CURSO DE PHP 5

Iván Sánchez Ortega

ACM Capítulo de Estudiantes

Facultad de Informática, UPM

Marzo 2005

Curso de PHP 5

ACM Capítulo de Estudiantes
Facultad de Informática - Universidad Politécnica de Madrid
Campus de Montegancedo s/n
28660 Boadilla del Monte
MADRID (SPAIN)

Índice general

Capítulo 1

Introducción

Bienvenidos todos al curso de ACM sobre PHP5. Este curso está dirigido a todas aquellas personas con conocimientos previos de programación (preferentemente con algo de C), interesados en conocer a fondo PHP5, un lenguaje dominante en la programación de sitios web dinámicos hoy en día.

En este curso se verán ciertos aspectos "oscuros" de PHP que suelen pasar desapercibidos en la mayoría de los manuales y tutoriales, pero que son funcionalidades interesantes, o que hay que tener en cuenta para programar correctamente.

Según diversas estadísticas, la arquitectura LAMP (Linux + Apache + MySQL + PHP) es, si no la más usada, una de las mayoritarias.

1.1. Paradigma de la programación web

La programación web es muy distinta a la programación tradicional, pero se fundamenta en muchos de sus conceptos. El más importante que hay que tener en cuenta es el concepto de entrada y salida estándar, ya que sobre él se estableció el estándar CGI, que permitió la programación de sitios web en cualquier lenguaje de programación que pudiera compilarse y sacar datos por salida estándar. Más adelante se especializaron algunos lenguajes para esta tarea, e incluso se crearon algunos específicos para ello (como fue el caso de PHP allá por el año 1995).

1.1.1. Modelo cliente-servidor

Antes de entender lo que es un CGI, o un programa del lado del servidor, hay que entender cómo funciona esto de la WWW:

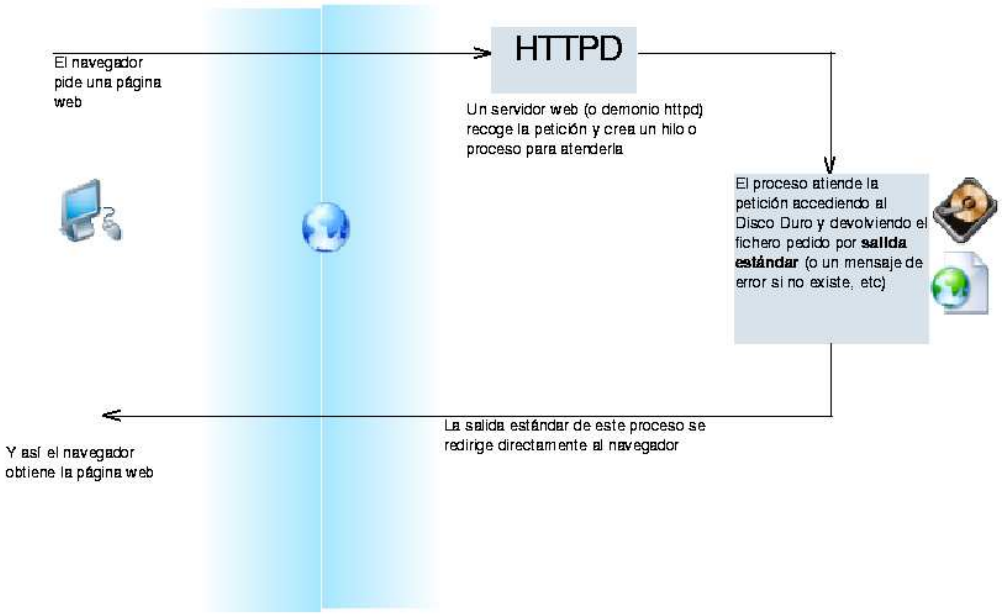

Figura 1.1: Modo cliente-servidor

Normalmente, un navegador web (o “cliente”) lanza una petición HTTP, normalmente por el puerto 80 de TCP, a un servidor web. Un demonio que se esté ejecutando en ese servidor web (apache, IIS, boa, tux, thttpd o cualquier otro software) recoge y analiza esa petición, y devuelve una página web (texto plano, formateado con HTML o XHTML) de vuelta al navegador.

1.1.2. Generación dinámica de contenido

El servidor lee del disco duro un fichero y se lo devuelve al navegador. ¿Qué pasa si en vez de eso, hacemos que una petición ejecute un programa, y devolvemos la salida estándar de ese programa al navegador?

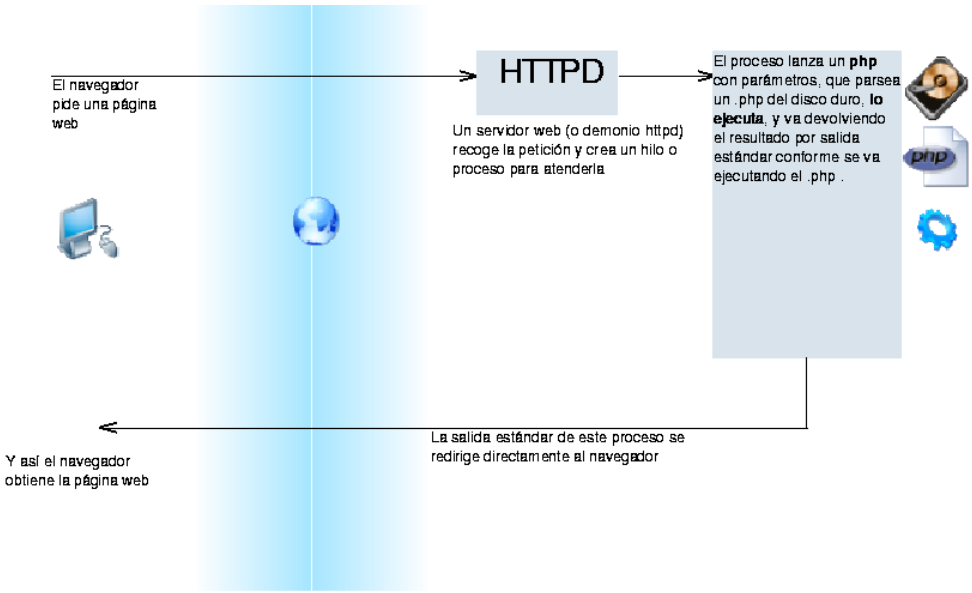

Figura 1.2: Servidor con scripts

Entonces ya tenemos una aplicación web: una página que no se almacena en el servidor web, sino que se genera

automáticamente cuando se visita. El estándar CGI indica que los datos (la petición de la página) se le han de pasar al programa por entrada estándar o por los argumentos de ejecución, y la salida estándar de ese programa será devuelta al navegador web.

Éste método se ha ido refinando, llevando a la automatización el proceso de análisis de la petición HTTP, dejando que el programador se centre en la lógica de su programa, y no en la manera de encauzar los datos. Como veremos, PHP es totalmente transparente a todo este proceso.

1.2. Recursos

Bien, supondremos que con esto hemos sido convincentes y estás deseando ponerte a programar en PHP, pero te estás preguntando qué necesitas...

1.2.1. CD y documentación del curso

En el CD que deberíais recibir junto con esta documentación tendréis las herramientas necesarias para ejecutar programas en PHP: servidores web con módulos de PHP e intérpretes, así como aplicaciones de ejemplo.

1.2.2. php.net

Una URL que no podéis olvidar: *PHP* (`http://www.php.net`). La página oficial, en la que se encuentra el manual oficial, y las últimas versiones disponibles.

Una curiosidad que conviene saber es que si se teclea como dirección `http://www.php.net/loquesea`, iremos directamente a la página de manual correspondiente a 'loquesea'. Esta característica es muy útil para buscar rápidamente la referencia de una función sin necesidad de usar un motor de búsqueda o navegar por el manual.

1.2.3. pear.php.net

PEAR (`http://pear.php.net`), o "PHP Extension and Application Repository"es un repositorio de código, clases de apoyo, pequeños frameworks y trozos de código que realizan funciones más o menos complejas a la vez que cotidianas, y que se pueden usar en vuestras aplicaciones libremente. Algunos ejemplos son las librerías de abstracción de bases de datos, las de manejo y dibujado de gráficos, autentificación o parseo de XML.

1.2.4. dotgeek.org

DotGeek es un servidor gratuito, mantenido por auténticos gurús de PHP, que se ofrece como servidor de pruebas. Si quieres probar tu programa con la versión de anoche de PHP5, o sencillamente eres muy vago como para instalar un servidor web en tu propio ordenador, pide una cuenta en *DotGeek* (`http://www.dotgeek.org`). También se celebran concursos y maratones de programación de vez en cuando, con premios.

1.2.5. php-mag.net

International PHP Magazine (`http://www.php-mag.net`) es una publicación bimensual sobre las últimas tendencias en PHP. Muchas eminencias del tema escriben artículos aquí, perfectos para estar a la última.

1.2.6. triqui.fi.upm.es

Todos aquellos estudiantes de la FI que tengan una cuenta en *Triqui* (`http://triqui.fi.upm.es`) ya tienen automáticamente un servidor web con soporte para PHP donde probar sus programas.

1.2.7. Un editor de texto

¿Y qué nos hace falta para programar? Un editor. Con un simple editor de texto basta. Mi recomendación personal es usar Kate en Linux, o un Ultraedit en Windows. podéis usar también un textpad, emacs, vi, glimmer o vuestro preferido, o bien notepad (o incluso un ed) si no tenéis nada mejor a mano.

Una de las razones por las que se subestima a PHP como lenguaje "para profesionales" es la falta de un entorno de programación completo. Esto es una falsa impresión, ya que *Zend* (`http://www.zend.com`) vende y distribuye Zend Studio, que es un entorno de programación específico para PHP. Zend es la empresa que más invierte en PHP (de hecho,

tiene a sueldo a muchos de los desarrolladores del núcleo). Sin embargo, lo normal es que se oiga hablar de PHP "que es gratis" y se pase por alto la existencia de las herramientas de Zend.

1.3. Lo más básico

¡No esperemos más! ¡Manos a la obra! ¡Programemos como locos el trozo de código más sencillo que se pueda!

1.3.1. Incrustando código

Como PHP se diseñó específicamente para programación web, se incrusta dentro de una página HTML, de la siguiente manera:

```
<html><body>

<? echo 'Hola Mundo!'; ?>

</body></html>
```

Todo lo que esté fuera de los tags especiales $<?-?>$ lo mandará sin procesar al navegador. Todo lo que esté dentro se ejecutará, y el resultado de esa ejecución (lo que salga por salida estándar) se devolverá al navegador.

Esta manera de incrustar código hace muy fácil añadir a una página HTML ya existente, pequeñas partes dinámicas. Hoy en día, y dado el nivel de implantación de PHP, es más normal ver scripts que comienzan por $<?$ y terminan por $?>$, y no tienen nada fuera de estos tags.

echo() es una función muy simple que únicamente pasa a salida estándar lo que se le da como parámetro.

(Realmente echo() no es una función, sino una construcción del lenguaje. Véase cómo es la única función que no necesita que sus parámetros se les pasen dentro de unos paréntesis.)

También vemos que toda instrucción de código se termina en punto y coma, como en C.

1.3.2. ¿Qué hora es?

Veamos un pequeño ejemplo mejor que el típico "Hola Mundo":

```
<html><body>

<? echo date('H:i:S'); ?>

</body></html>
```

En este caso, la función date() se encarga de calcular la hora actual en función de un formato que se le pasa como parámetro, y echo() se encarga de enviar eso a salida estándar (es decir, al navegador).

1.3.3. Comentarios

Para comentar código, se usa la sintaxis de C o de Bash:

```
<?

  // Un comentario de una sola línea

/* Un comentario
de varias
líneas   */

  # Comentario al estilo shell scripting.

?>
```

Como pasa en otros lenguajes, el anidar comentarios de bloque causa problemas.

Capítulo 2

Variables y Operadores

Todo esto es muy bonito, pero no vale de nada si no podemos almacenar información en algún sitio - veamos cómo son las variables en PHP.

2.1. Definición dinámica de variables

(dólar como en shell scripting)

Los creadores de PHP son vagos, muy vagos. Tan vagos que robaron muchos conceptos de otros lenguajes. Uno de ellos es el nombrado de las variables, que es igual que en bash scripting: Todos los nombres de variable empiezan por dólar ($), y no hace falta definir las variables con antelación:

```
<?

$el_sentido_de_la_vida = 42;

echo $el_sentido_de_la_vida;

?>
```

Al usar una variable por primera vez, ésta se define y se reserva espacio en memoria para ella. Además, el tipaje es dinámico. Esto quiere decir que puedo re-definir el tipo de una variable en tiempo de ejecución:

```
<?

$el_sentido_de_la_vida = 42;

$el_sentido_de_la_vida = "<Cuarenta y Dos!;

echo $el_sentido_de_la_vida;

?>
```

Otro dato importante: ¡ Los nombres de variable son sensibles a mayúsculas y minúsculas! Esto puede crear más de un quebradero de cabeza, y es un error muy común.

```
<?

$el_sentido_de_la_vida = 42;
$El_Sentido_De_La_Vida = "Cuarenta y Dos";

echo $el_sentido_de_la_vida;
echo '<br>';
echo $El_Sentido_De_La_Vida;

?>
```

2.2. Tipos básicos

2.2.1. Booleano

Una variable booleana es aquella que es verdadera o falsa.

```
<?

$de_verdad = true;

$de_mentira = false;

?>
```

2.2.2. Entero

Los números enteros son casi siempre de 32 bits, con signo (salvo en algunas arquitecturas). El entero más grande representable es, por tanto, 2^{31} y el menor posible es el $-(2^{31}) + 1$ (más/menos dos mil millones aproximadamente).

También se pueden representar números enteros directamente en notación octal o hexadecimal o binaria:

```
<?
$decimal = 42;

$octal = 0755;  // Con un 0 delante

$hexadecimal = 0x45D0D0AF;  // Con 0x delante
?>
```

2.2.3. Coma flotante

Números en coma flotante de 64 bits, según formato estándar de IEEE (salvo algunas arquitecturas que no lo soportan).

```
<?
$pi = 3.1416;   // Notación normal

$cinco_millones_y_medio = 5.5e6;  // Notación científica

$siete_milesimas = 7e-3;  // Notación científica

?>
```

2.2.4. Cadena

Las cadenas son cadenas de texto ASCII normal y corriente (no Unicode). El tamaño máximo de una cadena viene limitado únicamente por la cantidad de memoria disponible para el proceso de PHP que se ejecute en ese momento (que según configuración ”de fábrica”, es de 8 MiB). No existe el tipo carácter.

Las cadenas se encierran o bien en comillas simples (‘ ’) o bien en comillas dobles (“ ”). Las diferencias son:

- Las comillas simples **no** tienen en cuenta ningún carácter de escape, excepto \‘ (que se escapa por una comilla simple) y \\ (que se escapa por una contrabarra).
- Las comillas dobles hacen expansión de variables. Esto es, cualquier nombre de variable (con dólar incluído) que haya dentro de una cadena con comillas dobles será sustituído por su valor, siempre y cuando la variable sea de un tipo simple (no compuesto).
- Dentro de comillas dobles, no es necesario escapar la comilla simple.

```
<?
$nombre = 'Pepe';

echo 'Hola $nombre<br>';

$frase = "Hola $nombre<br>";
echo $frase;

echo "Hola \$nombre<br>";

echo "Las cadenas pueden
ocupar varias líneas
sin ningún problema, incluyendo
saltos de línea, <pero
hay que tener cuidado de
cerrar las comillas!";

?>
```

Los caracteres de escape son \n, \r, \t, \\, \", \', \$ y \x00, donde "00 "es un número hexadecimal, lo que se sustituye por el carácter ASCII correspondiente.

Las cadenas en PHP son "binary-safe ", es decir, pueden amacenar contenido que no sea texto, sin riesgo de perder información (recordemos el \0 de C). Esto permite trabajar con imágenes, sonido y otros formatos de información.

2.2.5. Nulo

El tipo nulo sólo tiene un valor posible: NULL. Se suele usar como valor de retorno para significar que no hay datos (lo cual es distinto de un cero decimal, o de una cadena vacía).

2.3. Tipos compuestos

(Se verán más tarde)

2.3.1. Recursos

Los punteros a fichero, socket, pipe o fifo, las conexiones a Bases de Datos, las estructuras para el tratamiento de imágenes y otras variables que se usan para trabajar con datos externos toman el tipo "resource ". No es posible (y mucho menos recomendable) manejar manualmente este tipo de variables.

2.3.2. Arrays

Un array es a la vez un vector, una lista, un conjunto, una tabla hash, una matriz, una pila, una cola y un árbol. Se verán en profundidad más adelante.

2.3.3. Objetos

El término "Programación Orientada a Objetos"da el suficiente miedo como para dedicarle una sección entera.

2.4. Operadores

2.4.1. Expresiones

PHP, al igual que C y muchos de sus derivados, maneja el concepto de expresiones. Una expresión es cualquier cosa terminada en punto y coma, y que vale algo. Por ejemplo:

```
<?

5;  // Esto "vale" 5.

5+1; // Esto "vale" 6.

$a = 5+1; // Esto "vale" 6.

substr("Hola",0,2); // Esto "vale" "Ho".

$b = substr("Hola,0,2); // Esto "vale" "Ho".

?>
```

2.4.2. Aritméticos

Los operadores aritméticos son la suma (+), la resta (-), la multiplicación (*), la división (/) y el módulo (%), así como el operador unario de negación (el símbolo de la resta, puesto antes de una expresión).

Hay que tener en cuenta que los operadores aritméticos harán conversión automática de tipos cuando sea necesario:

```
<?

$a = 5 + 1.2; // $a es un float y vale 6.2

$a = 3 / 2;  // La división de enteros devuelve un float

$a = "33 peras" + "9 manzanas"; // $a vale 42.
// Quién dijo que no se podían sumar peras y manzanas (o cadenas)?

?>
```

2.4.3. De cadenas

El punto (.) concatena cadenas (y hace conversión de tipos si es necesario).

```
<?

$uno = 1;

$a = 'Mary tenía ' . $uno . " corderito";

?>
```

2.4.4. Asignación

El operador básico de asignación da un valor a una variable. Pero hay otros operadores de asignación, que realizan operaciones aritméticas o de concatenación de cadenas sobre la variable sobre la que hacen la asignación:

```
<?
    $a = 1;   // $a vale 1.

    $a += 2;  // equivale a $a = $a + 2;  $a vale 3
    $a *= 8;  // equivale a $a = $a * 8;  $a vale 24
    $a /= 4;  // equivale a $a = $a / 4;  $a vale 6
    $a -= 1;  // qeuivale a $a = $a - 1;  $a vale 5

    $a .= " tartas de manzana";
    // $a es la cadena "5 tartas de manzana".
```

```
?>
```

2.4.5. Comparación

Los operadores de comparación devuelven siempre un booleano, son:

- == : Igualdad. Hace conversión de tipos así que (5 == 5.0) es verdadero y (5 == “5”) también es verdadero.
- != : Desigualdad. Como la igualdad, pero justo al revés.
- === : Equivalencia. Como la igualdad, pero además, comprueba los tipos. Así que (5 === 5) es verdadero, pero (5 === “5”) es falso.
- !== : Inequivalencia. Como la equivalencia, pero justo al revés.
- $<, >, <=, =>$: Mayor que, menor que, etc... Pueden comparar números, o pueden comparar cadenas alfabéticamente (según tabla ASCII).

2.4.6. De Errores

Un operador curioso es la arroba, el operador de supresión de error. Sirve para no mostrar el posible error que pueda ocurrir al ejecutar una función:

```
<?

$fd = @fopen('fichero.txt');
// Si el fichero no existe, no se mostrará ningún error

?>
```

Sin embargo, esto no quiere decir que no se vayan a producir errores. Siempre hay que comprobar el valor de retorno para ver si todo ha ido bien.

2.4.7. Post/pre incremento/decremento

Los operadores de post/pre incremento/decremento incrementan (o disminuyen) en una unidad el valor de una variable antes (pre) o después (post) de la evaluación de la expresión:

```
<?

$a = 5;
$b = $a++;  // Post-incremento: $b == 5 y $a == 6.

$a = 5;
$b = ++$a;  // Pre-incremento:  $b == 6 y $a == 6.

$a = 5;
$b = $a--;  // Post-decremento: $b == 5 y $a == 4.

$a = 5;
$b = --$a;  // Pre-decremento:  $b == 4 y $a == 4.

?>
```

2.4.8. Lógicos

El único operador lógico unario es la admiración (”!”), que niega la expresión que le sigue. Los operadores lógicos binarios son:

- && - “Y”lógico
- ||- “O”lógico
- and - “Y”lógico
- or - “O”lógico
- xor - xor lógico

Hay que tener en cuenta que los operadores binarios hacen evaluación perezosa. Si la primera expresión de un “O”lógico es falsa, la segunda expresión no se evalúa (no se calcula, y no se ejecuta en caso de que sea una función).

Como podréis imaginar, los operadores lógicos hacen conversión implícita de tipos, y siempre devuelven un booleano (verdadero o falso). En concreto, cualquier número distinto de 0 equivale a falso, al igual que una cadena vacía, las cadenas “0”y “false”, el array vacío, y NULL. Por esto es importante darse cuenta de cuando estamos esperando un booleano y recibimos otro tipo (por ejemplo, cuando una función puede devolver o bien un booleano a falso o bien un array vacío, y significan cosas distintas). Recordad usar el operador de identidad (“===”) cuando sea necesario.

El porqué de la existencia de dos operadores “Y”y dos “O”lógicos viene dada por su precedencia, “and”, “xor”y “or”tienen la precedencia más baja, y por tanto se evalúan después de todo. Un uso curioso de los operadores lógicos es el control de errores básico. En muchos programas se usa algo como lo siguiente:

```
<?

@hacer_algo_que_puede_fallar() or die('Algo ha ido mal');

?>
```

De esta manera, si la función “hacer_algo_que_puede_fallar”causa un error (y, por tanto devuelve algo distinto de cero), no se mostrará ningún error (operador @), y se terminará la ejecución mostrando un breve mensaje (“die();”).

2.4.9. Bit a bit

No muy usados (algunos los consideran una cosa “oscura”), los operadores de bit a bit son similares a los de C:

- & - “Y”aplicado bit a bit a dos variables.
- |- “O”aplicado bit a bit a dos varaibles.
- $\wedge$ - xor bit a bit; no usar como si fuera el operador de potencia!!
- ~ - negación de los bits de la variable que le sigue
- $\$a \gg \$b$ - desplazamiento de los bits de $a a la derecha, tantas veces como indique $b.
- $\$a \ll \$b$ - desplazamiento de los bits de $a a la izquierda, tantas veces como indique $b.

2.4.10. Ternario ? :

La panacea de la criptoprogramación; la expresión que hay que usar si no quieres que nadie entienda tu código; la manera más sencilla de liar a alguien; el modo más raro de compactar código de manera eficiente: es el operador ternario:

```
<?

$a = ( $b ? $c : $d );

?>
```

Si “$b”, entonces lo que hay entre paréntesis vale $c, y si no, vale $d.

2.5. Magia con las variables

2.5.1. Referencias a variables

En PHP, al ser un lenguaje de alto nivel, no hay punteros. Sin embargo, existen las referencias a variables. Una referencia no se debe entender como un puntero, sino como un segundo nombre para una misma variable.

En concreto, el sistema funciona de manera similar a un sistema de ficheros tipo UNIX (¿aún recordáis algo de SSOO de segundo, verdad?). Un puntero típico se podría comparar a un enlace simbólico: al consultarlo, me dice dónde está lo que busco. Una referencia es más bien como un enlace duro (o "enlace físico"o "hard link"): al consultarlo, me da directamente lo que quiero, de manera totalmente transparente. Otra similaridad que encontramos es el borrado: un fichero se elimina cuando tiene cero enlaces duros (hard links) apuntándole; una variable se destruye cuando hay cero referencias a la misma.

(En realidad, una referencia es un alias en la tabla de símbolos interna del motor de PHP.)

Para hacer referencias a variables, se utiliza el operador &, que convierte lo que le sigue en una referencia (puede ser una variable, o un valor de retorno de una función, o un objeto recién creado con "new"):

```
<?

$a = 42;

$b = & $a;   // $b vale 42. De hecho, $a y $b apuntan a la misma
// celda de memoria, en la que está el valor 42.

$b++;     // tanto $a como $b pasan a valer 43.

$c &= 58;   // Esto dará un error.

?>
```

2.5.2. Comprobando los tipos

Como hemos visto, todas las variables son multivariantes (tipado dinámico), y si nos descuidamos y se opera como no se debe, al final no sabremos con qué estamos trabajando. Para conocer el tipo de una variable existe un grupo de funciones, las "funciones de variables", que permiten saber el tipo de una variable en un determinado momento. Algunas de ellas son is_int(), is_array() o is_string(), y no necesitan demasiada explicación. También tenemos "gettype", que devuelve una cadena que contiene el nombre del tipo de la variable que se le pasa.

2.5.3. Type casting

¿Y si quiero darle a una variable exactamente el tipo que quiero? Bueno, puedo hacer type casting sobre una expresión, precediéndola del nombre del tipo entre paréntesis:

```
<?

$a = (string) 5;   // $a es la cadena "5".

$a = (boolean) 56-7; // $a es el booleano "true".

?>
```

También se puede usar la función settype() para el mismo propósito, pero hacer type casting suele resultar en un código más legible.

2.5.4. Variables variables

Una variable variable es una variable cuyo nombre es variable. Repetid eso seguido unas diez veces (si podéis). Este mecanismo es anterior a las referencias, y con él podemos crear variables con los nombres que queramos:

```
<?

$nombre = 'el_sentido_de_la_vida';

$nombre = 42;

echo $el_sentido_de_la_vida;
// Devolverá por salida estándar "42".

?>
```

2.6. (In)definición

En PHP podemos crear variables cuando nos venga en gana, simplemente asignando un valor a una variable hasta entonces inexistente. En un principio, no hay ninguna variable definida, y conforme se van usando se van definiendo (se van creando).

De cualquier manera, siempre podemos intentar saber el valor de una variable no definida. Para PHP, cualquier variable que no esté definida equivale a NULL.

2.6.1. unset()

Pero también podemos destruir una variable, liberando memoria, o simplemente limpiando datos de entrada para evitar sorpresas posteriores. Esto se hace con unset():

```
<?

$el_sentido_de_la_vida = 42;

unset($el_sentido_de_la_vida);

echo $el_sentido_de_la_vida;
// No imprimirá nada, ya que se evalúa
// la variable, ahora inexistente, como NULL.

?>
```

2.6.2. isset()

Por último, con isset() sabremos si una variable está definida o no. Esto es útil para ver si un navegador me ha mandado datos (como veremos más adelante), o ver si existe determinada variable global.

Capítulo 3

Estructuras de control

3.1. Bifurcación

Cuando queremos que el programa llegado a cierto punto ejecute o no ciertas instrucciones dependiendo de una o varias condiciones nos serviremos de las estructuras de control de flujo: if-else, if-elseif-elseif-else y switch-case.

3.1.1. if-else

La estructura de control if-else ejecuta una de las dos posibles ramas dependiendo de la evaluación de la condición. Por ejemplo:

```
<?

$a = 8;  // Esto "vale" 8.
$b = 3;  // Esto "vale" 3.

if ($a < $b)
// Evaluamos la condición: >a es menor que b?
   {
      echo "a es menor que b";  // Esto se ejecuta si la condición se evalúa a cierto.
   }
else
   {
      echo "a no es menor que b";  // Esto se ejecuta si la condición se evalúa a fals
   }

?>
```

En este caso la condición se evaluará a cierto y se ejecutará la primera rama. Es decir el programa imprimirá por pantalla “a es menor que b”

3.1.2. if-elseif-elseif-else

La estructura de control if-elseif-elseif-else permite anidar tantas condiciones como queramos. La palabra reservada *elseif* permite definir en una sola línea una nueva condición. Por ejemplo:

```
<?

// Introducimos diferentes mensajes de bienvenida para cada idioma
$espanol = "Hola";
$ingles = "Hello";
$frances = "Bonjour";

// Leemos una variable del navegador que nos indica cuál es su lengua oficial
```

```
$idioma=substr($_SERVER[\'HTTP_ACCEPT_LANGUAGE\'],0,2);

if ($idioma == "es")  // >La lengua del navegador es "español"?
  {
    echo "$espanol";  // Imprimimos la variable con el saludo en español.
  }
elseif ($idioma == "fr") // >La lengua del navegador es "francés"?
  {
    echo "$frances";  // Imprimimos la variable con el saludo en francés.
  }
else  // En cualquier otro caso.
  {
    echo "$ingles";  // Imprimimos la variable con el saludo en inglés.
  }

?>
```

Podemos anidar tantas estructuras elseif como queramos.

3.1.3. switch-case

La estructura de control switch-case podemos ejecutar unas u otras instrucciones dependiendo del valor de una variable. Por ejemplo:

```
<?

$color = "azul";

switch ($p_cardinales)
  {
      case "rojo":   // Si color = rojo
         echo "El color rojo es primario";
         break;

      case "amarillo":   // Si color = amarillo
         echo "El color amarillo es primario";
         break;

      case "azul":   // Si color = azul
         echo "El color azul es primario";
         break;

      default:   // Si es otro color
         echo "Es un color no primario";
  }

?>
```

En este caso ejecutaríamos la rama del case “azul ”y se imprimiría por pantalla *El color azul es primario*. Si la varible no es igual a ninguno de los case, entonces se ejecuta el código que hay tras la palabra reservado *default*. Contemplar un caso default por si fallán todos los demás es optativo pero altamente recomendable.

3.2. Bucles

Los bucles permiten ejecutar un conjunto de instrucciones un determinado número de veces dependiendo de la evaluación de una condición.

3.2.1. while

El bucle while posiblemente sea el más sencillo y el más usado. Mientras se cumpla la condición se ejecutan un conjunto de instrucciones. Su estructura es:

```
<?

while (condición)
{
   intrucciones a ejecutar.
}

?>
```

Un ejemplo que va incrementando el tamaño de la fuente de letra:

```
<?

$size = 1;

while ($size <= 6)
{
   echo"<font size=$size>Tamaño $size</font><br>\n";
   $size++;
}

?>
```

3.2.2. do-while

El bucle do-while es muy parecido al anterior la diferencia es que primero se ejecutan el conjunto de instrucciones y después se evalúa la condición para comprobar si se sale del bucle o se sigue iterando. Esto significa que en cualquier caso al menos una vez se ejecutará el bloque de instrucciones. Su estructura es:

```
<?

do
{
   intrucciones a ejecutar.
}
while (condición)

?>
```

3.2.3. for(inic;cond;accion)

El bucle for se comporta de manera similar a los anteriores pero con la salvedad de que se le pasan tres expresiones dentro del paréntesis. Las expresiones van separadas por **';'** y significan lo siguiente:

- Inicialización. Esta expresión se ejecuta sólo una vez antes de entrar en el bucle. Generalmente lo usaremos para inicializar un contador.
- Condición. Esta expresión se evalúa a cierto o falso y dependiendo de su valor se itera otra vez o no.
- Acción. Esta expresión se ejecuta en cada iteración después de ejecutar el bloque de instrucciones entre corchetes. Generalmente lo usaremos para incrementar un contador.

Un ejemplo para construir una tabla:

```
<?

$colum = 5;
$fil = 3;

echo("<table border=\"1\">\n");

for ($i=1;$i<=$fil;$i++)
  {
    echo("<tr>\n");
    for ($j=1;$j<=$colum;$j++)
      {
        echo ("<td>fila $i, columna $j<td>\n");
      }
    echo("</tr>\n");
  }

echo("</table>")

?>
```

Capítulo 4

Arrays

Nunca un nombre tan pequeño ha escondido una estructura de datos tan grande.

4.1. Construyendo un array

4.1.1. array()

La función array() acepta un número variable de parámetros, y siempre devuelve un array; con tantos elementos como parámetros se le pasen. Los arrays creados de esta manera empiezan en el elemento cero.

Para hacer referencia a un elemento del array en concreto, se pone el nombre del array, seguido de la clave que identifica a uno de sus elementos entre corchetes:

```
<?

$a = array();   // $a es un array vacío

$a = array('hola');
// $a tiene un solo elemento, $a[0], que vale "hola".


$a = array('Hola' , ' ' ,  'mundo' , '!');

echo $a[0] . $a[1] . $a[2] . $a[3];
// Esto sacará por salida estándar "Hola mundo!"

echo $a;
// Esto sacará la cadena de texto "Array" por salida estándar, ojo con esto!!

?>
```

(En realidad, array() no es una función, sino una construcción del lenguaje: puede usarse en sitios donde una función no podría, como valores por defecto para variables).

4.1.2. Expansión de arrays

Al ver las cadenas, se explicó lo que es la expansión de variables: al poner una variable dentro de una cadena delimitada por comillas dobles, se reemplaza ese nombre de variable por su valor. Para evitar ambigüedades, hay que añadir llaves cuando se quieran escapar arrays:

```
<?

$a = array('hola' , ' ' ,  'mundo' , '!');

echo "Hola $a[2]";
```

```
// Esto no va a funcionar bien.

echo "Hola {$a[2]}";
// Esto sacará "Hola mundo" por salida estándar.

?>
```

4.1.3. [0] [1] [2]

No sólo podemos crear un array desde cero, sino poco a poco, definiendo elementos uno a uno, igual que si definiéramos variables una a una.

```
<?

$a[0] = 'Hola';
$a[1] = ' ';
$a[2] = 'mundo';
$a[3] = '!';

?>
```

Por supuesto, no hay nada que nos diga que todos los elementos de un array tienen que ser del mismo tipo:

```
<?

$a[0] = 'Hola mundo!';
$a[1] = 3.1416;
$a[2] = true;
$a[3] = 42;

?>
```

Y también puedo destruir (indefinir) un elemento de un array, igual que si indefiniera una variable normal con unset() :

```
<?

$a[0] = 'Hola mundo!';
$a[1] = 3.1416;
$a[2] = true;
$a[3] = 42;

unset($a[2]);
// $a[3] sigue existiendo, no se "mueve" de sitio.

?>
```

Lo cual nos lleva a una interesante observación: los elementos de un array no tienen porqué ser “consecutivos”:

```
<?

$a[1] = 'Hola mundo!';
$a[24] = 3.1416;
$a[2] = true;
$a[50] = 42;

?>
```

Consecuencia: es mala idea, pero muy mala, iterar sobre un array suponiendo que sus elementos son consecutivos. Si en el ejemplo anterior sabemos que $a tiene cuatro elementos, y accedemos a $a[0], $a[1], $a[2] y $a[3], probablemente nos llevemos una sorpresa.

4.1.4. [ ]

El operador especial [ ] añade un elemento con clave numérica a un array, siendo la clave numérica una unidad más grande que la mayor clave numérica definida...

```
<?

$a[1] = 'Hola mundo!';
$a[24] = 3.1416;
$a[] = true;
// Equivale a $a[25] = true;
$a[50] = 42;
$a[3] = 333;
$a[] = 'Ranita';
// Equivale a $a[51] = 'Ranita';

?>
```

4.1.5. ['foo']

Pero no sólo se pueden definir arrays con claves numéricas, sino también con claves alfanuméricas:

```
<?

$persona['nombre'] = 'Iván';
$persona['apellidos'] = 'Sánchez Ortega';

$cosa = 'dni';

$persona[$cosa] = 123456789;
// Equivale a $persona['dni'] = 123456789;

?>
```

Normalmente se pone la clave alfanumérica entre comillas, por precaución. Hace unos años era costumbre hacer referencia a las variables con clave alfanumérica sin las comillas, pero si había definida una constante del lenguaje, o se usaba una palabra reservada como clave, todo empieza a fallar. Hoy en día también se puede usar la sintaxis sin comillas, pero es muy, muy, muy preferible evitar problemas a posteriori.

También se pueden usar variables que contengan una cadena, como se ve en el ejemplo anterior. Y también se puede usar una cadena con comillas dobles, y con una variable dentro que se expanda...

Un array puede tener sólo claves alfanuméricas (cadenas), sólo claves numéricas (enteros, positivos o negativos), ninguna clave (el array vacío), o ambos:

```
<?

$a[1] = 'Hola mundo!';
$a['pi'] = 3.1416;

?>
```

No se pueden usar como claves números en coma flotante, ni booleanos, ni ningún otro tipo de datos.

La función array() también permite definir claves alfanuméricas, o en el orden en que queramos:

```
<?

$a = array( 1 => 'Hola mundo!' , 'pi' = 3.1416 );

?>
```

4.1.6. Arrays recursivas

Como no, un elemento de un array puede ser un array en sí:

```
<?

$a[1] = 'Hola mundo!';
$a[24] = 3.1416;
$a[] = true;
// Equivale a $a[25] = true;
$a[] = array();
$a[] = array(1,2,3);

$matriz[0] = array(1,2,3);
$matriz[1][0] = 4;
$matriz[1][1] = 5;
$matriz[1][2] = 6;
$matriz[][0] = 7;
$matriz[][1] = 8;
// Esto no va a funcionar bien... >porqué?
$matriz[][2] = 9;

?>
```

Con un poco de ganas, y a base de arrays, se construyen árboles de datos sin la menor complicación:

```
<?

$arbol['izqda']['izqda']['dcha'] = 1234;
$arbol['izqda']['dcha']['izqda']['izqda'] = 4567;
$arbol['izqda']['dcha']['izqda']['dcha'] = 6543;

?>
```

4.1.7. print_r() , var_dump()

Antes se ha visto que si hacemos un echo() de un array, veremos un indescriptivo texto "Array"que no nos dice nada acerca del contenido. Hay varias maneras de ver los contenidos completos de un array; una de ellas es iterar sobre el array e imprimir todo; pero un método muy usado es el usar la función print_r(). Esta función imprimirá de manera legible los contenidos de cualquier variable; si se trata de un array, imprimirá todos sus elementos. El "_r"significa que es una función recursiva: si hay un array dentro de un array, podremos ver todos sus contenidos.

Hay que tener en cuenta que print_r() no imprime "$< br/ >$ ", sino saltos de línea ("\n "), así que hay que tener un poco de cuidado para formatear, o ver correctamente, el resultado de imprimir un array con print_r().

```
<?

$matriz[0] = array(1,2,3);
$matriz[1][0] = 4;
$matriz[1][1] = 5;
$matriz[1][2] = 6;
$matriz[][0] = 7;
$matriz[][1] = 8;
// Esto no va a funcionar bien... >porqué?
$matriz[][2] = 9;

print_r($matriz);
// No usar echo(print_r()); !!

echo '<pre>';
```

```
print_r($matriz);
echo '</pre>';

?>
```

Una función similar es var_dump(), pero esta última no formatea el resultado como lo hace print_r(). Sin embargo, es útil para conocer más datos de una variable: tipo, en caso de que sea una cadena, su longitud, etcétera.

4.2. Iteración sobre arrays

Tenemos uns estructura de datos potente, podemos ver todos sus elementos... pero para poder trabajar con todos ellos necesitamos iterar sobre el array...

4.2.1. Iteración sobre arrays: el malo

Lo que muchos noveles programadores hacen es pensar que todo array es numérico y "compacto", con las claves consecutivas, empezando desde cero. Después cuentan los elementos que tiene el array con count() (función que devuelve el número de elementos presentes en un array), y rematarlo todo con un for(;;) :

```
<?

$a[1] = 'Hola mundo!';
$a[24] = 3.1416;
$a[] = true;
// Equivale a $a[25] = true;
$a[] = array();
$a[] = array(1,2,3);

$c = count($a);

for ($i=0; $i<= $c; $i++)
{
 echo $a[$i];
}

?>
```

4.2.2. Iteración sobre arrays: el bueno → foreach, foreach con referencia

Para hacer bien las cosas, PHP nos ofrece una construccion del lenguaje para iterar sobre un array: foreach():

```
<?

$a[1] = 'Hola mundo!';
$a[24] = 3.1416;
$a[] = true;
// Equivale a $a[25] = true;
$a[] = array();
$a[] = array(1,2,3);

foreach($a as $elemento)
{
 echo $elemento;
}

?>
```

foreach() hace todo el trabajo por nosotros, sin posibilidad de error. Además, si queremos podemos también obtener cada par clave/valor, de la siguiente manera:

```
<?

$personas[12345678] = 'Pepito';
$personas[20300400] = 'Fulanito';
$personas[45268732] = 'Menganito';
$personas[21459870] = 'Zutanito';

foreach($personas as $dni => $nombre)
{
 echo "$nombre tiene el DNI número $dni <br/>";
}

?>
```

4.2.3. Iteración sobre arrays: el feo

En tiempos de PHP3 no existía foreach(), así que se usaba un método que ha quedado relegado a aplicaciones viejas y museos de código:

```
<?

$personas[12345678] = 'Pepito';
$personas[20300400] = 'Fulanito';
$personas[45268732] = 'Menganito';
$personas[21459870] = 'Zutanito';

reset($personas);
while ( list($dni,$nombre) = each($personas))
{
 echo "$nombre tiene el DNI número $dni <br/>";
}

?>
```

Puede parecer una sintaxis compleja, pero ahora veremos en qué se basa para iterar bien sobre un array...

4.2.4. list(), each(), prev(), current(), reset(). Puntero interno, devolviendo arrays.

En PHP, un array no es un array en el sentido tradicional de la palabra, sino una estructura de datos mucho más potente: se trata de un map ordenado. Y como todo buen map ordenado que se precie, consta de un puntero interno (invisible para el programador) que apunta a uno de sus elementos. Para trabajar con este puntero se usan, principalmente:

- reset() - hace que el puntero apunte al primer elemento, y devuelve su valor
- end() - hace que el puntero apunte al último elemento, y devuelve su valor
- current() - devuelve el valor del elemento actual
- next() - avanza el puntero al siguiente elemento, y devuelve su valor
- prev() - retrocede el puntero al elemento anterior, y devuelve su valor

Todas estas funciones devuelven el booleano falso en caso de que el array esté vacío o el puntero se haya intentado salir del array. Cuando se itera usando next() o prev(), conviene inicializar el puntero (ponerlo al principio o al final) para evitar sorpresas. Sin embargo, el usar prev() o next() tiene un problema, y es que si nos encontramos con un booleano falso, no sabemos si es que hemos llegado al final del array (o al principio), o si un elemento del array contiene el valor

booleano falso. Si además usamos comparaciones (==) y no identidades (===), nos encontraremos con este problema cuando el valor de un elemento sea 0, una cadena vacía, o cualquier otra variable que se evalúe a falso.

Para evitar este problema, se usa each(), que funciona de manera parecida a next(). Al usar each() sobre un array, la función devuelve tanto la clave y el valor del elemento actual (a su vez, en un array de la forma array(0⇒clave,1⇒valor)), o falso si se ha llegado al final del array; y avanza el puntero interno al siguiente elemento.

Y si tengo un array (como el que devuelve each()), ¿cómo lo convierto a variables normales y corrientes de manera sencilla? Pues con la función list():

```
<?

$a[1] = 'Hola mundo!';
$a['pi'] = 3.1416;

list ($hola,$pi) = $a;

?>
```

De esta manera, si quiero saber cuál es el par clave/valor del elemento actual y, de paso, avanzar al siguiente, sólo tengo que:

```
<?

list ($clave,$valor) = each($array);

?>
```

4.3. Magia con arrays

4.3.1. array_push(), array_pop(), array_shift(), array_unshift()

¿Necesitas una pila o una cola? ¡No hay problema!

- array_push() - Como [ ], añade un elemento al final del array (aunque puede añadir varios a la vez).
- array_pop() - Elimina el último elemento de un array y devuelve su valor.
- array_shift() - Elimina el primer elemento en un array (normalmente el que esté en la posición cero), y desplaza (renumera) el resto de elementos.
- array_unshift - Añade un elemento (o varios) al principio de un array, y desplaza (renumera) el resto.

Para crear una pila, usaremos array_push() y array_pop(). Para crear una cola se puede usar array_push() y array_shift().

4.3.2. explode(), implode()

Se pueden convertir arrays a cadenas y viceversa, de manera muy cómoda en muchas ocasiones, con explode() e implide(). explode() dividirá una cadena en varios trozos (un array de cadenas), haciendo la división por un separador que le especifiquemos. implode() hará lo contrario, "uniendo"todos los elementos de un array en una misma cadena:

```
<?

$frase = "En un lugar de la mancha";

$palabras = explode(' ',$frase);

$frase = implode ('-',$palabras);

echo $frase;
// En-un-lugar-de-la-mancha

?>
```

4.3.3. sort(), asort(), ksort(), natsort()

Antes hemos visto que un array es un map ordenado. Hay que tener en cuenta que el orden de un array depende de cómo se defina. Además, los operadores de igualdad (==) e identidad (===) funcionan de manera distinta, al tener en cuenta (o no) el orden de un array:

```
<?

$a[1] = 'Hola mundo!';
$a['pi'] = 3.1416;

$b['pi'] = 3.1416;
$b[1] = 'Hola mundo!';

$c = ($a == $b);   // $c es verdadero
$d = ($a === $b);   // $d es falso

?>
```

Aprovechando ésto, podemos aplicar funciones de ordenación sobre un array. Las básicas son:

- sort() - Ordena los valores y reasigna las claves numéricamente.
- asort() - Ordena los valores pero mantiene la asociatividad clave/valor.
- ksort() - Ordena por claves.

```
<?

$personas[12345678] = 'Pepito';
$personas[20300400] = 'Fulanito';
$personas[45268732] = 'Menganito';
$personas[21459870] = 'Zutanito';

asort($personas);
// Ahora $personas está ordenado alfabéticamente por los nombres.
ksort($personas);
// Ahora $personas está ordenado por DNI.

// sort($personas) destruiría la información que suponen los DNIs.

reset($personas);
while ( list($dni,$nombre) = each($personas))
{
 echo "$nombre tiene el DNI número $dni <br/>";
}

?>
```

La ordenación se hace ascendentemente, y según lo que resulte de aplicar los operadores de comparación “mayor que”y “menor que”. Si queremos hacer la ordenación de manera descendente, se usan las funciones gemelas rsort, arsort y krsort. Si no queremos usar la comparación habitual (mayor/menor que), podemos usar usort, uasort y uksort. Un caso concreto es la ordenación “natural ”, muy útil cuando ordenamos nombres de fichero que empiecen por números sin ceros a la izquierda.

4.3.4. (un)serialize()

A veces, cuando llega la hora de guardar los datos en un fichero o en una base de datos, puede que usar una estructura compleja no sea la mejor idea. Pero si queremos guardar un array y después volverlo a tener como nuevo, podemos “serializar”el array, convirtiéndolo en una cadena de texto. De igual manera, al “des-serializar”una cadena obtendremos un array (si la cadena no ha sido modificada, claro):

```
<?

$a[1] = 'Hola mundo!';
$a['pi'] = 3.1416;

$texto = serialize($a);

$b = unserialize($texto);

?>
```

4.3.5. Y otras cosas raras

Otras cosas raras que podemos hacer incluyen construir arrays a partir de un array de claves y otro de valores, calcular la diferencia, unión o intersección; aplicar una función a todos los elementos, recortar trozos, "barajar"arrays, extraer elementos al azar, aplanar, y un largo etcétera. Más información en el apartado de arrays del manual oficial de PHP.

Capítulo 5

Procesamiento de formularios

5.1. Los viejos tiempos

Antes de los inicios de PHP, se creó un estándar, el Common Gateway Interface (CGI), para definir cómo pasar datos a una aplicación web. Un programa que se comunique con un servidor web mediante CGI recibirá ciertos datos como variables de entorno, y ciertos datos por entrada estándar. En la época de PHP/FI (antes de PHP3) era comun usar parse_str() y decode_url() para acceder a los datos enviados a una aplicación web hecha en PHP.

Con el paso del tiempo y la proliferación de aplicaciones web, nació la necesidad de simplificar este proceso. Por eso, PHP parsea automáticamente las variables pasadas al script, para que el programador se las encuentre ya listas.

5.2. $_GET

La manera más fácil de pasar datos a nua aplicación web es mediante GET. Esto es, mediante la inclusión de pares variable-valor en la URL de la página a la que visitamos: con un signo de interrogación después de la URL, y pares "variable=valor"separados por &. Por ejemplo, la siguiente URL:

```
http://www.loquesea.org/donde/algo.php?var1=valor1&var2=valor2
```

Hará que el script algo.php reciba estos valores dentro de un array superglobal $_GET, un elemento del array por cada par variable-valor, como si se hubieran declarado de la siguiente manera:

```
<?
$_GET['variable1']= 'valor1';
$_GET['variable2']= 'valor2';
?>
```

De esta manera, es muy fácil construir URLs que sirvan para pasar datos a otros scripts, así como recoger estos datos. Sin embargo, si queremos pasar un dato que contenga un carácter "raro"que pueda dar problemas (espacios, interrogaciones, &, y por extensión cualquier carácter no estrictamente alfanumérico) es conveniente escapar este dato antes de pasarlo a la URL. Esto se puede hacer de manera automática usando encode_url().

Los formularios (X)HTML que usen el método GET harán que el navegador construya una URL con los datos introducidos, y la visite a continuación.

5.3. $_POST

El paso de parámetros por GET tiene varios inconvenientes. Los dos más notables son la limitación del tamaño (alrededor de 8 KiB en total), y la visibilidad de los datos a posteriori (en logs del servidor web, o en la caché o el historial del navegador web). Por estas razones, se suelen usar formularios con método POST.

Aquellos elementos pasados por POST se recogerán por un programa PHP como elementos de la variable superglobal $_POST, de manera análoga a $_GET. Hay que recordar que la clave del elemento del array equivale al nombre que se le ha dado al elemento del formulario en la página anterior, y el valor es el contenido de ese elemento (lo que ha introducido un usuario en un campo de texto, o el "value"de un elemento de una lista desplegable o una casilla de selección).

5.4. $_FILES

El único elemento de formulario que es tratado de manera especial es el usado para subir ficheros (“<file type='file'...>”), entre otras cosas porque el mecanismo para enviar el fichero difiere de el usado para enviar variables por POST. Si subimos un fichero al servidor por medio de un formulario, la variable superglobal $_FILES contendrá no el fichero, sino datos acerca de él (entre otros, el tamaño, nombre real, y localización temporal). El tratamiento de ficheros subidos al servidor está explicaco en un apartado específico del manual oficial de PHP.

5.5. $_COOKIE, $_SESSION

$_GET y $_POST no son las únicas maneras en las que un script puede recoger datos. Los datos pasados al servidor mediante cookies, o los datos almacenados en una sesión, son elementos dentro de las arrays superglobales $_COOKIE y $_SESSION, respectivamente. Ambos métodos se verán más adelante.

5.6. $_REQUEST

$_REQUEST es únicamente la unión de $_SESSION, $_COOKIE, $_POST, $_GET y $_ENV (que contiene las variables de entorno). Si no queremos preocuparnos de cómo ha llegado el dato a la aplicación web, es la superglobal a la que hay que hacer referencia.

En caso de que varios pares variable-valor lleguen a la vez, pero con distinto contenido, se llenará $_REQUEST con el que tenga mayor precedencia, según una directiva de configuración variables_order. El orden por defecto es “EGPCS”, que significa que cualquier variable de entorno quedará sobreescrita con una variable por GET con el mismo nombre, una variable por GET quedará sobreescrita por una variable POST del mismo nombre, etcétera.

5.7. $_SERVER

También hay datos que no se reciben por CGI, pero que están ahí, como por ejemplo la IP desde la que visitan la página, el nombre del servidor web, o la identificación del navegador. Estos datos se almacenan dentro del array superglobal $_SERVER. Para ver todos estos datos, podemos usar php_info(), que además nos dará información sobre la configuración de PHP, así como de los módulos cargados.

5.8. Superglobales

Lo que diferencia a las variables superglobales aquí descritas, aparte de que se “llenan”automáticamente con datos, es que pueden ser accedidas desde cualquier punto de un programa en PHP, bien sea desde el flujo principal, o dentro de cualquier función. Esto aporta mucha comodidad para ver los datos que el script recibe, pero a veces puede causar molestias y hacer que sea necesaria la utilización de una librería que evite la modificación de estos datos.

Capítulo 6

Ficheros

Fichero: dícese de la secuencia de bytes que tiene sentido y, en la mayoría de los casos, soporte físico. Un fichero es la forma básica de almacenar información en un soporte no volátil (típicamente un disco duro), pero también son un medio para intercambiar información con otros programas.

6.1. The C way: fopen(), fread(), fwrite(), fclose()

PHP no es muy original, y soporta acceso a ficheros de manera análoga a como lo hacen lenguajes como C: mediante un descriptor de fichero que se abren con fopen(), leyendo o escribiendo datos a través de ese descriptor con fread() y fwrite(), y cerrando el descriptor con fclose().

```
<?

$fd = fopen('/tmp/qwertyuiop','w');
// A fopen() se le pasa nombre de fichero y modo (en este caso escritura)

fwrite($fd,'Hola mundo!');

fclose($fd);

$fd2 = fopen('/tmp/qwertyuiop','r');
// Abrimos el mismo fichero, pero ahora para lectura

$texto = fread($fd2 , filesize('/tmp/qwertyuiop'));
// fread() necesita descriptor de fichero y longitud a leer

fclose($fd2);

echo $texto;

?>
```

Otras funciones para trabajar con descriptorres de ficheros, como fseek(), fgetc(), fputs(), etcétera, también están disponibles y funcionan de la misma manera que sus análogos en C.

Un punto que hay que tener en cuenta es que PHP cierra automáticamente todos los recursos abiertos cuando finaliza la ejecución de un programa, así que podemos tranquilamente olvidarnos de cerrar los ficheros (a no ser que tengamos concurrencia u otras circunstancias de por medio que nos obliguen a cerrarlo). Para el tratamiento esporádico de datos esto es una práctica aceptable; para el tratamiento intensivo de datos es mejor usar una BBDD.

6.2. The PHP way

El acceso a ficheros de forma “tradicional”no es recomendable cuando estamos en un lenguaje de scripting con una curva de aprendizaje muy baja. Por esto, PHP tiene maneras más fáciles y cómodas de acceder a ficheros.

6.2.1. file()

La función file() nos permite leer un fichero, y almacenar cada una de las líneas que lo componen (separadas e incluyendo el salto de línea al final de la misma) como elementos de un array. Es muy útil cuando necesitamos parsear poco a poco un fichero de texto con formato sencillo, pues tan sólo tenemos que iterar sobre el array que nos devuelve file() para recorrerlo.

6.2.2. file_get_contents(), file_put_contents()

Muchas veces, lo único que queremos de un fichero es leerlo o escribirlo completamente, para lo cual no hace falta usar el método fopen-fwrite-fread. Con file_get_contents() obtendremos en una variable el contenido completo de un fichero, mientras que con file_put_contents() volcaremos una variable a fichero:

```
<?

file_put_contents('/tmp/qwertyuiop',"Hola\nMundo!\n");
// Nombre de fichero, y lo que queremos escribir en él.

$cadena = file_get_contents('/tmp/qwertyuiop');

$array = file('/tmp/qwertyuiop');

echo $array[1];  // Mundo!
echo $array[0];  // Hola

echo $cadena; // Hola Mundo!

?>
```

6.2.3. fpassthru(), file_exists(), nl2br(), is_dir(), basename(), etc

Otra funcionalidad interesante que encontramos en el apartado de funciones de manejo de ficheros de PHP son:

fpassthru() : se le pasa un descriptor de fichero, y saca por salida estándar sus contenidos. Es muy útil cuando queremos pasar a un navegador los contenidos de un fichero grande, pero no queremos cargarlo en memoria (en una variable).

file_exists() : devuelve verdadero si existe un fichero con el nombre que se le pasa, falso en caso contrario. Muy útil para evitar errores al intentar abrir un fichero que no existe.

nl2br() : Una función de tratamiento de cadenas, que convierte saltos de línea (\n) a saltos de línea de HTML ($< br >$). Útil cuando se quiere mostrar en un navegador el contenido de un fichero de texto. Véase también htmlespecialchars().

is_dir(), is_file(), is_link() y otras funciones sirven para comprobar si un fichero es un fichero, o un directorio, o un enlace simbólico, etc.

basename() devuelve el nombre del fichero, quitando cualquier directorio que pueda contener. Es decir, quita todo lo que haya antes de la última barra. Útil cuando nos pasan un nombre de fichero desde fuera y queremos asegurarnos de que nuestro programa sólo accede a ficheros dentro de un determinado directorio.

6.3. include(), require()

A la hora de modularizar el código, lo normal es repartirlo en varios ficheros, y hacer que el programa principal use ese código. En lenguajes compilados, esto se consigue con directivas del preprocesador. En lenguajes interpretados como PHP, esto se consigue en tiempo de ejecución.

Con include() hacemos que se pase el flujo de la ejecución a otro fichero, conservando todo el estado (variables, funciones definidas, etc). El efecto final es equivalente a coger los contenidos del fichero que se incluye, y ponerlos en lugar de la línea que hace el include(). Sin embargo, el fichero incluído tiene que empezar y terminar por $<?$ y $?>$ si queremos que sus contenidos se ejecuten y no se saquen por salida estándar.

Lo normal es usar includes para incluir ficheros que contienen definiciones de funciones y/o clases. También es habitual hacer que un fichero incluído incluya a otro (por ejemplo, una función necesita que tengamos otra declarada). Esto puede provocar problemas si, por error, se intenta incluir dos veces el mismo fichero y como consecuencia se intenta definir dos veces la misma función. Para evitar esto usaremos include_once(). include_once() funciona de la misma manera que include(), pero con una excepción: si se llama una segunda (o sucesiva) vez con el mismo nombre de fichero, no hace nada.

También es habitual que nuestro programa dependa de uno de estos ficheros que estamos incluyendo. En estos casos, lo recomendable es usar require(), que funciona como include() salvo que producirá un error y detendrá la ejecución del programa si no se puede incluir el fichero (porque no exista o porque no haya permisos para leerlo). Análogamente, también existe require_once().

6.4. Ficheros especiales

Normalmente, trataremos ficheros que están en el disco duro de la máquina en la que estamos programando. Sin embargo, también podemos trabajar con ficheros remotos, o con ficheros especiales que no tienen soporte físico.

6.4.1. Ficheros remotos

PHP soporta “wrappers”para acceso a fichero. Esto quiere decir podemos abrir un fichero cuyo nombre empiece por “http:// ”, o trabajar con un fichero que esté en un servidor FTP (“ftp://...”). Por ejemplo:

```
<?

fpassthru(fopen('http://acm.asoc.fi.upm.es'));

?>
```

Actualmente, los wrappers que se pueden utilizar son:

- http:// y https:// - Sólo para leer páginas web.
- ftp:// y ftps:// - Para leer/escribir ficheros desde/a un servidor FTP o FTPS.
- php:// - Para leer de entrada estándar (php://stdin), o escribir a salida estándar o de error (php://stdout, php://stderr). Se usan en shell scripting, puesto que son prácticamente inútiles para programación web.
- zlib:, compress.zlib:// y compress.bzip2:// - Para acceder a los contenidos de un fichero comprimido de manera transparente.
- ssh2.*:// - Para diversas funciones (ejecución de comandos, redirección de puertos, transferencia de ficheros) a través de SSH2. Este wrapper no está activado por defecto.
- ogg:// - Para acceder de manera transparente a un stream de sonido comprimido en formato Ogg Vorbis. Este warpper no está activado por defecto.

6.4.2. pipes, fifos

Un pipe es una tubería que conecta un descriptor de fichero abierto en un programa con otro descriptor abierto en otro programa. Con las funciones de pipes de PHP podemos ejecutar otro programa, y recibir su salida estándar o enviar a su entrada estándar datos como si lo hiciéramos a través de un descriptor de fichero. Las funciones popen() y pclose() permiten un acceso sencillo a esta funcionalidad, y vienen descritas en el apartado de funciones de manejo de ficheros en el manual de PHP. Si queremos un control más fino de ejecución de programas externos, podemos usar la familia proc_* (proc_open(), proc_get_status(), proc_close(), etc) o las funciones de ejecución de programas (“Program Execution Functions ”), que tienen su propio apartado en el manual de PHP.

Un fifo funciona de manera similar a un pipe, sólo que tiene un nombre en el sistema de ficheros del ordenador sobre el que trabajamos, y pueden ser accedidos por programas independientes (no uno que llame o haga un fork al otro). El trabajo con FIFOs es idéntico al trabajo con ficheros tradicionales, excepto que un FIFO se crea con posix_mkfifo(), y se destruye cuando se deja de usar (tanto por el proceso que lee de él como el proceso que escribe a él). Además de esto, PHP cuenta con diversas funciones directamente relativas al estándar POSIX.

6.4.3. sockets

Un socket es otro tipo especial de fichero, que sirve para enviar o recibir datos a otro ordenador, normalmente mediante TCP/IP (aunque también podemos hacerlo mediante UDP, ICMP, o usando IPv6). De esta manera, podemos acceder manualmente a servicios de red si conocemos los detalles del protocolo o nos gusta hacer las cosas “a mano ”. Por ejemplo, abriendo un socket al puerto 23 de otro ordenador podemos hacer que nuestro programa en PHP envíe comandos a la otra máquina por medio de una sesión de telnet.

Las funciones de manejo de sockets se usan para establecer los parámetros de la comunicación (o cerrarla, u obtener metadatos sobre ella), y tienen su propio apartado en el manual de PHP. Para enviar y recibir datos, se usan fread() y fwrite() como si se tratara de un fichero.

Capítulo 7

Funciones

7.1. El Concepto

Una función (o subrutina, o procedimiento) es un fragmento de código que realiza una función específica, separada de un programa en el cual se integra.

7.1.1. Reusabilidad de código

Una de las razones básicas por las que usar funciones en nuestro código es para reutilizar código. Si en nuestra aplicación vamos a hacer lo mismo una y otra vez, no tenemos porqué escribirlo en el código una y otra vez. El reutilizar código conlleva de por sí otras ventajas, como son la reducción del número de líneas de código y un mantenimiento más fácil.

7.1.2. function foo()

La manera más básica de declarar una función es la siguiente:

```
<?

function foo()
{
    echo 'Hola mundo!';
}

foo();

?>
```

Aquí se observan varias cosas: Las funciones se declaran al estilo C, con la palabra reservada function, seguida de los parámetros (en este caso, ninguno) y el código que la compone entre llaves.

Las funciones se declaran en flujo de ejecución normal de PHP, al no haber una función "main ". Esto nos lleva al curioso concepto de la definición condicional de funciones, que no podemos encontrar en lenguajes no interpretados:

```
<?

if (date('Y') == 2005)
// >Estamos en el año 2005?
{
    function foo()
    {
        echo 'Hola 2005!';
    }
}
else
```

```
{
    function foo()
    {
        echo 'Hola mundo!';
    }
}

foo();

?>
```

También hay que tener en cuenta que una función no puede ser re-definida. Sin embargo, podemos saber qué funciones hay definidas usando get_defined_functions() y function_exists().

7.2. Argumentos

7.2.1. function foo($bar)

Si queremos declarar una función que acepte argumentos, únicamente hay que poner el nombre de los argumentos, como si fueran variables, en la definición de la función:

```
<?

function foo( $parametro )
{
    echo "He recibido $parametro .";
}

foo(5);
foo('Hola');

?>
```

Es importante observar que no hace falta declarar el tipo de los parámetros, puesto que toda variable en PHP es de tipo multivariante.

7.2.2. function foo($bar) { return($foobar); }

Las funciones que devuelvan un valor deben hacerlo sencillamente con el uso de la palabra reservada return, que terminará la ejecución de la función y devolverá la ejecución al programa llamante. No hace falta declarar el tipo de retorno, ya que sabemos que será multivariante...

```
<?

function suma_uno( $numero )
{
    return $numero + 1;
}

echo suma_uno(5);

?>
```

7.2.3. Argumentos predefinidos

Una función puede tener un número de argumentos predefinidos. Es decir, argumentos que no es necesario que se le pasen a la función al ser llamados, y cuyos valores tomará de la definición de la función si es necesario:

```
<?

function concatena ($cadena1='Hola ',$cadena2='Mundo!')
{
    return $cadena1.$cadena2;
}

echo concatena();
echo concatena('Uno','Dos');
echo concatena('Adios ');

?>
```

Hay que tener en cuenta que si a una función no se le pasan todos los argumentos no predefinidos, se generará un error. Por esta razón, es recomendable poner los argumentos predefinidos en último lugar.

7.2.4. Argumentos variables: func_get_args()

También podemos declarar funciones que acepten cualquier número de parámetros. Para acceder a todos los parámetros que no han sido definidos en la declaración de la función, usaremos func_num_args() y func_get_args():

```
<?

function concatena ()
{
    $numargs = func_num_args();
    if ($numargs ==0)
            return 'La función necesita al menos un parámetro.';

    $arg_list = func_get_args();
    return implode($arg_list);
}

echo concatena();
echo concatena('Uno','Dos');
echo concatena('Adios!');

?>
```

7.2.5. Referencias: function foo(&$bar)

El paso por referencia sirve para modificar el contenido de las variables que se pasan como argumento, sin tener que declararlas como globales. Se consigue poniendo un & antes del nombre de la variable en la definición de la función:

```
<?

function suma_uno ( & $a)
{
$a++;
}

$a = 5;
suma_uno($a);
echo $a;

?>
```

7.3. Visibilidad

7.3.1. Visibilidad reducida dentro de funciones

Se denomina visibilidad (o “scope ”) a las variables a las que se pueden acceder desde un determinado fragmento de código. En otras palabras, la visibilidad dentro de una función son las variables que se pueden “ver”dentro de la misma.

Normalmente, dentro de una función sólo podremos ver los parámetros que ha recibido, y las variables superglobales (véase capítulo sobre tratamiento de formularios). Si definimos (usamos) una variable dentro de una función, sólo se podrá usar dentro de la misma, y será destruida cuando la función devuelva el flujo de ejecución. Esto es extensible también a métodos de objetos.

7.3.2. Variables globales, scope global

Cualquier variable que se defina (use) fuera de cualquier función, en el cuerpo principal de un fichero PHP, se dirá que está en el “scope global”, son todas variables globales. Estas variables pueden ser accedidas desde una función declarándolas como globales:

```
<?

function imprime_a ()
{
global $a;

echo $a;
}

$a = 5;

imprime_a();

?>
```

Por lo general, es una mala idea acceder a variables globales dentro de una función. Siempre será mucho más limpio usar paso por referencia (o clases estáticas que se verán más adelante) para algoritmos complejos que requieran un punto fijo donde acceder a algunos datos, o usar alguna de las superglobales si es necesario.

Otra manera de acceder a las variables globales es usando la superglobal $GLOBALS en vez de declarar variables individuales como globales:

```
<?

function imprime_a ()
{
echo $GLOBALS['a'];
}

$a = 5;

imprime_a();

?>
```

7.4. Lambda-programación

La “Lambda-programación”, o programación con funciones anónimas, se basa en usar funciones definidas en tiempo de ejecución, que no tienen nombre por sí mismas, y que son referenciadas por variables (que, a su vez, pueden ser parámetros de otras).

Las técnicas de lambda-programación son muy extensas, y puede encontrarse mucha información sobre ellas en buenos libros o cursos de algorítmica, en particular aquellos que abarquen algún lenguaje de programación funcional, como por ejemplo Haskell.

En PHP, una función anónima toma la forma de una variable, aunque sería más preciso decir que la variable es un puntero a la función (como pasa en otros lenguajes). La manera de definir funciones anónimas es usando create_function(). Su uso y ejemplos están perfectamente documentsods en el manual oficial de PHP.

7.5. ¡Tic tac!

PHP no es un lenguaje concurrente, pero se puede simular concurrencia hasta cierto punto. Mediante register_tick_function() y unregister_tick_function() podemos hacer que una función se ejecute a intervalos regulares (por ejemplo, cada 5 instrucciones) durante la ejecución normal de nuestro programa. Un uso de esta funcionalidad es ir comprobando periódicamente el uso de memoria de un script y tomar nota de ello en un log.

7.6. Cierra la puerta al salir

A veces es recomendable ejecutar código aunque hayamos cerrado la salida estándar (es decir, una vez que el navegador web ha recibido todos los resultados). Esto se puede hacer con register_shutdown_function(), que se encargará de ejecutar las funciones que deseemos después de finalizar la ejecución normal del script. Todo código que sea ejecutado de esta manera no podrá acceder a salida estándar (puesto que ya se ha cerrado), lo cual puede dificultar la depuración. Además, esta técnica puede dar problemas en ciertos servidores web.

Capítulo 8

Programación Orientada a Objetos

La Programación Orientada a Objetos (o “Object Oriented Programming ”, OOP), sin embargo, es capaz de ayudarnos a producir código muy legible, extensible y, gracias a la potencia de PHP5, de manera muy creativa.

8.1. Objetos por todos lados

8.1.1. Métodos y atributos

Un objeto se puede definir como una entidad que guarda un estado propio, sobre la que se pueden efectuar una serie de operaciones. El estado propio se consigue gracias a las “propiedades”, variables dentro de un objeto, y los “métodos”, funciones dentro del objeto. Lo primero es definir la entidad de manera abstracta; es decir, la clase. Una vez hecho esto, podremos hacer objetos basados en esa clase, con “new”.

Para acceder a los métodos y propiedades de un objeto se usa una flecha (“→ ”). Si queremos acceder a las propiedades o métodos de un objeto dentro de ese mismo objeto, se usa la variable reservada ”$this”.

```
<?

class alumno
{
    // Propiedades (variables)
    public $nombre;
    public $dni;

    // Métodos (funciones)
    public function suspender_redes()
    {
        echo $this->nombre.’ ha suspendido redes y está muy triste.’;
    }

    public function aprobar_redes()
    {
        echo $this->nombre.’ ha aprobado redes y está muy contento.’;
    }
}

$pepe = new alumno;
$pepe->nombre = ’Pepito Pérez’;
$pepe->dni = 123456789;

$pepe->aprobar_redes();

?>
```

No es una buena idea darle a una clase el nombre de una palabra reservada (“class while”).

8.1.2. Instancias: instanceof (comprueba que es un objeto es una instancia de una clase)

Una confusión común es no diferenciar claramente entre “clase”y “objeto”. Una clase es una definición abstracta: una persona, una asignatura, un coche. Un objeto, que debe ser una instancia de una clase, es una entidad propia: Pepito Pérez, Estructuras de Datos II, el BX blanco con matrícula 6609-JW.

Dos objetos son iguales (==) si sus propiedades (variables) son iguales; dos objetos son idénticos (===) sólo si son la misma instancia (una referencia). También se pueden clonar objetos, lo cual crea una instancia igual, pero no idéntica. Más información sobre clones de objetos en el manual de PHP.

Para saber a qué clase pertenece un objeto en concreto, se puede usar instanceof o get_class():

```
<?

class alumno
{
    public $nombre;
    public $dni;
}

$pepe = new alumno;
$pepe->nombre = 'Pepito Pérez';
$pepe->dni = 123456789;

if ($pepe instanceof alumno)
{
     // Hacer cosas
}

echo get_class($pepe);

?>
```

8.1.3. Apto para todos los públicos: public, protected, private, visibilidad - E_STRICT

Hasta ahora, hemos visto propiedades y métodos públicos, que se pueden acceder desde fuera del objeto sin ningún problema.

Muchas veces, no queremos que las propiedades de un objeto se cambien desde fuera. Por ejemplo, si tenemos una pila, queremos que sólo se puedan hacer las operaciones de push y pop. Para ésto, declararemos propiedades como privada:

```
<?

class pila
{
    private $datos=array();

    public function push($dato)
    {
        array_push($this->datos,$dato);
    }
    public function push($dato)
    {
        return array_pop($this->datos);
    }
}

$mi_pila = new pila;

$mi_pila->push(5);
```

```
$mi_pila->datos = array(5,8,7);
// Esto dará un error, puesto que no puedo acceder a la propiedad $datos

?>
```

En el ejemplo hemos aprovechado para ilustrar también propiedades por defecto: al crearse una instancia de la clase pila, la propiedad $datos será siempre un array vacía.

8.1.4. Herencia

A veces ya tenemos una clase diseñada, pero queremos añadirle funcionalidad, o crear una clase similar. En estos casos en los que queremos ampliar lo que una clase puede hacer (pero queremos conservar la antigua), se usa la herencia de clases.

Cuando una clase (clase “hija”) hereda de otra (clase “padre”), al declararla como “class hija extends padre”, toma todas las propiedades y métodos del padre. De esta manera, si se llama a un método o propiedad y éste no está definido en el hijo, se tomará la definición del padre y se obrará en consonancia.

Sin embargo, hay que tener otra consideración: un método o propiedad declarada como privada sólo se puede usar en el contexto de una clase, no en el contexto de una clase que herede de ella (una clase hija). Si queremos que una propiedad o elemento se pueda acceder desde el contexto de una clase hija pero no desde fuera de un objeto, no la declararemos como privada (“private ”), sino como protegida (“protected ”).

Para usar algo que sólo está en el contexto de una clase padre (como puede ser una función que hayamos redefinido), accederemos a ese contexto usando “parent::”. Esto es el operador de contexto, que veremos con detenimiento más adelante.

Ilustraremos ésto con un ejemplo:

```
<?

class pila
{
    private $datos=array();

    public function push($dato)
    {
    array_push($this->datos,$dato);
    }
    public function push($dato)
    {
        return array_pop($this->datos);
    }
}

class pila_acotada extends pila
{
    private $elementos=0;
    private $max_elementos=30;

    public function esta_vacia()
    {
        return ($this->elementos == 0);
    }
    public function esta_llena()
    {
        return ($this->elementos == $this->max_elementos);
    }
    public function push($dato)
    {
        if ($this->esta_llena())
```

```
            return false;  // Error
        $this->elementos++;
        parent::push($dato);
        return true;
    }
    public function pop()
    {
        if ($this->esta_vacia())
            return false;  // Error
        $this->elementos--;
        return parent::pop();
    }
}

$mi_pila = new pila_acotada;

$mi_pila->push(5);

?>
```

Veamos qué hace este ejemplo. La funcionalidad básica (almacenar datos en una pila) está en el padre, y sólo en el padre. El hijo aporta una funcionalidad extra (la acotación), pero no se preocupa lo más mínimo de cómo se almacenan los datos (llama al push y el pop del padre, en su contexto). De hecho, el hijo no podría acceder directamente a $this→datos, puesto que está definida como variable privada. Sí que podría acceder, sin embargo, si fuera una variable protegida.

8.1.5. Expansión de propiedades

Ya vimos cómo una variable sencilla se expande automáticamente si se introduce dentro de una cadena delimitada con comillas dobles, y cómo se pueden expander elementos de arrays poniéndolos entre llaves. De igual manera, poniéndolas entre llaves, las propiedades de un objeto se pueden expandir también:

```
<?

class alumno
{
    public $nombre;
    public $dni;

    public function suspender_redes()
    {
        echo "{$this->nombre} ha suspendido redes y está muy triste.";
    }

    public function aprobar_redes()
    {
        echo "{$this->nombre} ha aprobado redes y está muy contento.";
    }
}

$pepe = new alumno;
$pepe->nombre = 'Pepito Pérez';
$pepe->dni = 123456789;

$pepe->aprobar_redes();

echo "{$pepe->dni}";
```

```
?>
```

Si se omiten las llaves, nos encontraremos con un poco descriptivo “Object”en vez de la propiedad que queremos.

8.2. Magia con “__”

En PHP, todas las funciones que empiezan con __ (dos guiones bajos) son cosas “mágicas”, y debe evitarse, en la medida de lo posible, que un programador ponga a sus propias funciones nombres que empiecen por __, puesto que en versiones posteriores del lenguaje podrían haber más funciones mágicas, y tendríamos problemas.

8.2.1. __construct()

La función constructora de clases se ejecuta al instanciar un objeto. En otros lenguajes, la función constructora toma el mismo nombre que la clase, pero PHP tomó la decisión de usar el mismo nombre para todos los constructores, para ganar en claridad.

El constructor de una clase se ejecuta justo después de crear la instancia. Puede (o no) llamarse con argumentos. Normalmente se usa el constructor para inicializar el estado del objeto, siempre que se necesite que el estado del objeto dependa de las condiciones que se den en el momento de crearlo (en el ejemplo a continuación, la fecha en la que se hace la instancia):

```
<?

class alumno
{
    public $nombre;
    public $dni;
    private $fecha_matriculacion;

    public __construct($fecha=NULL)
    {
        if (!$fecha)
            $this->fecha_matriculacion = date('m/Y');
            // Mes/Año actual
        else
            $this->fecha_matriculacion = $fecha;
            // Fecha que nos hayan pasado al instanciar el objeto
    }
    public function suspender_redes()
    {
        echo "{$this->nombre} ha suspendido redes y está muy triste.";
    }
    public function aprobar_redes()
    {
        echo "{$this->nombre} ha aprobado redes y está muy contento.";
    }
}

$pepe = new alumno;
// Se llama al constructor sin argumentos
$pepe->nombre = 'Pepito Pérez';
$pepe->dni = 123456789;

$juan = new alumno('09/2003');
// Se llama al constructor con un argumento
$juan->nombre = 'Juan Juarez';
$juan->dni = 987654321;

?>
```

8.2.2. __destruct()

De manera análoga al constructor, existe el destructor, que se ejecuta justo antes de hacer desaparecer un objeto. Recordemos que una variable (en nuestro caso, un objeto) se destruye siempre cuando se eliminan todas las referencias a ella, y siempre que termine la ejecución. No hace falta destruir específicamente todos los objetos que hemos creado, puesto que ya se destruirán al terminar la ejecución.

8.2.3. __autoload()

El __autoload es uno de los mejores inventos desde el pan de sandwich. Todo buen programador fragmenta su código en ficheros más o menos pequeños, más manejables que un fichero grande conteniendo todo el código - una muy buena práctica es definir una clase en un fichero con el nombre de la clase, y es una práctica muy extendida.

Ahora bien, lo más normal es incluir todos los ficheros que definen clases, aunque no se usen, no vaya a ser que intentemos instanciar algo que no hemos definido. Antes de la llegada de PHP5, algunos programadores optaron por soluciones un tanto raras para evitar tener que incluir y definir todas las clases (menos clases declaradas = menos código que interpretar en tiempo de ejecución = más rápido), sobre todo cuando juegan con unos pocos cientos de clases.

¿De qué va esto de __autoload() ? Si __autoload() se define en el contexto global (fuera de cualquier clase o función), y se intenta instanciar una clase no definida, se llamará a __autoload() pasando como parámetro el nombre de la clase, y se instanciará la clase cuando __autoload() haya terminado. Lo más normal es que __autoload() haga un include (o require) del fichero que define la clase que se busca.

```
<?

//  Antes
// require_once('class.alumno.php');
// require_once('class.pila.php');
// require_once('class.coche.php');
// require_once('class.tenedor.php');

//  Ahora
function __autoload($clase)
{
 require_once("class.$clase.php");
}

$juan = new alumno('09/2003');

$mi_pila = new pila;

?>
```

8.2.4. __sleep(), __wakeup()

Como los arrays, los objetos se pueden serializar para ser convertidos en una cadena de texto que después podemos salvar a fichero o BBDD. __sleep prepara al objeto para ser serializado (se llama automáticamente justo antes de serializar), y __wakeup se ejecuta justo después de des-serializar un objeto. (siempre que __sleep y/o __wakeup estén definidas en la clase, claro está).

Uno de los usos que se le da es cerrar ficheros y conexiones a BBDD en __sleep y volver a abrirlos en __wakeup.

8.2.5. __toString() - sólo con echo() !!

Normalmente, cuando hacemos un echo de un objeto, nos encontraremos con un insípido y poco descriptivo “Object”en salida estándar. Para aliviar esto, y facilitar en muchos casos el debug de programas grandes, está __toString. __toString se llama siempre que se hace un echo o un print() de un objeto:

```
<?

class alumno
```

```
{
    public $nombre;
    public $dni;

    public function suspender_redes()
    {
        echo "{$this->nombre} ha suspendido redes y
          está muy triste.";
    }

    public function aprobar_redes()
    {
        echo "{$this->nombre} ha aprobado redes y
          está muy contento. ";
    }

    public function __toString()
    {
        return $this->nombre;
    }
}

$pepe = new alumno;
$pepe->nombre = 'Pepito Pérez';
$pepe->dni = 123456789;

$pepe->aprobar_redes();

echo "$pepe hoy no quiere ir a clase.";

?>
```

En el ejemplo, se nos mostrará la propiedad $nombre cuando hagamos referencia al objeto en general. Esto, combinado con un print_r() o un var_dump(), puede ayudar enormemente a la depuración de código.

8.2.6. __get()

__get se llama automáticamente si se intenta acceder a una propiedad que no esté definida en una clase. Sirve, junto con __set y __call, para construir clases que aparenten ser más de lo que son, que aparenten tener más métodos o propiedades de los que realmente tienen.

```
<?

class ficheros_configuracion
{
    public function __get($var_name)
    {
        if (file_exists($f = '/etc/'.$var_name) && is_readable($f))
            return file_get_contents($f);
        else
            return false;
    }

}

$fich = new ficheros_configuración;

echo $fich->hosts;
```

```
// Imprimirá el contenido del fichero /etc/hosts .

echo $fich->hwhurgsbsd;
// Esto devolverá falso, porque /etc/hwhurgsbsd probablemente no existe.

?>
```

8.2.7. __set()

__set funciona de manera análoga a __get: se llama automáticamente cuando se quiere asignar un valor a una propiedad no definida.

```
<?

class ficheros_configuracion
{
    public function __get($var_name)
    {
        if (file_exists($f = '/etc/'.$var_name) && is_readable($f))
            return file_get_contents($f);
        else
            return false;
    }

    public function __set($var_name,$valor)
    {
        if (file_exists($f = '/etc/'.$var_name) && is_writable($f))
            return file_put_contents($f);
        else
            return false;
    }
}

$fich = new ficheros_configuración;

$fich->hosts = $fich->hosts . "\n127.0.0.1 www.loquesea.com\n";
// Añade una entrada al fichero /etc/hosts.

?>
```

8.2.8. __call()

La última de las funciones mágicas (al menos, hasta que introduzcan más en PHP5) es __call. Similar a __get y __set, __call se llama automáticamente cuando se llama a un método no definido. __call() recibe dos argumentos: el nombre del método que se ha querido llamar, y un array con los argumentos. De esta manera, podremos tener clases con todos los métodos virtuales que queramos.

8.3. Patrones de diseño

Un patrón de diseño es una solución expresada normalmente en forma de una serie de clases programadas de una determinada manera, que resuelven una serie de problemas con los que nos podemos encontrar muy a menudo. A continuación veremos un par de situaciones en las que podemos aplicar patrones de diseño para tener una solución elegante.

8.3.1. Constantes y cosas estáticas

Una clase puede tener propiedades constantes o estáticas, y funciones estáticas. Una constante es precisamente eso, una propiedad cuyo valor es siempre el mismo en el entorno de una clase, y no puede cambiarse. Una propiedad o método estático es el mismo para todo el entorno de una clase, pero puede modificarse (propiedad) o llamarse (método).

8.3.2. Paamayim Nekudotayim

Para acceder al entorno de una clase (y, por tanto, a los métodos o propiedades estáticos o constantes), se usa el Paamayim Nekudotayim, más conocido como “dos puntos dos puntos”o “::”. El porqué de este nombre extraño se debe a que, por un lado, muchos de los creadores de PHP3 provienen de oriente medio y, por otro, son tan cachondos que, en el motor de PHP, le pusieron a “::”la traducción en hebreo de “double colon”(“doble dos puntos”). Otro nombre para este operador es el “Scope Resolution Operator”o “Operador de Resolución de ámbito/entorno/alcance”.

Usando :: se accede al entorno de una clase. Hay que tener en cuenta que hay dos clases “especiales”, que son “self”y “parent”. “self”se refiere al entorno de la clase en la que estoy (cuando se ejecuta un método de una clase, obviamente). “parent”se refiere al entorno de la clase padre, en el caso de que esté en un método de una clase que ha heredado ed otra (como se vió antes).

8.3.3. Solteros de oro

Un patrón muy común es el “soltero”(“singleton”). Se basa en que el programador no puede hacer instancias de una clase, declarando el constructor como privado (si acaso, se puede instanciar un objeto, pero sólo dentro de un método estático). Algunos autores consideran que un soltero es un soltero sólo si hay una instancia de sí mismo en una propiedad estática.

Los solteros se usan cuando no se necesitan instancias, o cuando se necesita un acceso global sencillo desde cualquier parte (a través del ::). Pueden usarse para crear manejadores de bases de datos o configuraciones globales de fácil acceso, entre otros usos.

Por ejemplo, podemos crear una clase que nos permita acceder a ficheros mediante métodos estáticos:

```
<?

class ficheros_configuracion
{
    private constant $directorio = '/etc/';

    private function __construct() {}

    static function leer($fichero)
    {
        $fichero = basename($fichero);
        if (file_exists($f = self::directorio . $fichero) && is_readable($f))
            return file_get_contents($f);
        else
            return false;
    }

    static function guardar($fichero,$contenido)
    {
        $fichero = basename($fichero); // Por seguridad
        if (file_exists($f = self::directorio . $fichero) && is_writable($f))
            return file_put_contents($f);
        else
            return false;
    }
}

ficheros_configuracion::guardar(
    'hosts', ficheros_configuracion::leer('hosts') . "\n127.0.0.1 www.loquesea.com\n"
```

```
    // Añade una entrada al fichero /etc/hosts.

?>
```

8.3.4. Fábricas

Las fábricas tienen un propósito sencillo: devolver un objeto de una clase que determina la fábrica, dependiendo de ciertas condiciones.

Por ejemplo, si tengo varias librerías que sirven para el mismo propósito (QT, GTK+ y Motif para objetos gráficos, por ejemplo), las cuales implementan algo común (una barra de desplazamiento, por ejemplo) pero de manera distinta (seguramente tenemos qt_scrollbar, gtk_scrollbar y motif_scrollbar), una fábrica me permitirá tener una barra de desplazamiento sin preocuparme de qué librería use por debajo mi aplicación.

En PHP quizás este no es un ejemplo descriptivo (ya que normalmente no interactúa con el usuario a través de una interfaz gráfica). Pero ilustraremos la idea de las fábricas con un ejemplo sencillo:

```
<?

class coche
{
    public function conducir()
    {
        return 'Estoy conduciendo un ' . get_class($this);
    }
}

class coche_fiat extends coche
{
    public static $precio = 8900;
}
class coche_citroen extends coche
{
    public static $precio = 7650;
}
class coche_mercedes extends coche
{
    public static $precio = 15750;
}

class concesionario
// Mi fábrica de coches
{
    static function compra_un_coche_caro($dinero)
    {
        if ($dinero >= coche_mercedes::$precio)
            return new coche_mercedes;
        elseif ($dinero >= coche_fiat::$precio)
            return new coche_fiat;
        elseif ($dinero >= coche_citroen::$precio)
            return new coche_citroen;
        else
        {
            echo "No tienes dinero suficiente";
            return -1;
        }
    }
}
```

```
$mi_coche = concesionario::compra_un_coche_caro(10000);

echo $mi_coche->conducir();

?>
```

8.3.5. Clases abstractas

En algunos casos (como en el ejemplo de los coches), es mejor evitar que un programador instancie dierta clase. Para ello, podemos definir una clase como abstracta. Dicha clase no se podrá instanciar, pero sí heredar. Si además se declara alguno de los métodos de esa clase como abstracto, cualquier clase que herede de ella la tiene que implementar de manera no abstracta, o convertirse a su vez en una clase abstracta. En otras palabras, cualquier clase que contenga un método abstracto debe ser abstracta. La manera de convertir una clase abstracta en una no abstracta es heredarla y reimplementar todos los métodos abstractos que pudiera tener.

```
<?

abstract class coche
{
    public function conducir()
    {
        return 'Estoy conduciendo un ' . get_class($this);
    }
}

class coche_fiat extends coche
{
    public static $precio = 8900;
}

?>
```

8.3.6. Interfaces

Un interfaz sigue la idea de una clase abstracta, pero de una forma muy específica: un interfaz es sólo una lista de métodos que una clase debe implementar (como mínimo) para acomodarse al interfaz.

```
<?

interface coche
{
    public function echar_gasolina($litros);
    public function recorrer($kilometros);
}

class coche_fiat implements coche
{
    public static $precio = 8900;
    private $gasolina = 0;
    const max_gasolina = 55;

    public function echar_gasolina($litros)
    {
        if ($gasolina < 0)
            return -1;
        $this->gasolina = ($this->gasolina + $litros < self::max_gasolina)
```

```
             ? $this->gasolina + $litros : self::max_gasolina;
            echo "He echado gasolina y ahora tengo {$this->gasolina} litros.<br/>";
        }

        public function recorrer($kilometros)
        {
            // Supongamos 9 litros a los 100 km
            if ($kilometros < 0)
                return -1;
            $this->gasolina -= $kilometros * 9 / 100;

            if ($this->gasolina <= 0)
            {
                $this->gasolina = 0;
                echo "Me he quedado sin gasolina<br/>";
            }
            else
            {
                echo "He recorrido $kilometros kilómetros sin problemas y me
                   quedan {$this->gasolina} litros.<br/>";
            }
        }
    }

    $mi_coche = new coche_fiat;

    $mi_coche->echar_gasolina(20);
    $mi_coche->recorrer(150);
    $mi_coche->recorrer(100);

    ?>
```

8.3.7. Es mi última palabra!

Un método declarado como “final”(“finam function loquesea()”) no puede reimplementarse en clases que hereden de ésta. Una clase definida como final (“final class loquesea”) no puede heredarse.

8.3.8. Type hinting (sobrecarga de funciones)

Aunque PHP5 tiene un tipado débil y dinámico, se pueden sobrecargar funciones y métodos, haciendo que una función esté definida si y sólo si se los parámetros que se le pasan son de una determinada clase. No se pueden sobrecargar funciones para distinguir entre tipos simples, como enteros o cadenas.

```
    <?

    class coche_fiat {}
    class coche_citroen {}

    class taller_citroen
    {
        static function arreglar(coche_citroen $coche)
        {
            echo "Vale, vuelve dentro de una semana y lo tienes listo<br/>";
        }
    }

    $mi_coche = new coche_fiat;
```

```
taller_citroen::arreglar($mi_coche);

?>
```

Aquí, al intentar arreglar algo que el taller no puede, se devolverá un error.

Capítulo 9

BBDD

9.1. ¿Porqué?

En muchas aplicaciones web, al crecer, llega un momento en el que trabajar con una cantidad grande de datos se hace inmanejable si se usan ficheros para almacenarlo todo. En estos casos, la solución más práctica es usar una Base de Datos para almacenar toda al información, y trabajar con ella de forma más eficiente y rápida (desde el punto de vista del tiempo de ejecución).

9.2. ¿Cuál?

En el mercado de los sistemas de bases de datos, hay relativamente pocos competidores. A continuación veremos los sistemas más comúnmente usados:

9.2.1. SQLite

SQLite nació hace poco como una alternativa a los "grandes"sistemas de BBDD. Lo que hace atractivo a SQLite es que almacena la BBDD en un fichero, lo cual elimina la necesidad de un servidor de BBDD en el ordenador. Esto simplifica sobremanera la instalación y distribución de una aplicación que use una BBDD por debajo. Si no queremos complicarnos la vida con instalaciones de software, SQLite es la mejor opción para aplicaciones con PHP5.

SQLite no tiene un gran motor de BBDD, sino tan sólo alrededor de ˜ 250 KiB de código en unas 30000 líneas de C. Esta ligereza hace que PHP5 incluya a SQLite dentro de una instalación estándar. Puede no tener todas las funcionalidades de un gran motor de SQL, pero es mucho más que suficiente para aplicaciones pequeñas y medianas.

Sin embargo, el mayor problema de SQLite es la eficiencia. Aunque es tremendamente rápido para cantidades pequeñas de datos (y extremadamente portable), en cuanto se trabaja de manera rutinaria con varios MiB de datos, es recomendable pensar en otra alternativa.

Más información en `http://www.sqlite.org/`

9.2.2. MySQL

Uno de los pilares de la plataforma LAMP (Linux + Apache + Mysql + PHP), MySQL es un sistema servidor de bases de datos grauito, de código libre, altamente escalable, eficiente y con muchísima popularidad entre las aplicaciones desarrolladas tanto en PHP como en otros lenguajes. Cuenta con una implementación completa de ANSI SQL, incluyendo control de transacciones.

MySQL AB, la empresa detrás de MySQL, ofrece formación y herramientas avanzadas para diversos propósitos. Con esto, MySQL es una excelente elección para aplicaciones medianas o grandes.

Más información en `http://www.mysql.com/`

9.2.3. Oracle 10

Oracle, la corporación gigante en cuestión de BBDD y su motor de BBDD, Oracle 10. Oracle 10 es más o menos parecido a MySQL (servidor de BBDD, escalable, eficiente, soporta alta carga, distintos tipos de tablas, transacciones, etc), pero la principal diferencia es que Oracle es muy, muy caro. Es la única opción disponible si a tu jefe le ha hecho una visita un comercial de Oracle.

Quien quiera comprar una licencia de Oracle 10 debe dirigirse a `http://www.oracle.com/`

9.2.4. PostgreSQL

Otro gran motor de BBDD grauito y de código libre, con grandes funcionalidades, y también ampliamente usado en multiud de aplicaciones. De nuevo, una buena opción para aplicaciones medianas o grandes.

Más información en `http://www.postgresql.org/`

9.2.5. Otros competidores

Otras opciones menos populares son:

MS SQL Todos adoramos a microsoft y elegiríamos su tecnología (si no fuera porque tenemos más opciones, claro está).

dBase/DB2 Fueron sistemas de BBDD muy usados, pero hace 20 años. Aun es posible verlos en aplicaciones muy antiguas.

Mini SQL (mSQL) Un sistema ligero pero poco popular.
`http://www.hughes.com.au/products/msql/`

Ovrimos SQL server otro sistema poco conocido.

9.2.6. ¿Cuál elijo?

Comparar los distintos grandes sistemas de BBDD es entrar en una contínua guerra religiosa entre los fans de un sistema y los devotos de otro, que distuten calurosamente sobre la funcionalidad que tiene su sistema pero de la que carece el competidor. Por el momento, usaremos SQLite para los pequeños ejemplos. A la hora de elegir un sistema mayor, hay que estudiar pacientemente las alternativas y elegir la que mejor se adapte a nuestras necesidades y posibilidades.

Sólo veremos la funcionalidad que me permite interactuar con SQLite, desde PHP5. Para documentación sobre otros sistemas de BBDD, véase el manual de PHP.

9.3. ¿Cómo?

9.3.1. Funcionalidades básicas

Todo sistema de BBDD basado en SQL implementa una API con, al menos, la siguiente funcionalidad:

- Abrir una conexión a una BDD.
- Ejecutar una query SQL.
- Acceder a los resultados de una query SQL.
- Control de errores en sentencias SQL.
- Cerrar la conexión a la BDD.

9.3.2. Funciones de SQLite

En SQLite usaremos:

- sqlite_open() para abrir una BDD existente (o crear una vacía), siempre en un fichero.
- sqlite_query() para ejecutar queries SQL.
- sqlite_fetch_array() y sqlite_num_rows() para saber el resultado de una query..
- sqlite_last_error() para control de errores.
- sqlite_close() para cerrar la BDD.

9.3.3. Conexiones persistentes

Recordatorio: el flujo de ejecución de una aplicación web (véase "Paradigma de la programación web") no es contínuo, sino discreto. A cada página que un visitante acceda con su navegador web (o método similar), se ejecutará nuestro programa en PHP. Cuando nuestra aplicación accede a una BBDD, normalmente se abre una conexión a la BBDD al principio del programa y se cierra la conexión al final del programa.

Obviamente, esto no es muy eficiente, sobre todo cuando vamos a tener muchas visitas en un tiempo relativamente corto. Para aliviar esto, se usan las conexiones persistentes a BBDD. La diferencia entre abrir una conexión normal (sqlite_open) y una persistente (sqlite_popen) es que la conexión persistente, al cerrarse, queda abierta en el motor de PHP, con lo que se acelera sobremanera la siguiente conexión a la BBDD. La conexión se cierra a nivel de programa, pero queda abierta a nivel de intérprete, lista para posteriores conexiones.

Por desgracia, el dejar abierta la conexión de BBDD sólo funciona si nuestro intérprete de PHP5 es un módulo de apache; si ejecutamos PHP como cgi o desde una consola, no tenemos un proceso a más bajo nivel capaz de mantener abierta una conexión aunque el programa la haya cerrado.

9.4. ¿Cuándo?

Ya basta de teoría; empecemos ahora mismo a ver algún ejemplo.

9.4.1. Vendiendo pan, primera

Ilustraremos el uso de una BBDD sencilla con una pequeña aplicación para una supuesta panadería que quiere dar a conocer los productos que vende a través de su página web. Esta aplicación consta de las siguientes páginas:

crear_bdd.php Crea las estructuras necesarias para almacenar datos en la BBDD (en este caso, una única tabla). Sólo ha de ejecutarse una vez, cuando la BBDD todavía no ha sido creada.

index.php La única página que verán los visitantes. En ella se muestra un listado de los productos que están a la venta.

admin.php Desde aquí se accede al alta y baja de productos. Esta página no modifica nada, sólo es un punto de entrada a las páginas que sí modifican los datos.

alta.php Esta página recibe datos desde admin.php, a través de un formulario, y procesa el alta de un producto.

baja.php Esta página recibe datos desde admin.php, a través de un enlace, y procesa la baja de un producto.

(El código se incluye aparte, al ser de tamaño considerable)

En esta aplicación sencilla hay que tener muy claro el flujo de datos entre las páginas que componen el área de administración. Mediante admin.php, un empleado de la panadería puede preparar y componer una petición a la BBDD, que se procesará en otra página (alta o baja). El proceso es siempre así:

1. El servidor le da al usuario una página donde, mediante un formulario o una serie de enlaces, se le da la opción de introducir datos para después integrarlos en la BBDD o obrar según ellos.

2. El usuario rellena y envía el formulario, o pulsa un enlace, enviando datos de nuevo al servidor.

3. El sevidor recoge los datos que le ha enviado el usuario, los analiza y valida, y obra en consecuencia (modificando, si cabe, la BBDD).

9.4.2. Vendiendo pan, segunda

Ahora vamos a mejorar la aplicación que ya teníamos. Añadiremos una nueva columna a la tabla (descripción), y haremos una pequeña capa de abstracción de BBDD para no tener que arrastrar la conexión a la BBDD ($bd) durante la ejecución. También añadiremos otra página, para que los visitantes puedan ver la descripción de un producto.

En el código (que está aparte de la documentación) veremos cómo se puede usar una clase con métodos estáticos para hacer una abstracción correcta y accesible desde cualquier punto del código.

9.4.3. Con esto basta

Podríamos ampliar los ejemplos hasta tener un sistema de gestión de panaderías completo, incluyendo control de nóminas y facturación, con venta a domicilio y pagos con tarjeta de crédito, pero ese no es el caso.

Los ejemplos están ahí para ver cómo funciona una aplicación web con base de datos muy sencilla. Para probar los ejemplos, únicamente hace falta copiarlos en un subdirectorio del docroot de nuestro servidor web (normalmente /var/www si usamos linux y apache), y dar permisos para que PHP (más bien, las librerías de SQLite) pueda escribir el fichero de BBDD en el subdirectorio de la aplicación.

9.5. ¿Quién?

Muchas veces podemos tener gente “graciosilla”que se dedique a maltratar nuestra aplicación web. Ahora veremos los riesgos de seguridad básicos contra los que hay que estar avisado y prevenido.

9.5.1. Inyección de SQL

Una inyección de código se basa en que la aplicación no comprueba correctamente los datos de entrada, y por tanto es posible introducir sentencias SQL enteras a través de los datos que se le pasan al servidor web.

Por ejemplo, consideremos el siguiente código presente en la aplicación de ejemplo:

```
<?

bdd::query("select * from productos where id='{$_GET['id']}'");

?>
```

Esto es una sola sentecia SQL, donde se expande una variable que llega desde el navegador, ¿verdad? ¿o quizás es algo más que eso?

Supongamos que la variable le pasamos una variable al programa usando una URL del estilo de `"?id=0';delete * from productos; select * from productos where id='";` esto hará que, en nuestro código se expanda este valor dentro de la cadena y se ejecute algo como:

```
<?

// Normalmente
bdd::query("select * from productos where id='0'");

// Con inyección de SQL
bdd::query("select * from productos where id='0';delete * from productos;
            select * from productos where id=''");

?>
```

Con lo que nuestra sentencia SQL se convierte en tres, borrando todos los datos.

(En realidad, la tercera sencencia está sólo para que la segunda comilla simple, que está después de la variable que se expande, no dé error al ser parseada por el motor de SQL.)

¿Cómo protegerse ante ésto? Fácil: comprobando los datos de entrada. Cualquier dato que esté destinado a entrar en una query SQL tiene que ser pasado a numérico (mediante un type casting explícito, si el dato esperado es numérico) o bien hay que escapar todos los caracteres ”maliciosos”que pueda tener (mediante funciones como sqlite_escape_string(), si el dato es una cadena de texto).

9.5.2. Autentificación

Otro error, pero menos común, es dejar a la vista de cualquier usuario páginas que permitan la edición de datos que se supone que un usuario no debería poder tocar. En nuestro ejemplo, cualquier usuario podría visitar admin.php (alguien que se aburra mucho puede empezar a teclear painas al azar a ver si existen) y modificar todo nuestro sistema.

La solución pasa por autentificar al usuario; ya sea por usuario/contraseña, IP desde la que se visita, CAPTCHAs, o métodos biométricos o criptográficos. Cada cual tiene sus pros y sus contras; y veremos algo más sobre la autentificación de usuarios en el siguiente capítulo.

Capítulo 10

Autentificación y sesiones

10.1. ¿Nos conocemos de antes?

10.1.1. Memoria de pez

A riesgo de caer pesado, repetiré algo que ya se ha dicho antes: en la programación web, cada página que se visita equivale a une ejecución de un programa. Los más avispados habrán observado que esto conlleva un problema: un visitante o usuario habitual de una aplicación web visitará decenas de páginas y, por tanto, hará que en el servidor web se ejecuten decenas de programas en momentos de tiempo separados. ¿Cómo podemos lograr que estos programas, que se ejecutan de manera separada, compartan información (como, por ejemplo, la identificación de un usuario o su carrito en una aplicación de tienda on-line)?

10.1.2. Leche y galletas

Allá por los principios de los 90, se dió una primera solución al problema de arrastrar información entre distintas páginas de una misma aplicación web: las Cookies.

Una cookie es un par nombre-valor que conforma una variable, y que el navegador envía a cada página que visita en el servidor que le envió la cookie en primer lugar. Una cookie es la manera de decir “cada vez que me visites una página, envíame esta información para que yo sepa quién eres”.

En PHP, para mandar una cookie al navegador, se utiliza la función setcookie() (o setrawcookie() en raras ocasiones). Y para ver si el cliente nos manda datos de una cookie se mira la variable superglobal $_COOKIE, como veremos en el ejemplo:

```
<?

if (!$_COOKIE['primera_visita'])
    $texto = 'Vaya vaya, veo que es tu primera visita';
else
    $texto = 'Bienvenido de nuevo';

setcookie('primera_visita',1);
// Esto quiere decir "cada vez que me visites, mándame una variable
// 'primera_visita' con valor '1'".
// Cuando el navegador visite otra página dentro del servidor, le enviará
// esta información, que un programa en HPP recogerá a través de la variable
// superglobal $_COOKIE.

echo $texto;

?>
```

Es importante enviar las cookies antes de hacer ningún echo(), y antes también de cualquier texto fuera de $<? \ ? >$. Esto quiere decir que si nuestro programa (o alguno de los scripts de los que haga un include()) tiene una línea o un

espacio en blanco antes del primer <?, la cookie no se enviará correctamente al navegador. Veremos porqué al hablar de cabeceras HTTP.

(Una cookie, además del nombre y el valor, se compone también del servidor web para el que es válido, la caducidad, si se envía sólo a ciertas páginas en un path concreto del servidor, y si se aplica sólo a páginas pedidas por https).

10.1.3. Sobrepeso

Las cookies fueron una buena idea en un buen momento, pero con el auge y el crecimiento descontrolado de aplicaciones web por todos lados (a veces mal programadas), se hicieron visibles los fallos de este mecanismo: por un lado, por cada página que un usuario visita, se le mandan al servidor web todas las cookies que el servidor haya mandado al navegador en un primer momento; en algunas aplicaciones web esto puede suponer una sobrecarga de unos 10 o 15 Kb, que puede superar a la página en sí. Por otro lado, un usuario avispado con un navegador decente (hoy en día, cualquier cosa que no sea un IE) puede editar las cookies que manda al servidor y así engañar a la aplicación web.

Por eso, hoy en día, el uso intensivo de cookies ha dejado paso al uso intensivo de sesiones.

10.2. Sesiones

10.2.1. No más ping-pong

Las sesiones se basan de nuevo en el principio de que en navegador envíe información a cada página que visite en el servidor, de manera que se corrigen los problemas de usar cookies para todo. En primer lugar, el navegador mandará un solo par variable-valor al servidor (normalmente una variable con nombre “PHPSESSID”y un md5sum de un número aleatorio como valor). En segundo lugar, los datos que pasan de página a página (de ejecución de programa a ejecución de programa) se almacenan en el servidor y no en el navegador (en el cliente): un usuario listillo no puede cambiar estos datos que pasan de programa en programa.

¿Cómo se las arregla el mecanismo de sesiones para que el navegador mande un par variable-valor a cada página? Usando *una* cookie, de la que el programador no debe preocuparse en absoluto. Además, si el navegador no acepta cookies, PHP detectará esto, y añadirá un par variable-valor a cada petición que se haga por get, o a cada formulario que se envíe por post, para que este par variable-valor siempre llegue bien.

¿Cómo almaceno datos de sesión? Simplemente usando la variable superglobal $_SESSION. Los contenidos de esta variable serán idénticos para cada ejecución de cada página si se hacen desde el mismo navegador, y distintas para distintos navegadores. Es una manera mucho más natural (y mucho más segura) que las cookies para que el programador almacene datos persistentes entre las ejecuciones de distintas páginas de una misma aplicación web. A los elementos del array superglobal $_SESSION se les denomina “variables de sesión”.

```
<?

if (!$_SESSION['primera_visita'])
    echo 'Vaya vaya, veo que es tu primera visita';
else
    echo 'Bienvenido de nuevo';

$_SESSION['primera_visita'] = 1;

?>
```

Una funcionalidad típica es permitir acceso a determinadas páginas si y sólo si el visitante de la web ha introducido usuario y password. Suponiendo que tengamos una página con un formulario que pida estos dos datos, la comprobación se haría más o menos así:

```
<?

if ($_POST['usuario']=='scott' && $_POST['passwd']=='tiger')
{
    $_SESSION['usuario_autentificado'] = TRUE;
    $_SESSION['usuario'] = $_POST['usuario'];
}
else
```

```
{
    $_SESSION['usuario_autentificado'] = FALSE;
    unset ($_SESSION['usuario']);
}


if  ($_SESSION['usuario_autentificado'])
{
    // Dar la bienvenida, mostrar menú de operaciones sólo para usuarios, etc
}
else
{
    // Redirigir al visitante a la página de login
}

?>
```

Obviamente, el método deja bastante que desear cuando hay varios usuarios registrados. Otra posible opción, si tenemos una base de datos con una tabla que contenga información sobre usuarios y passwords, sería:

```
<?

$bd = sqlite_open('base_de_datos.bdd');

$r = sqlite_query("select * from usuario where usuario='{$_POST['usuario']}'
     and password = '{$_POST['passwd']}'");

if (sqlite_fetch_array($r))  // Si la consulta me devuelve una fila válida...
{
    $_SESSION['usuario_autentificado'] = TRUE;
    $_SESSION['usuario'] = $_POST['usuario'];
}
else
{
    $_SESSION['usuario_autentificado'] = FALSE;
    unset ($_SESSION['usuario']);
}


if  ($_SESSION['usuario_autentificado'])
{
    // Dar la bienvenida, mostrar menú de operaciones sólo para usuarios, etc
}
else
{
    // Redirigir al visitante a la página de login
}

?>
```

Este método tendría dos fallos fundamentales: primero, los passwords se almacenan en texto claro en la base de datos (si nos roban la base de datos, alguien tendría los passwords de muchas personas sin mayor complicación). Segundo, al no comprobar las variables de entrada de la query, somos susceptibles de sufrir una inyección de SQL. El primer problema se soluciona almacenando hashes de los passwords (véase la función md5() o las funciones de criptografía mediante algoritmos sha1 y similares); el segundo problema se resuelve aplicando sqlite_escape_string() sobre las variables que provengan del navegador y que se vayan a usar en la query.

En cualquier otra página de nuestra aplicación web, tan sólo tendríamos que comprobar la variable $_SESSION['usuario_autentificado '], y redirigir al usuario a la página de login si no tiene valor "verdadero". Por supuesto, en $_SESSION

podemos almacenar cualquier otra información que sea de importancia para la aplicación y extensible sólo a un usuario: contenidos del carrito en una tienda on-line, estilo gráfico de la web en caso de que podamos elegir entre varios, preferencias, soluciones a acertijos y *CAPTCHAs* (`http://www.captcha.net`), etcétera.

10.2.2. ¡Adivina!

A continuación se expone un ejemplo de cómo utilizar variables de sesión para almacenar un dato que queremos mantener entre distintas ejecuciones de una misma página web, pero que no queremos que el usuario pueda ver ni modificar, en este caso la respuesta a un pequeño acertijo:

```
<?

// To Do: Añadir doctype y cabeceras html!

session_start();

if (!isset($_SESSION['secreto']) || $_REQUEST['accion']=='reset')
{
    $_SESSION['secreto']  = mt_rand(1,100);
    // mt_rand = numero aleatorio
    $_SESSION['intentos'] = 0;
}

if ($_REQUEST['numero'] == $_SESSION['secreto'])
{
    echo "Enhorabuena, has adivinado el número tras {$_SESSION['intentos']}
      intentos. <br><br><a href='{$_SERVER['PHP_SELF']}?accion=reset'>
      Empezar de nuevo</a>";
}
else
{
    $_SESSION['intentos']++;

    echo "Intenta adivinar el número que he pensado (entre 1 y 100)!";

    // Debug...
    // echo " Pssst, no se lo digas a nadie: es el {$_SESSION['secreto']}";

    if (isset($_REQUEST['numero']))
        if ($_SESSION['secreto'] < $_REQUEST['numero'])
            echo " El número que tengo en mente es \textbf{menor} que
                   {$_REQUEST['numero']}";
        else
            echo " El número que tengo en mente es \textbf{mayor} que
                   {$_REQUEST['numero']}";

    echo "<br><br>Creo que es el: <form method=post
 action={$_SERVER['PHP_SELF']}><input type=text lenght=3 size=3 name=numero>
 <input type=submit></form>";

}

?>
```

10.2.3. Almacenando sesiones

Comentar que por defecto se almacenan automáticamente en un fichero de texto plano.

¿Cómo funciona esto de las sesiones? ¿Porqué en un ejemplo aparece la función session_start() y en los demás no? ¿A dónde vamos? ¿De dónde venimos? Los interesados en conocer las respuestas a las dos últimas preguntas, acudan a un filósofo o a una secta religiosa. Los interesados en conocer las respuestas a las dos primeras, sigan leyendo.

Las variables de sesión (los contenidos de $_SESSION) se guardan en el servidor, normalmente en un fichero plano o directamente en memoria si es posible, en una tabla que relaciona un identificador de sesión (normalmente un md5 de un número aleatorio) con un conjunto de variables de sesión. Lo que el motor de PHP debe hacer a cada ejecución es comprobar si el navegador le ha enviado un identificador de sesión, y en caso afirmativo recuperar las variables de sesión correspondientes.

El problema es que esta recuperación de variables de sesión no es automática en algunos casos. Lo es si la directiva de configuración "session.auto_start"está a 1, pero si no lo está, entonces hay que pedir explícitamente que se haga este proceso con session_start().

Si tenemos cientos o miles de visitantes, también podemos almacenar las variables de sesión en una base de datos, en vez de en ficheros planos, para aumentar la eficiencia y reducir la carga. También lo podemos hacer por motivos de privacidad (encriptar datos) o de replicación (si no podemos fiarnos de un sistema de fichero, al estar la aplicación repartida en varios servidores para balancear carga, por ejemplo). Esto se puede hacer usando session_set_save_handler() antes de session_start(). De esta manera, cada vez que un navegador mande un identificador de sesión, se consultará la base de datos y se sacarán de ahí las varaibles de sesión. Los más atrevidos pueden experimentar con esto y construir de manera automática variables de sesión mediante session_set_save_handler().

Toda la información referente a este aspecto, así como directivas de configuración, otras funciones para trabajar con sesiones de manera intensiva y algunos ejemplos se pueden consultar en el manual oficial de PHP, en la sección de funciones de sesión.

10.3. 403 Forbidden

10.3.1. Cabeceras HTTP

Comentar otras aplicaciones: cacheado, otros tipos de contenido.

En la década de los 90, además de añadir cookies al protocolo HTTP, se diseñó la manera de autentificar a un usuario usando únicamente HTTP, a nivel de servidor web. La manera de hacerlo es que el servidor tome pares usuario/password de algún sitio (en apache, de la configuración o de ficheros .htaccess), y devuelva una cabecera HTTP para que el navegador pida usuario/password y vuelva a hacer la petición. En concreto, podríamos tener el siguiente diálogo entre navegador y servidor web si visitamos www.dondesea.org/loquesea.php. En primer lugar, el navegador dice:

```
GET /loquesea.php HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/5.0 (compatible; Konqueror/3.3; Linux 2.6.10-1-k7; i686; es)
KHTML/3.3.2 (like Gecko)
Accept: text/html, image/jpeg, image/png, text/*, image/*, */*
Accept-Encoding: x-gzip, x-deflate, gzip, deflate
Accept-Charset: iso-8859-15, utf-8;q=0.5, *;q=0.5
Accept-Language: es, en
Host: www.dondesea.org
```

Aquí el navegador pide la página, y además dice quién es, y qué cosas puede admitir como respuesta (texto plano en distintos formatos e idiomas preferentemente, y con qué compresión transparente para el usuario, por ejemplo).

En caso de que http://www.dondesea.org/loquesea.php necesite autentificación HTTP, una respuesta típica podría ser:

```
HTTP/1.1 401 Authorization Required
Date: Tue, 01 Mar 2005 23:01:14 GMT
Server: Apache/2.0.52 (Gentoo/Linux)
WWW-Authenticate: Basic realm="Aplicación de lo que sea"
Content-Length: 492
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1
```

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>401 Authorization Required</title>
</head><body>
<h1>Authorization Required</h1>
This server could not verify that you are authorized to access the document
requested.  Either you supplied the wrong credentials (e.g., bad password), or
your browser doesn\'t understand how to supply the credentials required.</p>
<hr>
<address>Apache/2.0.52 (Gentoo/Linux) Server at www.dondesea.org Port 80
</address>
</body></html>
```

Es interesante notar que el servidor responde en dos fases: una primera de cabeceras, las cuales son generadas automáticamente por el servidor web, y normalmente un diseñador o programador no debe preocuparse de ellas. Después, tras dos saltos de línea consecutivos, la página web en sí (o, si se pide una imagen, la imagen, etcétera).

El navegador, al ver una cabecera “401 Authorization Required”, pedirá al usuario un usuario y password, y volverá a pedir la página al servidor, pero esta vez autentificándose:

```
GET /loquesea.php HTTP/1.1
Connection: Keep-Alive
User-Agent: Mozilla/5.0 (compatible; Konqueror/3.3; Linux 2.6.10-1-k7; i686; es)
KHTML/3.3.2 (like Gecko)
Accept: text/html, image/jpeg, image/png, text/*, image/*, */*
Accept-Encoding: x-gzip, x-deflate, gzip, deflate
Accept-Charset: iso-8859-15, utf-8;q=0.5, *;q=0.5
Accept-Language: es, en
Host: www.dondesea.org
Authorization: Basic uYT6OTefMjeyoXc
```

El usuario y password van en la línea “Authorization”, codificados sencillamente en base64 (véase base64_decode() y base64_encode()), separados por dos puntos. Esto no es un método muy seguro de transmitir información como usuario y password, así que ojito con los sniffers, y usad siempre que podáis HTTP sobre SSL (el conocido ”https://...”).

Si el servidor web ve que el usuario/password son correctos, enviará al navegador algo así como:

```
HTTP/1.1 200 OK
Date: Tue, 01 Mar 2005 23:01:15 GMT
Server: Apache/2.0.52 (Gentoo/Linux)
Last-Modified: Thu, 30 Dec 2004 02:56:31 GMT
ETag: "5ba9d-24e-6ee115c0"
Accept-Ranges: bytes
Content-Length: 590
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=ISO-8859-1

<?xml version="1.0" encoding="latin-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional/EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional">
<html>
<head>
</head>
<body>
```

```
Hola! Bienvenido a www.dondesea.org/loquesea.php !

</body>
</html>
```

Ahora bien, aunque normalmente no vamos a hacer nada con las cabeceras (PHP y el servidor web hacen el trabajo por nosotros), podemos tocarlas con la función header(), que debe llamarse antes de que se haya enviado al navegador el principio de la página web, puesto que en tal caso ya se habrán enviado las cabeceras y los dos saltos de línea, siendo imposible mandar otra cabecera después. Esto quiere decir que si se ejecuta un echo() (o algo con función similar, como print(), print_r() o var_dump()), o ya se ha devuelto parte de la página web si teníamos algo dentro de $<? \ ? >$, no es posible enviar cabeceras, y PHP devolverá un error.

¿Cómo podemos usar autentificación en una aplicación web com PHP? Con algo que tenga más o menos esta pinta:

```
<?

$usuario = sqlite_escape_string($_SERVER['PHP_AUTH_USER']);
$passwd  = sqlite_escape_string($_SERVER['PHP_AUTH_PW']);

$bd = sqlite_open('base_de_datos.bdd');

$r = sqlite_query("select * from usuario where
            usuario='$usuario' and password = '$passwd'");

if (!sqlite_fetch_array($r))
// Si la consulta *no* me devuelve una fila válida...
{
    header('WWW-Authenticate: Basic realm="My Realm"');
    header('HTTP/1.0 401 Unauthorized');
    echo '<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>401 Authorization Required</title>
</head><body>
<h1>Authorization Required</h1>
Anda, machote, la próxima vez pon el usuario y password correctos.
</body></html>';
    die();
}
else
{
    echo "<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>Bienvenido</title>
</head><body>
<h1>Hola $usuario</h1>
Bienvenido a www.dondesea.org
</body></html>";
}

// Resto de la página

?>
```

Y habría que hacer este tipo de comprobación al inicio de cada página que necesite autentificación. Una pista: es buena idea usar include()s.

10.3.2. Programando con cabeza

¿Qué más cosas podemos hacer con cabeceras HTTP? Entre las más comunes se incluyen: por una parte, el modificar la fecha de creación y enviar directivas no-cache para evitar (en la medida de lo posible) que un navegador mantenga una copia de la página cacheada. Por otra, mandar una cabecera "Location:"para hacer que el navegador se redirija autoaticamente a otra página. Y por otra, mandar una cabecera "Content-type:"si no queremos devolver una página web (html o xhtml) sino una imagen, un pdf, u otro tipo de documento.

Hay que tener en cuenta que cuando se manda una cabecera, el resto de ellas puede modificarse para adaptarse a ella. Por ejemplo, sin incluímos una cabecera "Location:", el servidor no devolverá un "200 OK"sino un "302 Found". Si mandamos una cabecera "Content-type:", no se mandará la cabecera "Content-type:"habitual, ni se mandarán dos cabeceras de este tipo.

Sobre el evitar que el navegador mantenga en caché una página, hay que decir que cada navegador (sobre todo IE) tiene sus manías y su manera de implementar esa funcionalidad (o incluso evitarla). Con respecto a la cabecera "Location:", hay que decir que es una manera estándar (y mucho, mucho más limpia que usar <meta>s de HTML o Javascript) de que los bots y crawlers se redirijan solos (véanse las funciones de CURL y la opción CURLOPT_FOLLOWLOCATION).

Pero ¿cómo es que mediante PHP voy a devolver algo que no sea HTML? En dos casos: cuando genere dinámicamente una imagen (véanse las funciones de tratamiento de imágenes mediante el uso de la librería gd), un pdf, un ps, un swf (flash) o un fichero comprimido mediante las funciones a tal efecto; o cuando queremos devolver los contenidos de un fichero (el típico "download.php?file=loquesea.zip").

Un breve ejemplo para ilustrar cómo devolver ficheros sería el siguiente:

```
<?

// Supongamos download.php?file=nombre_de_fichero

// Evitamos que podamos devolver algo que está en otro directorio distinto al
// que queremos

$fichero = basename($_GET['file']);
// con basename() obtenemos el nombre de fichero, a partir de la última barra.
// De esta manera nos evitamos muchos problemas.

$fichero = "/home/pepe/downloads/".$fichero;
// Supongamos que nos podemos bajar lo que esté ahí (y el servidor web tenga
// permisos para leer)

if ( !(file_exists($fichero) && is_readable($fichero)( )
{
    die('Fichero inválido');
}

$mimetype = shell_exec("file -bi $fichero");
// El programa externo "file" en unix me devolverá una cadena del tipo
// "image/png" que puedo usar directamente en la cabecera

header("Content-type: $mimetype");

// Y ahora enchufo el fichero completo al navegador
fpassthru($fichero);

?>
```

Capítulo 11

XML

11.1. Panacea

11.1.1. Una solución a todos los males

Hace ya unos cuantos años, se dijo que ATM sería la solución que unificaría a todas las tecnologías de red (ya fueran Ethernet, X25, fibra óptica, IP sobre paloma mensajera o un vagón de tren cargado de cintas magnéticas). De igual manera, hace unos pocos años, se dijo que XML sería la solución para los infinitos formatos de intercambio de información.

Al usar una sintaxis unificada y una manera más o menos sencilla de definir las posibles maneras de estructurar un tipo de información (los DTDs), se supone que XML es capaz de almacenar cualquier tipo de información fácilmente, de manera legible.

11.1.2. ¿O no?

Sin embargo, XML tiene sus desventajas. Una de ellas es que, aunque el formato estuvo listo en poco tiempo, no hay demasiadas herramientas sencillas y potentes para trabajar con XML: muchas de ellas son librerías de programación que implican recorrer una estructura jerárquica "a mano ". Otro problema es el espacio desaprovechado: para algunas aplicaciones que intercambian muy poca cantidad de datos, el encapsularlo dentro de XML puede suponer un aumento del 500 % en lo que a tamaño (y tiempo de transmisión) se refiere.

Moraleja: no es conveniente usar XML si existe otro formato, fácil de interpretar y de trabajar con él, que haga el trabajo (véanse serializaciones de estructuras de datos, por ejemplo). Sí es conveniente usar XML si queremos desarrollar un formato más o menos complejo que sea legible por una persona y que sea más o menos fácil de manejar mediante nuestros propios programas.

11.2. XML Simple: SimpleXML

Algunas herramientas para manejar XML, aunque potentes, pueden resultar pesadas de cara al programador. Pero con la entrada de PHP5, y la inclusión de simpleXML, trabajar con XML resulta fácil hasta puntos insospechados.

SimpleXML consta de una clase, "simplexml", que modela un nodo de XML, a cuyos atributos y nodos hijos se puede acceder con suma facilidad.

11.2.1. Lee y parsea

Para parsear un fichero en XML sólo hace falta instanciar un objeto de la clase simplexml. Usaremos una receta de pollo a la vasca en formato RecipeML que podemos encontrar en `http://dsquirrel.tripod.com/recipeml/indexrecipes.zip`:

```
<?

$fichero = 'Basque_Chicken.xml';

$xml = simplexml_load_file($fichero) or die ('Unable to load XML file!');
```

```
echo '<pre>';
print_r($xml);
echo '</pre>';

?>
```

Dentro de un nodo de XML (cualquier fragmento dentro de un tag de apertura <> y uno de cierre <>), podemos acceder a los nodos hijos como si fueran atributos del objeto; y podemos acceder a las propiedades del nodo (el "href"dentro de un <a href='http://loquesea.org'>) como elementos de un array (por ejemplo, $xml['href']):

```
<?

$fichero = 'Basque_Chicken.xml';

$xml = simplexml_load_file($fichero) or die ('Unable to load XML file!');

echo "Versión de RecipeML: {$xml['version']} <br>";
echo "Título de la receta: {$xml->recipe->head->title} <br>";

?>
```

11.2.2. ¡Busca, busca!

Cuando en un nodo de XML hay varios nodos hijos del mismo tipo, podemos recorrerlos como si de un array se tratara. Por ejemplo, si dentro del nodo $< ingredients >$ hay varios nodos $< ing >$:

```
<?

$fichero = 'Basque_Chicken.xml';

$xml = simplexml_load_file($fichero) or die ('Unable to load XML file!');

$tiene_sal = false;
foreach( $xml->recipe->ingredients->ing as $ingredient )
{
    $num_ingredientes++;
    if ($ingredient->item == 'Salt')
        $tiene_sal = true;
}

echo "Esta receta tiene $num_ingredientes ingredientes";
if ($tiene_sal)
    echo "{$xml->recipe->head->title} contiene sal.<br>";

?>
```

11.3. ¿Y ya está?

Básicamente esto es lo necesraio para parsear de manera sencilla información en XML. Aunque se puede hacer de maneras más complejas y potentes, el hecho de que la clase implemente propiedades virtuales y funcionalidad para aparentar ser un array y ser recorrida facilita mucho el navegar por la estructura del árbol de nodos.

11.3.1. Modificando información

Lo que se puede hacer sin ningún problema es cambiar la información de un nodo de simplexml (asignando un nuevo valor) y regenerando un fichero en XML a partir del nodo raíz. Por ejemplo, vamos a doblar la cantidad de sal en nuestra receta y después guardar todo en un nuevo fichero XML:

```
<?

$fichero = 'Basque_Chicken.xml';

$xml = simplexml_load_file($fichero) or die ('Unable to load XML file!');

foreach( $xml->recipe->ingredients->ing as & $ingredient )
{
    if ($ingredient->item == 'Salt')
    {
        $ingredient->amt->qty *= 2;
    }
}

$xml->asXML('Basque_Chicken_double_salt.xml');

?>
```

11.3.2. Creando XML

Lo que SimpleXML no permite (todavía) es la creación de nodos "a mano ". Sin embargo, gracias a la relativa simplicidad del formato XML, podemos generar contenidos directamente. A veces, la utilización de librerías o herramientas de XML puede resultar más pesado que la creación del propio XML de manera directa.

Capítulo 12

Tratamiento de errores

Como en cualquier lenguaje, en PHP5 se cometen fallos. En este capítulo veremos cómo se clasifican los distintos tipos de error, cómo atraparlos para conseguir más información o para ocultar el error al usuario final, y una estructura al "estilo Java "para tratamiento de errores.

12.1. Tipos de errores

12.1.1. No es tan grave como parece

Los errores se dividen en tres clases principales: notas (notice), avisos (warning) y errores propiamente dichos (error).

Las notas, o "notices ", no son errores, sino eventos totalmente normales, no perjudiciales, que el motor de php capta y procesa para la ejecución normal del código. El ejemplo más claro de notice que se puede generar es el acceso de lectura a una variable no declarada anteriormente. Esto, en otros lenguajes, produciría un error completo, pero en PHP esto hace que el motor interprete la variable como NULL y que todo prosiga su curso normal. Con la configuración predeterminada de PHP5, los notices no generan ningún mensaje de error por salida estándar.

Los avisos o "warnings "deben tenerse en cuenta, aunque suelen indicar fallos en la programación. Un warning nunca detendrá la ejecución de un programa, pero generará un mensaje de error por salida estándar. Algunos ejemplos de warning incluyen el llamar a una función con un número de parámetros incorrecto, o el intentar abrir un fichero inexistente o una conexión a base de datos errónea. Es una práctica más o menos común usar el operador de supresión de errores (la arroba) en aquellas líneas de código que sepamos que pueden generar un warning, pero siempre y cuando tratemos el posible error convenientemente. Por ejemplo:

```
<?

$f = @fopen('/tmp/un_fichero_que_no_existe','r');

if (!$f)
    echo 'No he podido abrir el fichero';

?>
```

Aunque es preferible evitar el uso del operador de supresión de errores desde un principio, produciendo código más limpio:

```
<?

if  (is_readable('/tmp/un_fichero_que_no_existe'))
    $f = @fopen('/tmp/un_fichero_que_no_existe','r');
else
    echo 'No he podido abrir el fichero';

?>
```

Sólo quedan los errores propiamente dichos, también llamados errores fatales. Un error fatal genera un mensaje de error por salida estándar, e inmediatamente después causa la detención del programa. Ejemplos de errores fatales incluyen el intentar crear una instancia de una clase inexistente, llamar a una función no definida. Los errores fatales no suelen suprimirse, ya que indican un fallo grave en la programación.

Además de los tres tipos de error básicos, PHP cuenta con distintos niveles de error, para las distintas capas en las que se pueden producir. Así, aparte de los errores generados por el propio código, tenemos errores y warnings generados por el motor de PHP, y errores y warnings generados por el parser/compilador. También se pueden provocar errores, que serán de otros tres tipos (user notice, user warning y user error).

En PHP5, se añadió otro nivel de error más, para ayudar al programador a seguir la sintaxis estricta de PHP5, y evitar el uso de funciones o directivas obsoletas. Un programa que en PHP4 es correcto, muy probablemente no dará ningún error en PHP5 siempre y cuando no se muestren los errores de sintaxis estricta (que actúan como notices). Un ejemplo de este tipo de error es el usar el obsoleto “var”y no “public”, “private”o “protected”para declarar propiedades de una clase.

Es muy recomendable activar la impresión de los errores de sintaxis estricta si queremos que nuestro código aproveche todas las capacidades de PHP5, y quede libre de funciones obsoletas.

12.1.2. Ahora te veo, ahora no

De todos los niveles de error posibles, podemos hacer que se muestren mensajes de error para aquellos niveles que deseemos. Esto se hace mediante la función error_reporting() y las constantes que indican los distintos niveles. Todo esto se encuentra perfectamente detallado en el manual de PHP, en la sección de funciones de tratamiento de errores.

```
<?

error_reporting(E_ALL);
// Muestra todos los errores, excepto los de sintaxis estricta

echo $a;

error_reporting(E_ALL | E_STRICT);
// Muestra todos los errores y también los de sintaxis estricta

class foo
{
}

$bar = new foo();

if (is_a($bar,'foo')) echo "sí";
// Es el objeto $bar una instancia de la clase foo ??

error_reporting(E_ALL & (~E_NOTICE) );
// Muestra todos los errores, excepto los notices.

?>
```

Además de poder cambiar en tiempo de ejecución los niveles de error que queremos ver, podemos también cambiar la directiva de configuración correspondiente (display_errors) en el fichero de configuración php.ini. Es conveniente recordar que si ejecutamos PHP como módulo en apache, será necesario reiniciarlo.

12.1.3. Me tropiezo yo mismo

En osasiones es conveniente causar errores en lugares determinados, ya sea para mantener información en un log o para ocultar un mensaje de error y mostrar otro distinto. Esto se consigue usando la función trigger_error(), que permite lanzar un error de tipo E_USER_NOTICE, E_USER_WARNING o E_USER_ERROR, de la siguiente manera:

```
<?

if  (is_readable('/tmp/un_fichero_que_no_existe'))
```

```
    $f = @fopen('/tmp/un_fichero_que_no_existe','r');
else
    trigger_error('No he podido abrir el fichero',E_USER_ERROR);

?>
```

12.2. ¿Quién es el general failure y qué hace en mi disco duro?

El usuario medio tiene un concepto bastante curioso de los errores de programación: o bien se siente aterrorizado ante un mensaje de error complejo, o bien no entiende lo que pasa (ni tampoco le importa mucho) y se va a otra parte.

12.2.1. Ojos que no ven...

Ojos que no ven, error que no aparenta serlo. Cuando se está desarrollando una aplicación web importante, a la vista del público, convinene mucho ocultar los posibles errores que puedan ocurrir. Pero no basta con ocultar unos pocos y hacer un tratamiento especial, sino ocultar todos. Esto se puede conseguir, como hemos visto, con error_reporting, pero sin embargo...

12.2.2. Ojos que sí ven...

Ojos que no ven, bug que toca las narices. Ocultar los errores "y ya está"puede ser perjudicial para el desarrollador, que en un momento determinado puede no saber qué está pasando en su aplicación, y necesita datos sobre los errores que pudiera haber. En este caso, lo recomendable es hacer nuestra propia función de tratamiento de errores, como por ejemplo:

```
<?php

// definimos una función de tratamiento de error
set_error_handler('oops');

// provocamos un error
funcion_que_no_existe();

// Y declaramos la función de tratamiento...
function oops($tipo, $mensaje, $fichero, $línea, $contexto) {
    echo "<h1>Error!</h1>
 Ha habido un error en nuestra aplicación. Por favor, escriba a nuestro
 <a href=mailto:webmaster@dondesea.org>webmaster</a> facilitándole la
 siguiente información, así como las circunstancias en las que se ha
 producido este error:
    <hr><pre>
    Código de error: $tipo
    Mensaje de error: $mensaje
    Script y línea: $fichero - $linea<br />";
    $variables = array_pop($contexto);
    echo "Último nivel de la pila de ejecución: ";
    print_r($variables);
    echo "</pre><hr>";
}

?>
```

Esto cambiará la apariencia con la que se muestran los mensajes de error. Pero si en vez de mostrarlo todo al usuario, lo enviamos todo a un programador...

```
<?php

// definimos una función de tratamiento de error
```

```
set_error_handler('oops');

// provocamos un error
funcion_que_no_existe();

// Y declaramos la función de tratamiento...
function oops($tipo, $mensaje, $fichero, $línea, $contexto) {
    echo "<h1>Error!</h1>
 Ha ocurrido un error en nuestra aplicación, pero nuestros ingenieros
 altamente cualificados ya están avisados para corregirlo lo más antes
 posible. Por favor, disculpe las molestias.";

    mail ('becario@dondesea.org','<Arregla esto!',"
    Código de error: $tipo
    Mensaje de error: $mensaje
    Script y línea: $fichero - $linea
    Contenidos de la pila:
    " . var_export($contexto,1));
}

?>
```

var_export() devolverá los contenidos completos de una variable de manera similar a como print_r los imprime, y mail() se encargará de enviar un correo electrónico a alguien, siempre y cuando el motor de php pueda ejecutar sendmail.

12.3. Excepciones

El modelo de tratamiento de errores de PHP5, como en tantos otros lenguajes, ha ido complicándose hasta llegar a un tratamiento moderno y completo con un sistema de excepciones. El uso de excepciones para provocar e interceptar errores, así como sistemas para su tratamiento, se está haciendo más popular cada día. Sin embargo, es una materia demasiado amplia como para tratarla en profundidad aquí. En el manual oficial de PHP5 está toda la información necesaria.

Capítulo 13

Miscelánea

PHP5 no se acaba aquí. Hay muchos otros aspectos que no hemos podido cubrir en este curso pero que son, sin duda, interesantes.

13.1. Código dinámico

PHP es un lenguaje interpretado y, como tal, puede ejecutar código dinámicamente. Esto quiere decir que puede generar código que luego ejecuta con eval(), es decir, que un programa puede programarse a sí mismo.

13.2. Aplicaciones no web

PHP también tiene aplicaciones fuera de la programación web, principalmente como lenguaje de shell scripting (ejecutado en consola directamente) o para generar aplicaciones gráficas con PHP-GKT.

13.3. Output Buffering

Mediante un mecamnismo denominado Output Buffering, o Control de Salida, podemos capturar y tratar los datos que, de otra manera, se impimirían directamente por salida estándar. Es decir, podemos capturar todo lo que se imprima con echo() y similares, y hacer un tratamiento de ello.

13.4. Pruebas automatizadas

Para proyectos grandes, poedmos usar el estándar TAP para comprobar que partes de la aplicación funcionan como es esperado.

13.5. Frameworks, sistemas de templates

Los interesados en crear proyectos grandes seguramente estarán interesados en usar sistemas que les permitan independizar el código de la visualización, mediante frameworks como PRADO o sistemas de templates como Smarty. Hay que tener en cuenta que PEAR (el repositorio de aplicaciones y extensiones de PHP) contiene muchos móculos ya probados para distintas funciones.

13.6. Imágenes, PDFs, Flash, RTF

PHP es capaz de generar dinámicamente imágenes, y otros tipos de documentos como PDF, SQF y RTF.

13.7. Iteradores sobre clases, PHP Standard Library

Mediante el uso de las funciones de la PHP Standard Library podemos ajustar aspectos de nuestras clases e implementar comportamientos únicos.

13.8. Cálculo numérico

PHP también cuenta con funciones de cálculo numérico de precisión arbitraria.

13.9. Servicios web

Mediante diversos módulos, podemos hacer que nuestra aplicación de servicios de manera estándar mediante SOAP y XML-RPC.

13.10. Encriptación

Podemos ocultar ciertos datos usando algoritmos como DES, Blowfish o SHA para cifrarlos antes de volcarlos a fichero o a BDD.

13.11. Extensiones al motor de PHP

Se pueden añadir funcionalidades al motor de PHP, escritas en C. A estas extensiones después se podrá acceder llamando a funciones dentro de nuestro código PHP.

13.12. Optimizadores y máquinas virtuales

PHP sufre de los inconvenientes de un lenguaje interpretado, pero para solventar esto ya existen diversos optimizadores de código (como el que comercializa Zend), y ya hay un proyecto para crear una máquina virtual y un lenguaje de código intermedio al estilo de Java (`http://www.parrotcode.org`).

13.13. Hasta el infinito y más allá

En definitiva, PHP es un lenguaje sencillo de aprender pero extremadamente potente, extensible y apto para muchas aplicaciones. Esperamos que todo lo visto en este curso haya sido de interés, y que PHP os resuelva muchos problemas. Gracias.