4]);

// Escribirá: Afganistán<br>Albania<br>Alemania<br>
Andorra<br>Angola
?>
```

Si **omitimos los números de índice** al momento de ir ubicando valores dentro de las celdas de la matriz, el intérprete de PHP le asignará automáticamente uno, y colocará **números correlativos**, comenzando por el menor número posible (un **cero, salvo que especifiquemos otra cosa**) para la primera celda de la matriz.

Notemos que, al dejar vacíos los corchetes, simplemente se omite un paso, que es el de asignarle nosotros un número específico al índice, pero, de todos modos, el intérprete de PHP lo completa automáticamente. Por lo tanto, en el momento en que se quiera leer esos datos, sigue siendo imprescindible especificar **cuál celda** de la matriz queremos leer.

No funcionaría si hacemos esto:

```
print ($paises[]); // faltaría indicar cuál celda mostrar.
```

ya que no estamos diciendo **cuál** de las celdas de la matriz queremos mostrar. Siempre dentro de los **echo** o **print** deberemos especificar explícitamente el índice al que queremos acceder.

### Mezcla de declaración explícita e implícita

También, podría suceder que quisiéramos especificar un índice en particular para la **primera** celda de la matriz, y luego sí dejar que el intérprete siga colocando los índices automáticamente, pero a partir de ese valor inicial que nosotros especificamos:

```
<?php
$dias[1] = "Lunes"; /* Especificamos un índice inicial (el
"1" en este caso), y luego dejamos que PHP coloque los
demás automáticamente: */

$dias[] = "Martes";
$dias[] = "Miércoles";
$dias[] = "Jueves";
$dias[] = "Viernes";

print ($dias[1]."<br>".$dias[2]."<br>");
print ($dias[3]."<br>".$dias[4]."<br>".$dias[5]);
/* Muestra:
Lunes<br>Martes<br>Miércoles<br>Jueves<br>Viernes
*/
```

```
?>
```

Al haber especificado nosotros **uno** de los índices, pero no los siguientes, PHP **continúa la numeración** desde el valor siguiente al último índice especificado.

### La función array

Esta declaración **implícita** de índices numéricos es la misma que PHP emplea cuando utilizamos una forma mucho más simple y más breve de declarar matrices, mediante el uso de la función llamada **array**, cuya sintaxis veremos a continuación:

```
<?php
$paises = array("Argentina", "Uruguay", "Chile", "Perú");
/* Crea una matriz llamada $paises de cuatro elementos con
índices numerados a partir de cero. */

$loteria = array(23,8,36,12,99);
/* Crea una matriz de cinco elementos con índices
numerados a partir de cero. */

$usuario = array("Juan Pérez", 24, "casado", 800);
/* Crea una matriz de cuatro elementos con índices
numerados a partir de cero. */
?>
```

Como vemos en el último caso, una única matriz puede almacenar **datos de distinto tipo** (caracteres, números enteros, decimales, etc.) y, por lo tanto, es necesario colocar entre comillas los textos para que PHP sepa que son eso, textos, y no números.

También, notemos que simplemente **una coma** separa un dato de otro.

Tal como en la mezcla de asignación explícita e implícita de índices vista anteriormente, cuando usamos la función **array** también podemos **forzar el índice** de uno de los elementos de la matriz (no necesariamente debe ser el primero de ellos), y eso se realiza con el operador **=>** de la siguiente manera:

```
<?php
```

```
$paises = array(1 =>"Argentina", "Uruguay", "Chile",
"Perú");
/* Crea una matriz llamada $paises de cuatro elementos,
cuyo primer elemento posee un "1" como índice, el segundo
un 2, y así sucesivamente. */
?>
```

Como dijimos, no es necesario que sea el primer elemento aquel al cual le forzamos un índice:

```
<?php
$paises = array("Argentina", 10 => "Uruguay", "Chile",
"Perú");
/* Crea una matriz llamada $paises de cuatro elementos,
cuyo primer elemento posee un "0" como índice, el segundo
un "10" y luego el resto continúa con "11" y "12". */
?>
```

### Índices numéricos no consecutivos

Es bastante común que los índices asignados a una matriz sean números salteados, no consecutivos como, por ejemplo, **códigos** de artículos. En ese caso, a medida que agregamos datos al vector, puede suceder que los índices no sean consecutivos y queden desordenados.

Por ejemplo:

```
<?php
$productos[1234] = "Televisor LG de 42 pulgadas";
$productos[145] = "Televisor Sony de 29 pulgadas";
$productos[899] = "Televisor portátil de 12 voltios";
?>
```

Esta sería una matriz de índices numéricos, no consecutivos.

En este ejemplo, el primer índice es **1234** y el que le sigue no es **1235**, como sería esperable si fuera consecutivo. Es **145** y, luego de **145**, tampoco sigue **146**, sino otro número cualquiera.

Ya veremos cómo interactuar con matrices que tengan este tipo de índices numéricos desordenados o no consecutivos, cuando aprendamos a recorrer las matrices mediante bucles.

### Índices alfanuméricos

En muchos casos, en especial cuando trabajemos con bases de datos, definir los índices de la matriz con **cadenas de texto (alfanuméricas)** en lugar de utilizar números, será de mucha utilidad para facilitar la lectura del código.

En PHP, se puede hacer de la siguiente manera:

```
<?php
$datos["nombre"] = "Juan Pérez";
$datos["edad"] = 24;
$datos["estado"] = "casado";
$datos["sueldo"] = 800;

print ($datos["nombre"]); // Escribe: Juan Pérez
?>
```

Esto se lee: “datos sub-nombre”, “datos sub-edad”, etc. (el prefijo “sub” delante del nombre del índice proviene de que tradicionalmente en programación se le llama “subíndice” al índice de una matriz)

Notemos que dentro de los corchetes, en lugar de números, hemos colocado **palabras descriptivas** de lo que contiene esa celda. Esos son los índices o identificadores alfanuméricos. Como todo texto en PHP, debe ir entre comillas.

Son muy recomendables para facilitar la lectura, para darnos cuenta qué contiene cada celda de una matriz.

### Matrices definidas automáticamente por el intérprete de PHP

Un caso especial dentro de las matrices de índices alfanuméricos, son aquellas matrices que el intérprete de PHP declara y completa con datos **automáticamente**, sin que nosotros tengamos que hacer nada, tan solo leerlas y utilizar la información que nos proporcionan.

Ya que la lista de los posibles valores de **índices alfanuméricos** de estas matrices es muy extensa (a veces, decenas de valores posibles en cada matriz), recomendamos consultar el manual oficial de referencia de PHP en línea, en: http://www.php.net

Si agregamos el nombre de la matriz que queremos conocer al final de la URL, accederemos directamente a la página del manual dedicada a este tema, por ejemplo: http://www.php.net/_server

A continuación, veremos un cuadro con los **nombres** de estas matrices definidas por el intérprete de PHP:

| MATRIZ | QUÉ CONTIENE | EJEMPLOS DE USO |
|---|---|---|
| **$_SERVER** | Contiene información disponible en el servidor Web: rutas, cabeceras HTTP enviadas por el navegador del usuario tales como el navegador utilizado, la dirección IP del usuario, etc. | echo $_SERVER[‘HTTP_USER_AGENT’]; |
| **$_ENV** | Contiene información acerca del entorno en el que el intérprete de PHP está siendo utilizado (nombre de la computadora, del servidor, etc.). | echo $_ENV[‘HOSTNAME’]; |
| **$_SESSION** | Contiene las variables de sesión que hayamos declarado. El índice es el nombre de la variable. | echo $_SESSION[‘mi_variable’]; |
| **$_GET** | Contiene las variables enviadas hacia el servidor mediante enlaces (adjuntadas a una petición HTTP). El índice es el nombre de la variable. | echo $_GET[‘mi_variable’]; |
| **$_POST** | Contiene las variables enviadas mediante formularios que declaren el método “post”. El índice es el nombre de la variable. | echo $_POST[‘mi_variable’]; |
| **$_COOKIE** | Contiene las variables almacenadas por el navegador del usuario en *cookies*. El índice es el nombre de la | echo $_COOKIE[‘mi_variable’]; |

| | | |
|---|---|---|
| | variable. | |
| **$_REQUEST** | Contiene las variables almacenadas en las tres matrices anteriores: $_GET, $_POST y $_COOKIE. Es decir, todas las variables que fueron enviadas por el navegador del usuario hacia el servidor. | echo $_REQUEST[‘mi_variable’]; |
| **$_FILES** | Contiene información acerca de los archivos que hayan sido enviados mediante un formulario que tuviera un control *input* de tipo *file*. | echo $_FILES[‘el_archivo’] [‘name’]; |
| **$GLOBALS** | Contiene información sobre todas las variables definidas, ya sea automáticamente por el servidor, como definidas por nosotros mismos. Notemos que es la única matriz definida automáticamente que no lleva un guion bajo delante de su nombre. | echo $GLOBALS[‘mi_variable’]; |

**Tabla 4-2. Matrices definidas automáticamente por el intérprete de PHP.**

Muchos de los valores de estas matrices **no están disponibles** en **todos** los servidores, ya que dependen de que el administrador del servidor haya habilitado su uso, por lo cual no es posible hacer una lista completa que esté disponible bajo todas las circunstancias. Habrá que probar en cada servidor antes de utilizar un dato proveniente de estas matrices.

Notemos que todos los nombres de estas matrices –salvo uno– comienzan con guion bajo, y que todas estas matrices definidas por el intérprete de PHP

llevan escrito su nombre completamente en **mayúsculas**, tal como si fueran una constante; eso permite que podamos diferenciarlas fácilmente dentro de nuestro código de las otras matrices que nosotros mismos hayamos declarado.

Será muy común de ahora en más que utilicemos en nuestros códigos datos obtenidos a partir de estas matrices definidas automáticamente por el intérprete de PHP. Ya en el siguiente capítulo comenzaremos a utilizar varias de estas matrices.

Y con esto damos por terminada esta introducción a los almacenes de datos provisorios que más utilizaremos en PHP: variables, constantes y matrices. Seguiremos profundizando más detalles y aplicando reiteradamente estos almacenes de datos en los siguientes capítulos, así como también aprenderemos a interactuar con los restantes almacenes de datos más perdurables.

# Enviando datos hacia el servidor

# 5

## Herramientas para enviar datos: enlaces y formularios

En los capítulos anteriores, nos hemos referido al concepto y a la sintaxis básica de PHP, y los hemos aplicado en ejemplos concretos. A partir de aquí, el contenido de esta obra se pondrá más interesante sobre todo para los programadores, ya que comenzaremos a interactuar con datos y código HTML de manera más amplia. Recién hemos visto cómo la función **print** o el comando **echo** son capaces de escribir, en el código HTML de una página, algún dato que estaba almacenado en una variable. Pero el valor de esa variable lo definíamos nosotros. ¿Qué pasaría si dejáramos que fuera el **usuario** de nuestro sitio Web el que le ingresara valores a algunas de nuestras variables?

La respuesta es que comenzaríamos a crear **páginas Web dinámicas**, es decir: páginas cuyo contenido cambiará (totalmente, o solo en parte), según las acciones que realice **el usuario** justo antes de pedir que se le muestre una de esas páginas. En definitiva: páginas interactivas.

Los navegadores Web, junto con el pedido habitual que solicita una página Web en particular, son capaces también de **enviar datos** al programa servidor Web que está funcionando en el *hosting*.

Esto significa que no son solo "receptores" de código HTML, al que convierten en información decorada y visible en nuestra pantalla, sino que los navegadores también son "emisores" de datos enviados desde nuestra

computadora hacia el *hosting*, y estos datos son remitidos en la forma de **variables**, de dos maneras:

1. mediante **enlaces** especiales
2. y mediante **formularios**.

A continuación, analizaremos ambas posibilidades, puesto que el envío de datos desde el navegador de un usuario hacia el servidor, constituye un giro fundamentalmente interactivo, que deja en manos de cada usuario los contenidos (dinámicos) que recibirá como respuesta del servidor.

## Enlaces con variables en la URL

La primera herramienta con la que contamos para definir remotamente un valor a una variable ubicada en el servidor son los **enlaces que envían variables**. Es importante aclarar que nos referimos a enlaces que contemplamos en un archivo que ya está en nuestra pantalla; es decir, que se encuentran en estado "potencial" en un archivo HTML que ya fue descargado en nuestra computadora, a la espera de que pulsemos alguno de ellos para que envíe un dato hacia el servidor.

Cuando creamos un enlace en HTML, habitualmente se parece al código que se observa en el siguiente cuadro (el lector lo podrá transcribir dentro de un archivo completo con la extensión **.html**):

```
<a href="destino.html">Esto es un enlace que solicita al
servidor el archivo llamado destino.html</a>
```

El objetivo de la etiqueta **<a>** (la a proviene de la palabra *anchor* –ancla–; es decir, un elemento que nos "enlaza" con otra página) es pedirle al servidor que nos entregue la página especificada en el atributo **href** que, en este caso, es la página **destino.html**. Obviamente, si no hemos ubicado ninguna página llamada **destino.html** dentro de la misma carpeta en la que está este archivo, nos dará un error cuando pulsemos el enlace que solicita ese recurso.

Sin embargo, en el mundo de las páginas dinámicas, podemos aprovechar la posibilidad que nos dan los enlaces de **decirle algo** al servidor, para que, además de pedirle que muestre una página en particular, nos permita **enviar un valor** a alguna **variable** que estará disponible en el servidor, para que la utilice el programa intérprete de PHP unos instantes después, mientras procesa el código PHP de esa página, justo antes de que le devuelva el código HTML generado a nuestro navegador, que espera una respuesta.

La sintaxis que permite el envío de variables junto con un enlace es muy simple: supongamos que queremos definir en el servidor una variable que se llame **nombre**, cuyo valor será **Pepe**, y que el enlace se dirige hacia una página

llamada **recibe.php**. Este enlace –que solicita una página y, a la vez, envía una variable con ese valor hacia el servidor– se expresaría de la siguiente manera:

```
<a href="recibe.php?nombre=Pepe">Solicitamos ver la
página "recibe.php" y de paso enviamos al servidor una
variable llamada "nombre" conteniendo el valor "Pepe"</a>
```

Es importante que observemos que dentro del valor del atributo **href**, luego de especificar la URL solicitada (la página **recibe.php**), se ha agregado un signo de pregunta: **?** Este signo indica que, a continuación, **enviaremos una o más variables** hacia el servidor Web.

La sintaxis para enviar una variable consiste en colocar primero su **nombre**; luego, un signo igual; después, su valor de definición.

A diferencia de la sintaxis empleada en el lenguaje PHP, no debemos anteponer ningún signo "$" al nombre de la variable, ni tampoco envolver entre comillas los valores de las variables, aun cuando sean alfanuméricos.
Esto es así ya que el envío de variables como parte de un enlace, no tiene relación con PHP ni con ningún otro lenguaje de programación en particular, sino que es una posibilidad que nos brinda el **protocolo HTTP**, que es el que comunica a un navegador con un servidor Web. No son variables de PHP hasta que llegan al servidor y el programa intérprete de PHP las lee.

En el ejemplo siguiente, crearemos un archivo llamado **enlaces.html** y, otro, denominado **destino.php**.

El archivo **enlaces.html** será el siguiente:

```
<p>
<a href="destino.php?nombre=Pepe">Este es el enlace de
Pepe</a><br>
<a href="destino.php?nombre=Pedro">Este es el enlace de
Pedro</a><br>
<a href="destino.php?nombre=Juan">Este es el enlace de
Juan</a>
</p>
```

Una vez que nuestro navegador visualiza esa página, según cuál de los enlaces pulsemos, enviaremos hacia el servidor un **valor** distinto para la variable

**nombre**. Siempre mandaremos la misma variable, pero con distinta información almacenada dentro de ella. Y, cabe señalar, que solo remitiremos uno de los valores posibles (no existe un *mouse* que nos permita pulsar simultáneamente más de un enlace).

En el archivo **destino.php**, daremos por sentado que llegó esa variable, por esta razón, escribiremos la orden print dentro del código HTML, que solicitará al intérprete de PHP que escriba el valor de la variable denominada **nombre**.

Para eso, usaremos esta sintaxis:

```
<?php
print($_GET["nombre"]);
?>
```

Por supuesto, a esta página le falta la DTD y la apertura de las etiquetas html, head, body, etc., y su correspondiente cierre, que deberemos agregar al codificar este ejemplo.

## Las matrices superglobales

### Matrices que almacenan datos automáticamente

En el capítulo anterior, hemos visto que el intérprete de PHP almacena automáticamente los datos que tiene a su alcance en varias **matrices**. Una de ellas es la denominada **$_GET** (lo pronunciamos "guion bajo get"). Se trata de una matriz que almacena todas las variables que hayan sido enviadas hacia el servidor **adjuntadas a un enlace**, es decir: adjuntadas a una petición de un navegador realizada mediante el método *get* del protocolo HTTP (dicho técnicamente es así pero, de forma más simple, PHP almacena, en esa matriz $_GET, las variables adjuntadas a un enlace).

Como en toda matriz, entre los corchetes de la matriz $_GET debemos indicar el **subíndice** que queremos leer en este momento; en este caso, será siempre alfanumérico, ya que se **utilizará el nombre de la variable** que hemos enviado en el enlace. En el ejemplo anterior, escribiremos "nombre" dentro de esos corchetes, ya que ése era el nombre de la variable que enviamos al final del enlace.

Siguiendo con el ejercicio, según el enlace que pulsemos, el contenido que veremos en la página **destino.php** será diferente. Si miramos la **barra de direcciones** de nuestro navegador, notaremos que la variable enviada y su valor son **visibles** en la URL de la página.

Algún lector se preguntará: ¿y cómo hacemos para enviar **más de una** variable a la vez, en un único enlace? Simplemente, uniendo cada par de **variable=valor** con un signo ampersand (&).

Ejemplo de cómo enviar **varias variables** en un solo enlace:

```
<p>
<a href="datos.php?nombre=Pepe&apellido=Perez&edad=17">
Este es el enlace de Pepe</a><br>
<a  href="datos.php?nombre=Pedro&apellido=Garcia&edad=9">
Es te es el enlace de Pedro</a><br>
<a href="datos.php?nombre=Juan&apellido=Gomez&edad=30">
Este es el enlace de Juan</a>
</p>
```

Ese código enviará hacia el servidor tres variables: nombre, apellido y edad.

Nota importante: si este código se incluye no en una página HTML5 actual sino dentro de un documento XHTML Strict, deberemos **codificar el ampersand** con su respectiva entidad HTML, de esta manera:
**&**
Por lo tanto, cada enlace será similar a éste:
<a  href="datos.php?nombre=Pedro**&**apellido=Garcia**&**edad=9">Este es el enlace de Pedro</a><br>

El contenido del segundo archivo, denominado **datos.php**, será así:

```
<?php
print ("<p>Los valores fueron: ");
print ("<br>");
print ($_GET["nombre"]);
print ("<br>");
print ($_GET["apellido"]);
print ("<br>");
print ($_GET["edad"]);
print ("</p>");
?>
```

Notemos al pasar que hemos ordenado al intérprete de PHP escribir, entre un dato y otro, un salto de línea o *break* (<br>), para que los valores queden claramente uno en cada línea, y no todos a continuación en un mismo renglón.

Una variante de este envío de variables, mediante el método *get*, es experimentar su paso directamente **desde el navegador**, escribiendo sus nombres y sus valores en la **barra de direcciones** de nuestro navegador; esto es muy práctico para probar qué valores están llegando a la siguiente página rápidamente, sin tener que crear una página con un enlace específico para probarlo. A menudo, utilizaremos esta técnica para detectar errores mientras programamos, si sospechamos que no llega al servidor el dato que esperamos.

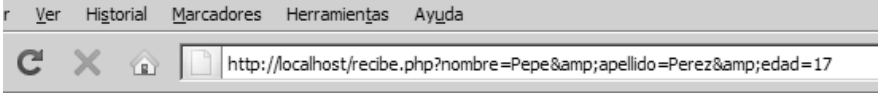


**Figura 5-1. Cómo enviar variables desde la barra de direcciones del navegador.**

Escribir eso dentro de la barra de direcciones del navegador, equivale a colocar dentro de un enlace el atributo **href** con este valor:

```
http://localhost/recibe.php?nombre=Pepe&apellido=Perez&ed
ad=17
```

Cuando pulsemos **Enter** esto se enviará al servidor y se logrará un efecto idéntico a los enlaces anteriores, puesto que pulsar un enlace o escribir algo en la barra de direcciones de un navegador logra el mismo efecto: en ambos casos, el navegador realiza una **petición del protocolo HTTP**, que utiliza el **método *get*** para adjuntar las variables.

## Formularios

Otro método para enviar variables hacia el servidor es el típico **formulario** de ingreso de datos, en el que una persona escribe algo en su navegador, mientras visualiza el formulario en su pantalla. Luego, se enviarán esos datos hacia el servidor para que se realice con ellos alguna operación (por ejemplo: enviar los datos a una base, o por correo electrónico, o utilizarlos para construir una página dinámica que se mostrará a continuación).

Un formulario consiste en una simple etiqueta del lenguaje HTML denominada *<form>*, por lo que no es preciso que la página en la que se incluye el formulario lleve extensión **.php**, sino que puede ser un archivo **.html** normal (aunque la extensión .php tiene ciertas ventajas que apreciaremos más adelante, cuando aprendamos a validar los datos ingresados en un formulario).

Los elementos principales de un formulario (además de la etiqueta <form>) son los siguientes:

1. El atributo **action**, que indica a qué página de destino se envían las variables. Además, es ésa la página que se nos mostrará cuando pulsemos en el botón **Enviar**, tal como si hubiésemos pulsado un enlace hacia esa página. Siempre apuntaremos hacia una página con extensión .php, para poder leer los datos enviados por el formulario.
2. El atributo **method**, que especifica uno de los dos posibles métodos o formas de enviar las variables hacia el servidor: **a la vista** de todos, en la URL del navegador (method="**get**"), o de manera oculta, invisible en la URL (method="**post**"). Este último método es el más utilizado en formularios y es el que recomendamos.
3. Algún **campo** o **control** de formulario (campo de texto, menú de selección, botón de tipo radio, casilla de selección, etc.), que permita al usuario el ingreso o la selección de datos. Lo fundamental de cada campo será su **nombre** (atributo *name*), ya que ése será el de la **variable** que estará disponible en la página de destino.
4. Un **botón** (un campo *input* de tipo *submit*) para enviar los datos.

Esos son los elementos mínimos con los que debe contar un formulario para que pueda enviar datos hacia el servidor.

Veamos un ejemplo de todos estos elementos aplicados en la creación de un archivo al que llamaremos formulario.html:

```
<form action="muestra.php" method="post">
      <input type="text" name="domicilio">
      <input type="submit" value="Enviar">
</form>
```

Y ahora realizaremos muestra.php, que es la página que recibirá la variable que contiene lo que haya escrito el usuario en el formulario, y la mostrará (obviamente, falta agregar la DTD y las etiquetas estructurales básicas):

```
<?php
print ("Su dirección es: ");
print ($_POST["domicilio"]);
?>
```

Esperamos a que llegue una variable llamada **domicilio** a esta página, y notemos que, como fue enviada al servidor mediante un formulario que especificaba el método *post*, el intérprete de PHP acomodó esa variable dentro de la matriz **$_POST**, por eso tenemos que leer el dato de una celda de esa matriz, para ser precisos, de la celda $_POST[“domicilio”].

Cada control de formulario de tipo **input** nos generará **una variable** que se enviará hacia el servidor, por lo cual, si quisiéramos que el usuario escriba varios datos, solo tendremos que agregar **varios inputs**, de cualquier tipo (campo de texto, área de texto, menú de selección, botones tipo radio, casillas de selección, etc.).

| Si el nombre del campo fuera: | El nombre de la variable sería: | Y se podría leer en el servidor como: |
|---|---|---|
| name=“domicilio” | domicilio | $_POST[“domicilio”] |
| name=“ciudad” | ciudad | $_POST[“ciudad”] |

**Tabla 5-1. De nombre de campo, a variable en el servidor.**

Recomendamos **repasar algún manual o tutorial de HTML** para recordar cuáles son todos los elementos (tipos de controles) posibles de un formulario.

En los capítulos siguientes, aprenderemos a validar los datos esperados y aplicaremos un manejo bastante más avanzado de los formularios que el que nos ofrece HTML.

## Ventajas y limitaciones de enviar variables por el método *get*

Hemos visto que existen dos métodos para que el usuario envíe datos hacia el servidor: mediante enlaces que envían variables, y mediante formularios. Método *get*, y método *post*. Veamos qué criterios debemos considerar al elegir, si presentamos al usuario un enlace o un formulario.

Comencemos por el método *get*. ¿Cuáles son sus ventajas?:

- Codificar el código HTML necesario para enviar un dato mediante una petición *get* (un enlace que adjunta variables) es muy sencillo, no necesitamos crear ningún formulario, simplemente creamos enlaces comunes, que envíen los datos necesarios al ser pulsados por el usuario.
- Es invisible para la mayoría de los usuarios, que creerán que pulsan un enlace común y no sospecharán que están enviando un dato hacia el servidor. Por esta “invisibilidad”, es probable que sea usado más que un formulario.

El uso más común de peticiones con el método *get* es en **consultas** a una base de datos, ya sea para una búsqueda, un filtrado de productos de un

catálogo por categorías, una botonera donde el usuario elige un rubro o sección de un sitio, etc. Es decir, cuando ese dato que enviamos puede estar a la vista (y hasta ser modificado fácilmente por un usuario "curioso" que escriba en la barra de direcciones), ya que no se usará para modificar datos de nuestra base de datos, sino solo para mostrar parte de la **información pública** del sitio que, de todos modos, sería factible encontrar.

Por ejemplo, un buscador:

```
http://www.google.com.ar/search?q=palabra
```

¿Desventajas? Las direcciones visitadas quedarán guardadas en el **historial** del navegador. Lo que posibilitaría que un usuario vuelva a enviar la misma variable con el mismo valor en otro momento. Eso, sumado a nuestra forma de programar, podría ser riesgoso si usáramos los enlaces para enviar al servidor datos importantes como, por ejemplo, usuarios y contraseñas. Imaginemos un enlace que fuese así:

```
login.php?usuario=pepe&password=secreto
```

Esto produciría un serio problema de seguridad a nuestros usuarios si, por ejemplo, acceden desde una computadora compartida, puesto que otros usuarios podrían visitar el enlace desde el historial del navegador y entrar donde no deben.

La conclusión es que **no debemos enviar datos importantes** mediante peticiones que usen el método *get*. Pero sí podemos usarlos para enviar datos de "navegación", como códigos de categorías, de productos, palabras a buscar, etc.

### Ventajas y limitaciones de enviar variables por el método post

El uso más común de un formulario con método *post* es enviar datos que serán **almacenados** en una base de datos, o que provocarán **operaciones** que habilitarán acceso a datos privados (como un formulario de acceso a una zona privada, o de registro).

¿Ventajas?

- Una de las potenciales ventajas de declarar en nuestros formularios que se utilice el método *post*, es lograr que nuestro navegador se conecte con el servidor y envíe los datos de los campos del formulario de forma absolutamente **invisible**, discreta, al menos en la barra de direcciones del navegador. Por supuesto, eso dificulta la modificación de la barra de direcciones para enviar datos extra que permite el método *get,* pero nada impide que un usuario "curioso" edite el formulario en su computadora, y envíe datos no esperados al servidor también mediante el método *post*.
- A diferencia del método *get*, los datos enviados por el método *post* no tienen un límite respecto a la **cantidad** de variables enviadas en una misma petición, o a la **longitud** de sus valores.

¿Desventajas? Varias:

- Si el acceso a una página depende del envío de datos al servidor mediante el método *post*, cada vez que queramos volver a ingresar a esa página deberemos enviar nuevamente los datos del formulario, y la mayoría de navegadores advertirá esto con un mensaje de **alerta** al usuario, quien no siempre comprenderá qué es lo que está sucediendo.
- Otra desventaja: si el usuario guarda en sus **Favoritos** una página que recibió datos mediante *post*, cuando utilice ese Favorito, ya no estarán disponibles los datos que envió la primera vez desde un formulario y, por lo tanto, el resultado que obtendrá ya no será el mismo (u obtendrá el típico mensaje de alerta).

Volvemos a reafirmar la conclusión que sostiene que para enviar información **privada**, usaremos *post*; para navegar por información **pública**, usaremos *get*.

Y también usaremos *get* cuando deseemos que nuestra página pueda ser guardada en los Favoritos del usuario sin provocar problemas de usabilidad.

Con esto damos por finalizada la introducción a las técnicas para posibilitar al usuario que envíe datos desde su navegador hacia el servidor, ya sea mediante enlaces que usen el método get, o mediante formularios que usen el método post, y sabemos que en el servidor, cuando se reciban esos datos, podremos ir a leerlos a las matrices $_GET y $_POST que es donde el programa intérprete de PHP ubica automáticamente estos datos enviados desde el navegador hacia el servidor.

En el siguiente capítulo, aprenderemos a validar estos datos que el usuario envía, antes de utilizarlos en nuestros códigos.

# Validaciones

# 6

## Validando datos de formularios y enlaces

Para mostrar datos que se enviaron desde un enlace o un formulario, podemos utilizar cualquiera de los métodos que ya hemos aprendido en capítulos anteriores: dejar preparadas las etiquetas HTML dentro de las que intercalaremos con **echo** las variables recibidas, o hacer que PHP genere tanto los datos de las variables como el propio código HTML que las envuelva.

Pero esperar un dato del usuario para mostrarlo a continuación, tiene sus riesgos. Como la llegada de estos datos depende del usuario, muchas veces nos encontraremos con la sorpresa de que el dato esperado **no llegó**; es decir, nada nos garantiza de manera infalible que el usuario haya proporcionado esos datos que estamos esperando para usarlos a continuación en el armado de una página dinámica que muestre esos datos.

Por eso, es sumamente importante que aprendamos a **verificar:**

1. que **hayan llegado** al servidor esos datos,
2. que hayan sido enviados por el medio que esperábamos. Es decir, verificaremos su **origen** (que hayan sido enviados mediante el método *get,* o *post,* según el caso),
3. y que su **valor** sea válido (que no esté vacío, que sea el tipo de dato que esperamos, que esté dentro del rango de valores permitido, etc.).

Para esta tarea tan común de verificación de datos antes de utilizarlos, se usan los condicionales, que es el tema que aprenderemos a continuación.

# Los condicionales

## If (si...)

Muchas veces querremos que nuestro código sea capaz de **decidir automáticamente** si se ejecuta un bloque de código, o no, dependiendo de **algo** que haya sucedido (la condición que se evaluará puede ser algo que haya sucedido con los datos que vienen del usuario, que él haya hecho clic en una opción o en otra, que una variable tenga un valor, o no, que se haya llegado a la página mediante un método u otro, etc.).

Algunos ejemplos: si la contraseña enviada por el usuario es igual a la que está definida dentro de nuestro código, le mostramos un contenido especial. Si es viernes, mostramos una frase deseando un buen fin de semana. Si el usuario eligió ver la categoría "A" de productos, le mostramos el listado únicamente de esos productos y no de otros.

Continuamente, nos encontramos con este tipo de situaciones en la vida real, por ejemplo: "si está lloviendo, llevo el paraguas"; *"si es de noche, enciendo una lámpara".* Es una cuestión de simple **lógica**.

Para proveer a nuestro código la capacidad de "evaluación de las circunstancias" y su posterior decisión automatizada, en PHP, disponemos del **condicional if**, que nos permite evaluar si una circunstancia es "verdadera", si está sucediendo y, según el resultado de esa evaluación, nos permite ejecutar un bloque de código.

Es decir: podemos hacer que la ejecución de un bloque de código sea **condicional** (de allí, el nombre), que solo en el caso de que la condición planteada sea verdadera, se ejecute; en caso contrario, no sucederá nada, se continuará ejecutando el código siguiente a ese condicional.

La estructura del **if** puede tener algunas variantes, la más simple de todas es la que sigue:

```
if (condición a evaluar) {
      Bloque que se ejecuta solo si esa condición
      resulta ser verdadera
}
```

Y si no era verdadera, no pasa nada, no se ejecuta nada. Aplicado a la vida real, sería así en el caso de la lluvia:

```
if (llueve) {
      Llevar el paraguas
}
```

O lo que es lo mismo, pero graficado en forma de diagrama:

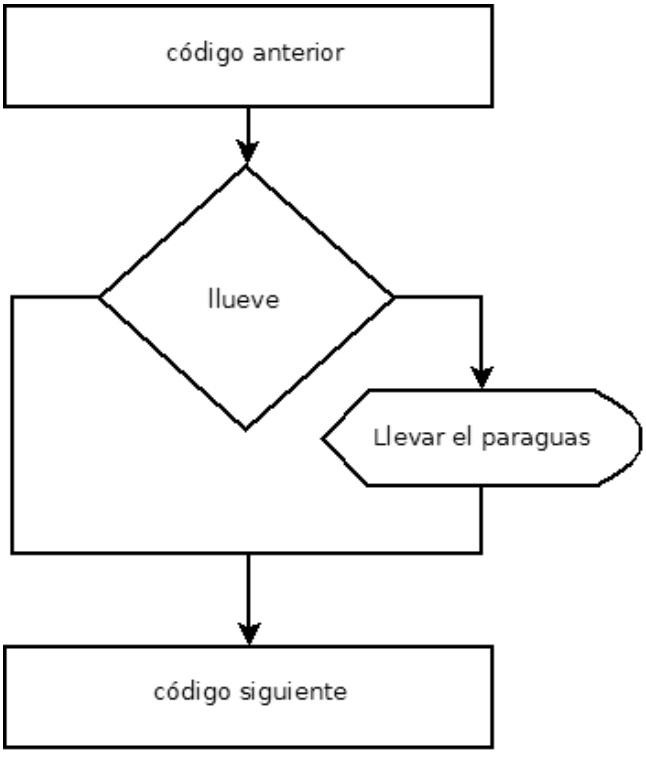


**Figura 6-1. Diagrama de un condicional simple.**

Esto significa que esas acciones envueltas entre las llaves del condicional, no siempre se realizarán (¡no salimos todos los días con el paraguas!), sino que esos bloques de código solo se ejecutarán cuando la condición que se evaluará sea verdadera (que llueva, en este caso).

De lo contrario, se seguirán ejecutando las órdenes que sigan a continuación.

Veamos un ejemplo en PHP:

```
<?php
if ($_POST["password"]== "superagente86") {
      echo "<h1>Maxwell, atienda el zapatófono que
      lo estamos llamando de Control</h1>";
}
?>
```

En este ejemplo, damos por hecho que desde otra página hemos enviado, mediante un formulario con *method="post"*, un input llamado *password*. Solamente, si dentro de ese campo hemos enviado el dato "superagente86", la condición será verdadera, y el echo escribirá el mensaje dentro de la página.

En caso contrario (que el usuario haya escrito cualquier otro texto), la condición será evaluada como falsa, y el bloque de código envuelto entre las llaves del **if** no se ejecutará, por lo tanto, el echo no escribirá nada. Ese echo, el código envuelto entre las llaves del **if**, es un código **cuya ejecución es condicional**; es decir, está sometida al resultado de evaluar si la condición es verdadera en el momento en que se procesa esta página.

Lo podemos probar para verlo en acción y también ingresar cualquier otro valor dentro del campo *password*; de esta manera, veremos cómo no se ejecuta el bloque condicional, y nos quedamos sin enterarnos del mensaje secreto.

# ¿Qué hacer si responde que no es verdad? El else y el elseif

## Else (si no)

A los condicionales que hemos visto, también les podemos agregar (opcionalmente) que se ejecute otro bloque de código en el caso de que la condición evaluada haya resultado falsa.

Para esos casos en los que no resultó verdadera la condición, contamos con el **else**. El **else nunca se puede ejecutar solo**, es apenas un complemento de un **if**.

El **else** convierte al **if** en una bifurcación entre dos caminos posibles, y decide automáticamente qué bloque de código ejecutar de entre **dos bloques posibles diferentes**. Dos bloques de código sujetos a una misma condición.

La estructura del **if** con un **else** es la siguiente:

```
if (condición a evaluar) {
      Bloque que se ejecuta solo si la condición
      resulta ser verdadera
} else {
      Bloque que se ejecuta solo si la condición es falsa
}
```

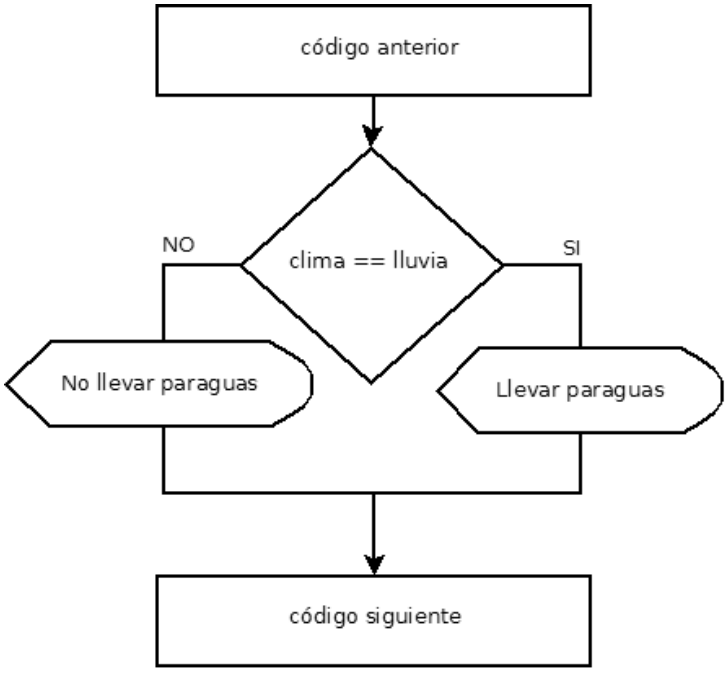


**Figura 6-2. Diagrama de un condicional con else.**

Una conclusión fundamental es que **jamás se ejecutan ambos bloques** en una misma ocasión. O se ejecuta el bloque inicial, el que está envuelto entre las llaves del **if**, o el bloque envuelto por las llaves del **else**, pero nunca ambos. O un bloque, o el otro. Es excluyente.

Esto es así porque la condición del **if no puede ser verdadera y falsa a la vez**, en un mismo momento. O es verdadera, o es falsa.

Si la condición es **verdadera**, se ejecuta el bloque de órdenes que está encerrado entre las primeras llaves, las del **if**, y no el bloque del **else**.

Si, en cambio, la condición es **falsa**, no se ejecuta el bloque del **if**, y solo se ejecuta el bloque delimitado entre las llaves del **else**. O uno, o el otro.

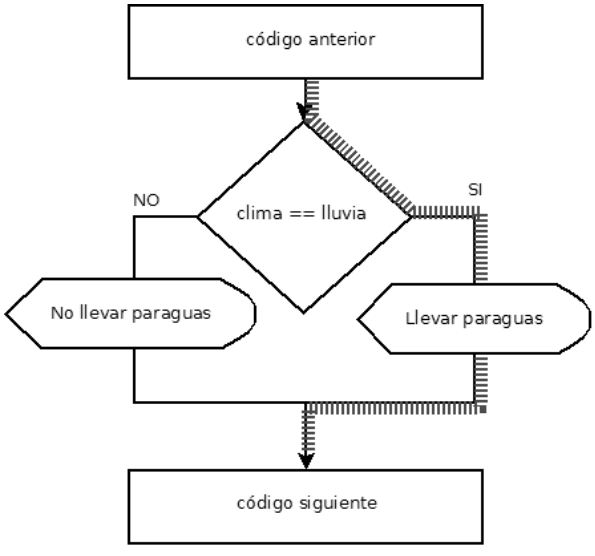


**Figura 6-3. Camino que se ejecuta si la condición es verdadera.**

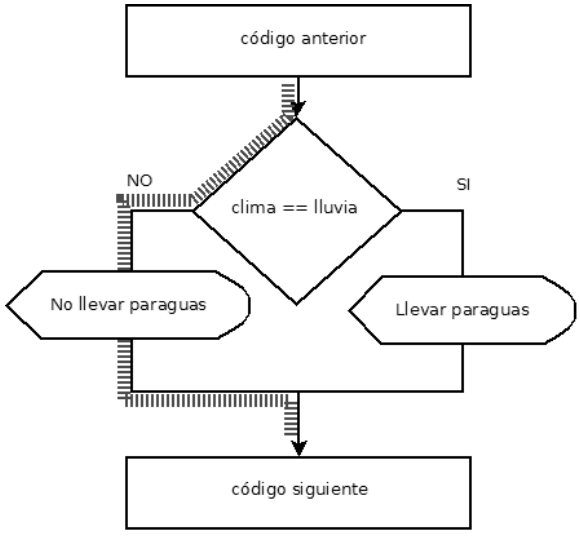


**Figura 6-4. Camino que se ejecuta si la condición es falsa.**

En nuestro ejemplo de la vida real anterior, podría ser así:

```
<?php
if ($clima == "lluvia") {
      echo "Llevar paraguas";
} else {
      echo "No llevar paraguas";
}
?>
```

Por supuesto, para que esto funcione, démosle previamente un valor a $clima, y probemos de cambiarlo; que a veces sea "lluvia", así vemos cómo se ejecuta el bloque verdadero, y que a veces sea otra cosa, así se ejecuta el bloque del else.

Notemos que, para mayor claridad, a los bloques de instrucciones que hay que ejecutar –lo que está **dentro de las llaves** del if o del else–, les dejamos un **tabulador**, o **3 o 4 espacios de sangría** a su izquierda, para facilitar su lectura.

Veamos otro caso real. Vamos a hacer un código bien "educado", que salude a las personas según su sexo.

Crearemos el archivo **elegir.html**, que consta del siguiente formulario (por supuesto, agregaremos la DTD, etiquetas html, head, body, etc.):

```
<form action="saludar.php" method="post">
      <select name="sexo">
         <option value="masculino">Masculino</option>
         <option value="femenino">Femenino</option>
      </select>
      <input type="submit" value="Enviar">
</form>
```

Por otro lado, recibimos el valor de la variable "sexo" en el archivo denominado **saludar. php**, y según lo que haya elegido el usuario, le mostraremos uno u otro saludo:

Código de saludar.php:

```
<?php
if ($_POST["sexo"] == "masculino"){
      print ("¡Hola hombre!");
} else {
      print ("¡Hola mujer!");
}
?>
```

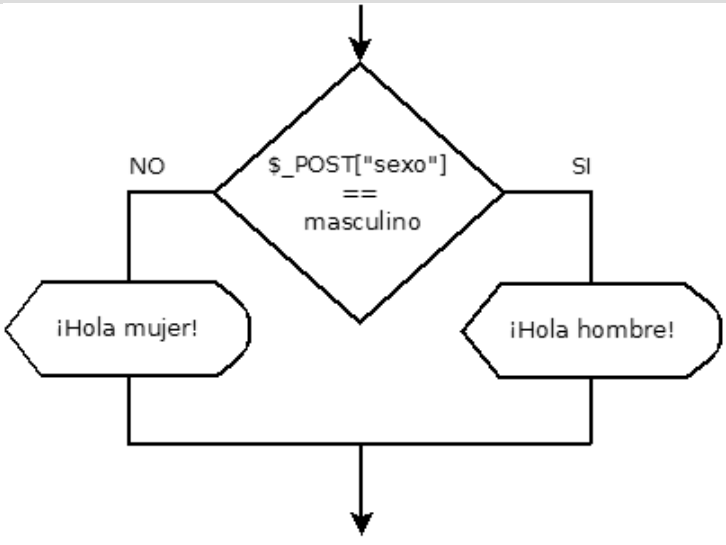


**Figura 6-5. Diagrama del ejemplo anterior.**

Analicemos el primer elemento de un condicional que, como su nombre lo indica, es la **condición** que se evaluará. Dentro de los paréntesis, al lado de la palabra **if**, hacemos una afirmación, que tomará la forma de una **comparación**, que solo se evaluará como **verdadera** o **falsa** (se las conoce en programación como condiciones *booleanas*). En este caso, lo que queremos saber es si **el valor actual de una variable** (en este ejemplo, la variable "sexo") es exactamente igual a "masculino".

Por eso, hemos escrito *if ($_POST["sexo"]* **==** *"masculino")* y, al haber colocado un **doble signo igual**, solamente estamos preguntando, sin "asignarle" un valor a la variable. El **doble signo igual** es el **operador de comparación**, que nos permite averiguar si dos cosas son iguales.

Si hubiéramos escrito *$_POST["sexo"]* **=** *"masculino"* (con **un solo** signo igual, que es el operador de **asignación**), todas las veces el if hubiera resultado

verdadero, ya que ese signo igual único le está asignando a la variable $_POST[“sexo”] el valor “masculino”; por lo tanto, la condición se interpretará de otra manera, ya que se evaluará si “pudo realizarse la asignación”, si se pudo guardar dentro de $_POST[“sexo”] el valor “masculino”, y como esto no suele fallar, siempre será verdadera esa condición.

## Elseif (si no, si...)

### Varias condiciones diferentes: el elseif

Hemos visto que la condición que se evaluará debe ser del tipo *booleana* –que significa que **solo** puede dar como respuesta a la pregunta un **verdadero** o un **falso**. En los casos en los que necesitemos plantear una serie de más de dos condiciones posibles, el **if** con un **else** ya no nos alcanzaría para contemplar todas las posibilidades. Necesitamos algo más para unir más de dos condiciones. Ese “algo más” es el **elseif**, que plantea una nueva condición que solo se ejecuta si la condición anterior no fue verdadera; es decir, es parte del **else** de la condición anterior y, a la vez, plantea una nueva condición. El **elseif**, de la misma manera que el else, no se puede ejecutar por sí solo, es una “segunda parte” de un condicional simple anterior (es la continuación de un **if** inicial).

```
<?php
if ($_POST["edad"] < 18){
      print ("¡Hola niño!");
} elseif ($_POST["edad"] < 30){
      print ("¡Hola joven!");
} elseif ($_POST["edad"] > 29){
      print ("¡Hola adulto!");
}
?>
```

Hemos creado tres condiciones excluyentes (que pudieron ser muchas más). Veamos otro ejemplo: esta vez, de una sucesión de un **if** inicial, un **elseif**, y un **else** final:

```
<?php
if ($_POST["sexo"] == "masculino"){
      print ("¡Hola Hombre!");
} elseif ($_POST["sexo"] == "femenino"){
```

```
      print ("¡Hola Mujer!");
} else {
      print ("Hola...");
}
?>
```

Como podemos observar, el **elseif** no solo cierra el **if** anterior, sino que abre uno nuevo y **plantea una nueva condición**, distinta, independiente de la anterior, pero con la particularidad de que será evaluada solamente en el caso de que la anterior condición hubiera resultado ser **falsa**. De lo contrario, si la anterior condición era **verdadera**, el programa ni se toma el trabajo de evaluar esta segunda condición... salta directamente hasta después de la llave de cierre del **if**, sin evaluar ninguna condición más.

De esta forma, pueden encadenarse muchas alternativas que requieran, para ser evaluadas, que una condición anterior hubiese resultado falsa.

Un dato muy interesante es que las condiciones sucesivas no tienen por qué evaluar el valor de una misma variable, lo cual nos da la libertad de ir evaluando **distintas** cosas en cada condición:

```
<?php
if ($_POST["sexo"] == "masculino"){
      print ("¡Hola Hombre!");
} elseif ($_POST["estado"] == "soltera"){
      print ("¡Hola Mujer soltera!");
} elseif ($_POST["edad"] > 70){
      print ("¡Hola abuela!");
}
?>
```

En este caso, como son dos las posibles respuestas de la primera condición, pudimos plantear una segunda condición que no vuelva a evaluar otra vez la misma variable "sexo", sino que evalúa otra cosa, en este caso, "estado". Y, luego, plantearemos otra condición con la "edad", solo en el caso de que la anterior condición no se hubiese ejecutado. Es importante tener en claro que solamente en el caso de que haya sido falsa la condición anterior, se ejecuta la siguiente. Si una de las condiciones es verdadera, ya no se ejecuta el resto de las condiciones.

En el caso de tener más de dos posibilidades que evalúan el valor de **una misma variable**, veremos que es mucho más práctico aplicar una estructura diferente, que es la del **switch** y los **case.** Veamos un ejemplo:

```
<?php
if ($dia == "lunes"){
      print ("¡Feliz día de la Luna!");
} elseif ($dia == "martes"){
      print ("¡Feliz día de Marte!");
} elseif ($dia == "miércoles"){
      print ("¡Feliz día de Mercurio!");
} elseif ($dia == "jueves"){
      print ("¡Feliz día de Júpiter!"
} elseif ($dia == "viernes"){
      print ("¡Feliz día de Venus!");
} elseif ($dia == "sábado"){
      print ("¡Feliz día de Saturno!"
} elseif ($dia == "domingo"){
      print ("¡Feliz día del Sol!");
}
?>
```

En este caso, en el que evaluamos una serie de valores posibles de una única variable, es mucho más breve y simple utilizar una estructura selectiva, como el **switch**, que aprenderemos a usar a continuación.

## Elegir entre valores conocidos de una única variable: el switch

De la estructura condicional denominada **switch**, se dice que es "selectiva", ya que selecciona uno de varios posibles caminos según el valor de una variable:

Para programar un **switch**, se coloca entre paréntesis, al lado de la palabra **switch**, la **variable** o **celda de matriz** cuyos valores posibles conocemos y queremos averiguar, en cada ejecución de la página, qué valor tienen para actuar en consecuencia.

Veamos este otro ejemplo, que evalúa una variable llamada $dia (a la que se espera que le hayamos dado como valor uno de los días de la semana):

```
<?php
switch ($dia){
// Aquí evaluaremos los posibles valores de $dia
}
?>
```

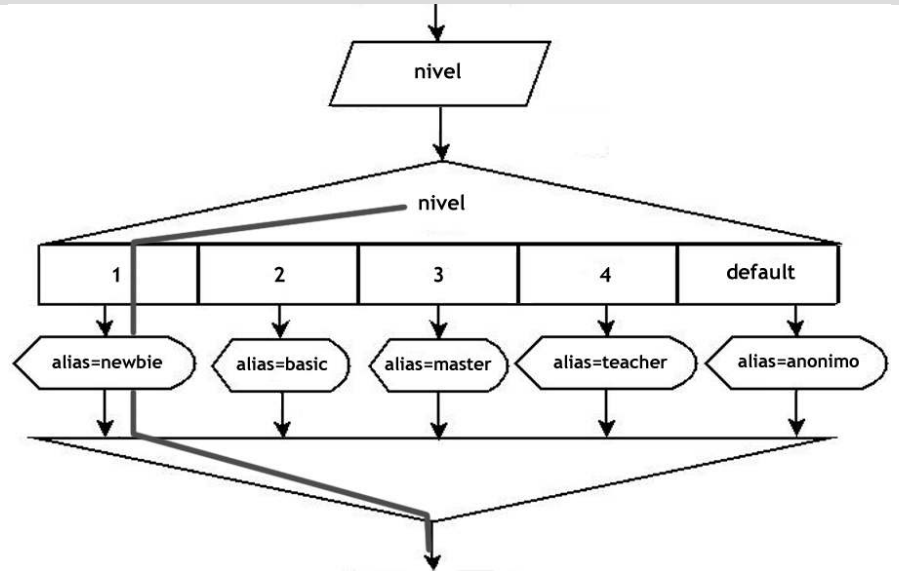


**Figura 6-6. Diagrama de un switch con sus caminos excluyentes.**

### Case (cada posible valor)

Dentro de las llaves del **switch**, a continuación de la variable a evaluar, se repiten una serie de bloques **case** (un "caso" posible, un valor posible de esa variable) **tantas veces como valores posibles** esperemos que tenga la variable examinada. Es decir, un *case* por cada valor.

Como solo uno de esos casos será ejecutado, dentro de él, simplemente hemos definido un valor para la variable $texto y, al final, mostramos esa variable, cuyo contenido –y, por ende, el texto que se mostrará– habrá tomado uno u otro valor, según lo evaluado en el **switch**, de acuerdo con cuál caso resultó ser verdadero.

Agreguemos los casos al ejemplo:

```
<?php
```

```
switch ($dia){
      case "lunes":
      $texto = "¡Feliz día de la Luna!";
      break;

      case "martes":
      $texto = "¡Feliz día de Marte!";
      break;

      case "miércoles":
      $texto = "¡Feliz día de Mercurio!";
      break;

      case "jueves":
      $texto = "¡Feliz día de Júpiter!";
      break;

      case "viernes":
      $texto = "¡Feliz día de Venus!";
      break;

      case "sábado":
      $texto = "¡Feliz día de Saturno!";
      break;

      case "domingo":
      $texto = "¡Feliz día del Sol!";
      break;
}
print ($texto);
?>
```

Notemos la sintaxis: luego de la palabra *case*, envolvemos entre comillas el valor de la variable, y luego colocamos dos puntos. Todas las órdenes de lenguaje PHP que vengan a continuación, hasta la palabra "*break*", se ejecutarán solo si ese es el valor de la variable $dia evaluada.

Como vemos, esta estructura solamente nos sirve en caso de que conozcamos **las posibles alternativas excluyentes** (todos los valores posibles de la variable que se evaluará).

### Default (valor por omisión)

Adicionalmente, podemos ofrecer una alternativa **por defecto** (si bien es opcional, y podría no ser incluida, es mucho mejor incluir siempre una acción por omisión).

Los *case* del ejemplo anterior no tienen prevista ninguna acción para el caso en que $dia tenga un valor distinto a los siete valores definidos.

Para evitar el riesgo de este "vacío" lógico, se utiliza en el último lugar de la sucesión de *case*, mejor dicho, **después** del último **case**, la expresión *default* (por defecto). El código envuelto en ese bloque, se ejecutará **en cualquier otro caso** en que la variable evaluada contenga cualquier otra cosa no prevista en ninguno de los *case* anteriores:

```
<?php
switch ($dia){
      case "lunes":
      $texto = "¡Feliz día de la Luna!";
      break;

      case "martes":
      $texto = "¡Feliz día de Marte!";
      break;

      case "miércoles":
      $texto = "¡Feliz día de Mercurio!";
      break;

      case "jueves":
      $texto = "¡Feliz día de Júpiter!";
      break;

      case "viernes":
      $texto = "¡Feliz día de Venus!";
      break;

      case "sábado":
```

```
        $texto = "¡Feliz día de Saturno!";
        break;

        case "domingo":
        $texto = "¡Feliz día del Sol!";
        break;

        default:
        $texto = "¡Feliz día fuera de calendario!";
        break;
}
print ($texto);
```

Podemos probar cargando distintos valores dentro de la variable $dia para ver cómo se ejecuta uno u otro **case**, o el **default**.

### Break (salir del switch)

Cada *case* se cierra con la instrucción **break** que, al ser ejecutada, nos hace **salir inmediatamente de la ejecución del switch**; si no estuviera ese *break* al final de cada caso, una vez encontrado un *case* verdadero y ejecutadas sus sentencias, se seguirían ejecutando innecesariamente las preguntas de los restantes *case*, perdiendo tiempo sin necesidad. Pero lo que es peor, **se ejecutaría el *default*** (que no tendría por qué ejecutarse si ya se ejecutó un *case* verdadero, recordemos que el sentido del *default* es que se ejecute solamente si todos los *case* resultaron falsos).

Ahora que conocemos las estructuras condicionales (if, else, elseif) y selectivas (switch), vamos a profundizar las herramientas complementarias de esas estructuras, que nos permitirán validar los datos de nuestros formularios con la mayor precisión. Las principales herramientas complementarias son los **operadores de comparación** y los **operadores lógicos**, que aprenderemos a continuación.

## Operadores de comparación y lógicos

### Operadores de comparación

Cuando planteamos una condición que debe ser evaluada por un condicional, sabemos que debe ser *booleana*, es decir, que se pueda evaluar únicamente como “verdadera” o “falsa”. No podemos poner como condición “¿qué hora es?”,

pero sí podemos poner como condición "son las cinco" (en PHP, sería algo parecido a: $hora==5).

Una condición es una **afirmación**. Y esa afirmación, generalmente, toma la forma de una **comparación**. Comparamos una variable, un dato conocido, contra un valor "probable" esperado, que posiblemente sea el valor que haya tomado esa variable. De esa manera, la condición no es más que una comparación entre el **valor real** que le fue dado a una variable, y uno de los **valores "supuestos" o "posibles"** de esa variable imaginado por nosotros, los programadores.

Pero para poder comparar de una forma verdaderamente útil y completa, debemos conocer los **operadores de comparación** con que contamos en PHP. Hasta ahora, solamente hemos usado el más simple de los operadores de comparación, el **==** (igual a), pero, en muchas ocasiones, necesitaremos plantear la condición no en términos de igualdad, sino que precisaremos saber si un número es mayor que otro, si es menor, si un texto es distinto de otro, etc. para que, según sea cierta, o no, esa condición, se ejecute uno u otro bloque de código.

Veamos, entonces, la lista de posibles operadores de comparación, para que podamos usarlos al elaborar condiciones complejas:

| Operador | Nombre | Ejemplo | Devuelve verdadero cuando... |
|---|---|---|---|
| = | Igual a | $x == $y | $x es igual a $y |
| != | Distinto de | $x != $y | $x es distinto de $y |
| <> | Distinto de | $x <> $y | $x es distinto de $y |
| < | Menor que | $x < $y | $x es menor que $y |
| > | Mayor que | $x > $y | $x es mayor que $y |
| <= | Menor o igual | $x <= $y | $x es menor o igual que $y |
| >= | Mayor o igual | $x >= $y | $x es mayor o igual que $y |

**Tabla 6-1. Operadores de comparación.**

Veamos algunos ejemplos de estos operadores utilizados en condiciones reales. Si queremos comparar si un valor ingresado es menor a un número concreto, tenemos dos maneras: usando el operador **<** o usando el operador **<=**:

```
<?php
if ($_POST["edad"] < 18){
      print ("Es menor a 18 años");
}
if ($_POST["edad"] <= 17){
      print ("Es menor a 18 años");
```

```
}
?>
```

Si queremos saber si algo es distinto a otra cosa, también tenemos dos maneras de compararlo: con el operador **<>** o con **!=**:

```
<?php
if ($_POST["nombre"] <> "Pepe"){
      print ("No sos Pepe");
}
if ($_POST["nombre"] != "Pepe"){
      print ("No sos Pepe");
}
?>
```

### Operadores lógicos:

En otros casos, necesitaremos crear condiciones complejas, que requieran **combinar (unir) dos o más condiciones simples en una**, para evaluarlas como una sola condición. Al elemento que permite combinarlas lo llamamos **operador lógico**.

Veamos un ejemplo simple aún sin entrar en detalles, para tratar de comprender el concepto. Supongamos que tenemos que averiguar si la edad de un usuario está entre 18 y 65 años para mostrarle un texto especial. Tienen que cumplirse **dos condiciones**: la primera es que su edad sea mayor a 17 (o "mayor o igual" a 18), y la segunda es que su edad sea menor a 66 (o "menor o igual" a 65):

```
$_POST["edad"] > 17
```

y

```
$_POST["edad"] < 66
```

Para evaluar con un solo condicional estas dos condiciones, necesitamos "unirlas", y convertirlas en **una sola** condición que sea evaluada como verdadera (si es mayor a 17 y, además, menor a 66) o falsa, en caso contrario. Eso que se usa para unir las dos condiciones en una sola, es lo que denominamos un **operador lógico**. En este ejemplo, el operador necesario es and (significa y, que es el nombre del operador de conjunción). Para que devuelva verdadera la expresión completa, necesitamos que sea verdadera la primera condición y que además sea verdadera la segunda.

En este caso, le pediríamos que ingrese la edad en un campo de texto (supongamos que el *name* de esa campo sea "edad") y la evaluaríamos de la siguiente manera (de paso, observemos que hemos redactado cada condición con otros operadores de comparación que en el caso anterior, esta vez usando **>=** y **<=** y modificando el valor que se comparará).

```
if ($_POST["edad"] >= 18 and $_POST["edad"] <= 65){

      print ("Está en el rango solicitado");

} else {
      print ("Fuera del rango");
}
```

Notemos que combinamos dos condiciones completas en una sola, incluso cada una de ellas podría haberse colocado por separado, en un if distinto, individualmente:

```
$_POST["edad"] >= 18
```

y

```
$_POST["edad"] <= 65
```

Lo único que hemos hecho es unirlas con el operador **and.** Por supuesto, podemos unir más de dos condiciones, usando un mismo operador o distintos. Imaginemos que necesitamos evaluar si la edad es mayor a 17 y menor a 66, o el estado civil es "viudo"; las tres condiciones por separado serían:

```
$_POST["edad"] >= 18
```

y

```
$_POST["edad"] <= 65
```

o

```
$_POST["estado"] == "viudo"
```

Al unirlo, usaríamos esta sintaxis:

```
if ($_POST["edad"] >= 18 and $_POST["edad"] <= 65 or $_
POST["estado"] == "viudo"){
```

```
      print ("Está en el rango solicitado");
}
```

En este caso, primero se evaluarían las condiciones unidas mediante el operador **and** (que sea mayor a 17 y menor a 66) y si eso es verdad, ya es suficiente para que toda la expresión se considere verdadera. Del mismo modo, si esa primera parte fue falsa, pero el estado era "viudo", al estar unidas por el operador **or**, devuelve verdadero, ya que ese operador necesita que al menos una de las condiciones sea cierta, sin importar si otras no lo son. Podemos unir cualquier cantidad de condiciones, variando los operadores libremente y creando una única condición que será evaluada en conjunto.

### Tablas de verdad:

Todos los operadores lógicos evalúan el "valor de verdad" (es decir, si es verdadera o falsa) de cada una de las condiciones por separado y, luego, según el operador utilizado para unir estas condiciones, devuelven **un único valor** de verdadero o falso para el conjunto completo de condiciones que formaban esa expresión.

Vamos a analizar cuatro operadores lógicos:

- **and** (conjunción, también puede escribirse en PHP como **&&**),
- **or** (disyunción, también puede escribirse en PHP como **||**),
- **xor** (disyunción excluyente, informalmente llamado "or excluyente"),
- **!** (negación).

Si a las dos condiciones que uniremos mediante operadores lógicos les damos un nombre simbólico y las denominamos **$x** e **$y**, los valores de verdad pueden ser solamente **cuatro**:

- que ambas condiciones sean verdaderas,
- que ambas condiciones sean falsas,
- que la primera sea verdadera y la segunda falsa,
- o que la primera sea falsa y la segunda verdadera. No hay otra alternativa.

Estas cuatro situaciones posibles pueden graficarse con esta tabla:

| $x | $y |
|---|---|
| V | V |
| F | F |
| V | F |
| F | V |

(Siendo “V” verdadero, y “F” falso).

Dependiendo de cuál operador utilicemos para unir ambas condiciones, obtendremos distintos resultados al evaluar el conjunto completo.

Veamos cuál es la “tabla de verdad” para unir condiciones, según cada uno de estos operadores lógicos.

**Conjunción:**

| $x | $y | $x y $y |
|---|---|---|
| V | V | V |
| V | F | F |
| F | V | F |
| F | F | F |

El operador de conjunción es un operador lógico bastante restrictivo, ya que requiere que todas las condiciones que forman parte de la expresión sean verdaderas para que devuelva verdadera la condición completa.

En cualquier otro caso, es decir, con que solo una de las condiciones sea falsa, la expresión completa se evalúa como falsa.

Veamos un ejemplo con PHP. Supongamos que disponemos de estos datos:

```
$usuario = "pepe";

$clave = 123456;
```

Y que queremos validarlos con un condicional que evalúe si el nombre es “pepe” **y además** la clave es “123456”. Necesitamos que **ambos** sean verdaderos para dar por verdadera la condición entera y darle acceso a ese usuario, entonces, usaremos un operador de **conjunción**:

```
if ($nombre == "pepe" and $clave == "123456"){

      print ("¡Bienvenido Pepe!");
```

```
} else {
      print ("No lo conocemos, disculpe pero queda fuera");
}
```

Es recomendable que probemos qué sucede cuando proporcionamos valores "falsos" (distintos a los esperados en la condición). Por ejemplo, podríamos cambiar los valores de esas dos variables de esta manera:

```
$usuario = "juan";
$clave = 123456;
```

En este caso, se evaluaría como **falsa** la condición, ya que la primera condición (que $nombre valga "pepe") se evalúa como falsa, así que, por más que la segunda sea verdadera, al haber unido a ambas con un operador de conjunción and, la condición entera se evalúa como falsa.

Del mismo modo, podríamos volver a cambiar los valores de esas dos variables, esta vez a algo como:

```
$usuario = "pepe";
$clave = 99999;
```

Y también en este caso, se evaluaría como **falsa** la condición, ya que la primera condición es verdadera, pero la segunda es falsa.

Desde ya que también se evaluaría como falsa si ambas partes fueran falsas (tanto el nombre como la clave).

### Disyunción

| $x | $y | $x y $y |
|---|---|---|
| V | V | V |
| V | F | V |
| F | V | V |
| F | F | F |

El operador de disyunción, contrariamente al anterior, es un operador lógico bastante amplio, ya que considera que la expresión es verdadera si **al menos una** de las condiciones que forman parte de la expresión es verdadera.

La única forma de que una disyunción resulte falsa es que todas las condiciones sean falsas, sin excepción.

Veamos un ejemplo con PHP. Imaginemos que un sitio de comercio electrónico realiza envíos a domicilio solamente en tres ciudades: Ciudad 1, Ciudad 2 y Ciudad 3; solicitamos al usuario que ingrese su ciudad de residencia en un campo llamado "ciudad", y evaluamos ese dato. Necesitamos que **al menos uno** de esos tres nombres de ciudad sea el que verdaderamente ingresó el usuario para dar por verdadera la condición entera, y realizar el pedido. Entonces, usaremos un operador de **disyunción**:

```
if ($_POST["ciudad"] == "Ciudad 1" or $_POST["ciudad"] ==
"Ciudad 2" or $_POST["ciudad"] == "Ciudad 3"){

      print ("¡Zona correcta! Recibirá su pedido.");

} else {
      print ("Está fuera del área de cobertura.");
}
```

Nuevamente, es recomendable que probemos cambiando los valores esperados, y que veamos qué sucede cuando proporcionamos valores falsos o uno de los verdaderos. En el caso de una disyunción, con que proporcionemos uno solo de los valores correctos, se evaluará como correcta la expresión.

### Disyunción excluyente:

| $x | $y | $x y $y |
|---|---|---|
| V | V | F |
| V | F | V |
| F | V | V |
| F | F | F |

La diferencia con el operador de disyunción común es que si ambas condiciones son verdaderas, devuelve falso.

Supongamos que disponemos de estos datos proporcionados en un pedido:

```
$tarjeta = "VISA";

$cupon = 19876;
```

Y que queremos validarlos con la regla de que si paga con tarjeta VISA, no se considerará ningún cupón de descuento. Por el contrario, si ingresa un cupón de descuento, la forma de pago no puede ser VISA. Son excluyentes:

```
if ($tarjeta == "VISA" xor $cupon <> ""){

      print ("Podemos tomar su pedido");

} else {

      print ("No puede elegir VISA y a la vez colocar un
      código de cupón de descuento, y tampoco puede elegir
      otro medio de pago sin ingresar un código de
      cupón.");

}
```

**Negación:**

| $x | !$x |
|---|---|
| V | F |
| F | V |

El operador de negación transforma la expresión en su contrario. Es decir, si la condición es verdadera, devuelve falso, y si la condición es falsa, devuelve verdadero.

Ya hemos utilizado este operador en otros casos, como por ejemplo en:

```
if ($_POST["nombre"] != "Pepe"{
       print ("No sos Pepe");
}
```

Devuelve verdadero si nombre **no es** igual a “Pepe”.

Resumamos, entonces, en un cuadro, los distintos operadores lógicos:

| Operador | Nombre | Ejemplo | Devuelve verdadero cuando: |
|---|---|---|---|
| **and** | Conjunción "y" | $x and $y | $x es verdadero e $y también es verdadero (deben ser **ambos** verdaderos para que devuelva verdadero, con que uno solo sea falso, devuelve falso) |
| **&&** | Conjunción "y" | $x && $y | $x es verdadero e $y también es verdadero (deben ser **ambos** verdaderos para que devuelva verdadero, con que uno solo sea falso, devuelve falso) |
| **or** | Disyunción "o" | $x or $y | $x es verdadero, o $y es verdadero (con que **uno solo sea verdadero**, devuelve verdadero) |
| **\|\|** | Disyunción "o" | $x \|\| $y | $x es verdadero, o $y es verdadero (con que **uno solo sea verdadero**, devuelve verdadero) |
| **xor** | Disyunción excluyente | $x xor $y | O bien $x es verdadero, o bien $y es verdadero (uno de ellos), pero **no ambos**. (si ambos son verdaderos, devuelve falso) |
| **!** | Negación | !$x | Devuelve verdadero si $x **no es verdadero** |

**Tabla 6-2. Operadores lógicos.**

## Tipos de validaciones

### ¿Está presente cada dato requerido?

En este "diálogo en etapas" que implican las páginas dinámicas, este "estar esperando" en una segunda página un dato que el usuario debería haber enviado hacia el servidor desde una página anterior (ya sea con un enlace que envía variables, o con un formulario), puede suceder que el dato esperado **nunca llegue**, con lo cual se frustraría el intento de usarlo.

Volvamos a analizar el código de un ejemplo anterior: supongamos que tenemos una página llamada formulario.html que contiene esto:

```
<form action="muestra.php" method="post">
```

```
      <input type="text" name="domicilio">
      <input type="submit" value="Enviar">
</form>
```

Y también tenemos una página llamada muestra.php, que es la página en la que estaremos esperando la variable "domicilio", para usarla:

```
<?php
print ("Su direccion es: ");
print ($_POST["domicilio"]);
?>
```

Este esquema está dando por supuesta una situación ideal, una forma esperada de navegar por estas dos páginas, que no necesariamente será el camino que seguirá todo usuario. Damos por supuesto lo siguiente:

1. que el usuario **ingresará primero** a formulario.html,
2. que luego **escribirá algo** en el campo llamado "domicilio",
3. que a continuación **pulsará el botón** de **Enviar**,
4. que recibirá instantes después como respuesta el resultado del procesamiento de la página muestra.php, que **le mostrará su domicilio**.

Posiblemente, ése sea el comportamiento de la mayoría de los usuarios, pero... ¿y qué pasará con las excepciones que no sigan ese camino ideal?

¿Qué pasaría si un usuario que ya utilizó estas dos páginas y conoce las dos URL (o las guardó en sus Favoritos), entra directamente a muestra.php, **sin haber pasado antes por el formulario**? Es decir, si entra directamente al que suponíamos que iba a ser el "segundo paso" de la serie de pasos, salteándose el primero.

Lo que se obtendrá será un mensaje de error similar a éste:

```
Su domicilio es:

Notice: Undefined index: domicilio in... etc.
```

Es decir, no se encuentra definida ninguna celda llamada "domicilio" dentro de la matriz $_POST que se está intentando leer, en el archivo de destino, cuando ordenamos que el intérprete de PHP muestre su valor con la orden **print($_POST["domicilio"]);**.

De más está decir que los mensajes de error del intérprete de PHP son algo que jamás debería aparecer en nuestras páginas: si hemos programado bien nuestro código, podríamos evitar por completo esos errores.

Evitar estos errores es muy simple, es cuestión de anticiparse a la posibilidad de ausencia de ese dato esperado, verificando primero **si está** presente el dato. Solo si está presente, procederemos a usarlo y, en caso contrario, mostraremos alguna indicación para el usuario (por ejemplo, un enlace hacia el formulario que debió haber utilizado).

El problema en esta situación es que se está intentando leer la celda **$_POST["domicilio"]**. Y, al no haber pasado antes por el formulario, esa celda directamente no existe. Y no se puede leer algo que no existe...

Para **anticiparnos** a la posibilidad de ausencia de un dato esperado, usaremos un condicional, que verifique si está definida la variable o celda de matriz que estamos a punto de utilizar; si está presente, en ese caso haremos uso de ella.

Para verificar si una variable o celda de matriz se encuentra definida, usaremos la sintaxis siguiente:

```
<?php
if ( isset($_POST["domicilio"]) ){

      print ("<p>Su direccion es: ");
      print($_POST["domicilio"]);
      print ("</p>");
} else {
      print ("<p>Oops! Parece que no pasó por el
            <a href=\"formulario.html\">formulario</a>.
            Por favor, pase por él y envíenos su
            domicilio.</p>");
}
?>
```

Notemos que hemos previsto las dos posibilidades:

- que esté presente, es decir que haya sido enviado el dato "domicilio" (en cuyo caso, lo usamos), o
- que no haya sido proporcionado, en cuyo caso solicitamos al usuario que pase por el formulario y lo utilice.

Siempre utilizaremos esta validación cuando estemos **esperando** que el usuario proporcione un dato, ya que es posible que éste no nos llegue. Por esta razón, debemos anticiparnos a esa posibilidad para no terminar en un mensaje de error.

Si queremos validar de una sola vez la llegada al servidor de **más de una** variable, podemos unir con comas los nombres de esas variables (o celdas de matrices) dentro del *isset*:

```
<?php
if ( isset($_POST["nombre"], $_POST["apellido"],
$_POST["edad"]) ){
/* Solo si todos esos datos están presentes, devuelve
verdadero y se ejecuta lo que esté entre las llaves del if
*/
}
```

También, podríamos unir distintos *isset* con operadores **lógicos**:

```
<?php
if ( isset($_POST["nombre"]) and isset($_POST["apellido"])
and isset($_POST["edad"]) ){
/* solo si todos esos datos están presentes, devuelve
verdadero y se ejecuta lo que esté entre las llaves del if
*/
}
```

La validación de "si está definido" (si está presente, si fue enviado) un dato, es la más necesaria de todas, y la usaremos siempre que esperemos recibir un dato del lado del usuario y que, por lo tanto, pudiera llegar a suceder que no sea proporcionado, que no esté presente.

### Si no está vacío

Otra posibilidad diferente (aunque parecería ser igual a la anterior, no lo es) es que el usuario sí pase por el formulario, pero lo envíe hacia el servidor **sin haber escrito nada** dentro del campo. Es decir, el dato esperado sí estará "definido", habrá llegado al servidor, pero con un valor de "nada", totalmente **vacío**.

La diferencia entre "estar definido" y "estar vacío" es similar a la diferencia entre el hecho de que haya llegado una carta a nuestro buzón de correo, y que la carta esté vacía.

- El **no estar definida** una variable equivale a que **no nos llegó** ninguna carta.
- **Estar definida** equivale a que llegó la carta (pero todavía puede ser que esté llena o vacía, hasta que no la abramos no lo sabremos).
- Estar **vacía** da por sentado que **llegó** (está definida), pero **sin contenido**.
- Estar **llena** es la situación ideal: **llegó** (está definida) y **tiene contenido** (no está vacía).

Validar esta posibilidad es viable mediante los operadores de **comparación** que hemos utilizado en otras ocasiones. Según la forma en la que expresemos la condición, podemos utilizar distintos operadores de comparación. Podemos querer saber si algo **está** vacío y, en ese caso, escribiríamos así la condición:

```
<?php
if ($_POST["domicilio"] == ""){
      // si es cierto, es que está vacío
}
?>
```

Y, en otros casos, si queremos saber si algo **no está** vacío, podemos usar tanto el operador **<>** como el operador **!=**, cualquiera de los dos:

```
<?php
if ($_POST["domicilio"] <> ""){
      // si es cierto, es que NO está vacío
}
?>
```

O lo que es igual:

```
<?php
if ($_POST["domicilio"] != ""){
      // si es cierto, es que NO está vacío
}
?>
```

Notemos que hemos utilizado el operador de negación **!** para que la condición sea verdadera si "no es igual" a vacío (y expresamos el valor vacío

mediante “nada” entre comillas, literalmente nada, ni siquiera un espacio vacío entre la apertura y el cierre de las comillas).

De esta manera, sabremos si un dato ha sido llenado con **algún valor** por el usuario antes de utilizarlo en nuestro código.

Aplicado al ejemplo anterior, podríamos combinar **ambas** validaciones en una sola (que esté presente el dato y que no esté vacío) mediante un operador lógico que una ambas condiciones:

```
<?php

if ( isset($_POST["domicilio"]) and $_POST["domicilio"]
!="" ){
      print ("<p>Su direccion es: ");
      print ($_POST["domicilio"]);
      print ("</p>");
} else {
      print ("<p>Oops! Parece que  no pasó por el
      <a href=\"formulario.html\">formulario</a>.
      Por favor, ¨pase y escriba su domicilio.</p>");
}
?>
```

### Si su valor está dentro de un rango

Otras veces necesitaremos comparar valores (generalmente numéricos) para saber si están dentro de un **rango de valores** posibles. Por ejemplo, si la variable “mes” de un formulario de un calendario, es un número mayor a cero y menor a 13 (los doce meses del año):

```
<?php
$mes = 10; // esto podría llegar desde un formulario

if ($mes > 0 and $mes <= 12){
      print ("<p>Usted eligió el mes: $mes</p>");
} else {
      print ("<p>Ha elegido un mes inexistente</p>");
}
?>
```

Es importante que observemos cómo hemos usado el **operador lógico and** para definir un rango mediante la unión de dos condiciones diferentes: que $mes sea mayor a cero (primera condición), y que $mes sea menor o igual a 12 (segunda condición). Con esos dos límites, tenemos definido un **rango** de valores permitido. Si ambas condiciones son ciertas, se ejecuta el código envuelto entre las llaves del if; en caso contrario, se ejecuta lo envuelto entre las llaves del else.

Si en lugar de un único rango, se tratara de una **serie de rangos** de valores, deberíamos utilizar una serie de elseifs:

```
<?php
$total = 145; // esto podría llegar desde un formulario

if ($total > 0 and $total <= 500){

      if ($total <= 100){

          print ("¡Muy barato! Menos de 100 pesos...");

      } elseif ($total <= 200){

          print ("Buen precio, entre 101 y 200 pesos.");

      } elseif ($total <= 500){

          print ("Algo caro, entre 201 y 500 pesos.");

      }
} else {

      print ("El valor está fuera del rango permitido de 1
              a 500");

}
?>
```

Notemos cómo estamos "anidando" distintos **niveles** de condiciones: la primera condición define el rango posible entre 1 y 500, y solo en caso de estar dentro de ese rango, se pasa al siguiente nivel de validación (para evaluar exactamente en cuál franja de precios se ubica). Esa primera condición tiene un

*else* que comunica un mensaje al usuario en caso de haber ingresado un total fuera del rango permitido.

Pero, una vez verificado que el valor de $total está entre 1 y 500, se procede a una serie de validaciones excluyentes, conectadas, de las cuales solo uno de los rangos será verdadero: o es menor a 100, o está entre 101 y 200, o está entre 201 y 500. No hay otra posibilidad.

Llegados a este punto, ya estamos en condiciones de hacer validaciones completas a todos los datos que esperamos que el usuario envíe hacia el servidor.

# Identificación con cookies y sesiones

# 7

## Cookies: datos que identifican a un navegador

Las *cookie*s son simples archivos de texto plano (sin formato, sin ninguna decoración), creados por el navegador del usuario **en su propia computadora**, en el momento en que un servidor Web se lo solicita.


**Figura 7-1. El servidor Web le pide al navegador que cree una *cookie*.**

Dentro de ese archivo de texto, el navegador del usuario **almacenará los datos** que el servidor le enviará y, luego –esto es funda- mental–, el navegador **reenviará automáticamente hacia el servidor** todos estos datos en cada "petición" que el usuario realice hacia ese mismo dominio (es decir, en cada enlace que pulse o en cada formulario que lleve hacia una página de ese sitio que le ordenó crear la *cookie*).


**Figura 7-2. El navegador reenvía al servidor los datos de la *cookie*.**

De esa manera, ese servidor podrá **identificar** a ese usuario a lo largo de su navegación por las distintas páginas de ese mismo sitio, algo que le servirá tanto para mostrarle contenidos específicos creados para ese usuario, como para otras tareas más invisibles para el usuario pero muy útiles para el dueño del sitio como, por ejemplo, llevar estadísticas de las visitas de cada usuario.

Algunos usos posibles: el servidor puede almacenar el nombre del usuario para saludarlo cada vez que entre a una página de ese sitio; memorizar los productos que lleva elegidos de un carrito de compras hasta que complete la operación de compra; recordar el idioma preferido por el usuario para navegar por ese sitio; o la fecha de su última visita. Y muchísimos otros usos más.

La forma en la que un servidor Web (típicamente Apache, el servidor Web instalado en la mayoría de los *hosting*s) le ordenará al navegador del usuario (Firefox, Explorer, Opera, etc.) que cree alguno de esos archivos de texto denominados *cookie*s, estará totalmente bajo nuestro control, ya que será **nuestro propio código PHP** el que le ordenará al servidor Web que éste, a su vez, le solicite al navegador la creación de la *cookie* y, una vez creada, otro código PHP

será capaz de solicitar que se **almacenen** datos dentro de ese archivo, o que se lean los que allí fueron guardados.

La manera en que estos archivos de texto se crean en el disco rígido del usuario no debe preocuparnos: de eso se encarga el **navegador** del usuario, luego de que ejecutemos la instrucción de PHP que, en unos instantes, aprenderemos.

ATENCIÓN: esta comodidad de "que lo haga todo el navegador" es también un arma de doble filo, porque así como el navegador nos simplifica el trabajo, también puede **impedirnos** por completo utilizar *cookies*, si el usuario ha configurado a su navegador para que no permita crearlas. Por lo cual, cuando usemos *cookies*, en la medida de lo posible, deberíamos tener un **plan B** alternativo (por ejemplo, plantearnos si no sería mejor usar sesiones) para no dejar afuera a quien tenga desactivada esta posibilidad en su navegador.

## Almacenar variables en cookies

En una *cookie*, se pueden almacenar **muy pocos** datos (hasta un máximo de 4093 caracteres por cada *cookie*), y solo es posible almacenar hasta 20 *cookies* por dominio (esto es importante tenerlo presente por si en distintas secciones de nuestro sitio hacemos uso de *cookies*), y un navegador solo puede almacenar hasta 300 *cookies* en total, compartidas entre todos los sitios que ese usuario visita (de lo contrario, el navegador empezará a **eliminar** las *cookies* más antiguas, lo que constituye la razón de que algunos sitios que usan *cookies* para recordar nuestros datos, nos vuelvan a pedir una y otra vez que nos identifiquemos, periódicamente: es porque su *cookie* fue borrada, desplazada por otras *cookies* más nuevas de otros sitios que fuimos visitando).

Debido a esa "escasez" de recursos de las *cookies,* es muy común almacenar únicamente un **código** en la *cookie*, para que identifique a ese usuario en ese navegador en esa computadora, y el resto de los datos asociados se almacena en el servidor (en el *hosting*, generalmente en bases de datos).

Un antiguo mito (más de diez años, ¡una eternidad en la historia de la Web!) genera toda clase de rumores en torno a las *cookies*: se dice que revelan nuestros datos privados, que espían nuestra computadora, y muchas otras fantasías, causadas por el desconocimiento de sus verdaderos límites.

La única capacidad real de una *cookie* es la de almacenar algunos pocos datos en **variables**, que luego podremos utilizar cuando, desde la misma máquina y usando el mismo **navegador**, ese usuario ingrese nuevamente a nuestra página Web.

Este detalle casi trivial merece ser recalcado: no es el usuario en sí, y ni siquiera es la computadora del usuario la que queda identificada, sino apenas el **navegador** que se utilizó; si desde la misma computadora ingresara al mismo sitio Web el hermano o el compañero de trabajo de nuestro usuario utilizando el mismo navegador, lo confundiríamos con nuestro usuario.

Y, por el contrario, si el mismo usuario ingresara usando otro navegador, o desde otra computadora, no lo reconoceremos...

Y algo más: solamente se puede leer la *cookie* desde **el dominio que la creó**, así que de "espionaje", poco y nada. Salvo en casos demasiado sofisticados, que solo son posibles para grandes empresas de publicidad *on-line*, que hospedan sus banners en un mismo dominio pero los publican en distintos sitios y, de esta manera, detectan el código de su *cookie* a lo largo de varios sitios, y pueden llevar estadísticas del perfil de visitas de un usuario: si visitó tal diario, y luego tal otro sitio de remates, etc.

Pero eso únicamente crea perfiles para mostrar publicidades afines al usuario, y no datos individualizables con nombre y apellido (salvo excepciones que, como las brujas, no existen, pero "que las hay, las hay..." aunque requieren que alguno de los sitios de la cadena comparta sus bases de datos, algo que entra en el terreno de lo legal más que de la programación).

Existen algunas consideraciones a tener en cuenta antes de pasar al código. Las *cookies* se generan a través de una orden que debe llegar al navegador del usuario **antes** de que éste procese el código HTML que nos hará ver una página en nuestra pantalla.

Para lograr esto, la orden para la creación de la *cookie* debe viajar en el *header* o encabezado de la petición HTTP que nos trae el archivo HTML, desde el servidor hacia el navegador; por lo que la ejecución de la función que crea un archivo de *cookies* **debe ser anterior a cualquier código HTML o a cualquier echo de PHP** que haya en la página (igual que con las sesiones, como veremos pronto), para que llegue dentro de los encabezados de la petición HTTP.

En el siguiente ejemplo, veremos cuál sería la ubicación del código que crea una *cookie*:

```
<?php
setcookie("nombre","Juancito");
?>
<!DOCTYPE html>
<html>
```

```
(aquí iría todo el contenido de la página)
...etc...

</html>
```

Analicemos el código precedente. La función **setcookie**, incluida en el lenguaje PHP, recibe como primer "argumento" (primera cosa dentro de sus paréntesis) el **nombre** de la variable que se almacenará en la *cookie* y, como segundo argumento (separado por una coma del argumento anterior), el **dato** que desea almacenarse dentro de esa variable, que el navegador guardará en un archivo de texto dentro de la computadora del usuario:

```
setcookie("nombre","Juancito");
```

La función **setcookie** permite especificar varios otros argumentos dentro de sus paréntesis, pero son todos opcionales:

```
setcookie (nombre, valor, duración, ruta, dominio,
seguridad)
```

Algunos de ellos los veremos unos párrafos más adelante. Pero con esos dos datos (el **nombre** de la variable y su **valor**), separados por comas y envueltos entre comillas, es suficiente para crear una *cookie* y guardar un dato.

Este ejemplo, que define una variable denominada "nombre" con el valor "Juancito", ha creado realmente una *cookie* (probemos de ejecutarlo en nuestro servidor local, o en nuestro *hosting*). Como **no se ve nada** especial en la pantalla, nos queda la duda: no sabemos si se ha logrado crear esta *cookie*, o no...

Afortunadamente, es muy fácil comprobar si se creó, o no: podemos ver con nuestro propio navegador todas las *cookies* que ese navegador ha almacenado, y cuál es el valor que guarda dentro cada una de ellas.

En cada navegador, es diferente la forma de acceder a las *cookies.*

Por ejemplo, en Firefox, accedemos entrando al menú **Herramientas → Opciones → Privacidad → Eliminar *cookies* de forma individual**, que nos muestra el listado de *cookies* que existen en ese navegador, así como su contenido, su fecha de expiración y otras informaciones complementarias.

En este caso, se vería como en la Figura 7-3.


**Figura 7-3. *Cookie* almacenada y su valor.**

De esta forma, verificamos si la *cookie* fue creada y con qué valor. Esto es muy útil mientras programamos y necesitamos detectar si funcionó nuestro código o si hubo errores. Pero por supuesto, la forma más usual de trabajar con los datos guardados en las *cookies* es la de leerlos y utilizarlos directamente desde los códigos PHP que programemos en nuestras páginas y que aprenderemos a continuación.

## Leer variables de cookies

Vamos a tratar de entender cómo es el circuito de lectura de un dato que estaba en una *cookie*. Como dijimos, la *cookie* es reenviada automáticamente por el navegador hacia el servidor *(hosting)* cada vez que el navegador detecta que se está realizando una petición a una URL de un dominio que ya tiene almacenada una *cookie* en ese mismo navegador. Es decir, en cuanto **pulsamos un enlace, despachamos un formulario o escribimos una URL y pulsamos enter**, si cualquiera de estas tres acciones apunta hacia un sitio que anteriormente nos guardó una *cookie*, en ese mismo instante, nuestro navegador **envía hacia el servidor las variables contenidas en la *cookie*** correspondiente a ese sitio (las vaya a usar o no ese servidor, que no es cuestión que le interese al navegador, él solo las envía sin preguntar), sin necesidad de que tengamos que hacer nada especial, todo sucede de forma invisible y automática.

Esas variables almacenadas en *cookies* que se envían hacia el servidor, se almacenan por el servidor Web en una **matriz**, al igual que sucedía con los datos enviados por enlaces *get* o por formularios *post*, solo que esta vez la matriz se llama **$_COOKIE**.

Imaginemos un ejemplo: en una de nuestras páginas, le pedimos al usuario que mediante un formulario complete su nombre, y lo guardamos en su propia máquina, en una variable de *cookie* llamada "nombre". Luego, cuando hagamos un echo de **$_COOKIE["nombre"]** en cualquier otro momento y desde cualquiera de las páginas de nuestro sitio, esto será posible gracias a que el navegador ya leyó en el disco rígido del usuario esta variable y la reenvió automáticamente hacia el servidor. Esto nos ahorra espacio de almacenamiento en nuestra base de datos, y causa la impresión de estar "reconociendo" a nuestros usuarios en cuanto entran a nuestro sitio... aunque lo que en verdad se reconoce es **el archivo** que hemos dejado guardado en esa computadora.

Por lo tanto, podemos concluir que la información guardada en *cookies*:

- **Perdura** más tiempo de lo que dura el simple proceso de una página PHP (ya veremos que es posible definir cuánto tiempo durará antes de que el navegador la elimine automáticamente, aunque siempre existe la posibilidad de que sea el propio usuario el que elimine sus *cookies* antes del tiempo previsto).
- Las *cookies* creadas en algunas de las páginas de nuestro sitio, se pueden leer **desde otras páginas** PHP de **nuestro propio sitio** (lo que permite identificar, a lo largo de varias páginas, a ese visitante, siempre hablando de páginas PHP dentro de un mismo dominio, nunca de otro distinto).

Volviendo al código, para utilizar los datos almacenados en una *cookie* en nuestras páginas (en la misma que la creó o en otras páginas del mismo dominio), simplemente debemos **mostrar alguna celda** de la matriz $_COOKIE.

Probemos con la creación de esta página, y llamémosla inicial.php:

```
<?php
setcookie("mivariable", "datos, datos y más datos",
time()+60);
/* Recordemos que esto es imprescindible que esté al
inicio del archivo */
?>
<!DOCTYPE html>
<html lang="es">
<head>
...etc...
</head>
<body>
<h1>¡Ya se creó la cookie!</h1>
</body>
</html>
```

Mediante este código, definimos en la máquina del usuario una variable denominada **"mivariable"**, en la que guardamos un texto que dice "datos, datos y más datos".

El tercer argumento de la función **setcookie** especifica la duración o el momento de **vencimiento** de la *cookie;* es decir, cuándo el navegador del usuario deberá eliminar el archivo de texto de su disco rígido. En este caso, se ha pasado como tercer argumento el instante actual (la función time(), tema que veremos más adelante en este libro) más 60 segundos, esto significa que luego de un minuto se autodestruirá esa *cookie*. Si la intentamos leer pasados los 60 segundos, no estará disponible el dato, ya no existirá (si no especificamos ningún momento de borrado, la *cookie* se eliminará automáticamente en cuanto el usuario cierre su navegador).

Si lo queremos probar luego de ejecutar una vez el código que crea la *cookie*, ingresemos con nuestro navegador en **otra página** de nuestro mismo

servidor que contenga una lectura de esa variable, podríamos llamarla otra-pagina.php, y contendría algo así:

```
<!DOCTYPE html>
<html lang="es">
<head>
...etc...
</head>
<body>
<h1>
<?php
if ( isset($_COOKIE["mivariable"]) ) {

      echo "La cookie contiene: ";
      echo $_COOKIE[“mivariable”];

} else {

      echo "Parece que no pasó por la página inicial.php,
      vuelva a ella así se crea la cookie.";

}
?>
</h1>
</body>
</html>
```

Si en lugar de hacer en dos páginas diferentes este uso de datos de *cookies*, quisiéramos hacerlo en una sola página, el código sería similar al siguiente (podemos llamar a esta página en-una-sola.php):

```
<?php
if (!isset($_COOKIE["mivariable"])) {
      // Si NO está presente, la creamos:
      setcookie("mivariable", "datos, datos y más datos",
      time()+60);
```

```
}
?>
<!DOCTYPE html>
<html lang="es">
<head>

...etc...
</head>
<body>
<h1>
<?php
if ( isset($_COOKIE["mivariable"]) ) {
      // Si está disponible, la usamos:

      echo "La cookie contiene: ";
      echo $_COOKIE["mivariable"];

} else {

      echo "Como es la primera vez que entra a esta página,
      acaba de enviarse al navegador la orden de crear la
      cookie, junto con este código y texto que está
      leyendo. Si vuelve a solicitar esta página otra vez
      pulsando F5, el navegador enviará hacia el servidor
      la cookie y podrá ser leída y utilizada.";

}
?>
</h1>
</body>
</html>
```

Nota: **no podemos leer una *cookie* en el mismo momento en que ordenamos crearla** con la función **setcookie**, sino que recién a partir de la siguiente petición al servidor, el valor de la *cookie* será enviado por el navegador hacia el servidor y allí podrá utilizarse.

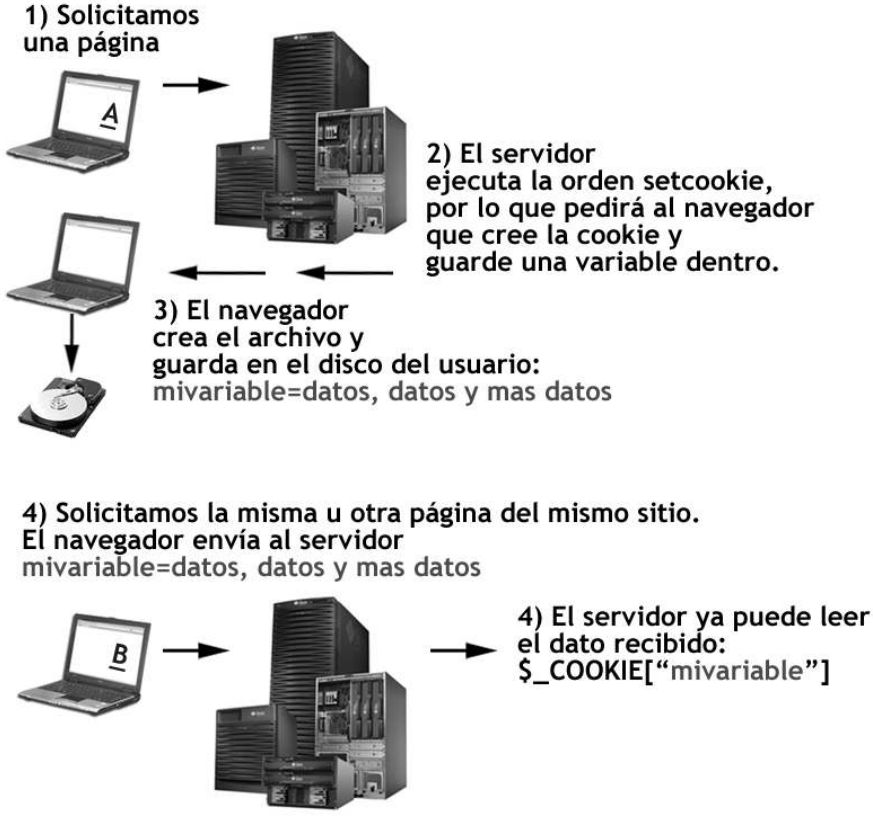


**Figura 7-4. Circuito de creación y lectura de una cookie.**

## Borrar una variable o una cookie

No es lo mismo **eliminar** el archivo de *cookie* completamente del disco del usuario, que simplemente vaciar el **valor** de alguna de sus variables.

La forma de vaciar el **valor** de una variable de una *cookie* es muy simple, y consiste en definirla tal como si la estuviéramos creando, pero **sin especificarle ningún valor**:

```
<?php
setcookie("nombre");
?>
```

De todos los parámetros posibles que podemos proporcionar dentro de los paréntesis al crear una *cookie*, el único obligatorio es el **nombre de la variable** y, en el caso de ser el único que proporcionemos, produce el **vaciamiento** de los datos que hubiéramos almacenado con anterioridad en esa variable.

> Con la misma lógica que en el caso de crear una *cookie*, si luego de ordenar borrar una *cookie*, en el mismo código de la misma página que se está ejecutando en el servidor intentamos leer esa *cookie*, ésta **todavía existirá**, ya que recién en la siguiente petición al servidor dejará de estar disponible su dato.

Complementariamente, si queremos **eliminar** del servidor inmediatamente una **variable** de *cookie* ya leída, será necesario utilizar la función **unset:**

```
<?php
unset($_COOKIE["nombre"]);
?>
```

Esto no solo vacía el valor de la variable, sino que elimina la variable misma.

En cambio, para eliminar por completo el archivo de la *cookie* del disco del usuario, será necesario utilizar un argumento extra de la función **setcookie**, que aprenderemos a continuación.

En la figura 7.5 podemos ver un esquema del proceso de eliminación de una cookie.

## Argumentos opcionales

Ya hemos utilizado en los distintos ejemplos varios de los argumentos o parámetros que podemos utilizar dentro de los paréntesis de la función **setcookie**.

La totalidad de estos argumentos es la siguiente:

```
setcookie (nombre, valor, duración, ruta, dominio,
seguridad)
```

El único parámetro obligatorio es el **nombre** de la *cookie* o variable (que de ser usado sin ningún otro dato, causa la eliminación de esa variable de la *cookie*, tal como hemos aprendido recién).

Si al utilizar esta función deseamos **saltear** (omitir) alguno de los argumentos para luego especificar otro, igualmente **debemos completar su posición** con **""** (nada, ni siquiera un espacio, entre comillas), salvo en el caso de la duración o vencimiento, que debe completarse con un **0**, y salvo también el último de los argumentos, el de limitar la *cookie* que se transmitirá solo bajo una conexión **https** (segura), en el cual los valores posibles de ese argumento son 1 (se obliga a transmitirla por **https**) o 0 (no es necesario **https**).

Veamos, entonces, los argumentos posibles que aún no hemos visto, como la ruta y el dominio.

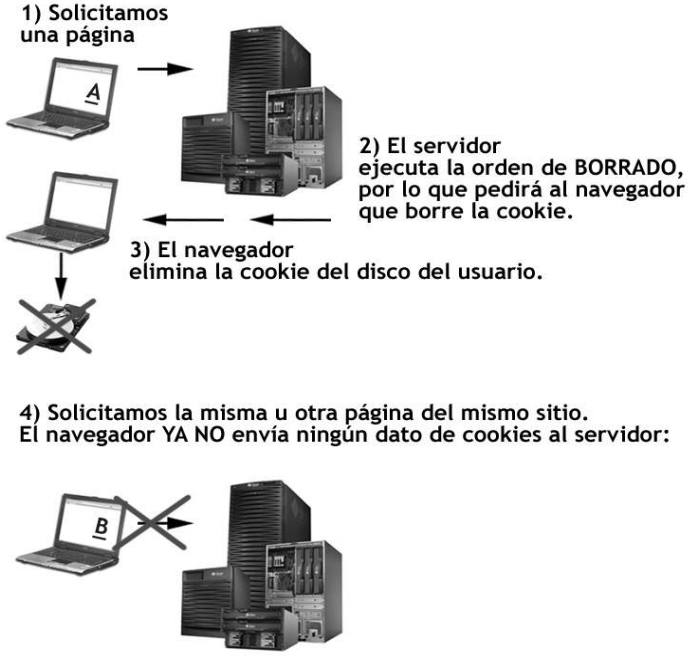


**Figura 7-5. Circuito de borrado de una *cookie*.**

El cuarto argumento de la función que crea una *cookie*, la **ruta**, indica **desde qué directorio (carpeta)** de un servidor se accederá a los datos de esa

*cookie*, impidiendo la lectura de sus datos a toda página PHP que no esté almacenada dentro de esa carpeta.

Un ejemplo:

```
<?php
setcookie("prueba",$valor,time()+3600,"/blog/","",1);
?>
```

Estamos definiendo una serie de cosas:

- creamos una variable llamada **"prueba"**,
- su contenido es lo que sea que estuviera almacenado en la variable **$valor**,
- este dato dejará de existir una hora después de su creación (60 minutos * 60 segundos = **3600** segundos).
- Esta variable "prueba" solo se podrá leer desde páginas PHP que estén almacenadas dentro de la carpeta **"blog"** de nuestro sitio.
- No especificamos ningún subdominio, dejando las comillas **vacías**.
- Y obligamos utilizar un **servidor seguro (https)** para transmitir la variable (por lo tanto, la *cookie* no se creará si no disponemos de un servidor **https**).

El quinto argumento, el del **dominio**, especifica **desde qué dominio** se accederá a la *cookie* (esto es usado para **subdominios** de un mismo sitio, para que únicamente desde ese subdominio se puedan leer los datos de la *cookie*).

Otro ejemplo:

```
<?php
setcookie("ingreso",$datos,time()+(60 * 60 * 24),"","ar.
undominio.com",0);
?>
```

En este caso, la *cookie* tendrá una duración de 24 horas, y solamente se podrá leer desde cualquier carpeta del subdominio **ar** dentro del dominio **undominio.com** que se especificó, sin que sea necesaria una conexión segura para enviar los datos.

Y, por último, si queremos **eliminar una *cookie*** por completo, vaciando el valor de su variable y eliminando físicamente el **archivo** del disco del usuario, la forma de hacerlo es indicando un momento de vencimiento anterior al actual (en el siguiente ejemplo, el "cero" del tercer argumento):

```
<?php setcookie("nombre",$datos,0);
?>
```

Tengamos presente, entonces, que existen tres herramientas distintas para "eliminar" datos relacionados con las *cookies*, pero que no son iguales:

1. Vaciar una **variable** de *cookie* sin eliminarla, con **setcookie** sin valor.
2. Eliminar del **servidor** una variable ya leída con **unset**.
3. Eliminar el **archivo** físico del disco del usuario con **setcookie** y fecha anterior.

## Ejemplo completo

Veamos ahora un ejemplo algo más complejo, pero completo y listo para usar, de uso de *cookies* para generar **código HTML personalizado**.

En este caso, el objetivo será generar apenas **una palabra** que se escribirá con un echo de PHP dentro de la etiqueta <link> de las páginas de nuestro sitio:

```
<link rel="stylesheet" href="xxxx.css">
```

Por lo cual, según el valor de esa palabra, será **una hoja de estilos distinta** la que el navegador aplicará para ese usuario durante su navegación a través de ese sitio (que permita ofrecer algunas opciones de **diseño personalizado** a cada usuario como, por ejemplo, que elija tamaños de fuentes distintos, cantidad de columnas a elección, colores e imágenes de fondo personalizados, etc. Todo esto, simplemente eligiendo cuál hoja de estilos se aplicará, que se decidirá por el valor almacenado en una *cookie*).

La secuencia de pasos será ésta:

1. Verificaremos si el usuario acaba de elegir algún estilo mediante un menú de selección (verificamos si en **$_POST** existe la variable que da nombre al *select* que hayamos proporcionado para elegir estilos);
2. si lo hizo, almacenaremos en una *cookie* el nombre del estilo elegido, y además lo **escribiremos** dentro de <link>;
3. de lo contrario, si en $_POST no proporcionó nada, verificamos con otro condicional si ya tiene la *cookie* almacenada de antes, leyéndola de **$_COOKIE** y, en ese caso, leeremos la variable que fue proporcionada por esa *cookie* y escribiremos su valor dentro de la etiqueta <link>;

4. y, por último, si tampoco en $_COOKIE encontramos datos, es que el usuario llega a la página por primera vez y todavía no eligió nada, así que, en ese caso, escribiremos un **valor por defecto** dentro de <link>.

¡Manos a la obra! Crearemos un archivo llamado **pagina.php**:

```
<?php

if ( isset($_POST["eleccion"]) ){
      /* si el dato está ubicado allí, es que acaban de
      usar el formulario y enviaron un estilo elegido. */

      $hoja = $_POST["eleccion"];
      /* Entonces, dejamos preparada una variable para
      usarla dentro de <link> */

      setcookie("elegido",$hoja,time()+(60 * 60 * 24 *
      30));

      /* Y guardamos una variable a la que llamamos
      "elegido" dentro de una cookie; su valor lo tomamos
      de $_POST["eleccion" (o de $hoja que es lo mismo), y
      definimos su vencimiento para dentro de 30 días.

} else {
      /* Si no encontramos en $_POST el dato, vemos si ya
      estaba en una cookie desde una visita anterior: */

      if ( isset($_COOKIE["elegido"]) ){
         /* Si está, colocamos dentro de la misma variable
         $hoja lo que el usuario había elegido: */

         $hoja = $_COOKIE["elegido"];

      } else {
         /* Si tampoco estaba allí, mostraremos una hoja
```

```
         por defecto: */
         $hoja = "hoja-por-defecto";
      }
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
      <title>Usando cookies para elegir hoja de
      estilos</title>
      <meta charset="utf-8">
<?php
/* Sea cual sea el camino tomado anteriormente, la
variable $hoja debe tener algún valor, así que lo leemos y
lo usamos para que se cargue la hoja respectiva: */
if ( isset($hoja) ) {

      echo '<link rel="stylesheet" type="text/css"
      href="'.$hoja.'.css" />';
}
?>
</head>

<body>
      <h1>¡Elijamos un diseño!</h1>
      <form action="pagina.php" method="post">
      <fieldset>
      <p>Cambiemos el estilo a nuestro gusto:</p>
      <select name="eleccion">
        <option value="minimalista">Minimalista</option>
        <option value="antigua">Antigua</option>
        <option value="moderna">Moderna</option>
        <option value="zoom">Fuentes grandes</option>
        <option value="tres">Tres columnas</option>
        <option value="dos">Dos columnas</option>
      </select>
      <input type="submit" value="Elegir">
```

```
      </fieldset>
      </form>
</body>
</html>
```

Para que el rompecabezas funcione, los nombres que le daremos a nuestras hojas de estilo serán los que hemos usado dentro de este código: "hoja-por-defecto.css", "minimalista. css", "antigua.css", "moderna.css", "zoom.css", "tres.css" y "dos.css".

Por supuesto, hagamos que los **cambios** en cuanto a diseño en las distintas hojas CSS sean notorios, para que al navegar por las distintas páginas de nuestro sitio veamos cómo se mantiene el estilo elegido al comenzar la navegación.

Agreguemos en cada página de nuestro sitio los contenidos propios de cada página, e incluyamos un menú para navegar entre ellas, y asegurémonos de repetir tanto el menú de selección para elegir un estilo, como el código PHP necesario para que lo detecte y lo guarde en las *cookies* o lo lea en ellas, completando la etiqueta <link>.

## Ideas para aplicar cookies

Para terminar con el tema de las *cookies*, sugeriremos algunas ideas para otros usos distintos a los que ya vimos.

Por ejemplo, podríamos memorizar si un usuario ya visitó un producto o una categoría de productos de un catálogo. Si gracias a una *cookie* detectamos que visitó el producto "Mesa de ping pong", entonces podríamos mostrarle una publicidad de "Paletas de ping pong", o de "Clases de ping pong a domicilio". Contenido a medida...

Otro uso interesante sería detectar cuáles de nuestros visitantes ya ingresaron anteriormente a una página de un formulario de inscripción o de contacto, pero **sin usarlo** (el haberlo usado hubiese cargado en una *cookie* un valor especial al llegar a la página de recepción del formulario), y mostrarles un contenido especial que los "anime" a completar la consulta o la inscripción que no se decidieron a concretar en su anterior visita.

Del mismo modo, en un sitio que venda algún producto *(e-commerce)* mediante un carrito de compras, podríamos detectar si alguien está usando la *cookie* (es decir, si ya eligió productos o se interesa por alguno de ellos), pero **no finalizó** la compra, y mostrarle un texto específico (igual que en el caso anterior, en la página de finalización de compra deberíamos modificar o borrar la *cookie* anterior del carrito, para que ya no se le muestre ese texto).

Y, complementariamente, si alguien "sí" compró un producto o servicio, al momento de la compra, podríamos almacenarle una *cookie* para que en cierta cantidad de días **más tarde** (que dependerá de la duración del producto vendido), al entrar a nuestro sitio, le recordemos que se está por acabar su producto y que es el momento de que lo compre nuevamente.

En sitios que publican contenidos con cierta frecuencia (diaria, semanal, mensual), podríamos guardar la fecha de la última visita del usuario a nuestro sitio y, calculando la diferencia entre la fecha actual y la última visita, mostrarle un cartel de "¡Contenido Nuevo!" en los enlaces que apunten hacia contenidos publicados luego de su **última visita** (muy pronto aprenderemos a manejar funciones para cálculos con fechas que nos permitirán hacer este tipo de cosas).

Realmente, son muchísimas las aplicaciones que se nos pueden ocurrir mediante el uso de *cookies*. Pero, siempre dentro del tema de identificación del usuario, es posible ir un nivel más allá, y comenzar a trabajar con sesiones, que es el tema que veremos a continuación.

# Sesiones: datos que identifican a un usuario

¿Qué es una sesión? Simplemente, otra forma más de almacenar datos relacionados con un usuario en particular, tal como en el caso de las *cookies*. Pero el lugar en el que quedan almacenados los datos (el **servidor**, en el caso de las sesiones) es lo que diferencia a las sesiones de las *cookies* (que como ya vimos, almacenaban sus datos en la computadora del usuario).

Y, ¿cuáles son las ventajas de almacenar datos de un usuario en nuestro propio **servidor**? Principalmente, la seguridad de su persistencia, ya que los datos no dependen de que el navegador del usuario borre una *cookie*. Además, la capacidad de almacenamiento es muchísimo mayor que el pequeño límite de 4Kb de las *cookies*. Y pueden funcionar, incluso, aunque el navegador del usuario esté configurado para no aceptar *cookies*.

La dinámica de funcionamiento de una sesión es muy sencilla: al ejecutarse la orden de inicio de una sesión, se crean automáticamente dos cosas:

1. Un **archivo de texto** dentro de una carpeta temporal en el servidor, que es donde se guardarán todos los **datos (variables)** asociados a esa sesión. El nombre de ese archivo es muy importante y consiste en una parte fija ("sess" en este ejemplo), más un guion bajo y un código aleatorio único, siempre distinto, que puede ser similar a esto: `sess_xyza123xyzab12x`

2. Y, del lado del usuario, se crea un **código** que se guarda en una variable de *cookie*, o que también puede transmitirse mediante una **constante** llamada SID *(session identifier,* identificador de sesión) agregada a todos los enlaces o formularios de nuestro sitio. Ese código contiene aquella parte del nombre del archivo recientemente creado en el servidor que sigue al guion bajo _, por lo cual, en el ejemplo anterior, el valor del código que estará del lado del usuario será: `xyza123xyzab12x`

Este **par de elementos**, ubicados en dos lugares diametralmente opuestos (servidor por un lado, y disco de la computadora del usuario, por el otro), permiten que, al recibir cada petición, el servidor Web pueda relacionar al usuario que porta un **código**, con su respectivo **archivo de sesión** almacenado en el servidor.

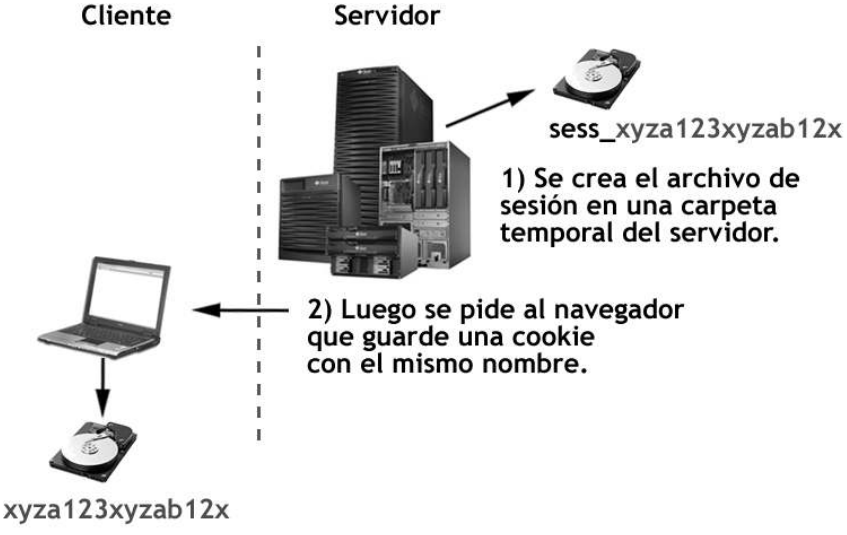


**Figura 7-6. Ambas partes de una sesión: archivo en servidor, y *cookie* del lado del usuario.**

En las siguientes peticiones del usuario, mientras navega por páginas del mismo sitio, será la llegada de ese código al servidor (ya sea desde una *cookie* o desde un enlace que envía el SID) lo que permita **relacionar** a quien envió ese código con los **datos (variables)** guardados en su archivo de sesión respectivo.

A continuación, vamos a tratar de comprender en detalle cómo "se le sigue el rastro" a un usuario de un sitio Web mediante sesiones.

Habitualmente, sin que utilicemos sesiones, cuando un usuario está en una página y pulsa un enlace solicitando ver otra página del mismo sitio, el servidor

Web **no sabe quién** le está haciendo ese pedido de entre los muchos usuarios que pudieron haber solicitado lo mismo. No sabe que ese usuario estaba viendo anteriormente tal otra página, ni cuánto tiempo pasó viéndola. Cada pedido para ver una nueva página, es totalmente **independiente** de cualquier solicitud anterior. Entonces, se hace difícil trazar el recorrido que cada usuario realizó (con fines estadísticos, para analizar el comportamiento de nuestros usuarios a medida que navegan por nuestro sitio, o simplemente para mantener datos relacionados con ese usuario, generalmente usuario y contraseña, a lo largo de toda su visita).

Este problema de olvidar en el servidor lo que el usuario escribió (o seleccionó) en la página anterior, es clásico de los formularios en varios pasos, ya que van solicitando en distintas páginas datos y más datos. Anteriormente (más de quince años atrás), esto se solucionaba mediante **campos ocultos** de formulario (*input type="hidden"*), que reenviaban hacia el servidor todos los datos del usuario que nos interesaba preservar hasta la página siguiente. Si los datos son demasiados, esto es bastante tedioso (por ejemplo, todos los productos de un sitio de comercio electrónico que fueron elegidos por un mismo usuario, y la cantidad de cada uno, o todos los datos personales, laborales y académicos, en el caso de un formulario en varias páginas de una Web de empleos).

Para resolver este tipo de problema, podemos recurrir a las sesiones, cuyo funcionamiento consiste en relacionar a un navegador, en un momento en particular, con un **código único**. Con solo transmitir ese código de una página a otra, podremos acceder a todo el resto de datos asociados a ese código que estuvieran guardados en el servidor. A ese código se lo denomina identificador de sesión, o Session ID, de allí que su sigla sea SID.

Las sesiones no se "activan" por sí solas, sino ejecutando una función de PHP; pero esa ejecución puede programarse para que suceda por el solo hecho de **llegar** a una página, o mediante **alguna acción** específica del usuario:

- Por ejemplo, en un catálogo de *e-commerce*, por el solo hecho **de entrar** a la página de inicio, se suele crear automáticamente una sesión, para que se vayan memorizando en variables de sesión los productos elegidos por el usuario y la cantidad de cada uno, mientras navega y pasa de una página a otra del catálogo y va eligiendo más productos, modificando su cantidad o eliminando alguno de ellos del carrito de compras. El famoso carrito de compras no es más que variables almacenadas en un archivo de sesión.
- En otros casos, no se inicia la sesión hasta que el usuario **haga algo**; por ejemplo, hasta que escribe su nombre de usuario y contraseña y envía un formulario; en ese momento, ha iniciado una sesión y ya son identificables sus datos hasta que la cierre, puesto que su usuario y contraseña quedaron guardados en variables de sesión. Este es el caso en un sitio de

*home-banking*, o en un campus virtual, un *webmail*, o cualquier otro sitio que requiera identificarse para acceder.

Básicamente, la tarea que hace una sesión es generar un **identificador** único para cada sesión (una constante llamada **SID** -Session ID-) y permitir que se le asocien **variables**, cuyos valores pertenecerán solo a esa sesión (ejemplo: los productos elegidos en un carrito de compra por "ese" usuario).

| | Usando sesiones | Usando formularios |
|---|---|---|
| **Se crea un identificador** | SI | NO |
| **Se declaran variables...** | variable 1<br>variable 2<br>variable 3<br>variable 4...<br>(variables de sesión) | variable 1<br>variable 2<br>variable 3<br>variable 4...<br>(variables comunes) |
| **¿Cómo las transmitimos a la siguiente página?** | Pasando solamente el ID de sesión, ya están TODAS las variables de sesión disponibles en la nueva página. | Pasando una por una cada variable en campos ocultos del formulario. |

**Tabla 7-1. Comparación entre variables de sesión y variables comunes.**

## ¿Es mejor usar cookies o identificadores de sesión?

Ya hemos visto que el punto débil de las *cookies* es que **dependen de que el usuario** las haya habilitado, o no, en su navegador; incluso hay circunstancias en las que ni siquiera el usuario voluntariamente puede activar sus *cookies*, sino que su creación puede estar impedida por un *firewall* o una directiva o filtro de un servidor de una red, que imposibilite a todos sus usuarios el uso de *cookies*. En consecuencia, por más que el usuario quisiera habilitarlas, no se lo permitirá su administrador de redes. Esto es bastante frecuente en ámbitos corporativos.

Para superar esta limitación, se puede recurrir a las sesiones. Podríamos pensar, entonces, que las sesiones son "un paso más allá" de las *cookies*, aunque esto depende de la forma en la que sean programadas por nosotros, ya que, paradójicamente, la forma más común de utilizar sesiones, la más difundida, es propagar el identificador de sesión mediante *cookies*, y se pierde toda su potencia, ya que, nuevamente, depende de la activación de las *cookies* por parte del usuario...

Tenemos dos formas de transmitir el identificador de sesión entre páginas:

1. Almacenando el identificador de sesión en una *cookie* en la máquina del usuario;
2. O enviando entre una página y otra el **SID** (o Session ID) mediante **URLs**; es decir, dentro de los enlaces o del atributo *action* de nuestros formularios.

Por defecto, el intérprete de PHP intenta el primer método, que define una *cookie*, pero en el caso de que no lo pueda hacer (porque el navegador del usuario no lo soporta, o porque el servidor esté configurado de una manera especial) deberíamos prever el método alternativo.

- La ventaja principal del uso de *cookies* para propagar el identificador de sesión es que permite que el usuario **salga de nuestro sitio**, visite otras páginas y, al regresar dentro del tiempo de vigencia de la *cookie*, encuentre que su sesión todavía permanece activa, vigente (eso puede ser cómodo en unos casos, y peligroso en otros). Otra ventaja es que es más difícil el "robo" de sesiones, que un usuario pase a otro el identificador de sesión. La gran desventaja de usar *cookies* es que pueden estar **desactivadas**.
- Por el contrario, en caso de enviar el **SID en los enlaces** y no utilizar *cookies*, la principal desventaja es que, en cuanto salgamos con el navegador hacia otro sitio distinto del nuestro, la sesión **habrá caducado** (luego, cuando regresemos al sitio, tendremos que crear una sesión nueva, diferente a la anterior). Otra desventaja es que deberemos **modificar a mano** cada enlace del sitio y cada formulario, algo relativamente fácil en un sitio nuevo, pero posiblemente muy difícil en un sitio ya existente. Y otra desventaja más es que, en un instante, el identificador de sesión se puede enviar de un usuario a otro, y podría haber dos personas utilizando una misma sesión (algo propicio para fraudes y otros engaños). La única ventaja es que no se depende de las *cookies* y, por lo tanto, **siempre** estará disponible este método para todos los usuarios.

Ambas posibilidades tienen sus ventajas y sus contras, tendremos que pensar en cada caso qué nos convendría utilizar, según nuestros objetivos y usuarios.

Cualquiera sea la forma que elijamos, lo verdaderamente imprescindible es mantener disponible el **identificador de sesión** a lo largo de la navegación entre varias páginas de un mismo sitio, ya que sin obtener página tras página el identificador de sesión activa, el servidor no podría acceder a las variables de sesión de ese usuario.

## Una clave que se compara contra un dato guardado en el servidor

Antes de comenzar a aprender las funciones de PHP relacionadas con el manejo de sesiones, pensemos que toda esta información sobre los identificadores de sesión y sus variables asociadas "en algún lugar" físico debe quedar almacenada: como ya sabemos, **por cada sesión** iniciada por un usuario, **se crea un archivo** de texto cuyo nombre es el mismo que el identificador de sesión. Este archivo se guarda en el servidor (*hosting*), en un directorio especificado en la variable **session_save_path** del archivo php.ini.

### Modificar la ruta donde se almacenan los archivos de sesiones

Si por algún motivo quisiéramos elegir dónde guardará las sesiones nuestro servidor local, deberemos editar el archivo **php.ini** (como vimos al inicio del libro, un archivo de texto que contiene la configuración del programa intérprete de lenguaje PHP en esa máquina) y buscar la variable **session.save_path** y escribirle la ruta en la que queremos que almacene los archivos de las sesiones (ruta relativa a la carpeta raíz del servidor, si se trata de un Linux):

```
session.save_path = "/tmp"
```

o, si se trata de un servidor local que use Windows, ruta que incluya el disco, y usando barras invertidas:

```
session.save_path = "C:\temp"
```

En el caso de no poder acceder al archivo php.ini (por ejemplo, en un *hosting* alquilado), podemos ejecutar la función **phpinfo()** en una página, y veremos todas las variables relativas a sesiones (busquemos el texto *session* en esa página). Allí, veremos los valores dados a la configuración de sesiones en el php.ini de ese *hosting* (solo que cambiar alguno de ellos quedará supeditado a la voluntad y calidad del servicio de cada empresa de *hosting* en particular).

Como ya sabemos, para ejecutar **phpinfo()**, simplemente escribiremos esto:

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

Si queremos cambiar el lugar físico de almacenamiento de los archivos de sesión, podemos intentar lo siguiente (no en todos los servidores funcionará, será cuestión de intentarlo):

```
<?php
ini_set("session.save_path", "/ruta/deseada");
```

```
?>
```

La "ruta" será aquella carpeta donde queramos que se guarden nuestros archivos de sesión. Por supuesto, antes crearemos la carpeta con nuestro programa de FTP y le daremos los permisos de escritura. Si encontramos problemas, es recomendable consultar al soporte de nuestra empresa de *hosting* para que nos orienten.

## Cómo iniciar una sesión

Es momento de empezar a trabajar con sesiones. Vamos a iniciar una sesión y a mostrar el identificador de sesión creado por la función **session_start()**:

```
<?php
session_start();

$identificador = session_id();
echo "El identificador de esta sesión es: $identificador";
?>
```

La función **session_start** debe incluirse en todas las páginas en las que se pretenda seguir con una sesión activa, si ya la hubiera, o empezar una nueva.

Un detalle importantísimo es que debe ser **lo primero** que se colocará en la página, antes de cualquier salida de código por **print** o **echo**, ya que la sesión intenta **definir una *cookie*** en el usuario, y esta capacidad solo puede ejecutarse cuando los encabezados de la petición HTTP (los *headers*) aún no han sido enviados.

Si hubiera tan solo un espacio en blanco, o un carácter, o una etiqueta HTML, o, incluso, la DTD del archivo antes de usar la función que las crea, ésta **no funcionará y provocará un error**.

El error se nos informará con el siguiente mensaje:

**Warning: Cannot send session cookie - headers already sent by (output started at /ruta/al-archivo.php in ruta/al-archivo.php on line X**

Ver este mensaje es señal inequívoca de que hemos escrito algo antes de abrir el *tag* de PHP y ejecutar la función session_start().

## Cómo declarar variables de sesión

Hemos llegado al momento de aprender una funcionalidad realmente útil de las sesiones, que consiste en la posibilidad de declarar **variables de sesión**. Es tan sencillo como almacenar un dato dentro de una celda de la matriz llamada $_SESSION, que guarda todas las variables de sesión, de forma similar a lo que hacen las matrices $_GET, $_POST, $_COOKIE, etc. para datos originados mediante otros métodos.

Todo dato almacenado dentro de $_SESSION estará disponible en cualquier página del sitio que mantenga la sesión activa (es decir, que ejecute la función **session_start** al comienzo del archivo).

Veamos un ejemplo: este es el **archivo "pagina1.php"**:

```
<?php
session_start();
$_SESSION["nombre"] = "pepe";
/* Desde ya que ese dato en vez de asignarlo aquí, podría
venir de un formulario, o de un enlace.
Por ejemplo:
$_SESSION["nombre"] = $_POST["campoX"];
o bien:
$_SESSION["nombre"] = $_GET["enlaceX"];
*/
?>
<!DOCTYPE html>
<html lang="es">
<?php echo "Hola: ".$_SESSION["nombre"]; ?>
<p><a href="pagina2.php">Siguiente página</a></p>
<!-- etc -->
</html>
```

Y este es el **archivo "pagina2.php"**:

```
<?php
session_start();
// Debemos ejecutar esta función para continuar la sesión
?>
```

```
<!DOCTYPE html>
<html lang="es">
<?php echo "Su nombre es: ".$_SESSION["nombre"]; ?>
<!-- etc -->
<p><a href="pagina1.php">Volver</a></p>
<!-- etc -->
</html>
```

Notemos que simplemente con ejecutar la función **session_start**, está disponible, en la segunda página, para que se lea todo el contenido de la matriz $_SESSION (en este caso, solamente el "nombre", pero pueden ser muchos más datos).

Es el contenido completo de la matriz $_SESSION lo que se guarda en el **archivo de sesión** de cada usuario. Un archivo de sesión por dentro es similar a esto:

```
nombre|s:4:"pepe";
```

Donde "nombre" es el nombre de la variable, y luego del separador **|** vemos información acerca del contenido de esa variable (**s** de *string* o caracteres alfanuméricos, **4** es la cantidad de caracteres, y "pepe" es el contenido de la variable).

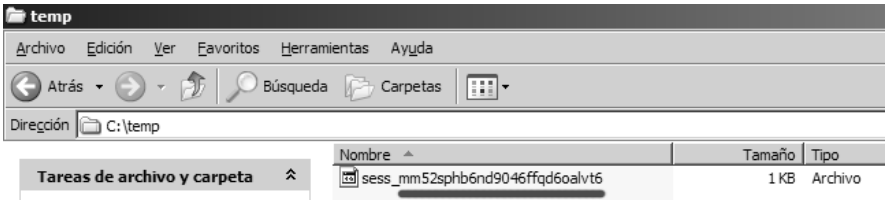


**Figura 7-7. Vemos el archivo de sesión que fue creado en el servidor.**

Del lado opuesto, en el disco del usuario, se ha creado una *cookie* que contiene una constante llamada *PHPSESSID* con un valor único, correspondiente al nombre del archivo de texto creado en el servidor.

Ese código es el único dato que almacenará la *cookie* del lado del usuario y servirá para identificarlo.

Ver Figura 7-8.


**Figura 7-8. Vemos la *cookie* de sesión creada en el disco del usuario.**

## Cómo transmitir el identificador de sesión sin cookies

Puede sucedernos que en determinados servidores no tengamos habilitada la opción de utilizar alternativas a las *cookies* para guardar el identificador de sesión. En esos casos, cuando ejecutemos la función **phpinfo()** veremos que esos servidores tienen configurado lo siguiente:

```
session.use_only_cookies = On
```

Al estar configurado en *On*, nos obliga a usar *cookies* y no funcionará ninguna otra alternativa. En cambio, si está configurado en *Off,* podremos usar métodos alternativos, sin *cookies*.

Si tenemos acceso al archivo php.ini, en un servidor local, veremos que, en ese caso, contendrá una línea que dice:

```
session.use_only_cookies = 1
```

Donde el valor **1** indica que es **obligatorio** usar exclusivamente *cookies*, es decir, no funcionará otro método. Pero si el valor que vemos es **0**, podremos utilizar métodos alternativos.

El método alternativo a las *cookies* para propagar el identificador de sesión, consiste en agregar la constante **SID** en todos los enlaces y formularios, incluyéndola en todas las URLs.

El ejemplo anterior quedaría así:

```
<?php
session_start();
?>
<!DOCTYPE html>
<html lang="es">
<?php
echo "<p>Su nombre es: ".$_SESSION["nombre"]."</p>"; ?>
<p><a href="pagina1.php?<?php echo SID;?>">Volver</a></p>
<!-- etc -->
</html>
```

Cuando veamos este enlace en nuestro navegador, veremos que, en realidad, esa orden que muestra la constante **SID** "escribió" dentro del enlace algo similar a esto:

```
pagina1.php?PHPSESSID=jvav952q2ek0l8g9tjp5g7rur3
```

Donde PHPSESSID es el nombre de la sesión, y el resto es su valor único (cuando lo probemos, veremos otro código diferente, por supuesto).

Una variante con la que podemos encontrarnos es la de pasar estos datos sin usar la constante SID, utilizando **dos funciones** de PHP que escriben exactamente lo mismo que acabamos de ver: el nombre de la sesión: PHPSESSID, y su valor:

```
<p>
```

```
<a href="pagina1.php?<?php echo session_name();?>=<?php
echo session_id();?>">Volver</a>
</p>
```

Como ya nos podemos imaginar, la función **session_name()** escribe el nombre de la sesión (PHPSESSID) y, la función **session_id()**, el código único aleatorio que identifica a esa sesión.

Siguiendo con el escenario de no recurrir a *cookies*, si en lugar de utilizar un enlace en alguna página de nuestro sitio, necesitáramos un **formulario** que lleve a otra página, deberemos agregar al atributo **action** del formulario el paso de la constante SID, de esta manera:

```
<?php
session_start();
$_SESSION["dato"] = "clientenuevo";
?>
<form action="pagina2.php?<?php echo SID; ?>">
      <input...etc...>
</form>
```

La variable "dato", en este ejemplo, estará disponible en la página siguiente al construir el valor del atributo "action" de esta manera.

Desde ya, también lo podemos lograr con el otro método, usando las dos funciones de PHP:

```
<?php
session_start();
$_SESSION["dato"] = "clientenuevo";
?>
<form action="pagina2.php?<?php echo
session_name();?>=<?php echo session_id();?>">
      <input...etc...>
</form>
```

Pero es más recomendable (por lo simple y breve) utilizar la constante SID.

## Cómo borrar una variable de sesión, o todas

Así como ya sabemos asignar un valor para que sea almacenado dentro de una variable de sesión, complementariamente, necesitaremos **borrar** las variables utilizadas por una sesión (por ejemplo, al "salir" de una zona privada de una Web donde nos manteníamos identificados mediante variables de sesión).

No debemos confundir el borrar el **valor** de una variable (donde la variable sigue existiendo, aunque vacía), con el eliminar la **variable** directamente.

En el primer caso, alcanza con hacer:

```
$_SESSION["dato"] = "";
```

En cambio, ahora veremos cómo eliminar la variable directamente, no solo su valor almacenado.

Para ello, contamos con la función **unset(),** que elimina la variable de sesión que especifiquemos, por ejemplo:

```
<?php
session_start();
unset($_SESSION["dato"]);
/* Si a continuación queremos usar esa variable, no estará
más disponible: */
echo $_SESSION["dato"]; // producirá un error
?>
```

Es importante evitar que se utilice **unset($_SESSION)** porque nos quedaríamos sin la posibilidad de usar variables de sesión en el resto de la página.

Incluso, podemos eliminar todas las variables de sesión (y sus valores, por supuesto) de una sola vez con:

```
$_SESSION = array();
```

Pero, tanto si eliminamos una sola o pocas variables de sesión, como si eliminamos todas las variables de sesión de una vez, debemos tener presente que no estamos eliminando el **archivo** de sesión del servidor, ni la *cookie* almacenada en el navegador, por lo cual la sesión que relaciona a este usuario con su archivo en el servidor sigue activa, aunque sin datos almacenados.

## Cómo eliminar el archivo de sesión y la cookie

La manera más drástica de cerrar una sesión es eliminar directamente el **archivo** de sesión y la *cookie*, ya que, en el primer caso, borramos todas las variables de sesión allí almacenadas y, en el segundo, eliminamos toda relación entre un navegador y el archivo de igual nombre en el servidor.

Para hacer un botón que culmine la sesión y elimine todos los datos asociados a ella, simplemente debemos realizar un enlace hacia un archivo PHP que ejecute la función **session_destroy()**. No necesita ningún argumento entre sus paréntesis.

Un detalle muy importante es que solo pueden cerrarse las sesiones **abiertas**; es decir, ejecutar lo siguiente:

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

provocaría un mensaje de **error**; es preciso que en la página "de salida", en la que cerramos la sesión, **la retomemos** antes de cerrarla:

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

Hasta aquí, hemos eliminado del servidor el archivo de sesión y todas sus variables. Pero todavía en el navegador existe la *cookie*.

Una forma más rotunda de eliminar la sesión (y es la que recomendamos utilizar) sería la siguiente:

```
<?php
// 1. Retomamos la sesión
session_start();

// 2. Eliminamos las variables de sesión y sus valores
$_SESSION = array();

/* 3. Eliminamos la cookie del usuario que identificaba a
esa sesión, verificando "si existía" */
```

```
if (ini_get("session.use_cookies")==true) {
      $parametros = session_get_cookie_params();
      setcookie(session_name(), '', time()-99999,
      $parametros["path"], $parametros["domain"],
      $parametros["secure"], $parametros["httponly"]);
}

// 4. Eliminamos el archivo de sesión del servidor

session_destroy();
?>
```

Notemos que en el condicional averiguamos si se está utilizando una *cookie* para transmitir el identificador de sesión, y si esto es cierto, entonces procedemos a leer todos los parámetros de esa *cookie* ejecutando la función **session_get_cookie_params()**, para, a continuación, usar esos parámetros dentro de una nueva ejecución de la función **setcookie**, que eliminará la *cookie* al haber definido su tiempo de expiración en el pasado.

## La duración de las sesiones y la seguridad

La **duración** de las sesiones puede establecerse en el archivo php.ini en la variable **session.cookie_lifetime**; si es **0**, la sesión y todas sus variables relacionadas se destruyen solo cuando el usuario **cierra el navegador**. Si no, caducan en la cantidad de segundos que diga en esta variable. Esto es muy importante tenerlo en cuenta, ya que muchos fraudes en Internet se producen mediante la **usurpación de la sesión** de otro usuario.

Pensemos un ejemplo: estábamos navegando en una máquina compartida y en algún momento ingresamos a un sitio donde nos hemos identificado, un *webmail* como Gmail, por ejemplo; luego, nos retiramos sin cerrar la sesión. Se sienta otro usuario frente a la máquina, abre el mismo navegador, ingresa al mismo sitio (Gmail), y sin que él haga nada, recupera y continúa nuestra sesión, es decir, sigue adentro del sitio (el *webmail*, por ejemplo) identificado como si fuéramos nosotros los que estuviéramos navegando. Con lo cual, lee nuestros mensajes, envía mensajes en nuestro nombre, borra cosas, etc. Y ni pensemos cuando recuperan la sesión de un carrito de compras en la que hemos ingresado nuestro número de tarjeta, o de nuestro *home-banking*...

## Ejemplo de acceso restringido

A continuación, aplicaremos todos los conceptos que hemos aprendido en un ejemplo completo, que permitirá a nuestros usuarios navegar por una **zona restringida** de nuestro sitio, mediante un usuario y contraseña guardados en variables de sesión.

Crearemos ante todo el primer archivo, que denominaremos index.php:

```
<?php
session_start();
?>
<!DOCTYPE html>
<html lang="es">
<head>
      <title>Zona de acceso restringido</title>
      <meta charset="utf-8">
</head>
<body>
<?php
/* 1. Si acaban de enviar el formulario de acceso, leemos
de $_POST los datos: */
if( isset($_POST["usuario"]) and isset($_POST["clave"]) ){

      // 2. En ese caso, verificamos que no estén vacíos:
      if( $_POST["usuario"]=="" or $_POST["clave"]=="") {

      echo "Por favor, completar usuario y clave";

      /* 3. Si no estaban vacíos, comparamos lo ingresado
con el usuario y clave definidos por nosotros, en este
caso "pepe" y "123456". Aquí modificaremos esos datos y
los cambiaremos por el usuario y clave que nos gusten. */
      } elseif ($_POST["usuario"]=="pepe" and
$_POST["clave"]=="123456"){

      /* 4. Si eran correctos los datos, los colocamos en
variables de sesión: */
```

```
      $_SESSION["usuario"]=$_POST["usuario"];
      $_SESSION["clave"]=$_POST["clave"];
      echo "Usted se ha identificado como:
".$_SESSION["usuario"];

}
/* 5. Aquí podríamos colocar un else con un mensaje si
los datos no eran correctos. */
}
?>
<div id="menu">
<ul>
<li><a href="primera.php">Primera página privada</a></li>
<li><a href="segunda.php">Segunda página privada</a></li>
<li><a href="tercera.php">Tercera página privada</a></li>
</ul>
</div>

<div id="formulario">
<form name="acceso" method="post" action="index.php">
<fieldset>
<legend>Ingrese sus datos de acceso:</legend>
<label for="usuario">Su usuario:</label>
<input type="text" id="usuario" name="usuario"><br>
<label for="clave">Su clave:</label>
<input type="text" id="clave" name="clave"><br>
<input type="submit" id="ingresar" name="ingresar"
value="Ingresar">
</fieldset>
</form>
</div>
</body>
</html>
```

El código de las páginas "privadas" (primera.php, segunda.php y tercera.php, en este ejemplo) será similar: en todos los casos, comenzarán por retomar la sesión, y verificarán que esté presente el usuario y la clave, y que sus

valores sean los esperados, y pasada esta verificación, mostrarán su contenido; de lo contrario, solicitarán al usuario dirigirse hacia el formulario inicial:

```
<?php
session_start();
?>
<!DOCTYPE html>
<html lang="es"> ...etc...
<?php
if( isset($_SESSION["usuario"]) and
$_SESSION["usuario"]<>""){

      echo "Usted se ha identificado como:
".$_SESSION["usuario"];
// Notemos que dejamos SIN CERRAR la llave del if
?>
<h1>PEGAR aquí TODA LA "PAGINA SECRETA"</h1>
<div id="menu">
<ul>
<li><a href="primera.php">Primera página privada</a></li>
<li><a href="segunda.php">Segunda página privada</a></li>
<li><a href="tercera.php">Tercera página privada</a></li>
</ul>
</div>

<?php
      // Recién aquí cerramos el if
} else {
?>
<p>La sesión no está abierta, por favor utilice el <a
href="index.php">formulario de acceso</a></p>
<?php
} // cierre del else
?>
```

## Internacionalización usando sesiones

En este otro ejemplo, veremos que una forma muy común de preparar nuestras páginas para la **internacionalización** de su interfaz y/o contenidos, es armar páginas HTML sin "ni una palabra" en ningún idioma, sino solo **echos** de variables (o constantes) de PHP.

Por ejemplo, veamos un supuesto index.php:

```
<?php
session_start();

if ( isset($_POST["idioma"]) ) {

      $_SESSION["idioma"] = $_POST["idioma"];
      $idioma = $_SESSION["idioma"];
      include("idiomas/$idioma.php");

} elseif ( isset($_SESSION["idioma"]) ) {
      $idioma = $_SESSION["idioma"];
      include("idiomas/$idioma.php");

} else {

      include("idiomas/espanol.php");
      // Un idioma por defecto.
}
?>
<!DOCTYPE html>
<html lang="es">
<head>
<!-- etc. -->
</head>
<body>
<div id="titulo"><?php echo TITULO; ?></div>
<div id="subtitulo"><?php echo SUBTITULO; ?></div>
<div id="avance"><?php echo AVANCE; ?></div>
<div id="noticia"><?php echo NOTICIA; ?></div>
```

```
etc...
<div id="menu">
<form name="formu1" method="post" action="index.php">
<fieldset>
<legend>Elija su idioma</legend>
<select name="idioma">
      <option value="espanol">Castellano</option>
      <option value="ingles">English</option>
</select>
<input type="submit" value="Elegir">
</fieldset>
</form>
</div>
</body>
</html>
```

Y, en la carpeta "idiomas", crearemos un archivo por idioma:

`idiomas/espanol.php`

`idiomas/ingles.php`

`idiomas/frances.php`

etc.

Estos archivos contienen TODOS los textos del sitio, almacenados en LOS MISMOS nombres de constantes:

Contenido del archivo **espanol.php**:

```
<?php
define("TITULO", "Bienvenidos a mi sitio en castellano");
define("SUBTITULO", "Este es un subtítulo en
castellano...");
define("AVANCE", "Este es un texto complementario...");
define("NOTICIA", "Este es el texto largo en
castellano...");
?>
```

Contenido del archivo **ingles.php**:

```
<?php
define("TITULO", "Welcome to my website");
define("SUBTITULO", "This is a subtitle...");
define("AVANCE", "This is a complementary text...");
define("NOTICIA", "The large text in english...");
?>
```

Para agregar otros idiomas, simplemente necesitaremos añadir una opción más al menú *select* HTML por cada idioma, para que le deje elegir ese idioma al usuario. Ese *select* debe tener un atributo **name** cuyo valor sea "idioma", para que genere $_POST["idioma"], que es lo que damos por supuesto en este código; luego de validar su presencia con *isset*, guardaremos el idioma elegido en una variable de sesión. Por supuesto, también debemos crear el archivo con los textos en ese idioma dentro de la carpeta "idiomas".

Podemos crear todas las páginas del sitio que queramos, con enlaces entre ellas, que memorizarán el idioma elegido por el usuario a lo largo de todas ellas (si la cantidad de páginas o el largo de los textos lo justifican, será conveniente crear varios archivos con las definiciones de los textos, quizás uno por sección del sitio, o, incluso, uno por cada página del sitio, para no obligar a leer completo un solo archivo extremadamente largo en todas las páginas del sitio; solo será cuestión de "avisar" con alguna variable en cuál sección o página nos encontramos, para leer el archivo respectivo, de forma similar a como lo estamos haciendo ahora para elegir el archivo de idioma).

Y con esto damos por finalizada nuestra introducción al manejo de sesiones y en PHP. Ya estamos en condiciones de identificar a nuestros usuarios, y podemos crearles contenidos personalizados y "perdurables" a lo largo de una sesión.

# Los bucles y los archivos de texto

# 8

## Recorriendo línea por línea la información almacenada

En este capítulo, interactuaremos con los más sencillos almacenes de datos **perdurables**, que son los **archivos de texto**. Prepararemos algunos de ellos para que las páginas de PHP los **lean** automáticamente y muestren sus datos dentro del diseño que hayamos preparado. En otros términos, cualquier cambio de contenidos sobre esos archivos de texto, se reflejará de inmediato en las páginas PHP. Además, aprenderemos cómo hacer que se **escriban** automáticamente dentro de esos archivos de texto nuevos datos, ya sea porque los recibimos de nuestros usuarios mediante formularios o porque decidimos guardar la información generada automáticamente por nuestras páginas en el momento en que se las utiliza (como en el caso de un contador de visitas de una página, en el que se guarda el número actualizado).

Para poder leer la información de estos archivos, será necesario dominar un concepto clave en programación: el **bucle** (o **ciclo**), un recurso que nos permitirá que leamos –"línea por línea" (renglón por renglón) o, incluso, letra por letra– el contenido de esos archivos de texto, como así también nos posibilitará el recorrido de la información almacenada en matrices y, más adelante, en bases de datos. En consecuencia, comenzaremos por aprender a utilizar las estructuras de bucles del lenguaje PHP y, luego, trabajaremos con archivos de texto, en los que haremos uso de los bucles como principal herramienta para manipularlos.

# Tipos de bucles: for, while, do while, foreach

Los bucles son estructuras que determinan el flujo de ejecución de nuestros bloques de código, pero no en forma "condicional" o "selectiva" –como en el caso de los **if**, los **else** y los *switch*–, sino que se especializan en la ejecución **repetitiva** de un determinado bloque de código PHP.

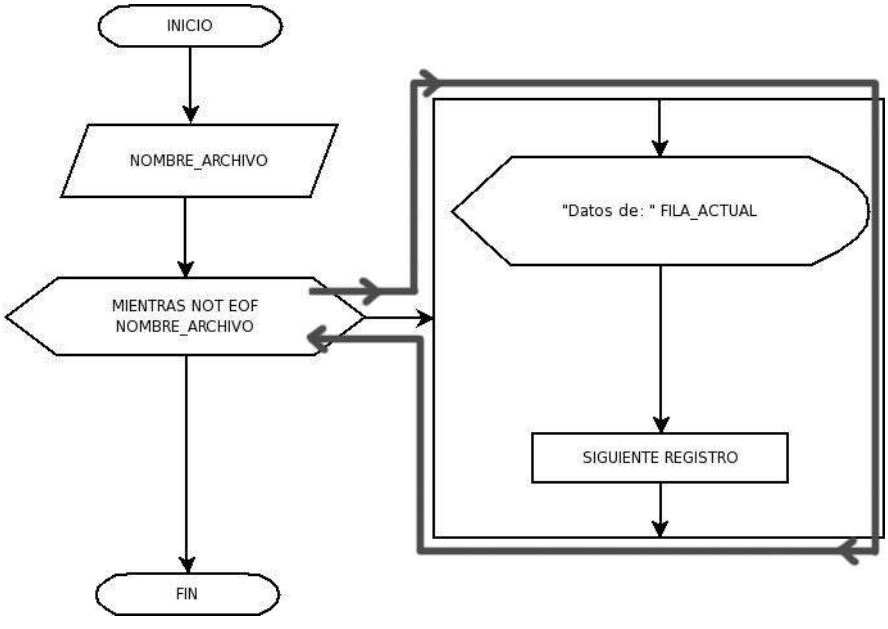


**Figura 8-1. Esquema de un bucle.**

Esto, que en un primer momento no parecería ser demasiado útil, se vuelve una herramienta imprescindible para poder trabajar con **grandes cantidades** de información. Pensemos –por ejemplo– en una página que deba mostrar un elemento **select** HTML que contenga una lista de años: desde el 1800 hasta el 2020. En lugar de hacer 220 órdenes echo, una para cada **option** del **select**, podemos hacer una sola orden echo pero dentro de un bucle, que comience en el 1800 y muestre de a un **option** por vuelta, **mientras no se haya llegado al 2020.** Un par de líneas de código, en vez de centenares.

Imaginemos que tenemos que mostrar miles de nombres de empleados de una em- presa, nombres almacenados dentro de un archivo de texto, a razón de uno por línea. Nada mejor que un bucle para que lea "de a una" línea de texto por vez.

Son muchísimos los usos de los bucles, ya lo veremos a medida que avancemos en este capítulo. Pero son ideales para tres tareas fundamentales:

1. leer datos de **matrices**,
2. leer filas de archivos de texto,
3. leer registros traídos de una **base de datos**.

Existen **cuatro** tipos distintos de bucles en PHP: **for**, **while**, **do while** y **foreach**. A continuación, veremos cuáles son los usos ideales de cada uno de ellos, mediante ejemplos listos para usar.

## Cómo repetir una acción en una cantidad fija de veces: el bucle for

Supongamos que precisamos hacer una simple lista, Que tiene que mostrar los 31 días de un mes, y cada línea debe decir "Hoy es el día ***X***", o sea, la primera línea debe decir "Hoy es el día 1", la siguiente "Hoy es el día 2", y así sucesivamente. Queremos lograr un resultado similar a esto:

```
Hoy es el día 1
Hoy es el día 2
Hoy es el día 3
etc...
Hoy es el día 31
```

En vez de escribir 31 líneas de código con 31 echos o prints (uno por cada frase, por cada día del mes), podemos lograr idéntico resultado mediante un bucle **for**, en solo un par de líneas de código:

```
for ( $dia=1; $dia<32; $dia=$dia+1){
       echo "<p>Hoy es el día $dia</p>";
}
```

Este tipo de ciclo o bucle **for** se utiliza para ejecutar una orden a repetición un **número exacto** de veces; es decir, el número de repeticiones debe conocer con anterioridad (en este caso, sabemos la cantidad de días del mes en cuestión: 31).

Como podemos observar, en el ejemplo anterior, este tipo de bucle denominado **for** tiene los siguientes elementos:

1. **Inicialización** de una **variable**: allí, se le da un valor inicial a una variable que será el elemento "clave" dentro del bucle. Esta inicialización se ejecuta antes de que comience el bucle, **por única vez**.

   En el ejemplo anterior, corresponde a esta parte:

   ```
   $dia=1;
   ```

   En este ejemplo, le damos el valor inicial **1** (pero puede ser cualquier otro).

2. **Evaluación** del valor de esa **variable**: se evalúa si el valor de la variable que fue inicializada todavía está dentro del rango definido, es decir, si la condición continúa siendo verdadera pese al cambio de valor de la variable. Se verifica justo antes de cada repetición del bucle. Si resulta que la condición sigue siendo verdadera, se ejecuta el código envuelto entre las llaves del bucle. Caso contrario, se finaliza el bucle y se continúa ejecutando el código posterior al bucle.

   En el ejemplo anterior, la condición o evaluación corresponde a esta parte del código:

   ```
   $dia<32;
   ```

   Si $dia vale menos de 32, será verdadera, y se ejecutará una vez el código envuelto entre las llaves del bucle, es decir, este código:

   ```
   {
     echo "<p>Hoy es el día $dia</p>";
   }
   ```

3. **Modificación** de esa **variable** *a posteriori*: se le asigna a la variable un nuevo valor, y esa modificación se realiza cada vez que se termina de ejecutar el código envuelto entre las llaves del bucle. Es decir, al finalizar cada vuelta.

   En el ejemplo:

   ```
   $dia=$dia+1;
   ```

   Vemos que se le suma **1** a lo que ya valía $dia en este momento, y el resultado, se vuelve a almacenar dentro de la misma variable $dia. Como consecuencia, el valor de $dia se **incrementará** en cada vuelta del bucle, haciendo posible que, cuando llegue al valor **32**, la condición del bucle se evalúe como falsa (32 no es "menor a 32") y, por lo tanto, se finalice. Sin esta modificación del

valor de la variable, nunca se saldría del bucle, se entraría en un bucle "infinito".

Observemos que esta modificación del valor de la variable utilizada como "índice" del bucle, no necesariamente implica **"incrementar"** el valor de la variable, también se podría **decrementar** y, en ese caso, la condición a evaluar debería plantearse en términos de si la variable aún es "mayor" a cierto valor mínimo:

```
for( $faltan=10; $faltan>0; $faltan=$faltan-1){
   echo "<p>Están faltando $faltan</p>";
}
```

Otra observación es que la modificación (ya sea incremento, o decremento) no tiene por qué ser "de uno en uno". Podemos, por ejemplo, ir de dos en dos, de tres en tres o de diez en diez, lo que se necesite:

```
for ( $par=0; $par<100; $par=$par+2){
   echo "<p>Vamos por $par</p>";
}
```

Cualquiera sea el tipo de **modificación** que hagamos con el valor de la variable clave, el objetivo es que tome todos los valores por los que decidamos que pase (en este último ejemplo, decidimos que recorra todos los números pares, de dos en dos, desde el cero hasta el 100 inclusive), hasta que, en un determinado momento, el valor "se salga" del rango (con el valor 100, en este ejemplo, que ya no es menor a 100) y, por lo tanto, la condición sea evaluada como falsa, y se termine el bucle. ¡No queremos bucles infinitos!... debemos asegurarnos de que siempre, en algún momento, la modificación que definimos termine cambiando el valor de la variable a otro en el cual la condición ya no se cumpla.

Si escribimos, paso a paso, los valores que toman las variables, comprenderemos mejor el funcionamiento repetitivo de los bucles. Pondremos a prueba el bucle del ejemplo, mostrando los valores de las variables paso a paso.

En un primer momento, se ejecuta la **inicialización** de la variable clave, en la que nosotros mismos definimos que $dia vale **1**.

De esta manera, en el inicio, la **condición** fijada ($dia < 32) se evalúa como *verdadera*, ya que **1** realmente es menor a **32**.

Por lo tanto, **se ejecuta el código** envuelto entre las llaves del bucle (el echo) una vez, y se escribe esto en el lugar exacto en el que estaba escrito ese código dentro de nuestra página:

```
<p>Hoy es el día 1</p>
```

A continuación de esa ejecución del código envuelto entre las llaves del bucle, se ejecuta la **modificación** (el incremento, en este caso) de la variable $dia, que pasa a valer 2. Con esto, se termina la primera vuelta del bucle.

Luego, para comenzar la segunda vuelta del mismo, ya no se inicializa la variable $dia (recordemos que dijimos que se le da un valor inicial solamente una vez).

Directamente, se evalúa la **condición** sobre la base del nuevo valor que tomó la variable $dia (por ahora, un **2**). Como **2** todavía es menor a **32**, la condición es *verdadera* también en esta vuelta, y **se ejecuta** una vez más el código envuelto entre las llaves del bucle, el echo, que produce esta vez un resultado levemente distinto al anterior, ya que $dia ahora vale **2**, en consecuencia escribirá:

```
<p>Hoy es el día 2</p>
```

Pasado este momento, se ejecuta la modificación o **incremento**, y $dia pasa a valer 3. En la siguiente vuelta, la **condición** sigue siendo *verdadera*, y se escribe esto:

```
<p>Hoy es el día 3</p>
```

y $dia se **modifica**, y pasa a valer **4**. Y así sucesivamente... hasta que $dia llegue a valer **31**. Entonces, se escribe esto:

```
<p>Hoy es el día 31</p>
```

Y, a continuación, se ejecuta el **incremento**, y $dia pasa a valer **32**. Cuando el bucle vuelva a evaluar la **condición**, esta vez se considerará **falsa**, ya que **32** no es menor a **32** (es igual, pero no "menor"). Por lo tanto, se dará por finalizado el bucle y dejará de ejecutarse el código envuelto entre sus llaves, y se pasará a ejecutar el código siguiente al bucle (si hubiera algún código PHP más abajo de la llave de cierre del bucle).

Podemos deducir que, en un bucle ***for***, la variable definida en el inicio, evaluada en la condición y modificada en cada vuelta del bucle, **tomará todos los valores** del rango que hayamos definido. En cada vuelta –en cada nueva ejecución del bucle–, tomará un valor distinto (primero valdrá **1**, luego **2**, y así hasta que tome el valor **32**, que finalizará su ejecución). Entonces, el bucle terminará cuando la condición planteada resulte falsa. Por esta razón, nos aseguraremos que se cumpla esta condición para que no se produzca un "bucle infinito" (se lo denomina así aunque no es infinito, ya que, dependiendo de la

configuración de nuestro servidor Web, éste podría finalizar la ejecución de manera abrupta y, por lo tanto, suspender la disponibilidad de la página).

En la mayoría de las explicaciones de bucles tradicionales de los manuales de programación, se explica el funcionamiento de un bucle usando la palabra **índice** para referirse a la variable que se modifica en cada vuelta. De allí que sea extremadamente común –casi una convención universal–, encontrar que a la variable utilizada como "cuentavueltas" se la denomine, para abreviar, **$i** (la "i" viene de *index* o "índice").

Para concluir, definiremos un esquema abstracto de la estructura de un bucle **for:**

```
for ($valor_inicial; $valor_final; $modificacion_valor){
      Bloque a ejecutar, envuelto entre
      las llaves del bucle;
}
```

### El complemento perfecto de las matrices ordenadas y de los archivos de texto

Los bucles de tipo **for** son los más usados para recorrer las **matrices ordenadas** de índices numéricos consecutivos, ya que, mediante la función **count**(), que nos facilita el lenguaje PHP, conocemos la **cantidad de elementos (celdas)** de la matriz y, por lo tanto, la cantidad de vueltas que dará el bucle hasta recorrer todas las posiciones de esa matriz para leerlas o mostrarlas.

Recordemos que un bucle de tipo **for** necesita obligadamente que le proporcionemos el número máximo al que debe llegar su variable, que sirve de control. Como complemento, PHP proporciona funciones para almacenar, dentro de una **matriz**, todos los renglones de un archivo de texto (y los registros obtenidos de una consulta a una **base de datos**). Si se tienen almacenados los datos en una matriz, será muy simple conocer el número de elementos y, en consecuencia, la podremos recorrer mediante un bucle de tipo **for.**

Veamos, entonces, un ejemplo de cómo resulta simple y útil utilizar un bucle *for* para mostrar una sencilla **matriz** cargada con productos:

```
<?php
$productos = array(1 => "manzanas","naranjas","peras",
"uvas");

$cuantos = count($productos); /* La función count devuelve
un número, el de la cantidad de celdas de la matriz que le
```

```
proporcionamos dentro de sus paréntesis, $productos en
este caso).*/

for ($i=1; $i<=$cuantos; $i=$i+1){
   print("<p>El producto".$i."es:".$productos[$i]."</p>");
}
?>
```

Esto generará el siguiente código HTML:

```
<p>El producto 1 es: manzanas</p>
<p>El producto 2 es: naranjas</p>
<p>El producto 3 es: peras</p>
<p>El producto 4 es: uvas</p>
```

Por lo tanto, mostrará en la pantalla del navegador lo siguiente:

El producto 1 es: manzanas

El producto 2 es: naranjas

El producto 3 es: peras

El producto 4 es: uvas

A modo de ejemplo, imaginemos la simpleza que significa almacenar en una matriz miles de datos de un catálogo de productos de un sitio de comercio electrónico. En este caso, con un par de líneas de código, podemos generar el HTML que mostrará el listado completo.

Otro caso: una matriz con un listado de países del mundo (proveniente de un archivo de texto o de una base de datos), que podremos fácilmente mostrar, dentro de **options** de un elemento **select** de HTML. Esta facilidad la aprovecharemos también en listados de años, de meses, de mensajes de un foro, de comentarios realizados a un texto cualquiera. La lista de aplicaciones es casi infinita.

## El concepto de contador

Hemos visto que los tres elementos de un bucle **for** eran la **inicialización** de una variable, su **evaluación** y su **modificación.** Este último elemento aplica una estructura típica en programación, denominada contador. Un contador lleva la

cuenta de algo en una variable. Es decir: guarda, en una misma variable, el resultado de sumar "lo que ya valía esa variable", más un número.

Por ejemplo, pensemos en alguien que cuenta las vueltas de un auto alrededor de un circuito de carreras. Cuando arranca, el contador está en cero:

```
$vueltas = 0;
```

Cuando el auto complete la primera vuelta y pase por la línea de largada, le debe sumar "uno" a la cantidad anterior de vueltas, por lo que quedaría así:

```
$vueltas = $vueltas + 1;
```

Esto se leerá así: "guardar en $vueltas lo que ya valía $vueltas, más 1".

De esta manera, al terminar la primera vuelta, $vueltas vale **1**, ya que $vueltas valía inicialmente **0** y, como le sumamos **1**, 0 + 1 = 1.

Al terminar de dar otra vuelta, se vuelve a ejecutar esa misma operación:

```
$vueltas = $vueltas + 1;
```

Como $vueltas valía **1**, al sumarle **1**, pasa a valer **2**. Ahora $vueltas vale 2, y así sucesivamente.

La estructura es siempre la misma: que la variable sea igual a lo que valía, más un número.

Si el número que estamos sumando es un **1** (que es lo más común), podemos abreviar la notación del contador de esta manera:

```
$vueltas++;
```

Eso es, exactamente, lo mismo que escribir: $vueltas = $vueltas + 1. Por eso, una notación típica en los bucles **for** es la siguiente:

```
for ($i=0; $i<$maximo; $i++){
// etc.
}
```

El tercer elemento del bucle que dice **$i++** es un contador que en cada vuelta del bucle le suma **1** a lo que ya valía **$i** .

Pero el número que le sumamos no debe ser necesariamente un **1**. También, podemos contar de dos en dos, de tres en tres, de cien en cien, o con el intervalo que queramos, **pero ese intervalo es fijo**; es decir, siempre es el mismo mientras usemos ese contador. Si empezamos a contar de dos en dos, ese contador contará siempre de dos en dos, no será posible que en una vuelta sume 1 y en otra 3, por ejemplo (esto lo veremos cuando aprendamos la estructura "acumulador"). Pero, aquí, en el "contador", el valor que se incrementa en cada

vuelta puede ser cualquier número, pero **a lo largo del funcionamiento del contador ese número no cambiará**.

Veamos un ejemplo de "dos en dos":

```
$medias = 0;
```

La variable $medias vale **0** hasta ahora.

Retiro un par de medias del lavarropas y lo pongo en el cajón:

```
$medias = $medias + 2;
```

La variable $medias ahora vale "2". Agrego otro par:

```
$medias = $medias + 2;
```

La variable $medias pasa a valer **4**, y así sucesivamente. A simple vista, es muy fácil reconocer esta tipo de estructura, ya que veremos el nombre de una variable, un signo igual, otra vez el nombre de la misma variable, y luego la operación que la modifica en un número fijo:

```
$vuelta = $vuelta + 1;
```

Asimismo, utilizamos la estructura de contador para que el valor de una variable **disminuya** en intervalos fijos:

```
$falta = $falta - 1;
```

En el caso de que el decremento sea de a **1** por vuelta, también existe una notación abreviada, con un doble signo menos:

```
$falta--;
```

### El concepto de acumulador

Muy similar a la estructura anterior, el acumulador es una variable que va almacenando valores dentro de ella, pero cada incremento no necesariamente es igual al anterior, como en el caso del contador, donde siempre sumábamos la misma cantidad debido a que era una constante (un número fijo). En un **acumulador**, el incremento se lee de una **variable**, por lo cual puede ser siempre **distinto**. Es muy útil para almacenar totales y subtotales, calcular el importe de una compra sumando los importes parciales, etc.

A simple vista, se reconoce esta estructura porque consiste en una variable que es igual a sí misma más otra variable:

```
$importe = $importe + $precio;
```

Por ejemplo, pensemos en alguien que está acumulando la suma de una compra. Cuando arranca, el acumulador está en cero:

```
$importe = 0;
```

Cuando ingresemos el precio del primer producto, debemos sumar ese precio al acumulador, por lo que quedaría así:

```
$importe = $importe + $precio;
```

Esto se leería: "guardar en $importe lo que ya valía $importe más lo que vale la variable $precio".

De esta manera, al terminar de cargar el primer producto (supongamos que costaba $50), los números serán: 0 + 50 = 50. La variable $importe ahora vale **50**.

Al terminar de cargar otro producto (supongamos que esta vez costaba $345), se vuelve a ejecutar la misma operación:

```
$importe = $importe + $precio;
```

Como $importe valía 50, los números serán: 50 + 345 = 395. La variable $importe ahora vale 395. Y así, sucesivamente...

La estructura del acumulador, entonces, consiste en una variable que es igual a su valor anterior, más un nuevo número variable, que no es fijo. De la misma manera que con los contadores, también existe una abreviatura para los acumuladores:

```
$importe += $precio;
```

Anteponiendo el signo **+** al operador de asignación, se produce primero la suma de

$precio al valor anterior de $importe, y luego se le asigna a $importe el resultado ya incrementado. Lo mismo en el caso de un decremento:

```
$importe -= $precio;
```

Muy pronto, en este capítulo, aplicaremos estas estructuras –tanto el contador como el acumulador– dentro de bucles.

## Cómo recorrer una matriz con foreach

Pasemos a descubrir las ventajas de otro tipo de bucle distinto al for. En este caso, nos ocuparemos del tipo de bucle denominado **foreach**, que tiene una utilidad específicamente orientada a recorrer matrices (si se le pasa como parámetro algo que no sea una matriz, generará un error). La ventaja principal es que **foreach** puede recorrer matrices con índices **no consecutivos**, desordenadas o de índices alfanuméricos, sin que sea un problema (recordemos que el **for** solo trabaja con matrices de índices numéricos ordenados y con un intervalo constante entre ellos, ya que ése es el incremento que se le puede dar a su variable "índice").

Este tipo de bucle tiene dos sintaxis posibles. Comencemos por la "mínima", que es la siguiente:

```
foreach ($matriz as $valor){
// Bloque a ejecutar donde usemos el dato leído en $valor
}
```

En cada vuelta del bucle, leerá **una celda** de la matriz y colocará su valor dentro de la variable $valor. En la siguiente vuelta, leerá el elemento siguiente de la matriz, y así hasta terminar el contenido de la matriz completa.

Veamos el siguiente ejemplo:

Archivo foreach.php:

```
<?php
$animales[4] = "Perro";
$animales[5] = "Gato";
$animales[21] = "Tortuga";
$animales[3] = "Hamster";
$animales[45] = "Canario";

foreach ($animales as $valor){
      print("<p>El animal actual es ".$valor."</p>");
}
?>
```

Este código producirá el siguiente código HTML:

```
<p>El animal actual es Perro</p>
<p>El animal atual es Gato</p>
<p>El animal actual es Tortuga</p>
<p>El animal actual es Hamster</p>
<p>El animal actual es Canario</p>
```

La segunda sintaxis posible de **foreach** nos permite que, en cada vuelta del bucle, no solo leamos el **valor** de la celda de la matriz, sino también el **índice** de esa celda, y lo almacenemos en otra variable:

```
<?php
```

```
$animales[4] = "Perro";
$animales[5] = "Gato";
$animales[21] = "Tortuga”;
$animales[3] = "Hamster";
$animales[45] = "Canario";

foreach ($animales as $clave => $valor){
      print("<p>El elemento de índice: ".$clave." contiene
el valor: ".$valor."</p>");
}
?>
```

Este tipo de bucle **foreach** (comúnmente utilizado con su sintaxis simple, la que solo lee un valor), debe ser la primera opción cuando recorremos una **matriz**, debido a su eficiencia y sencillez.

## Cómo repetir algo una cantidad desconocida de veces: el while

Algunas veces, tendremos que repetir una acción hasta que “suceda algo”, pero **no sabremos**, a ciencia cierta, **cuándo** sucederá “ese algo”. Incluso, ni siquiera sabemos si alguna vez realmente sucederá (es decir: puede no ejecutarse ni siquiera una sola vez el contenido del bucle). Para esos casos, no nos sirve ni un bucle **for**, ni un **foreach**. Por ejemplo: mientras una función de envío de *e-mails* devuelva “verdadero”, enviar el siguiente *e-mail*.

En estos casos, el tipo de bucle ideal es el **while**. Está especialmente pensado para plantear **condiciones booleanas** (tal como las que usamos en los condicionales, aquellas que se pueden evaluar únicamente como “verdaderas” o “falsas”).

Veamos un ejemplo, llamemos a este archivo **while.php**:

```
<?php
while($ganador != "SI"){

      $ganador = leerApuesta();
```

```
}
?>
```

En cada vuelta, este bucle lee el resultado de una apuesta imaginaria, proporcionada por una supuesta función denominada **leerApuesta** (por supuesto, hasta que no sepamos definir esta función será imposible probarlo). Por lo tanto, no sabemos cuántas vueltas dará antes de encontrar que el valor de $ganador es SI. Para este tipo de bucles –que dependen de una condición en la que no podemos prever cuándo será verdadera–, es ideal el uso de un bucle de tipo **while.**

Veamos un ejemplo más para que notemos lo complejo que puede resultar la utilización del tipo de bucle "incorrecto". En el cuadro que sigue, se muestra cuán complejo resulta un **while** para recorrer una matriz (tarea que realiza con mucha mayor simpleza el bucle **foreach**):

```
<?php
while (list($clave,$valor) = each($_POST["numeros"])){

      print("<p>El elemento ".$clave." contiene un
".$valor."</p>");

}
?>
```

Este código recorrerá la matriz $_POST["numeros"] sin importar cuál sea su longitud, o de qué tipo sean sus índices (alfanuméricos o numéricos e, incluso, desordenados). A continuación, veremos cómo operan estas dos funciones accesorias que hemos necesitado agregar: por un lado, la función **list**, y, por el otro, la función **each.**

## Las funciones list y each

La función **each** captura **un elemento completo** de una matriz, el "actual": aquel al cual el puntero o índice interno del bucle, que lee esa matriz, esté apuntando en ese momento y, además de traer ese elemento completo, con sus dos subelementos (recordemos que un elemento de una matriz es un par "índice-valor", por ejemplo, [0]="algo"), desplaza el puntero al siguiente elemento de la matriz. Es decir: la matriz queda preparada para que se lea la celda siguiente en la próxima vuelta del bucle.

Cuando se ejecuta la función **each**, a su izquierda **la debe preceder una variable**, que es donde quedará almacenado todo el elemento de la matriz leído por **each**; esta función **each** se encarga de convertir a esa variable ("$este", en el ejemplo siguiente) en una **matriz**, con cuatro celdas, una para cada uno de los cuatro datos que entregará:

Ejemplo de **each:**

```
<?php
$datos["nombre"] = "Juan Pérez";
$datos["edad"] = 24;
$datos["estado"] = "casado";
$datos["sueldo"] = 800;

$este = each($datos);
// Convierte a $este en una matriz de 4 celdas

//Para visualizarlo, imprimiremos las 4 celdas de $este:
print("<p>El índice de este elemento es
".$este[0]."</p>");
print("<p>El valor de este elemento es ".$este[1]."</p>");
print("<p>El índice de este elemento es
".$este["key"]."</p>");
print("<p>El valor de este elemento es
".$este["value"]."</p>");
?>
```

Notemos que **each** convirtió a $este en una **matriz** de **cuatro** elementos: los dos primeros tienen índices numéricos: el índice del primer elemento creado es un **0**, el del segundo un **1**, y contienen, respectivamente, el que era el **índice del elemento** de la matriz $datos que se leyó en esta vuelta del bucle ("nombre", en este ejemplo), y el **valor de ese mismo elemento** ("Juan Pérez", en este caso).

Luego, **el tercero y cuarto elementos** de esa matriz denominada $este, que fue creada por **each**, **vuelven a contener la misma información repetida** ("nombre" y "Juan Pérez"), pero, esta vez, con índices **alfanuméricos** llamados *key* y *value* (clave y valor).

En el ejercicio anterior, para ganar en claridad, hemos simplificado y utilizamos solo las primeras dos celdas creadas por **each:** en la primera almacenamos el índice y, en la segunda, el valor de ese elemento seleccionado. Pero, en lugar de adjudicarle esos dos datos a una variable/matriz (como $este),

se los adjudicamos mediante la función **list a dos nuevas variables** que recibirán esos datos: list($clave,$valor).

La función **list** guarda en la variable **$clave** la **clave** del elemento de la matriz leído por **each** y, en ***$valor,*** guarda el **valor** de ese elemento. Otra vez, "nombre" y "Juan Pérez", respectivamente.

Entonces, hacemos el bucle **while** "mientras" el list($clave,$valor) **sea igual** al **each** de $numeros. Es decir, mientras se pueda producir esta **asignación** de datos de **each** hacia **list**. Mientras **each** pueda leer datos de una celda de la matriz original, y pueda pasársela a las dos variables que **list** tiene preparadas, a la espera.

Copiamos el código nuevamente aquí para volver a examinarlo prestando atención a los detalles:

Archivo list-each.php:

```
<?php
while (list($clave,$valor) = each($_POST["numeros"])){
      print("<p>El elemento ".$clave." contiene un
".$valor."</p>");
}
?>
```

En cada vuelta del bucle, las variables **$clave** y **$valor** contendrán el **índice** y el **valor** del elemento que **each** acaba de leer.

Se saldrá del bucle cuando sea **falsa** esa **asignación** (cuando el valor que lea **each** de la matriz original ya no se pueda transferir a **list...** porque **each** no haya podido leer nada); es decir: cuando se haya recorrido hasta el último elemento inclusive de la matriz $numeros, y ya no quede ningún elemento más que se pueda leer con **each**.

Este método es una alternativa al **foreach** para recorrer matrices de índices alfanuméricos, o numéricos no consecutivos. Desde ya que **foreach** es mucho más simple, pero no todos los programadores lo utilizan, y nos encontraremos muchas veces con códigos que utilizan **while** combinado con **list** y con **each**. Ya estamos en condiciones de comprender esos códigos.

## El bucle do-while

Una variante del bucle **while**, que se utiliza menos, es la del bucle **do-while**. Es similar a la primera, pero su condición **se evalúa al final** de cada vuelta, no al

principio. De este modo, se "garantiza" que se ejecutará **al menos una vez** el bloque de código envuelto entre las llaves del **do-while**, ya que la condición se evaluará luego de la primera ejecución de ese bloque de código y no cuando se inicia el bucle. En un bucle **while** normal, ocurría lo contrario, puesto que la condición se ejecutaba antes de la primera vuelta, por lo tanto, podía suceder que fuera falsa, y que nunca se ejecutara el código dentro del **while**.

La sintaxis de este tipo de bucle es la siguiente:

```
<?php
$limite = 5;
do {

      echo $limite;

} while ($limite > 9);

?>
```

El **echo** encerrado entre las llaves del bucle se ejecutará, al principio, una sola vez y, luego, cuando se evalúe la condición, se comprobará su falsedad ($limite vale **5** y, por lo tanto, no es mayor a nueve), en consecuencia, se dará por terminada la ejecución. Pero **ya se ejecutó** una vez el código encerrado entre sus llaves.

Prestemos atención al detalle de que la línea que define la condición, en el **do-while** debe finalizarse con un punto y coma (a diferencia del **while**, que no lo lleva).

# Condicionales dentro de bucles

Una técnica muy utilizada es la de evaluar, dentro de un bucle, algo "a repetición", para tomar dos o más caminos alternativos en cada vuelta del bucle. Por ejemplo, pensemos en un bucle que muestra un listado de datos (una tabla con varias filas), y queremos que solamente en las filas pares se agregue un class="par" dentro del *<tr>* respectivo. Necesitamos saber en cada vuelta del bucle si se trata de una fila par, o no.

Para eso, colocaremos un condicional dentro del bucle, de esta manera:

```
<?php
```

```
$productos = array(1 => "manzanas","naranjas","peras",
"uvas");

$cuantos = count($productos);

echo "<table>";

for ($i=1; $i<=$cuantos; $i=$i+1){

      if ($i%2 == 0){
         echo '<tr class="par"><td>La fila de
'.$productos[$i].' es par</td></tr>';
      } else {
         echo '<tr class="impar"><td>La fila de
'.$productos[$i].' es impar</td></tr>';
      }
}

echo "</table>";
?>
```

Si simplemente definimos en la hoja de estilos CSS un color distinto para alguno de los class “par” o “impar”, lograremos que nuestra tabla se vea así:

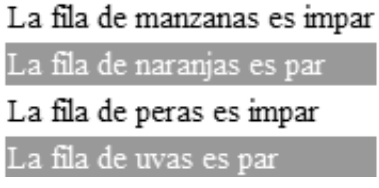


**Figura 8-2. Resultado generado gracias a un condicional dentro de un bucle.**

Esto puede lograrse con solo con esta breve declaración en la hoja de estilos CSS:

```
.par {
background-color: #999;
```

```
color: #fff;
}
```

Muy pronto, en este mismo capítulo, volveremos a utilizar esta técnica.

## La técnica de la señal o flag

Una de las aplicaciones más comunes de los **condicionales dentro de bucles**, es la técnica denominada **flag** (bandera o señal). Sirve para detectar si algo sucedió, y memorizarlo hasta salir del bucle.

Consiste en tres pasos:

1. Declarar y darle un **valor inicial** a una variable **antes** de comenzar un bucle.
2. Colocar un **condicional** dentro del bucle que potencialmente pueda llegar a **cambiar** el valor de esa variable.
3. Y verificar, luego del **cierre** del bucle, si el **valor** inicial de la variable sigue igual o fue cambiado (que indicaría que el condicional se ejecutó).

Esta técnica es muy utilizada para recorrer grandes cantidades de información tratando de detectar "si pasó algo".

Veamos un ejemplo que trata de detectar un valor en especial dentro de una serie de valores mezclados:

```
<?php
$claves = array(1 =>
"secreta","ultra","anonima","123456");

$cuantos = count($claves);

// Declaramos la señal
$encontrada = "no";

for ($i=1; $i<=$cuantos; $i=$i+1){

// Suponemos que llega de un formulario
```

```
if ($_POST["clave"] == $claves[$i]){
      // Si se encuentra, modificamos el valor de la señal:
      $encontrada = "si";
}
}

// Fuera del bucle, preguntamos por el valor de la señal:
if ($encontrada == "si"){
      echo '<p>Encontrado!</p>';
} else {
      echo '<p>NO se encontró</p>';
}
?>
```

# Los archivos de texto

## Un contenedor simple y permanente, para datos no muy confidenciales

Hasta ahora, las únicas técnicas que hemos aprendido para solicitarle un dato a un usuario y **no olvidarlo** inmediatamente después de cargada la página, fueron las de almacenarlo en *cookies* o en sesiones.

Pensemos en el ejemplo de alguien que completa un formulario para ser "usuario registrado" de nuestro sitio Web (lo que le permitiría acceder, por ejemplo, a una página privada, reservada solo para quienes se registraron con un nombre de usuario y una contraseña). Si ese formulario almacenara los datos en *cookies* o en sesiones, sería casi imposible para nosotros –dueños del sitio– recuperar esos datos para agregar manualmente a este usuario a la lista de personas "autorizadas". No tenemos acceso a las *cookies* (ya que se guardan en la máquina del usuario) y casi nunca obtendremos acceso a las sesiones, debido a que se almacenan en una carpeta temporal del servidor *(hosting)* que usemos, por lo que, salvo que el servidor sea nuestro y que tengamos acceso físico o remoto a su disco, no podremos acceder a esa información.

En lugar de eso, sería mucho más cómodo para nosotros –y para nuestros usuarios también– que sus datos **se almacenaran solos** en algún lugar y que se leyeran inmediatamente, para habilitarle el acceso al usuario desde ese mismo momento. Ese lugar podría ser **un archivo de texto** en el servidor.

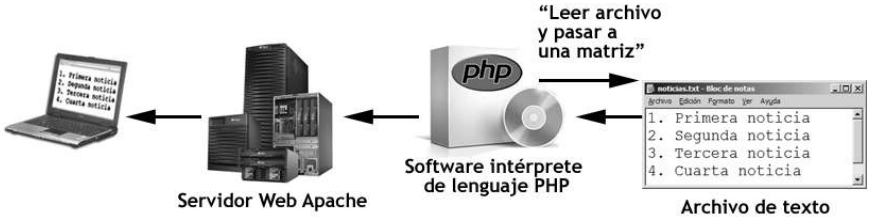


**Figura 8-3. Uso de archivos de texto como lugar donde almacenar datos y de donde leerlos automáticamente.**

De las formas de **almacenamiento permanente** (no cuentan las variables ni las matrices porque son "volátiles"), el guardar los datos en un **archivo de texto** es la más simple de utilizar (y, por supuesto, también las **bases de datos**, que trataremos en otro capítulo; sin embargo, quizás nos sorprendería saber que muchas bases de datos, en el fondo, guardan sus datos en archivos de texto ubicados en el servidor).

Aprenderemos de qué maneras se puede **guardar** automáticamente información dentro de archivos de texto, y de qué formas se puede **leer** esa información ya almacenada en dichos archivos.

La interacción automática entre nuestras páginas PHP y los archivos de texto (salvo una sola excepción que ya veremos), sigue siempre estos tres pasos consecutivos, en este orden:

1. **Abrir** el archivo.
2. Hacer alguna **operación** con el contenido de ese archivo.
3. **Cerrar** el archivo.

A continuación, veremos estos pasos en detalle:

## 1. Modos de apertura de archivos

Comencemos por la **apertura** de un archivo: el programa intérprete de PHP necesita abrir el archivo antes de poder utilizarlo (tal como nosotros necesitamos "abrir" un archivo de texto con un editor). Esto se realiza mediante la función **fopen** (*file open* o **apertura de archivo**). Esta función requiere que le pasemos dos parámetros: la **ruta** hasta el archivo que se abrirá y el **modo** de apertura; es decir, para cuál operación es que lo estamos abriendo (para lectura, escritura, agregado de datos, o la combinación de dos de estas operaciones).

A continuación, veremos cuáles son algunos de los posibles **modos de apertura** de un archivo.

| Modo | Descripción | Acción permitida |
|---|---|---|
| r | **Read** (leer) | Abre el archivo para **leer** sus datos. |
| w | **Write** (escribir) | Lo abrimos para **escribir**. Lo que se escriba reemplazará todo otro contenido anterior que hubiera en el archivo (lo sobrescribirá). Si al ejecutar fopen en modo "w" el archivo de texto especificado no existía, intentará crearlo. |
| a | **Append** (agregar) | Abre para **añadir**. Añade los datos *al final* de los otros datos que hubiera en el archivo. Si el archivo no existe, lo crea. |
| r+ | **Read**, más su complementario | Abre para **añadir** datos, y además permite **leerlos**. En caso de añadir, los datos serán añadidos *al principio* del archivo. |
| w+ | **Write**, más su complementario | Abre para **escribir** y luego **leer**. Si el archivo existía, suprime todo lo que hubiera previamente en ese archivo. Si el archivo no existe, lo crea vacío. |
| a+ | **Append**, más su complementario | Abre para **añadir** y **leer**. A diferencia del modo "r+", añade los datos *al final* del archivo, sin eliminar lo que ya contenía el archivo. Si el archivo no existe, lo crea. |

**Tabla 8-1. Modos de apertura de un archivo de texto.**

Existen algunos otros modos adicionales, pero como no son tan usados, los dejamos para una eventual profundización sobre el tema que quiera realizar el lector.

Para comprender cómo funciona el primer paso de la apertura de un archivo en un "modo" determinado, veremos un ejemplo de uso de **fopen**:

```
<?php
$abierto = fopen ("archivo.txt", "r");
?>
```

Esta línea abre un archivo para **leer** su contenido (ya que especificamos el modo "r" –read–), y crea una variable ($abierto) que será el **identificador de ese recurso** disponible (el archivo ya abierto); con ese identificador nos referiremos,

de aquí en más, al archivo ya abierto durante la ejecución de las siguientes operaciones de lectura, agregado o escritura.

Además, como esa variable contiene el resultado de la operación de apertura, según el intento haya sido "verdadero" o "falso", podemos preguntar con un **if si se pudo abrir** el archivo, si devolvió un "verdadero" la ejecución de esa función, antes de seguir realizando otras tareas posteriores que den por sentado que el archivo fue abierto:

```
<?php
if ($abierto = fopen ("archivo.txt", "r")){

      // Aquí haríamos alguna operación...

} else {

      echo 'No pudo abrirse el archivo';

}
?>
```

### 2. Operaciones posibles: lectura, escritura, agregados de datos

Una vez abierto el archivo, seguramente será de nuestro interés **hacer algo**, es decir, realizar alguna operación con los datos del archivo abierto (si no, ¿para qué lo abrimos?).

Las posibles operaciones son:

- la **lectura** de datos del archivo,
- la **escritura** de nuevos contenidos dentro del archivo, que reemplacen todo otro contenido anterior,
- y el **agregado** de nuevos datos al archivo, sin eliminar lo que existía dentro del archivo.

A estas tres operaciones las describiremos en detalle unos párrafos más adelante, en este mismo capítulo.

### 3. Cierre de un archivo

En todos los casos, terminada alguna de esas tres posibles operaciones que acabamos de mencionar, **cerraremos** el archivo, para que no ocupe memoria innecesariamente (de la misma manera que cuando trabajamos en nuestra

propia PC, donde cerramos los archivos a medida que los dejamos de usar, para evitar sobrecargar la memoria RAM de nuestra máquina).

El **cierre** de un archivo se realiza con la función fclose (*file close* o cerrar archivo), que requiere que entre sus paréntesis especifiquemos la variable que hacía de identificador de recurso que creamos al momento de abrir el archivo (en nuestro ejemplo: $abierto).

Ejemplo de cierre de un archivo abierto:

```
<?php
$abierto = fopen ("archivo.txt", "r");
// Aquí leeríamos, o escribiríamos, o añadiríamos algo...

// y finalmente, lo cerraríamos:
fclose ($abierto);
?>
```

Ahora que ya sabemos **abrir** y **cerrar** un archivo, estamos preparados para aprender en detalle en qué consisten las **operaciones** realmente importantes que se pueden realizar con los contenidos de un archivo de texto: **operaciones de lectura, escritura o agregado de datos**, que realizaremos una vez que el archivo esté abierto y antes de cerrarlo.

# Formas de leer datos desde un archivo de texto

Para comenzar, le daremos a las páginas PHP la capacidad de leer automáticamente datos almacenados con anterioridad dentro de un archivo de texto.

Existen varias **funciones** que permiten realizar distintos tipos de lecturas, ya sea del archivo completo, de sus renglones uno a uno, de unos pocos caracteres, o de un solo carácter por vez. Veamos cada una de ellas y en qué casos conviene aplicarlas.

## Leer línea por línea con la función file

Comencemos por una de las funciones más simples y usadas que posee el lenguaje PHP para leer archivos de texto, ya que es la única que **no requiere**

realizar los pasos de apertura y cierre, se usa directamente, sin preámbulos. Se trata de la función **file** (que significa "archivo"). Esta función permite acceder al contenido de un archivo de texto especificado entre sus paréntesis, y leer su contenido completo, convirtiendo a cada **línea (renglón)** del archivo de texto, en una **celda** dentro de una matriz, para que después la recorramos con un **bucle** (tal como aprendimos en el capítulo anterior), y utilicemos esos datos, ya sea para realizar alguna operación o para mostrarlos directamente al usuario.

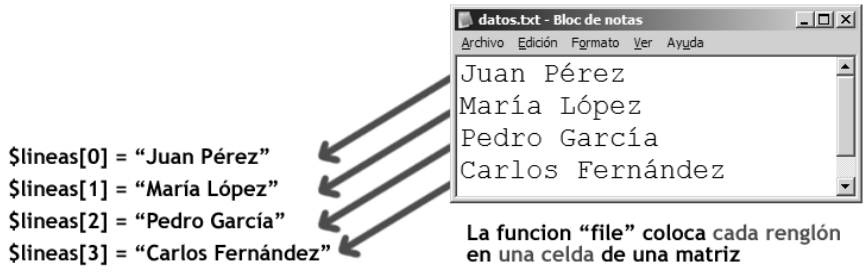


**Figura 8-4. Proceso que realiza la función file.**

El único parámetro que necesita que escribamos entre sus paréntesis es la **ruta** hacia el archivo que se leerá (si el archivo está en la misma carpeta que el archivo PHP que se está ejecutando, será suficiente con que escribamos el nombre del archivo):

```
<?php
$archivo = "datos.txt";// La ruta al archivo

$lineas = file($archivo);
/* A $lineas lo convertimos en una matriz con tantas
celdas como líneas -renglones- tenía el archivo de texto
*/

// Ahora, vamos a recorrer esa matriz con un bucle:
for ($i=0; $i<count($lineas); $i++) {

      print ("<p>La línea ".($i+1)." contiene:
".$lineas[$i]."</p>");

}
?>
```

**Explicación de este ejemplo:**

Primero, hemos guardado la ruta que apunta hacia el archivo que se leerá dentro de una variable denominada **$archivo**; a continuación, almacenamos dentro de **$lineas** el resultado de haber ejecutado la función **file** en el archivo especificado en **$archivo**, luego, **$lineas** quedó convertida en una **matriz**, que contiene tantas celdas como renglones tenía el archivo de texto leído (se considera una línea hasta que encuentra un **enter**, un salto de línea).

Es decir, que **$lineas[0]** tendrá almacenada la **primera línea** del archivo de texto (recordemos que las matrices numeran sus celdas desde cero, por esta razón, lo ajustamos en el momento de mostrarlo, sumándole 1 en la parte que decía: ***$i+1***, para evitar que dijese "La línea 0..." y hacer que diga "La línea 1..."), **$lineas[1]** contendrá la **segunda línea** y así, sucesivamente. A continuación, en este ejemplo, hemos realizado un bucle **for** para recorrer la matriz **$lineas** desde su primera hasta su última celda (para saber cuántas celdas tiene, ejecutamos la función **count**). Dentro del bucle, el intérprete de PHP simplemente ha ido escribiendo, celda por celda, todo el contenido de la matriz **$lineas**, por lo tanto, indirectamente, hemos mostrado todo el contenido del archivo de texto original, línea por línea, de principio a fin.

Haciendo uso de esta función, podríamos crear una aplicación que nos permitiera mostrar frases aleatorias (o imágenes aleatorias) dentro de nuestras páginas, simplemente "mezclando" la matriz una vez leída, y usando solo una de sus celdas, que siempre contendrá algo distinto.

Si lo hacemos con frases, debemos asegurarnos de que cada una ocupe **un solo renglón** dentro del archivo de texto, es decir, se considerará una frase diferente cada vez que exista un salto de línea, un **enter**.

Por ejemplo, éste sería el archivo frases.txt:

```
Piensa como piensan los sabios, mas habla como habla la
gente sencilla – Aristóteles.
Tanto si piensas que puedes, como si piensas que no
puedes, estás en lo cierto – Henry Ford.
Todo lo que somos es el resultado de lo que hemos pensado;
está fundado en nuestros pensamientos y está hecho de
nuestros pensamientos – Buda.
```

Este sería el código:

```
<?php
```

```
$archivo = "frases.txt";// La ruta al archivo

$frases = file($archivo);
/* A $lineas lo convertimos en una matriz con tantas
celdas como líneas -renglones- tenía el archivo de texto.
*/

shuffle($frases); /* Mezclamos el contenido de esa matriz
con la función shuffle, por lo cual, no sabemos cuál frase
quedó en el primer lugar, y la mostramos: */

echo ("<p>Nuestra frase del día es ".$frases[0]."</p>");

?>
```

Si mostráramos imágenes, ocurriría lo mismo, la única diferencia estaría en el contenido del archivo de texto, que ya no contendría frases sino nombres de imágenes, uno por renglón (supongamos que este archivo se llama **fotos.txt**):

```
perros.jpg
gatos.jpg
elefantes.jpg
```

Este código, una vez leída y mezclada la matriz, y obtenida la primera de sus celdas, utilizará el nombre de foto que leyó para completar una etiqueta *img*:

```
<?php
$archivo = "fotos.txt";
$fotos = file($archivo);
shuffle($fotos);

echo '<img src="fotos/'.$fotos[0].'" width="280"
height="200">';

?>
```

En muchas otras ocasiones, no nos será útil acceder al archivo para leerlo de a una línea completa por vez, como hace la función **file**, sino que tendremos almacenada **muy poca** información: apenas una sola palabra, o una sola línea, o un número. Por ejemplo, un contador de visitas de una página que tenga guardada la cifra de visitas dentro de un archivo de texto. Tal vez necesitemos abrir un archivo no para leerlo, sino para **escribir** datos en él.

Para estos casos, disponemos de varias funciones que, a diferencia de **file**, siempre ejecutan la misma rutina para acceder al archivo:

1. **Abren** el archivo.
2. Ejecutan alguna **operación** con los datos del archivo.
3. **Cierran** el archivo.

Para que nuestras páginas puedan **leer** datos almacenados en un archivo (además de la función **file**, que ya hemos visto y que no requería ni de apertura ni de cierre del archivo), aprenderemos a utilizar las siguientes funciones: **fpassthru**, **fread**, **fgetc**, **fgets**, **fgetss**.

A continuación, consideraremos en qué caso conviene utilizar cada una de ellas.

## Función fpassthru

La función **fpassthru** *(file pass through* o pasar a través del archivo) **lee** y, a la vez, **escribe** (no precisa ni un **echo** ni un **print**) **la totalidad** del contenido de un archivo de texto. Lo muestra entero, de principio a fin.

Podemos copiar y pegar el siguiente código en un archivo denominado noticia.php para probarlo:

```
<p>
<?php
$abierto = fopen ("noticia.txt", "r");
fpassthru ($abierto);
fclose ($abierto);
?>
</p>
```

Para que esto funcione, deberemos crear, con el bloc de notas, un archivo de texto denominado noticia.txt, y le escribiremos algunas líneas de contenidos (una supuesta noticia):

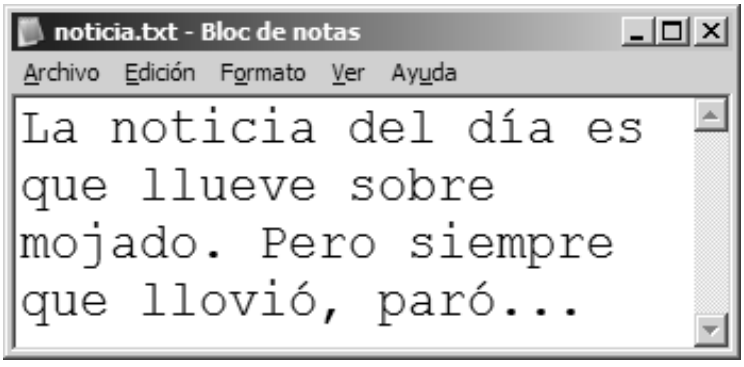


**Figura 8-5. Creamos el archivo de texto que almacenará los datos.**

El resultado de ejecutar la función **fpasstrhu** será que se mostrará, en el lugar exacto en el que llamemos a esa función, el texto completo del archivo al que apuntamos:

```
<p>La noticia del día es que llueve sobre mojado. Pero
siem- pre que llovió, paró...</p>
```

Esto nos mostrará el contenido completo de noticia.txt en la pantalla cuando entremos a noticia.php. Recordemos que esta función es similar a un **echo** del archivo completo.

No nos servirá usar esta función cuando el objetivo sea leer el archivo de texto para **almacenar**, en alguna variable, su contenido, o cuando precisemos pasar ese contenido a una función propia que le aplique algún otro **proceso** a los datos leídos. Solo sirve para **mostrar la totalidad** del archivo. Puede ser útil para que personas sin conocimientos de HTML modifiquen el contenido de un archivo de texto y luego lo suban por FTP al servidor, para que cambie automáticamente el contenido de la página noticia.php.

## Función fread

Esta función también permite leer un archivo de texto. A diferencia de **file**, que lee por renglones, y de **fpassthru**, que lee el archivo entero y lo muestra, la función **fread** *(file read* o lectura de archivo) lee una **cantidad** exacta de caracteres de un archivo; además del **identificador** del archivo abierto (que, en los ejemplos anteriores, denominamos $abierto), necesita que le indiquemos la **cantidad exacta de caracteres** que leerá (en ejemplo siguiente, 15).

¡Atención! Si en el archivo de texto hubiera un enter (salto de línea), lo considerará como DOS (2) caracteres, y no uno. ¿Por qué? Porque de verdad son dos caracteres lo que genera un salto de línea: son los caracteres \n -barra invertida y letra n de *New line*. Tengámoslo en cuenta en nuestros cálculos...

La cantidad máxima de caracteres que **fread** puede leer son 8192 (en realidad, se trata de *bytes* más que de caracteres, por lo tanto, se cuentan también los caracteres no imprimibles, como los saltos de línea, tabuladores, etc.). Entonces, es importante que consideremos que para leer archivos de más de 8Kb de tamaño, ya no podremos usar **fread**.

Veamos entonces un ejemplo de **fread**:

```
<?php
$abierto = fopen ("noticia.txt", "r");
$principio = fread ($abierto,18);
fclose ($abierto);

print ($principio);
?>
```

Esto mostrará solo los primeros 18 caracteres del archivo. Suponiendo que usamos el mismo archivo de texto anterior, cuando llamemos a **fread** con ese límite, obtendremos solamente esta parte del contenido del archivo de texto:

```
La noticia del día
```

Entonces, la función **fread** se usa para leer textos pequeños, "frases del día" o frases célebres, comentarios breves, números, etc.

## Función fgetc

Veamos otra función para leer datos de un archivo de texto. La función **fgetc** *(file get character* o traer un carácter de un archivo) como su nombre lo indica, lee **un solo carácter** de un archivo. En los (rarísimos) casos en los que hay que leer únicamente un carácter, se usa así:

```
<?php
$abierto = fopen ("noticia.txt", "r");
$caracter = fgetc ($abierto);
```

```
print ($caracter);/* imprime solo el primer caracter del
archivo; en nuestra anterior noticia, mostraría una " L".
*/
fclose ($abierto);
?>
```

Su utilidad radica en que puede usarse dentro de un **bucle** que recorra todo el contenido del archivo de texto **letra por letra**, o hasta que aparezca un determinado carácter especial de finalización.

Pero, para que esto ocurra, primero necesitamos aprender una función estrechamente ligada a esta tarea y que se denomina **feof**.

## Función feof

La función **feof** *(file end of file* o final de archivo del archivo) devuelve "verdadero" cuando se llegó hasta el **final** del contenido de un archivo.

Observemos, en el siguiente ejemplo, el uso de la función **feof**. La condición *booleana* que se usa dentro de este bucle **while** para seguir leyendo una letra más es que todavía "no sea el fin del archivo":

```
<?php
// Abrimos el archivo.
$abierto = fopen ("noticia.txt", "r");

// Inicializamos una variable.
$cadena = "";

// Mientras NO sea el final del archivo abierto:
while (!feof ($abierto)){

      // leemos un carácter nuevo.
      $nuevo = fgetc ($abierto);

      // Concatenamos ese carácter nuevo a lo anterior.
      $cadena = $cadena . $nuevo;
}
```

```
fclose ($abierto); // Cerramos el archivo.

print ($cadena); // Mostramos todo lo leído.
?>
```

Veamos ahora cómo mostrar el contenido del archivo hasta que se encuentre un **carácter** definido de antemano. Este ejercicio nos dará la oportunidad de aplicar otra vez el concepto de **anidar un condicional dentro de un bucle**, que vimos en este mismo capítulo. Este condicional se ejecutará una y otra vez (una vez por cada vuelta del bucle), aplicando la evaluación de su condición al valor "actual", el que se está leyendo en ese momento, en esa vuelta del bucle:

```
<?php
$abierto = fopen ("archivo1.txt", "r");
$cadena = "";

while ( false !== ($caracter = fgetc($abierto)) ){

      if ($caracter!="z"){
         $cadena = $cadena . $caracter;
         // Concatenamos cada caracter, uno por vuelta.

      } else {

         break;
         // Si es "z", salimos del bucle.
      }

}

fclose ($abierto);

print ($cadena);/* Muestra hasta la "z" o hasta el final
si no hubo ninguna "z" */
?>
```

Eso es todo con **fgetc**. Sumamente útil para leer archivos de texto largos, completos, o hasta cierto punto que indiquemos con algún carácter especial, pudiendo almacenar en una variable lo leído.

## Función fgets

La función **fgets** (obtener del archivo) es similar a **fread**, pero con la particularidad de que lee exclusivamente **dentro de una única línea** del archivo (por lo que es habitual para leer muy pocos caracteres, ya que para usarse en bucles es más sencillo utilizar **file**), y lee la cantidad de caracteres especificados menos uno; (en el siguiente ejemplo, en el que usaremos como límite un 15, leerá hasta el carácter número 14, inclusive). Si encuentra el fin del archivo, devuelve hasta el último carácter que le fue posible leer.

Ejemplo:

```
<?php
$abierto = fopen ("textos.txt", "r");
$caracteres = fgets ($abierto,10);
fclose ($abierto);

print ($caracteres);
?>
```

La función **fgets** es la más útil cuando solo precisamos leer unos **pocos caracteres** dentro de un archivo de texto, por ejemplo, en el caso de un archivo que almacene nada más que la cantidad de visitas a una página:

```
5587
```

Por más visitas que tengamos, siempre la cantidad de dígitos será muy poca (con apenas 7 dígitos tenemos para casi 10 millones de visitas, cosa que puede llevar bastante tiempo y, llegado el caso, podríamos modificar muy fácilmente la llamada a **fgets** para pasar a leer 8, 9, o más caracteres, y seguirá funcionando con decenas o centenas de millones de visitas).

## Función fgetss

La función **fgetss** (observemos la doble "s" al final) es idéntica a **fgets** pero **omite las etiquetas HTML y PHP** que encuentre dentro del archivo. Se usa principalmente para abrir un archivo XML o HTML (en general, ubicado en otro servidor, puede ser una noticia, el clima actual, etc.). Esta función se encarga de "limpiarlo" de etiquetas HTML, para luego –por ejemplo– aplicarle la hoja de estilos de nuestro propio sitio.

Vamos a aplicarlo en un ejemplo para entenderlo mejor:

```
<?php
$abierto = fopen ("clima.html", "r");
$caracteres = fgetss ($abierto,150);
fclose ($abierto);

print ($caracteres);
?>
```

Supongamos que el archivo "clima.html" contiene esto:

```
<p>Pronostico Provincial de las 06:00 horas.</p>
<p>Martes</p>
<p>Sudeste</p>
<b>Mañana:</b>
Nubosidad variable. Vientos regulares a moderados del
sector norte, rotando al sector oeste. Templado.
```

Si ejecutamos la función **fgetss** como en el ejemplo anterior, obtendremos este resultado:

```
Pronóstico Provincial de las 06:00
```

Recordemos que, al igual que **fgets**, la función **fgetss** solo lee **una línea**, y termina ante el primer salto de línea que encuentra. Por supuesto, de la misma manera que hicimos con **fgetc**, también podemos utilizarla dentro de un **bucle**, con lo cual nos permitirá recorrer un archivo HTML o PHP completo, quitando todas sus etiquetas y quedando solamente el texto:

```
<?php
$identificador = fopen ("pagina.html", "r");

while(!feof($identificador)){

      $linea = fgetss($identificador, 1024);
      echo $linea."<br>";

}

fclose($identificador);
?>
```

Eso nos mostrará solo los textos del archivo “pagina.html” completo, sin ninguna etiqueta HTML.

Esta función posee un tercer parámetro que es **opcional**, pero es el más interesante: nos permite especificar cuál etiqueta **no** será eliminada y, por lo tanto, se mantendrá dentro del código generado:

```
<?php
$abierto = fopen ("clima.html", "r");
$caracteres = fgetss ($abierto,150,"<p>");
fclose ($abierto);

print ($caracteres);
?>
```

En este caso, el resultado será:

```
<p>Pronostico Provincial de las 06:00</p>
```

Es decir, conservará, además del texto, aquellas etiquetas HTML “permitidas”.

Si queremos permitir **más de una** etiqueta HTML, debemos colocarlas una seguida de la otra, sin espacios ni comas entre ellas, dentro del tercer parámetro:

```
fgetss ($abierto,150,"<p><li>");
```

Con todas estas funciones, ya estamos en condiciones de que nuestras páginas PHP puedan **leer** contenidos de los archivos de texto.

# Cómo escribir y acumular datos en un archivo de texto

Muchas veces, necesitaremos realizar el camino inverso al de leer los datos de un archivo, es decir, precisaremos **almacenar** dentro de un archivo de texto un dato, ya sea que lo haya proporcionado el usuario, con un formulario o enlace, o bien provocado por algún código programado por nosotros (por ejemplo, un código que lleve la cuenta de las visitas a una página).

La primera consideración es que, para poder **escribir** datos (o **agregarlos**, que también requiere escribir esos datos), es necesario primero darle **permiso de escritura** al archivo de texto. Esto se realiza con cualquier programa de FTP, generalmente haciendo clic derecho sobre el archivo, y definiendo los permisos en "escritura", o en caso de contar con el comando **CHMOD** (*Change Mode* cambiar modo) poniéndolo en 777 (permiso de escritura para todos los usuarios de ese servidor).

Esto es válido únicamente para *hosting*s Linux. En caso de *hosting*s Windows, se llega a lo mismo desmarcando el atributo de "Solo lectura" dentro de las "Propiedades" del archivo (a las que también se accede haciendo *clic* derecho sobre el archivo desde dentro de nuestro programa de FTP).

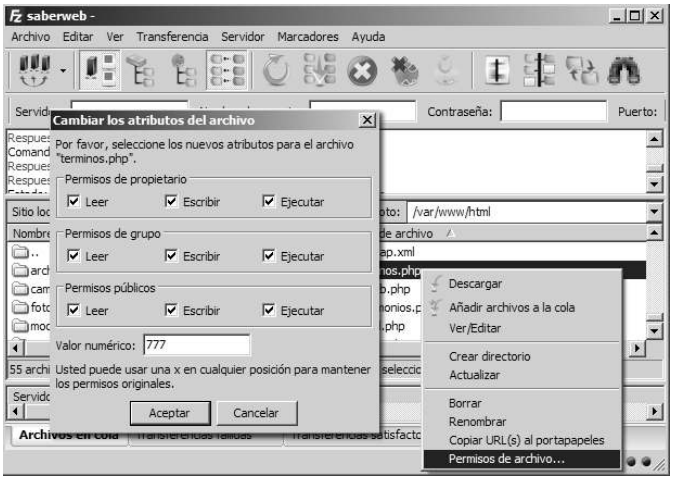


**Figura 8-6. Permisos de escritura al archivo de texto donde escribiremos datos automáticamente.**

Una vez habilitado el permiso de escritura, ya estamos en condiciones de comenzar a escribir dentro de los archivos de texto.

## Funciones fputs y fwrite

Cuando necesitemos **agregar** datos a un archivo, ya sea acumulativamente o en reemplazo de los datos anteriores, disponemos de dos funciones: **fputs** y **fwrite** (insertar dentro de un archivo o escribir archivo, respectivamente).

Veamos su sintaxis (ambas son exactamente iguales): en este ejemplo, supondremos que tenemos un formulario que solicitará escribir el nombre y el *e-mail* del usuario y, en la página siguiente (llamémosla escribe.php), se almacenarán estos datos en el archivo deposito.txt.

Atención: para probar este ejercicio, es necesario que primero se construya un formulario que envíe esas dos variables: **nombre** y ***e-mail***.

Código de escribe.php:

```
<?php
// Abrimos el archivo.
$abierto = fopen ("deposito.txt", "w");

// Preparamos los datos, desde ya que deberíamos
validarlos antes en un caso real
$nombre = $_POST["nombre"];
$email = $_POST["email"];
$texto = $nombre." - ".$email;

/* Intentamos escribirlos, validando que se haya podido
hacer */
if (fputs ($abierto,$texto)){
      print ("<p>Gracias por sus datos</p>");
} else {
      print ("<p>Hubo un error. Intente nuevamente</p>");
}

// Cerramos el archivo.
```

```
fclose ($abierto);
?>
```

Pero, ¿qué sucederá si ejecutamos nuevamente el script y luego miramos el contenido del archivo de texto? Sencillamente, que solo conservará **el último dato** que escribió en el archivo, ya que se **sobrescribió** todo el contenido anterior del archivo, borrándolo por completo justo **antes** de escribir el nuevo. Podemos probar de abrir manualmente el archivo (con el bloc de notas) y veremos el contenido, antes y después de ejecutar la página anterior.

## Acumular datos sin borrar lo anterior

Y entonces, ¿cómo hacemos para ir **acumulando** los datos de los usuarios dentro de un mismo archivo de texto? Simplemente, cambiando el **modo de apertura** del archivo, poniéndole a *(append* o agregar) en lugar de **w** en el momento de abrir el archivo con **fopen**. Veamos un ejemplo suponiendo que el formulario envía hacia esta página las mismas variables "nombre" y "email":

Código de suscripcion.php:

```
<?php
// Abrimos el archivo, pero esta vez para añadir.
$abierto = fopen ("suscriptores.txt", "a");

// Preparamos los datos
$nombre = $_POST["nombre"];
$email = $_POST["email"];
$texto = $nombre." - ".$email;

/* Intentamos añadirlos, validando que se haya podido
hacer */
if (fputs ($abierto,$texto)){
      print ("<p>Gracias por sus datos</p>");
} else {
      print ("<p>Hubo un error. Intente nuevamente</p>");
}
```

```
// Cerramos el archivo.
fclose ($abierto);
?>
```

Desde ya, revisemos que el archivo suscriptores.txt tenga el permiso de escritura (o **777**, si es numérico el sistema de permisos de nuestro programa de FTP); de lo contrario, mostrará el mensaje de error.

Con esto, ya contamos con las herramientas suficientes como para poder **almacenar** datos automáticamente dentro de archivos de texto, y para **leerlos** automáticamente desde nuestras páginas y mostrarlos. Y, además, comprendemos la mecánica de las páginas cuyos contenidos están "en otro lugar" (por ahora, en un archivo de texto, pero típicamente, en una base de datos). Esto nos ayudará mucho cuando, en los próximos capítulos, aprendamos cómo interactuar con bases de datos.

# Creando y usando funciones

# 9

## Reutilizando nuestros códigos

En todos los ejercicios que venimos realizando desde el comienzo de este libro, hemos utilizado numerosas **funciones** que ya vienen predefinidas en el lenguaje PHP: algunos ejemplos de ellas son **print**, **list**, **each**, **file**, **fread**, **fputs**, etc. Hemos "ejecutado" o llamado a esas funciones. Simplemente, las hemos usado. Pero esta facilidad de uso se basa en algo que no vemos, y es el hecho de que algún ser pensante se tomó el trabajo de **definir**, en algún lugar, cada una de esas funciones, es decir, "declaró" qué harían, paso a paso, cada vez que nosotros las ejecutáramos. Por ejemplo, imaginemos que el intérprete de PHP ejecuta este código:

```
<?php
$matriz = file($archivo);
?>
```

En alguna "otra parte" (no es magia, es simplemente un código escrito, similar a los que ya sabemos programar), un programador definió un conjunto de instrucciones a las que les puso como nombre **función file**, y determinó tres cosas:

1. Que al llamarla deberíamos proporcionarle un dato determinado entre sus paréntesis (a esos datos se les denomina **parámetros** o argumentos de las funciones). En este caso, el parámetro

necesario de la función **file** es la ruta hacia el archivo que se leerá (y es de esta manera porque así lo decidió la persona que creó la función **file**, ya que, de otro modo, si no le indicábamos cuál era el archivo que queríamos leer, mal podría la función adivinar ese dato).

2. Quien creó esta función, también decidió que la función **file** sería capaz de abrir el archivo, de leer cada renglón del archivo indicado, memorizarlo y, finalmente, cerrarlo; es decir, definió cuál sería **la tarea** que realizaría la función **file**.

3. Por último, ese programador también decidió qué **devolvería** la función **file** como resultado de su ejecución: celdas de una matriz, por lo que se requiere que la ejecución de la función **file** no la realicemos aislada, sino que le asignemos a una matriz el resultado producido por su ejecución. En consecuencia, siempre veremos una estructura similar al llamar a la función **file**: un nombre de matriz a la que le asignamos el resultado producido por **file**. Algo como: **$matriz = file("archivo.txt")**;

Todo eso lo decidió la persona (o equipo de personas) que creó la función **file**.

De este ejemplo, podemos deducir que las funciones primero se **declaran** y luego, se **ejecutan**.
Y también podemos concluir que, para declarar una función, debemos definir cuáles serán sus **parámetros** de entrada, qué **tareas** realizará la función y qué valor **devolverá** como resultado.

Estructurar nuestro código dividiéndolo en bloques especializados (funciones) nos ofrece una potencia muy grande, ya que cada función que creemos sabrá hacer algo con los datos que le proporcionamos: la función **print**, por ejemplo, "sabe" escribir esos datos dentro del código HTML de una página antes de enviarla hacia nuestro navegador; la función **fread**, "sabe" leer cierta cantidad de caracteres de un archivo de texto abierto. Es decir, las funciones son capaces de **hacer una tarea específica**, especializada.

Aunque la función se crea una única vez, es **reutilizable** todas las veces que lo necesitemos, en todo nuestro sitio y en cualquier otro proyecto en el que precisemos realizar una tarea similar. Nos ahorra trabajo a futuro.

Complementariamente a las más de 1000 funciones predefinidas que trae incorporadas PHP, nosotros también **podemos crear nuestras propias funciones**, todas las que queramos. Por ejemplo, podríamos crear una función para validar los datos ingresados en un formulario, otra función distinta para mostrar un

listado de resultados de una base de datos, otra para almacenar en un archivo de texto el número de visitas realizadas a nuestra página, y así con cada tarea específica que se nos ocurra.

# Planificando nuestros sistemas Web

## El modelo basado en la ejecución de tareas (funciones)

Se denomina **Programación Estructurada** o **Programación Procedimental** a la forma clásica de programar (en muchos lenguajes, no solo en PHP) basada en la organización de nuestro código en **funciones** propias, creadas por nosotros mismos.

Informalmente, programar creando **funciones propias** equivale a sacar carnet de programador profesional, que acaba con el código secuencial llamado irónicamente "código chorizo" (sí, eso que hemos venido haciendo hasta ahora, es decir, programar sin nuestras propias funciones, en un único hilo de líneas y líneas de código, una tras otra, lo cual dificulta bastante la reutilización de ese código en otra parte de nuestro sitio o en otros proyectos).

Un programa o **script** (o un archivo PHP), bajo este paradigma basado en funciones, quedará reducido, simplificado, a una breve **sucesión de llamadas a una o más funciones**, que recibirán los datos necesarios y los irán procesando.

```
<?php
mostrarMensajes();
enviarAviso($usuario);
guardarRespuesta($codigo);
?>
```

Por ejemplo, pensemos en un sitio de comercio electrónico (*e-commerce*). Tendremos que inventarle funciones específicas para cada una de las tareas que se realizarán en el sitio, desde el registro de un usuario nuevo, pasando por la carga de nuevos productos a su base de datos, la modificación o eliminación de datos de productos, el agregado de un pedido nuevo, la actualización de inventario *(stock)* de los productos vendidos, hasta la elección de formas de pago, de envío, etc.

El sistema completo se dividirá en **varias páginas** PHP (portada, lista de productos, registro, pedidos, etc.), y cada una de esas páginas se concentrará en la realización de una tarea específica. A esta tarea principal la organizaremos en

una o más funciones que crearemos nosotros mismos, haciendo que cada **página PHP** funcione como una **unidad procesadora de datos**:

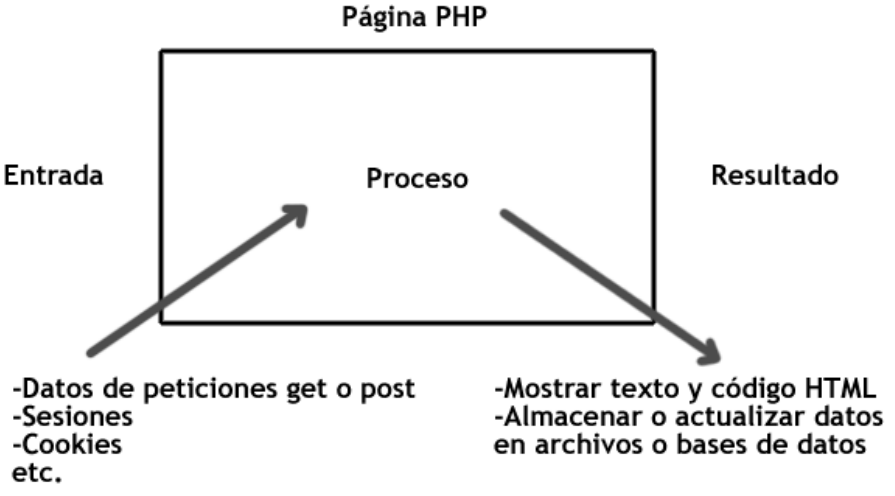


**Figura 9-1. Entrada, proceso y resultado.**

Normalmente, cada página PHP poseerá los tres momentos ilustrados en la Figura 9-1:

1. **Entrada de datos**: es el momento en el que se proporcionan **datos** a la página, ya sea mediante enlaces que envían variables, o formularios, o lectura de *cookies* o sesiones, o cualquier otra fuente de ingreso de datos. Esta etapa requiere que **validemos** la disponibilidad de los datos recibidos, para su posterior utilización.
2. **Proceso de datos**: es donde "haremos algo" con esos datos proporcionados. Por ejemplo, si nos enviaron datos de un formulario, podremos revisar su sintaxis, quizás hasta modificarlos y, finalmente, guardarlos en un lugar de almacenamiento o derivárselos a otra función para que los siga procesando. Ésta es la parte que estructuraremos mediante **funciones**.
3. **Resultados**: de ser necesario, se generará texto y código HTML para mostrar al usuario. Este resultado se puede generar directamente desde dentro de una función, o desde el punto en el que fue llamada la función.

# La función: una caja cerrada que procesa datos

Analicemos el siguiente ejemplo, tomado de una página PHP que recibe una variable proveniente de un campo de formulario, donde se espera que el usuario complete su casilla de correo para suscribirse a un boletín electrónico. Supongamos que queremos que esta página realice esta serie de tareas:

1. Verifica que hayan llegado los datos esperados y que no estén vacíos.
2. Comprueba que la casilla de correo especificada sea válida.
3. Abre un archivo de texto en modo escritura (para añadir).
4. Añade los datos y cierra el archivo de texto.
5. Devuelve un mensaje de éxito o fracaso al punto en que fue llamada la función.

Si volvemos a leer la lista anterior, veremos que es probable que tengamos que **repetir** algunas de esas **tareas** una y otra vez en distintas páginas, por ejemplo: comprobar que una casilla de correo sea válida (lo haremos en todos los formularios que tenga nuestro sitio), o abrir, escribir y cerrar un archivo de texto (lo haremos siempre que necesitemos almacenar datos). Esas tareas "recurrentes", que no son exclusividad de esta página que estamos programando, sino que podríamos llegar a necesitarlas en muchas otras páginas de nuestro sitio, o en otros futuros sitios que realicemos, son las **tareas** candidatas a convertirlas en **funciones**.

Nuestro trabajo más "creativo" como programadores, será **identificar** aquellas tareas específicas y repetitivas, que podríamos llegar a reutilizar a lo largo de nuestro proyecto, para así poder armar la **lista de las funciones** que crearemos y usaremos en cada página de nuestro sitio.

Volvamos al ejemplo anterior: habíamos identificado ya un par de tareas que sería conveniente encapsular dentro de funciones. Una era la tarea de **validar** la casilla de correo proporcionada y otra era la de **escribir** un dato dentro de un archivo de texto. Entonces, crearemos dos funciones a las que podríamos darles el nombre de **comprobarCasilla** y **escribirArchivo**.

Un comentario al paso –pero importante– acerca del nombre que demos a nuestras funciones: conviene que sean **verbos** en infinitivo, que reflejen claramente la acción

que realizará la función y, respecto a su sintaxis, sigamos las mismas reglas que para definir variables: sin espacios ni caracteres especiales, solo letras minúsculas y, comúnmente, haciendo uso de la sintaxis *camel case* que aprendimos, poniendo en mayúsculas la inicial de cada palabra que no sea la primera dentro del nombre.

Una vez decididos a crear esas dos funciones, pensemos cómo quedaría estructurado el **código de la página** que recibirá la casilla ingresada en el formulario.

Esa página podría seguir esta serie de pasos:

1. Verificar si llegó el dato esperado, y si no está vacío.
2. Llamar a la función **validarCasilla**, para ver si es correcta la casilla ingresada.
3. Si es correcta, llamar a la función **escribirArchivo** para almacenar esa casilla.
4. Si pudo almacenarse, mostrar algún mensaje al usuario.
5. Si no pudo almacenarse, mostrar algún mensaje al usuario.
6. Si no es correcta la casilla, mostrar algún mensaje al usuario.
7. Si no llegó el dato esperado o está vacío, mostrar algún mensaje al usuario.

Si pasáramos lo expresado en el párrafo anterior a código, nuestra página podría ser similar a lo siguiente:

```
<?php
/* Primer paso: verificamos que el dato de entrada esté
presente, no esté vacío, y lo traspasamos a una variable
local llamada $correo: */

if( isset($_POST["correo"]) and $_POST["correo"]<>"" ){

      $correo = $_POST["correo"];

      // Segundo paso: validar la casilla
      if( comprobarCasilla($correo)==true ){
```

```
         /* Tercer paso: si era correcta, guardarla en un
archivo */
         if( escribirArchivo($correo)==true ){

            /* Cuarto paso: si pudo almacenarse, mostrar un
mensaje */
            echo "<p>¡Gracias por suscribirse!</p>";

         } else {
      /* Quinto paso: si no pudo almacenarse, mostrar un
mensaje */

            echo "<p>Error al guardar los datos, intente
nuevamente</p>";

      } // fin pasos 3, 4 y 5

      } else {
      /* Sexto paso: si la casilla no era válida, mostrar
un mensaje */

      echo "<p>La casilla no es válida, vuelva al
formulario</p>";

      } // fin paso 2

} else { /* Séptimo paso: si no estaba presente o estaba
vacía la casilla, mostrar un mensaje */

echo "<p>No proporcionó ninguna casilla de correo, vuelva
al formulario</p>";

} // fin paso 1
?>
```

Notemos que las **validaciones** que comprueban si está presente un dato y si no está vacío, conviene que sean **previas** al momento de traspasar esos datos

(como valores de los parámetros de entrada) a una función. Esas verificaciones deberían ser el único código "suelto" que contendrán nuestras páginas PHP. Todo el resto, será mejor que lo estructuremos en **llamadas** a distintas funciones.

Observemos, también, que cada **llamada** a una función la colocamos dentro de un **condicional**, para poder saber si "funcionó", si salió bien el proceso que esperábamos que realice la función. Como vimos en el capítulo sobre condicionales, no es imprescindible colocar **==true**, ya que es la pregunta por omisión que realiza un condicional, pero preferimos dejarlo esta vez por claridad. Es decir, también podríamos haber escrito las llamadas de la siguiente manera:

```
if( comprobarCorreo($correo) ){
if( escribirArchivo($correo) ){
etc.
```

De hecho, será lo que haremos de ahora en más: cada vez que veamos algo envuelto entre las llaves de un condicional, sabremos que se está preguntando si el valor devuelto por esa expresión se evalúa como "verdadero" *(true)*.

El objetivo de colocar dentro de un condicional la llamada a la función, es ofrecer una respuesta o mensaje **diferente** al usuario en caso de éxito o de fallo de cada uno de los pasos, y además nos da la posibilidad de poder modificar el hilo de ejecución (si falló una llamada a la primera función, no tiene sentido perder tiempo en llamar a la segunda). Esto solo es posible en funciones *booleanas*, es decir, que devuelven *true* o *false* como resultado de su ejecución (pronto veremos que no todas las funciones tienen ese comportamiento).

Habiendo visto ya cómo **llamar** a estas dos funciones, ahora veamos cómo las **definiríamos**, analizando qué consideraciones es bueno que nos planteemos a la hora de declarar una función propia.

# Declarar una función

Ya sabemos ejecutar, usar una función. Pero ahora vamos a aprender cómo se **declara** una función (como se "define", paso a paso, lo que va a ser capaz de hacer con los datos que le proporcionemos). Será necesario seguir una serie de pasos:

1. Escribiremos la palabra *function*, a continuación dejaremos un espacio en blanco y, luego, escribiremos el **nombre** que le pondremos a esa función.

2. Inmediatamente al lado del nombre de la función, sin dejar espacios, colocaremos un par de **paréntesis**, que pueden quedar **vacíos** o contener los posibles **argumentos** (parámetros de entrada), es decir, los datos que recibirá la función para que los procese.
3. Una vez cerrado el par de paréntesis, colocaremos, envuelto dentro de un par de llaves, el **bloque de sentencias** que ejecutará la función al ser llamada. Las tareas que realizará con los datos que recibió.
4. Por último, definiremos la **respuesta** que devolverá la función al punto en que fue llamada.

Este sería su esquema básico:

```
function nombre () {

      tareas a ejecutar

      respuesta final;

}
```

Y este otro también podría serlo (esta vez, incluyendo parámetros dentro de los paréntesis):

```
function nombre (parámetro1, parámetro2, etc.) {

      tareas a ejecutar

      respuesta final;

}
```

## Nuestra primera función

Ya podemos pasar de la teoría a la práctica. Vamos a **declarar** una función. Dejemos atrás el ejemplo anterior y comencemos con uno nuevo.

Crearemos el archivo funciones.php, que contendrá no solo ésta, sino todas las funciones que crearemos en nuestro sitio (luego haremos un *include* de este archivo en cada página donde precisemos utilizar alguna función).

Archivo funciones.php:

```
<?php
function listarMatriz($matriz){

      $codigo = "<ul>";

      foreach ($matriz as $valor){

         $codigo = $codigo."<li>".$valor."</li>";

      }

      $codigo = $codigo."</ul>";

      return $codigo;

}
?>
```

Esta función así declarada nos permitirá imprimir, dentro del código HTML de una página, los contenidos de cualquier matriz que le pasemos como **argumento** (parámetro) en el momento de "llamar" a la función.

"Llamar" a la función significa ejecutarla, hacerla funcionar con datos reales.

Deben pasársele **la misma cantidad** de argumentos o parámetros (son sinónimos) que los especificados en la declaración de la función entre sus paréntesis, y en el mismo **orden** (en este ejemplo, hemos decidido que se necesitaba un solo parámetro, el nombre de la matriz que se imprimirá).

De esta manera, se **llamaría** a esta función desde otro archivo, llamadas.php:

```
<?php
include("funciones.php");
```

```
// Primero cargamos con datos un par de matrices:
$propiedades = array("casas", "departamentos", "campos",
"terrenos", "quintas");

$vehiculos = array("autos", "motos", "camiones",
"cuatriciclos", "bicicletas");

print("<h2>Lista de Propiedades:</h2>");
// Ahora llamamos a la función para que las muestre:

echo listarMatriz($propiedades);
/* Esto llama a la función, y el echo escribe lo que ésta
devolvió */

print("<h2>Lista de Vehículos:</h2>");

$resultado = listarMatriz($vehiculos);
/* Esto llama a la función otra vez, pero ahora
almacenamos en una variable intermedia lo que devolvió, y
luego la mostramos con echo: */
echo $resultado;
?>
```

Como ambos bloques del código anterior (la **declaración** y la **llamada**) deben estar disponibles y ejecutarse (respectivamente) en el mismo archivo PHP que se está procesando en el servidor, y como es común que necesitemos que una misma función esté disponible en distintas páginas de nuestro sitio, normalmente haremos uso de un **archivo externo**, que contendrá todas las **declaraciones** de funciones de nuestro sitio, archivo que incluiremos luego dentro de las páginas de nuestro sitio mediante una orden **include**; en este caso, cumple ese rol de depósito de funciones el archivo funciones.php.

Por lo tanto, el código completo de este ejemplo quedaría estructurado de esta manera: por un lado, el archivo funciones.php con la declaración de ésta y de cualquier otra función; por el otro, el archivo llamadas.php.

En nuestros sitios, comúnmente tendremos más de un archivo que haga un **include** de nuestro archivo de funciones, para tenerlas disponibles y usarlas:

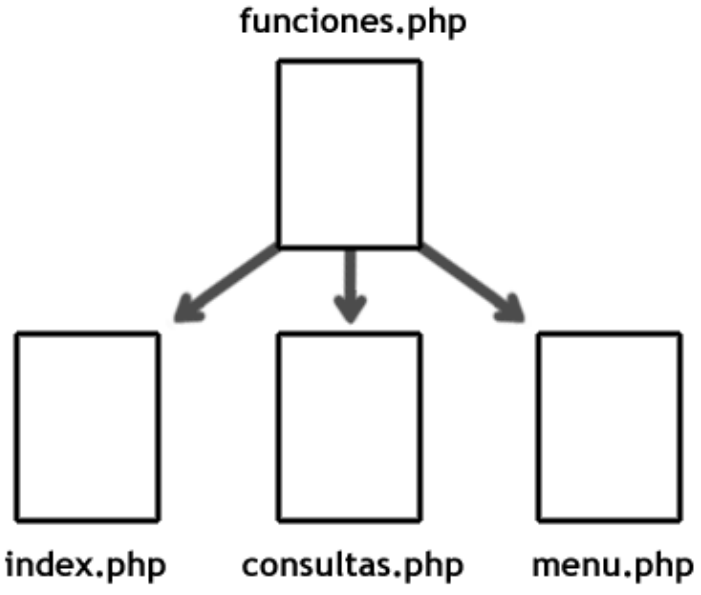


**Figura 9-2. Archivo externo con declaraciones de funciones, que será incluido desde otras páginas.**

En muchos códigos, veremos que los programadores se refieren a este archivo externo que contiene declaraciones de funciones como "biblioteca" de funciones e, incluso, mal traducido, "librería" de funciones (por el original en inglés: *library*).

## Parámetros de entrada

Si volvemos a analizar nuestra función del ejemplo anterior, veremos que **recibe un valor** que se le pasa en el momento en que llamamos a la función: en este ejemplo, recibe primero la matriz "propiedades" y, luego, en la siguiente llamada, recibe la matriz "vehículos".

Como en la declaración de la función, al lado de su nombre, entre los paréntesis, escribimos una variable a la que decidimos llamar **$matriz**, esta variable es la que va a contener, **dentro de la función,** la matriz que sea proporcionada como argumento desde fuera de la función.

Dentro de la función no llamamos a la matriz que fue enviada desde el "mundo exterior" por su verdadero nombre, ya que siempre será un nombre distinto el que contendrá los datos allí fuera (así como en este caso eran "propiedades" y "vehículos", le podríamos proporcionar a la función muchas otras matrices, siempre de distintos nombres). Por eso, le daremos un "alias" o **sobrenombre** de uso interno que, en este caso, es $matriz, y que será el nombre de variable que pondremos dentro de los paréntesis y el que utilizaremos para referirnos a «eso que llegó» desde fuera de la función, y lo usaremos dentro de ella para manipular ese dato.

Veamos otro ejemplo para comprenderlo mejor:

```
<?php
function cuadrado($numero){
      $total = $numero * $numero;
      return $total;
}
$cantidad = 6;
$resultado = cuadrado($cantidad);
// Ya ejecutamos la función, $resultado almacena un 36

print("<p>".$cantidad." al cuadrado es:
".$resultado."</p>");
// Mostramos el resultado
?>
```

En este caso, le hemos pasado a la función, al momento de llamarla, el contenido de la variable $cantidad. Pero, dentro de la función, a ese parámetro lo llamamos $numero, y usamos esta variable para todos los cálculos internos de la función.

A esto se lo denomina pasarle argumentos "por valor" a una función: la función no trabaja sobre la variable original externa, no la modifica, sino que crea **una copia** del valor, una copia de la variable original con otro nombre: un "alias" (en este ejemplo, $numero es el nombre de "uso interno" de la función).

Notemos algo importante sobre la dinámica de una función:

1. Le damos **nombre** a la función.

2. Definimos sus “datos de entrada”, su materia prima, es decir, sus **parámetros**.
3. Dentro de la función, realizamos **operaciones** con esos datos.
4. Finalmente, la función termina su ejecución y **devuelve su producto terminado** mediante la palabra “**return**”.

Veámoslo una vez más analizando el código anterior hasta comprenderlo por completo: si bien la variable que le pasamos como parámetro a la función al momento de llamarla se denominaba **$cantidad**:

```
$resultado = cuadrado($cantidad);
```

podría llamarse de cualquier otra forma sin que eso modifique nada, por ejemplo:

```
$dato = 1234;

$resultado = cuadrado($dato);
```

e incluso podríamos pasarle un “literal”, una constante, es decir, un número directamente:

```
$resultado = cuadrado(7);
```

y funcionaría perfectamente.

Al número que le hemos pasado entre los paréntesis, en la **declaración** de la función, habíamos decidido denominarlo “internamente” **$numero**:

```
function cuadrado($numero)
```

Por esta razón, en la “receta” paso a paso dentro de las llaves de la función, nos referimos a ese dato que llegó de afuera como **$numero**, y no con el nombre que tenía fuera de la función, ya que podría ser cualquiera, nunca sabremos qué nombre tendrá el dato que le proporcionaremos a la función, solo es importante saber **qué hacer** con ese dato que nos pasaron al llamarla.

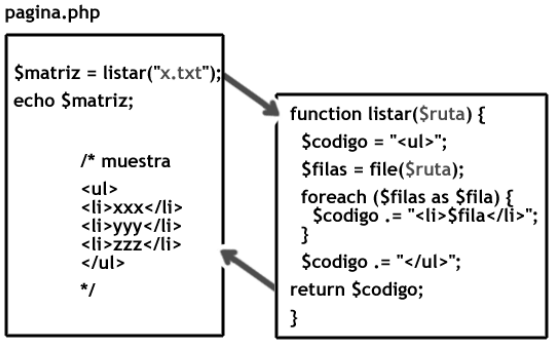


**Figura 9-3. Función con un único parámetro de entrada.**

### Funciones con varios parámetros

Hasta ahora, hemos declarado funciones que utilizan un único parámetro. Pero cuando la función necesite manipular más datos, podemos especificar más de una variable entre los paréntesis en el momento de declarar la función, y luego proporcionar la misma cantidad de datos y en el mismo orden al momento de llamarla.

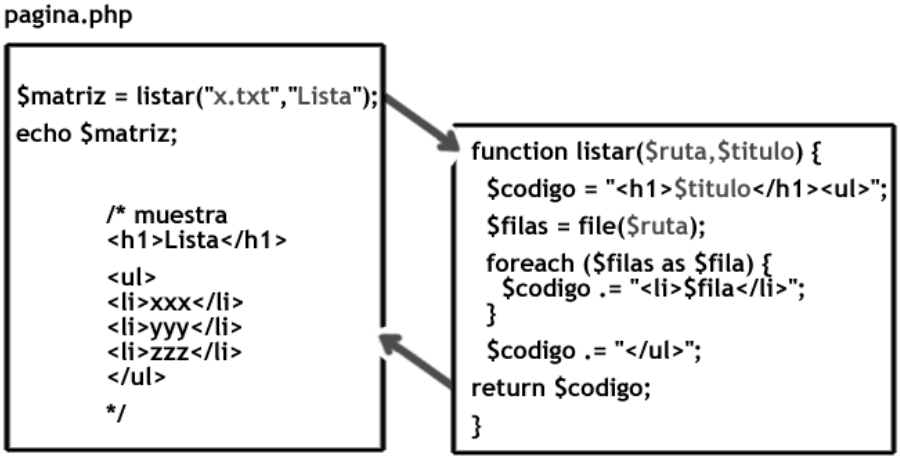


**Figura 9-4. Función con más de un parámetro de entrada.**

Veamos otro ejemplo:

```
<?php
function aumento($precio, $porcentaje){
      $total = $precio + ($precio * $porcentaje / 100);
      return $total;
}

$resultado = aumento(12.99, 5);
/* Proporcionamos la misma cantidad de datos que de
parámetros tenga la función, en este caso, dos, separados
por comas */
print("<p>Nuevo precio: ".$resultado."</p>");
?>
```

Por supuesto, pueden ser más de dos los parámetros de entrada, en realidad, pueden ser todos los que precisemos:

```
function matricular($alumno, $curso, $turno, $tutor,
$beca){
      // etc.
}
```

### Funciones sin parámetros

Complementariamente, también puede ser necesario crear una función que no precise **ningún dato de entrada** para hacer su tarea. Por ejemplo, consideremos una función que se ocupe de contar las visitas a una página Web:

```
<?php
function contarVisita(){
      /* Aquí abrirá el archivo, leerá la cifra de visitas,
le sumará 1, escribirá el nuevo valor, y cerrará el
archivo */
}
?>
```

Al llamar a una función que no necesita parámetros, debemos colocar, de todos modos, los paréntesis al lado del nombre de la función, pero **vacíos**, sin ningún dato entre medio (ni siquiera un espacio en blanco):

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

Eso solo será suficiente para que la visita sea contabilizada dentro del archivo de texto. Otro ejemplo de una función sin parámetros sería el siguiente:

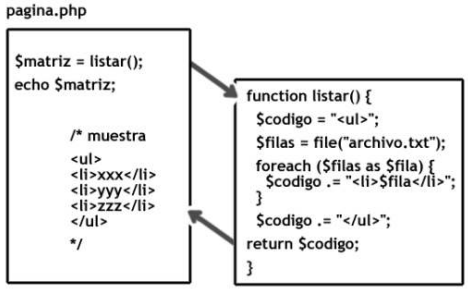


**Figura 9-5. Función sin ningún parámetro de entrada.**

Ya sea que definamos ninguno, uno o más parámetros de entrada, lo importante es que debemos mantener la **coherencia** entre la cantidad de parámetros definidos al declarar la función, y la cantidad de datos proporcionados entre los paréntesis al llamar a la función y, además de la cantidad, respetar el **orden** en el que han sido definidos, ya que la función, al ejecutarse, tomará el dato que llegue en primer lugar, y lo colocará como valor de la primera variable ubicada entre los paréntesis al declarar la función, el segundo dato lo colocará como valor de la segunda variable, y así sucesivamente.

## Devolviendo datos a la salida con return

Sin que importe la cantidad de parámetros de entrada que hayamos definido, toda función debería devolver algún "producto", uno y solo uno, en el preciso lugar en donde fue llamada.

Para eso, se utiliza la palabra **return** colocada al final de la función, justo antes del cierre de sus llaves. A continuación de la palabra **return**, debemos colocar el valor que devolverá la función como resultado de su ejecución. La función devolverá una sola cosa, un solo dato (del tipo que sea, una variable, matriz, constante, etc.), pero uno. Si dentro del código de la función hubiera más de un **return** (porque se lo colocó dentro de un condicional, por ejemplo), solo uno de los **return** se ejecutará, ya que en cuanto la función ejecuta la orden **return**, se "sale" inmediatamente de la ejecución de la misma y se sigue ejecutando el código exactamente a continuación del punto en que fue llamada la función.

Unas veces ese producto será **texto** o código HTML para mostrar en una página, como por ejemplo aquí:

```
<?php
function aumento($precio, $porcentaje){
      $total = $precio + ($precio * $porcentaje / 100);
      return $total;
}
```

En este caso, esta función devolverá una **cifra**: la que surja de la operación que realiza con los datos de entrada, y que estaba almacenada en la variable $total. Otras veces, esa variable contendrá un código HTML con textos, como en el caso en el que se muestre un listado de productos, un elemento *select* HTML con sus *options*, etc.

Pero, otras veces, no será necesario mostrar nada como resultado de la ejecución de la función (pensemos en el contador de visitas que pusimos como

ejemplo unos párrafos atrás). En ese caso, conviene que la función devuelva un valor de "éxito" o uno de "fracaso", según haya podido terminar sus tareas normalmente, o no. Para devolver un valor de éxito, usaremos la constante **true**, y para un valor de fracaso, la constante **false**. Esto nos permitirá evaluar con un condicional si la función pudo hacer su trabajo o no.

Veamos cómo, sobre la base de un ejemplo anterior, permitimos que se realice esta validación, haciendo que nuestra función sea *booleana*, es decir, que su valor de retorno o devolución sea true o false:

```
function escribirArchivo($casilla){
      /* Omitimos todos los pasos previos, y nos
concentramos en el final. Una función booleana debería
terminar con un condicional: */

      if( fclose($abierto) ){

          return true;

      } else {

          return false;

      }
}
```

Al tener como únicos posibles valores de retorno **true** o **false**, podemos validar cuál de los dos valores devolvió al momento de llamarla:

```
if( escribirArchivo($correo)==true ){

      echo "<p>¡Gracias por suscribirse!</p>";

} else {

   echo "<p>Error al guardar los datos, intente
nuevamente</p>";
}
```

Según nuestras necesidades, definiremos funciones que devuelvan textos, o números, o matrices, o constantes, o que sean booleanas; seguiremos practicando esto a lo largo del resto del libro.

También, puede darse el caso de que necesitemos devolver una **matriz** como único valor de **return**, algo muy útil para poder devolver varios datos a la vez mediante una sola función (este es el comportamiento de varias funciones incorporadas que ya conocemos, como por ejemplo **file** y **array**).

Veamos un ejemplo:

```
function verificar($nombre,$apellido,$edad,$correo){
/* Omitimos todos los pasos previos, y nos concentramos en
el final que devuelve una matriz: supongamos que hemos
verificado el primer dato ($nombre) y hemos colocado el
resultado (true o false) dentro de $v1, luego hemos
verificado el $apellido y su resultado quedó en $v2 y así:
*/

      $resultado = array($v1,$v2,$v3,$v4);

      return $resultado;

}

/* Donde llamemos a la función, deberemos recibir su
producto: */

$matriz = verificar("Juan","Pérez","34","j@perez.com");

/* En este punto, $matriz será una matriz que contendrá
exactamente lo que contenía $resultado dentro de la
función: 4 celdas cuyos valores serán true o false. */
```

### Las ventajas de evitar echo y print desde dentro de las funciones

Cuando la función tiene como meta producir un texto o código HTML, muchos programadores caen en la tentación de que la función vaya haciendo **echo** o **print** desde dentro de ella.

Cuando se termina de ejecutar el código que tiene entre sus llaves la función, nos damos cuenta de que ya no es posible hacer nada más con la

información que generó la función, porque **ya fue escrita** dentro del código HTML de la página, ya es irreversible lo que produjo. Veamos un ejemplo de este enfoque no muy práctico, aunque difundido:

```
<?php
function cuadrado($numero){
      $total = $numero * $numero;
      echo $total;
}

$cantidad = 6;
cuadrado($cantidad);
/* Allí, donde llamamos a la función, ya se mostró su
Resultado */
?>
```

Esto nos obliga a intercalar llamadas a la función en medio de órdenes **echo** o **print**, lo cual da por supuesto que no va a haber errores a la hora de ejecutar esa función, y elimina la posibilidad de validar los datos.

Por ejemplo:

```
$cantidad = 6;
print("<p>".$cantidad." al cuadrado es:
".cuadrado($cantidad)."</p>");
// Mostramos el resultado
?>
```

En lugar de eso, y tal como lo hemos mostrado en todos los ejemplos anteriores, es mucho mejor que la función solamente vaya **acumulando** códigos y texto dentro de una variable, y lo devuelva mediante **return**. Por ejemplo:

```
<?php
function cuadrado($numero){
      $total = $numero * $numero;
      return $total;
      }
```

Siempre es mejor usar un **return** como final de la función, y en el lugar desde donde llamamos a la función, podemos validar qué devolvió la función y **decidir** si mostramos el resultado en el momento, o si no lo mostramos, o si le tenemos que aplicar algún otro proceso a ese dato que entregó la función y pasarlo como parámetro de entrada a otra función que tome lo que fabricó la primera función para seguir modificándolo. Parece un detalle menor, pero es una técnica clave.

Es mucho mejor **ir acumulando textos** dentro de la función (guardados en una **variable**), sin que la función "emita" mensajes hasta que termine su ejecución.

Veamos otro ejemplo, esta vez con abundante texto y código HTML; veamos cómo se va acumulando (concatenando) dentro de la variable $codigo:

```
<?php
function armarTabla($filas){

      $codigo = '<table>';

      for ( $i=1; $i<=$filas; $i=$i+1){

      /* Llamamos a una función que se especialice en
calcular el módulo y devuelva true o false */
          if ( modulo($i,2)==false ){
             $codigo = $codigo.'<tr><td class="par">Esta
             es la fila número'.$i.'</td></tr>';
          } else {
             $codigo = $codigo.'<tr><td class="impar">Esta
             es la fila número'.$i.'</td></tr>';
          }
      }
$codigo = $codigo.'</table>';

return $codigo; /* Le decimos qué datos devolverá al lugar
donde fue llamada, o sea la variable $codigo */
}
```

Fuera de la función, en el lugar donde llamemos a esta función, vamos a acumular en una **variable** lo que devuelva la función con su **return**:

```
$tabla = armarTabla($_POST["numero"]);
```

Cuando se ejecute esta línea de código, la variable **$tabla** contendrá lo mismo que contenía **$codigo** adentro de la función (es decir, contiene todo el código HTML de la tabla entera).

Ahora, tenemos la libertad de decidir qué hacer: podemos hacer un **print** o **echo** de $tabla si nos conviene, pero además tenemos la posibilidad de pasarle a otra función todo lo que fabricó la función **armarTabla()**, dándonos la oportunidad de armar una "cadena de montaje" como en una fábrica, en la que cada función se especializa en hacer una tarea breve y concreta, lo que ayuda muchísimo a la modularidad del código (a que sea reutilizable, modular –estructurado en módulos, bloques–).

Por ejemplo:

```
$decorado = decorarTabla($tabla);
```

Le pasamos a una supuesta nueva función denominada **decorarTabla** todo lo que produjo la función anterior, gracias a que lo teníamos almacenado en la variable **$tabla** y a que no hicimos ningún **echo** ni **print** desde dentro de la función.

Solo nos resta ver un último tema relacionado con las funciones: el alcance de las variables dentro y fuera de las funciones.

## Alcance de las variables en las funciones: locales y globales

Algunas veces, nos sucederá que no es solo el dato que fue proporcionado como **parámetro** a la función (el que "entró") el que necesitamos usar dentro de ella, sino que a veces necesitaremos acceder a **datos externos la función**, ya sea para **leerlos** (tal como si hubieran sido proporcionados como parámetros de entrada) o para que la función **modifique** el valor de esos datos externos.

A las variables externas a una función, las denominaremos variables **globales** y, a las que declaramos y usamos dentro de una función, **locales**.

Veamos este ejemplo, que lee una variable **global** cuyo nombre es $aumento, y la modifica:

```
<?php

function inflacion($precio,$suba){
      $precio = $precio * $suba;
```

```
      global $aumento; /* Estamos anunciando que nos
referiremos a una variable externa a la función */

      $aumento = 1.50; /* Esta línea modifica la variable
externa, cada vez que sea llamada esta función */
      return $precio;
}

$producto1 = 100;
$aumento = 1.10; /* Ésta es la variable externa, hasta
aquí vale 1.10 ya que aún no llamamos a la función */

print("<p>El producto 1 vale: ".$producto1." antes del
aumento, y la variable \$aumento todavía vale:
".$aumento."</p>");

$resultado = inflacion($producto1,$aumento);
print("<p>Ahora el producto 1 vale: ".$resultado."</p>");

print("<p>La variable \$aumento después de llamar a la
función vale: ");

print($aumento."</p>");
?>
```

Por omisión, las variables que declaramos dentro de las funciones son locales (es decir, se pueden usar solo dentro de la función) pero, a veces, necesitamos acceder a variables que ya fueron **declaradas antes** de ejecutar la función y modificarles su valor; estas variables **"externas" a la función** se denominan **globales**. Se puede acceder a ellas desde adentro de una función anteponiéndoles la palabra **global**, como lo hemos hecho con la variable **$aumento** en la función **inflación** que recién creamos.

Notemos que la modificación a la variable global la debemos hacer **antes del return**, ya que el **return** "sale" de la función y, por ello, **deja de ejecutarse lo que viene después**, y quedaría sin modificarse el valor de la variable global.

También podemos acceder a una variable global desde dentro de una función mediante la matriz **$GLOBALS["nombre-de-la-variable"]**, ya que esta

matriz almacena **todas las variables globales** que se hayan declarado en ese archivo PHP antes de ejecutar la función.

Un ejemplo:

```
<?php

function multiplicar($numero,$otro){
      $numero = $numero * $otro;
      $GLOBALS["cifraB"] = 9;
      /* Estamos cambiando una variable de fuera de la
función */
      return $numero;
}

$cifraA = 100;
$cifraB = 5;
print("<p>El primer número vale: ".$cifraA."</p>");
// vale 100
print("<p>El segundo número (o sea, \$cifraB) vale:
".$cifraB."</p>"); // vale 5

print("<p>La multiplicación da como resultado: ");
$resultado = multiplicar($cifraA,$cifraB);
print($resultado."</p>"); // muestra 500

print("<p>Y ahora, \$cifraB luego de ejecutar la función
vale: ".$cifraB."</p>");// ahora vale 9
?>
```

Es bueno que sepamos que el uso de global y *GLOBALS* solo para **leer** una variable global es desaconsejable, ya que sería mucho mejor **pasarle** ese dato a la función como **parámetro** (entre los paréntesis de entrada). Puesto que la idea de una función es que debería desempeñarse como una "caja" cerrada, en la que los datos entran por sus «puertas» **(parámetros)** y salen por su **return**.

Hemos llegado al fin de lo que necesitamos saber para crear nuestras propias funciones.

De ahora en más, todos nuestros códigos podrán ser organizados mediante **funciones propias**, creadas por nosotros para que realicen todas las tareas específicas que necesitemos y listas para ser reutilizadas a lo largo de todos nuestros futuros proyectos.

# Funciones incorporadas más usadas

# 10

## Manejo de caracteres, fechas y envío de correos

Ahora que comprendemos las ventajas de estructurar nuestro código en funciones reutilizables, será muy conveniente dedicar un tiempo a conocer las principales funciones que posee el lenguaje PHP, ya que nos ahorrarán muchísimo tiempo, al evitarnos la tarea de encontrar soluciones a problemas muy comunes. El lenguaje PHP posee **más de 1000 funciones predefinidas**, es decir, listas para que las ejecutemos, porque ya fueron declaradas por los programadores que crearon este lenguaje. Estas funciones se agrupan en categorías. A continuación, veremos ejemplos de algunas de las categorías "clave" de funciones que PHP posee: funciones para manejo de caracteres, funciones que permiten trabajar con fechas y horarios, y otras para envío de correos electrónicos.

En el **manual** en línea del sitio Web oficial de PHP, podemos consultar la ayuda sobre **cualquier función** que nos interese, de una manera muy sencilla: agregando al nombre del dominio una **barra** y el **nombre de la función**. Por ejemplo, si queremos saber algo acerca de la función **fgets**, escribiremos:

```
http://www.php.net/fgets
```

Recomendamos ir a la página de cada función luego de haberla estudiado y probado sus ejemplos básicos, para poder profundizar en el conocimiento de lo que **hace** esa función, cuáles son sus **parámetros** de entrada, y cuál es el tipo de dato que **devuelve** con su return, además de ver otros códigos creados por los

usuarios de PHP que utilizaron esa función junto con sus experiencias relacionadas con ella (encontrarán esto último al pie de cada página, en el listado de comentarios de usuarios).

# Funciones de manejo de caracteres

La mayoría de nuestros códigos operan sobre datos que pueden ser de muy distintos tipos, pero sin duda uno de los tipos de datos más usados son las **cadenas de caracteres** (los textos).

Por lo tanto, será sumamente necesario poder manipular con soltura estos datos alfanuméricos, mediante la serie de funciones que PHP trae predefinidas dentro de esta categoría, funciones que –por ejemplo– nos pueden servir para contar caracteres de un texto, buscar y reemplazar partes de ese texto, segmentar y unir partes de textos para almacenarlos, y muchas otras tareas necesarias en la validación y manipulación de cadenas de caracteres.

Veamos, a continuación, las principales funciones de manejo de caracteres que disponemos en PHP.

## Limpiar espacios en blanco

Anteriormente, cuando utilizamos la función **file** para leer datos de un archivo de texto, un detalle que quizás pasó desapercibido es el de los **saltos de línea ("\n")** que había al final de cada renglón y que eran tomados como "caracteres" adicionales. Por lo tanto, si necesitábamos comparar una supuesta contraseña escrita por el usuario contra uno de los renglones del archivo de texto leído mediante file, la celda de la matriz (que guardaba un renglón entero y que incluía el salto de línea al final) nos decía en todos los casos que no coincidía la contraseña, por más que el usuario la hubiera escrito bien. Esto es lógico, ya que no es lo mismo "pepe" que "pepe**\n**".

A los espacios en blanco, los saltos de línea y los tabuladores, se los denomina **caracteres de control**, y si bien se trata de caracteres invisibles, son iguales a cualquier otro, y su presencia es normal cuando leemos datos almacenados en el interior de un archivo de texto, o provenientes de un campo de formulario de múltiples líneas (*textarea*). Por eso, son muy utilizadas las funciones que **eliminan** estos caracteres de control dentro de una cadena de caracteres cualquiera, para poder comparar su valor con total precisión.

### Función trim

La función **trim** (recortar, limpiar) elimina cualquier carácter de control, tanto al **principio** como al **final** de una cadena de caracteres. Suprime toda esta lista de caracteres:

| Carácter tal como es almacenado (sin las comillas) | Descripción |
|---|---|
| " " | Espacio en blanco |
| "\t" | Tabulador |
| "\n" | Nueva línea |
| "\r" | Retorno de carro |
| "\0" | El byte NULL |
| "\x0B" | Tabulador vertical |

**Tabla 10-1. Caracteres que elimina la función trim.**

Un ejemplo:

```
<?php
$cadena = "\tabcd \n";
$limpio = trim($cadena);
print ($limpio);
/* Escribe "abcd", sin tabulador ni espacios ni salto de
línea. */
?>
```

### Función ltrim

La función **ltrim** *(letf trim* o limpiar a la izquierda) elimina los caracteres de control (espacios, tabuladores, nueva línea, etc.) que hubiera **al principio (a la izquierda)** de la cadena de caracteres, pero no los del final.

### Función rtrim o chop

La función **chop** (o su alias **rtrim** –**right trim**, limpiar a la derecha–) elimina solo los caracteres de control que hubiera **al final** (a la derecha) del texto.

Atención: en ninguno de los dos casos, se eliminan caracteres de control ubicados "en medio" de la cadena de texto, solo al principio y al final.

## Comparar evitando errores: strtolower y strtoupper

Volviendo a uno de los usos más comunes de estas funciones de limpieza de textos, que es el de **comparar** lo ingresado por un usuario contra un texto almacenado para **validarlo**, si un usuario escribiera su contraseña o nombre de usuario teniendo pulsada la tecla de **Bloqueo de Mayúsculas**, no se le permitiría su acceso (no es lo mismo "A" que "a").

Pero como esta situación es previsible que suceda frecuentemente, podemos evitarla convirtiendo siempre a mayúsculas (o a minúsculas) todo lo escrito por el usuario, y compararlo con lo que traigamos de un archivo de texto o base de datos, también convertido a mayúsculas o a minúsculas, respectivamente.

Para eso, nos sirven las funciones **strtolower** *(string to lower case,* o texto a minúsculas) que lleva el texto a minúsculas, y **strtoupper** *(string to upper case,* o "texto a mayúsculas"), que convierte todos sus caracteres en mayúsculas.

Veamos un ejemplo:

```
<?php
$claves = file("claves.txt");
$cuantos = count($claves);

$encontrado = "no";

for ($i=0; $i<$cuantos; $i++){
      if ( strtolower(trim($claves[$i])) == strtolower($_
POST["password"]) ){
          $encontrado = "sí";
      }
}

if ($encontrado == "no"){
      include ("error.php");
} else {
      include ("secreto.php");
}
?>
```

## Contar la cantidad de letras de un texto

Podemos contar la **cantidad de caracteres** de una cadena de caracteres con la función *strlen (string lenght* o longitud de la cadena). El único argumento que recibe es una variable que contenga el texto. Devuelve un **número** entero, que es la cantidad de caracteres de ese texto.

Un uso interesante de esta función es evitar aceptar textos demasiado largos, que superen un máximo permitido, en comentarios, mensajes de un foro, etc. De la misma forma, podemos verificar la longitud **mínima** y **máxima** de nombres de usuarios, contraseñas, casillas de correo electrónico, etc.

Por ejemplo, supongamos que queremos validar un comentario de varias líneas enviado por los usuarios; podríamos crear una función ***booleana*** que recibirá la cadena de texto ingresada y la longitud máxima permitida como parámetros:

```
<?php
function verificarLongitud($cadena,$maximo){

      $cuantas_letras = strlen($cadena);

      if ($cuantas_letras < $maximo){
          return true;
      } else {
          return false;
      }
}
?>
```

Luego, simplemente validaremos con un condicional si la cantidad supera o no el límite que decidamos pasar como parámetro en el momento de llamar a esta función:

```
if (verificarLongitud($_POST["comentario"],200)){
      echo "Correcto";
} else {
      echo "Supera el máximo de 200 caracteres...";
}
```

## Obtener "partes" de una cadena de caracteres

Muchas veces, necesitaremos recortar solamente una parte, un fragmento de una cadena de caracteres, para compararla con otra cosa, validar algo, colocarla dentro de una variable, etc. Otras veces, querremos verificar que una pequeña parte de ese texto coincida con algo que hemos escrito; por ejemplo, que la extensión de un archivo sea "xxxxx**.jpg**".

Para ello, nos podemos valer de funciones para el manejo de cadenas de caracteres, tales como **substr**, **strpos** y **strstr**. Veamos para qué sirve cada una de ellas.

### Función substr

La función **substr** (*substring* o subcadena) se utiliza para obtener **parte de una cadena** de texto. Se deben especificar tres argumentos:

1. cuál es **la cadena**,
2. a partir de **cuál carácter** queremos obtener,
3. y **cuántos** caracteres queremos incluir. Este último argumento es optativo y, si lo omitimos, nos devuelve hasta el final de la cadena.

A continuación, algunos ejemplos para comprender mejor cómo funciona:

```
<?php
$texto = "Este es un curso de PHP";
$inicio = 5;
// Los caracteres se numeran desde cero
$parte = substr($texto, $inicio);
echo $parte;
/* Mostrará: "es un curso de PHP", es decir, desde el
caracter 5 hasta el final del string. */
?>
```

Los caracteres que queremos leer se cuentan a partir de **cero**, por esta razón, la "e" que ocupa el sexto lugar en la cadena (el [5], recordemos que el primero es el [0]) se toma como inicio de la parte deseada.

En cambio, si especificamos el **tercer** argumento, podremos recortar una parte desde un inicio hasta una **cantidad específica** de caracteres:

```
<?php
```

```
$texto = "Esta es una parte del string";
$inicio = 12;
$cuantos = 5;
$parte = substr($texto, $inicio, $cuantos);
echo $parte;
// Mostrará: "parte".
?>
```

Si el punto de inicio es **negativo**, se cuenta de atrás para adelante hasta llegar a ese punto inicial. Luego, se seleccionarán los caracteres que siguen hacia el final, normalmente:

```
<?php
$cadena = "www.sitio.com.ar";
$sitio = substr($cadena,-7,7);
print($sitio);
// Mostrará ".com.ar"
?>
```

Si el punto de inicio es **positivo**, pero el largo es **negativo**, esos dos caracteres se considerarán el punto de inicio y el punto de finalización (se permite la selección de un fragmento del que no conocemos la longitud, pero sí la cantidad de caracteres previos y posteriores); por ejemplo, podemos extraer el nombre del dominio a partir de un dato ingresado de la siguiente manera:

```
<?php
$direccion = "www.pagina.com.ar";
$sitio = substr($direccion,4,-7);
print($sitio);
// Imprimirá "pagina"
?>
```

### Función strpos

Conoceremos ahora la función **strpos** *(string position* o posición de una cadena). Esta función sirve para saber **en qué posición**, dentro de una cadena de caracteres, aparece por primera vez un carácter o cadena de caracteres buscados.

Sus argumentos son: la **cadena** de texto dentro de la cual buscar, y el **carácter o caracteres** de los que se quiere conocer la posición.

Ejemplo:

```
<?php
$cadena = "América Latina unida";
$abuscar = "unida";

$posicion = strpos($cadena,$abuscar);
print($posicion);
// Imprimirá 15 -comienza desde cero-.
?>
```

Pensemos ahora en los usos que le podemos dar a la función **strpos**. El truco para usarla en búsquedas y validaciones consiste en preguntar con un **if** si esa posición es distinta de "*false*, cero o vacía", lo que significará que el texto sí **fue encontrado**.

Debido a esa relativa complejidad en el dato que puede llegar a devolver esta función, necesitaremos utilizar un operador de comparación que no vimos antes: el de **identidad**, que consiste de un triple signo igual **===**, y que evalúa no solo que los valores comparados sean iguales, sino que, además, únicamente devuelve verdadero si, además de ser iguales, ambos valores comparados son del **mismo tipo de dato** (cadena de caracteres, entero, *booleano*, etc.). De esta manera, podremos diferenciar la situación de **no** haber encontrado "algo", que no es lo mismo que si lo hubiéramos encontrado en la posición "cero" (esta situación se comprenderá al analizar el ejemplo que daremos más adelante).

Además de los dos argumentos básicos que ya mencionamos (la **cadena** donde buscar, y **el carácter o caracteres** de los que se quiere saber la posición o detectar su presencia), esta función tiene un tercer argumento –opcional– que es **a partir de cuál** carácter de la cadena se empezará la búsqueda. Si no lo especificamos, devolverá el resultado contando desde la primera posición de la cadena.

Veamos otro ejemplo orientado a detectar si está presente, o no, alguna cadena de caracteres:

```
<?php
$cadena = "América Latina unida";
$texto_buscado = "A";
```

```
$posicion = strpos($cadena,$texto_buscado);

if( $posicion===false ){
      echo "No se encontró";
} else {
      echo "Sí se encontró, en la posición: $posicion";
}
?>
```

En este ejemplo, buscamos la letra **A** que corresponde a la posición **0** dentro de la cadena de caracteres (recordemos que los caracteres dentro de una cadena se numeran tal como las celdas de una matriz, a partir de cero). Si hubiéramos utilizado el operador de comparación normal (el doble signo igual ==), esta condición hubiera devuelto que "no encontró" la letra **A**, ya que el valor de **0** que retornaría *strpos* sería considerado como un sinónimo de **false**. En cambio, al utilizar el operador de identidad: **===**, se compara el **0** almacenado en $posicion contra el **false**, que son dos datos de distinto tipo, por esta razón, la comparación no es verdadera y se ejecuta el código del **else**.

Esta función es muy práctica para casos en los que debemos verificar **que exista** un carácter en especial dentro de una cadena de texto –como, por ejemplo, una arroba dentro de una dirección de correo electrónico– o, por el contrario, para detectar un carácter **prohibido** dentro de un texto.

Una posibilidad que se nos abre es la de evaluar una **serie** de caracteres permitidos o prohibidos, carácter por carácter, dentro de un bucle:

```
<?php
function permitirLetras($cadena){
      $letras =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
      $longitud = strlen($cadena);

      for( $i=0; $i<$longitud; $i++){

       if(strpos($letras,substr($cadena,$i,1))=== false ){
      /* $cadena no es válido, contiene algún carácter
prohibido */
```

```
            return false;
          }
        }
        /* Si terminó el bucle y no pasó por el "return
false", entonces $cadena es válido */
        return true;
}

if( permitirLetras("Pepé") ) {
        echo "Correcto";
} else {
        echo "Utiliza caracteres prohibidos";
}
?>
```

El punto clave está en extraer letra a letra de la cadena con substr($cadena,$i,1), que va leyendo, en cada vuelta del bucle, una letra diferente de lo escrito por el usuario, ya que cambia el carácter de comienzo proporcionado por $i.

Entonces, en cada vuelta del bucle, se comprueba si esa letra escrita por el usuario se encuentra dentro del texto de caracteres permitidos. La lógica de esta comparación permite que la modifiquemos fácilmente para incluir en la variable $letras aquellos caracteres que consideremos válidos (podrían ser los números del 0 al 9 para validar campos numéricos, o incluir letras, números y guiones para campos alfanuméricos, o acentos, eñes, etc.).

### Función strstr

La función **strstr** devuelve como resultado toda la cadena de texto hasta su final, a partir del **carácter** especificado.

Por ejemplo:

```
<?php
$email = "casilla@dominio.com.ar";
$caracter = "@";

$dominio = strstr($email,$caracter);
```

```
print($dominio);
// Imprimirá "@dominio.com.ar"
?>
```

Puede especificarse un tercer argumento, opcional, que invierte el sentido hacia el cual captura un segmento de texto: en lugar de ir desde el carácter buscado hasta el final de la cadena, en este caso, captura **desde el inicio** de la cadena **hasta** el carácter indicado.

Veamos un ejemplo:

```
<?php
$email = "casilla@dominio.com.ar";
$caracter = "@";

$dominio = strstr($email,$caracter,true);
print($dominio);
// Imprimirá "casilla"
?>
```

## Buscar algo en un texto con preg_match

Es común querer saber si un **carácter** o conjunto de caracteres "existe" dentro de otra cadena de caracteres; por ejemplo, para validar que en un campo donde el usuario debía escribir una dirección de correo electrónico esté presente una arroba, o un punto. O, por el contrario, para verificar que "no" haya ingresado caracteres **no permitidos**. Normalmente, con la función **strpos** –que tratamos en los párrafos precedentes– podremos realizar estas validaciones sencillas. Pero hay casos más complejos, en los que será necesaria una mayor potencia; para esos casos lo que tenemos a nuestra disposición la función preg_match.

En el caso de utilizar preg_match, las posibilidades se amplían, ya que podemos usar **expresiones regulares** (una herramienta de búsqueda mediante coincidencia de patrones que, quienes sean programadores, seguramente ya conocen, y que a continuación aprenderemos en su forma más elemental. Por esta razón, invitamos al lector a que investigue más acerca de ella).

```
<?php
```

```
/* Suponemos que venimos de un formulario donde el usuario
ya completó su dirección en la variable $email
*/
if ( preg_match("/@/", $email) ){
      print ("<p>Casilla correcta</p>");
} else {
      print ("<p>A su casilla le falta la arroba</p>");
}
?>
```

La búsqueda de una coincidencia también se hubiera podido realizar con *strpos* y, de hecho, resulta mucho más rápida. Guardemos la potencia de las expresiones regulares para su utilización en los casos en los que la búsqueda requiere una expresión regular capaz de obtener resultados que serían imposibles de hallar con funciones más simples.

Recordemos que el carácter o caracteres que se buscarán se deben envolver **entre barras**.

Veamos, entonces, un ejemplo más completo, que utiliza un **modificador** (la i que está a continuación de la barra de cierre) para obtener resultados en la búsqueda sin que importe si lo que queremos encontrar está en minúsculas o en mayúsculas.

```
/* Suponemos que el usuario escribió un Comentario en la
página anterior, y se almacenó en la variable $comentario
*/

$prohibido = "tonto";

if ( preg_match("/$prohibido/i", $comentario) ){
      print ("Escribió una palabra no permitida");
} else {
      print ("Gracias por su comentario");
}
?>
```

Notemos, luego del cierre de la barra final que envuelve lo que buscamos, la presencia del **modificador i** (de *ignore*, ignorar diferencias de mayúsculas o

minúsculas). Este modificador encontrará la palabra, inclusive si el usuario la escribió con mayúsculas en todas o en alguna de sus letras.

De la misma manera, podemos utilizar otros modificadores, por ejemplo, si envolvemos a la palabra a buscar entre dos **\b**, esto servirá para que la coincidencia sea **de palabra completa**, es decir, si buscamos "mar", que no devuelva verdadero si encuentra "amar" o "marrón". La sintaxis, sobre la base del ejemplo anterior, sería:

```
if ( preg_match("/\b$prohibido\b/i", $comentario) ){
```

Pero ahora será cuando se revele la verdadera potencia de las expresiones regulares: si lo que queremos es encontrar si los caracteres pertenecen a un **rango** determinado, podemos envolver ese rango entre **corchetes**, separando el valor inicial y final con **guiones**. Por ejemplo, si el valor permitido para cada dígito de un campo son solo números presentes en el rango que va del 0 al 9, lo podemos expresar de la siguiente manera:

```
if ( preg_match("/[^0-9]/", $numero) ){
```

Del mismo modo, es posible validar que el texto contenga únicamente letras de la **a** a la **z**:

```
if ( preg_match("/[^a-z]/", $numero) ){
```

Un complemento sumamente útil que podemos agregar para finalizar, es la posibilidad de verificar que una **casilla de correo** electrónico sea la correcta. Para evitar el uso de complejas expresiones regulares, PHP incorpora una función denominada **filter_var**, que permite realizar esta validación de una forma sumamente sencilla:

```
<?php
$casilla="pepe@deja espacio.com";

if ( filter_var($casilla,FILTER_VALIDATE_EMAIL)){

      echo "<p>Casilla correcta</p>";

} else {

      echo "<p>No es correcta la casilla</p>";

}
?>
```

Recomendamos que el lector continúe investigando la función **filter_var** en el manual oficial de PHP.

## Buscar y reemplazar: str_replace

Muchas veces, en lugar de limitarnos a rechazar lo que no cumpla con determinada regla de validación, preferiremos **cambiar a la fuerza** las palabras que consideramos incorrectas, o que corresponden a cierto "código" que definimos para que sea reemplazado por otra cosa (un uso muy útil de esto último es convertir caracteres usados en emoticonos por su correspondiente etiqueta img).

Para esa tarea, podemos recurrir a la función str_replace (string_replace, o reemplazar cadena).

Un ejemplo:

```
<?php

$cadena = "Este es mi comentario alegre :-) Saludos!";

$buscar = ":-)";

$reemplazo = '<img src="sonrisa.gif" alt="sonrisa">';


$nuevacadena = str_replace($buscar, $reemplazo, $cadena);


print($nuevacadena); /* Mostrará la imagen del emoticono
dentro del texto */

?>
```

## Dividir o ensamblar un texto con delimitadores: explode e implode

Cuando almacenamos información en un archivo de texto, una antigua (y práctica) técnica de almacenamiento de datos consiste en que cada línea se

subdivida en “partes” mediante un carácter **separador**, para, de esta manera, contener más de un dato por renglón. Así comenzaron las primitivas bases de datos y a estos renglones con separadores se los conocía como “registros”.

Por ejemplo, encontrarnos con una línea de estas características sería muy común en un supuesto archivo denominado usuarios.txt:

```
usuario1:clave1 usuario2:clave2
```

O, también, este otro caso:

```
Pepe|Perez|24|Soltero|Calle 1|234|3ro|A|
Carlos|Garcia|36|Casado|Calle 3|456|1ro|B|
```

Si necesitamos acceder a solo una de las “partes” de la línea, necesitaremos saber cuál es el carácter que obra de **separador** (en el ejemplo de los usuarios y claves, es el dos puntos : y, en el segundo caso, es |) para poder extraer o separar las partes divididas por ese separador y almacenarlas en una matriz, donde el [0] contendrá la primera parte, el [1] la segunda y así, sucesivamente. Sabiendo que la primera parte es el nombre, la segunda el apellido, etc. (lo sabemos, ya que nosotros mismos decidimos que se almacene así la información), podremos acceder a cualquiera de los datos del renglón con bastante precisión, tal como accedemos a los datos almacenados en cualquier matriz mediante su índice.

Veamos un ejemplo que acceda a un usuario y clave, almacenados en el mencionado archivo usuarios.txt, que separa usuarios y claves con el separador : –dos puntos. Este ejemplo solamente muestra cómo leer esos datos pero, una vez leídos, sería muy fácil validarlos:

```
<?php
$filas = file("usuarios.txt");
$cuantos = count($filas);

for ($i=0; $i<$cuantos; $i++){

      $partes = explode(":", $filas[$i]);
      print ("<p>Usuario: ".$partes[0]." - Clave:
".$partes[1]."</p>");

}
?>
```

Otro uso común de **explode** es **contar palabras** de un texto (cargando el texto entero en una matriz, usando como divisor el espacio en blanco entre palabras, y luego contando las celdas de esa matriz).

Ahora, la acción contraria: podemos tomar diferentes datos, de cualquier origen –por ejemplo, de distintos campos de un formulario– y **unirlos** en una única línea –lo que equivale a "pegar" partes o piezas–, intercalando entre parte y parte un carácter que haga de **separador**. Un ejemplo de la utilidad de esto es guardar en un solo renglón de un archivo de texto un conjunto de datos relacionados (es decir, "fabricar" un registro completo, similar a los que utilizamos en estos últimos ejercicios).

Para ello, usaremos la función **implode** de la siguiente manera:

```
<?php
/* Suponemos que el usuario completó sus datos en la
página anterior, y se almacenaron en las variables
$nombre, $email y $comentario */
$datos = array($nombre, $email, $comentario);

$unido = implode(":", $datos); /* Se le pasa la matriz
entera y la convierte en una cadena de caracteres que
queda almacenada en la variable $unido */

print ($unido);
?>
```

## Convertir saltos de línea a breaks HTML

En las áreas de texto de varias líneas (etiqueta *textarea* de HTML), en las que el usuario suele escribir una consulta o comentario, la descripción de un producto, etc., cada vez que él pulse la tecla **Enter** estará escribiendo un **"\n"** (salto de línea). El contenido de la petición HTTP que se envíe hacia el servidor remitirá los caracteres de control, incluidos los saltos de línea, de forma similar a lo que sigue:

```
Hola\nEstoy consultando por el producto publicado. Mi
teléfono es 4444-4444.\nAgradeceré su
llamado.\nSaludos\nJuan Pérez.\n
```

Si luego almacenamos ese texto en algún archivo de texto (o base de datos) cuando más adelante deseemos leerlo, veremos que **no se visualizarán los enter** escritos por el usuario. Esto sucede porque los **\n** para HTML no significan lo mismo que en los casos en los que se encuentran dentro de un archivo de texto plano (en el archivo HTML, generan saltos de línea en el **código fuente**, no en la pantalla).

Por ejemplo, el código anterior mostrado dentro de un archivo HTML con un **echo** o **print** generará lo siguiente dentro del código fuente:

```
[...]
<body> Hola
Estoy consultando por el producto publicado. Mi teléfono
es 4444-4444.
Agradeceré su llamado.
Saludos
Juan Pérez.
</body>
</html>
```

Sin embargo, el código expresado arriba, mostrará lo siguiente en la pantalla del navegador:


**Figura 10-1. Los saltos de línea en el código no son saltos HTML.**

Como observamos, los saltos de línea no se ven en el navegador. Pero podemos solucionar esta pérdida causada por el cambio de formato desde texto plano a código HTML, convirtiendo, con la función nl2br (*new line to break* o Nueva línea a *break*), cada uno de los saltos de línea en una etiqueta **<br>** de HTML, justo **antes** de guardar el texto generado por el usuario dentro de una base de datos o archivo de texto plano.

Al código del ejemplo anterior, suponiendo que llega de un formulario en un campo denominado "comentario", lo procesaríamos de la siguiente manera:

```
<?php
$formateado = nl2br($_POST["comentario"], false);
```

```
/* false devuelve <br>, true devuelve <br /> como en
XHTML */
echo $formateado;
?>
```

Lo que generaría este código fuente:

```
[…]
Hola<br>
Estoy consultando por el producto publicado. Mi teléfono
es 4444-4444.<br>
Agradeceré su llamado.<br>
Saludos<br>
Juan Pérez.
```

De esta manera, ahora sí mostrará en el navegador cada frase en una línea distinta, tal como el usuario la escribió en el campo de texto. Si nuevamente debiéramos almacenar este código en un archivo de texto plano, podríamos utilizar una de las recientemente aprendidas funciones de búsqueda y reemplazo, buscando cada **<br>** y reemplazándolo otra vez por un "\n".

## Funciones printf y sprintf

El uso de estas dos funciones es similar al de la conocida función **print**, es decir, escribir un texto dentro del código HTML que será enviado al navegador. Pero con el agregado de algunos **parámetros** que nos permiten **especificar el formato** que se le debe aplicar a los datos que le pasemos a la función.

Por ejemplo, si tenemos en una variable un número con siete decimales, podemos pedir que se muestren *solo dos*; en el caso de un número de un dígito, **rellenar con ceros** a la izquierda hasta completar la cantidad de dígitos que especifiquemos, por ejemplo: 00004.

La diferencia entre ambas funciones radica en lo siguiente: deberemos almacenar obligatoriamente en una variable el resultado de la ejecución **sprintf** y, más tarde, decidiremos si la imprimimos o la utilizamos en otra operación que contendrá el dato ya convertido; en cambio, **printf** envía directamente a imprimir por pantalla el resultado de la conversión en el mismo que la realiza (es decir, hace el mismo procedimiento que la función **print**).

Lo importante de estas funciones es que dentro de los paréntesis se especifica primero **de qué forma** se deben mostrar los resultados y, luego, se proporciona el lugar donde están **almacenados** los datos (típicamente, una variable). Dicho de forma poco técnica: "dos decimales, $numeros". Esto, escrito en forma "no técnica", significaría que se le dejan dos decimales a lo que haya en la variable $numeros. Ahora veremos cómo especificarlo para que funcione.

Los **parámetros** que se pueden especificar para dar formato a los datos son los siguientes:

1. **Carácter de relleno** (es opcional). El carácter que especifiquemos se utilizará para rellenar el dato al que lo aplicaremos hasta completar la cantidad de dígitos que hayamos señalado. El carácter de relleno puede ser un **espacio** o un **0** (cero). Si no aclaramos cuál usar, por omisión rellenará con espacios en blanco. Cualquier otro carácter de relleno distinto que deseemos utilizar, lo escribiremos anteponiéndole una comilla simple ("'").
2. **Alineación** (es opcional). Permite alinear el dato hacia la izquierda o hacia la derecha. Por defecto, se alinea hacia la derecha; un carácter **-** (signo menos) alineará hacia la izquierda.
3. **Longitud mínima**. Especifica cuántos caracteres (como mínimo) debería tener de longitud el dato. De no llegar a esa cantidad, lo rellenará con lo que hayamos especificado en el carácter de relleno.
4. **Decimales**. Indica cuántos decimales deberán mostrarse en números de coma flotante. Esta opción no tiene efecto para cualquier otro tipo de dato que no sea *double* (coma flotante).
5. **Tipo de dato**. Permite especificar el tipo de dato al que convertir el resultado.

| Carácter a colocar | El resultado quedará del tipo: |
|---|---|
| % | Un carácter literal de porcentaje. No se precisa argumento. |
| B | El argumento es tratado como un entero y presentado como un número binario. |
| C | El argumento es tratado como un entero, y presentado como el carácter con dicho valor ASCII. |
| D | El argumento es tratado como un entero y presentado como un número decimal. |
| F | El argumento es tratado como un doble y presentado como un |

| | |
|---|---|
| | número de coma flotante. |
| O | El argumento es tratado como un entero, y presentado como un número octal. |
| S | El argumento es tratado como una cadena de caracteres (string) y es presentado como tal. |
| X | El argumento es tratado como un entero y presentado como un número hexadecimal (minúsculas). |
| X | El argumento es tratado como un entero y presentado como un número hexadecimal (mayúsculas). |

**Tabla 10-2. Los tipos de dato posibles de printf y sprintf.**

Habitualmente, **d**, **f** y **s** son los modificadores más usados en este quinto parámetro de **printf** o **sprintf** (ya que estas letras corresponden a enteros, números con decimales y textos, respectivamente).

Veamos distintos ejemplos de aplicación de esta función.

Ejemplo de **sprintf** que convierte el dato a números **enteros**, rellenando con **ceros**:

```
<?php
/* Supongamos que tenemos almacenados en variables un día,
mes y año. */
$fecha = sprintf("%02d-%02d-%04d", $dia, $mes, $anio);
?>
```

Ejemplo que formatea valores monetarios a **dos decimales**:

```
<?php
$importe1 = 25.55;
$importe2 = 15.15;

$total = $importe1 + $importe2;
// Si imprimimos $total mostrará "40.7"

$dosDecimales = sprintf("%01.2f", $total);
echo $dosDecimales;
```

```
// Ahora mostrará "40.70" con dos decimales
?>
```

Seguiremos utilizando, a lo largo del libro, estas funciones para trabajar con cadenas de caracteres. En caso de necesidad, podemos seguir investigando las cerca de 100 funciones que posee esta categoría de manejo de caracteres, en el manual oficial de PHP: http://ar.php.net/manual/es/ref.strings.php

# Funciones de fecha y hora

## El concepto de timestamp

En PHP, tenemos a nuestra disposición diversas funciones para realizar operaciones con fechas y horas; pero para comprender cómo trabajan estas funciones, antes debemos dominar un concepto fundamental: el **timestamp** o "registro de hora" utilizado en los sistemas operativos Unix/Linux desde hace mucho tiempo.

El valor **timestamp** de un instante dado es la **cantidad de segundos** transcurridos desde el **primero de enero de 1970** hasta ese momento. Un ejemplo: 10 de enero de 1980 a las 10.30 de la mañana en formato **timestamp** se escribiría: 316359000 ya que ésa es la cantidad de segundos desde las cero horas del 01/01/1970 hasta las 10.30 del 10/01/1980.

Este número es muy práctico ya que, al ser un número **entero**, permite realizar operaciones con fechas simplemente sumándolas o restándolas, cosa que con fechas formateadas con barras o guiones no es posible (o, al menos, exige bastantes maniobras de conversión, nada sencillas).

A continuación, veremos cómo obtener el valor de **timestamp** de cualquier fecha, y cómo utilizarlo para distintas operaciones que involucren fechas y horas.

## Obtener fecha y hora actual con la función *time*

La función más básica para obtener el valor de *timestamp* de la fecha y hora actuales es la función llamada: **time** (tiempo). Esta función no necesita ningún parámetro obligatorio, y nos devuelve la cantidad de segundos pasados desde las cero horas del 01/01/1970 hasta el momento exacto en que es ejecutada (tomando en cuenta la hora definida en el servidor en el que se está ejecutando).

Probémosla:

```
<?php
$ahora = time();
print ($ahora);
?>
```

Si creamos una página con este código y actualizamos la página varias veces en el navegador, veremos cómo, cada vez que la ejecutamos, se va incrementando el número (ya que la cantidad de segundos transcurridos desde el 01/01/1970 hasta el momento actual cada vez es mayor).

Si queremos hacer operaciones, como por ejemplo, calcular un plazo de dos horas a partir del momento actual, podemos sumarle la **cantidad de segundos** necesaria (lo mismo vale para restas, multiplicaciones o divisiones: siempre la unidad de medida son los "segundos"):

```
<?php
$ahora = time();
$dosHorasMas = $ahora + 7200;
/* 60 segundos cada minuto, por 120 minutos que tienen las
dos horas, da 7200 segundos */
print ("Vence en el valor de timestamp: ".$dosHorasMas."
segundos");
?>
```

## Convirtiendo de timestamp a formatos más humanos

Si una vez obtenido el valor **timestamp** mediante la función **time** necesitamos mostrarlo de una forma más entendible, precisaremos emplear la función **getdate** (obtener fecha). Esta función necesita como parámetro un número de **timestamp** y, como resultado, genera una **matriz** de diez celdas, guardando un dato específico en cada una de las diez celdas, lo que luego nos permite utilizar por separado la hora, los minutos, los segundos, el día, mes, año, etc.

| Dato que se almacena | Índice donde se almacena |
|---|---|
| Hora | "hours" |
| Minutos | "minutes" |
| Segundos | "seconds" |
| Día del mes, en número | "mday" (*month day*) |
| Mes, en número | "mon" |
| Año, en número | "year" |
| Día de la semana, en número (comienza en "0" para el domingo, "1" para lunes, etc. y llega hasta el "6" -sábado-) | "wday" (*week day*) |
| Días transcurridos desde el principio del año, en número | "yday" (*year day*) |
| Día de la semana, la palabra completa, en inglés | "weekday" |
| Mes, la palabra completa, en inglés | "month" |

**Tabla 10-3. Los índices de la matriz que genera getdate.**

Veamos un ejemplo de su uso:

```
<?php
$ahora = time();
$detalle = getdate($ahora);// Convirtió a $detalle en una
matriz con las diez celdas que acabamos de mencionar

print("<p>Hora: ".$detalle["hours"]."<br>");
print("Minutos: ".$detalle["minutes"]."<br>");
print("Segundos: ".$detalle["seconds"]."</p>");

print("<p>Día: ".$detalle["mday"]."<br>");
print("Mes: ".$detalle["mon"]."<br>");
print("Año: ".$detalle["year"]."</p>");

print("<p>Día de la semana: ".$detalle["wday"]."</p>");

print("<p>Días desde el principio del año:
```

```
".$detalle["yday"]."</p>");

print("<p>Nombre en inglés del día de la semana:
".$detalle["weekday"]."</p>");

print("<p>Nombre en inglés del mes:
".$detalle["month"]."</p>");
?>
```

## Otra forma de mostrar fechas y horas: la función date

Una función de uso similar, pero un tanto más breve, es la función **date**.

También, puede (opcionalmente) partir de un número de **timestamp** referido a un momento en el pasado o en el futuro, o, si no especificamos ninguno, supone que queremos partir del instante **actual**.

La diferencia de *date* con respecto a la función anterior **getdate**, es que **date** no genera una matriz, sino una **cadena de texto formateada**, que nos permite especificar fácilmente muchas maneras distintas de mostrar una fecha u hora con simplemente una sola letra.

Veamos su tabla de letras clave:

| Para mostrar | Letra |
|---|---|
| Hora, de 01 a 12 | h |
| Hora, de 00 a 23 | H |
| Hora, sin ceros, de 1 a 12 | g |
| Hora, sin ceros, de 0 a 23 | G |
| Minutos, de 00 a 59 | i |
| Segundos, de 00 a 59 | s |
| Día del mes, con ceros, de 01 a 31 | d |
| Día del mes, sin ceros, de 1 a 31 | j |
| Mes, con ceros, de 01 a 12 | m |
| Mes, sin ceros, de 1 a 12 | n |
| Nombre del mes completo, en inglés | F |

| | |
|---|---|
| Abreviatura del mes, 3 letras, en inglés | M |
| Año, cuatro cifras | Y |
| Año, dos cifras | y |
| Día de la semana, en número (comienza en 0 para el domingo, y llega hasta el 6 -sábado-) | w |
| Días transcurridos desde el principio del año, en número | z |
| Día de la semana, la palabra completa, en inglés | l (es una "ele" minúscula) |
| Abreviatura del día de la semana, 3 letras, en inglés | D |
| Mes, la palabra completa, en inglés | F |
| Abreviatura del mes, 3 letras, en inglés | M |
| Número de días del mes, de 28 a 31 | t |
| Segundos desde el 1ro. de enero de 1970 (timestamp) | U |
| Diferencia horaria en segundos (de -43200 a 43200) | Z |
| "am" o "pm" | a |
| "AM" o "PM" | A |
| Si el año es bisiesto ("1") o no ("0") | L |
| Sufijo ordinal para los días, en inglés, 2 caracteres | S |

**Tabla 10-4. Tabla de letras de la función "date":**

Lo más interesante de la función *date* es que, dentro de ella, se pueden especificar **separadores** o **palabras** (con la única excepción de las letras del listado precedente, ya que si las incluimos, serían reemplazadas por su valor de fecha u hora).

Todo el conjunto de caracteres dentro de los paréntesis de la función **date** debe envolverse entre comillas.

```
<?php
print (date("d-m-Y")); // Observar el separador "-"
?>
```

El siguiente ejemplo daría un error, ya que las **d** imprimirán el número del día cada vez que aparecen, y la **l** mostrará el día de la semana.

```
<?php
print ("Hoy es ".date("d de m del Y"));
?>
```

Una solución sería **concatenar** tramos de texto con tramos en los que ejecutamos la función **date**:

```
<?php
print ("Hoy es ".date("d")." de ".date("m")." del
".date("Y"));
?>
```

Además de usarse sin especificar ningún valor de **timestamp** (con el valor por defecto, que es el momento actual), como en estos ejemplos, a la función **date** también se le puede especificar, como segundo parámetro, un **valor de timestamp**.

```
<?php
$hora = date("H:i:s",1451606399);
echo $hora;
?>
```

Esto mostrará la hora, minutos y segundos separados por : de la fecha proporcionada en el valor de **timestamp** que le pasamos como segundo parámetro.

## Zonas horarias

Pero, a veces, puede sucedernos que el valor de **timestamp** que le hemos proporcionado, sea interpretado con alguna diferencia horaria, debido a que el archivo de configuración php.ini de nuestro servidor o *hosting* tenga definida una **zona horaria** diferente a la nuestra o a la del público mayoritario de nuestro sitio.

Podemos corregir este desfasaje de dos maneras:

1. **Localmente**, editando el archivo php.ini y buscando esta parte:

```
 [Date]
; Defines the default timezone used by the date functions
; http://php.net/date.timezone
date.timezone = America/Argentina/Buenos_Aires
```

Allí, podemos especificar el código de la zona horaria según nuestra ubicación. La lista completa de zonas horarias que se utiliza como valor de "date.timezone" se encuentra en: http://www.php.net/manual/es/timezones.php

2. En un *hosting,* si bien no podremos editar el archivo php.ini, sí ejecutaremos la función date_default_timezone_set, que requiere, como parámetro, que le proporcionemos la misma cadena de caracteres que en el caso anterior.

   Veamos un ejemplo:

```
<?php
date_default_timezone_set('America/Argentina/Buenos_Aires'
);
?>
```

A partir de ese momento, pero solo durante la ejecución de esa página, estará vigente la zona horaria definida.

Suele ocurrir que, en ciertos países, en algunos años se decide **cambiar** el horario en alguna temporada, atrasando o adelantando una o más horas arbitrariamente. Para corregir esto, tendremos que crearnos una función propia que modifique todas las fechas que vayamos a utilizar, sumándole (o restándole) al valor de **timestamp** la cantidad de segundos necesarios (3600 segundos en el caso de una hora).

## De día, mes y año, a valor de timestamp: la función mktime

La función **mktime** *(make time* o crear tiempo) toma como parámetros una fecha y hora dadas, y nos devuelve el número entero **en formato timestamp** correspondiente a la fecha y hora proporcionadas. El orden de los argumentos es:

**hora**, **minutos**, **segundos**, **mes**, **día** y **año** (el mes va antes que el día). Si lo necesitamos, podemos especificar un último argumento opcional que determina si es **horario de verano** (en países en los que se adelanta la hora) en cuyo caso se escribe un **1** en ese argumento, un **0** si es horario de invierno, y un **-1** si se desconoce el dato.

Veamos un ejemplo en el que omitimos el argumento opcional (es lo más común); notemos que, al contrario de **date**, en **mktime** los argumentos **no llevan comillas**, ya que no son cadenas de caracteres, sino números enteros.

Atención: reiteramos que en esta función el mes va antes del día, como se escribe en algunos países.

```
<?php
$navidad = mktime(23,59,59,12,24,2020);
// No lleva comillas
print("La navidad de 2020 sucederá en el instante:
".$navidad);
?>
```

Un uso complementario de esta función es corregir errores como resultado de operaciones de suma o resta entre fechas. Por ejemplo, si le pasamos como argumento una fecha errónea como "32 de enero", nos devuelve un número **timestamp** que, si lo convertimos con **getdate**, veremos que ya fue convertido a "1 de febrero".

## Validar si una fecha es correcta con checkdate

Muchas veces, necesitaremos validar una fecha que haya ingresado un usuario en un campo de formulario. Para ello, nos sirve la función **checkdate**, que recibe como argumentos el mes, el día y el año (cuidado con el orden, el mes va antes que el día, recordar la función anterior).

Esta función dará como válidas las fechas entre el año 0 y el 32767, inclusive, meses del 1 al 12, y días entre el 1 y el 28 a 31, según el mes. Incluso considerando, de acuerdo con el año, la posibilidad de que sea bisiesto o no. Es una función *booleana*, es decir que su resultado solo puede ser *true* o *false*, lo cual nos facilita evaluarla dentro de un condicional.

Veamos ejemplos de su uso:

```
<?php
$primera = checkdate(12,32,2020);

if ($primera){ /* preguntamos tácitamente si la función
devolvió "true", verdadero */
      print("Fecha correcta");
} else {
      print("Fecha imposible");
}
?>
```

De esta forma, terminamos nuestro recorrido por las principales funciones de manejo de fecha y hora que trae predefinidas el lenguaje PHP. Si necesitamos consultar la totalidad de funciones agrupadas bajo esta categoría (cerca de cincuenta funciones), podemos hacerlo, como siempre, en el manual oficial de PHP: http://ar.php.net/manual/es/ref.datetime.php

# Funciones de envío de correos electrónicos

Hasta aquí, en este libro, hemos aprendido a utilizar distintos espacios de almacenamiento para nuestros datos (variables, constantes, matrices, *cookies*, sesiones y archivos de texto). Pero existe también la posibilidad de que, en lugar de almacenar cierta información, queramos **enviarla por correo electrónico**: el típico caso de un formulario de contacto del que desea recibir una copia el dueño o administrador del sitio. Pero, también, para recibir un aviso automatizado cuando haya sucedido cierta acción dentro de un sitio (por ejemplo, cuando se haya vendido un producto, o actualizado su precio o disponibilidad, o para enviar una nueva contraseña de acceso a quien la olvidó). Asimismo, para enviar un mensaje a repetición, mediante un bucle, a una lista de suscriptores.

Para estas tareas, PHP nos ofrece una función denominada, simplemente, **mail** (correo), que permite que enviemos correos electrónicos con formato de solo texto o con formato HTML, con cabeceras adicionales como copias, copias ocultas, archivos adjuntos, etc.

A continuación, aprenderemos a utilizar esta función.

## Un servidor de correos en el hosting

Atención: Para que todos estos ejemplos puedan funcionar, es preciso disponer de un servidor Web con un programa servidor de correos configurado y con acceso a Internet, en caso contrario, no podrá enviarse ningún mensaje.

Es común que los *hostings* tengan un servidor de correos habilitado, por lo que lo ideal es probar estos ejercicios en un *hosting* real (deberíamos tener contratado uno con PHP y MySQL).

En el caso de una PC hogareña con un servidor local, deberemos instalar un servidor de correos (para Windows, recomendamos MDaemon o Mercury y, bajo Linux, Postfix o Sendmail). Como la configuración de un servidor de correos es un tema técnico que puede ser bastante complejo, que excede largamente el alcance de este libro y compete más bien a administradores de sistemas operativos, buscaremos ayuda para configurarlo en la Web del servidor de correo que elijamos instalar.

## Envío básico

En capítulos anteriores, hemos aprendido a "recibir" la información que el usuario escribe en un formulario, leyéndola en las matrices $_GET o $_POST, y la hemos utilizado en la siguiente página. Pero, hasta ahora, simplemente nos limitamos a mostrar lo que el usuario elegía o escribía, o lo hemos almacenado en un archivo de texto.

Vamos a aprender cómo **enviar esos datos** hacia una casilla de correo electrónico de destino. Este envío será realizado automáticamente cuando ejecutemos la función mail de PHP: en ese momento, el software intérprete de PHP solicitará al servidor de correos especificado en el archivo de configuración php.ini que realice el envío y, si este programa servidor de correos logra realizarlo, la función devolverá *true*; de lo contrario, si falla, *false*.

La función mail de PHP necesita que le pasemos como mínimo los siguientes **tres parámetros** para funcionar:

1. La casilla de **destino** a la que debe enviarse el correo.
2. El **asunto** *(subject)* del correo.
3. El contenido o **cuerpo** del correo.

Existe la posibilidad de definir un **cuarto** parámetro de esta función destinado a agregar cabeceras del protocolo SMTP (*Simple Mail Transfer Protoco*l, o Protocolo Simple de Transferencia de Correo) pero, como es opcional, lo veremos unos párrafos más adelante.

Esta función **mail**, además de realizar la tarea de envío de un correo, es una función *booleana*, por lo que devuelve con su *return* un *true* o *false*, que es muy útil para validar dentro de un condicional si se pudo enviar el mensaje o no.

Podríamos pensar, sobre la base de nuestros conocimientos adquiridos hasta aquí, que bastaría con armar un formulario en el que **el usuario** fuese quien escriba estas tres cosas, en dos campos de texto y un área de texto (*textarea*), generando esas tres variables (casilla de destino, asunto y mensaje), para que, en la página siguiente, las usemos como parámetros de la función mail, pero... esto sería sumamente peligroso en un sitio Web real, ya que el **remitente** de esos mensajes enviados será el programa servidor de correos de nuestro *hosting;* imaginemos que los usuarios podrían enviar correo basura (SPAM) ya que **el remitente somos nosotros** (nuestro servidor, nuestra dirección *IP* perfectamente individualizable, lo que puede costarnos bastante dinero si el NIC decide penalizar y bloquear esa dirección IP, que inmediatamente dejaría inactivos a cientos de sitios Web alojados en el mismo servidor).

Para evitar este problema, no permitiremos que el usuario escriba alguno de esos tres argumentos. Imaginemos los casos:

1. Un formulario donde el usuario nos hace una **consulta** (al dueño del sitio). En este caso, pondremos la dirección de **destino** de forma fija, definiéndola dentro de una variable, y quien escriba no verá nunca hacia qué dirección sale enviado el correo, y sobre todo, **no podrá cambiarla**. Sí podrá especificar un asunto (si lo consideramos necesario, si no, lo definimos también nosotros) y eso sí, un cuerpo del mensaje.
2. Un formulario para **recomendar** nuestro sitio. En este caso, saldrá un correo despachado hacia la dirección de destino que el usuario elija (la de su "amigo"), pero **el texto** del mensaje y **el asunto** los escribiremos nosotros como dueños del sitio. A lo sumo, dejaremos que el usuario agregue un comentario, pero acompañado de una advertencia de que el mensaje fue enviado automáticamente.

Es decir, que siempre "desactivaremos" las combinaciones peligrosas.

## Casos prácticos

A continuación, aplicaremos la función **mail** en dos casos prácticos: el típico formulario de consultas que todo sitio posee y el formulario que permite recomendar una página a un amigo. Por supuesto que a medida que avancemos en el aprendizaje de PHP, se nos ocurrirán muchos otros usos prácticos de la función **mail**, en especial, cuando hayamos aprendido a interactuar con bases de datos.

### Formulario de consultas

Dentro del cuerpo de una página llamada consultas.html, agregaremos lo siguiente:

```
<form action="respuesta.php" method="post">
<fieldset>
      <legend>Consultas</legend>
      <p><label>Ingrese el tema de su consulta:
      <input type="text" name="asunto"></label>
      </p>
      <p><label>Ingrese su consulta:
      <input type="text" name="mensaje"></label>
      </p>
      <input type="submit" value="Enviar">
</fieldset>
</form>
```

Este formulario define dos variables (“asunto” y “mensaje”), que estarán disponibles en la página de destino (a la que llamaremos respuesta.php), dentro de las celdas: $_POST[“asunto”] y $_POST[“mensaje”].

La página respuesta.php contendrá esto (desde ya que es aconsejable validar que hayan sido enviadas las variables esperadas, y que no estén vacías, pero eso lo dejamos para aplicarlo por nuestra cuenta):

```
<?php
$destino = "micasilla@misitio.com";

mail($destino, $_POST["asunto"], $_POST["mensaje"]);
```

```
print("<p>Muchas gracias por su mensaje</p>");
?>
```

La función **mail** necesita que le pasemos exactamente en ese orden los tres parámetros o argumentos: el primero, debe ser una **dirección de correo** electrónico (en este caso, la tomamos de la variable $destino, que no se la dejamos definir al usuario sino que la llenamos con un valor nosotros mismos); segundo, el **asunto** del correo, que lo tomamos de la celda "asunto" de la matriz $_POST; y, finalmente, el **cuerpo** del mensaje, que lo leemos de la celda "mensaje" de $_POST.

### Recomendar una página

En el caso de "recomendar" una página, el usuario escribirá en el formulario la dirección de correo electrónico del amigo a quien le recomienda nuestro sitio, pero, para evitar problemas, **el asunto y el cuerpo** del mensaje lo definiremos nosotros, tal como hicimos con $destino del ejemplo anterior.

Supongamos que hemos puesto en el formulario de "Recomendar página" un campo para que el usuario escriba su nombre, uno para que escriba su correo electrónico, otro para que escriba el nombre de su amigo (el que recibirá el mensaje) y otro para la dirección del amigo (hacia la cual se despachará el mensaje de recomendación).

Sería bastante interesante que dentro del Asunto del mensaje, al destinatario le llegara algo como: "**Pepe: Juancito te recomienda visitar www.sitio.com.ar**".

¿Cómo hacemos esto? **Concatenando** cosas. Ya que a la función **mail** no podemos agregarle más que tres datos en el orden que vimos (más uno opcional que aplicaremos pronto) dentro de sus paréntesis, tendremos que preparar toda la información que se enviará dentro de esos tres lugares posibles, concatenando datos para el asunto y para el cuerpo del mensaje.

Veamos un ejemplo:

Archivo recomendacion.php:

```
<form action="envios.php" method="post">
      <fieldset>
      <legend>Recomendar esta página:</legend>
      <p><label>Su nombre: </label>
      <input type="text" name="nombreSuyo"></p>
```

```
      <p><label>Su Email: </label>
      <input type="text" name="emailSuyo"></p>
      <p><label>Nombre del destinatario: </label>
      <input type="text" name="nombreAmigo"></p>
      <p><label>Email del destinatario: </label>
      <input type="text" name="emailAmigo"></p>
      <p><label>Enviar datos: </label>
      <input type="submit" value="Enviar"></p>
</fieldset>
</form>
```

La página que procesa y envía el correo es la siguiente (aquí decidimos denominarla envios.php). Al ver su código, prestemos especial atención a la forma en la que se **concatenan textos** literales y **variables** en el asunto y en el mensaje:

```
<?php
$asunto = $_POST["nombreAmigo"].":
".$_POST["nombreSuyo"]." te recomienda visitar
www.sitio.com.ar";

$mensaje = $_POST["nombreSuyo"]." ha visitado
www.sitio.com.ar y te recomienda que visites este sitio";

if (mail($_POST["emailAmigo"], $asunto, $mensaje)){
      print("<p>Gracias por su recomendación</p>");
} else {
      print("<p>Ha fallado el servidor de correos, intente
más tarde</p>");
}
?>
```

Un detalle: podemos ejecutar más de una vez, por ejemplo, **dos** veces la función **mail**, logrando que le llegue un mensaje al "amigo" y otro correo a nosotros, conteniendo los datos de ambos, de quien envió y de quien recibió la recomendación (puede ser interesante para armarnos una lista de suscriptores):

```
if (mail($_POST["emailAmigo"], $asunto, $mensaje)){
      print("<p>Gracias por su recomendación</p>");
} else {
      print("<p>Ha fallado el servidor de correos, intente
más tarde</p>");
}
// Una vez realizado el envío anterior, proseguimos:
$miAsunto = "Nueva recomendación";

$miMensaje = $_POST["nombreAmigo"]." fue recomendado por
".$_POST["nombreSuyo"]." y sus casillas son: ".$_
POST["emailAmigo"]." y "$_POST["emailSuyo"];

if (mail("nosotros@nuestrositio.com"], $miAsunto,
$miMensaje))
{
      print("<p>Hemos recibido sus datos</p>");
} else {
      print("<p>Ha fallado el servidor de correos, no
poseemos copia de sus datos</p>");
}
```

Incluso, sería interesante ejecutar por tercera vez la función **mail**, para que al "recomendador" le llegue un mensaje de agradecimiento (poniendo su casilla como destino, en el primer parámetro, y preparando un par de variables con los textos específicos que le enviaremos en el asunto y en el cuerpo del mensaje).

Ya que al pasar nos hemos referido a la cantidad de veces que ejecutamos la función **mail**, seamos conscientes de que podemos incluso ejecutarla "a repetición", dentro de un bucle, por ejemplo, para enviar un mismo mensaje a toda una lista de destinatarios almacenada en un archivo de texto o en una base de datos.

## Agregar un remitente (From)

El único detalle a resolver es que los mensajes despachados por el servidor de correos de los servidores llegan con un remitente... que es el *nombre del servidor.* Esto queda evidentemente "feo", poco profesional.

¿Cómo podemos reemplazarlo para que se vea nuestro nombre y dirección de correo electrónico como **remitente**?

Aquí es donde podemos usar el **cuarto** argumento (opcional) que tiene la función mail, reservado para especificar una o más **cabeceras** *(headers)* del protocolo SMTP.

Una de estas cabeceras del mensaje es el **From**, es decir, la dirección de quien está enviando el mensaje (el remitente).

Se agregaría de esta manera (solo cambiamos la línea de la función **mail** y la anterior):

```
<?php
$asunto = $_POST["nombreAmigo"].":
".$_POST["nombreSuyo"]." te recomienda visitar
www.sitio.com.ar";
$mensaje = $_POST["nombreSuyo"]." ha visitado
www.sitio.com.ar y te recomienda que visites este sitio";

$remite = "From: Hernán Beati <hernan@beati.com.ar>";

/* Notemos cómo agregamos esta cabecera en cuarto lugar
dentro de los paréntesis */

if (mail($emailamigo, $asunto, $mensaje, $remite)){
      print("<p>Gracias por recomendar nuestro sitio</p>");
} else {

      print("<p>Ha fallado el servidor de correos, no se ha
enviado su mensaje</p>");
}

?>
```

**Nota para curiosos**: para conocer cuáles son las cabeceras más comunes *(Cc, Bcc,* etc.) del protocolo **SMTP** (el que utilizan los servidores para enviar correos electrónicos) solo tenemos que abrir con nuestro programa lector de correo

electrónico (Outlook, Thunderbird) el "código fuente" de un mensaje, y mirar las cabeceras.

## Enviando mensajes con saltos de línea y formato HTML

Si dentro del cuerpo de un correo electrónico enviado por PHP quisiéramos incluir **saltos de línea** para que quede más elegante el mensaje o, incluso, darle formato como si fuera una página Web (o un archivo HTML), podríamos probar alguna de estas dos opciones:

### Saltos de línea en texto plano

Si enviamos el correo en forma de texto plano (ésa es la manera en que PHP envía los correos si no especificamos nada), el salto de línea se genera escribiendo esto:

```
\r\n
```

(una barra hacia la izquierda –el Alt 92 del teclado numérico– y una **r** de *Return*, luego otra barra hacia la izquierda y una **n** de *Newline*, o nueva línea).

Por ejemplo:

```
$mensaje = $_POST["nombre"]." envía la siguiente
consulta:\ r\n".$_POST["comentario"]."\r\nSu correo
electrónico es: ".$_ POST["correo"]."\r\nSu teléfono es:
".$_POST["telefono"];
```

Al recibir este mensaje en un programa lector de correos, se verá un renglón debajo del otro, por obra de los **\r\n**, visualizándose el mensaje de la siguiente manera:

```
Pepe envía la siguiente consulta:
Hola, quería solicitar un presupuesto, escríbanme.
Su correo electrónico es: pepe@pepe.com
Su teléfono es: 4444-4444
```

Notemos que comienza una nueva línea cada vez que colocamos entre comillas dobles un **\r\n**.

### Cuerpo del mensaje con código HTML

Otra opción es que, en el momento de hacer el envío del mensaje, el programa intérprete de PHP incluya ciertas **órdenes**, que le especifiquen al programa lector de correos que recibirá ese mensaje, que el cuerpo de ese correo contiene **etiquetas HTML** y que, por lo tanto, deberá procesarlo como si fuera un archivo HTML o una página Web normal.

Para especificar este formato HTML podemos utilizar el cuarto parámetro de la función **mail**, el mismo donde hemos declarado la cabecera **From**, ya que este cuarto parámetro se refiere a todas las "cabeceras" o *headers* del mensaje que se va a enviar. Mediante una de esas **cabeceras**, PHP le puede avisar al lector de correos que el código de ese mensaje está escrito en lenguaje HTML. Para colocar en el cuarto parámetro más de una cabecera, debemos separarlas con **saltos de línea** (los *\r\n,* que acabamos de aprender en el punto anterior).

Entonces, si además de especificar un remitente con el encabezado **From**, quisiéramos especificar con otro encabezado el tipo de lenguaje (HTML) que se está utilizando para escribir el cuerpo del mensaje, lo haremos de la siguiente manera:

```
<?php
$destino = "alguien@sitio.com";

$asunto = "Datos de contacto";

$cuerpo = '<h1>Promoción especial</h1><p>Visite nuestra <a
href="http://www.sitio.com">página Web</a> y aproveche
esta oferta: <img
src="http://www.sitio.com/imagenes/promocion.jpg"
alt="¡Tres paquetes por un peso!"></p>";

$cabeceras = "MIME-Version: 1.0\r\nContent-type:
text/html; charset=utf-8\r\nFrom: Hernán Beati
<hernan@beati.com. ar>\r\n";
/* Notemos cómo cada cabecera se separa de otra mediante
un salto \r\n */


if (mail($destino, $asunto, $cuerpo, $cabeceras)){
```

```
      print("<p>Gracias por su consulta en formato
HTML</p>");
} else {

      print("<p>Ha fallado el servidor de correos, intente
nuevamente</p>");
}
?>
```

La función **mail** será muy útil cuando armemos sistemas completos, con bases de datos, ya que será la forma más directa de que nuestros sistemas se mantengan comunicados automáticamente con los usuarios registrados, enviando noticias, mensajes, cambios de contraseñas, etc. muy fácilmente.

Vamos a aplicar ahora varias de las funciones aprendidas en este capítulo, no solo de envío de correos sino de manejo de cadenas de caracteres, para crear una función propia que impida que utilicen nuestros formularios para el envío de SPAM.

Nos concentraremos en que nuestra función detecte, en cada campo donde pueda escribir el usuario, si está presente cierta **cadena de caracteres** que consideremos "riesgosa", ya que potencialmente podría agregar **cabeceras SMTP** a nuestro envío, con la posibilidad de que en lugar de un solo envío, un *spammer* realice varios envíos de su texto a distintas direcciones, gracias a la cabecera *Bcc*.

Veamos cómo podría ser esta función:

```
<?php
function validarDatos($campo){
      /* Matriz con las posibles cabeceras a utilizar por
un spammer */
      $cabeceras = array("Content-Type:","MIME-
Version:","Content-Transfer-Encoding:", "Return-
path:","Subject:", "From:","Envelope-to:", "To:","bcc:",
"cc:");
      /* Ahora comprobamos que entre los datos no se
encuentre alguna de las cadenas de la matriz. Si se
encuentra alguna de esas cadenas, se muestra un mensaje */
```

```
      $bandera = "si";// Usamos un flag o señal

      foreach ($cabeceras as $valor){

         if( strpos( strtolower($campo),strtolower($valor))
!==false ) {
            $bandera = "no"; // Cambiamos la señal
         }      // Cierra if
      }// Cierra bucle

if ($bandera == "no"){
      return false;
} else {
      return true;
}
}// Cierra función
?>
```

Un ejemplo de llamada a esta función para validar un campo:

```
if ( isset($_POST["nombre"]) ){

      if (!validarDatos($_POST["nombre"]) ){
         echo "<p>ATENCION: Detectamos que se está
intentando enviar caracteres no permitidos dentro del
Nombre.</p>";
      }

}
```

Con esto, ya estamos en condiciones de seguir explorando por nuestra propia cuenta, en el manual oficial de PHP, el resto de las **funciones predefinidas** que nos ofrece este lenguaje, y de utilizar las funciones de manejo de caracteres, de fecha y hora, y de envío de correos en nuestros próximos trabajos que, a partir del capítulo siguiente, integrarán la enorme potencia de las bases de datos.

# Creando bases de datos

# 11

## El almacén de datos más potente para nuestros sitios Web

En capítulos anteriores, hemos realizado ejercicios en los cuales le solicitábamos al intérprete de PHP que **obtuviera información** de un archivo de texto, o que **guardara información** dentro de un archivo. Ese archivo de texto era el lugar físico donde se almacenaban los datos, y el programa intérprete de PHP podía acceder a ellos para leerlos, borrarlos o agregarles más información.

Con las bases de datos será similar. Le pediremos al programa intérprete de PHP que haga de "intermediario" entre la **base de datos** y nuestras **páginas** que mostrarán o proporcionarán esos datos (según en qué sentido los hagamos circular).

De la misma manera que con los archivos de texto, contamos con numerosas funciones de PHP predefinidas, capaces de **abrir una conexión** con una base de datos, de **escribir** o **leer** datos allí guardados, de traspasarlos a una matriz, y otras funciones muy prácticas que nos facilitan la interacción con la información que almacenemos en bases de datos.

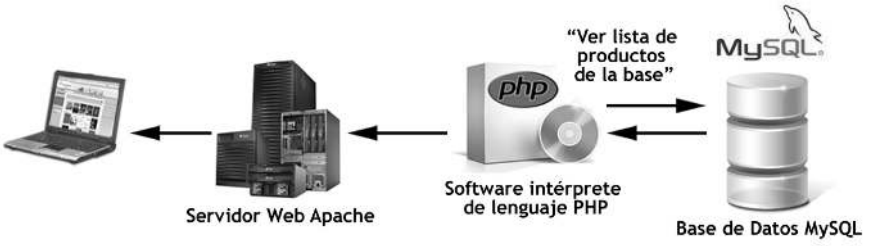


**Figura 11-1. El intérprete de PHP solicita a MySQL datos y los muestra en una página.**

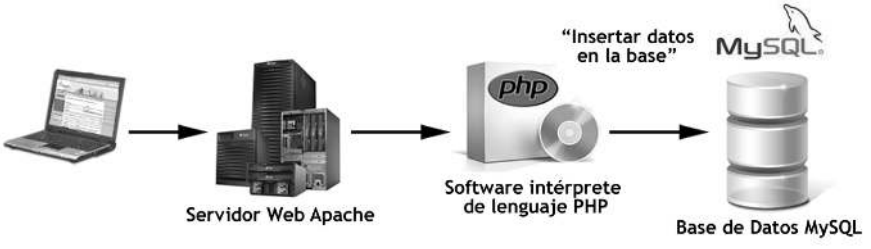


**Figura 11-2. El intérprete de PHP envía datos a MySQL para que los almacene.**

# Diferencia entre archivos de texto y bases de datos: el lenguaje SQL

Pero, ¿por qué necesitaríamos bases de datos si ya manejamos archivos de texto? La principal diferencia entre almacenar la información en un archivo de texto y hacerlo en una base de datos es la **estructuración** y posterior **manipulación** que podemos hacer con los datos.

Es mucho más fácil trabajar con **filas** que posean varias **columnas** en una base de datos, que en un archivo de texto; también es más fácil **buscar** y encontrar determinadas filas ordenadas, cuyo valor en alguno de sus campos cumpla con una determinada **condición** dentro de una base de datos; puesto que el acceso es más rápido y seguro, y se puede administrar un gran (enorme) volumen de datos con mayor facilidad.

En cambio, en un archivo de texto, lo máximo que podemos hacer es utilizar un carácter como separador entre dato y dato (una rudimentaria aplicación del concepto de campos o columnas propio de las bases de datos), y mientras que

nuestras únicas operaciones posibles son: "agregar" unos caracteres o un renglón al principio o al final del archivo; "reemplazar" el contenido completo del archivo; "leer" secuencialmente una cantidad de caracteres o el archivo completo; todas operaciones basadas en la **ubicación física** de los datos (necesitamos saber **en qué renglón está** el dato que buscamos); en una base de datos, contamos con muchísimas más herramientas proporcionadas por el **software gestor** de la base (en nuestro caso, el programa denominado MySQL).

La principal herramienta diferenciadora que nos ofrece un gestor de bases de datos es su capacidad para interpretar un **lenguaje declarativo** que permite la ejecución de diversas operaciones sobre una base de datos. En el caso de MySQL, el lenguaje declarativo se llama **SQL** (*Structured Query Language* o Lenguaje de Consultas Estructurado).

Gracias a este lenguaje, expresaremos qué datos exactos necesitamos en una forma **declarativa**, cercana al lenguaje natural que utilizamos para hablar entre personas (aunque deberemos usar palabras específicas del idioma inglés), y será el software gestor (MySQL) el que se ocupará de realizar aquellas operaciones "físicas" sobre el archivo que almacene los datos, liberándonos de la tediosa tarea de tener que programar en ese nivel tan bajo de manipulación de archivos, renglón por renglón, dentro del disco rígido.

Expresaremos una frase parecida a la siguiente: *"Traer la lista de los productos que pertenezcan a la categoría electrodomésticos ordenados de menor precio a mayor precio"*, y el software MySQL hará todo el resto del trabajo por nosotros.

¿Para qué necesitamos lidiar con bases de datos en nuestras Webs? Porque **prácticamente todos** los sitios Web medianos o grandes, todos los portales, los sitios de noticias que se actualizan al instante y las redes sociales las utilizan para almacenar los **contenidos** de sus páginas, contenidos elaborados por personas ubicadas en cualquier parte del mundo, que se agregan a la base de datos mediante un formulario, y que, sin que ningún diseñador necesite darles formato, quedan publicados instantáneamente.

Esta "magia" es posible gracias a las bases de datos (y al lenguaje SQL). Otros usos sumamente habituales son: buscar dentro de los contenidos de esas páginas, almacenar los datos de los usuarios registrados en un sitio, recoger sus opiniones mediante encuestas, guardar mensajes de un foro, comentarios dejados en un libro de visitas o un *blog,* mostrar productos de un catálogo de comercio electrónico, las actividades diarias de una agenda, los contactos de una libreta de direcciones, los elementos de un portal, un campus virtual, y un casi infinito etcétera. Estos usos amplían las funcionalidades que podemos incluir en nuestros sitios Web, abriéndonos nuevos mercados, al permitirnos ofrecer servicios que son imposibles sin bases de datos.

## Programas que utilizaremos

Los nuevos programas implicados serán, en nuestro caso, dos:

1. El programa **gestor de bases de datos** denominado **MySQL** (un programa que si bien originalmente funciona mediante línea de comandos, nunca lo usaremos de esta forma); y,
2. Justamente, para facilitarnos la interacción con el programa anterior, utilizaremos una **interfaz** o serie de pantallas donde podremos interactuar con la base de datos a partir de herramientas visuales; nosotros emplearemos **phpMyAdmin**, aunque podemos investigar también MySQL Front, MySQL Administrator, HeidiSQL, o cualquier otra interfaz visual para MySQL.

Ambos programas (MySQL y phpMyAdmin) ya los tenemos instalados si hemos utilizado un instalador como XAMPP o similares.

Por medio de estos dos programas, lo que realmente estaremos ejecutando son consultas escritas en **lenguaje SQL**. Lo básico de este lenguaje también lo aprenderemos muy pronto.

# Conceptos fundamentales: base, tabla, registro y campo

Vamos a ponernos de acuerdo con el vocabulario: no es lo mismo una base que una tabla, ni es lo mismo un registro que un campo. Veamos las diferencias.

## Base de datos

Nuestro gestor de bases de datos MySQL nos permitirá crear tantas **bases de datos** como proyectos necesitemos (y como espacio tengamos en nuestro disco rígido, o en el *hosting* que utilicemos).

Conceptualmente, una base de datos es un "paquete" que contiene **toda** la información necesaria para el funcionamiento de un sistema o proyecto completo. Por ejemplo, una base de datos llamada "tienda" puede almacenar todos los datos de un sistema de comercio electrónico (incluyendo datos sobre los productos, las ventas, el inventario, la facturación, las formas de pago, las formas de envío, etc.). Cada base de datos es un almacén donde se guarda información sobre un **conjunto completo de información relacionada**, necesaria para que un sistema completo funcione.

Físicamente, cada nueva base de datos en MySQL crea un **directorio** o **carpeta** que contendrá los archivos de datos de esa base. Al igual que sucede en la relación entre carpetas y archivos, por sí sola, la base no significa nada, es similar a una carpeta, un simple contenedor. Lo que guarda la información son los objetos que tiene almacenados adentro. Pues bien: la información dentro de las bases de datos se guarda en **tablas** (las bases de datos son simplemente **conjuntos de tablas**).

## Tablas

Volviendo al ejemplo de la base de datos de una "tienda", esta base podría contener las siguientes **tablas**:

- productos,
- categorías,
- usuarios,
- pedidos,
- envíos,
- pagos.

Es decir, cada uno de los conceptos sobre los cuales necesitamos guardar datos corresponderá a una **tabla**. Por ejemplo, crearemos una tabla para los "productos", ya que la información que tenemos acerca de cada producto comparte **una misma estructura**: tendremos un nombre para cada producto, una descripción, un precio, una cantidad disponible, etc.

Todos los contenidos de una misma tabla deben compartir la misma estructura.

## Campos

La estructura de una tabla se define por la cantidad de **campos** en que fraccionemos la información que guarda. Los posibles campos (podemos imaginarlos como "columnas" de una planilla de cálculo) para una tabla de –por ejemplo– "productos", podrían ser el código de producto, el nombre del producto, su descripción, la categoría a la que pertenece, el precio, etc. Cada campo tendrá definido un **tipo de dato** que limitará lo que podrá almacenarse en él (datos numéricos, alfanuméricos, fechas, etc.) y, también, le definiremos a cada campo una **longitud** máxima (el "ancho" de la columna, siguiendo el ejemplo de una planilla); es decir, la cantidad máxima de caracteres que prevemos almacenar en ese campo.

## Registros

Cada ítem de esa tabla (cada "producto", en el ejemplo anterior) se almacenará en un **registro** (una fila horizontal, un renglón).

Cambiemos de ejemplo, y veamos los elementos de una tabla dedicada a almacenar "mensajes" que los usuarios enviaron mediante un formulario:

| id | nombre | email | mensaje |
|---|---|---|---|
| 1 | Juan Pérez | juan@perez.com | ¡Hola amigos! |
| 2 | Carlos García | carlosgarcia@hotmail.com | Saludos desde América. |
| 3 | María González | mgonzalez@gmail.co | Me gusta PHP. |

Lo que vemos en la primera fila (los títulos en negrita) representan lo que sería la **estructura** de la tabla: en este caso, qué información se guardará relativa a cada "mensaje" (ya que estamos ante una tabla llamada "mensajes").

En el ejemplo, hemos decidido estructurar la información en **cuatro columnas**: "id", "nombre", "email" y "mensaje". A estas columnas se las denomina **campos** (se dice: el campo "id", el campo "nombre", el campo "email" y el campo "mensaje"):

| id | nombre | email | mensaje |
|---|---|---|---|

Luego, cada **fila** (horizontal) representa un dato completo o un **registro**, es decir, la suma de todos los campos (la información completa que se dispone) sobre **uno** de los "mensajes" recibidos y sobre uno de los objetos acerca de los cuales almacenamos información.

La fila 1 (el primer registro) contiene los datos del primer mensaje (el de Pérez):

| 1 | Juan Pérez | juan@perez.com | ¡Hola amigos! |
|---|---|---|---|

La fila 2 (el segundo **registro**) tiene los datos del mensaje de García:

| 2 | Carlos García | carlosgarcia@hotmail.com | Saludos desde América. |
|---|---|---|---|

y así sucesivamente.

Entonces, en resumen:

- una base de datos contiene una o más tablas,

- una tabla se estructura en campos (columnas),
- cada fila o renglón se llama registro.

Ya fijaremos estos conceptos a medida que los sigamos ejercitando.

# Creando bases y tablas con phpMyAdmin

Existen varias maneras de **crear** una base de datos, pero la más común es disponer de un programa que nos permita crearla de manera "visual" y no mediante códigos SQL ni líneas de comandos.

El software más utilizado para esto es el phpMyAdmin (es de código abierto, gratuito, y suele venir preinstalado en los *hosting*s que poseen MySQL).

Si queremos hacer esto localmente, ante todo, como siempre, tendremos que encender nuestro servidor Web Apache y nuestro gestor MySQL:

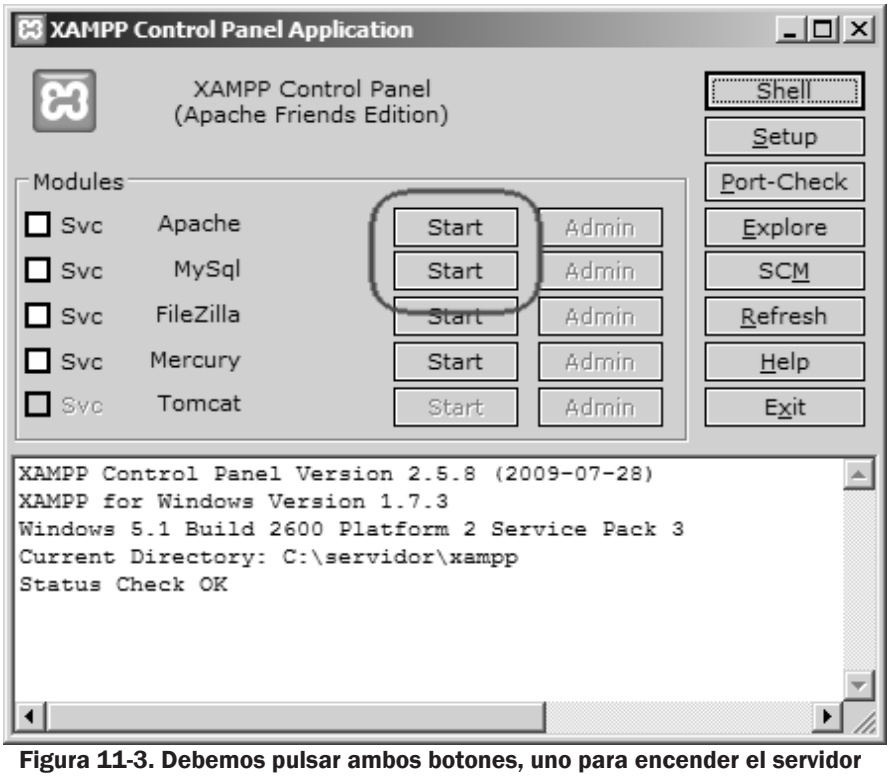


**Figura 11-3. Debemos pulsar ambos botones, uno para encender el servidor Web Apache, y otro para MySQL.**

Luego, con el navegador **abriremos el phpMyAdmin**. La ruta puede variar según el instalador que hayamos utilizado y según la versión de phpMyAdmin que tengamos instalada, pero en el caso del XAMPP, podremos abrir el navegador en la página del phpMyAdmin si entramos a:

http://localhost/phpmyadmin/

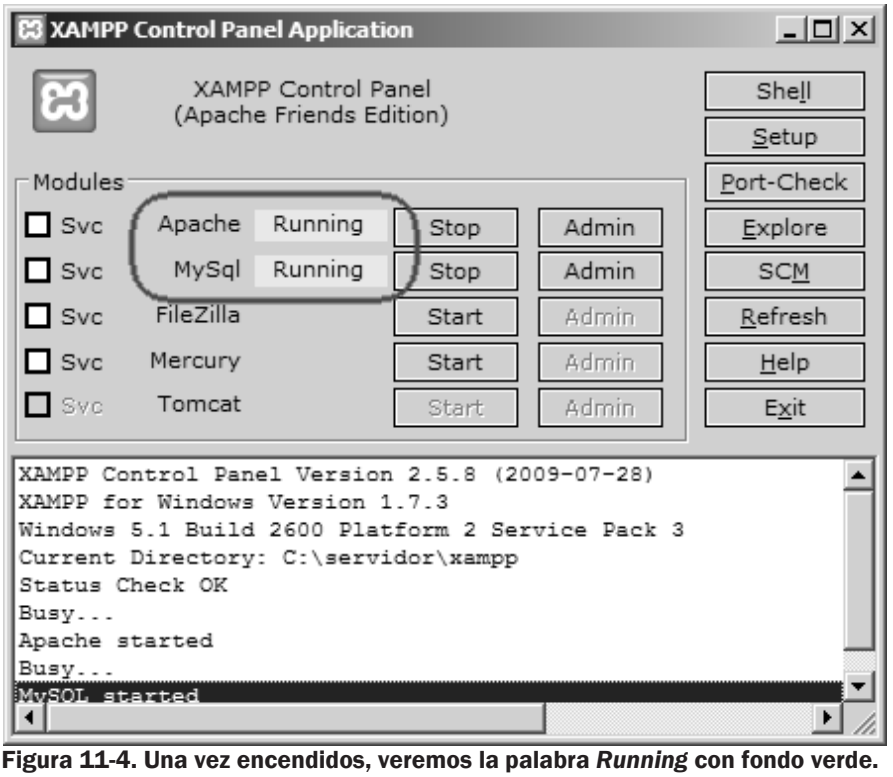


**Figura 11-4. Una vez encendidos, veremos la palabra *Running* con fondo verde.**

O también podemos pulsar el botón de acceso directo que dice **Admin** a la altura del renglón de MySQL en el panel de administración del XAMPP:

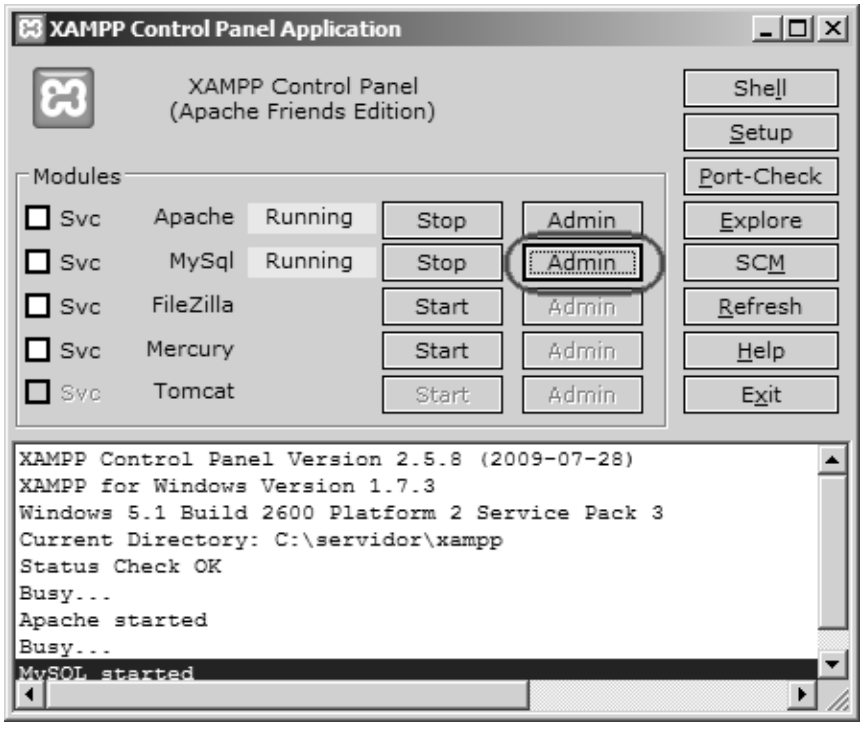


**Figura 11-5. En el XAMPP, pulsamos el botón "Admin" para abrir el phpMyAdmin.**

Cuando ingresemos por primera vez a esa dirección, luego de identificarnos, veremos algo similar a la siguiente figura:

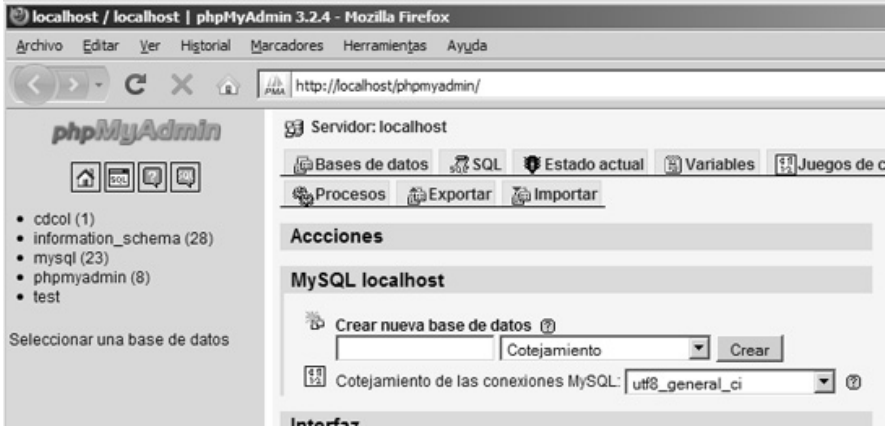


**Figura 11-6. Página de inicio de la aplicación Web phpMyAdmin.**

## Seguridad

Llegados a este punto, en el que vamos a comenzar a utilizar bases de datos, es buen momento para que le demos una mayor **seguridad** a nuestra instalación local ya que, hasta ahora, no hemos protegido dos puntos críticos:

1. Cualquier persona que conozca o adivine nuestra dirección IP, puede navegar por las páginas que sirve nuestro servidor Web local, incluyendo el phpMyAdmin (por esta razón, podría borrar o leer todas nuestras bases de datos); esto se remedia definiendo un **usuario y clave para ingresar al servidor local** con un navegador (tendremos que ingresarlos cada vez que queramos probar nuestros archivos localmente).
2. Debemos cambiar la **contraseña del usuario llamado "root"**, que tiene permisos de administrador en el programa MySQL, y que, por ahora, está vacía (no tiene ninguna contraseña aún).

Podemos realizar estas dos tareas muy fácilmente desde la página inicial del XAMPP, entrando a:


**Figura 11-7. Enlace de acceso a configurar niveles de seguridad en nuestro servidor local.**

Una vez que entremos, veremos que esa página PHP realiza una serie de comprobaciones, resaltadas en color rojo, amarillo o verde según su riesgo:


**Figura 11-8. Chequeo de seguridad.**

Para solucionar todo esto, en la página del chequeo, el XAMPP nos proporciona un enlace al que deberemos entrar:

http://localhost/security/xamppsecurity.php

Luego, seguiremos las instrucciones en pantalla.

Atención: es crítico que anotemos, en un lugar seguro, estos datos (usuarios y claves), ya que, de lo contrario, no podremos ingresar más al servidor local y deberíamos reinstalarlo.

Este enlace abrirá una pantalla que, ante todo, nos pedirá que le asignemos una clave al usuario "root" de MySQL. Colocamos la clave que deseemos (para ejemplo, vamos a suponer que escribimos la palabra "clave"), y la repetimos, y dejamos marcadas las opciones que aparecen por defecto:

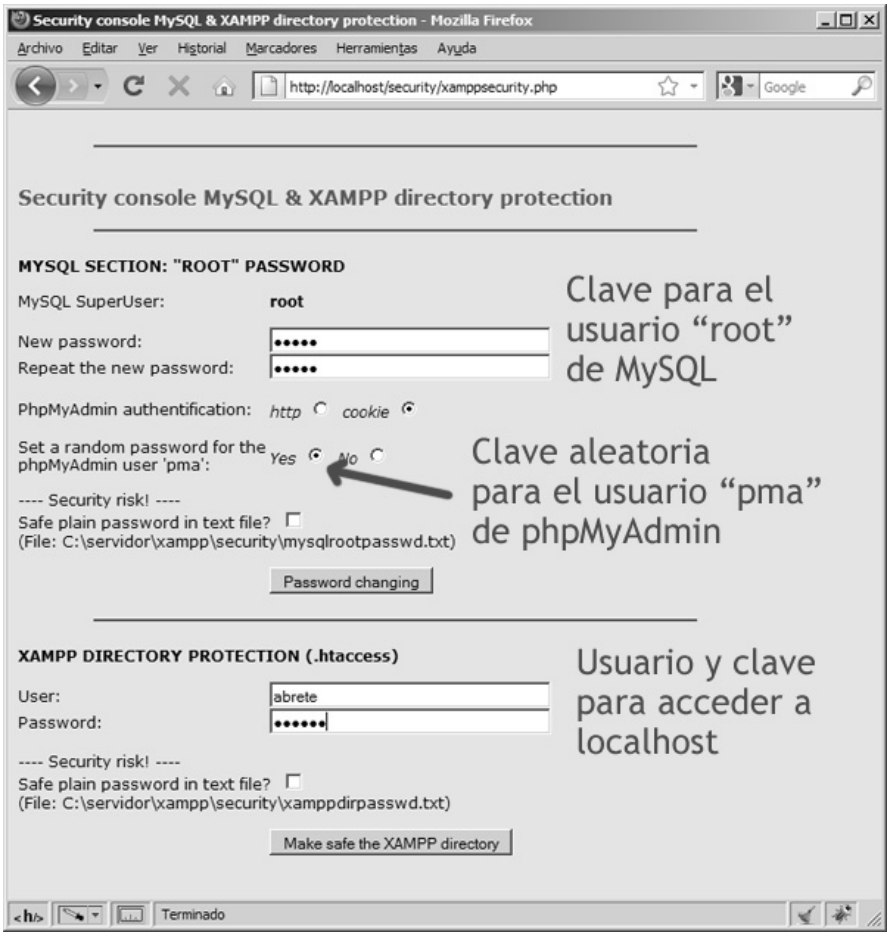


**Figura 11-9. Clave para el usuario "root" de MySQL.**

Pulsamos el primer botón gris que dice *Password changing*, que nos pedirá que apaguemos y encendamos nuevamente MySQL (lo hacemos desde el panel de control del XAMPP, pulsando **Stop** a la altura del renglón correspondiente a MySQL, y, luego, nuevamente **Start**).

Después, pasaremos a la parte inferior de esa pantalla y definiremos una clave de acceso para entrar con el navegador a las páginas servidas por nuestro

servidor Web local, nuestro *localhost*. En este caso, hemos colocado un usuario llamado "abrete" y una clave "sesamo", y pulsamos el botón gris al final de la pantalla, que dice *Make safe the XAMPP directory.*

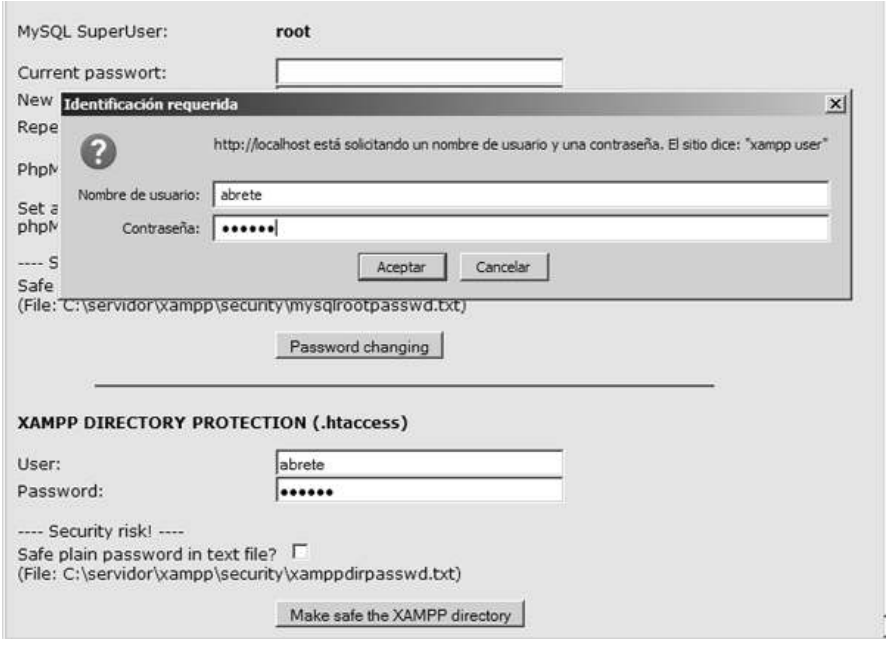


**Figura 11-10. Desde ahora, deberemos ingresar nuestro usuario "abrete" y la clave "sesamo" para entrar a localhost.**

Nos indica que los datos de acceso fueron guardados en dos lugares:

C:\servidor\xampp\security\xampp.users

C:\servidor\xampp\htdocs\xampp\.htaccess

Una vez configurado todo esto, estamos listos para trabajar con bases de datos de una manera físicamente segura, que impida el acceso a nuestros datos. Anotemos estos usuarios y claves, ya que continuamente los necesitaremos para poder interactuar con nuestras bases de datos.

Por supuesto, esta operación que realizamos localmente, en un *hosting* ya se han ocupado de hacerla, por lo que solo deberemos conocer la **URL de acceso al phpMyAdmin**, y el **usuario** y **clave de MySQL** (ya no será "root" ese usuario, sino otro que podremos crear desde algún panel de administración de los servicios del *hosting,* que deberemos consultarlo a la empresa de *hosting* que contratemos, ya que en cada empresa el proceso y las herramientas pueden variar, incluso en algunos *hostings,* ellos crearán nuestro usuario y clave para MySQL).

Regresando a nuestro servidor local, si ahora volvemos a ingresar a:

http://localhost/security/

veremos que los puntos de chequeo esta vez ya fueron dados por aprobados, y los observamos en color verde (o amarillo para los que no se aplican, como el servidor de FTP y Tomcat, que no necesitamos activar).

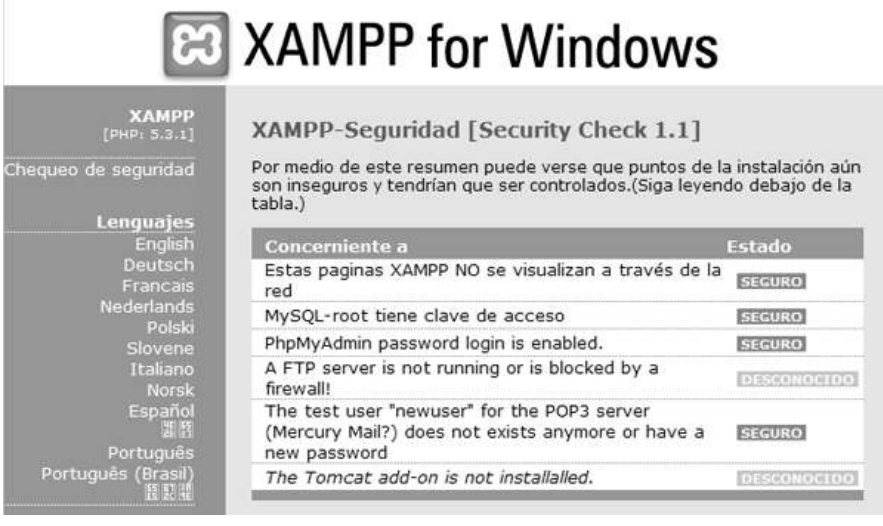


**Figura 11-11. Ahora, nuestra instalación está asegurada.**

## Crear una base de datos

Una vez asegurado el acceso físico a nuestra instalación, abriremos el navegador y entraremos a localhost, ingresando el **usuario** y **clave** que hayamos creado (el "abrete" y "sesamo", en nuestro ejemplo).

A continuación, como cada vez que necesitemos interactuar con nuestra base de datos, abriremos el phpMyAdmin. En caso de haber utilizado el XAMPP, la URL que escribiremos será:

http://localhost/phpmyadmin/

Nos pedirá que nos identifiquemos con un **usuario** y **clave** de MySQL, y utilizaremos el usuario "root" con su contraseña llamada "clave" que habíamos definido anteriormente.

Una vez identificados, se nos permitirá el acceso al phpMyAdmin. Tengamos en cuenta que si no utilizamos la base por más de 1440 segundos (24

minutos) nos pedirá que ingresemos nuevamente nuestro usuario y clave para MySQL.

Con el phpMyAdmin abierto, lo primero que vamos a hacer ahora es crear una **nueva base de datos** (muchos *hosting*s ya traen preinstalada una, con lo cual en esos *hosting*s nos saltearíamos este paso, y directamente usaríamos la base que ya exista).

En cambio, localmente (en nuestro servidor de pruebas) podremos crear una nueva base de datos para cada proyecto en el que estemos trabajando.

**Atención**: las bases de datos que estamos viendo enumeradas a la izquierda de nuestro phpMyAdmin, llamadas "cdcol", "information_schema", "mysql", "phpmyadmin" y "test", que vienen preinstaladas, y que contienen las tablas necesarias para el funcionamiento de MySQL y de phpMyAdmin, son bases de datos demasiado importantes como para que corramos el riesgo de borrarles o modificarles algún dato por accidente, ya que guardan la configuración y los permisos de todos los usuarios de nuestro servidor local, por lo cual, en caso de modificarlas o borrarlas, no podremos utilizar más MySQL y deberemos reinstalar los programas. ¡No toquemos esas bases! (de hecho, en un *hosting*, ni siquiera nos permitirán visualizarlas, son patrimonio del administrador del *hosting*).

Para crear una nueva base de datos, dentro del phpMyAdmin escribiremos (en la zona que se resalta a continuación) el **nombre** que le queremos dar (vamos a denominarla "cursos"):


**Figura 11-12. Creación de una nueva base.**

Hagamos, paso a paso, lo que nos muestra este ejemplo: dentro del campo de texto escribamos **cursos** (en minúsculas), pues ése será el nombre de nuestra nueva base de datos.

Luego de escribir el nombre, elegiremos el juego de caracteres que almacenaremos (para textos en español, será el **utf8_spanish_ci**, que corresponde al español tradicional, y permite que utilicemos la **ñ** y la **ch** y **ll**). Si tuviéramos que crear una base para un cliente coreano, japonés, árabe, chino, etc., deberíamos elegir el correspondiente juego de caracteres:

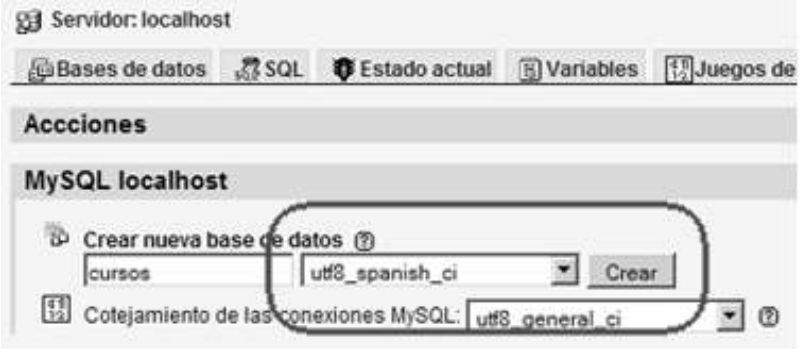


**Figura 11-13. Eligiendo juego de caracteres.**

Finalmente, pulsaremos el botón **Crear** y, a continuación, el **nombre** de la base recién creada aparecerá en la columna de la izquierda, debajo del menú de selección que nos muestra todas las bases de datos que tengamos en nuestro servidor, así como también aparece el nombre de la base de datos activa en la ruta superior (*breadcrumb* o migas de pan) que siempre nos indica dónde estamos parados:

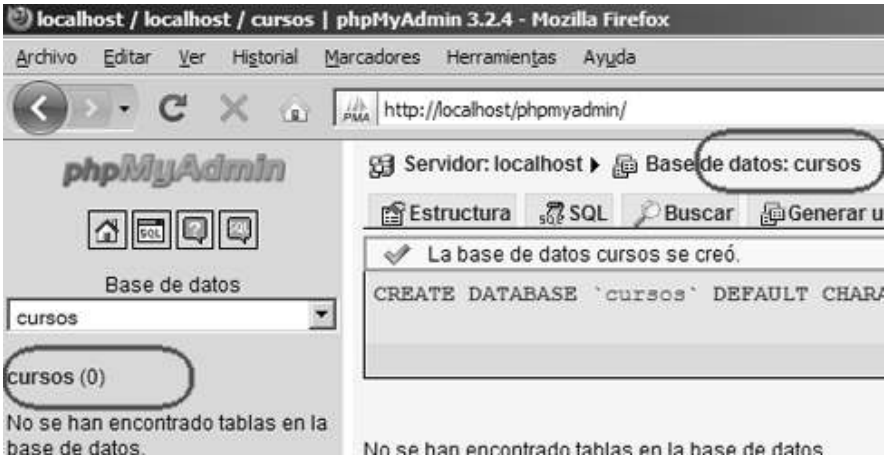


**Figura 11-14. Estamos parados dentro de nuestra nueva base de datos llamada "cursos".**

Seguramente, coincidiremos en que ha sido muy fácil. Ya hemos creado nuestra primera base de datos.

Pero antes de seguir avanzando, comprobemos qué sucedió en el nivel "físico" de nuestro disco rígido al crear esta nueva base. Si hemos usado el XAMPP y lo hemos instalado en la ruta que recomendamos al inicio del libro, entonces podremos entrar con el programa Mi PC (o cualquier otro explorador de archivos), hasta llegar a **C:/servidor/XAMPP/mysql/ data/** y allí, encontraremos **una carpeta por cada base de datos** que hayamos creado; en este caso, vemos, al lado de las bases que vienen por defecto, nuestra nueva base "cursos":

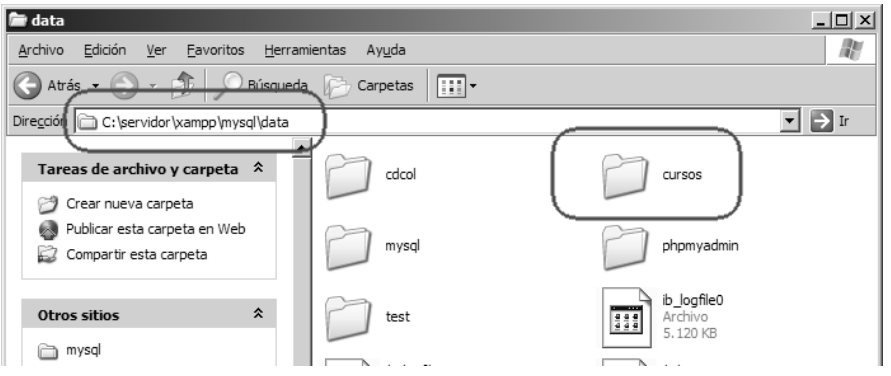


**Figura 11-15. Los archivos físicos de nuestra nueva base de datos.**

## Crear una tabla

En este punto, ya estamos listos para crear nuestra primera **tabla** dentro de nuestra flamante base de datos (recordemos que una base de datos es una simple "carpeta" que organiza nuestras tablas, pero los lugares donde se almacenan realmente los datos son las **tablas**).

Para ello, primero haremos un clic en la columna izquierda, sobre **el nombre de la base** dentro de la cual queremos crear una tabla (nuestra base llamada "cursos" aún no tiene ninguna tabla creada). Esto recargará la parte derecha de la pantalla y veremos un mensaje avisando que todavía no hay tablas en esa base:

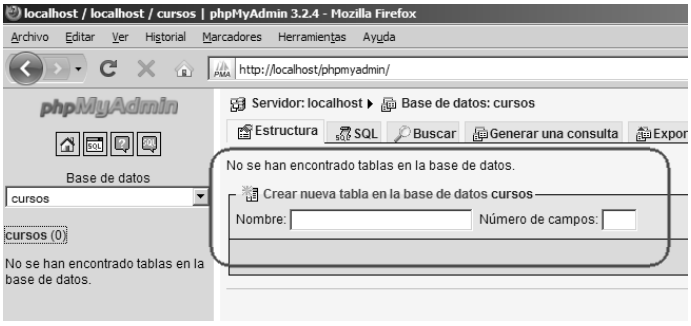


**Figura 11-16. Zona de creación de una nueva tabla.**

Ahora podremos crear una tabla muy fácilmente en la base de datos, simplemente escribiendo el **nombre de la tabla** que crearemos y la **cantidad de campos** (columnas) que deseamos que posea. Esto lo indicaremos en los dos campos de formulario remarcados en la figura anterior.

Como primer ejercicio, crearemos una **tabla** llamada "mensajes" cuyo fin será almacenar el **nombre**, el **correo electrónico** y un **mensaje** que irán dejando los usuarios en un típico formulario de consultas de nuestro sitio Web.

Es decir, a primera vista, parecería que la nueva tabla solo necesitaría tener tres campos, uno para cada dato que almacenará (nombre, correo y mensaje); pero en las bases de datos **siempre** se utiliza un campo extra, cuyo valor debe ser **único** en cada registro, siempre diferente, constituyéndose en un código que permitirá identificar cada registro de forma inequívoca, irrepetible. A este campo extra se lo suele denominar **id** (ya que cumple la función de **identificador** de cada registro), por lo cual tendremos cuatro campos: **id**, **nombre**, **email** y **mensaje**:

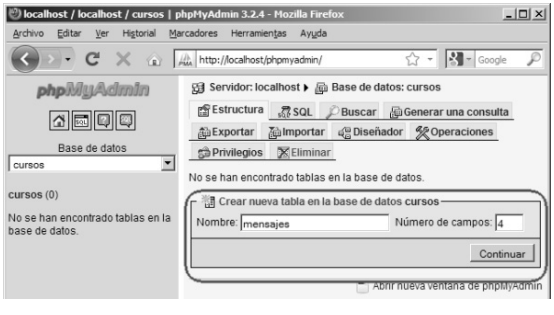


**Figura 11-17. Dando nombre y cantidad de campos a una nueva tabla.**

Luego de pulsar el botón **Continuar**, aparecerá la siguiente pantalla, en la que tendremos que escribir los **nombres** de cada uno de los cuatro campos o columnas que tendrá nuestra tabla. Hagámoslo paso a paso:

1. En el primer campo de texto, justo debajo del título que dice "Campo", escribiremos el **nombre de cada campo** (id, nombre, email, mensaje), en ese orden, uno debajo de otro, todos en la primera columna (la destacada dentro del recuadro en la figura anterior).

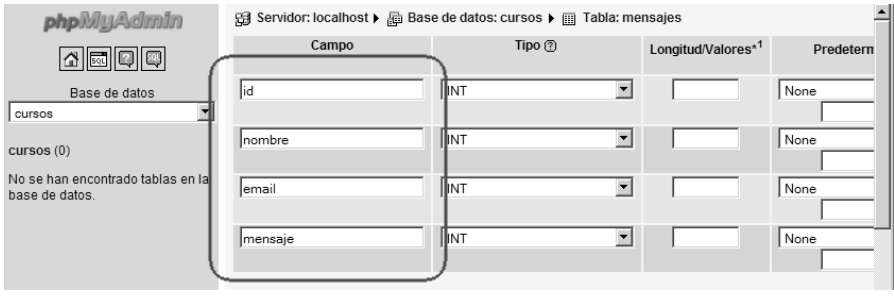


**Figura 11-18. Nombres de campos.**

2. En la segunda columna, denominada **Tipo**, elegiremos el **tipo de dato** que podrá almacenar cada uno de estos campos. Ya veremos muy pronto otros tipos de datos posibles, pero por ahora adelantemos que los tipos de datos normalmente más utilizados son **INT** (*integer*, es decir, números **enteros**, sin decimales, como los que precisa el campo id), **VARCHAR** (*variable character* o caracteres variables, que almacena letras y números, hasta un máximo de 255 caracteres, como los necesarios para los campos nombre y email), y **TEXT** (para textos mayores a 255 caracteres, como los de nuestro campo mensaje). Así que elegiremos estos tipos de dato en la columna "Tipo":

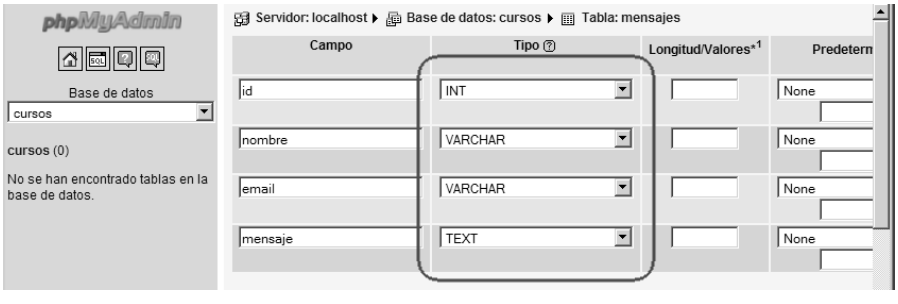


**Figura 11-19. Tipos de datos.**

3. En la tercera columna, definiremos **la cantidad máxima de caracteres** que almacenará cada campo (cuatro dígitos para el id

–suponemos que no tendremos nunca más de 9999 mensajes–), 60 dígitos para cada "nombre" y cada "email", y agregaremos que en los campos de tipo TEXT como "mensaje" no deberemos poner nada en **Longitud**, ya que ésta debe quedar vacía.

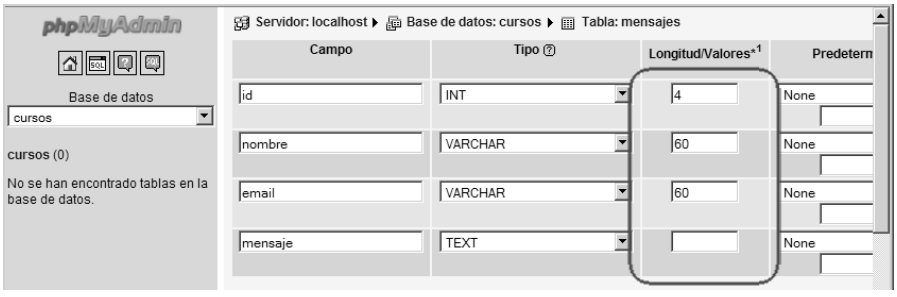


**Figura 11-20. Longitud de cada campo.**

4. Ahora, nos desplazaremos hacia la derecha de la pantalla. En la columna **Nulo**, si dejamos sin marcar la casilla de selección, haremos que ese campo sea NOT NULL; es decir, será **obligatorio** que le completemos algún valor cuando agreguemos un registro. Si no queremos que esto sea obligatorio y que se pueda dejar vacío y que se añada igual el registro completo con el resto de campos que sí se hubieran completado, entonces marcaremos esa casilla de selección, lo que equivale a definir ese campo como potencialmente NULL, es decir, que pueda ser nulo o vacío. Por ahora, determinamos todos los campos como NOT NULL, es decir, no se permitirá valores nulos (vacíos) en ninguno de los campos cuando pretendamos insertar un nuevo registro.

   Para eso, no tenemos nada que hacer, ya que por defecto las casillas están desmarcadas:

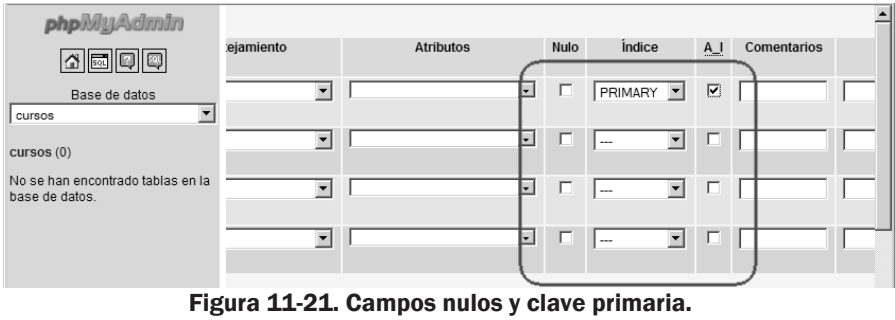


**Figura 11-21. Campos nulos y clave primaria.**

5. Ahora, exclusivamente en el renglón perteneciente al campo id (el primero) deberemos elegir en la columna **Indice** la opción **Primary**, tal como vemos en la imagen anterior, lo que indica que ese

campo será el que identificará cada registro de forma única será su **clave primaria**.

Además, al lado del menú de selección, marcaremos la casilla con la abreviatura **A_I** (Auto Increment), que hace que el contenido o "valor" de este campo id, sea completado **automáticamente** cada vez que agreguemos un registro, con números que se irán incrementando de uno en uno, sin repetirse nunca. No nos preocupemos por ahora si no logramos captar la totalidad de estos detalles, volveremos sobre ellos en próximos ejemplos.

6. Ahora, pulsamos el botón **Grabar**. ¡Atención!: no lo confundamos con el botón **Continuar** que, de forma totalmente confusa, han colocado justo al lado de **Grabar**:

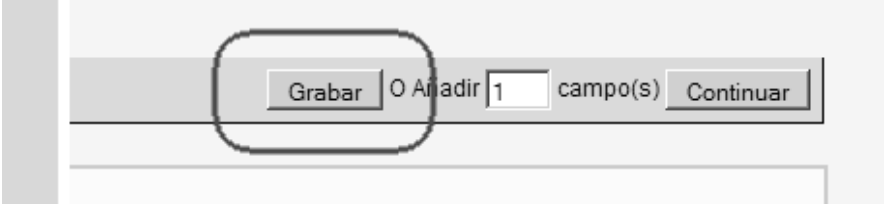


**Figura 11-22. Pulsamos el botón Grabar.**

Y una vez hecho esto, ¡ya tenemos nuestra primera tabla completamente lista para usar! A continuación, realizaremos con ella los procesos más necesarios en una base de datos: agregar, modificar y borrar datos.

# Proceso de altas, bajas y modificaciones

En el ejemplo anterior, hemos creado una tabla. Esta tarea de creación o definición de la estructura de una tabla de datos siempre la realizaremos mediante una **interfaz visual** (como el phpMyAdmin o similares), ya que es mucho más sencillo hacerlo de esta forma que mediante código SQL escrito a mano.

Por el contrario, las tareas de **manipulación de datos** dentro de la tabla ya creada, las haremos desde páginas PHP creadas a tal efecto: crearemos un *backend* o panel de administración, que consistirá en una serie de páginas protegidas con usuario y contraseña; páginas que serán usadas por el dueño o administrador del sitio, donde esta persona autorizada usará formularios que, una vez completados, agregarán registros a alguna de nuestras tablas, podrá modificar el valor de algún campo de determinado registro y, también, podrá borrarlos.

Pero, hasta tanto no aprendamos lo necesario como para hacer esas tareas mediante código del lenguaje SQL (algo que haremos muy pronto), utilizaremos provisoriamente el phpMyAdmin y realizaremos "visualmente" las principales tareas de **administración del contenido** de una tabla, que son:

1. **Agregar** nuevos registros.
2. **Borrar** registros.
3. **Modificar** registros.
4. **Ver un listado** de los registros.

A estas operaciones de manipulación de los datos de una tabla se las conoce como *proceso de* **ABM** (sigla de "Altas", "Bajas" y "Modificaciones"). Algunos le agregan a esta sigla una última letra **L** o **C** (la inicial de "Listados" o "Consultas", respectivamente), denominándolo ABML o ABMC.

También, agregaremos otra tarea: realizar una **copia de seguridad** (*backup*) y restaurarla. Comenzaremos entrando al phpMyAdmin y seleccionando la base **cursos:**

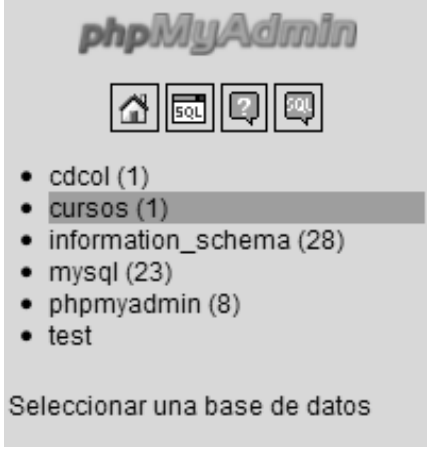


**Figura 11-23. Seleccionamos la base con la que trabajaremos.**

Luego, una vez que estamos trabajando con la base **cursos**, veremos en el menú de la columna izquierda un listado con las **tablas** que contiene esta base (por ahora hay una sola, llamada **mensajes**), y pulsaremos su nombre:


**Figura 11-24. Seleccionamos la tabla con la que trabajaremos.**

Cuando pulsemos el nombre de nuestra tabla **mensajes**, veremos la "estructura" de la tabla (sus campos, tipos de datos, etc.), pero no su "contenido".

Incluso, podremos darnos cuenta de que no hay todavía registros con datos dentro de la tabla, porque aparecerá un ícono de "prohibido" en nuestro cursor cuando pasemos por sobre el enlace de la palabra **Examinar** (al estar vacía la tabla, no hay nada que podamos examinar):

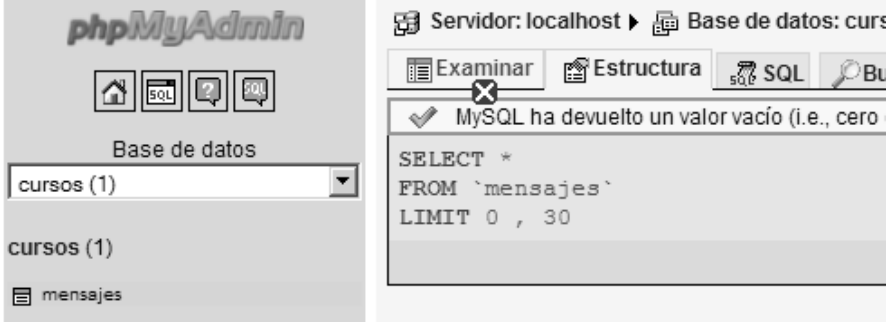


**Figura 11-25. No podemos examinar una tabla vacía.**

Entonces, llegó el momento de insertar datos dentro de nuestra tabla.

## Dar de alta un registro

Para agregar un primer registro con datos dentro de los campos de nuestra tabla, en el enlace **Insertar** pulsaremos:

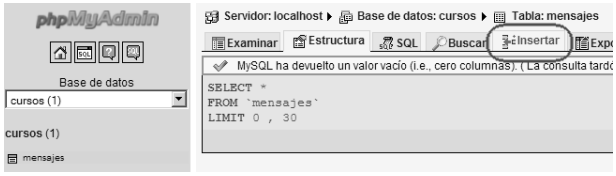


**Figura 11-26. Pulsamos el botón de Insertar un registro.**

Al hacerlo, aparecerá un formulario desde el que agregaremos uno o dos registros (el segundo es optativo) y, en este caso, añadiremos uno solo:

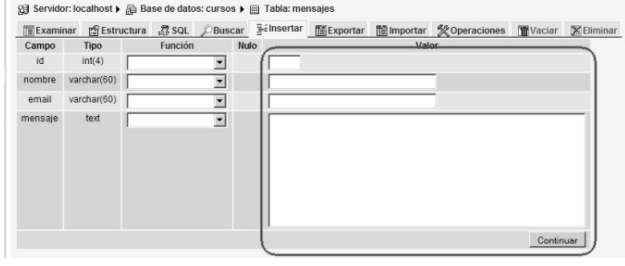


**Figura 11-27. Formulario para agregar los datos de un registro nuevo.**

Completaremos los datos en la zona derecha del formulario, la que está envuelta en un recuadro en la imagen anterior. Al campo **id** no será necesario ingresarle ningún valor, si bien en la imagen introducimos un 1, perfectamente podríamos dejarlo vacío, ya que el propio programa gestor de MySQL le asignará un valor único, sucesivo, automáticamente (al valor más alto existente del campo **id** le sumará 1, y ése será el valor que le pondrá al campo **id** del nuevo registro). Esto se debe a que hemos definido este campo como "A_I" (auto-increment).

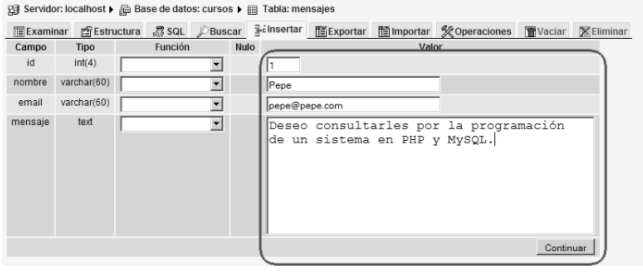


**Figura 11-28. Completamos los datos de un registro.**

Una vez que pulsemos en **Continuar**, veremos un mensaje de que se logró insertar la fila, y ya estará habilitado el enlace de la palabra **Examinar**:

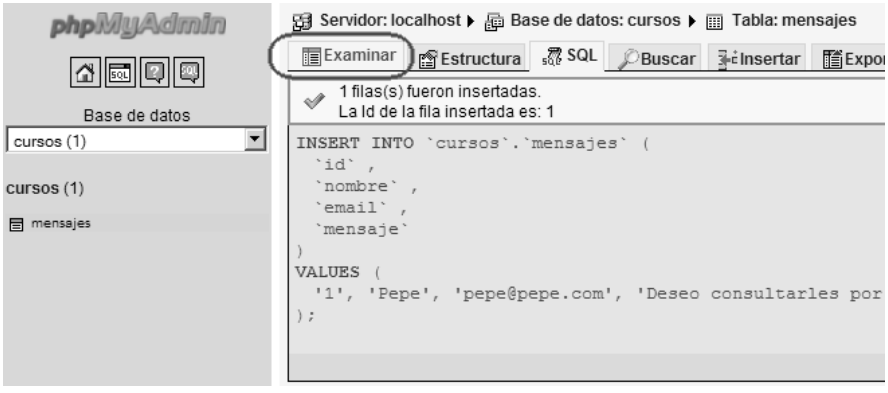


**Figura 11-29. Se habilita el botón de Examinar.**

## Ver un listado con datos

Pulsaremos, entonces, el enlace de **Examinar** para ver el contenido los datos de nuestra tabla:

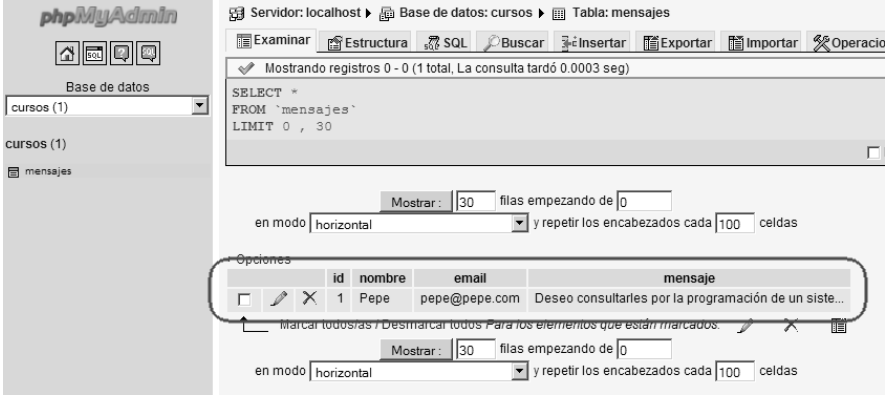


**Figura 11-30. Contenido del primer registro de nuestra tabla.**

En el extremo izquierdo del registro (antes del valor 1 perteneciente al campo id), observaremos dos botones, un **lápiz** y una **equis**. El primero permite **editar** (modificar) ese registro; la segunda, **eliminarlo**.

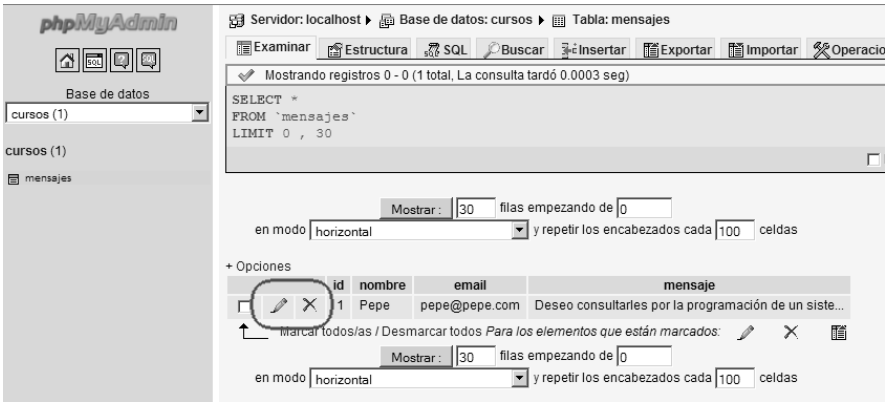


**Figura 11-31. Botones para modificar o eliminar un registro.**

## Modificar un registro

Si queremos modificar el valor de alguno de los campos de un registro (debido a un cambio de precio en un producto, por ejemplo), pulsaremos el **lápiz** correspondiente al registro (renglón) que queremos editar, y aparecerá un formulario con los valores de cada campo de este registro ya escritos; en ese momento, escribiremos o borraremos lo que queramos, pulsando, finalmente, en **Continuar** para actualizar el registro definitivamente.

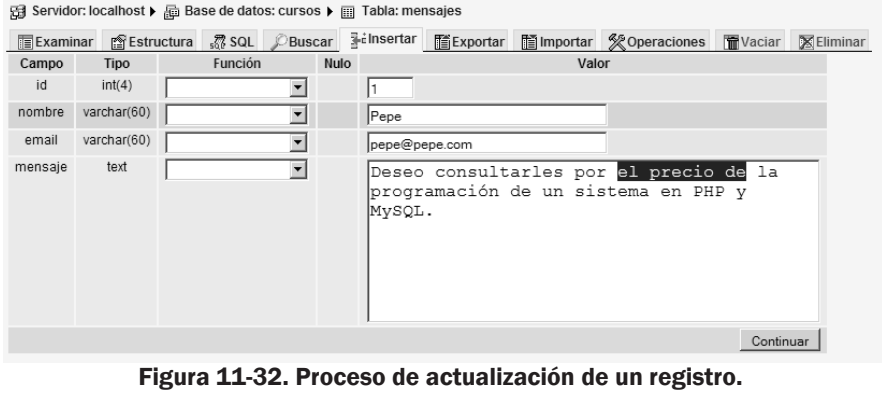


**Figura 11-32. Proceso de actualización de un registro.**

## Borrar un registro

Antes de aprender a borrar un registro, vamos a agregar al menos un registro más al que ya teníamos, para que nuestra tabla no quede vacía luego de eliminar el –

hasta ahora– único registro que posee. Una vez que añadamos algún registro, ya podemos borrar uno, simplemente pulsando la **equis** ubicada en el renglón del registro que queremos eliminar.


**Figura 11-33. Pulsamos la equis a la altura del registro que queremos borrar.**

Nos aparecerá una advertencia que pedirá nuestra confirmación:

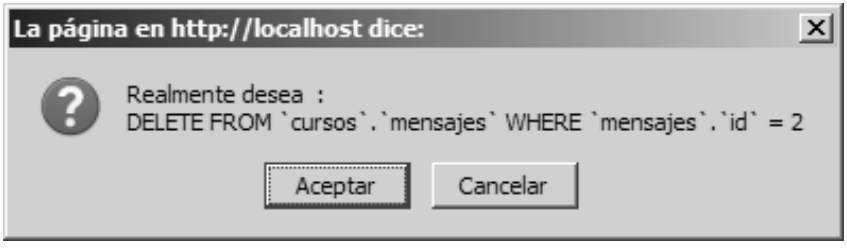


**Figura 11-34. Nos pide confirmación, y pulsamos Aceptar.**

Una vez que pulsamos en **Aceptar**, el registro elegido desaparecerá **definitivamente** de nuestra tabla (usar con precaución).

## Copia de seguridad: cómo exportar una tabla y volver a importarla

Una tarea muy necesaria cuando trabajamos con bases de datos, es la posibilidad de llevarnos tanto la estructura como los datos almacenados en una tabla hacia otra computadora (hacia un *hosting*, o para hacer un *backup*).

Para ello, pulsaremos en el enlace de **Exportar**:

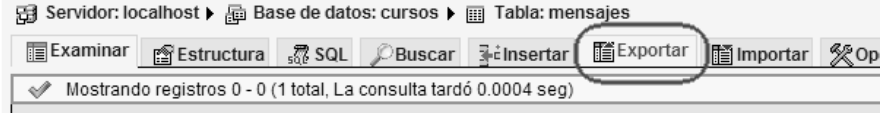


**Figura 11-35. Pulsamos Exportar.**

y luego marcaremos las opciones **SQL**, **Estructura**, **Datos** y **Enviar** (Fig. 11-36).

Estas opciones nos abrirán un cuadro de diálogo para que podamos guardar en nuestro disco rígido un archivo de texto con el nombre de la tabla que estamos exportando, con la extensión **.sql**. Por ejemplo, **mensajes.sql**. Este archivo contiene todas las órdenes escritas en lenguaje SQL, necesarias para volver a crear en otro servidor una tabla idéntica a la que hemos exportado.

A este mismo archivo podemos "Importarlo" en otro phpMyAdmin de un *hosting*, o en caso de usarlo como copia de seguridad luego de algún borrado de datos, en este mismo servidor, simplemente yendo al menú **Importar**, y eligiendo este archivo **mensajes.sql**, que hemos dejado a salvo como copia de respaldo. Esto reconstruirá la tabla con todos sus campos y con idénticos tipos de datos, y cargará dentro de ella todos los datos (registros) que hubiera tenido almacenados la tabla original en el momento en el que la exportamos (Fig. 11-37).

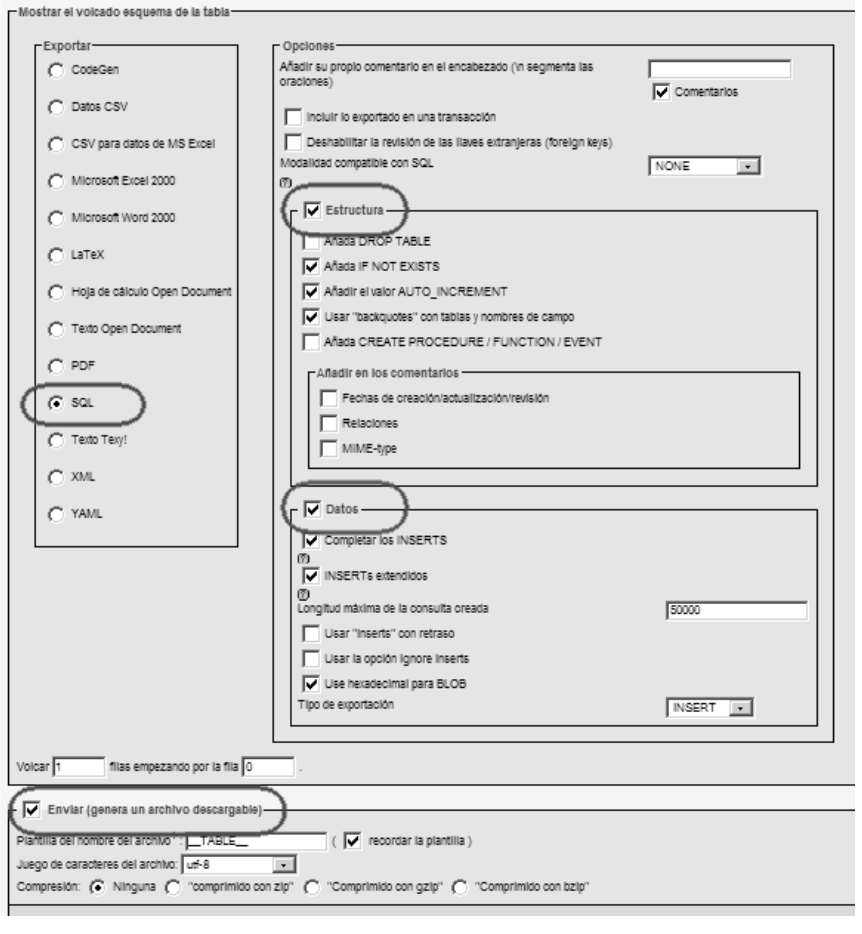


**Figura 11-36. Exportaremos código SQL, tanto la estructura de la tabla como sus datos almacenados y lo descargaremos.**

A continuación, antes de pasar a programar nuestras primeras páginas PHP que lean datos alojados en una base de datos, vamos a profundizar un poco más en los **tipos de datos** que podemos definir en cada campo de nuestras tablas, y analizaremos la conveniencia de utilizar **atributos** para controlar con mayor precisión cuál será el contenido de nuestros campos.

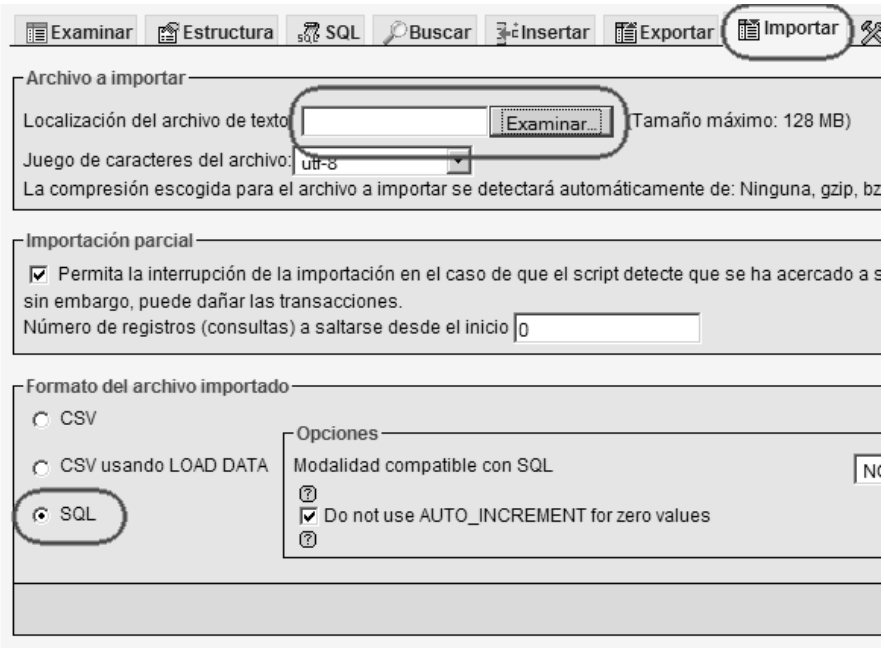


**Figura 11-37. Cómo importar una copia de seguridad.**

# Los tipos de datos más usados

Toda vez que tengamos que crear una tabla que sirva para almacenar datos de una aplicación Web, deberemos poner a prueba nuestra capacidad para definir los tipos de datos que con mayor eficiencia puedan almacenar cada dato que necesitemos guardar.

Los campos de las tablas MySQL nos dan la posibilidad de elegir entre tres grandes tipos de contenidos: datos **numéricos**, datos para guardar cadenas de caracteres **(alfanuméricos)** y datos para almacenar fechas y horas.

Desde ya que es muy obvio poder distinguir a cuál de los tres grupos corresponderá, por ejemplo, un campo que guarde la "edad" de una persona: será un dato numérico. Pero, dentro de los distintos tipos de datos numéricos, ¿cuál será la mejor opción? Un número entero, pero, ¿cuál de los distintos tipos de enteros disponibles? ¿Qué tipo de dato permitirá consumir menor espacio

físico de almacenamiento y brindará la posibilidad de almacenar la cantidad de datos que se espera almacenar en ese campo? (dos dígitos, o máximo tres, en el caso de la edad).

Esas son preguntas que solo podremos responder a partir del conocimiento de los distintos **tipos de datos** que nos permite definir MySQL. Vamos, entonces, a analizar los usos más apropiados de cada uno de estos tres grandes grupos.

## Datos numéricos

La diferencia entre uno y otro tipo de dato es simplemente el **rango de valores** que puede contener. Dentro de los datos numéricos, podemos distinguir dos grandes ramas: **enteros** y **decimales**.

### Numéricos enteros

Comencemos por conocer las opciones que tenemos para almacenar datos que sean números **enteros** (edades, cantidades, magnitudes sin decimales); poseemos una variedad de opciones:

| Tipo de dato | Bytes | Valor mínimo | Valor máximo |
|---|---|---|---|
| TINYINT | 1 | -128 | 127 |
| SMALLINT | 2 | -32768 | 32767 |
| MEDIUMINT | 3 | -8388608 | 8388607 |
| INT o INTEGER | 4 | -2147483648 | 2147483647 |
| BIGINT | 8 | -9223372036854775808 | 9223372036854775807 |

**Tabla 11-1. Tipos de dato numéricos enteros.**

Veamos un ejemplo para comprender mejor qué tipo de dato nos conviene elegir para cada campo. Si necesitamos definir un campo para almacenar la "edad" de nuestros usuarios, sería suficiente con asignar a ese campo un tipo de dato TINYINT, que permite almacenar como máximo el valor 127 (es decir, por más que tenga tres dígitos, no nos dejará almacenar un 999, ni siquiera un 128, solo un número hasta el número 127 inclusive). Como la edad posible de las personas queda incluida en ese rango, es suficiente un TINYINT.

Ahora bien, si queremos definir el tipo de dato para el campo id (identificador) de la tabla de productos de un gran mercado que vende varios miles de artículos diferentes, ya no será suficiente con un TINYINT, y deberemos conocer con mayor precisión la cantidad de artículos distintos que comercializa

(en la actualidad y la cantidad prevista en un futuro próximo, para que nuestro sistema de almacenamiento no quede obsoleto rápidamente). Suponiendo que ese mercado venda 10.000 artículos distintos, el tipo de dato adecuado para el campo id será SMALLINT, ya que nos permitirá numerar hasta algo más de 32.000 artículos (tal como vimos en el cuadro anterior).

En el supuesto de que el campo id deba utilizarse para una tabla de clientes de una empresa telefónica con 5 millones de usuarios, ya no nos serviría un SMALLINT, sino que deberíamos utilizar un MEDIUMINT.

En el caso de que esa empresa tuviera 200 millones de clientes, deberíamos utilizar un campo de tipo INT.

Asimismo, si quisiéramos definir un campo que identifique a cada uno de los seres humanos que habitamos en el planeta, deberemos recurrir a un campo BIGINT, ya que el tipo INT solo permite hasta 2 mil millones de datos diferentes, lo cual no nos alcanzaría.

Vemos, entonces, que hay que considerar siempre cuál será el valor máximo que se almacenará dentro de un campo, antes de elegir el tipo de dato más adecuado.

Pero no solo los valores máximos deben considerarse, también debemos tener en cuenta los **valores mínimos** que pudieran almacenarse. Por ejemplo, para guardar el puntaje de un juego, que pudiera tomar valores negativos, o el saldo de una cuenta corriente, o una tabla que incluya valores de temperatura bajo cero, y casos similares. En el cuadro anterior, hemos visto que cada tipo de dato posee valores mínimos negativos simétricos a los valores positivos máximos que podía almacenar.

### Valores sin signo

Ahora bien: existe la posibilidad de **duplicar** el límite de valor máximo positivo de cada tipo de dato, si eliminamos la posibilidad de almacenar valores negativos. Pensemos en los ejemplos anteriores: la edad no tiene sentido que sea negativa, entonces, si eliminamos la posibilidad de que ese campo almacene valores negativos, duplicaríamos el límite positivo de almacenamiento, y el campo de tipo TINYINT que normalmente permitía almacenar valores del -128 al 127, ahora dejará almacenar valores desde el 0 hasta el 255. Esto puede ser útil para almacenar precios, cantidades de objetos o magnitudes que no puedan ser negativas, etc.

Observemos en la tabla cómo se duplican los valores máximos **sin signo**, y luego aprenderemos cómo configurar esta posibilidad de quitar el signo desde el phpMyAdmin:

| Tipo de dato | Bytes | Valor mínimo | Valor máximo |
|---|---|---|---|
| TINYINT | 1 | 0 | 255 |
| SMALLINT | 2 | 0 | 65535 |
| MEDIUMINT | 3 | 0 | 16777215 |
| INT o INTEGER | 4 | 0 | 4294967295 |
| BIGINT | 8 | 0 | 18446744073709551615 |

**Tabla 11-2. Tipos de dato numéricos enteros sin signo.**

¿Cómo definimos que un campo no tiene signo? Mediante el modificador UNSIGNED que podemos definirle a un campo numérico:


**Figura 11-38. Cómo definir un campo como unsigned.**

Elegimos en la columna **Atributos** el valor de UNSIGNED y este campo ya no podrá contener valores negativos, duplicando su capacidad de almacenamiento (en el caso de ser de tipo entero, pronto veremos que en los tipos de coma flotante, si se lo define como UNSIGNED no altera el valor máximo permitido).

Comentemos al pasar, que es importante que, en el momento de definir un campo, en la columna **Longitud** escribamos un número coherente con la capacidad de almacenamiento que acabamos de elegir. Por ejemplo, en un TINYINT para la edad, colocaremos como longitud un tres, y no un número mayor ni menor.

### Numéricos con decimales

Dejemos los enteros y pasemos ahora a analizar los valores numéricos **con decimales.** Estos tipos de dato son necesarios para almacenar precios, salarios, importes de cuentas bancarias, etc. que no son enteros. Tenemos que tener en cuenta que si bien estos tipos de datos se llaman “de coma flotante”, por ser la

coma el separador entre la parte entera y la parte decimal, en realidad, MySQL los almacena usando un **punto** como separador.

En esta categoría, disponemos de tres tipos de datos: FLOAT, DOUBLE y DECIMAL.

La estructura con la que podemos declarar un campo FLOAT implica definir dos valores: la **longitud total** (incluyendo los decimales y la coma), y cuántos de estos dígitos son la **parte decimal**. Por ejemplo:

FLOAT(6,2)

Esta definición permitirá almacenar como mínimo el valor -999.99 y como máximo 999.99 (el signo menos no cuenta, pero el punto decimal sí, por eso son seis dígitos en total, y de ellos dos son los decimales).

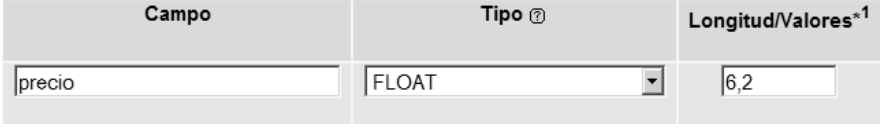


**Figura 11-39. Las partes de un campo FLOAT.**

La cantidad de decimales (el segundo número entre los paréntesis) debe estar entre 0 y 24, ya que ése es el rango de precisión simple.

En cambio, en el tipo de dato DOUBLE, al ser de doble precisión, solo permite que la cantidad de decimales se defina entre 25 y 53.

Debido a que los cálculos entre campos en MySQL se realizan con doble precisión (la utilizada por DOUBLE), usar FLOAT, que es de simple precisión, puede traer problemas de redondeo y pérdida de los decimales restantes.

Por último, DECIMAL es ideal para almacenar valores monetarios, donde se requiera menor longitud, pero la **máxima exactitud** (sin redondeos). Este tipo de dato le asigna un ancho fijo a la cifra que almacenará. El máximo de dígitos totales para este tipo de dato es de 64, de los cuales 30 es el número de decimales máximo permitido. Más que suficiente para almacenar precios, salarios y monedas.

El formato en el que se definen en el phpMyAdmin es idéntico para los tres: primero la longitud total, luego, una coma y, por último, la cantidad de decimales.

## Datos alfanuméricos

Para almacenar datos alfanuméricos (cadenas de caracteres) en MySQL, poseemos los siguientes tipos de datos: CHAR, VARCHAR, BINARY, VARBINARY,

TINYBLOB, TINYTEXT, BLOB, TEXT, MEDIUMBLOB, MEDIUMTEXT, LONGBLOB, LONGTEXT, ENUM y SET.

Veremos ahora cuáles son sus características y cuáles son las ventajas de usar uno u otro, según qué datos necesitemos almacenar.

## CHAR

Comencemos por el tipo de dato alfanumérico más simple: CHAR (*character*, o carácter). Este tipo de dato permite almacenar textos breves, de hasta 255 caracteres de longitud como máximo. Su característica principal es que consumirá en todos los registros exactamente el ancho máximo en caracteres que le definamos, aunque no lo utilicemos. Por ejemplo, si definiéramos un campo "nombre" de 14 caracteres como CHAR, reservará (y consumirá en disco) este espacio:

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| J | u | a | n | | P | é | r | e | z | | | | |
| C | a | r | l | o | s | | G | a | r | c | í | a | |
| J | o | s | é | | R | a | m | í | r | e | z | | |
| L | u | i | s | | F | e | r | n | á | n | d | e | z |
| P | e | p | e | | L | ó | p | e | z | | | | |

**Figura 11-40. Espacio reservado por CHAR.**

Por lo tanto, no es eficiente cuando la longitud del dato que se almacenará en un campo es desconocida *a priori* (típicamente, datos ingresados por el usuario en un formulario, como su nombre, domicilio, etc.).

¿En qué casos usarlo, entonces? Cuando el contenido de ese campo será completado por nosotros, programadores, al agregarse un registro y, por lo tanto, estamos seguros de que la longitud **siempre será la misma**. Pensemos en un formulario con botones de radio para elegir el "sexo": independientemente de lo que muestren las etiquetas visibles para el usuario, podríamos almacenar un solo carácter **M** o **F** (masculino o femenino) y, en consecuencia, el ancho del campo CHAR podría ser de un dígito, y sería suficiente. Lo mismo sucede con códigos que identifiquen provincias, países, estados civiles, etc.

## VARCHAR

Complementariamente, el tipo de dato VARCHAR (*character varying*, o caracteres variables) es útil cuando la longitud del dato es **desconocida**, cuando depende de la información que el usuario escribe en campos o áreas de texto de un formulario.

La longitud máxima permitida era de 255 caracteres hasta MySQL 5.0.3, pero desde esta versión cambió a un máximo de 65.535 caracteres. Este tipo de dato tiene la particularidad de que cada registro puede tener una **longitud diferente**, que dependerá de su contenido: si en un registro el campo "nombre" (supongamos que hubiera sido definido con un ancho máximo de 20 caracteres) contiene solamente el texto: "Pepe", consumirá solo cinco caracteres: cuatro para las cuatro letras, y uno más que indicará cuántas letras se utilizaron. Si luego, en otro registro, se ingresa un nombre de 15 caracteres, consumirá 16 caracteres (siempre uno más que la longitud del texto, mientras la longitud no supere los 255 caracteres; si los supera, serán dos los bytes necesarios para indicar la longitud).

Por lo tanto, será más eficiente para almacenar registros cuyos valores tengan **longitudes variables**, ya que si bien "gasta" uno o dos caracteres por registro para declarar la longitud, esto le permite ahorrar muchos otros caracteres que no serían utilizados. En cambio, en el caso de datos de longitud, siempre constante, sería un desperdicio gastar un carácter por registro para almacenar la longitud y, por eso, convendría utilizar CHAR en esos casos.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| J | u | a | n | | P | é | r | e | z | | | | |
| C | a | r | l | o | s | | G | a | r | c | í | a | |
| J | o | s | é | | R | a | m | í | r | e | z | | |
| L | u | i | s | | F | e | r | n | á | n | d | e | z |
| P | e | p | e | | L | ó | p | e | z | | | | |

**Figura 11-41. Espacio utilizado por VARCHAR.**

## BINARY y VARBINARY

Estos dos tipos de datos son idénticos a CHAR y VARCHAR, respectivamente, salvo que almacenan *bytes* en lugar de caracteres, una diferencia muy sutil para un nivel básico a intermedio de MySQL como el que pretendemos alcanzar con este libro.

## TEXT

Antes de la versión 5.0.3 de MySQL, este campo era el utilizado "por excelencia" para descripciones de productos, comentarios, textos de noticias, y cualquier otro **texto largo**. Pero, a partir de la posibilidad de utilizar VARCHAR para longitudes de hasta 65.535 caracteres, es de esperar que se utilice cada vez menos este tipo de campo. La principal desventaja de TEXT es que no puede **indexarse** fácilmente (a diferencia de VARCHAR). Muy pronto, veremos qué es un índice y por qué es tan importante que un campo sea fácil de indexar. Tampoco se le puede asignar

un valor **predeterminado** a un campo TEXT (un valor por omisión que se complete automáticamente si no se ha proporcionado un valor al insertar un registro). Solo deberíamos utilizarlo para textos realmente muy largos, como los que mencionamos al comienzo de este párrafo.

### TINYTEXT, MEDIUMTEXT y LONGTEXT

Similares a TEXT, solo varía el tamaño máximo permitido: en TINYTEXT, es de 255 caracteres (por lo cual no tiene mucho sentido utilizarlo, es mucho mejor un VARCHAR), en MEDIUMTEXT, es de 16.777.215 caracteres y, en LONGTEXT, es de cuatro gigas (o lo máximo que permita manipular el sistema operativo, teniendo en cuenta que el tamaño máximo permitido para los paquetes enviados entre MySQL y PHP suele ser de 16 Mb, que es, de todos modos, "inmenso" para un simple texto).

### BLOB

Es un campo que guarda información en formato binario y se utiliza cuando desde PHP se almacena en la base de datos el contenido de un archivo binario (típicamente, una imagen o un archivo comprimido ZIP) leyéndolo byte a byte, y se quiere almacenar todo su contenido para luego reconstruir el archivo y servirlo al navegador otra vez, sin necesidad de almacenar la imagen o el ZIP en un disco, sino que sus bytes quedan guardados en un campo de una tabla de la base de datos. El tamaño máximo que almacena es 65.535 bytes.

De todos modos, y como lo hemos mencionado en este ejemplo, respecto al tipo de dato para una imagen, usualmente no se guarda "la imagen" (sus bytes, el contenido del archivo) en la base de datos porque, en un sitio grande, se vuelve muy pesada y lenta la base de datos; sino que se almacena solo la URL que lleva hasta la imagen. De esa forma, para mostrar la imagen simplemente se lee ese campo URL y se completa una etiqueta **img** con esa URL, y esto es suficiente para que el navegador muestre la imagen. Entonces, con un VARCHAR, alcanza para almacenar la URL de una imagen. El campo BLOB es para almacenar directamente "la imagen" (o un archivo comprimido, o cualquier otro archivo binario), no su ruta.

### TINYBLOB, MEDIUMBLOB y LONGBLOB

Similares a BLOB, solo cambia la longitud máxima: en TINYBLOB es de 255 bytes, en MEDIUMBLOB es de 16.777.215 bytes, y en LONGBLOB es de cuatro Gb (o lo máximo que permita manipular el sistema operativo).

### ENUM

Su nombre es la abreviatura de "enumeración". Este campo nos permite establecer cuáles serán los valores posibles que se le podrán insertar. Es decir, crearemos una **lista de valores** permitidos, y no se autorizará el ingreso de ningún

valor fuera de la lista, y se permitirá elegir **solo uno** de estos datos como valor del campo. Los valores deben estar separados por comas y envueltos entre comillas simples. El máximo de valores diferentes es de 65.535. Lo que se almacenará no es la cadena de caracteres en sí, sino el número de índice de su posición dentro de la enumeración. Por ejemplo, si al crear la tabla definimos un campo de esta manera:

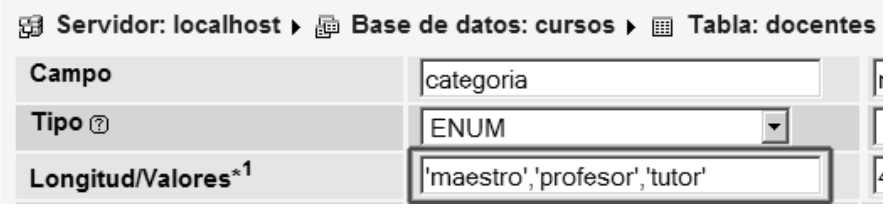


**Figura 11-42. Ingreso de valores de una enumeración.**

En este ejemplo, la categoría será **excluyente**, no podremos elegir más de una, y todo docente (uno por registro) deberá tener una categoría asignada.

### SET

Su nombre significa "conjunto". De la misma manera que ENUM, debemos especificar una lista, pero de hasta 64 opciones solamente. La carga de esos valores es idéntica a la de ENUM, una lista de valores entre comillas simples, separados por comas. Pero, a diferencia de ENUM, sí podemos llegar a dejarlo vacío, sin elegir **ninguna** opción de las posibles. Y también podemos elegir como valor del campo **más de uno** de los valores de la lista. Por ejemplo, damos a elegir una serie de temas (típicamente con casillas de verificación que permiten selección múltiple) y luego almacenamos en un solo campo todas las opciones elegidas.

Un detalle importante es que cada valor dentro de la cadena de caracteres no puede contener comas, ya que es la coma el separador entre un valor y otro.

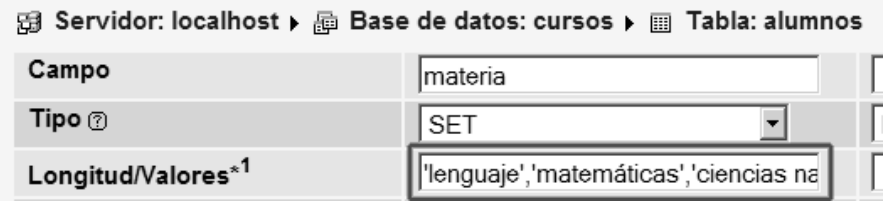


**Figura 11-43. Definiendo valores múltiples dentro de un conjunto SET.**

Una vez definido ese tipo de dato, podemos cargar valores **múltiples** para ese campo dentro de un mismo registro, pulsando **Control** mientras hacemos clic

en cada opción. Eso significará, en este ejemplo de una tabla de alumnos, que ese alumno está cursando **ambas** materias seleccionadas.

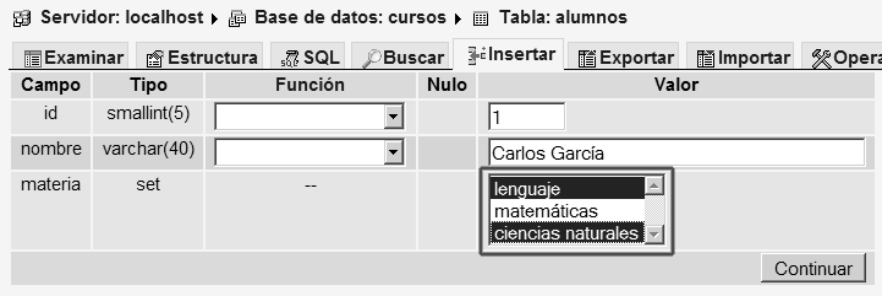


**Figura 11-44. Asignando valores múltiples pulsando Control.**

Si una vez agregado algún registro pulsamos en **Examinar** para ver el contenido de la tabla, veremos que este registro, que en un campo contenía un valor múltiple, incluye lo siguiente:


**Figura 11-45. Valores múltiples.**

Con esto, damos por finalizado el grupo de tipos de datos alfanuméricos. Pasemos ahora al último grupo, el de fechas y horas.

## Datos de fecha y hora

En MySQL, poseemos varias opciones para almacenar datos referidos a fechas y horas.

Veamos las diferencias entre uno y otro, y sus usos principales, así podemos elegir el tipo de dato apropiado en cada caso.

### DATE

El tipo de dato DATE nos permite almacenar **fechas** en el formato: AAAA-MM-DD (los cuatro primeros dígitos para el año, los dos siguientes para el mes y los últimos dos para el día). Atención: en los países de habla hispana estamos acostumbrados a ordenar las fechas en Día, Mes y Año, pero para MySQL es exactamente al revés. Tengamos en cuenta que esto nos obligará a realizar

algunas maniobras de reordenamiento utilizando funciones de manejo de caracteres.

Si bien al leer un campo DATE siempre nos entrega los datos separados por guiones, al momento de **insertar** un dato nos permite hacerlo tanto en formato de número continuo (por ejemplo, 20151231), como utilizando cualquier carácter separador (2015-12-31 o cualquier otro carácter que separe los tres grupos).

El rango de fechas que permite manejar va desde el 1000-01-01 hasta el 9999-12-31. Es decir, que no nos será útil si trabajamos con una línea de tiempo que se remonte antes del año 1000 (¿alguna aplicación relacionada con la historia?), pero sí nos resultará útil para datos de un pasado cercano y un futuro muy largo por delante, ya que llega casi hasta el año 10.000.

### DATETIME

Un campo definido como DATETIME nos permitirá almacenar información acerca de un instante de tiempo, pero no solo la fecha sino también su horario, en el formato: AAAA-MM-DD HH:MM:SS, siendo la parte de la fecha de un rango similar al del tipo DATE (desde el 1000-01-01 00:00:00 al 9999-12-31 23:59:59), y la parte del horario, de 00:00:00 a 23:59:59.

### TIME

Este tipo de campo permite almacenar horas, minutos y segundos, en el formato HH:MM:SS, y su rango permitido va desde -839:59:59 hasta 839:59:59 (unos 35 días hacia atrás y hacia adelante de la fecha actual). Esto lo hace ideal para calcular tiempos transcurridos entre dos momentos cercanos.

### TIMESTAMP

Un campo que tenga definido el tipo de dato TIMESTAMP sirve para almacenar una fecha y un horario, de manera similar a DATETIME, pero su formato y rango de valores serán diferentes.

El formato de un campo TIMESTAMP puede variar entre tres opciones:

- AAAA-MM-DD HH:MM:SS
- AAAA-MM-DD
- AA-MM-DD

Es decir, la longitud posible puede ser de 14, 8 o 6 dígitos, según qué información proporcionemos.

El rango de fechas que maneja este campo va desde el 1970-01-01 hasta el año 2037.

Además, posee la particularidad de que podemos definir que su valor **se mantenga actualizado** automáticamente, cada vez que se **inserte** o que se **actualice** un registro. De esa manera, conservaremos siempre en ese campo la fecha y hora de la última actualización de ese dato, que es ideal para llevar el control sin necesidad de programar nada. Para definir esto desde el phpMyAdmin, deberemos seleccionar en **Atributos** la opción on update CURRENT_TIMESTAMP, y como valor predeterminado, **CURRENT_TIMESTAMP**:

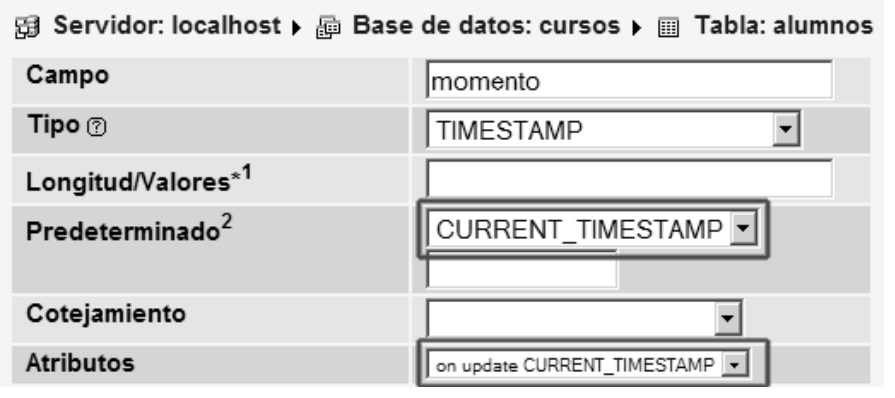


**Figura 11-46. Campos cuyo valor se actualizará automáticamente al insertar o modificar un registro.**

### YEAR

En caso de definir un campo como YEAR, podremos almacenar un año, tanto utilizando dos como cuatro dígitos. En caso de hacerlo en dos dígitos, el rango posible se extenderá desde 70 hasta 69 (del 70 hasta el 99 se entenderá que corresponden al rango de años entre 1970 y 1999, y del 00 al 69 se entenderá que se refieren a los años 2000 a 2069); en caso de proporcionar los cuatro dígitos, el rango posible se ampliará, yendo desde 1901 hasta 2155.

Una posibilidad extra, ajena a MySQL pero relativa a las fechas y horarios, es generar un valor de *timestamp* con la función **time** de PHP (repito, no estamos hablando de MySQL, no nos confundamos a causa de tantos nombres similares). A ese valor, lo podemos almacenar en un campo INT de 10 dígitos. De esa forma, será muy simple **ordenar** los valores de ese campo (supongamos que es la fecha de una noticia) y luego podremos mostrar la fecha transformando ese valor de *timestamp* en algo legible mediante funciones de manejo de fecha propias de PHP.

# Atributos de los campos

Ya hemos visto los diferentes tipos de datos que es posible utilizar al definir un campo en una tabla, pero estos tipos de datos pueden poseer ciertos modificadores o "atributos" que se pueden especificar al crear el campo, y que nos brindan la posibilidad de controlar con mayor exactitud qué se podrá almacenar en ese campo, cómo lo almacenaremos y otros detalles.

Aunque algunos de estos atributos ya los hemos utilizado intuitivamente al pasar en algunos de los ejemplos anteriores, a continuación vamos a analizarlos más en detalle.

## ¿Null o Not Null?

Algunas veces tendremos la necesidad de tener que agregar registros sin que los valores de todos sus campos sean completados, es decir, dejando algunos campos vacíos (al menos provisoriamente).

Por ejemplo, en un sistema de comercio electrónico, podría ser que el precio, o la descripción completa de un producto, o la cantidad de unidades en depósito, o la imagen del producto, no estén disponibles en el momento en que, como programadores, comencemos a trabajar con la base de datos. Todos esos campos, nos conviene que sean definidos como **NULL** (nulos), para que podamos ir agregando registros con los datos básicos de los productos (su nombre, código, etc.) aunque todavía la gente del área comercial no haya definido el **precio**, ni el área de *marketing* haya terminado las **descripciones**, ni los diseñadores hayan subido las **fotos** (es típica esta división de tareas en empresas grandes, y hay que tenerla presente, porque afecta la declaración de campos de nuestras tablas).

Si definimos esos campos que no son imprescindibles de llenar de entrada como NULL (simplemente marcando la **casilla de verificación** a la altura de la columna NULL, en el phpMyAdmin), el campo queda preparado para que, si no es proporcionado un valor, quede vacío pero igual nos permita completar la inserción de un registro completo.

Por omisión, si no marcamos ninguna casilla, todos los campos son NOT NULL, es decir, es obligatorio ingresar algún valor en cada campo para poder cargar un nuevo registro en la tabla.

## Valor predeterminado (default)

Muchas veces necesitaremos agilizar la carga de datos mediante un **valor por defecto** *(default).* Por ejemplo, pensemos en un sistema de pedidos, donde, al

llegar el pedido a la base de datos, su estado sea "recibido", sin necesidad de que el sistema envíe ningún valor, solo por agregar el registro, ese registro debería contener en el campo "estado" el valor de "recibido". Este es un típico caso de valor predeterminado o por *default.*

En phpMyAdmin, podemos especificar que un campo tenga un valor predeterminado de tres maneras posibles:

1. Escribiendo nosotros a mano el valor (como en el caso de "recibido"), en cuyo caso debemos elegir la primera de las opciones de la columna "Predeterminado", la que dice "Como fuera definido:", y debemos escribir el valor en el campo de texto existente justo debajo de ese menú:


**Figura 11-47. Ingresando un valor predeterminado.**

2. Podemos definir que el valor por omisión sea NULL, es decir, que si no se proporciona un valor, quede el valor NULL como valor de ese campo.
3. Y, por último, podemos definir para un campo de tipo TIMESTAMP que se inserte como valor por defecto el valor actual de TIMESTAMP *(Current_Timestamp),* algo que ya hemos visto en detalle al referirnos a este tipo de dato.

En los dos últimos casos, debemos dejar vacío el cuadro de texto inferior.

Es importante señalar que no se puede dar un valor predeterminado a un campo de tipo TEXT ni BLOB (y todas sus variantes).

## Ordenamiento binario

Definir un campo de texto CHAR o VARCHAR como BINARY (binario) solo afecta en el **ordenamiento** de los datos: en lugar de ser indiferentes a mayúsculas y minúsculas, un campo BINARY se ordenará teniendo en cuenta esta diferencia, por lo cual, a igualdad de letra, primero aparecerán los datos que contengan esa letra en mayúsculas.

Se define desde el phpMyAdmin eligiendo BINARY en el menú **Atributos**:

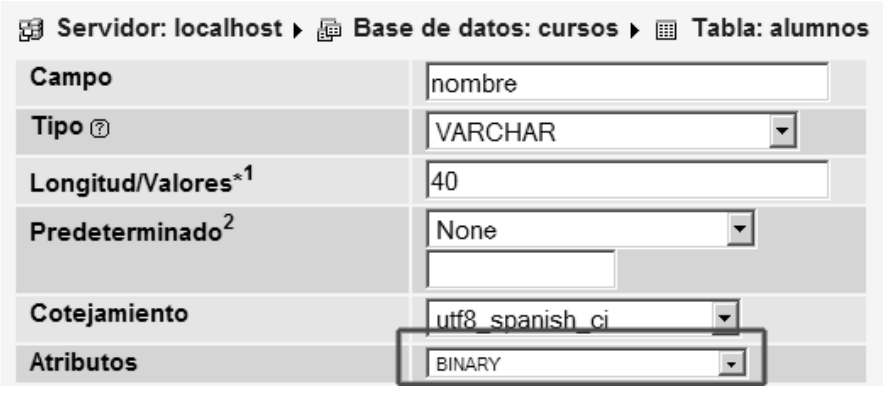


**Figura 11-48. Definiendo un campo como BINARY.**

## Índices

El objetivo de crear un **índice** es mantener ordenados los registros por aquellos campos que sean frecuentemente utilizados en búsquedas, para así agilizar los tiempos de respuesta. Un índice no es más que una tabla "paralela", que guarda los mismos datos que la tabla original, pero en lugar de estar ordenados por orden de inserción o por la clave primaria de la tabla, en el índice se **ordenan por el campo** que elegimos indexar.

Por ejemplo, si hubiera un buscador por **título** de noticia, en la tabla de noticias le asignaríamos un índice al campo "título", y las noticias estarían ordenadas de la "a" a la "z" por su título. La búsqueda se realizará primero en el índice, encontrando rápidamente el título buscado, ya que están todos ordenados y, como resultado, el índice le devolverá al programa MySQL el identificador (id) del registro en la tabla original, para que MySQL vaya directamente a buscar ese registro con el resto de los datos, sin perder tiempo. Todo esto, por supuesto, de manera completamente invisible para nosotros.

Para indicar que queremos crear un índice ordenado por un campo determinado, al campo que se indexará debemos especificarle, dentro del atributo **Índice**, del menú de selección que aparece a su derecha, la opción **Index**:

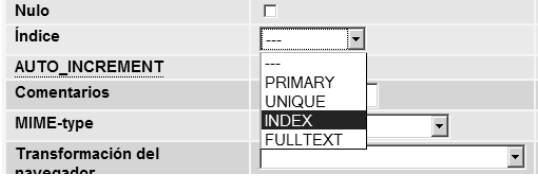


**Figura 11-49. Indexando un campo para agilizar las búsquedas.**

Como podemos apreciar, dentro del menú **Índice** aparecen otras opciones: **Primary**, **Unique** y **Fulltext**. Veamos en qué consiste cada una de estas variantes.

### PRIMARY Key y Auto_increment

Siempre, en toda tabla, uno de los campos (por convención, el primero, y también por convención, usualmente llamado **id** –por "identificador"–), debe ser definido como **clave primaria** o *Primary Key*. Esto impedirá que se le inserten valores repetidos y que se deje nulo su valor.

Habitualmente, se especifica que el campo elegido para clave primaria sea **numérico**, de tipo **entero** (en cualquiera de sus variantes, según la cantidad de elementos que se identificarán), y se le asigna otro atributo típico, que es **Auto_Increment**, es decir, que no nos preocupamos por darle valor a ese campo: al agregar un registro, MySQL se ocupa de incrementar en uno el valor de la clave primaria del último registro agregado, y se lo asigna al nuevo registro. Este campo no suele tener ninguna relación con el contenido de la tabla, su objetivo es simplemente **identificar** cada registro de forma única, irrepetible.


**Figura 11-50. Definiendo una clave primaria, con Auto_Increment.**

Podemos definir un solo campo como clave primaria, o dos o más campos combina- dos. En caso de haber definido dos o más campos para que juntos formen el valor único de una clave primaria, diremos que se trata de una clave primaria "combinada" o "compuesta".

### UNIQUE

Si especificamos que el valor de un campo sea **Unique**, estaremos obligando a que su valor **no pueda repetirse** en más de un registro, pero no por eso el campo se considerará clave primaria de cada registro. Esto es útil para un campo que guarde, por ejemplo, números de documentos de identidad, la casilla de correo electrónico usada para identificar el acceso de un usuario, un nombre de usuario, o cualquier otro dato que no debamos permitir que se repita. Los intentos por agregar un nuevo registro que contenga un valor ya existente en ese campo, serán rechazados.

### FULLTEXT

Si en un campo de tipo TEXT creamos un índice de tipo FULLTEXT, MySQL examinará el contenido de este campo **palabra por palabra**, almacenando cada palabra en una celda de una matriz, permitiendo hacer búsquedas de palabras

contenidas dentro del campo, y no ya una simple búsqueda de coincidencia total del valor del campo, que son mucho más rápidas pero no sirven en el caso de búsquedas dentro de, por ejemplo, el cuerpo de una noticia, donde el usuario desea encontrar noticias que mencionan determinada palabra.

Se ignoran las palabras de menos de cuatro caracteres y palabras comunes como artículos, etc. que se consideran irrelevantes para una búsqueda, así como también se ignoran diferencias entre mayúsculas y minúsculas. Además, si la palabra buscada se encuentra en más del 50% de los registros, no devolverá resultados, ya que se la considera irrelevante por "exceso" de aparición (deberemos refinar la búsqueda en este caso).

Con esto, damos por terminada esta revisión bastante exhaustiva. Ya hemos aprendido a crear una base de datos y una tabla, definiendo con total precisión sus campos valiéndonos de tipos de datos y atributos, por lo tanto, estamos en condiciones de comenzar a programar todo lo necesario para que nuestras páginas PHP se conecten con una base de datos y puedan enviarle o le soliciten datos. Justamente, éste será el tema del próximo capítulo.

# Llevando datos de la base a las páginas

# 12

## Cómo leer datos desde una base con PHP

Uno de los principales objetivos de la creación de una tabla en una base de datos es que todos (o parte de) los datos contenidos en ella se puedan leer y mostrar dentro de una página de nuestro sitio Web.

En este capítulo, aprenderemos los pasos necesarios para **leer** los datos almacenados en una **tabla** de una base de datos y, a continuación, los mostraremos dentro del código HTML de una página.

Los pasos que se seguirán son los siguientes:

1. Que el programa intérprete de PHP **se identifique** ante el programa gestor de MySQL (que obtenga la autorización para solicitar datos).
2. Una vez autorizado, prepararemos la **orden del lenguaje SQL** que queremos ejecutar para traer los datos que nos interesan.
3. **Ejecutaremos la orden** SQL, y obtendremos un **paquete de datos** como respuesta.
4. Recorreremos el paquete de datos mediante un bucle que vaya **extrayendo de a un registro por vez**, registro que descompondremos en campos que iremos mostrando hasta que no haya más datos para mostrar.

Vamos a aprender estos pasos con un ejemplo concreto: construiremos una página que muestre el contenido de la tabla "mensajes" creada anteriormente.

## Probar la conexión

El programa intérprete de PHP –que procesa nuestro código PHP– le debe demostrar al programa gestor de MySQL que tiene autorización para solicitarle datos.

Para intentar establecer una conexión entre estos dos programas, utilizaremos la función **mysqli_connect**, que requiere que completemos cuatro parámetros: **host**, **usuario** y **clave** (de MySQL, por supuesto: el usuario "root" al que le habíamos creado la contraseña "clave"), y **base**, en ese orden, en consecuencia la estructura básica de esta función será:

```
<?php
$conexion    =    mysqli_connect("host","usuario","clave",
"base");
?>
```

¿Qué significa cada uno de estos **parámetros**?

1. **Host** es la computadora en la que se encuentra instalado el programa gestor de MySQL; típicamente, será la misma computadora en la que se está ejecutando nuestro intérprete de PHP, ya sea que lo estemos haciendo localmente en nuestra casa, o en un *hosting*. Siendo así, el primer parámetro de la función mysql_connect deberá decir **localhost** (el host local, el mismo en el que se está ejecutando nuestro código PHP). En el raro caso de que MySQL se encuentre en otra computadora distinta a la que está ejecutando PHP, deberemos colocar la **dirección IP** de esa máquina como primer parámetro.
2. **Usuario** es el nombre de un usuario del software MySQL que tiene autorización para ejecutar consultas. En nuestra instalación de XAMPP, el usuario que se había creado automáticamente para utilizar la base de datos se llamaba **"root"**, pero en un *hosting* el nombre de usuario será diferente, ya sea porque lo ha creado el administrador del *hosting,* o porque lo creamos nosotros mismos desde un panel de administración del sitio Web (esto varía mucho en cada *hosting,* por lo que lo dejamos para que lo consulten al soporte técnico de su empresa proveedora). Suele ser el usuario con el que nos identificamos para entrar en el phpMyAdmin.

3. **Clave:** es la contraseña del usuario con el que nos estemos identificando ante el programa MySQL. En nuestra instalación de XAMPP, habíamos especificado la contraseña "clave", pero en un *hosting* tendremos que pedirla o crearla, como en el ejemplo del usuario que se explicó en el paso anterior.
4. **Base:** el nombre de la base de datos a la que nos estaremos conectando.

Por esta razón, localmente nuestra función **mysqli_connect** quedaría así:

```
<?php
$con = mysqli_connect("localhost","root","clave","base");
?>
```

En el raro caso de una Intranet con **distintas máquinas** para el intérprete de PHP y el programa gestor de MySQL, quedará algo como lo que se expresa a continuación (obviamente los datos no son reales):

```
<?php
$con =
mysqli_connect(123.456.10.1,"usuario","password","base");
?>
```

Al ser **mysqli_connect** una función que solo puede devolver "verdadero" o "falso" (booleana), es posible envolverla dentro de un condicional que evaluará si MySQL nos autoriza que le enviemos consultas, o no:

```
<?php
$con = mysqli_connect("localhost","root","clave","base");
if (!$con){
      echo "<p>MySQL no conoce ese usuario y password, o
esa base, y rechaza la conexión</p>";
} else {
      echo "<p>MySQL le ha dado permiso a PHP para ejecutar
consultas con ese usuario y clave</p>";
}
```

```
?>
```

Con esto, ya tenemos listo el primer paso. Nuestro programa intérprete de PHP ya tiene autorización para realizar consultas al programa gestor de MySQL. Ahora, es momento de crear la **consulta** que ejecutaremos.

## La orden "SELECT": entendiendo el lenguaje SQL

Para que el programa gestor de MySQL nos entregue un conjunto de datos, deberemos aprender a decírselo de forma que nos comprenda y, para ello, utilizaremos órdenes del **lenguaje SQL** *(Structured Query Language* o Lenguaje de consultas estructurado).

En este caso que necesitamos **leer datos** de una tabla, la orden será **SELECT** (seleccionar). Mediante esta orden, le pediremos al gestor de MySQL que nos entregue exactamente aquellos datos que queremos mostrar en nuestras páginas.

La sintaxis básica de la orden **SELECT** es la siguiente:

```
SELECT campo1, campo2, campo3 FROM tabla
```

Esto se traduciría como "Seleccionar los campos indicados, de la tabla indicada".

Con un ejemplo será más simple entender su uso:

```
SELECT nombre, email FROM mensajes
```

Esta orden seleccionará los campos (columnas) **nombre** y **email** a través de todos los registros de nuestra tabla denominada **mensajes**. Es decir, devolverá **todos los registros (filas)** de la tabla, pero "cortados" verticalmente, extrayendo solo las dos columnas solicitadas (nombre y email), en consecuencia, nos llegaría lo siguiente (probemos insertar varios registros antes de hacer esto):

| | |
|---|---|
| **Pepe** | pepe@pepe.com |
| **Carlos** | carlos@garcia.com |
| **Alberto** | alberto@perez.com |

Si, en cambio, quisiéramos traer "todos los campos" de la tabla (tarea muy frecuente y, que en caso de tener muchos campos la tabla, podría resultar bastante tedioso completar sus nombres uno por uno) podemos utilizar un carácter que cumple las funciones de comodín para simbolizar "todos los campos de la tabla". Este carácter es el **asterisco**, con el cual la orden quedaría:

```
SELECT * FROM mensajes
```

Esta orden traería todas las columnas (campos) de la tabla mensajes (y, desde ya, todos los registros también):

| | | | |
|---|---|---|---|
| 1 | Pepe | pepe@pepe.com | Bla, bla, bla... |
| 2 | Carlos | carlos@garcia.com | Bla, bla, bla... |
| 3 | Alberto | alberto@perez.com | Bla, bla, bla... |

**Nota al margen**: aquellos curiosos que deseen "jugar" y practicar consultas SQL desde el phpMyAdmin, pueden hacerlo pulsando en la solapa SQL y, a continuación, podrán escribir sus órdenes en lenguaje SQL dentro del área de texto y pulsar **Continuar** para ver el resultado de su ejecución.

Resumiendo, para finalizar este segundo paso, podemos almacenar la orden SQL en una **variable** (hasta el momento en que la necesitemos):

```
<?php
$consulta = "SELECT * FROM mensajes";
?>
```

Ahora que tenemos preparada nuestra **orden SQL**, es momento de pasar al tercer paso y hacer que el intérprete de PHP se la envíe al gestor de MySQL, obteniendo alguna respuesta.

## Ejecutar la orden y obtener una respuesta

Para que PHP envíe una consulta SQL hacia el gestor de MySQL utilizaremos una función que posee el lenguaje PHP llamada mysqli_query (hacer una consulta -query- a una base de datos MySQL).

Esta función requiere dos parámetros: el primero es el link de conexión obtenido por la función mysqli_connect, y el segundo, la orden SQL a ejecutar.

Veamos un ejemplo completo:

```
<?php
```

```
$con = mysqli_connect("localhost","root","clave","base");
if (!$con)
      echo "<p>Hubo un error en la conexión a la base</p>";
} else {
      $consulta = "SELECT * FROM mensajes";
      if( $datos = mysqli_query($con, $consulta) ){
         /* Aquí usaremos esos datos obtenidos */
      } else {
         echo "No se pudo realizar la consulta";
      }
}
?>
```

Notemos que indicamos los dos datos que necesita mysqli_query: la variable $con que se refiere a la conexión establecida a una base, y la variable $consulta donde recientemente hemos almacenado la orden del lenguaje SQL que vamos a ejecutar.

Pero si ejecutáramos este código tal cual como está aquí, nos perderíamos el objetivo principal de esta consulta, que era **obtener datos** de la tabla. Estamos ejecutando esta consulta, que a cambio nos debería devolver un "paquete" de datos, al que hemos llamado $datos, pero... no los estamos utilizando luego. Recordemos que es necesario que recibamos los datos que nos devuelve, en el mismo instante en el que ejecutamos la consulta, es decir: **debemos asignarle a alguna variable el resultado** de ejecutar la función mysqli_query, para que sea esta variable la que contenga la respuesta recibida.

Por ejemplo:

```
<?php
$datos = mysqli_query($consulta);
?>
```

De esta manera, culminamos el tercer paso y estamos listos para el cuarto y último paso.

## Integrar los datos al código HTML

Solo resta recorrer con un bucle el paquete de datos e ir **generando el código HTML que incluya los datos** que iremos mostrando en la pantalla.

Comencemos diciendo que $datos no es una variable común, ni una matriz, sino que es un **resultado de una consulta** o, más sencillamente, un "paquete cerrado" de datos que no podremos utilizar directamente, sino que tendremos que **descomprimirlo** para que se pueda mostrar.

El paquete de datos contiene **varias filas** (los mensajes de Pepe, García, Pérez, etc.), pero, a su vez, **cada fila contiene varios campos** (id, nombre, email y mensaje).

Será preciso ir tomando con pinzas **de a una fila horizontal por vez** del paquete que es $datos, y una vez que tengamos seleccionada una fila, deberemos traspasarla dentro de una matriz (podremos llamarla $fila) que contenga en cada celda, un campo de esa fila:

$fila["id"], $fila["nombre"], $fila["email"] y $fila["mensaje"].

Exactamente esto es lo que hace una función llamada mysqli_fetch_array: toma como parámetro un paquete como el paquete $datos que tenemos y le **extrae una fila completa**, que debemos asignar a una matriz que llamaremos $fila.

El código sería así:

```
<?php

$fila = mysqli_fetch_array($datos);

?>
```

En este momento, $fila es una matriz (o *array*, por eso lo de *...fetch_array*) que contiene **la primera fila** completa de la tabla, con los valores de sus cuatro campos ubicados en cuatro celdas de la matriz $fila. Los índices alfanuméricos de esta matriz $fila, que contiene un registro de la tabla, son los **nombres** que tenían los campos en la tabla.

En este momento, $fila["id"] contiene un "1", $fila["nombre"] contiene "Pepe", $fila["email"] contiene "pepe@pepe.com" y $fila["mensaje"] contiene "Bla, bla, bla...".

Pero, ¿qué pasará con el resto de filas que no hemos leído? La función mysqli_fetch_ array solo lee **una** fila cada vez que se ejecuta. Para poder leer no solo la primera, sino también el resto de las filas, deberíamos realizar un **bucle**, que ejecute esta lectura y asignación de una fila de datos por vez,

**repetitivamente**, mientras todavía quede dentro de $datos alguna fila que asignar (será un bucle de tipo *while*, ya que no sabemos de antemano cuántas filas nos devolverá la consulta).

En cuanto se almacene una fila en la matriz, la deberemos mostrar inmediatamente (escribir en pantalla con un **print** o **echo**), ya que una vuelta después del bucle, el contenido de $fila será borrado para permitir almacenar los valores de la **siguiente fila** de la tabla.

La condición del bucle será "mientras podamos descomprimir con mysqli_fetch_array una nueva fila del paquete de datos, y almacenarla en una matriz"... Pensemos esto detenidamente, y veamos cómo sería esa parte:

```
<?php
while ( $fila = mysqli_fetch_array($datos) ){


      echo "<p>";
      echo $fila["id"];
      echo " - "; // un separador
      echo $fila["nombre"];
      echo " - "; // un separador
      echo $fila["email"];
      echo " - "; // un separador
      echo $fila["mensaje"];
      echo "</p>";


}
?>
```

Para terminar, veamos un ejemplo completo, pasado en limpio, de los cuatro pasos juntos:

```
<?php
// 1) Conexión
```

```
if ($conexion = mysqli_connect("localhost","root","clave",
"cursos")){


      echo "<p>MySQL le ha dado permiso a PHP para ejecutar
consultas con ese usuario y password</p>";
      // 2) Preparar la orden SQL
      $consulta = "SELECT * FROM mensajes";
      // 3) Ejecutar la orden y obtener datos
      $datos = mysqli_query($consulta);
      // 4) Ir imprimiendo las filas resultantes
      while ( $fila = mysqli_fetch_array($datos) ){


         echo "<p>";
         echo $fila["id"];
         echo " - "; // un separador
         echo $fila["nombre"];
         echo " - "; // un separador
         echo $fila["email"];
         echo " - "; // un separador
         echo $fila["mensaje"];
         echo "</p>";
      } // fin bucle
   } else { echo "<p>No se consiguieron datos</p>"; }
} else {
      echo "<p>MySQL no conoce ese usuario y password</p>";
}
```

Ahora sí que estamos preparados para mostrar los datos almacenados en nuestras tablas. Vamos a volver a hacer esto miles de veces, cambiando "productos" por "mensajes", "empleados", "comentarios", o lo que sea que haya que **mostrar** en una página de nuestro sitio.

Desde ya que la forma de mostrar estos datos dentro del código HTML irá variando: en algunos casos, precisaremos mostrar un texto (párrafos como los del ejemplo anterior); en otros, necesitaremos generar etiquetas **option** dentro de un elemento **select**; otras veces, filas de una tabla; o botones de tipo **radio**, o casillas de verificación.

Lo ideal es que nos vayamos creando una serie de **funciones**, que reciban como parámetros el nombre de la tabla y los campos que se seleccionarán, más algún dato específico (label, name, elemento a mostrar seleccionado, etc.) y que devuelvan el bloque HTML completo cargado con los datos de la tabla.

En este mismo capítulo, veremos algunos ejemplos útiles, pero antes vamos a profundizar un poco más en la estructura de la orden SELECT del lenguaje SQL, para que podamos aprovechar toda su potencia.

# Complementos de la orden SELECT del lenguaje SQL

Ya hemos visto la sintaxis mínima de la orden SELECT del lenguaje SQL, que nos permite seleccionar determinados campos de una tabla.

Pero ahora vamos a ir un poco más allá y conoceremos un modificador **condicional** denominado **WHERE**, que resulta muy útil para aumentar su potencia a la hora de extraer datos con precisión de la base de datos (no exageramos si decimos que prácticamente todas nuestras consultas SQL poseerán un WHERE).

Para poner en práctica este concepto, crearemos una nueva tabla en nuestra base de datos. La llamaremos "empleados" y poseerá los siguientes campos:

```
CREATE TABLE empleados (

id TINYINT( 3 ) NOT NULL AUTO_INCREMENT PRIMARY KEY ,
nombre VARCHAR( 40 ) NOT NULL ,

apellido VARCHAR( 40 ) NOT NULL , edad TINYINT( 2 ) NOT
NULL ,

pais VARCHAR( 30 ) NOT NULL , especialidad VARCHAR( 30 )
NOT NULL
```

```
) ENGINE = MYISAM
```

Podemos escribir esta orden dentro de la ventana de ejecución de código SQL del phpMyAdmin, y la tabla será creada idéntica a la original:

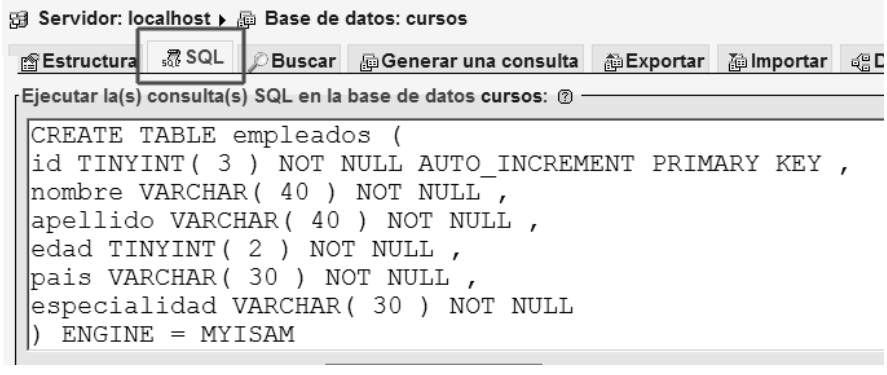


**Figura 12-1. Creación de una tabla a partir de un código SQL.**

Una vez creada la tabla, vamos a cargar **varios registros** con diferentes valores en los distintos campos, para poder utilizarlos en las órdenes de selección condicional. Recomendamos cargar los siguientes datos, ya que fueron elegidos especialmente para mostrar resultados significativos en las consultas que realizaremos a continuación:

| id | nombre | apellido | edad | pais | especialidad |
|---|---|---|---|---|---|
| 1 | Pedro | Fernández | 34 | España | Matemáticas |
| 2 | José | García | 28 | México | Sistemas |
| 3 | Guillermo | Pérez | 32 | España | Contabilidad |
| 4 | Alberto | Maza | 45 | México | Matemáticas |
| 5 | Luis | Puente | 43 | Argentina | Sistemas |
| 6 | Claudio | López | 41 | España | Medicina |
| 7 | Mario | Juárez | 51 | México | Sistemas |
| 8 | Alan | Flores | 25 | Perú | Sistemas |

**Figura 12-2. Datos de ejemplo, cargados con la intención de usarlos en selecciones condicionales.**

## El condicional WHERE

A nuestras órdenes SELECT será muy común que les agreguemos **condiciones**, para que nos devuelvan un conjunto menor de resultados: solo aquellos registros que cumplan con la condición. La condición tomará en cuenta el valor de alguno de los **campos**. Por ejemplo, imaginemos que necesitamos una lista de los empleados de la tabla anterior, pero no todos, sino solo aquellos cuya especialidad sea "Matemáticas".

La sintaxis de la expresión condicional WHERE es similar a la de un **if**: se trata de una afirmación en la forma de una **comparación**, cuyos elementos serán un **campo** de la tabla, un **operador** de comparación y un **valor** contra el que se compara:

```
...WHERE campo operador 'valor'
```

Lo que traducido a un ejemplo más concreto, podría ser:

```
...WHERE especialidad='Matemáticas'
```

Y completando la orden SELECT con esta condición, tendríamos una orden completa:

```
SELECT nombre, apellido FROM empleados
WHERE especialidad='Matemáticas'
```

Si escribimos esta orden en la ventana de ejecución de código SQL del phpMyAdmin, y pulsamos en **Continuar**, obtendremos como resultado justo los datos que necesitábamos:

Obtuvimos solo el nombre y el apellido de los empleados cuyo campo "especialidad" tenía un valor exactamente igual a "Matemáticas".

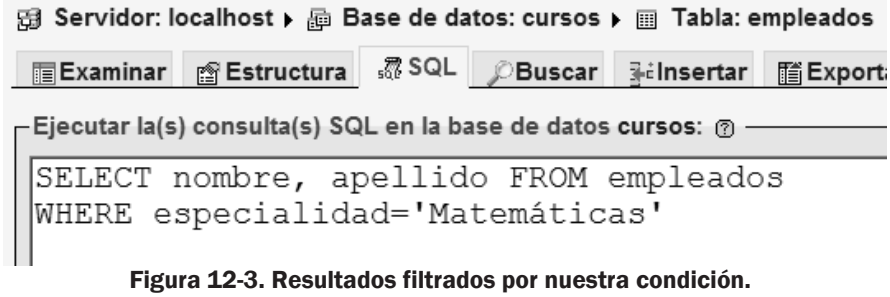


**Figura 12-3. Resultados filtrados por nuestra condición.**

### Operadores para utilizar con WHERE

Los **operadores** de los que disponemos dentro de un WHERE son muy similares a los que usamos en los condicionales de PHP, salvo el último, denominado LIKE, que enseguida analizaremos:

| Operador | Significa |
|---|---|
| = | Igual |
| > | Mayor que |
| < | Menor que |
| >= | Mayor o igual |
| <= | Menor o igual |
| <> | Distinto de |
| LIKE | Que incluya... (ahora lo analizaremos) |

**Tabla 12-1. Operadores de comparación de SQL.**

El operador LIKE permite establecer condiciones de "similaridad" mucho menos estrictas que la simple igualdad del contenido completo de un campo permitida por el ya conocido signo igual. Por ejemplo, si en lugar de necesitar un listado de empleados cuya especialidad sea exactamente "Matemáticas", necesitáramos un listado de empleados cuya especialidad "comience con M", podríamos usar este operador. Sirve para encontrar un carácter (o más) al principio, en medio o al final de un campo.

Se vale de un "comodín" que es el signo de **%** (porcentaje), que simboliza la presencia potencial de caracteres en esa dirección.

Se usa de la siguiente forma:

1. ...WHERE especialidad LIKE **'M%'**

   Nos devolverá aquellos registros en los cuales el campo especialidad contenga una letra "M" **al principio** del campo, sin importar si a continuación sigue cualquier otra cosa (o incluso si no le sigue nada). En nuestra tabla de ejemplo, incluirá a quien tenga "**M**edicina" y "**M**atemáticas" como especialidad.

2. ...WHERE especialidad LIKE **'%M%'**

   Devolverá los registros que en el campo especialidad contengan una letra "M" (mayúscula o minúscula, es indistinto) al principio, en el medio o al final del campo. En nuestra tabla, nos devolverá a

quienes tengan especialidades de "**M**edicina", "**M**atemáticas" y "Siste**m**as".

3. ...WHERE especialidad LIKE **'%M'**

   Seleccionará los que en el campo especialidad contengan una letra "M" justo **al final**, como **última letra** del campo (o como **única** letra del campo), pero sin que la siga otra letra.

   Es fundamental que el valor buscado, junto con el carácter de "%", estén envueltos entre **comillas simples**.

Este operador LIKE es el que típicamente se utiliza en cualquier campo de búsqueda que indague dentro de un campo de textos o títulos de noticias, comentarios, mensajes de foros, nombres de productos, etc. Lo importante es que nos permite encontrar la palabra buscada en **parte** del título o nombre del producto. Por ejemplo, si buscamos ...WHERE nombre LIKE '%TV%', encontraremos registros que en el nombre de un producto contengan "**TV** LG 42 pulgadas", o "Mini **TV** de pulsera", o "Monitor LED con sintonizador de **TV**", que sería imposible de seleccionar con el operador "=" normal (que exige que el contenido completo del campo sea igual a lo buscado).

Veamos ahora un operador útil para **rangos**: se trata del operador **BETWEEN**, que nos permite especificar un límite mínimo y un límite máximo. Probémoslo con la edad de nuestros empleados. Si precisamos un listado de empleados de entre 40 y 50 años, lo obtendríamos con esta orden:

```
SELECT * FROM empleados WHERE edad BETWEEN 40 AND 50
```

Podemos indicar la exclusión de un rango mediante NOT BETWEEN:

```
SELECT * FROM empleados WHERE edad NOT BETWEEN 18 AND 40
```

También, podemos **unir** varias condiciones mediante los operadores lógicos AND y OR. Veamos un ejemplo de OR (nos devolverá tanto los empleados que en su campo "pais" figure el valor "España" como los que sean de "Argentina"):

```
SELECT * FROM empleados WHERE pais='España' OR
pais='Argentina'
```

En cambio, al combinar condiciones con AND, los resultados serán más específicos: solo aquellos registros que cumplan con todas las condiciones unidas mediante AND:

```
SELECT * FROM empleados WHERE pais='Argentina' AND
especialidad='Sistemas'
```

También, podemos buscar coincidencias con varios valores posibles para un campo, y no con uno solo, usando IN:

```
SELECT * FROM empleados
```

```
WHERE pais IN ('México','Argentina','Perú')
```

De esta manera, proporcionamos, en un solo paso, una **lista de valores** para el campo "pais", que, en este ejemplo, nos devolverá los registros de empleados cuyo país sea México, Argentina o Perú.

## Ordenando los resultados

Al momento de obtener los resultados de una consulta SQL, podemos desear que éstos se nos entreguen de forma ordenada, por el valor de alguno de los campos. Para ello, se utiliza la expresión ORDER BY, a la que debe seguir el nombre del campo por el cual queremos ordenar los resultados.

Probemos obtener una lista de empleados ordenados alfabéticamente por el apellido:

```
SELECT * FROM empleados ORDER BY apellido
```

Contamos con dos modificadores adicionales que especifican si el ordenamiento será realizado de menor a mayor (ascendente: ASC, no es necesario especificarlo ya que es la forma en la que se ordenan por defecto), o de forma descendente (de mayor a menor, en cuyo caso el modificador es DESC). Por ejemplo:

```
SELECT * FROM empleados ORDER BY edad DESC
```

## Limitando la cantidad de resultados

Cuando la cantidad de resultados pueda ser muy grande, será conveniente mostrar solamente cierta cantidad de resultados (tal como hace Google en su listado de resultados). Esto es muy usado al mostrar resultados "paginados" (con la opción de ver otros conjuntos siguientes o anteriores de resultados, mediante < y > o similares).

Para limitar la cantidad de registros que devuelve una consulta, usaremos **LIMIT**. Este modificador requiere que especifiquemos dos valores: **a partir de qué** número de resultado devolver y **cuántos** resultados devolverá.

Con un ejemplo será más fácil de entender:

```
SELECT * FROM empleados ORDER BY apellido LIMIT 0,3
```

Eso nos devolverá tres registros, a partir del primero, pero no del primero físicamente en la tabla, sino el primero de los resultados que hubiera devuelto la consulta ordenada por apellido, es decir, el de apellido más cercano a la **A**.

Si en un paginador tuviéramos que realizar otra consulta para traer los tres siguientes, lo único que cambiará es el valor de inicio de LIMIT:

```
SELECT * FROM empleados ORDER BY apellido LIMIT 3,3
```

Eso devolverá tres registros, posteriores al tercero (tomando en cuenta el apellido).

## Seleccionando valores no repetidos

Podemos obtener fácilmente los valores únicos de un campo, sin tener en cuenta sus repeticiones.

Por ejemplo, si queremos obtener el listado de especialidades, sin repetir ninguna, que al menos uno de los empleados tiene asignada, podemos solicitarlo agregando la palabra **DISTINCT** delante del campo que se seleccionará:

```
SELECT DISTINCT especialidad FROM empleados
```

Eso nos devolverá la lista de especialidades, una por registro.

## Funciones estadísticas

Muchas veces, no necesitaremos que una consulta nos devuelva "los datos" que están almacenados en la tabla, sino alguna información estadística "acerca de" esos datos. Por ejemplo, **cuántos** registros logró seleccionar una consulta, cuál es el valor **mínimo** o **máximo** de un campo, cuál es la **sumatoria** de los valores de ese campo a lo largo de todos los registros, o cuál es el valor **promedio**:

| Función | Qué devuelven |
|---|---|
| COUNT | La cantidad de registros seleccionador por una consulta. |
| MIN | El valor mínimo almacenado en ese campo. |
| MAX | El valor máximo almacenado en ese campo. |
| SUM | La sumatoria de ese campo. |
| AVG | El promedio de ese campo. |

**Tabla 12-2. Funciones estadísticas.**

En caso de que necesitemos consultar "si existe" algún dato en una tabla (lo sabremos si su cantidad es al menos 1), o si queremos saber cuántos registros existen que cumplan con determinada condición, podemos utilizar la función COUNT:

```
SELECT COUNT(*) FROM empleados WHERE pais='México'
```

Esta función devuelve, no un conjunto de registros, sino un único número: la **cantidad de registros** existentes que cumplan con la condición especificada (y si no especificamos ninguna, nos devolverá el número total de registros de la tabla).

Las otras cuatro funciones, al igual que COUNT, devuelven un único **número** como resultado. Se utilizan del mismo modo, junto a la palabra SELECT, y al tratarse de funciones, todas envuelven entre **paréntesis** el nombre del campo que van a analizar:

```
SELECT MIN(edad) FROM empleados;

SELECT MAX(edad) FROM empleados;

SELECT SUM(edad) FROM empleados;

SELECT AVG(edad) FROM empleados;
```

Existen muchísimos otros operadores, funciones y modificadores en el lenguaje SQL, por lo que recomendamos seguir investigando por nuestra cuenta en algún buen manual específico del lenguaje SQL.

# Funciones propias para mostrar datos

Es sumamente útil crear una "biblioteca" de funciones, especializadas en leer datos de una base de datos y mostrarlos luego dentro de diferentes tipos de etiquetas HTML. Tarde o temprano, tendremos que realizar una tarea similar en nuestros sitios.

Nos conviene dividir todo el trabajo en varias funciones complementarias:

1. Una función que intente establecer una **conexión** con la base y que trate de **seleccionarla** (si llega a fallar cualquiera de estos dos pasos, no se podrá hacer nada más).
2. Una función que ejecute una **consulta** y obtenga como respuesta un "paquete" de datos. Necesitará como parámetro de entrada la orden SQL que ejecutará, y devolverá un "paquete" de datos (o *false* en caso de que el paquete de datos esté vacío).
3. Varias funciones, cada una especializada en **mostrar los datos** recibidos, envueltos en etiquetas HTML específicas.

Vamos paso por paso:

1) Para crear una función que se encargue de la **conexión** a la base de datos, una práctica previa —muy recomendable— es definir, en un único **archivo**

**externo**, todas las variables relativas a la conexión con MySQL como, por ejemplo, el **host**, **usuario**, **contraseña** y **nombre de la base** de datos, ya que su valor suele cambiar de un *hosting* a otro.

Si lo hacemos de esta manera, cuando probemos nuestro sistema en un *hosting* diferente, no tendremos que ir cambiando a mano esos datos en cada página de nuestro sitio.

Supongamos que creamos un archivo llamado datos.php con ese objetivo. Su contenido será el siguiente:

```
<?php
$host = "localhost";
$usuario = "root";
$clave = "clave";
$base = "cursos";
?>
```

Luego, haremos un **include** de ese archivo dentro de cada página donde vayamos a conectarnos con la base de datos, y usaremos esas variables en las funciones.

Por ejemplo:

```
<?php
// Incluimos esos datos:
include("datos.php");


// Usamos esas variables:
if ($conexion = mysqli_connect($host,$usuario,$clave,
$base)){


      // etc.
?>
```

De esta forma, si cambiamos de servidor (entre un servidor local de pruebas y un *hosting,* o de un *hosting* de un cliente al *hosting* de otro cliente), donde no tendremos los mismos nombres de bases, usuarios y claves, solo tendremos que modificar **un único archivo** (datos.php) una sola vez, y no

tendremos que modificar absolutamente nada dentro del código de nuestras páginas.

Lo ideal es tener un archivo datos.php en el servidor local con los datos de conexión locales, y otro datos.php en el *hosting,* con los datos de conexión del *hosting.*

Habiendo preparado en un archivo aparte los datos de conexión, ahora ya podemos crear una **función propia** que se encargue de la conexión a la base de datos. Si bien en PHP contamos con una función para establecer una conexión con MySQL (mysqli_connect), la idea es comprobar si se pudo realizar la conexión mdiante un condicional, y para eso es ideal crear una función propia.

La idea es que esa función se encargue de brindar al usuario un mensaje en caso de fallo, ya que un fallo a este nivel impedirá el uso del sistema, y al usuario poco le importará si no puede usar el sistema porque "falló la conexión" o porque "no pudo seleccionarse la base de datos", ambas cosas le son desconocidas.

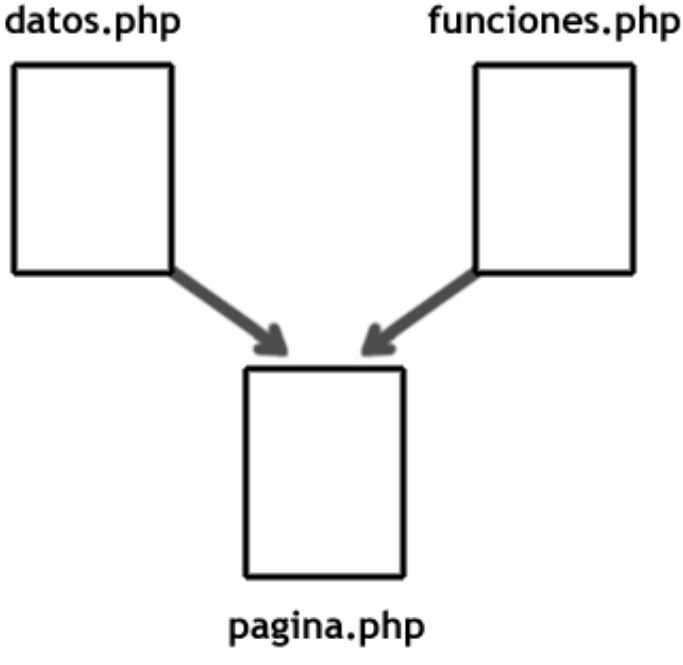


**Figura 12-4. Esquema de relación entre archivos de datos, funciones y páginas.**

Crearemos entonces una función que devuelva una conexión, así podemos evaluarla dentro de un condicional al momento de ejecutarla:

```
<?php
function conectarBase($host,$usuario,$clave,$base){
      if (!$enlace =
mysqli_connect($host,$usuario,$clave,$base)){
      return false; /* Si no pudo conectarse, devuelve
"false"  */
} else {
      return $enlace;
      /* Si pudo conectarse, devuelve el link a esa
conexión */
}
}
?>
```

A esta función, tal como vimos en la figura anterior, la crearemos dentro de un **archivo externo** llamado funciones.php, al que también lo incluiremos dentro de las páginas mediante una orden **include**:

```
include("datos.php");
include("funciones.php");
```

Sin embargo, en esta función, todavía existe un problema, que podemos comprobar fácilmente si modificamos intencionalmente alguno de los datos de conexión (probemos cambiando el **host**, usuario o clave, por uno que no sea el correcto), y es que, si llega a fallar el intento de conexión, aparecerá en la pantalla un **mensaje de error** en inglés.

Podemos **ocultar** fácilmente ese mensaje de error si anteponemos a la función de PHP que produce el error (mysqli_connect, en este caso) el operador de control de errores de PHP, que es una simple **arroba**:

```
<?php
function conectarBase($host,$usuario,$clave,$base){
      if (!$enlace =
@mysqli_connect($host,$usuario,$clave,$base)){
```

```
      /* Notemos la arroba antepuesta a la función que
devolvía error */
      return false;
} else {
      return $enlace;
}
}
?>
```

Este operador de control de errores (la arroba) lo podemos anteponer a cualquier expresión que devuelva datos (como, por ejemplo, una llamada a una función, una orden include, una variable o una constante) y, de esta forma, evitamos que se muestre el mensaje de error de PHP en caso de fallo.

Volviendo a la función que acabamos de crear, la utilizaríamos desde nuestras páginas dentro de un **condicional**:

```
<?php
// Incluimos los datos de conexión y las funciones:
include("datos.php");
include("funciones.php");

// Usamos esas variables:
if ( conectarBase($host,$usuario,$clave,$base) ){

      // Aquí haríamos el resto de operaciones...

} else {

      echo "<p>Servicio interrumpido</p>";
```

```
}
?>
```

2) Ahora, es momento de crear una función que nos permita realizar **una consulta** a la base de datos, obteniendo un **paquete de datos** como respuesta (o *false* en caso de fallo).

Esa función podría ser así:

```
<?php
function consultar($conexion, $consulta){
      if (!$datos = mysqli_query($conexion, $consulta) or
mysqli_num_ rows($datos)<1){
         return false; /* Si fue rechazada la consulta por
errores de sintaxis, o ningún registro coincide con lo
buscado, devolvemos false */
      } else {
         return $datos; /* Si se obtuvieron datos, los
devolvemos al punto en que fue llamada la función */
      }
}
?>
```

Notemos el uso de la función **mysqli_num_rows**, que devuelve la **cantidad de registros** que obtuvo una consulta. En nuestro condicional, estamos planteando una doble condición: si la consulta en sí falló (y devolvió *false*) —eso es lo que se evalúa antes del or—, o si la cantidad de registros obtenidos fue... ninguno. Hemos decidido aunar en una sola condición ambas situaciones, desde ya que podríamos descomponerlo en diferentes condicionales, pero en este momento creemos que sería complicar la lectura del ejemplo.

De la misma que en nuestra función anterior, utilizaremos nuestra función **consultar** dentro de un condicional, cuando estemos seguros de que la conexión a la base funcionó.

Veamos cómo va quedando el código de la página que va llamando a estas funciones sucesivamente:

```
<?php
// Incluimos los datos de conexión y las funciones:
include("datos.php");
include("funciones.php");
// Usamos esas variables:
if ( $con = conectarBase($host,$usuario,$clave,$base) ){
      $consulta = "SELECT * FROM empleados";
      if ( $paquete = consultar($con, $consulta) ){
         /* Aquí llamaremos a una función que muestre esos
datos  */
      } else {
         echo "<p>No se encontraron datos</p>";
      }
} else {
      echo "<p>Servicio interrumpido</p>";
}
?>
```

3) Por último, solo nos falta crear distintas funciones que se ocupen de **mostrar el "paquete" de datos** recibido, con la diferencia de que cada una se especializará en mostrar los datos dentro de distintas etiquetas HTML: veremos cómo mostrar datos dentro de un menú de selección, una tabla, botones de radio y casillas de verificación.

## Menú de selección dinámico

Comencemos por uno de los elementos HTML más usados para mostrar datos de una base: los **menús de selección** (elementos **select** HTML). Es muy común que estos elementos muestren listados de países, provincias, estados civiles,

categorías y cualquier otra variedad de opciones excluyentes entre sí, de las que el usuario deba elegir una obligatoriamente. Las opciones que se mostrarán, como podemos imaginar, son el contenido de una tabla de la base de datos.

Una función que produzca el código de un **select** podría poseer definidos, al menos, los parámetros **name** y **label**, además de un "paquete de datos" obtenido luego de realizar una consulta a la base de datos.

Nos anticipamos a un par de detalles que observaremos en este código:

- Hemos colocado **saltos de línea \n** donde finaliza cada renglón de código HTML, para que podamos leerlo con mayor claridad al mirar el código fuente recibido en nuestro navegador.
- Y hemos convertido al **juego de caracteres** UTF-8 los datos alfanuméricos recibidos de la base de datos, para mostrar los acentos y eñes sin problemas. Para ello, le hemos aplicado la función de PHP llamada utf8_encode a los datos alfanuméricos.

Hechas estas aclaraciones, veamos cómo podría ser una función que muestre datos dentro de un menú de selección:

```
<?php
function generaMenuSeleccion($datos,$name,$label){


      $codigo = '<label>'.$label.'</label>'."\n";


      $codigo = $codigo.'<select name="'.$name.'">'."\n";


      while($fila = mysqli_fetch_array($datos)){
         $codigo = $codigo.'<option
value="'.$fila["id"].'">'.utf8_encode($fila["pais"]).'</
option>'."\n";
      }


      $codigo = $codigo."</select>\n";
```

```
      return $codigo;
}
?>
```

En la página donde vayamos a llamar a esa función, recibiremos en una variable el código que fue acumulando la función internamente en su variable local $codigo, y allí quedará (en la variable **$codigoMenu**, en este ejemplo) hasta que decidamos usarlo. En este caso, inmediatamente lo mostramos con un **echo**.

Recordemos que ya contábamos con un **$paquete** de datos, que es lo que pasaremos como parámetro a la función:

```
$codigoMenu = generaMenuSeleccion($paquete,$name,$label);

echo $codigoMenu;
```

Estas dos líneas irían colocadas en la parte del código anterior donde dejamos el comentario que decía "// Aquí llamaremos a una función que muestre esos datos".

Lo cual producirá, en el caso de nuestra tabla de empleados, el siguiente código HTML:

```
<label>Empleados</label>
<select name="empleados">
      <option value="1">Pedro</option>
      <option value="2">José</option>
      <option value="3">Guillermo</option>
      <option value="4">Alberto</option>
      <option value="5">Luis</option>
      <option value="6">Claudio</option>
      <option value="7">Mario</option>
      <option value="8">Alan</option>
```

```
</select>
```

Vemos los valores de *label* y del *name* (proporcionados como **parámetros** al momento de llamar a la función), y luego los valores envueltos entre la apertura y cierre de cada etiqueta **option** (obtenidos gracias a la **consulta** a la base de datos).

Esta misma lógica, podemos aplicarla para envolver los datos obtenidos en cualquier otra etiqueta HTML. Veremos, a continuación, algunos ejemplos más.

## Generando tablas, filas y datos de tabla

Un uso muy común de los datos obtenidos de una consulta a una base de datos es mostrar- los en una **tabla**.

Como sabemos, una tabla consiste en una etiqueta **table** que envuelve todo el contenido de la tabla, una etiqueta *tr* (*table row*, o fila de tabla) que envuelve a cada fila horizontal, y una etiqueta *td* (*table data*, o dato de tabla) que envuelve a cada celda que contenga datos. Tendremos que generar estas tres partes con nuestro código PHP.

Veamos cómo podría ser una función que mostrará datos en una tabla:

```
<?php
function tabular($datos){

      // Abrimos la etiqueta table una sola vez:
      $codigo = '<table border="1" cellpadding="3">';

      // Vamos acumulando de a una fila "tr" por vuelta:
      while ( $fila = @mysqli_fetch_array($datos) ){

         $codigo .= '<tr>';

         /* Vamos acumulando tantos "td" como sea
necesario: */
         $codigo .='<td>'.utf8_encode($fila["id"]).'</td>';
         $codigo .=
'<td>'.utf8_encode($fila["nombre"]).'</td>';
```

```
            $codigo .=
'<td>'.utf8_encode($fila["apellido"]).'</td>';
            $codigo .=
'<td>'.utf8_encode($fila["edad"]).'</td>';
            $codigo .=
'<td>'.utf8_encode($fila["pais"]).'</td>';
            $codigo .=
'<td>'.utf8_encode($fila["especialidad"]).'</td>';

            // Cerramos un "tr":294

            $codigo .= '</tr>';
         }

         /* Finalizado el bucle, cerramos por única vez la
tabla: */
         $codigo .= '</table>';

         return $codigo;
}
```

Podemos usarla igual que la función anterior, a partir de un paquete de datos ya obtenido y asignando el código que fabrica a una variable, que luego mostramos con **echo**.

## Los botones de radio y las casillas de verificación

Los últimos casos que veremos son los de los botones de radio y las casillas de verificación.

Comencemos por los botones de radio: recordemos que para ser parte de una misma serie excluyente de opciones, deben compartir el **mismo valor** en el atributo **name**, aunque cada botón tenga su propio **id** único. Y que además de un **fieldset** y **legend** descriptivo del conjunto completo, cada botón debe tener su propio **label**:

```
<?php
```

```
function crearRadios($datos,$leyenda,$name){

      // Abrimos el fieldset con su leyenda:
      $codigo =
'<fieldset><legend>'.$leyenda.'</legend>'."\n";

      // Vamos mostrando un label y un input por vuelta:
      while ( $fila = @mysqli_fetch_array($datos) ){

         // Un label:
         $codigo .= '<label>'.utf8_encode($fila["nombre"]);

         // Un input:
         $codigo = $codigo.'<input type="radio"
name="'.$name.'" id="dato'.$fila["id"].'">'."\n";

         // Cerramos el label:

         $codigo .= '</label><br>'."\n";
      }

      $codigo .= '</fieldset>'."\n";

      return $codigo;
}
?>
```

Podríamos llamarla dentro del mismo archivo que realizaba la conexión y ejecutaba una consulta obteniendo un paquete de datos, de la siguiente manera:

```
$radios = crearRadios($paquete,'Votemos el empleado del
mes','empleado');


echo $radios;
```

Por último, las casillas de verificación: como sabemos, en ellas cada **name** debe ser diferente, por lo que lo formaremos uniendo a una parte escrita de manera fija el identificador de cada registro, como hicimos con los **id** en el ejemplo anterior, de los botones de radio:

```
<?php
function crearCasillas($datos,$leyenda){

      // Abrimos el fieldset con su leyenda:
      $codigo = '<fieldset><legend>'.$leyenda.'</
legend>'."\n";

      // Vamos mostrando un label y un input por vuelta:
      while ( $fila = @mysqli_fetch_array($datos) ){

         // Un label:
         $codigo .= '<label>'.utf8_encode($fila["nombre"]);

         // Un input:
         $codigo = $codigo.'<input type="checkbox"
name="dato'.$fila["id"].'"
id="dato'.$fila["id"].'">'."\n";

         // Cerramos el label:
         $codigo .= '</label><br>'."\n";
      }

      $codigo .= '</fieldset>'."\n";

      return $codigo;
}
?>
```

Y la llamaríamos de la siguiente manera:

```
$casillas = crearCasillas($paquete,'Tu voto puede sumarse
a más de un empleado','empleado');
```

```
echo $casillas;
```

Será muy útil seguir creando tantas funciones como consideremos necesario, para mostrar datos de distintas formas dentro de etiquetas HTML, ya que las necesitaremos continuamente en nuestro trabajo con bases de datos.

El objetivo de la creación de todas estas funciones es **organizar** nuestro código para que no quede suelto (que lo vuelve muy difícil de reutilizar). Podemos notar cómo la cantidad de código suelto dentro de nuestras páginas es cada vez menor, consistiendo en apenas un par de **includes** (del archivo de "datos" y del de "funciones"), algunos condicionales que validan datos y deciden cuál función ejecutar, y nada más.

Esta forma de programación modularizada o estructurada en funciones, facilita muchísimo el mantenimiento, y hace que futuros cambios (siempre imprevisibles) no nos obliguen a modificar las "páginas" una por una, sino solo el archivo donde esté la **declaración** de nuestras funciones.

Con esto, ya dominamos suficientes técnicas para enviar datos desde la base hacia nuestras páginas. Llega el momento de agregar interactividad y hacer el recorrido inverso, llevando datos desde la pantalla del usuario hacia nuestra base de datos, tema del que nos ocuparemos en el siguiente capítulo.

# Llevando datos de las páginas a la base

# 13

## Cómo escribir datos en una base desde PHP

Ya hemos realizado páginas que **leen** datos de nuestra base de datos pero, muchas veces, necesitaremos transitar el camino inverso, es decir, que nuestras páginas PHP agreguen datos a una base.

Serán, básicamente, dos los tipos de usuario que necesitarán agregar registros a una tabla mediante formularios HTML/PHP:

1. Los **administradores** del contenido de un sitio Web, cuando ingresen a alguna página privada (protegida con usuario y clave, parte del panel de administración o *back- end* del sitio), para –por ejemplo– agregar productos a un catálogo, noticias a un portal y tareas similares. El objetivo de que los administradores agreguen esa información, es que luego sea visualizada por los visitantes que entren al sitio, es decir, que la naveguen utilizando páginas dinámicas como las que aprendimos a crear en el capítulo anterior, cuyo contenido estaba almacenado en la base de datos.
2. Los **usuarios** de nuestro sitio también podrán agregar en ciertas circunstancias datos a nuestra base, cuando envíen un comentario de una noticia, un mensaje en un foro, completen sus datos en un formulario de registro, es decir, utilicen páginas HTML para escribir algo y enviarlo hacia el servidor desde el *front- end* (las páginas "públicas" de nuestro sitio).

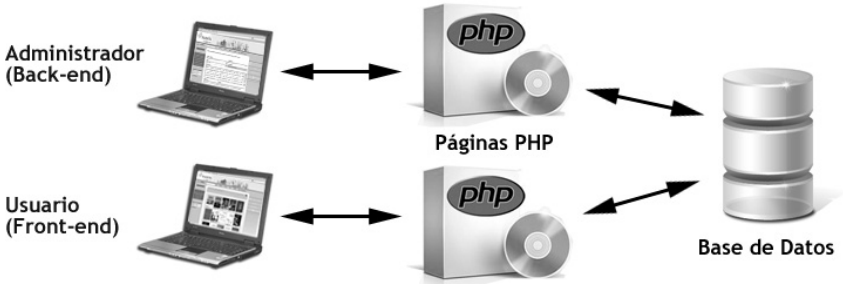


**Figura 13-1. El administrador interactúa con el *back-end,* y el usuario, con el *front-end.***

Sea quien fuere el tipo de usuario que agregará datos a nuestra base, lo hará utilizando una misma técnica: un **formulario HTML** para escribir los datos del lado del cliente (navegador) y, en la página de destino de ese formulario, un **código PHP** que, al estar ubicado del lado del servidor, podrá insertar los datos en la base y luego devolver alguna respuesta al navegador.

Entonces, estamos hablando de un proceso que tiene dos etapas o "momentos":

- el momento inicial donde el usuario completa el **formulario**, del lado del cliente;
- y, el segundo, cuando una página PHP recibe en el servidor las variables que el usuario completó y las utiliza para ejecutar una consulta SQL que **inserta los datos** en la base.

Típicamente, este proceso se dividirá en **dos páginas distintas:** un archivo HTML para el formulario y una página PHP para el código que se ejecutará en el servidor, insertará el dato en la base y mostrará un mensaje de éxito o error.

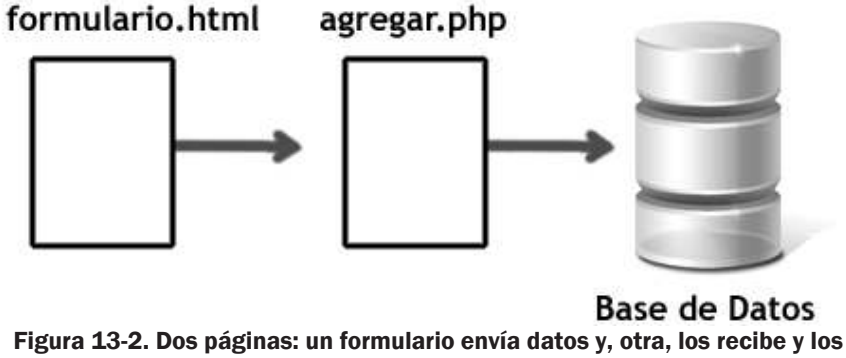


**Figura 13-2. Dos páginas: un formulario envía datos y, otra, los recibe y los inserta en la base.**

En la segunda página (agregar.php), para insertar los registros en la base, los pasos necesarios serán:

1. Que el programa intérprete de PHP **se identifique** ante el programa gestor de MySQL y **seleccione una base** (ya hemos creado una función que hacía esto en el capítulo anterior, así que podremos reutilizarla).
2. Prepararemos en una variable **la orden del lenguaje SQL** necesaria para insertar datos en la base.
3. **Ejecutaremos esa orden** SQL (no será necesario crear una función específica para insertar datos, ya que esta consulta no trae datos, sino que los envía hacia la base, por lo cual no es necesario generar ni recorrer luego ningún "paquete de datos"). Ejecutaremos la orden con mysqli_query directamente, y será clave evaluar **si devolvió verdadero** o no la ejecución de esa consulta y, en función de ello, mostraremos un mensaje de éxito (si el dato fue insertado) o de error.

Pasemos a la acción:

## Creando el formulario

Como primera medida, crearemos un **formulario** que nos dejará escribir los datos que queremos agregar a la tabla de "mensajes". Este formulario tendrá dos campos de texto, para el **nombre** y el **email**, respectivamente, y un área de texto para el *mensaje* (el **id** no lo insertará el usuario, ya que es un campo *auto-increment*).

A este formulario lo llamaremos formulario.html, y su código podría ser algo así:

```
<form method="post" action="agregar.php">
<fieldset>
<legend>Ingrese su consulta</legend>
      <p>
      <label>Escriba su nombre:
      <input type="text" name="nombre">
      </label>
      </p>
      <p>
      <label>Escriba su correo electrónico:
      <input type="text" name="email">
```

```
      </label>
      </p>
      <p>
      <label>Escriba su mensaje:
      <textarea name="mensaje" cols="30"
rows="5"></textarea>
      </label>
      </p>
      <p>
      <input type="submit" value="Enviar">
      </p>
</fieldset>
</form>
```

Además de prestar atención a los **name** (ya que serán los nombres de las variables que podremos leer en la siguiente página desde una celda de la matriz $_POST), lo más importante aquí es que el atributo **action** del formulario está apuntando hacia una segunda página, llamada agregar.php; que será la que codificaremos a continuación.

## Conexión a MySQL

En la segunda página (es decir, aquella página hacia la cual apuntaba el **action** del formulario anterior y que decidimos denominar agregar.php), luego de validar si se recibieron las variables, el intérprete de PHP **intentará conectarse** al programa MySQL, y tratará de **seleccionar una base**.

Para estas tareas, ya habíamos creado una función en el capítulo anterior, así que vamos a reutilizarla:

```
<?php
// Incluimos los datos de conexión y las funciones:
include("datos.php");
include("funciones.php");

// Usamos esas variables:
if ( $con=conectarBase($host,$usuario,$clave,$base) ){
```

```
      // Aquí irá el resto

} else {

      echo "<p>Servicio interrumpido</p>";

}
?>
```

Por supuesto, recordemos colocar los archivos datos.php y funciones.php en la misma carpeta en la que estamos trabajando para que esto pueda funcionar.

Ahora, pasemos al segundo paso.

## La orden INSERT del lenguaje SQL

Una vez autenticados ante MySQL y seleccionada la base, prepararemos, dentro de una variable, la consulta del lenguaje SQL que se utiliza para insertar datos en una tabla de la base: la orden INSERT.

La sintaxis básica de la orden INSERT es la siguiente:

```
INSERT INTO tabla (campo 1, campo2, campo3) VALUES
('valor1', 'valor2', 'valor3');
```

Esto podría traducirse como "Insertar en la tabla indicada, en los campos enumerados, los valores siguientes".

Veamos un ejemplo para entenderlo mejor:

```
INSERT INTO mensajes (id, nombre, email, mensaje)
VALUES     ('0','Hernán','hernan@beati.com.ar','Bla,
bla, bla')
```

Observemos que los **valores** (enumerados entre paréntesis luego de la palabra VALUES) siempre van envueltos entre **comillas simples**, y que existe una

correlación de **orden**, ya que se insertará, en el primer campo de la lista (el campo **id**, en este caso), el primero de los *value* (el 0); en el segundo (nombre), el segundo *value* (Hernán), y así sucesivamente.

Notemos también que hemos proporcionado un **cero** como valor para el campo id, ya que será generado automáticamente (lo hemos definido como *auto-increment*). De esta manera, nos despreocuparemos y, para todos los registros que vayamos a insertar, siempre proporcionaremos un cero como valor del id, y será MySQL quien se ocupe de asignarle el valor correspondiente (por supuesto, todo esto gracias a que hemos definido al campo *id* como *auto-increment*).

Recordemos que podemos ejecutar consultas SQL desde el phpMyAdmin, pulsando en la solapa SQL, escribiendo órdenes en lenguaje SQL dentro del área de texto, y luego pulsaremos **Continuar** para ver el resultado de la ejecución. Es recomendable que probemos varias veces esta orden INSERT para familiarizarnos con ella.

A esta orden INSERT, del lenguaje SQL, la dejaremos escrita dentro de una **variable**, para que la podamos pasar como parámetro a una función que se encargue de ejecutarla.

Pero no vamos a escribir allí los datos "a mano", como en el ejemplo anterior, sino que deberemos modificar esta orden para que realmente reciba los datos enviados desde el formulario de la primera página, por lo cual quedará algo como lo que sigue:

```
<?php
// Validamos que hayan llegado estas variables, y que no
estén vacías:

if (
isset($_POST["nombre"],$_POST["email"],$_POST["mensaje"])
and $_POST["nombre"]!="" and $_POST["email"]!="" and ]
$_POST["mensaje"]!="" ){

      /* Traspasamos a variables locales, para evitar
complicaciones con las comillas: */

      $nombre = $_POST["nombre"];
      $email = $_POST["email"];
      $mensaje = $_POST["mensaje"];
```

```
      // Preparamos la oden SQL:
      $consulta = "INSERT INTO mensajes
(id,nombre,email,mensaje)
VALUES('0','$nombre','$email','$mensaje')";

      // Aquí ejecutaremos esa orden

} else {

      echo '<p>Por favor, complete el <a
href="formulario.html">formulario</a></p>';
}
?>
```

Lo más importante aquí es que verificamos la presencia de las variables, y las dejamos preparadas, listas para usar dentro de $consulta.

Es muy útil, llegados a este punto previo a la ejecución de la consulta, que probemos **qué contiene** la variable $consulta haciendo un **echo** de ella. Podemos completar algunos datos de prueba en el formulario, enviarlo, y ver qué muestra el **echo**; a eso que muestre, podemos copiarlo y pegarlo dentro de la ventana SQL del phpMyAdmin para, de esta manera, confirmar si la sintaxis de la orden fue la correcta.

Recordemos que es necesario que incluyamos este bloque de código dentro del bloque anterior que realizaba la conexión a la base, en la parte exacta en la que habíamos dejado preparado un comentario que decía: "// Aquí irá el resto".

### Ejecutar la consulta

Ahora que hemos verificado que la sintaxis de la consulta es la correcta, ya podemos hacer que PHP la **ejecute realmente**, mediante la utilización de la conocida función mysql_query, y, de la misma manera que en el capítulo anterior, ubicaremos esta consulta dentro de un condicional para comprobar si verdaderamente se insertó el registro.

Observemos que el siguiente código debe ser intercalado en el punto del código anterior en el que dejamos un comentario que decía:

"// Aquí ejecutaremos esa orden".

```
if ( mysqli_query($con,$consulta) ){

      echo "<p>Registro agregado.</p>";

} else {

      echo "<p>No se agregó...</p>";

}
```

Si luego de ejecutado esto, entramos al phpMyAdmin, pulsamos en la columna izquierda sobre el nombre de la base de datos **cursos** y luego pulsamos el nombre de la tabla **mensajes**, ahora podremos hacer *clic* en **Examinar** y veremos el nuevo registro que se ha agregado a nuestra tabla.

Sin embargo, si por casualidad escribimos en el texto de alguno de los campos una **tilde** o **eñe**, notaremos que se visualiza con una codificación incorrecta. Siempre que enviamos una consulta que lleva datos **hacia** una base de datos, debemos especificar el juego de caracteres, que será *UTF-8,* en nuestro caso, para no tener problemas con los **caracteres especiales** (acentos, eñes). Esto lo haremos mediante la orden SET NAMES.

En este caso, la aplicaremos luego de establecida la conexión y seleccionada la base (inmediatamente después del condicional que evalúa la función conectarBase), de esta forma:

```
@mysqli_query($con, "SET NAMES 'utf8'");
```

Con esto, el proceso completo de agregar registros a una tabla desde un formulario ya funciona pero, antes de usarlo en nuestros sitios, analizaremos un típico problema de seguridad (y veremos cómo solucionarlo) de este proceso de agregar a la base información que fue ingresada por algún usuario.

## Filtrar los datos de formularios: evitando inyección SQL

Un punto fundamental cuando vamos a enviar datos escritos por el usuario dentro de una consulta SQL, es "limpiarlos" antes de usarlos en una consulta que

los envíe hacia MySQL, ya que algún usuario con malas intenciones puede enviar datos **dañinos** para nuestra base de datos o, simplemente, lograr el **acceso** a zonas de nuestro sitio que no son públicas, utilizando una técnica muy popular, que analizaremos a continuación para poder prevenirla.

A esta técnica se la conoce como SQL *injection* (inyección SQL), y consiste en utilizar un campo de un formulario para "complementar" una consulta SQL (SELECT, INSERT, DELETE o UPDATE). Por supuesto, los atacantes conocen el lenguaje SQL muy bien.

Veamos un ejemplo de esta técnica en funcionamiento, para que podamos comprobar el riesgo real de no "limpiar" los datos escritos por los usuarios.

Vamos a crear una tabla llamada "usuarios", que tendrá solo tres campos: **id** (TINYINT), **usuario** y **clave** (ambos VARCHAR). Cargaremos un solo registro, un supuesto usuario "pepe" con la clave "clave".

Una vez creada esta sencilla tabla e insertado ese registro de prueba, crearemos un archivo "formulario.html" en el que se pueda ingresar un usuario y clave para acceder a un supuesto contenido secreto:

```
<form method="post" action="secreto.php">
<fieldset>
<legend>Ingrese sus datos</legend>
      <p>
      <label>Escriba su Usuario:
      <input type="text" name="usuario">
      </label>
      </p>
      <p>
      <label>Escriba su Contraseña:
      <input type="text" name="clave">
      </label>
      </p>
      <p>
      <input type="submit" value="Enviar">
      </p>
</fieldset>
</form>
```

En la segunda página (secreto.php), incluiremos, como siempre, los archivos de conexión y de funciones, y ejecutaremos una consulta SELECT para

ver si el usuario y la clave ingresados por el usuario existen en nuestra tabla de usuarios: si no existen, no les dejaremos ver el contenido "secreto" de nuestra página.

Veamos cómo sería el código de secreto.php (hemos retomado el ejemplo del capítulo anterior, por lo que es imprescindible que coloquemos todos los archivos necesarios en la misma carpeta, como datos.php y funciones.php):

```
<?php
// Incluimos los datos de conexión y las funciones:
include("datos.php");
include("funciones.php");

// Validamos que hayan enviado un usuario y una clave, y
que no estén vacíos:
if ( isset($_POST["usuario"],$_POST["clave"]) and $_
POST["usuario"]<>"" and $_POST["clave"]<>""){

      // Traspasamos a variables locales:
      $usuarioIngresado = $_POST["usuario"];
      $password = $_POST["clave"];

      // Nos conectamos:
      if ( conectarBase($host,$usuario,$clave,$base) ){

       $consulta = "SELECT * FROM usuarios WHERE
       usuario='$usuarioIngresado' AND clave='$password'";

       if ( $paquete = consultar($consulta) ){

          echo "<p>Bienvenido al contenido secreto</p>";

      } else {

         echo "<p>No tiene derecho a acceder a nuestro
contenido secreto</p>";

      }
```

```
} else {

      echo "<p>Servicio interrumpido</p>";

}

} else {

      echo '<p>No ha completado el formulario.</p>';


}

echo '<p>Regresar al <a
href="formulario.html">formulario</a></p>';
?>
```

Si probamos lo expresado en el cuadro anterior escribiendo un usuario y clave que no coincidan con “pepe” y “clave”, no obtendremos acceso al contenido secreto, pero sí obtendremos acceso si escribimos “pepe” y “clave”. Hasta aquí, todo aparenta funcionar a la perfección...

Pero observemos qué intentará hacer un usuario que quiera ingresar a nuestro contenido secreto. Estando frente al formulario, escribirá esto:

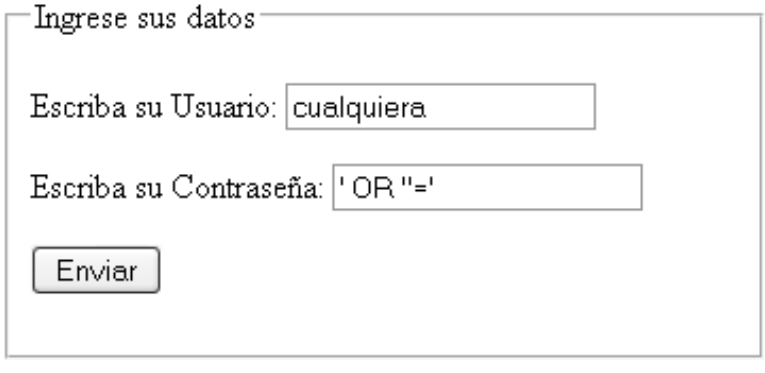

**Figura 13-3. Inyección de código SQL.**

Notemos que el contenido de ese campo será:

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

Lo cual llega a ser significativo si reemplazamos los valores de las variables por lo ingre- sado por el usuario, y lo colocamos dentro de la orden SQL que será ejecutada a continuación:

```
$consulta = "SELECT * FROM usuarios WHERE
usuario='cualquiera' AND clave='' OR ''=''";
```

Notemos que se modifica radicalmente la consulta; ya no importa qué haya sido ingresado en el campo usuario (puede ser cualquier cosa), ni importa que se haya dejado vacía la clave, ya que con su primera comilla simple, ha cerrado la comilla que quedaba abierta para la clave, pero luego ha agregado: OR "=" que es una afirmación verdadera, también podría haber puesto: OR 1=1, o cualquier otra tautología que provoque que la expresión completa sea evaluada como **verdadera**. El punto de que sea evaluada como verdadera, es que, recordemos... está siendo ejecutada dentro de un condicional, que es la única manera de saber si "existían" ese usuario y esa clave en la tabla de usuarios. Conclusión: este condicional dará **siempre verdadero** para ese usuario, que podrá entrar libremente a nuestra zona "secreta", simplemente inyectando el mencionado código en nuestro formulario de acceso.

¿Cómo podemos evitarlo? Muy fácilmente: contamos con una función denominada mysqli_real_escape_string que nos permite sanear los datos ingresados por el usuario, es decir, cambiar su valor por otro que resulte inofensivo para las consultas SQL que se enviarán hacia la base de datos.

Solo debemos aplicarla a las variables recibidas y desaparece el riesgo de inyección SQL:

```
$usuario =
mysqli_real_escape_string($_POST["usuario"]);
$password =
mysqli_real_escape_string($_POST["clave"]);
```

Si por curiosidad completamos nuevamente el formulario con **' OR "='** y luego hacemos un *echo* de $password para ver qué contiene, veremos que las comillas fueron desactivadas ("escapadas" con una barra invertida delante):

```
\' OR \'\'=\'
```

Esta operación lo vuelve completamente inofensivo debido a que ya no respeta la sintaxis del lenguaje SQL y, por lo tanto, el usuario que intentó esta inyección SQL ya no podrá acceder a nuestro contenido secreto.

Ahora que ya podemos **insertar** datos en la base desde nuestras páginas PHP con total seguridad, aprenderemos a **eliminar** un registro de la base mediante formularios (tarea típica de un panel de administración o *back-end*, desde el cual, por ejemplo, podríamos eliminar un mensaje publicado en un foro que sea ofensivo o que realice SPAM, eliminar un producto agotado, que ya no esté en venta en un catálogo, etc.).

# Cómo eliminar datos de una base con PHP

A continuación, veremos los pasos de la operación que **elimina** un registro entero de nuestra tabla desde una página Web.

Retomemos el ejemplo del capítulo anterior (recordemos ubicar los archivos datos.php y funciones.php en la misma carpeta).

La parte que mostraba el listado de empleados era como sigue:

```
<?php
// Incluimos los datos de conexión y las funciones:
include("datos.php");
include("funciones.php");

// Usamos esas variables:
if ( $con = conectarBase($host,$usuario,$clave,$base) ){

      $consulta = "SELECT * FROM empleados";

      if ( $paquete = consultar($con, $consulta) ){

        // Llamamos a una función que muestre esos datos
        $codigoTabla = tabular($paquete);
        echo $codigoTabla;

      } else {
```

```
        echo "<p>No se encontraron datos</p>";

      }

} else {

      echo "<p>Servicio interrumpido</p>";

}
?>
```

La parte clave del bucle de nuestra función llamada tabular, donde se armaba el código HTML de cada renglón o tr del listado, era la siguiente:

```
while ( $fila = @mysqli_fetch_array($datos) ){
      $codigo .= '<tr>';
      // Vamos acumulando tantos "td" como sea necesario:
      $codigo .= '<td>'.utf8_encode($fila["id"]).'</td>';
      $codigo .= '<td>'.utf8_encode($fila["nombre"]).
      '</td>';
      $codigo .= '<td>'.utf8_encode($fila["apellido"]).
      '</td>';
      $codigo .= '<td>'.utf8_encode($fila["edad"]).
      '</td>';
      $codigo .= '<td>'.utf8_encode($fila["pais"]).'</td>';
      $codigo .= '<td>'.utf8_encode($fila["especialidad"]).
      '</td>';
      // Cerramos un "tr":
      $codigo .= '</tr>';
}
```

Esto producía el siguiente resultado:

| | | | | | |
|---|---|---|---|---|---|
| 1 | Pedro | Fernández | 34 | España | Matemáticas |
| 2 | José | García | 28 | México | Sistemas |
| 3 | Guillermo | Pérez | 32 | España | Contabilidad |
| 4 | Alberto | Maza | 45 | México | Matemáticas |
| 5 | Luis | Puente | 43 | Argentina | Sistemas |
| 6 | Claudio | López | 41 | España | Medicina |
| 7 | Mario | Juárez | 51 | México | Sistemas |
| 8 | Alan | Flores | 25 | Perú | Sistemas |

**Figura 13-4. Listado original.**

Para agregar a este listado tanto la operación de **borrar** como la de **modificar** un registro, agregaremos un par de nuevas columnas *td* a cada fila, para que incluyan cada una un enlace que muestre las palabras **BORRAR** y **MODIFICAR**, respectivamente.

Para ello, agregaremos al bucle anterior dos nuevos **td** al final de los demás, de esta manera:

```
$codigo .= '<td>BORRAR</td>';
$codigo .= '<td>MODIFICAR</td>';
```

Este agregado, por el momento, se verá así:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | Pedro | Fernández | 34 | España | Matemáticas | BORRAR | MODIFICAR |
| 2 | José | García | 28 | México | Sistemas | BORRAR | MODIFICAR |
| 3 | Guillermo | Pérez | 32 | España | Contabilidad | BORRAR | MODIFICAR |
| 4 | Alberto | Maza | 45 | México | Matemáticas | BORRAR | MODIFICAR |
| 5 | Luis | Puente | 43 | Argentina | Sistemas | BORRAR | MODIFICAR |
| 6 | Claudio | López | 41 | España | Medicina | BORRAR | MODIFICAR |
| 7 | Mario | Juárez | 51 | México | Sistemas | BORRAR | MODIFICAR |
| 8 | Alan | Flores | 25 | Perú | Sistemas | BORRAR | MODIFICAR |

**Figura 13-5. Listado con palabras agregadas.**

Pero ahora necesitamos que, al pulsar una de estas palabras, cada una de ellas "envíe" hacia el servidor en un enlace (que aún no hemos hecho) una variable con el **código** del registro que deseamos borrar o modificar. Ésta es la forma para que la página de destino sepa "a cuál registro" se está queriendo eliminar o modificar.

Comencemos convirtiendo la palabra **BORRAR** en un simple enlace HTML, que apunte hacia una página que denominaremos borrar.php:

```
$codigo                    .=                    '<td><a
href="borrar.php">BORRAR</a></td>';
```

Al implementar este cambio, el listado se verá así:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | Pedro | Fernández | 34 | España | Matemáticas | BORRAR | MODIFICAR |
| 2 | José | García | 28 | México | Sistemas | BORRAR | MODIFICAR |
| 3 | Guillermo | Pérez | 32 | España | Contabilidad | BORRAR | MODIFICAR |
| 4 | Alberto | Maza | 45 | México | Matemáticas | BORRAR | MODIFICAR |
| 5 | Luis | Puente | 43 | Argentina | Sistemas | BORRAR | MODIFICAR |
| 6 | Claudio | López | 41 | España | Medicina | BORRAR | MODIFICAR |
| 7 | Mario | Juárez | 51 | México | Sistemas | BORRAR | MODIFICAR |
| 8 | Alan | Flores | 25 | Perú | Sistemas | BORRAR | MODIFICAR |

**Figura 13-6. Listado con enlace en BORRAR.**

Ahora viene lo más difícil; necesitamos que cada uno de esos enlaces, al ser generado dentro del bucle, obtenga de $fila["id"] el código del registro que se

está armando en ese momento, y lo deje escrito dentro del enlace, en una variable que se enviará por el método *get* hacia la siguiente página:

```
$codigo .= '<td><a href="borrar.php?codigo='.$fila["id"].
'">BORRAR</a></td>';
```

Que producirá el siguiente resultado:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | Pedro | Fernández | 34 | España | Matemáticas | BORRAR | MODIFICAR |
| 2 | José | García | 28 | México | Sistemas | BORRAR | MODIFICAR |
| 3 | Guillermo | Pérez | 32 | España | Contabilidad | BORRAR | MODIFICAR |
| 4 | Alberto | Maza | 45 | México | Matemáticas | BORRAR | MODIFICAR |
| 5 | Luis | Puente | 43 | Argentina | Sistemas | BORRAR | MODIFICAR |
| 6 | Claudio | López | 41 | España | Medicina | BORRAR | MODIFICAR |
| 7 | Mario | Juárez | 51 | México | Sistemas | BORRAR | MODIFICAR |
| 8 | Alan | Flores | 25 | Perú | Sistemas | BORRAR | MODIFICAR |


**Figura 13-7. Cada enlace con su identificador.**

Visualmente, nada ha cambiado en los enlaces, pero si apoyamos el puntero del mouse sobre cada una de las palabras **BORRAR** (sin pulsarlo, solo deslizándolo por encima de cada enlace), notaremos en la **barra de estado**, al pie y a la izquierda del navegador, que cada enlace además de apuntar hacia borrar.php, al ser pulsado enviará también la variable "código" con su correspondiente valor.

Notemos que si pasamos el mouse sobre las distintas palabras **BORRAR** a lo largo de todos los renglones, veremos que en cada renglón el valor de la variable "codigo" es **diferente**, ya que se corresponde con el valor de $fila["id"] de cada registro. De esta manera, la página de destino podrá saber cuál es el registro que se desea eliminar, con total certeza.

Ahora, solo nos resta el final: crear la página borrar.php hacia la que apuntan todos esos enlaces, que es la página que realmente borrará el registro cuyo id coincida con el indicado en la variable "codigo".

## La orden DELETE del lenguaje SQL

Dentro de la página borrar.php, deberíamos estar recibiendo desde el listado anterior la variable "codigo", que contiene el **id** de alguno de los registros, el que se desea borrar.

Por supuesto, deberemos validar si ha llegado al servidor y si no está vacía. Esto se podría hacer de la siguiente manera:

```
<?php
if ( isset($_GET["codigo"]) and $_GET["codigo"]<>"" ) {

      // Aquí irá el resto de las instrucciones

} else {

    echo '<p>No especificó qué desea borrar, por favor
      regrese al <a href="listado.php">listado</a></p>';
}
?>
```

Si está presente la variable "codigo", procederemos a eliminar el registro que posea el id que tiene especificado y, para ello, deberemos seguir los siguientes pasos:

1. Que el programa intérprete de PHP se **identifique** ante MySQL y **seleccione** una base (ya tenemos preparada una función que hace esto).
2. Preparar en una variable la **consulta** SQL que ejecutaremos, que esta vez consistirá en la orden DELETE.
3. **Ejecutar** esa orden (con mysqli_query, nada novedoso para nosotros, y al igual que en la inserción, tampoco devolverá datos, por lo que será suficiente con evaluar con un condicional el resultado obtenido).
4. Según el resultado del condicional, mostraremos un **mensaje** y permitiremos al usuario que regrese al listado, para seguir realizando otras tareas.

El único punto diferente a lo que ya sabemos es el segundo, así que veremos cuál es la sintaxis de la orden DELETE que nos permitirá eliminar registros de una tabla.

La estructura de esta orden del lenguaje SQL es:

```
DELETE FROM tabla WHERE campo='valor';
```

Notemos que el valor va entre comillas simples, como todos los valores que enviamos hacia MySQL, y también notemos el agregado de una condición WHERE. Esta condición es **imprescindible** en el caso de una orden DELETE, ya que permite restringir la aplicación de la orden a apenas uno o varios registros que cumplan con esa condición, en caso contrario, de no especificar ninguna condición, borraríamos **todos** los registros de la tabla completa.

> Atención: esta orden es la más peligrosa del lenguaje SQL: DELETE FROM tabla; Al no poseer una condición WHERE, elimina todos los registros de la tabla especificada.

Por lo tanto, siempre que hagamos un DELETE agregaremos un WHERE para limitar los registros que se eliminarán. En este caso, deseamos eliminar solo uno, aquel registro que en el campo ***id*** posea el valor especificado en la variable "codigo", variable que el usuario ha enviado desde el listado al pulsar "ese" enlace, el correspondiente a "ese" registro que desea borrar.

Por lo tanto, en este ejemplo, nuestra orden SQL quedaría así armada:

```
DELETE FROM empleados WHERE id='$_GET["codigo"]';
```

Orden que colocaremos, como siempre, dentro de una variable llamada $consulta. Pero ahora nos surge un ligero inconveniente: al envolver entre comillas **dobles** el contenido completo de la variable, entrarán en conflicto con las comillas dobles que envuelven a "codigo". Tenemos dos maneras de remediarlo:

- Envolver entre **llaves** el valor que se enviará a MySQL, es decir, la parte final de la orden DELETE que está envuelta entre comillas simples, en este caso: $_GET["codigo"]
- O traspasar a una variable **local** (que no contiene comillas) el contenido de $_GET["codigo"].

En el primer caso (usando llaves), la variable $consulta quedaría así:

```
$consulta="DELETEFROMempleadosWHEREid='{$_GET["codi
go"]}'";
```

En el segundo caso (traspasando todo a una variable local), quedaría de la siguiente manera:

```
$codigo = $_GET["codigo"];
$consulta = "DELETE FROM empleados WHERE
id='$codigo'";
```

A continuación, ejecutaremos esta orden con mysqli_query, evaluando con un condicional si devolvió *true* (verdadero) la orden de eliminación (es decir, si MySQL pudo eliminar el registro indicado) y, en ese caso, mostraremos al usuario

un mensaje de “Registro eliminado”; caso contrario, mostraremos un mensaje de error.

Por esta razón, el código completo de la página borrar.php quedaría así:

```
<?php
// Incluimos los datos de conexión y las funciones:
include("datos.php");
include("funciones.php");

// Verificamos la presencia del código esperado:
if ( isset($_GET["codigo"]) and $_GET["codigo"]<>"" ){

      // Nos conectamos:
      if ( $con = conectarBase($host,$usuario,$clave,$base)
){

      /* Traspasamos a una variable local para evitar
problemas con las comillas: */
      $codigo = $_GET["codigo"];

      $consulta = "DELETE FROM empleados WHERE
id='$codigo'";
      if ( mysqli_query($con,$consulta) ){
         echo "<p>Registro eliminado.</p>";
      } else {
         echo "<p>No se pudo eliminar</p>";
      }
   } else {
      echo "<p>Servicio interrumpido</p>";
   }
} else {
```

```
      echo '<p>No se ha indicado cuál registro
eliminar.</p>';

}
echo '<p>Regresar al <a href="listado.php">listado
</a></p>';
?>
```

Esto ya está listo para que lo probemos, y eliminará el registro que le indiquemos al pulsar un enlace cualquiera del listado.

# Cómo modificar datos de una base con PHP

## El formulario más complicado: la actualización

Solo nos resta la acción más compleja (o, al menos, la más larga) de todas, la de actualizar o modificar los valores de un registro.

Reemplazaremos dentro de la función denominada **tabular** la línea que generaba el último **td** de la tabla, el que mostraba la palabra MODIFICAR. Nuestro objetivo es hacer que esa palabra se convierta en un enlace, que envíe un código hacia una nueva página que crearemos que se llamará modificar.php. Esto no nos debería resultar un problema: es lo mismo que hemos hecho antes para eliminar un registro, apenas cambia el nombre del archivo de destino del enlace y la palabra que se mostrará:

```
$codigo .= '<td><a href="modificar.php?codigo='.
$fila["id"].'">MODIFICAR</a></td>';
```

Que producirá el siguiente resultado:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | Pedro | Fernández | 34 | España | Matemáticas | BORRAR | MODIFICAR |
| 2 | José | García | 28 | México | Sistemas | BORRAR | MODIFICAR |
| 3 | Guillermo | Pérez | 32 | España | Contabilidad | BORRAR | MODIFICAR |
| 4 | Alberto | Maza | 45 | México | Matemáticas | BORRAR | MODIFICAR |
| 5 | Luis | Puente | 43 | Argentina | Sistemas | BORRAR | MODIFICAR |
| 6 | Claudio | López | 41 | España | Medicina | BORRAR | MODIFICAR |
| 7 | Mario | Juárez | 51 | México | Sistemas | BORRAR | MODIFICAR |
| 8 | Alan | Flores | 25 | Perú | Sistemas | BORRAR | MODIFICAR |

**Figura 13-8. Enlaces para modificar.**

Pero la complicación reside en que, para que el usuario pueda modificar a su gusto los valores de los campos, se requiere un paso adicional con respecto a la eliminación: no alcanza con saber el id del registro a "actualizar" únicamente, ya que no sabemos qué campos de ese registro querrá actualizar el usuario, ni qué nuevos valores le querrá dar. Por lo tanto, antes de actualizar, deberemos **mostrar** todos los datos de ese registro, pero dentro de atributos *value* de campos de un formulario, para que aparezcan escritos sus valores actuales y, una vez que el usuario modifique libremente lo que está viendo en su pantalla, esos datos deberán ser enviados hacia **otra página PHP**, que estará esperando estos nuevos valores en el servidor, y esa es la página que realmente ejecutará la orden de actualizar los valores de ese registro en la base.

Así que todo el proceso constará de tres páginas:

1. El **listado** con enlaces (listado.php).
2. La página que mostrará un **formulario con los datos** del registro que se modificará ya cargados, listos para que el usuario escriba, borre o reemplace lo que quiera (página que podemos llamar modificar.php).
3. Y una última página, que recibirá las variables de ese formulario ya modificado por el usuario, y **ejecutará una consulta** de actualización en la base (página que llamaremos modificado.php).


**Figura 13-9. Pedido de un formulario con los datos incluidos, listos para editar.**

Una vez modificados en su navegador, los datos se envían nuevamente al servidor (Fig. 13-10).

Como de costumbre, en la página a la que se llega desde el listado (reiteramos que decidimos llamarla modificar.php), validaremos que esté llegando un código y que no esté vacío.

A continuación, haremos un SELECT, tal como los que hicimos en el capítulo anterior, con el objetivo de traer desde la tabla **solamente el registro completo** que se desea modificar. Una vez obtenidos esos datos, deberemos "escribirlos" dentro de los atributos **value** de los campos de un formulario HTML. Para esto, crearemos una función propia, que podamos reutilizar con ligeras variantes en otros proyectos.

Veamos cómo hacer esta función necesaria para el archivo modificar.php, paso a paso.

Para seleccionar **solamente el registro** indicado, aplicaremos el mismo filtro que hemos usado recientemente en la orden DELETE: un condicional **WHERE**.

```
$consulta = "SELECT * FROM empleados WHERE id='$codigo'";
```

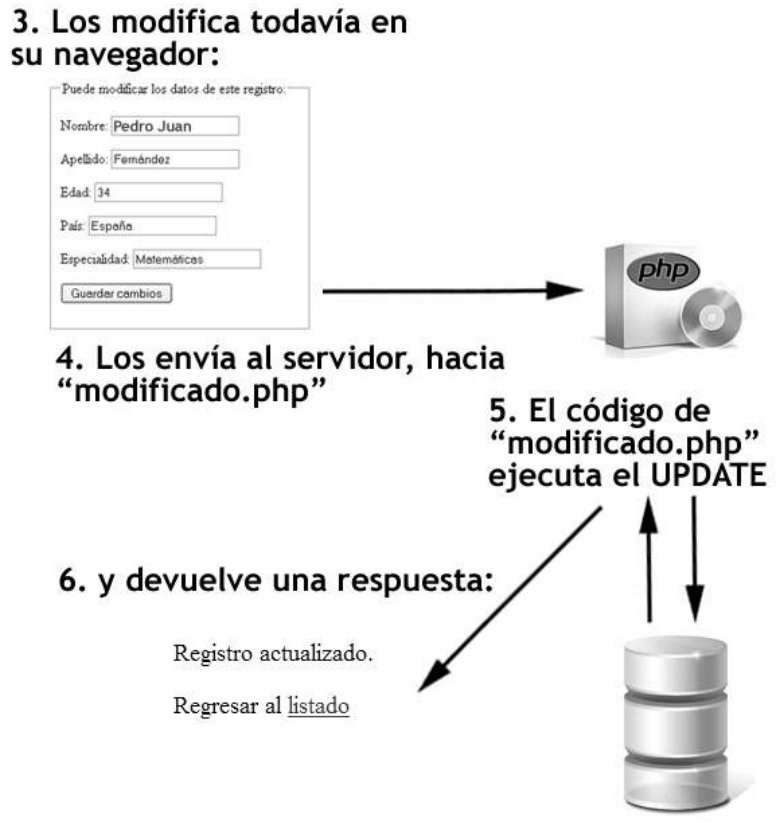


**Figura 13-10. Se envían los datos modificados y actualizan la base de datos.**

Esta consulta, una vez ejecutada con la función **consultar** que creamos con el objetivo de ejecutar órdenes SELECT (algo muy sencillo a estas alturas) nos traerá datos de una sola fila de la tabla, los que deberemos asignar a un "paquete" que llamaremos típicamente $datos.

El código de esta página denominada modificar.php hasta este punto, podría ser el siguiente:

```
<?php
// Incluimos los datos de conexión y las funciones:
include("datos.php");
include("funciones.php");

// Verificamos la presencia del código esperado:
```

```
if ( isset($_GET["codigo"]) and $_GET["codigo"]<>"" ){
$codigo = $_GET["codigo"];

      // Nos conectamos:
      if ( $con = conectarBase($host,$usuario,$clave,$base)
){

      $consulta = "SELECT * FROM empleados WHERE
id='$codigo'";

      if ( $paquete = consultar($con, $consulta) ){

         // Aquí llamaremos a una función que muestre esos
datos dentro de atributos value de un formulario

      } else {

         echo "<p>No se encontraron datos</p>";

      }

   } else {

      echo "<p>Servicio interrumpido</p>";

      }

} else {

      echo '<p>No se ha indicado cuál registro desea
modificar.</p>';

}

echo '<p>Regresar al <a href="listado.php">listado</a>
</p>';
?>
```

Ahora, crearemos una función que llamaremos **editarRegistro**, cuya tarea consistirá en cargar los valores de cada campo del registro seleccionado dentro de atributos **value** de campos de un formulario.

Para ello, la función necesitará recibir como parámetro el paquete de datos denominado $paquete, y deberá extraerlo a una matriz (que típicamente llamaremos $fila), usando la conocida función **mysqli_fetch_array**, tal como hicimos en el listado inicial, pero con una ligera variante: esta vez no será necesario un bucle, ya que la consulta sin duda traerá "un solo registro" (es imposible guardar dos o más registros con el mismo **id** en una tabla cuyo campo **id** fue declarado como clave primaria).

Por lo cual, ejecutaremos lo que sigue una sola vez:

```
$fila = mysqli_fetch_array($paquete);
```

Y ya tendremos cargada la matriz $fila con los datos completos de ese registro, tal como estaban guardados en la tabla.

Podríamos ir declarando las tareas básicas de esta nueva función **editarRegistro**:

```
<?php
function editarRegistro($datos){

      // Extraeremos a $fila el registro:
      if ($fila = mysqli_fetch_array($datos)){

         /* Aquí acumularemos en $codigo cada dato de $fila
ubicado dentro de atributos value de campos */

      } else {

         $codigo = false;

      }

      return $codigo;
}
?>
```

Ahora, viene la parte más complicada: nuestra función debe generar un **formulario HTML** que dentro de sus campos muestre escritos todos los datos que actualmente están dentro de la matriz $fila.

Veamos cómo hacerlo.

La forma más cómoda de crear este formulario es generar su código mediante un **editor** de HTML, cuidando que su atributo *action* apunte hacia una página (que todavía no hemos creado) que llamaremos modificado.php.

Ese formulario contendrá cinco campos **input** de tipo **text** (no cuenta el campo id porque no permitiremos que sea modificado), y un botón para enviar.

En cuanto hayamos creado con nuestro editor el código HTML de este formulario, lo pegaremos dentro de nuestra función **editarRegistro**, inmediatamente después de haber traspasado a $fila el contenido de $datos.

Pegaremos el código del formulario dentro de la variable $codigo, cuyo valor lo delimitaremos con **comillas simples** (debido a las abundantes comillas dobles que deben envolver los valores de los atributos en el código HTML). Nuestra función por ahora va quedando así:

```
<?php
function editarRegistro($datos){

      // Extraeremos a $fila el registro:
      if ($fila = mysqli_fetch_array($datos)){

         /* Aquí acumularemos en $codigo cada dato de $fila
ubicado dentro de atributos value de campos */
         $codigo = '<form action="modificado.php"
method="post">
      <fieldset><legend>Puede modificar los datos de este
registro:</legend>
      <p>
      <label>Nombre:
      <input name="nombre" type="text">
      </label>
      </p>

      <p>
      <label>Apellido:
      <input name="apellido" type="text">
```

```
      </label>
      </p>

      <p>
      <label>Edad:
      <input name="edad" type="text">
      </label>
      </p>

      <p>
      <label>País:
      <input name="pais" type="text">
      </label>
      </p>

      <p>
      <label>Especialidad:
      <input name="especialidad" type="text">
      </label>
      </p>

      <p><input type="submit" name="Submit" value="Guardar
cambios"></p>
      </fieldset>
      </form>';

  } else {

      $codigo = false;

  }

return $codigo;
}
?>
```

Esto es apenas un código básico como para comenzar. No nos olvidemos que este formulario debe mostrar escritos, dentro de cada campo, los **valores** del registro que se modificará, que ya los tenemos almacenados dentro de la matriz $fila. Si queremos, podemos probar de incluir una llamada a esta función dentro de nuestra página modificar.php, así vemos cómo muestra el formulario (aunque vacío).

Éste será el código definitivo de la página denominada modificar.php:

```
<?php
// Incluimos los datos de conexión y las funciones:
include("datos.php");

include("funciones.php");

// Verificamos la presencia del código esperado:
if ( isset($_GET["codigo"]) and $_GET["codigo"]<>"" ){
$codigo = $_GET["codigo"];

      // Nos conectamos:
      if ($con = conectarBase($host,$usuario,$clave,$base)
){

      $consulta = "SELECT * FROM empleados WHERE
id='$codigo'";

      if ( $paquete = consultar($con, $consulta) ){

        /* Aquí llamaremos a una función que muestre esos
datos dentro de atributos value de un formulario: */
        $resultado = editarRegistro($paquete);
        echo $resultado;

      } else {

        echo "<p>No se encontraron datos</p>";

      }

  } else {

      echo "<p>Servicio interrumpido</p>";

  }

} else {
      echo '<p>No se ha indicado cuál registro desea
modificar.</p>';
}

echo '<p>Regresar al <a href="listado.php">listado
</a></p>';
?>
```

Si en el listado pulsamos alguno de los enlaces que dicen MODIFICAR, por ahora veremos el formulario vacío:

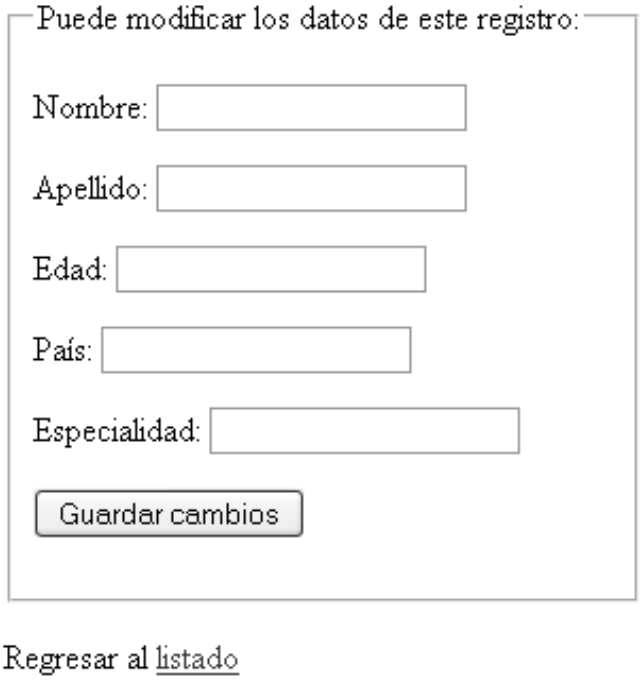


**Figura 13-11. El formulario de edición, sin los datos aún.**

Volvemos a nuestra función **editarRegistro**. Para escribir los datos que faltan dentro del formulario, deberemos ir concatenando los distintos campos de $fila dentro de los atributos **value** de cada **input**. Antes de hacerlo, los traspasaremos a variables locales, para evitar problemas con las comillas; de paso, codificaremos en UTF-8 los datos traídos de la base:

```
$nombreActual = utf8_encode($fila["nombre"]);
$apellidoActual = utf8_encode($fila["apellido"]);
$edadActual = $fila["edad"]; // La edad es un número
$paisActual = utf8_encode($fila["pais"]);
$especialidadActual = utf8_encode($fila["especialidad"]);

/* Reemplazaremos la variable $codigo completa por este
nuevo código donde ya incluimos los datos: */
$codigo = '<form action="modificado.php" method="post">
```

```
      <fieldset><legend>Puede modificar los datos de este
registro:</legend>
      <p>
      <label>Nombre:
      <input name="nombre" type="text"
value="'.$nombreActual.'">
      </label>
      </p>
      <p>
      <label>Apellido:
      <input name="apellido" type="text"
value="'.$apellidoActual.'">
      </label>
      </p>
      <p>
      <label>Edad:
      <input name="edad" type="text"
value="'.$edadActual.'">
      </label>
      </p>
      <p>
      <label>País:
      <input name="pais" type="text"
value="'.$paisActual.'">
      </label>
      </p>
      <p>
      <label>Especialidad:
      <input name="especialidad" type="text"
value="'.$especialidadActual.'">
      </label>
      </p>
      <p>
      <input type="submit" name="Submit" value="Guardar
cambios">
      </p>
</fieldset>
</form>';
```

Ahora sí, veremos los datos actuales escritos dentro de cada campo, listos para ser modificados por el usuario:

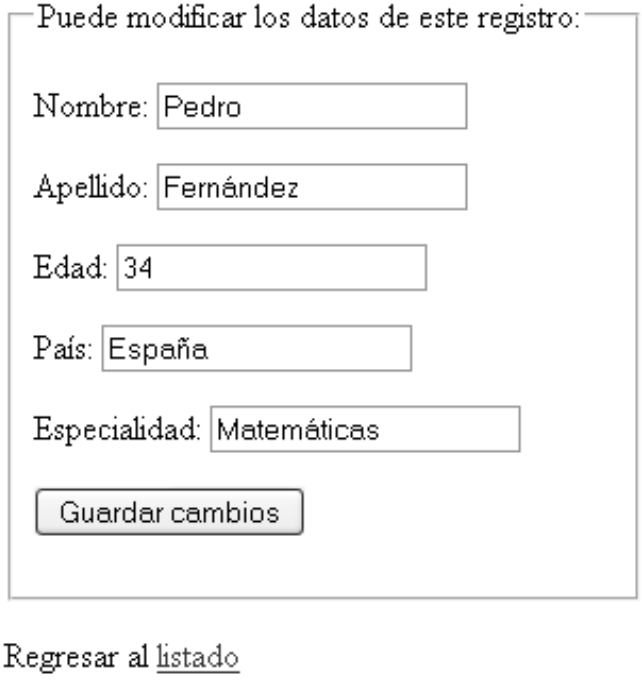


**Figura 13-12. El formulario de edición con los datos actuales cargados en cada campo.**

Pero este formulario, tal como está programado hasta este momento, solo enviará hacia modificado.php las variables "nombre", "apellido", "edad", "pais" y "especialidad". ¿Cómo sabrá, entonces, la orden SQL que ejecutaremos en esa página final (modificado.php) –que, desde ya, contendrá un WHERE–, a cuál registro debe actualizar los datos? Buena pregunta. Hasta ahora, no tendrá forma de saber qué registro hay que actualizar.

Deberemos hacer que este formulario envíe también el **código (id)** del registro que se actualizará. Entonces, agregaremos un campo **input** más, pero que, a diferencia de los otros, no será completado por el usuario, sino que su valor lo definiremos nosotros mismos dentro de un campo oculto (un **input** de tipo *hidden)*, al que le pondremos como *name* "codigo", y cuyo **value** lo escribiremos usando el valor del código que estamos editando, es decir, el que en esta página tenemos almacenado aún dentro de $fila["id"]).

Añadiremos la siguiente línea en el lugar donde traspasábamos variables locales:

```
$codigoActual = $fila["id"];
```

Y añadiremos un input más dentro del código del formulario:

```
<input name="codigo" type="hidden"
value="'.$codigoActual.'">
```

No repetimos el código completo, ya que lo podremos descargar desde la Web de este libro.

Con esto, nuestra página modificar.php queda terminada y, nuestra función **editarRegistro**, también.

Pasemos ahora a codificar la última página, modificado.php, que es la que finalmente ejecutará la orden SQL que actualizará los valores actuales y los cambiará por los nuevos valores ingresados por el usuario en el formulario que acabamos de crear.

## La orden UPDATE del lenguaje SQL

En la última página de este proceso, modificado.php, junto con las demás variables del formulario, nos estará llegando el **código** que pasamos mediante un **input** oculto, por lo que estamos listos para ejecutar la actualización (por supuesto, validaremos si realmente llegó cada dato, y si no está vacío).

Ésta es la sintaxis de la orden SQL que realiza la actualización de los valores de un registro:

```
UPDATE tabla SET campo='valor' WHERE campo='valor';
```

**Atención:** de la misma manera que en la orden DELETE, es fundamental especificar un WHERE, ya que, de no hacerlo, los valores ingresados por el usuario serán aplicados a todos los registros de la tabla y quedarán todos iguales.

En nuestro ejemplo, luego de traspasar a variables locales todos los datos que se hayan recibido de la matriz $_POST, la orden UPDATE quedaría así:

```
UPDATE empleados SET nombre='$nombre',
apellido='$apellido', edad='$edad',
especialidad='$especialidad' WHERE id='$codigo';
```

Por esta razón, el código completo de la última página (modificado.php) de nuestro sistema de altas, bajas y modificaciones, será:

```
<?php
// Incluimos los datos de conexión y las funciones:
include("datos.php");
include("funciones.php");

/* Verificamos la presencia de los datos esperados
(deberíamos validar sus valores, aunque aquí no lo hagamos
para abreviar): */

if ( isset($_POST["nombre"],$_POST["apellido"],$_
POST["edad"],$_POST["especialidad"],$_POST["codigo"]) ){

      // Nos conectamos:
      if ( conectarBase($host,$usuario,$clave,$base) ){

      // Evitamos problemas con codificaciones:
      @mysqli_query($con, "SET NAMES 'utf8'");
      /* Traspasamos a variables locales para evitar
problemas con comillas: */
      $nombre = $_POST["nombre"];
      $apellido = $_POST["apellido"];
      $edad = $_POST["edad"];
      $pais = $_POST["pais"];
      $especialidad = $_POST["especialidad"];
      $codigo = $_POST["codigo"];

      $consulta = "UPDATE empleados SET nombre='$nombre',
apellido='$apellido', edad='$edad', pais='$pais',
especialidad='$especialidad' WHERE id='$codigo'";
```

```
      if ( mysqli_query($con,$consulta) ){
         echo "<p>Registro actualizado.</p>";
      } else {
         echo "<p>No se pudo actualizar</p>";
      }
  } else {
      echo "<p>Servicio interrumpido</p>";
  }
} else {
      echo '<p>No se ha indicado cuál registro desea
modificar.</p>';
}
echo '<p>Regresar al <a href="listado.php">listado
</a></p>';
?>
```

A partir de este momento, ya estamos preparados para interactuar desde cualquiera de nuestras páginas PHP con información almacenada en bases de datos, realizando las cuatro operaciones básicas: leer, agregar, borrar o modificar datos.

# Radiografía de un sistema con back-end y front-end

Pensemos en cualquiera de los sistemas o aplicaciones Web que usamos a diario: un sistema de comercio electrónico, un *home banking,* un *weblog* para publicar noticias, un campus virtual, una red social. Todos estos sistemas tienen una arquitectura común: están compuestos por una doble interfaz, es decir, tienen dos "zonas" completamente diferenciadas: una zona de páginas para uso de los clientes/alumnos/etc., y otra serie de páginas para uso privado del dueño/administrador del sistema.

Desde ya que todos los contenidos de este tipo de sitios Web dinámicos, están almacenados en una base de datos: los **datos** son el núcleo del sistema.

La cadena de producción de los contenidos de la base de datos, será normalmente: Administrador → Base de datos → Usuarios

Aunque también podría ser que los mismos usuarios aporten datos: Usuarios → Base de datos → Usuarios

Y por qué no:

Usuarios → Base de datos → Administradores

(podría ser que los usuarios escribieran mensajes de soporte o consulta para los administradores).

La herramienta con la que interactuarán estos **administradores** para cargar contenidos en la base de datos, será una serie de páginas protegidas con contraseña, desde la cual elegirán ver un listado de contenidos (productos, cuentas, alumnos, cursos, noticias, etc.) y usarán formularios como los que aprendimos en este capítulo para dar de alta nuevos contenidos, modificarlos o borrarlos (tareas de ABM). Éste es el ***back-end*** (el "detrás de la escena" del sitio).

Por otro lado, los **usuarios** interactuarán con la misma información, pero desde otra serie de páginas, que pueden ser públicas o de acceso restringido, pero en las cuales las acciones permitidas son mucho más limitadas: cada usuario solo podrá ver "su" información (sus compras, saldos, cursos, comentarios, etc.) y únicamente podrá agregar aquella información que "le pertenece" (sus pedidos, sus mensajes, sus tareas, etc.). Éste es el ***front-end***. Las páginas por las que navega el usuario.

Vamos a tomar como ejemplo para analizarlo paso a paso (sin codificarlo) un **sistema de pedidos**, en esta ocasión, para el sitio Web de una Pizzería. Será muy fácil reutilizarlo con ligeros cambios para otros tipos de negocio, ya que el circuito de recepción de un pedido suele tener muy pocas variantes entre un negocio y otro.

El ***front-end*** será utilizado por los clientes para hacer su pedido, y el ***back-end*** será utilizado por el operador que modifica el estado del pedido, carga productos, actualiza precios, etc.

Necesitaremos comenzar haciendo un listado de las **tareas** que cada tipo de usuario realizará (podemos mostrarlas visualmente como diagramas de casos de uso). Luego, de estos casos de uso deduciremos qué **páginas** será necesario crear para navegar por cada caso de uso, y cuáles serán las principales **funciones** que será preciso definir. Deduciremos cuántas **tablas** tendremos que crear en la base y cuáles serán los principales **campos** y sus tipos de dato.

## Front-end: el punto de vista del usuario

Imaginemos ante todo las **tareas** de ese cliente que intentará comprar algo utilizando este sistema.

Para ello, pongámonos del lado del **cliente** (comprador): ¿qué procesos son necesarios para que nos envíen una pizza a nuestra casa? El proceso podría consistir en recorrer el listado de productos para elegir los que más nos gusten, y luego transmitir nuestro nombre, domicilio, un teléfono, el detalle del pedido –las cantidades de cada producto, bebidas, postres, etc.– y algún comentario extra que consideremos necesario agregar (por ejemplo: "tocar el timbre rojo").

Puede ser que, a continuación, se nos confirme el importe total del pedido, y se nos pregunte con cuánto pagaremos (para llevar el vuelto o cambio justo). De esta manera, nos indicarán que ya está tomado el pedido y, tal vez, hasta nos den un código de pedido, con el cual podríamos consultar el estado del pedido en caso de demoras.

Resumiendo, podríamos encontrar estas **tareas**:

- Ver listado de productos.
- Completar formulario de pedido.
- Recibir código de pedido.
- Revisar estado del pedido

Si pensamos en las **páginas** que permitirán realizar esas tareas:

- listado.php
- formulario.html
- agregar.php
- pedir-estado.php
- ver-estado.php

Y si pensamos en las **funciones** que serán necesarias, podríamos imaginar al menos las siguientes:

- conectarBase
- consultar
- tabular
- verEstado

Releyendo la lista de páginas, vemos que con las tres primeras funciones, podremos ver el listado de productos en venta. Luego, para mostrar el formulario de pedido no precisamos ninguna función (salvo que quisiéramos crear una para

validar los datos ingresados, lo cual seguramente sería muy práctico). Para agregar el pedido a la base tampoco se precisa una función, y lo mismo para ingresar el código del pedido en un formulario y consultar su estado, aunque sí se precisaría la función *verEstado* para que nos entregue el estado del pedido solicitado.

Todas estas funciones interactuarán con la base de datos. Por lo tanto, tenemos que descubrir cuáles son los **conceptos clave** de lo que tendremos que mostrar o almacenar información. En este sistema, podemos identificar dos grandes conceptos: el de los **productos** que se mostrarán a la venta, y el de **pedidos** que se recibirán. Estas serán entonces las dos tablas que podría tener el sistema (desde ya que podría tener más, como una tabla de usuarios registrados, etc. pero decidimos mantenerlo simple para ir razonando más fácilmente qué es necesario y qué es accesorio).

La tabla de **productos** podría contener campos como:

| Id | TINYINT |
|---|---|
| producto | VARCHAR |
| descripción | TEXT |
| url_imagen | VARCHAR |
| Rubro | ENUM (pizza, empanada, postre, bebida) |
| precio | FLOAT |

Y la tabla de **pedidos** podría contener:

| id | INT |
|---|---|
| nombre | VARCHAR |
| domicilio | VARCHAR |
| teléfono | VARCHAR |
| comentarios | TEXT |
| con_cuanto_paga | VARCHAR |
| importe | FLOAT |
| código | SMALLINT |

| | |
|---|---|
| fecha | DATE |
| estado | ENUM (pizza, empanada, postre, bebida) |

Pensando técnicamente qué parte del proceso de ABM realizará el usuario de un formulario de pedido, primero verá la lista de productos. Para ello, realizará un SELECT. Luego, verá un formulario y, al enviarlo, realizará un INSERT en la tabla de pedidos. Dará de alta un registro. Después, si desea consultar el estado de su pedido, entrará en un formulario y, al enviarlo, hará un SELECT.

Lo podemos esquematizar mediante un diagrama de casos de uso:

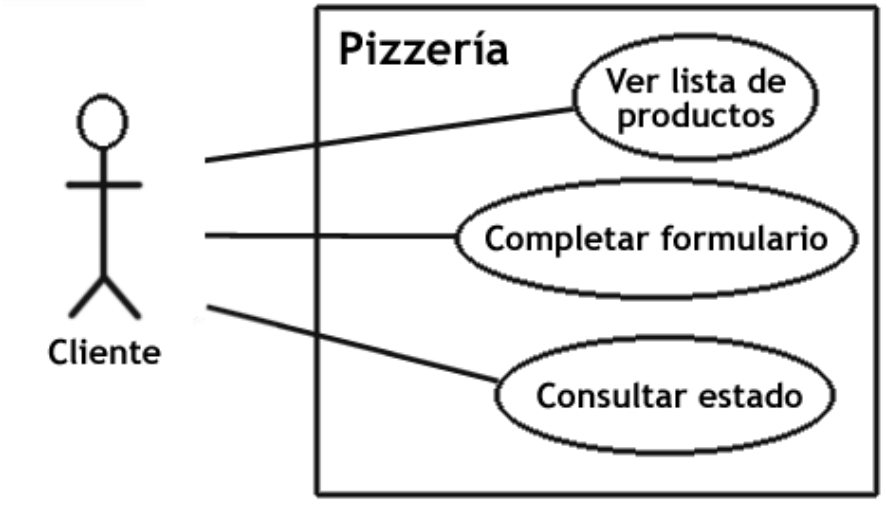


**Figura 13-13. Diagrama de casos de uso del Cliente.**

En este diagrama, incluimos la tarea de consultar el estado del pedido, cosa que realizará el usuario en las páginas pedir-estado.php y ver-estado.php, introduciendo en la primera de ellas el código que le fue generado al realizar el pedido.

Pasemos ahora del otro lado del mostrador.

## Listado de tareas del administrador

Por el otro lado, un operador (administrador) será el encargado de recibir un aviso (podría ser un correo electrónico que se envíe automáticamente al realizarse un

pedido nuevo), imprimirá el pedido y lo pasará en mano al empleado o cocinero que preparará físicamente la caja con el pedido. En ese momento, podría ingresar al sistema y **cambiar el estado** de ese pedido, que en un principio era "recibido"; indicando que ahora está "en preparación" (y un correo electrónico automático le podría avisar al usuario de ese cambio). Una vez que sale de la cocina el pedido y se le entrega al chico de la moto, podría cambiarse el estado del pedido a "en camino". Cuando regrese la moto confirmando que fue entregado, le cambiará el estado a "entregado". Si por algún motivo se tuviera que cancelar el pedido, le cambiará el estado a "anulado".

Todos estos cambios de estado los podrá realizar el administrador desde una serie de páginas de acceso restringido, donde se le solicite usuario y clave, y se utilicen sesiones para validar el acceso.

Para el operador del sistema, sus tareas consistirán en **agregar productos** nuevos (o eliminarlos), **actualizar precios**, **ver el listado** de pedidos (ordenados del más reciente al más antiguo, o aquellos que tengan como estado "en espera") y **cambiar el estado** de un pedido.

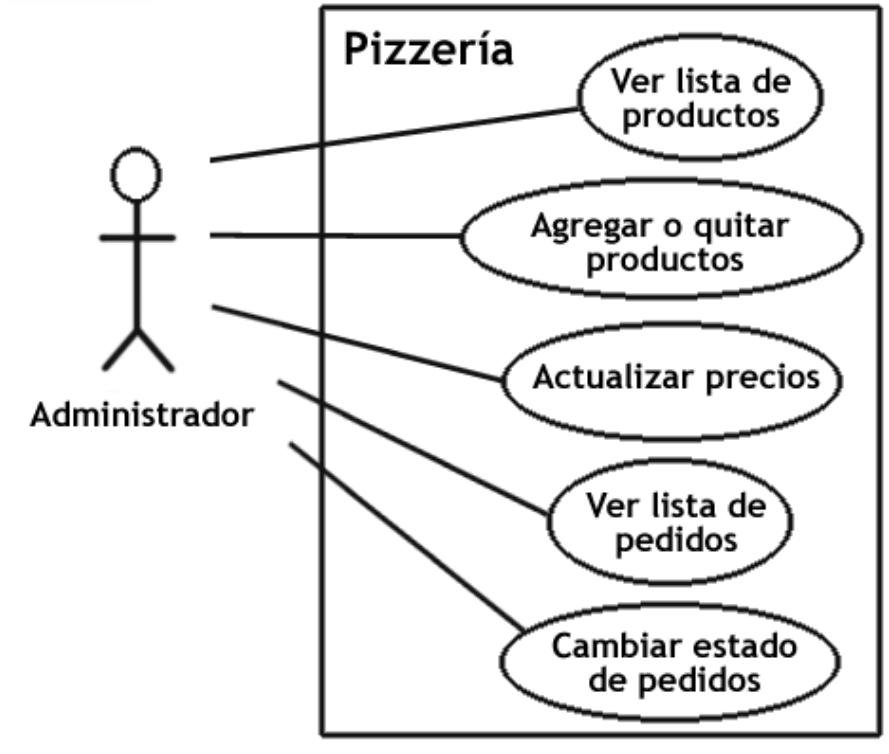


**Figura 13-14. Diagrama de casos de uso del Administrador.**

Es decir, técnicamente, solicitará listados (consultas SELECT), realizará actualizaciones (UPDATE) y dará de alta nuevos productos (INSERT) o los eliminará (DELETE).

Una vez comprendida la lógica subyacente en este sistema, fácilmente podremos extrapolarla a casi cualquier otro sistema que se nos pueda ocurrir. La lógica siempre es la misma: agregar datos a la base, modificarlos, borrarlos, y mostrar la totalidad o solo parte de esos datos.

Éste es el esquema de la mayoría de los proyectos de software libre: todos ellos, con su *back-end* y su *front-end*.

Ahora, antes de conocer cuáles son los más útiles de estos sistemas, en el Apéndice del libro, solo nos resta comprender una técnica de programación muy utilizada en muchos de esos proyectos (aunque no la utilicemos para crear desde cero nuestros sistemas, será necesario que la comprendamos): la programación orientada a objetos, tema del Apéndice final del libro.

# Apéndice:
## Programación orientada a objetos

El objetivo de este capítulo no es cambiar mágicamente nuestra forma de programar y transformarnos, de un momento a otro, en unos expertos en programación orientada a objetos, ya que esto requeriría determinados conocimientos que van mucho más allá de PHP y que abarcan temas como el modelado de objetos con UML, los patrones de diseño de objetos y bastante experiencia en el análisis de sistemas.

Este libro fue pensado para ser útil a un lector diseñador o programador Web principiante, que pueda utilizar sistemas licenciados como software libre como base para sus desarrollos. Por esta razón, en este capítulo, nuestro objetivo consiste en alfabetizarnos en la **lectura** y comprensión de la sintaxis orientada a objetos, pero sobre la base de trabajos creados por gente experta en el tema. Aprenderemos a **entender** esa forma de programar, sabremos **cómo utilizar** clases, métodos y propiedades y, como máximo, nos animaremos, una vez conocida a fondo una clase, a **modificar o agregar** algún método a esa clase. El resto queda para un libro de PHP de nivel avanzado.

# Otro paradigma, otras técnicas

Conoceremos ahora qué es la **Programación Orientada a Objetos** (POO o, más comúnmente, OOP, por sus siglas en inglés *Object Oriented Programming*) y cuáles son algunas de sus características, para poder aplicarla con el lenguaje PHP, ya que es muy frecuente encontrar códigos en la Web programados de esta manera.

La Programación Orientada a Objetos no es un lenguaje más, sino una **forma de programar**, en la que nuestros programas ya no están conformados por

un conjunto de funciones, como lo hacíamos hasta ahora, sino por **objetos** que colaboran entre sí para realizar distintas tareas.

Estos objetos se crean a partir de moldes denominados **clases** (por ejemplo, la clase "pedido", la clase "producto", etc.), muchas veces deducidos de las tablas de nuestra base de datos (en este ejemplo, "pedidos" y "productos", respectivamente).

Nuestro trabajo –centrado hasta aquí en la definición de funciones para que procesen datos– cambiará para definir objetos que servirán para enviar mensajes. Estos objetos usarán sus propios métodos y datos para hacer su trabajo.

Un objeto es un "paquete cerrado" que contiene funciones (denominadas **métodos** en OOP) y sus propios datos (**propiedades**, en OOP; es decir, variables con un valor). A continuación, aprenderemos el vocabulario básico para abordar este nuevo paradigma.

# Clases, objetos, instancias, métodos y propiedades

Las **clases** funcionan como moldes que permiten **crear objetos**; dicho de un modo más preciso, "instancias" (individuos). Es decir, a partir del molde "perro", creamos una instancia "Bobby", otra "Sultán" y otra "Lassie": todas instancias de la clase "perro".

Estos **objetos** están formados por una serie de **propiedades y métodos** que les dan identidad, diferenciándolos de cualquier otro objeto similar. Un objeto de tipo "perro" podrá tener algunas propiedades como color, raza, dueño, nombre, edad y, a la vez, contendrá métodos como ladrar, correr, morder, etc.

Veamos, con otro ejemplo, de qué se trata este concepto.

Supongamos que somos diseñadores industriales y necesitamos crear un nuevo diseño de teléfono celular. Nuestro aparato debería poseer algunos elementos o **características** comunes a todos los demás teléfonos, como un teclado numérico, un micrófono, un auricular, y algunas **funcionalidades** comunes a todos los teléfonos, como la posibilidad de dar tono, discar, cortar, etc.

Para que cada fabricante no deba partir de cero cada vez que crea un nuevo modelo, inicia su diseño desde un ente teórico, abstracto, que es "el" teléfono, básico, común, que posee todas las propiedades y métodos comunes a todo teléfono. Este ente ideal sería la clase "teléfono", es decir, un modelo o molde imaginario con las capacidades imprescindibles para cualquier teléfono.

En programación, a las "características" comunes a todos los objetos de una misma clase las denominaremos **propiedades** y, a las "funcionalidades" comunes a todos los objetos de una clase, **métodos**.

| Clase teléfono | |
|---|---|
| Propiedades | Métodos |
| Teclado | Dar tono |
| Micrófono | Discar |
| Auricular | Cortar |

La clase "teléfono" de este ejemplo es tan teórica como los vertebrados o mamíferos; en la naturaleza, no existe ningún animal concreto que sea solo un mamífero, sino distintas **aplicaciones reales** (en programación, **instancias**) de mamíferos: un perro, una vaca, etc.

A su vez, la clase "perros" permitirá tener dentro otras clases que serían las distintas razas, y de una raza en particular, por ejemplo, Dobermann, se instanciará un perro específico.

Los individuos (instancias) son los **objetos** que realmente "existen"; el resto (mamíferos, perros, dobermanns) son solo construcciones teóricas, "clases" de objetos, cada una con características cada vez más particulares a medida que nos alejamos de la clase padre inicial.

Los individuos, en programación, se llaman **objetos** o **instancias**, y siempre se crean a partir de una **clase** declarada previamente que define sus características.

Al programar de esta manera, crearemos una **clase** que sirva para un propósito en particular, y luego podremos generar "objetos" instanciados a partir de esa clase, que harán la tarea para la que fue creada, más cualquier agregado especial que le queramos hacer como individuos particulares.

Otro ejemplo:

Pensemos en una clase llamada "enviaMails" que genere formularios capaces de enviar correos electrónicos. El **objeto** que se creará será el **formulario** así que, para concebir la clase, debemos concentrarnos en las características (propiedades) y funcionalidades (métodos) de un formulario.

Nuestra clase necesitará cierta información imprescindible para poder cumplir con su objetivo que es enviar mensajes. Por ejemplo, un campo donde escribir un remitente, un destinatario, un asunto y un mensaje. Si le proporcionamos estos datos, deberá ser capaz de ejecutar la funcionalidad de despachar un correo electrónico.

Por única vez, nos tomaremos el trabajo de definir las propiedades y los métodos universales de la clase y, de allí en más, nuestra tarea se habrá

facilitado en gran medida y, cuando precisemos un formulario que envíe mails desde un sitio, crearemos una "instancia" (un objeto) de esta clase "enviaMails", que tendrá las características básicas mencionadas (y, tal vez, otras especiales, únicas de esta instancia que se nos ocurran tales como, por ejemplo, la posibilidad de mandar mensajes a celulares o la de mantener una agenda de contactos, etc.).

En el futuro, al tener que hacer en otro sitio Web un sistema de envío de mails, simplemente volveremos a instanciar la clase "enviaMails" que hemos creado, agregándole otras funciones extra diferentes si fuera necesario, pero siempre ahorrándonos rehacer el cuerpo principal de características y funcionalidades generales.

## Cómo definir una clase, propiedades y métodos

Una clase, entonces, es un "molde" o modelo a partir del cual podemos crear objetos.

Cuando definimos una clase, debemos determinar también qué propiedades y métodos tendrán los objetos que crearemos a partir de ella.

Sus **propiedades** serán sus características, y podemos pensarlas como **variables** que se definen con un valor inicial. En el ejemplo anterior, algunas propiedades de la clase Dobermann serían:

```
$patas = 4;
$cola = "cortada";
$color = "negro";
$fuerza = "mucha";
```

En cambio, los **métodos** de una clase se corresponden con las cosas que es capaz de **hacer** la clase (son "verbos" en infinitivo): volviendo al ejemplo del perro, serían correr, morder, comer, saltar, defender, atacar, saludar, mover la cola, etc.

En programación tradicional, se relacionan con el concepto de **funciones**, pero con la característica de que, en realidad, los métodos son **acciones que modifican alguna propiedad del objeto** en cuestión. Por ejemplo, el método "correr" modifica las propiedades "latitud" y "longitud" del perro, es decir, su posición en el planeta.

Entonces, hablando técnicamente, las clases son "paquetes" de **propiedades** y **métodos**.

Veamos una primera clase bastante simplificada (la idea no es hacerla funcionar sino reconocer sus partes):

```
<?php
class Formulario {
      private $campo;
      public $codigo;

      function __construct(){
           print "Esto es el constructor de la clase
           Formulario";
      }

      function mostrarFormulario(){
       $this->codigo='<form method="post"
                           action="procesa.php">
        <fieldset>
           <label>Destino:
           <input type="text" name="destino"></label>
           <label>Asunto:
           <input type="text" name="asunto”></label>
           <label>Mensaje:
           <textarea name="mensaje" cols="10"
             rows="30"></textarea></label>
           <label>Remitente:
           <input type="text" name="remite"></label>
           <input type="submit" value="Enviar">
        </fieldset>
        </form>';
       return($this->codigo);
      }

     function enviarCorreo($destino, $asunto, $mensaje,
       $remite){
        $this->campo[0]= $destino;
        $this->campo[1]= $asunto;
        $this->campo[2]= $mensaje;
```

```
        $this->campo[3]= $remite;
        return($this->campo);
    }
}// Esta llave cierra la clase
?>
```

La clase continúa, no funciona, está incompleta, se utiliza aquí con fines didácticos. Observemos los siguientes detalles:

1. La clase **se define** anteponiéndole la palabra class al nombre que le daremos (el nombre, en singular y con la primera letra en mayúscula), y luego abriremos una llave:

   ```
   class Formulario {
   ```

2. A continuación, definiremos todas las características o **propiedades** que tendrán los objetos que crearemos a partir de ella, declarando **variables** a las que se les debe anteponer un modificador de visibilidad, es decir, determinaremos desde dónde se podrá acceder a esta variable, de acuerdo con las siguientes categorías: public, private o protected (ya veremos esto más adelante);

   ```
   private $campo;

   public $codigo;
   ```

3. Luego se declararán los **métodos** que podrán utilizar los objetos creados a partir de esta clase, que no son más que **funciones** declaradas de la misma manera a la que estamos acostumbrados (también, se les puede anteponer alguno de los modificadores de visibilidad: *public, private* o *protected*, aunque, en este caso, no lo hayamos incluido).

```
function __construct(){}
function mostrarFormulario(){}
function enviarCorreo($destino, $asunto, $mensaje,
$remite){}
```

4. Notemos que cuando en el cuerpo de los métodos nos referimos a las propiedades (las variables que hemos definido), para darles un valor, lo hacemos mediante la palabra $this-> y, a continuación, el nombre de la variable sin el signo $ (porque ya lo tiene $this):

```
$this->codigo='';
```

La palabra *$this* hace referencia a "este objeto", esta instancia, la que está ejecutando el método. Decir $this->patas = 4 es lo mismo que decir: "adjudicar el valor 4 a la variable 'patas' de este objeto de la clase 'perro' en el que estamos".

5. Observemos un detalle: el primer método de la clase se llama **__construct**; este nombre lo convierte en un **método constructor** de objetos; es decir, que en cuanto creemos un objeto instanciado de esta clase, **se ejecutará automáticamente** este método constructor, asignándole valores a algunas de sus propiedades y haciendo que ya tenga disponibles ciertos métodos listos para usar en esa nueva instancia "recién nacida":

```
Function __construct(){}
```

## Cómo crear una instancia

Una vez definida la clase que servirá de modelo para nuestro trabajo, ya podemos empezar a **crear objetos** a partir de ese molde. Cada objeto que hagamos a partir de una clase se denomina "instancia".

```
<?php
include("formulario.class.php");
/* La clase siempre va en un archivo aparte. Es el código
que analizamos recién.*/
$miFormulario = new Formulario();
?>
```

Desde este instante, $miFormulario ya es un objeto, a pesar de que "parece", a simple vista, una variable. Es una **instancia** de la clase **Formulario**. Por esta razón, ya posee todas las propiedades definidas dentro de la clase y también la capacidad de ejecutar cualquiera de sus métodos. Además, en el mismo momento en el que creamos la instancia (cuando se ejecuta *new*) se acaba de ejecutar automáticamente el método **constructor** de la clase, que asigna valores iniciales a las propiedades de esta instancia.

Con esta instancia, se pueden realizar cualquiera de las tareas típicas:

- Leer el valor de alguna de sus propiedades.
- Definir un nuevo valor a alguna de sus propiedades.
- Ejecutar alguno de sus métodos.

Se pueden **leer** los valores de sus propiedades de la siguiente manera (aunque pronto veremos la forma de hacerlo más eficientemente utilizando un método denominado getter):

```
<?php
include("formulario.class.php");

$miFormulario = new Formulario();

echo $miFormulario->codigo;
?>
```

Se pueden **definir** nuevos valores a sus propiedades de la siguiente manera (pronto veremos cómo hacerlo con un método llamado *setter*):

```
<?php
include("formulario.class.php");

$miFormulario = new Formulario();

$miFormulario->campo[0] = "pepe";
?>
```

También, se pueden **ejecutar sus métodos** como sigue:

```
<?php
include("formulario.class.php");

$miFormulario = new Formulario();

$miFormulario->enviarCorreo("lector@libro.com",
"Saludos","Te mando saludos", "hernan@beati.com.ar");
?>
```

Veamos, ahora sí, un ejemplo completo, que funcione realmente; llamemos a este archivo al Cuadrado.class.php:

```
<?php
class alCuadrado {

      private $numero;

      function __construct(){
          $this->numero = 4;
      }

      function calcularCuadrado(){
          return $this->numero * $this->numero;
      }
}
?>
```

Para crear una instancia a partir de esta clase, en otro archivo, llamado calculador.php incluiremos la clase y crearemos un objeto a partir de ella:

```
<?php
include ("alCuadrado.class.php");

$instancia = new alCuadrado();

print ("Elevar el 4 al cuadrado da como resultado: ".
$instancia->calcularCuadrado());
?>
```

Ahora bien, no siempre vamos a dejar el **4** allí, ya que necesitamos enviar datos a nuestro objeto. ¿En qué momentos le podemos **pasar un dato** a la clase, para que lo utilice en alguna operación?

En los casos que se describen a continuación:

1. Al crear el objeto, con su método constructor.
2. Al ejecutar otro cualquiera de sus métodos.
3. O definiéndole una propiedad que luego sea usada por un método.

1) Veamos la primera alternativa (al ejecutarse automáticamente el método **constructor**): Archivo alCuadrado.class.php:

```
<?php
class alCuadrado {

      private $numero;

      function __construct($cifra){
          $this->numero = $cifra;
      }

      function calcularCuadrado(){
          return ($this->numero * $this->numero);
      }
}
?>
```

Archivo calcular.php:

```
<?php
include ("alCuadrado.class.php");

$cantidad = 10;
/* Número a elevar (también podría llegar desde un
formulario o enlace que envíe variables a esta página). */

$instancia = new alCuadrado($cantidad);
/* Aquí le pasamos el número en el momento en que se
ejecuta el método constructor.*/

print("Elevar ".$cantidad." al cuadrado da: ");
echo $instancia->calcularCuadrado();
?>
```

2) La segunda manera de pasar un dato es en el momento de ejecutar otro de sus métodos; en este caso, lo más práctico sería que se hiciera cuando se

llame al método *"calcularCuadrado"*. Notemos que quitamos el método constructor, ya que no utilizaremos su funcionalidad.

Modificamos el archivo alCuadrado.class.php:

```
<?php
class alCuadrado {

      private $numero;

      function calcularCuadrado($cifra){
          $this->resultado = $cifra * $cifra;
          return ($this->resultado);
      }
}
?>
```

Archivo calcular.php:

```
<?php
include ("alCuadrado.class.php");

$instancia = new alCuadrado();

print("Resultado: ".$instancia->calcularCuadrado(100));
/* Número a elevar (podría llegar desde un formulario o
enlace que envíe variables a esta página). */
?>
```

3) La tercera forma es que le definamos **una propiedad** que luego será usada por un método:

Archivo alCuadrado.class.php:

```
<?php
class alCuadrado {

      public $cifra;
```

```
    function calcularCuadrado(){
        return ($this->cifra * $this->cifra);
    }
}
?>
```

Archivo calcular.php:

```
<?php
include ("alCuadrado.class.php");

$resultado = new alCuadrado();

$resultado->cifra = 45;
/* Aquí le definimos una propiedad como si fuera
una variable interna, de alcance local. */

print("Resultado: ".$resultado->calcularCuadrado());
?>
```

El problema con esta última operación es que estaríamos violando una de las reglas más básicas de la programación orientada a objetos: la **ocultación** de los datos internos en una especie de "caja negra" a la que solo se puede entrar mediante la ejecución de los métodos de una clase. Desde el código de nuestra página, no se debería dar un valor a una propiedad (como tampoco se debería leer el valor de una propiedad directamente: sería mejor usar algún método creado a tal efecto). Para estas tareas sumamente comunes (**dar un valor** y **leer el valor** de una propiedad), aprenderemos, a continuación, a crear métodos que tendrán siempre la misma estructura, y que se denominan *getter* (traedor, lector), y *setter* (definidor). Veámoslos en detalle.

## Métodos getter y setter

### Método getter

La estructura de un método *getter* es sumamente sencilla: no recibe ningún parámetro y se limita a devolver con *return* el valor de una propiedad:

```
<?php
class alCuadrado {

      private $cifra;

      function getCifra(){
          return ($this->cifra);
      }
}
?>
```

### Método setter

Por el contrario, la estructura de un método *setter,* sí debe recibir un parámetro obligatoriamente, que será el valor nuevo que se le asignará a la propiedad en cuestión. No devolverá nada con ningún *return,* simplemente le dará un valor a la propiedad:

```
<?php
class alCuadrado {

      private $cifra;

      function setCifra($valor){
          $this->cifra=$valor;
      }
}
?>
```

A partir de ahora, será normal identificar los métodos *getter* o *setter* a simple vista, por su estructura: si un método no tiene parámetros y devuelve el valor de una propiedad, es un *getter.* Si tiene un solo parámetro y se lo asigna a una propiedad, sin hacer ningún *return,* es un *setter.*

De todos modos, es importante que evitemos caer en un error común: definir un *getter* y un *setter* a todas las propiedades de un objeto, puesto que iría contra la idea principal de la OOP, que es "esconder" la mayor cantidad de términos de información de los objetos que sea posible.

Para este objetivo, cuyo propósito es diferenciar los datos a los que se puede acceder desde una instancia y a los que no, definimos, al momento de crear a cada una de nuestras propiedades, uno de los modificadores de visibilidad: *public, private* o *protected.* Veremos en detalle cuáles son las consecuencias de esta operación.

## Definiendo la visibilidad

### Pública

Cualquier método o propiedad, si no especificamos nada, tiene por defecto visibilidad "pública". También, podemos especificarlo explícitamente, si le anteponemos la palabra *public*. De ambas formas, su valor podrá ser leído y definido desde una instancia cualquiera de esta clase o, incluso, **desde fuera** de ella (que es útil solo en casos muy concretos). Esto equivaldría, por ejemplo, a definir variables dentro de una función como globales, o a leer variables globales desde dentro de la función: una práctica poco recomendable.

Debemos tratar de limitar al mínimo la cantidad de "propiedades" definidas como *public*. Por el contrario, es normal que muchos de los "métodos" sí sean definidos como *public*. Serán los métodos ofrecidos por la clase para interactuar con ella. Se los llama su **interfaz**: son los métodos públicos que una clase propone para que le envíen mensajes desde el mundo exterior. Son su "control remoto", las palancas y los botones desde los que le hacemos hacer cosas a nuestro objeto.

### Privada

En cambio, deberíamos tratar de definir la mayoría de las "propiedades" con visibilidad *private*; es decir, privadas, para que solo desde **dentro de la clase** donde está definida la propiedad, pueda ser leída. Al estar definida como *private*, se podrá leer igualmente por todos sus métodos (incluso aquellos que sean públicos, como los *getter* y los *setter* que estudiamos anteriormente).

No solo las "propiedades" pueden ser *private*, también se puede indicar que tengan visibilidad privada algunos "métodos", para que los use exclusivamente la clase que los definió.

### Protegida

Por último, existe la posibilidad de declarar propiedades o métodos como *protected* (protegidos), con lo cual, además de poder acceder a ellos desde la clase que los definió, también podremos hacerlo desde una clase "hija" o "padre" de la que los definió (esto nos lleva al tema siguiente: la herencia entre clases).

# Cómo aplicar el concepto de herencia

El concepto de herencia nos permite crear una nueva clase que esté basada en otra clase ya existente. Esto nos sirve cuando nos damos cuenta de que precisamos una nueva clase que posea métodos o propiedades muy semejantes a los de otra clase ya definida. En lugar de declarar dos veces lo mismo, simplemente "extendemos" la clase primera, como si creáramos un duplicado de ésta, y tendremos una clase "hija". Luego, a ese duplicado, le podemos agregar algunas otras propiedades y/o métodos propios de esa clase hija y no de la original, es decir, especializamos un poco más sus funcionalidades. La nueva clase va a poder acceder a sus propios métodos y propiedades, pero también a los de su clase "padre" que hayan sido declarados como *protected* o como *public* (pero no podrá acceder a los que hayan sido declarados como *private*).

La idea de fondo de la OOP es definir primero unas pocas clases muy generales, y luego ir derivando, de esas primeras clases, otras clases cada vez más especializadas.

Un ejemplo: a partir de la clase "cuadrado" que habíamos creado en el punto anterior, sería muy fácil crear una clase que calcule el cubo de los números que le pasemos; el cálculo del cubo de un número necesita obtener primero el cuadrado de ese número, y ya que eso lo averiguaba la clase "cuadrado", haremos que la clase "cubo" herede de la clase "alCuadrado" su método "calcularCuadrado", y le agregaremos algo más para que calcule el cubo del número elegido.

Veamos cómo se haría (mostramos todo en un mismo archivo, aunque recomendamos colocar cada clase en un archivo aparte):

```
<?php
class alCuadrado {

      public $numero;

      function elevaAlCuadrado(){
          return ($this->numero * $this->numero);
      }
}

class alCubo extends alCuadrado{

      function elevaAlCubo(){
```

```
        return ($this->elevaAlCuadrado() * $this->numero);
    }
}

$cubo = new alCubo();

$cubo->numero = 3;
// Podría recibirse desde una variable

print("<p>Resultado de ".$cubo->numero." al cuadrado:
".$cubo->elevaAlCuadrado()."</p>");

print("<p>Resultado de ".$cubo->numero." al cubo: ".$cubo-
>elevaAlCubo()."</p>");
?>
```

## Atributos de clase

Uno de los últimos conceptos que veremos (para que podamos reconocerlo cuando lo veamos implementado en las clases que usaremos) es el de los atributos de clase. Es decir, definiremos atributos que no serán atributos de una "instancia" (objeto) de la clase, sino de la clase misma, para su uso interno.

### Constantes

Uno de estos posibles atributos de clase son las constantes. No llevan signo $ delante del nombre, y su valor debe ser constante (un número, una cadena de caracteres), es decir, no puede ser fruto de una variable ni de una operación matemática, ni de una llamada a una función. Una típica constante, similar a las que ya sabemos definir en PHP con la función *define.* Por ejemplo, el usuario y la contraseña de la base de datos.

Se declaran anteponiendo la palabra *const* a la propiedad y se leen anteponiendo *self::* a la propiedad.

Veamos ejemplos:

```
<?php
class Base {
```

```
      const usuario="root";
      const clave="clave";

      function getUsuario(){
         return self::usuario;
      }

      function getClave(){
         return self::clave;
      }
}

$base = new Base();

echo $base->getUsuario();
echo "<br>";
echo $base->getClave();
?>
```

### Static

Otro de los posibles atributos de clase es *static*. Se utiliza para anteponerlo a cualquier propiedad o método, para que luego pueda ser utilizado sin que sea necesario crear una instancia de la clase.

Al usarse sin crear una instancia, no se utiliza $this ni el operador flecha ->, sino que se antepone el nombre de la clase y luego se emplea el operador de resolución de ámbito (o doble dos puntos, más simple) ::, que ya hemos visto en el ejemplo anterior.

```
<?php
class Prueba {

      public static $empresa = "ACME";

}
```

```
echo Prueba::empresa;

?>
```

No solo una propiedad puede ser *static,* también un método puede serlo:

```
<?php
class miClase {

      public static function estatizar();

      }

miClase::estatizar;
?>
```

## Versiones anteriores

En versiones de PHP anteriores a la 5, existen algunas diferencias en la forma en la que se programa con orientación a objetos, de las cuales solo vamos a mencionar un par, las más comunes, ya que nos encontraremos a menudo con ellas en cuanto comencemos a descargar clases de libre uso, que hayan sido programadas con la sintaxis de PHP 4.

El primer punto clave es que en PHP 4 no existían los modificadores de visibilidad. Todas las propiedades y todos los métodos eran **públicos**. En lugar de anteponerles *public, private* o *protected*, en PHP 4 se anteponía la palabra var:

```
<?php
class unaClaseAntigua {

      var $nombre;

}
?>
```

El segundo punto notorio es que, en PHP 4, no existía el constructor "__construct" sino que el constructor debía ser un método **con el mismo nombre** que la clase. Idéntico nombre. Eso era suficiente para que, al momento de crear una instancia con *new*, se ejecutara automáticamente ese método:

```
<?php
class unaClaseAntigua {

      function unaClaseAntigua(){

         // etc...
      }

}
?>
```

Hechas estas aclaraciones, ya manejamos el vocabulario básico de la programación orientada a objetos. No importa si todavía no dominamos la sintaxis, o nos cuesta entender el concepto: lo comprenderemos lentamente, a medida que experimentemos modificando clases creadas por otros.

# Recursos para utilizar libremente miles de clases prearmadas

Ahora que conocemos la sintaxis de una clase, vamos a comenzar a trabajar con una metodología que podría resumirse en "no inventar la rueda": **reutilizar** lo que ya existe.

Si algo existe, es mucho más eficiente reutilizarlo que inventar algo nuevo desde cero que haga lo mismo, por más que nos lleve algo de tiempo comprender su uso, ya que siempre este tiempo invertido será muchísimo menor que el de desarrollar desde cero. Cuanto más grande sea el trabajo por hacer, más razonable será usar algo prearmado.

Un sitio clave para quienes programamos en PHP orientado a objetos es PHPClasses: http://www.phpclasses.org

Una vez que nos registremos gratuitamente, podremos navegar por las distintas **categorías** de *scripts* (son literalmente "miles" de clases creadas por otros programadores que las comparten sin ánimo de lucro, para que sean usadas y mejoradas por todos).

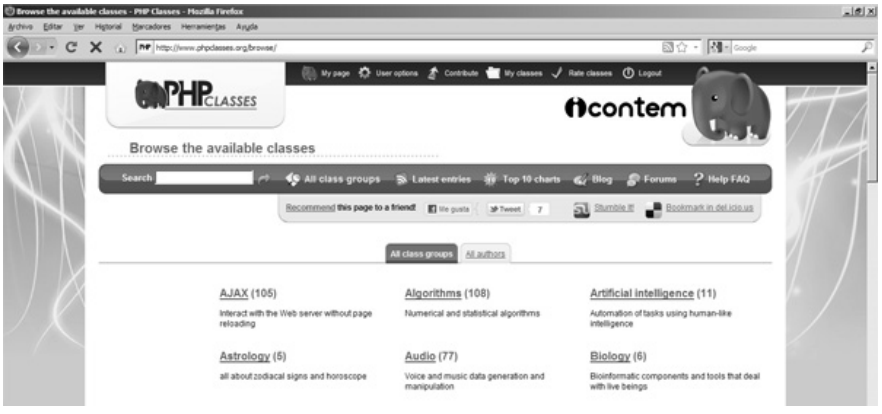


**Figura A-1. El principal sitio de clases de libre uso.**

Otro sitio con clases (aunque no tantas como el anterior) es PHPSnaps: http://www. phpsnaps.com

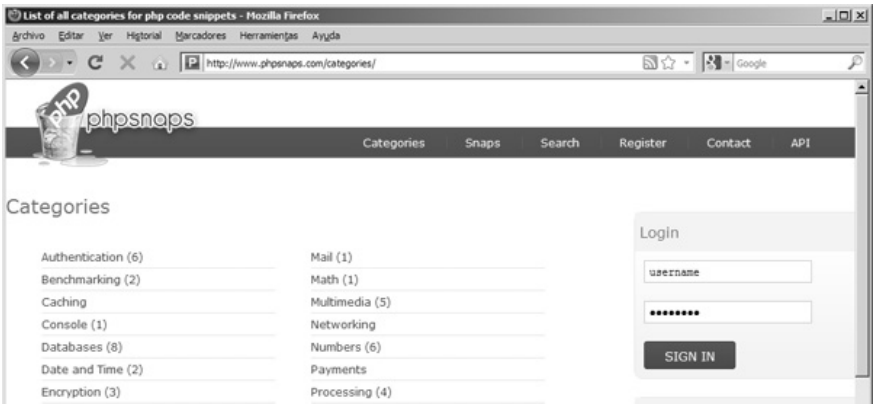


**Figura A-2. Otro sitio dedicado a las clases: PHPSnaps.**

Vamos a utilizar uno de estos sitios para lograr un objetivo en muy poco tiempo, gracias a la potencia de las clases prearmadas: colocaremos un mapa de Google en nuestro sitio usando una clase hecha por otro programador.

## Creando un mapa de Google en un instante

Para este ejemplo, vamos a elegir una clase, de las muchas que hay en PHPClasses, y ejerceremos el criterio de selección típico que se utiliza cuando uno inicia un trabajo a partir de clases ajenas: una vez leídas las especificaciones

generales de cada clase que responde a la búsqueda realizada o a la categoría explorada, y descartadas las clases "que no hacen" lo que esperamos, posiblemente nos queden apenas tres o cuatro clases que prometen hacer lo que necesitamos.

Las descargaremos, las probaremos, leeremos su archivo leame.txt y sus comentarios dentro del código, y veremos si sirven o si las descartamos, repitiendo este análisis hasta quedar con una sola, la "ganadora", y que tomaremos como base para nuestro trabajo. Es muy probable que antes de decidirnos a usar una, tengamos que probar dos o tres clases que luego no usaremos. Pero esto no es un problema, son unos pocos minutos de investigación necesarios para elegir la mejor opción.

Una vez elegida la clase que usaremos, debemos estudiarla analizando "qué hace" cada método y cuáles son los parámetros necesarios para utilizarlos.

Manos a la obra.

Vamos a intentar encontrar alguna clase que muestre **mapas de Google**. Para ello, usaremos el buscador interno de PHPClasses, y escribiremos "Google Maps". Como resultado, obtendremos no solo las clases relacionadas con mapas de Google, sino con *sitemaps* de Google, que no es lo mismo. Por lo cual, aquí hacemos nuestra primera selección, y las candidatas se reducen: solo los primeros resultados parecería que hacen lo que necesitamos:


**Figura A-3. Listado de búsqueda.**

Entramos al primero de los resultados, la clase llamada "N/X API to Google Maps". Leemos su archivo about.txt y nos explica que el archivo que contiene la declaración de la clase es nxgooglemapsapi.php y que el resto son ejemplos (lo cual es muy bueno, siempre nos conviene utilizar clases que vengan con **ejemplos** de uso).

Nos recuerda además que, para utilizar Google Maps, precisamos obtener una **clave de Google**. En el archivo readme.txt, vuelve a insistir que, para que esto funcione, debemos conseguir una clave de Google y escribirla en la parte del archivo nxgooglemapsapi.php que dice:

```
define(GoogleMapsKey, '<your api key here>');
```

Así que, ante todo, deberemos registrarnos y obtener una clave para que Google nos permita utilizar su API (*Application Programming Interface*, o Interfaz de programación de aplicaciones). Lo podemos hacer entrando a: http://code.google.com/intl/es-ES/apis/maps/signup.html

Allí creamos un proyecto, pulsamos su nombre y entramos a APIs y autenticación -> Credenciales, y en la zona de "Acceso a clave pública" pulsamos "Crear clave nueva" -> Clave de navegador, escribimos parte del dominio del sitio que usará el mapa, pulsamos "Crear", y copiamos el valor de "Clave de la API" y lo pegamos en la línea 37:

```
define('GoogleMapsKey','ABQIAAAAoM-kEW8yHxWwveOZA..etc');
```

Si ahora probamos el archivo example1.php, notaremos una serie de mensajes de advertencia por la forma en que han sido declaradas las constantes, sin tomar la precaución de envolver entre comillas su nombre. Lo remediamos fácilmente, envolviendo entre comillas el nombre de cada constante, en las líneas 37 a 44 del archivo de la clase:

```
screen
 */

 define('GoogleMapsKey',
'ABQIAAAAoM-kEW8yHxWwveOZAouVXhTkQdzC1XuexHlQDsWmu58XcfHJ8xQB
xtA9nt_7NDWTsfJfHHxosdNZg');

 define( 'GLargeMapControl'     , 'GLargeMapControl()');
 define( 'GSmallMapControl'     ,   'GSmallMapControl()');
 define( 'GSmallZoomControl'   , 'GSmallZoomControl()');
 define( 'GScaleControl'        , 'GSCALEControl()');
 define( 'GMapTypeControl'      , 'GMapTypeControl()');
 define( 'GOverviewMapControl' , 'GOverviewMapControl()');
```

**Figura A-4. Envolvemos entre comillas los nombres de las constantes, sin incluir espacios en blanco.**

Y... ¡nuestro mapa ya está listo para usar! (Fig. A-5)

Si ahora leemos los **comentarios** que contiene dentro el archivo de la clase, luego de indicarnos que posee la licencia GPL y que, por lo tanto, podemos

utilizar libremente este código, nos indica varias cosas, entre ellas, las más importantes son:

1. Qué hace cada **método**.
2. Cuáles son los **controles** que es posible agregar al mapa y cómo hacerlo. Veamos en detalle qué podemos aprender de estos comentarios.

### Qué hace cada método

Comencemos por las principales tareas:

- Ancho y alto del mapa: se pueden definir con dos típicos métodos *setter*, llamados **setWidth** y **setHeight**:

```
$instancia->setWidth(800);

$instancia->setWidth(600);
```

- Nivel de zoom inicial: se define con el método **setZoomFactor**, que agrega una pequeña validación que nos indica el rango de valores permitidos: de 0 a 17:

```
$instancia->setZoomFactor(16);
```

- Agregar un marcador con una dirección escrita en formato natural, es decir: nombre de calle y número, ciudad, país, y especificando un mensaje que aparecerá al pulsar el marcador sobre el mapa. Para esto, se usa el método **addAddress**:

```
$instancia->addAddress('Av. Corrientes 1050, Buenos

Aires','Obelisco<br>Símbolo de Buenos Aires',true);
```

El último parámetro booleano (*true,* en este caso) indica si esta dirección debe ser el centro del mapa (obviamente, solo una dirección por mapa puede serlo).

- Agregar una ubicación sobre la base de sus coordenadas: **addGeoPoint**:

```
$instancia->addGeoPoint(19.4270499,-9.1275711,'Mexico

DF',true);
```

De la misma manera que en el caso anterior, el último parámetro corresponde al centrado de ese punto en el mapa. Para saber las coordenadas de cualquier lugar del planeta, el archivo example4.php de esta clase nos ofrece una solución lista para usar, aconsejamos probarla ahora mismo para poder ensayar con valores reales de coordenadas.

- Centrar el mapa en torno a una coordenada geográfica específica: **setCenter**.

```
$instancia->setCenter(19,-90);
```

- Agregar controles (barras de zoom, botones de tipo de mapa o vista satelital, escala, etc.). Todos los controles son adicionados al mapa mediante un método denominado **addControl**. Este método es el que más opciones tiene, así que lo veremos en detalle a continuación.

### Los parámetros posibles de addControl

La lista de controles que es posible mostrar mediante el método **addControl** incluye:

- **GLargeMapControl**: son los controles de zoom y las flechas de desplazamiento grandes que llenan la zona superior izquierda del mapa. Si le pasamos este valor al ejecutar el método **addControl**, se verán esos controles. Pero si borramos o comentamos la línea donde se solicita su inclusión (en example1.php, es la línea 12), no aparecerá más ese control:

```
$instancia->addControl(GLargeMapControl);
```

- **GSmallMapControl**: si en lugar de los controles grandes queremos unos más pequeños (ideal para un mapa de pequeñas dimensiones), usaremos esta constante, que solo mostrará las cuatro flechas para moverse, el signo "+" y el "-" para el zoom (estilo minimalista):

```
$instancia->addControl(GSmallMapControl);
```

- **GSmallZoomControl**: solamente muestra el signo "+" y el "-" para controlar el zoom, sin ninguna flecha para moverse por el mapa:

```
$instancia->addControl(GSmallZoomControl);
```

- **GScaleControl**: muestra la escala en la que se está mostrando el mapa, para que a simple vista podamos tener noción de las distancias.

```
$instancia->addControl(GScaleControl);
```

- **GMapTypeControl**: agrega los botones que permiten elegir el tipo de mapa: Mapa, Vista satelital o Híbrido entre ambos.

```
$instancia->addControl(GMapTypeControl);
```

- **GOverviewMapControl**: muestra, en el ángulo inferior derecho del mapa, una vista contextual de la zona que rodea a la que actualmente se ve en

el mapa. Esta zona nos permite arrastrar con el *mouse* y cambiar el área que muestra el mapa.

```
$instancia->addControl(GOverviewMapControl);
```

Todos estos valores de constantes deben ser indicados (como vimos en example1.php) como valores del único parámetro del método **addControl**. Por eso, reiteramos que resulta tan importante que leamos cada uno de los métodos, para ver qué hacen, qué parámetros necesitan recibir, cuáles son los valores posibles y qué consecuencias tienen.

En los archivos example3.php y example4.php, podemos seguir investigando otras funcionalidades, como la de movernos hasta un punto al pulsar en un enlace, o incluir un campo de búsqueda que nos permita ingresar una dirección y nos devuelva sus coordenadas (algo muy útil cuando precisamos saber las coordenadas de un lugar).

Hemos logrado nuestro objetivo de agregar un mapa en muy pocos minutos, gracias a haber tomado como punto de partida una clase creada por otro programador.

Luego de nuestra primera aproximación a esta nueva forma de programar, si queremos seguir profundizando en la sintaxis y lógica de la programación orientada a objetos, como siempre, recomendamos revisar el manual oficial de PHP: http://www.php.net/oop

Ahora, ya estamos en condiciones de ir todavía un nivel más arriba, y aprender a utilizar sistemas enteros listos para usar, que es de lo que hablaremos en el Apéndice Web de este libro:

http://libroweb.alfaomega.com.mx

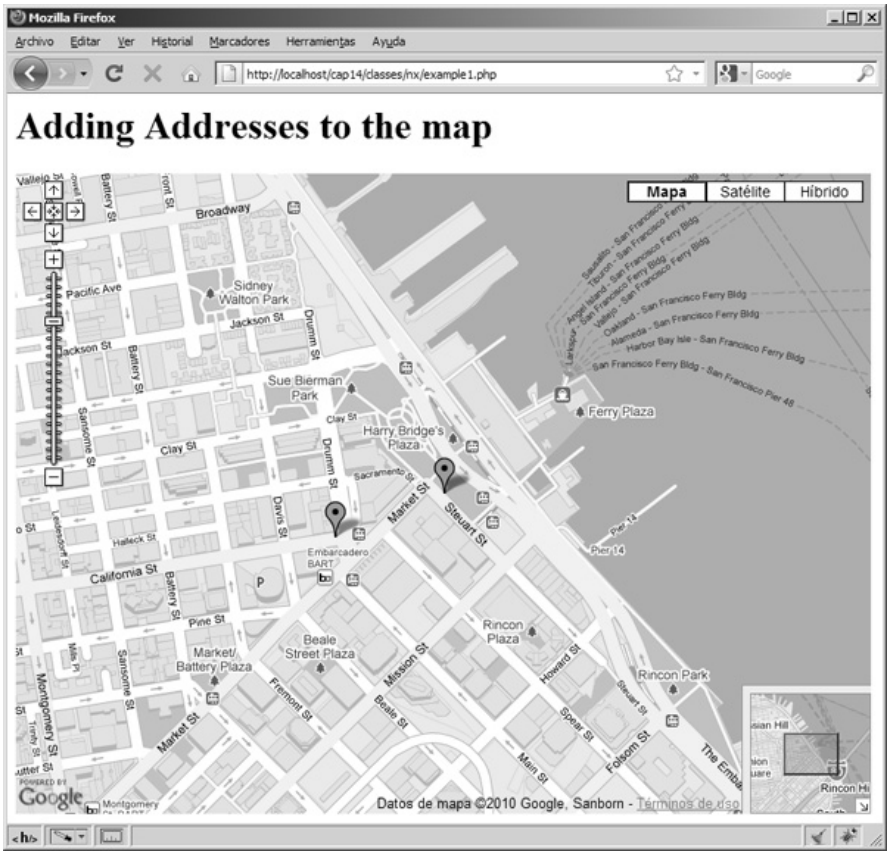


**Figura A-5. El mapa ya se visualiza en nuestra página.**