

```
require_once("settings.php");
```

```
# Inicia la sesión del usuario
```

```
function login() {  
    $user_valido = validar_user_y_pass();  
    if($user_valido) {  
        $_SESSION['date_login'] = time();  
    }  
    goto_page(PAGINA_HOME);  
}
```

```
# Validar usuario y contraseña
```

```
function validar_user_y_pass() {  
    $user = $_POST['user'];  
    $password = $_POST['password'];  
    if ($user != '' && $password != '') {  
        $hash = hash_password($password);  
        if ($hash == $password_hash($user)) {  
            return true;  
        }  
    }  
    return false;  
}
```

```
# Traer el POST data y retornar el hash MD5
```

```
function get_post_data() {
```

Programador PHP

```
    return $hash;  
}
```

Eugenia Bahit

```
# Destruir sesión
```

```
function logout() {  
    unset($_SESSION);  
    $datos_cookie = session_get_cookie_params();  
    setcookie(session_name(), NULL, time()-999999,  
        $datos_cookie["path"],  
        $datos_cookie["domain"],  
        $datos_cookie["secure"],  
        $datos_cookie["httponly"]);  
    goto_page(PAGINA_LOGIN);  
}
```



Programador PHP de [Eugenia Bahit](#) se distribuye bajo una Licencia [Creative Commons Atribución-NoComercial-SinDerivadas 3.0 Unported](#).

Comparte el conocimiento

Eres libre de:

- **Copiar, distribuir y compartir** este libro

Bajo las siguientes condiciones:

- Reconocer y respetar la autoría de la obra
- No hacer uso comercial de ella
- No alterar el contenido



Índice General

Introducción a las tecnologías GLAMP.....19

GLAMP y LAMP.....19

Diferencia entre GNU/Linux y Linux.....19

Free Software Foundation y el Proyecto GNU.....20

Información complementaria.....20

 Sitios Web de Referencia.....20

 Bibliografía complementaria.....20

Preparación básica del entorno de desarrollo.....21

 Instalación de Ubuntu GNU/Linux en Windows.....21

 Instalación de Ubuntu GNU/Linux como único Sistema Operativo
 22

 Instalación de Apache Server.....22

 Otras opciones de Apache:.....25

 It Works!.....26

 Instalación de MySQL.....27

 Instalación de PHP.....27

 Verificando la versión de PHP.....27

Configurando el directorio de tu Web Local.....28

 Crear el directorio para tu Web local.....28

 Modificando el directorio raíz de tu Web local.....29

Conociendo PHP.....31

Etiquetas de apertura y cierre31

 Conocer el valor actual de short_open_tag.....33

Instrucciones y estructuras.....34

Identación en PHP.....34

Impresión en pantalla.....35

Variables y tipos de datos básicos.....37

Null, var_dump() e isset().....	40
Malas prácticas con variables, que afectan el uso de memoria	43
Operadores aritméticos.....	44
Haciendo cálculos con el IVA.....	44
settype ¿un bug no resuelto o feature objetable?.....	45
HTML y PHP en un mismo archivo.....	47
Una mala práctica: colocar código HTML embebido dentro de variables de PHP.....	47
Una buena práctica para evitar lo anterior.....	48
Comentando y documentando el código fuente.....	49
Inclusión de archivos en PHP.....	51
Diferencia entre inclusión de archivos remotos y locales.....	52
Diferencia entre include y require.....	53
Include y require "_once".....	53
Estructuras de Control - Parte I (condicionales).....	54
Definición.....	54
Condicionales if, else y else if.....	54
Operadores lógicos y de comparación.....	55
Operadores de comparación.....	55
Diferencia entre igualdad e idéntico en la comparación.....	56
Operadores lógicos.....	56
Creando condicionales.....	57
Estructuras de control con Switch.....	59
¿Cuándo utilizar if y cuando switch?.....	61
Tipos de Datos complejos: Matrices simples y multidimensionales.....	62
Matrices en PHP.....	62
Sintaxis básica.....	62
Imprimir en pantalla con print_r.....	63

Acceso a los ítems de un array.....	63
Modificando elementos.....	64
Agregar elementos.....	64
Estructuras de Control - Parte II (bucles I).....	66
Recorriendo matrices dinámicamente con foreach.....	66
Sintaxis básica del constructor foreach.....	66
Un ejemplo de iteración compleja con foreach.....	67
Modificando matrices dinámicamente con foreach.....	68
Estructuras de Control - Parte III (bucles II).....	71
While, un bucle simple.....	71
Un ejemplo sencillo.....	71
Un ejemplo práctico.....	72
Do while, tan simple como while pero con una ventaja.....	73
Un ejemplo simple.....	73
Bucles for, los más complejos de PHP.....	75
Sintaxis:.....	75
Curiosidades sintácticas de la bipolaridad no diagnosticada de PHP	78
Goto, si lo usas... es tu elección!.....	79
Un ejemplo no-práctico para entender goto.....	79
Funciones definidas por el usuario.....	82
Definición.....	82
Declarando Funciones.....	82
Sintaxis básica.....	82
Sobre el nombre de las funciones.....	82
Sobre los parámetros.....	82
Llamando a una función.....	83
Sobre la finalidad de las funciones.....	84

Paso de variables por referencia en funciones.....	84
Modificando variables globales mediante el uso de global.....	85
Llamadas de retorno.....	86
Pasar argumentos en una llamada de retorno.....	87
Argumentos no conocidos.....	88
Conocer la cantidad de argumentos.....	88
Obtener una lista completa de todos los argumentos.....	89
Obtener un argumento específico.....	89
Saber si una función puede ser llamada (callable).....	89
Material de lectura adicional.....	90
Diferentes formas de recoger argumentos para hacer una llamada de retorno.....	91
Forma 1: recibir argumentos en un array.....	91
Forma 2: recibir argumentos 1 a 1.....	92
Llamadas recursivas.....	93
Helpers.....	95
Un helper que retorna la fecha actual.....	95
Un helper que modifica una variable global, haciendo una llamada de retorno.....	96
Taller de Funciones.....	97
Trabajando con el Sistema de Archivos.....	98
Recorrido rápido por las principales funciones.....	98
Apertura de archivos.....	98
Modos de apertura.....	99
Ruta hacia el archivo.....	100
Utilizar o no include_path.....	100
Lectura de Archivos.....	100
Escribir en un archivo.....	101
Moviendo el puntero dentro del archivo.....	102

Un contador de visitas sencillo.....	102
¡Cuidado con los permisos!.....	103
Trabajando con directorios.....	103
Creando el gestor.....	103
Explorando el contenido de un directorio.....	104
Filtrando el tipo de elemento.....	106
Otras funciones que necesitarás con frecuencia.....	108
Comprobar la existencia de un archivo o directorio.....	108
Comprobar si un archivo o directorio es legible.....	109
Comprobar si un archivo o directorio puede escribirse.....	110
Más funciones sobre el sistema de archivos.....	110
Procesamiento de texto y manipulación de strings....	110
Ampliando la definición de variables de tipo string.....	111
Escapando caracteres.....	112
Caracteres de escape.....	112
Funciones para manipulación de strings.....	114
Funciones de escape.....	114
Funciones de conversión.....	115
Evitando ejecución de código no deseado.....	116
Funciones de formato.....	116
Funciones de manipulación.....	120
Manipulando subcadenas en cadenas.....	123
Funciones de encriptación.....	125
Resumen de las principales funciones de string.....	127
Taller de Archivos y Procesamiento de Formularios.....	129
Constantes, variables variables y variables superglobales.....	130
Constantes.....	130

Definición clásica de constantes en PHP.....	130
Definición de constantes en PHP 5.3.....	131
Finalidad de las constantes.....	132
Variables variables.....	133
Variables superglobales.....	134
Envío de correo electrónico con PHP.....	136
La función mail() y su sintaxis.....	136
El parámetro “destinatario”: formatos admitidos.....	136
Cabeceras adicionales como parámetro extra.....	137
Comprobando que el e-mail pudo enviarse.....	137
Enviando mensajes en formato HTML.....	139
Funciones para el manejo de Fecha y Hora.....	140
Funciones simples de fecha y hora.....	140
Obtener la fecha y hora actual en un array asociativo.....	140
Obtener fecha y hora actual con formato en una cadena de texto.....	141
Validar una fecha.....	144
Cálculo de fecha / hora sencillo.....	144
Ejemplos prácticos de cálculos basados en fechas.....	146
¿Cuánto tiempo ha pasado?.....	146
¿Qué edad tiene...?.....	147
¿En qué fecha nació...?.....	147
Funciones matemáticas.....	149
Obtener un número elevado a la potencia.....	149
Obtener el número más alto y el número más bajo.....	149
Redondear un número con N cantidad de decimales.....	150
Redondear un número hacia abajo.....	150
Redondear un número hacia arriba.....	150

Obtener un número entero aleatorio.....	151
Funciones para el manejo de matrices.....	151
Dividiendo y uniendo arrays.....	151
Dividir un array en matrices más pequeñas.....	151
Obtener la porción específica de un array.....	153
Combinar dos arrays, utilizando uno para las claves y otro para los valores.....	153
Combinar dos o más arrays.....	154
Combinar dos o más arrays multidimensionales de manera recursiva.....	154
Ordenando Arrays por sus valores.....	155
Ordenar un array de menor a mayor.....	155
Ordenar un array de mayor a menor.....	155
Ordenar un array de menor a mayor manteniendo la relación con los índices.....	156
Ordenar un array de mayor a menor manteniendo la relación con los índices.....	156
Ordenando Arrays por su clave.....	156
Ordenar un array de menor a mayor por su clave.....	156
Ordenar un array de mayor a menor por su clave.....	157
Comparando funciones de ordenamiento de arrays.....	157
Agregar y Eliminar elementos de un array.....	158
Agregar elementos al final del array.....	158
Agregar elementos al comienzo del array.....	158
Eliminar el último elemento de un array.....	159
Eliminar el primer elemento de un array.....	159
Eliminar valores duplicados en un array.....	159
Búsquedas y filtros.....	160
Contar la cantidad de veces que los elementos aparecen en un array.....	160

Contar la cantidad de elementos de un array.....	160
Obtener la suma matemática de los valores de un array.....	161
Obtener las diferencias entre dos o más arrays.....	161
Filtrar datos de un array, utilizando una función de retorno...	162
Verificar si un array contiene una clave determinada.....	163
Obtener todas las claves de un array o todos los valores.....	163
Verificar si un array contiene una valor determinada.....	164
Buscar un valor detrminado en un array y obtener su clave correspondiente.....	164

Cookies y Sesiones de usuario.....165

¿Qué es una cookie?.....	165
Las cookies no son eternas.....	166
¿Qué son las sesiones de usuario?.....	166
Usos e importancia.....	167
Lo básico.....	167
Creación, lectura, modificación y eliminación de cookies.....	167
Crear una cookie.....	167
Leer una cookie.....	169
Modificar una cookie.....	169
Eliminar una cookie.....	169
Un ejemplo práctico con Cookies.....	170
Trabajando con Sesiones.....	173
Primeros pasos con sesiones.....	173
Crear una nueva sesión.....	174
Leer una sesión.....	175
Modificar la sesión.....	175
Eliminar una variable de sesión.....	175
Un caso práctico de uso de sesiones.....	176
Funciones necesarias.....	178

Funciones de acceso al sistema.....	178
Funciones para destruir la sesión del usuario.....	179
Funciones para verificación y validación de sesiones.....	180
La función que redirige a los usuarios.....	182
Pasos finales.....	182
Tratamiento y control de errores.....	184
Tipos de errores.....	184
Configurando errores en tiempo de ejecución.....	186
Un ejemplo sencillo pero altamente productivo.....	186
Utilizando el símbolo @ para silenciar errores.....	187
Trabajando con Bases de Datos MySQL.....	189
Acerca de MySQL.....	190
Instalación y configuración de MySQL.....	190
Iniciar, reiniciar y detener el servidor MySQL.....	191
Administración de MySQL.....	193
Conectarse y desconectarse al servidor.....	193
Comandos para administrar MySQL desde el shell interactivo	193
Sobre el lenguaje SQL.....	195
Tipos de datos más comunes (recomendados).....	195
Sintáxis básica de las sentencias SQL.....	196
Crear tablas en una base de datos.....	196
Insertar datos en una tabla.....	198
Seleccionar registros.....	198
Modificar registros.....	199
Eliminar registros.....	200
Consultas avanzadas.....	201
La cláusula WHERE.....	201

Ordenando consultas: la cláusula ORDER BY.....	203
Alias de tablas y campos.....	203
Funciones del lenguaje SQL de MySQL.....	204
Contar la cantidad de registros: COUNT().....	204
Sumar totales: SUM().....	204
Concatenar cadenas: CONCAT().....	204
Convertir a minúsculas y mayúsculas: LCASE() y UCASE().	205
Reemplazar datos: REPLACE().....	205
Obtener los primeros o últimos caracteres: LEFT() y RIGHT()	205
Redondear números: ROUND().....	205
Obtener solo la fecha de un campo DATETIME o TIMESTAMP: DATE().....	205
Obtener una fecha formateada: DATE_FORMAT().....	205
Obtener el registro con el valor máximo y mínimo: MAX() y MIN().....	206
Optimización de bases de Datos.....	206
Todos los registros deben tener un ID único.....	206
Crear índices en las tablas.....	207
Indica cuáles campos no pueden ser nulos.....	207
Utiliza el motor InnoDB.....	208
Obtener mayor información.....	208
Trabajando con MySQL desde PHP.....	209
MySQL desde PHP con el conector mysql.....	210
Conectarse a la base de datos.....	210
Seleccionar una base de datos.....	211
Ejecutar una consulta simple.....	211
Ejecutar una consulta de selección múltiple y capturar sus resultados.....	211
Capturamos el array con los resultados.....	211

Liberar los resultados.....	212
Cerrar la conexión.....	212
Algunos ejemplos concretos.....	213
Consulta de selección.....	213
Insertar varios registros en un solo paso.....	213
MySQL desde PHP con el conector mysqli.....	215
Abrir una conexión mediante mysqli.....	216
Preparar la consulta.....	216
Ejecutar la consulta.....	217
Cerrar la consulta.....	217
Cerrar la conexión.....	217
Ejemplo de inserción completo.....	218
Capturar resultados de una consulta de selección.....	218
Ejemplo completo de consultas de selección.....	219

Introducción al Paradigma de la programación orientada a objetos.....221

Pensar en objetos.....	222
Y ¿qué es un objeto?.....	222
Ahora ¿qué me dices si describimos las cualidades de un objeto?.....	223
Pero algunos objetos, también se componen de otros objetos... ..	224
Y también hay objetos que comparten características con otros objetos.....	227
Los objetos, también tienen la capacidad de “hacer cosas” ...	230
Objetos y más objetos: la parte difícil.....	231
Al pan, pan. Y al vino, vino. Las cosas por su nombre.....	232
Programación Orientada a Objetos.....	233
Elementos y Características de la POO.....	233

Clases.....	234
Propiedades.....	234
Métodos.....	235
Objeto.....	236
Herencia: característica principal de la POO.....	236
Accediendo a los métodos y propiedades de un objeto.....	237
Acceder a las propiedades de un objeto, dentro de la clase	237
Composición.....	238
Ejemplo práctico de Herencia vs. Composición.....	240
Programando con Historias de Usuario.....	242
Criterios de Aceptación	243
Dividiendo Historias de Usuario en Tareas.....	244
Introducción a la Programación eXtrema.....	247
TDD - Test-Driven Development.....	248
¿Qué es el desarrollo -o programación- guiado por pruebas?.....	248
Test Unitarios.....	251
Características de los Test Unitarios.....	252
Anatomía.....	253
Algoritmo para escribir pruebas unitarias.....	257
PRIMER PASO: Escribir el Test y hacer que falle.....	257
SEGUNDO PASO: Escribir la mínima cantidad de código para que el test pase.....	259
TERCER PASO: Escribir un nuevo test y hacer que falle.....	260
CUARTO PASO: Escribir el algoritmo necesario para hacer pasar el test.....	261
Unit Testing con PHPUnit.....	265
Métodos Assert de PHPUnit.....	265

Ejercicio.....	268
Unificación del código en Repositorios.....	269
Sobre los Sistemas de Control de Versiones.....	269
Integración continua con Bazaar.....	271
Instalación de Bazaar.....	271
Bazaar por línea de comandos.....	272
Presentarse ante Bazaar.....	273
Iniciar un nuevo proyecto.....	273
Clonar el repositorio central: crear los repositorios locales.....	273
Nociones básicas para integrar código de forma continua.....	275
Guardando el path del repo central.....	276
Integración continua avanzada con Bazaar.....	277
Resumen de comandos de uso frecuente.....	278
Resumen para uso diario de Bazaar.....	280
Refactoring.....	282
El problema.....	282
La solución.....	283
Cuándo y cómo tomar la decisión de refactorizar.....	284
Una solución a cada problema.....	285
Variables de uso temporal mal implementadas.....	285
Métodos que reciben parámetros.....	288
Expresiones extensas.....	289
Métodos extensos.....	289
Código duplicado en una misma clase.....	291
Código duplicado en varias clases con la misma herencia.....	292
Código duplicado en varias clases sin la misma herencia.....	293
Introducción a la Arquitectura de Software.....	295
¿Qué es la arquitectura de software?	295

Atributos de calidad	295
Niveles de abstracción.....	297
Estilo Arquitectónico.....	298
Patrón Arquitectónico.....	299
Patrón de Diseño.....	300
Introducción al Patrón Arquitectónico MVC.....	301
Entendiendo el funcionamiento de MVC.....	301
Modelos en MVC.....	307
Objetos puros: características de un modelo.....	307
Creando modelos bajo MVC en PHP.....	308
Herencia y Composición.....	308
Acceso a bases de datos.....	310
Código fuente de una capa de abstracción a nivel del core	311
Object Relational Mapping (ORM).....	312
Ventajas y desventajas del utilizar ORMs.....	313
ORM Frameworks para PHP.....	314
Bibliografía recomendada.....	314
Las vistas.....	316
¿Por dónde empezar a desarrollar las vistas?.....	316
Desarrollando la GUI.....	317
Componentes de la GUI.....	317
Arquitectura.....	318
Preparando la GUI para interactuar con la lógica.....	319
Comodines ¿Qué son y cómo implementarlos?.....	320
Diseñando la lógica de negocios.....	323
Lógica principal.....	324
Lógica a nivel de módulo.....	334

Características mínimas que debe tener la lógica...	334
Creando la lógica de la vista para un modelo.....	337
El controlador: el alma de MVC.....	340
Front Controller: controlando la aplicación a nivel del core.....	342
Configurando URLs amigables para nuestra aplicación.....	343
Configuración de Apache.....	344
Modificar el VirtualHost.....	345
Creando el archivo .htaccess.....	346
Creando un Application Handler.....	346
La clase AppHandler.....	347
Conclusión.....	350
La clase FrontController.....	350
Conclusión.....	353
Creando controladores para nuestros modelos.....	354
Notas adicionales sobre el ejemplo.....	359
Completando la aplicación.....	360
Inicializador del núcleo.....	363
Los patrones de diseño Front Controller y Application Handler en MVC.....	366
Web Services: creación de una API REST.....	369
Conceptos básicos.....	369
Web Services.....	369
API.....	369
REST.....	369
Entendiendo las Arquitecturas REST.....	370
Características de una API-REST.....	370
Funcionamiento de la API REST de nuestra MVC App.....	372

Objetivo del Web Service.....	372
Formato de la URI.....	375
Diseño de la API.....	376

Introducción a las tecnologías GLAMP

GLAMP son las siglas de cuatro tecnologías libres, que conforman la base de las aplicaciones Web basadas en:

- Sistema Operativo: **GNU/Linux**
- Servidor Web: **Apache**
- Servidor de bases de datos: **MySQL**
- Lenguaje de programación híbrido (multiparadigma) y de alto nivel: **PHP**

GLAMP y LAMP

La mayoría de las veces, encontraremos bibliografía que al momento de referirse a las tecnologías GLAMP, suprimen la “G” del comienzo, cometiendo el grave error de llamarlas simplemente LAMP. De la misma forma, en una gran cantidad de casos, la documentación se refiere al Sistema Operativo GNU/Linux, como “Linux”, suprimiendo las siglas “GNU”.

Pero ¿Qué tiene aquello de errado? La respuesta a esta pregunta, está en la gran diferencia entre GNU/Linux y Linux.

Diferencia entre GNU/Linux y Linux

Linux, es un kernel, es decir, el núcleo de un Sistema Operativo, mientras que **GNU/Linux, el Sistema Operativo que utiliza el Kernel Linux** como núcleo.

El Kernel Linux, parte fundamental del Sistema Operativo, fue desarrollado por **Linus Torvals**, utilizando como modelo a UNIX. Una de las diferencias fundamentales entre los núcleos Linux y UNIX, es que el primero, es Software Libre, mientras que el segundo no lo es.

Por otra parte, mientras existe un único Kernel Linux (con versiones diferentes), existen decenas y hasta cientos de **distribuciones GNU/Linux**, es decir, diferentes Sistemas Operativos basados en el Kernel Linux, entre las cuales se destacan: **Debian, Ubuntu, Kubuntu, Fedora, Gentoo, Slackware, CentOS, ArchLinux, Asturix**, entre otros cientos.

Free Software Foundation y el Proyecto GNU

La **Free Software Foundation**, organización sin fines de lucro, fundada por Richard Stallman, principal precursor del Software Libre, es el organismo que creó, difunde y promueve, el Sistema Operativo GNU/Linux, a través del **Proyecto GNU**.

Información complementaria

Sitios Web de Referencia

Sitio Web de la **Free Software Foundation**: www.fsf.org

Sitio Web del **Proyecto GNU**: www.gnu.org

Sitio Web del **Kernel Linux**: <http://www.kernel.org/>

Sitio Web de la **Linux Foundation**:
<http://www.linuxfoundation.org/>

Bibliografía complementaria

[Introduccion al software libre.pdf](#) (Universitat Obierta de

Catalunya)

[Sistema operativo gnu linux basico.pdf](#) (Universitat Obierta de Catalunya)

Preparación básica del entorno de desarrollo

En este curso, nos enfocaremos en tecnologías GLAMP, a partir de la distribución **Ubuntu 10.04 LTS (Lucid) -o superior-** de GNU/Linux, basada en Debian.

En caso de YA contar con otra distribución, versión de Ubuntu o Debian, puedes saltar estos párrafos e ir directamente a la instalación de Apache.

Instalación de Ubuntu GNU/Linux en Windows

Si eres usuario de Windows y deseas conservar tu Sistema Operativo actual, **puedes descargar Ubuntu Windows Installer** desde el sitio Web oficial de Canonical (empresa que desarrolla y mantiene Ubuntu) en la siguiente URL:

<http://www.ubuntu.com/download/ubuntu/windows-installer>

Ubuntu Windows Installer se instalará desde el propio MS Windows© como si fuese un Software más, permitiéndote iniciar tu ordenador con Ubuntu o MS Windows© según elijas.

Para instalar Ubuntu Windows Installer, **sigue las instrucciones de los pasos 2 y 3 de la URL de descarga**, las cuales podrás visualizar pulsando el botón “Show me how” de cada uno de los pasos.

Instalación de Ubuntu GNU/Linux como único Sistema Operativo

Para instalar Ubuntu como único Sistema Operativo, sigue los siguientes pasos:

1. ingresa en <http://www.ubuntu.com/download/ubuntu/download>
2. En el paso 1, selecciona la versión de Ubuntu que deseas descargar. Para procesadores de un solo núcleo, selecciona la versión 10.04 LTS. Para procesadores más modernos, puedes seleccionar la última versión (versión que aparece seleccionada por defecto en el desplegable de versiones). Si tienes dudas sobre si elegir la versión para 32 o 64 bits, elige la de 32-bits. Pulsa el botón “Start download” y aguarda a que se descargue el archivo.
3. Una vez descargado el archivo, podrás quemarlo en un CD/DVD o un Pendrive USB. En el paso 2 de la URL de descarga, selecciona CD o USB stick según tus preferencias y el Sistema Operativo desde el cual harás la copia (Windows o Mac). Pulsa el botón “show me how” y sigue las instrucciones de quemado.
4. A continuación, salta al paso 4 del sitio de descarga (el 3 es solo para probar Ubuntu sin instalarlo); pulsa el botón “show me how” y sigue las instrucciones para instalar Ubuntu en tu ordenador.

Instalación de Apache Server

Antes de instalar Apache en tu distribución GNU/Linux, crearemos un **Lanzador de la terminal** (llamado “acceso directo” en Windows), para ya tenerlo “a mano”. Para ello, sigue los siguientes pasos:

1. En el panel superior (donde figuran los menús

“Aplicaciones, Lugares y Sistema”), haz clic derecho con el ratón luego del menú “Sistema”, y selecciona la opción **“Añadir al Panel”**.

2. A continuación, haz doble clic sobre la primera opción **Lanzador de Aplicación Personalizado**.
3. En la ventana de creación de lanzador, ingresa los datos como se muestra a continuación:
 - **Nombre:** Terminal
 - **Comando:** gnome-terminal
 - **Comentario:** Abrir terminal en modo gráfico
4. Pulsa el botón **“Aceptar”**. Verás ahora, el símbolo de la terminal en tu panel superior.

Una vez creado el lanzador, vamos a continuar instalando Apache. Para ello, **abre una terminal**, pulsando una vez, sobre el ícono del lanzador, que acabas de crear.

Una vez en la terminal, lo primero que haremos, será asegurarnos de tener actualizado el sistema operativo y de esta forma, securizarlo. Para ello, escribe el siguiente comando:

```
sudo apt-get update
```

Deberás **ingresar tu contraseña**. Mientras la escribas, no se mostrará ningún carácter en la pantalla (ni siquiera asteriscos).

SOBRE LOS COMANDOS

sudo: te convierte en super usuario. Único usuario que tiene permisos para instalar paquetes en tu sistema operativo.

apt-get: es la utilidad para manejar paquetes en distribuciones GNU/Linux basadas en Debian. Alternativamente, puede utilizar el comando **aptitude** en

vez de apt-get.

update: opción de apt-get que sincroniza los archivos del índice de paquetes con los repositorios oficiales (dicho de otra forma, obtiene un índice de actualizaciones)

Una vez finalizada la sincronización del índice de actualizaciones, escribe:

```
sudo apt-get upgrade
```

Nuevamente deberás ingresar tu contraseña. Posiblemente, deban instalarse actualizaciones. Entonces, **te preguntará si deseas continuar**. Deberás **pulsar la tecla y** (de “yes”) y luego la **tecla enter** y esperar que finalicen las actualizaciones.

Recuerda: siempre, antes de instalar cualquier paquete, debes ejecutar previamente, los comandos **sudo apt-get update** y luego **sudo apt-get upgrade**.

Una vez actualizado el sistema operativo, procederemos a instalar Apache. Para ello, escribe el siguiente comando:

```
sudo apt-get install apache2
```

SOBRE LOS COMANDOS

install es la opción de apt-get que indica que se instalará uno o más paquetes

apache2 es el nombre del paquete que se instalará

Ya tenemos el Sistema Operativo y el servidor Web instalado. Para asegurarnos de que Apache esté funcionando, vamos a escribir el siguiente comando en la terminal:

```
sudo /etc/init.d/apache2 start
```

TIP:

Al escribir comandos, nombres de archivos y/o directorios en la terminal, **pulsando la tecla de tabulación, se autocompletan.**

Cuando al pulsar la tecla de tabulación, **un pitido es emitido**, puede significar una de dos cosas: **a)** que el comando, nombre de archivo o directorio no se ha localizado; **b)** la más frecuente, que existen varias opciones posibles para autocompletar.

Por eso, **cundo un pitido sea emitido, pulsa la tecla de tabulación dos veces consecutivas.** Si existen varias opciones, te las mostrará en pantalla.

Otras opciones de Apache:

Iniciar Apache:

```
sudo /etc/init.d/apache2 start
```

Apagar Apache:

```
sudo /etc/init.d/apache2 stop
```

Reiniciar Apache:

```
sudo /etc/init.d/apache2 restart
```

se debe utilizar siempre, tras realizar alguna modificación a Apache.

Los **cambios de configuración de Apache**, se realizan modificando el archivo `apache2.conf` que se encuentra en el directorio `/etc/apache2/`.

Este archivo, solo puede modificarse, accediendo a él, como super usuario:

```
sudo gedit /etc/apache2/apache2.conf
```

el comando `gedit`, abrirá el archivo con la aplicación Gedit: un editor de textos.

Recargar Apache:

```
sudo /etc/init.d/apache2 reload
```

se utiliza generalmente, cuando un nuevo sitio Web es configurado.

It Works!

Cuando Apache ha sido instalado e inicializado, podrás ver la página de bienvenida, ingresando la URL <http://localhost> o <http://127.0.0.1> en tu navegador de Internet.

Para **abrir el navegador de Internet**, ve a **Aplicaciones > Internet > Navegador Web Firefox**.

También puedes abrir **Firefox desde la terminal**, escribiendo `firefox`.

Es posible también, **abrir una URL en Firefox desde la terminal**, escribiendo:

```
firefox http://www.google.com
```

(o la URL a la cual desees acceder)

Instalación de MySQL

(ver sección: PHP y MySQL más adelante)

Instalación de PHP

Instalar PHP en Ubuntu es tan simple que solo requiere un comando. Aprovecharemos esta simplicidad, para “matar dos pájaros de un tiro”, e instalar, además de PHP, PHP-CLI: un intérprete de línea de comando para PHP, que nos permitirá probar código escrito en este lenguaje, utilizando un shell interactivo desde la terminal.

Para **instalar PHP y PHP-CLI**, escribe:

```
sudo apt-get install php5 php5-cli
```

Verificando la versión de PHP

En tu terminal, escribe:

```
php -v
```

Obtendrás un resultado similar al siguiente:

```
eugenia@cocochito:~$ php -v
PHP 5.3.2-1ubuntu4.14 with Suhosin-Patch (cli) (built: Feb 11 2012 06:50:46)
Copyright (c) 1997-2009 The PHP Group
Zend Engine v2.3.0, Copyright (c) 1998-2010 Zend Technologies
```

Configurando el directorio de tu Web Local

Por defecto, Apache espera que los archivos de tu sitio Web local, se alojen en el directorio `/var/www/`, pero trabajar en este directorio puede ser muy incómodo, ya que solo podrás acceder a él, con permisos de super usuario.

Para evitar este escollo, tenemos dos formas diferentes de solucionarlo:

- **Opción 1:** Crear un enlace simbólico (llamado “acceso directo” en Windows), en la carpeta `/var/www/`, que redirija hacia la carpeta de tu home, donde quieras hospedar los archivos de tu Web local.
- **Opción 2:** Modificar el directorio raíz de tu Web local y establecerlo apuntando hacia el directorio de tu home donde quieras hospedar los archivos de tu Web local. Esta segunda opción, será la forma que elegiremos en este curso. Sin perjuicio de ello, aprenderemos sobre como llevar adelante la opción 1.

Crear el directorio para tu Web local

Abre una terminal y escribe el comando `cd ~` que te llevará directamente a la home de tu usuario y a continuación escribe el comando `pwd` que te indica en que directorio te encuentras actualmente. Recuerda que el comando `cd` sirve para moverte desde la terminal, por todos los directorios.

Verás algo como esto:

```
eugenia@cocochito:/etc/apache2$ cd ~
```

```
eugenia@cocochito:~$ pwd
/home/eugenia
```

Lo anterior, indica que el directorio de la home de mi usuario, es /home/eugenia. El tuyo será /home/nombre_de_tu_usuario.

Una vez allí, crearemos un directorio llamado curso-php, a fin de almacenar allí, todos los archivos que utilicemos en el curso para luego, convertirlo además, en la home de tu Web local.

Para crear el directorio, escribe lo siguiente:

```
mkdir curso-php
```

El comando **mkdir** es el utilizado para crear directorios.

Modificando el directorio raíz de tu Web local

En la terminal, navega con el comando **cd** hasta el directorio de Apache donde se almacenan los archivos de configuración de los sitios Web hospedados:

```
cd /etc/apache2/sites-available/
```

A continuación, **lista los archivos de esa carpeta** con el comando **ls** (ele ese) seguido de la opción **-l** (ele) que te permitirá listarlos uno debajo del otro (sin la opción **-l**, se mostrarían uno al lado del otro). Verás algo como lo que sigue:

```
eugenia@cocochito:~$ cd /etc/apache2/sites-available/
eugenia@cocochito:/etc/apache2/sites-available$ ls -l
total 4
-rw-r--r-- 1 root root 960 2011-10-21 23:13 default
```

El archivo **default** es el que modificaremos. Para **modificar el**

archivo default, necesitamos hacerlo como super usuario y lo abriremos con Gedit, escribiendo:

```
sudo gedit default
```

Una vez allí, localiza la línea que establece cuál será la raíz de tu Web local:

```
DocumentRoot /var/www
```

Y modifícala por:

```
DocumentRoot /home/tu-usuario/curso-php
```

A continuación, el bloque que establece directivas de configuración especiales para el directorio raíz de tu Web local. Dicho bloque, es el que comienza por:

```
<Directory /var/www/>
```

Reemplaza allí, `/var/www/` por `/home/tu-usuario/curso-php/`

```
<Directory /home/tu-usuario/curso-php/>
```

Guarda los cambios y cierra Gedit. Finalmente, tendremos que reiniciar Apache para que los cambios se vean reflejados. Para ello, en la terminal, escribe:

```
sudo /etc/init.d/apache2 restart
```

Como bien comentamos antes, hubiese sido posible, crear un enlace simbólico en `/var/www/` que apuntara a `/home/tu-usuario/curso-php/`.

De haber optado por esta alternativa, nos hubiésemos valido

del comando **ln** (ele ene) con la opción **-s**, destinado a crear enlaces simbólicos (o symlinks).

Como la escritura en el directorio `/var/www/` está restringida a usuarios con permiso de root (super usuarios), deberíamos haber ejecutado dicho comando anteponiendo el comando `sudo`.

La sintaxis para crear enlaces simbólicos dentro de una carpeta, es:

```
ln -s destino nombre_del_enlace_simbolico
```

Donde destino será la ruta completa del directorio (o archivo) al que queremos apuntar, y nombre_del_enlace_simbolico el “alias” para ese symlink.

Para crear un enlace simbólico, llamado `miweb`, dentro del directorio `/var/www/` y que apunte a `/home/eugenia/curso-php/` deberíamos haber hecho lo siguiente:

```
cd /var/www/  
sudo ln -s /home/eugenia/curso-php/ miweb
```

Entonces cuando accediéramos a `/var/www/miweb/` hubiésemos estado viendo los archivos de `/home/eugenia/curso-php/`.

Conociendo PHP

Etiquetas de apertura y cierre

Como se explicó anteriormente, existen dos posibilidades para

definir que un archivo debe ser interpretado en PHP:. Veremos aquí, las ventajas y desventajas de cada uno de ellos.

Opción #1 (recomendada):

```
<?php
// aquí irá todo el contenido en lenguaje PHP
?>
```

Esta opción, se sugiere como alternativa recomendada, puesto que independientemente del valor establecido en `short_open_tag` en el `php.ini`, funcionará por defecto y sin necesidad de modificar el archivo `php.ini`, en cualquier servidor.

Por otro lado, la utilización de esta alternativa, trae aparejadas las siguientes ventajas:

1. Permite la utilización de XML en el servidor. El lenguaje XML utiliza como etiquetas de apertura y cierre, `<?` y `?>`. Alternativamente, permite también `<?xml` y `?>`. Por lo tanto, utilizando `<?php` se permite ejecutar código XML como tal.
2. Evita tener que embeber¹ código XML dentro de PHP
3. Es una forma de definir un lenguaje estandar de PHP.

AVISO:

Para poder utilizar XML (además de PHP), se recomienda establecer el valor de `short_open_tag` en **Off**, en el archivo `php.ini`, puesto que el valor por defecto se encuentra establecido en **On**.

¹ Embeber código se refiere a hacer un `print` (o `echo`) con PHP, para escribir utilizando otro lenguaje. Es una de las prácticas de programación más desaconsejadas, ya que dificulta la lectura de código haciendo difícil la escalabilidad y mantenimiento de aplicaciones.

Opción #2:

```
<?
// aquí irá todo el contenido en lenguaje PHP
?>
```

Esta alternativa, representa una forma abreviada de las etiquetas anteriores. Utilizarla, requiere de configurar el archivo `php.ini`, estableciendo el valor de `short_open_tag` a `On`.

```
; This directive determines whether or not PHP will recognize code between
; <? and ?> tags as PHP source which should be processed as such. It's been
; recommended for several years that you not use the short tag "short cut"
; and
; instead to use the full <?php and ?> tag combination. With the wide spread
; use
; of XML and use of these tags by other languages, the server can become
; easily
; confused and end up parsing the wrong code in the wrong context. But
; because
; this short cut has been a feature for such a long time, it's currently
; still
; supported for backwards compatibility, but we recommend you don't use them.
; Default Value: On
; Development Value: Off
; Production Value: Off
; http://php.net/short-open-tag
short_open_tag = On
```

Puede leer más información sobre `short_open_tag` en <http://www.php.net/manual/es/ini.core.php#ini.short-open-tag>

AVISO:

Nótese que no existe posibilidad de modificar **`short_open_tag`** en tiempo de ejecución.

Conocer el valor actual de `short_open_tag`

Para conocer el valor actual de `short_open_tag`, ejecute el siguiente comando en una terminal:

```
php -r 'echo phpinfo();' | grep short_open_tag
```

Una forma resumida de lo anterior, puede ser también:

```
php -i | grep short_open_tag
```

dónde el parámetro **-i** da la misma salida que **phpinfo()**

También puede buscar este valor, directamente en el archivo `php.ini`:

```
grep short_open_tag /etc/php5/apache2/php.ini
```

AVISO:

Reemplace `/etc/php5/apache2/php.ini` por la ruta del `php.ini` en su servidor, de ser necesario.

Instrucciones y estructuras

Existen dos tipos de instrucciones en PHP: aquellas instrucciones que se ejecutan en una única línea y las estructuras de control que almacenan dichas instrucciones.

Las **instrucciones simples**, siempre deben finalizar con un punto y coma (;) mientras que las **estructuras de control**, encerrarán dichas instrucciones entre llaves { }.

```
<?php
estructura de control {
    instrucción 1;
    instrucción 2;
}
?>
```

Indentación en PHP

PHP es un lenguaje que no requiere de indentación (sangrado) para ser interpretado. Por el contrario, todo el código fuente

PHP puede ser escrito sin indentación, aunque esto, es una práctica desaconsejada, ya que al igual que el código embebido, dificulta la lectura y la consecuente escalabilidad y mantenimiento de la app.

Estandarización de código

Como regla de estilo, se sugiere utilizar indentación de 4 espacios en blanco y dejar una línea en blanco, entre estructuras de control.

La indentación, es utilizada para diferenciar estructuras de control y algoritmos, dentro del código fuente:

```
<?php
estructura de control 1 {
    instrucción a;

    estructura de control 1.1 {
        instrucción b;

        estructura de control 1.1.1 {
            instrucción c;
        }
    }

    estructura de control 1.2 {
        instrucción d;
    }
}

estructura de control 2 {
    instrucción e;
}
?>
```

Impresión en pantalla

En PHP, existen varias funciones para imprimir contenido en pantalla. Las dos funciones básicas son **echo** y **print**.

Estandarización de código

Como regla de estilo se sugiere optar por una de ellas y no utilizar ambas funciones en una misma app.

```
<?php  
echo "Hola Mundo";  
?>
```

Imprime Hola Mundo en pantalla.

```
<?php  
print "Adiós Mundo";  
?>
```

Imprime Adiós Mundo en pantalla.

Variables y tipos de datos básicos

Una variable es elemento destinado a almacenar datos. Ésta, puede almacenar datos numéricos (enteros o flotantes), cadenas de texto, booleano (verdadero [true] o falso [false]), etc.

Una variable se **define** (es decir, se crea), se le **asigna un valor** (es decir, se almacenan datos), puede **modificarse** (cambiar de valor) y **eliminarse**.

Definición de una variable: Las variables en PHP se definen anteponiendo el signo dólar (\$) seguido del nombre que se le quiera dar a esta.

Nombre de las variables: El nombre de éstas, debe guardar ciertas reglas:

- Solo pueden comenzar por carácter alfabético o guión bajo (_)
- El nombre puede estar conformado por mayúsculas, minúsculas, guiones bajos (_) y números

Asignación de valores: para asignar valor a una variable, se coloca el signo igual (=) seguido del valor.

Tipos de datos: cuando el valor de una variable, es una **cadena de texto**, éste, debe escribirse entre comillas dobles ("), aunque también entre comillas simples ('). A fin de estandarizar el código, utilizaremos siempre comillas dobles para cadenas de texto. Tanto los valores **numéricos** (ya sean éstos, números enteros o de coma flotante) y los **booleanos**, no requieren ser entre-comillados.

AVISO:

Para los números de coma flotante se utiliza el punto (.) y NO la coma (,)

Veamos un ejemplo de **definición y asignación de variables**:

```
<?php
$codigo_de_producto = 1284;
$nombre_producto = "Agua Mineral Manantial x 500 ml";
$precio = 3.75;
$vence = False;
$hay_stock = True;
$stock_en_deposito_1 = 20;
$stock_en_deposito_27 = 5;
$stock_en_deposito_73A = 54;
?>
```

Es posible además, **asignar a una variable el valor de otra variable**:

```
<?php
$nombre_de_producto_por_defecto = "Producto alimenticio";
$nombre_producto = $nombre_de_producto_por_defecto;
echo $nombre_producto; // imprime: Producto alimenticio
?>
```

También es posible, **insertar el valor de una variable, dentro de una cadena de texto**:

```
<?php
$nombre_de_producto_por_defecto = "Producto";
$nombre_producto = "$nombre_de_producto_por_defecto en oferta";
echo $nombre_producto; // imprime: Producto en oferta
?>
```

Pero ¿qué sucede si se necesita **concatenar el valor de una variable a una cadena de texto** pero sin mediar espacios?

Por ejemplo, si en el caso anterior, se desea que `$nombre_producto` sea "Productos en oferta". Estos casos, se resuelven envolviendo dicha variable entre llaves:

```
<?php
$nombre_de_producto_por_defecto = "Producto";
$nombre_producto = "{$nombre_de_producto_por_defecto}s en oferta";
echo $nombre_producto; // imprime: Productos en oferta
?>
```

En PHP, también es posible **concatenar variables** mediante el operador de concatenación "punto" (.):

```
<?php
$nombre_de_producto_por_defecto = "Producto";
$nombre_producto = $nombre_de_producto_por_defecto . " en oferta";
echo $nombre_producto; // imprime: Producto en oferta
?>
```

Aunque esta última práctica, debe utilizarse responsablemente, puesto que en determinadas ocasiones puede resultar difícil de leer y descifrar la salida final que tendrá:

```
<?php
$codigo_de_producto = 1284;
$nombre_producto = "Agua Mineral Manantial x 500 ml";
$precio = 3.75;
$vence = False;
$hay_stock = True;
$stock_en_deposito_1 = 20;
$stock_en_deposito_27 = 5;
$stock_en_deposito_73A = 54;

$detalles_del_producto = "(" . $codigo_de_producto . ") " .
$nombre_producto . ". Precio: USD " . $precio . "-";
?>
```

El ejemplo anterior, podría resultar más legible, de la siguiente forma:

```
<?php
$codigo_de_producto = 1284;
$nombre_producto = "Agua Mineral Manantial x 500 ml";
$precio = 3.75;
```

```
$vence = False;
$hay_stock = True;
$stock_en_deposito_1 = 20;
$stock_en_deposito_27 = 5;
$stock_en_deposito_73A = 54;

$detalles_del_producto = "($codigo_de_producto) $nombre_producto. Precio: USD
$precio.-";
?>
```

Para **modificar una variable**, reemplazando su valor, solo basta con reasignarle datos:

```
<?php
$codigo_de_producto = 1284;
$nombre_producto = "Agua Mineral Manantial x 500 ml";
$precio = 3.75;
$vence = False;
$hay_stock = True;
$stock_en_deposito_1 = 20;
$stock_en_deposito_27 = 5;
$stock_en_deposito_73A = 54;

$detalles_del_producto = "($codigo_de_producto) $nombre_producto. Precio: USD
$precio.-";

echo $detalles_del_producto;

$detalles_del_producto = "No hay detalles definidos";
print $detalles_del_producto;
?>
```

Null, var_dump() e isset()

No solo es posible modificar el valor de una variable. También es posible:

a) vaciarla manteniendo su tipo:

```
<?php
$producto = "Coca-Cola x 1,5 Lts.";
$producto = "";
?>
```


b) vaciarla sin conservar su tipo:

```
<?php
$producto = "Coca-Cola x 1,5 Lts.";
$producto = NULL;
?>
```

c) o, eliminarla (destruirla) por completo:

```
<?php
$producto = "Coca-Cola x 1,5 Lts.";
unset($producto);
?>
```

En todo momento, PHP nos permite **conocer el tipo y valor de una variable**, mediante la función **var_dump()**:

```
<?php
$producto = "Coca-Cola x 1,5 Lts.";
var_dump($producto);
# salida: string(20) "Coca-Cola x 1,5 Lts."

$producto = "";
var_dump($producto);
# salida: string(0) ""

$producto = NULL;
var_dump($producto);
# salida: NULL

unset($producto);
var_dump($producto);
/*
  Generará un error, ya que la variable $producto ha sido destruida
  Salida:
    PHP Notice:  Undefined variable: producto ...
    NULL
*/
?>
```

var_dump() imprimirá los resultados en pantalla (tipo y valor de una variable), pero también, es posible conocer el tipo de una variable (no su valor), sin imprimirlo en pantalla, con

gettype():

```
<?php
$a = 25;
$tipo_a = gettype($a);
echo $tipo_a; #imprimirá integer
?>
```

Es muy útil además, **saber si una variable ha sido definida** (y no se ha destruido con `unset()`) **y tiene un tipo asignado** (es decir, no es `NULL`). Para ello, dispones de la función **`isset()`**. Esta función, devolverá **`True`** si ha sido definida y no es `NULL`. De lo contrario, retornará **`False`**:

```
<?php
$producto = "Coca-Cola x 1,5 Lts.";
echo isset($producto);
# retorna True

$producto = "";
echo isset($producto);
# Retorna True

$producto = NULL;
echo isset($producto);
# retorna False

unset($producto);
echo isset($producto);
# retorna False
?>
```

¿NULL o unset()? ¿Cuál de los dos usar?

Cuando una variable ya no es necesaria, debe priorizarse el uso de **`unset`** sobre `NULL`, ya que con **`unset()`**, se libera la dirección de la memoria en la cual había sido escrita dicha variable.

Malas prácticas con variables, que afectan el uso de memoria

Es posible también, **agregar a una variable**, otros **datos al final de la cadena**. Para ello, se utiliza el signo punto (.) antecediendo al signo igualdad (=):

```
<?php
$codigo_de_producto = 1284;
$nombre_producto = "Agua Mineral Manantial x 500 ml";
$precio = 3.75;
$vence = False;
$hay_stock = True;
$stock_en_deposito_1 = 20;
$stock_en_deposito_27 = 5;
$stock_en_deposito_73A = 54;

$detalles_del_producto = "(";
$detalles_del_producto .= $codigo_de_producto;
$detalles_del_producto .= ") ";
$detalles_del_producto .= $nombre_producto;
$detalles_del_producto .= " Precio: USD ";
$detalles_del_producto .= $precio;
$detalles_del_producto .= "-";

echo $detalles_del_producto;
?>
```

Pero esta práctica, reduce el rendimiento de la aplicación, ya que cada instrucción, será almacenada en una dirección de memoria diferente, mientras que de la forma anterior, solo requiere una dirección de memoria para su almacenamiento.

Operadores aritméticos

PHP permite realizar operaciones aritméticas de lo más variadas y por consiguiente, utilizar PHP "como calculadora". Para ello, disponemos de los siguientes operadores aritméticos:

Operadores aritméticos

Ejemplo	Nombre	Resultado
$- \$a$	Negación	Opuesto de $\$a$.
$\$a + \b	Adición	Suma de $\$a$ y $\$b$.
$\$a - \b	Sustracción	Diferencia de $\$a$ y $\$b$.
$\$a * \b	Multiplicación	Producto de $\$a$ y $\$b$.
$\$a / \b	División	Cociente de $\$a$ y $\$b$.
$\$a \% \b	Módulo	Resto de $\$a$ dividido por $\$b$.

Referencia del manual oficial de PHP:

<http://www.php.net/manual/es/language.operators.arithmetic.php>

Haciendo cálculos con el IVA

```
<?php
$alicuota_iva = 21;
$codigo_de_producto = 1284;
$nombre_producto = "Agua Mineral Manantial x 500 ml";
$precio_bruto = 3.75;
$iva = 3.75 * 21 / 100;
$precio_netto = $precio_bruto + $iva;
?>
```

Estandarización de código

Utiliza nombres descriptivos para las variables;

Si el nombre es compuesto, separa cada palabra por un guión bajo;

Escribe los nombres de variables en minúsculas;

Cuando debas asignar múltiples valores a una variable, utiliza una sola instrucción toda vez que sea posible;

Utiliza comillas dobles para encerrar las cadenas de texto, en vez de comillas simples;

Utiliza espacios en blanco antes y después de un operador aritmético para facilitar la lectura;

settype ¿un bug no resuelto o feature objetable?

PHP, asume que un número encerrado entre comillas es lógicamente, una cadena de texto:

```
__eugenia_1978_esAR__@mydream:~$ php -r '$a = "33"; var_dump($a);'  
string(2) "33"
```

Sin embargo, realizará operaciones aritméticas de forma correcta, aunque alguno de los números, sea de tipo string:

```
__eugenia_1978_esAR__@mydream:~$ php -r '$a = "33"; $b = 10; echo $a + $b;'  
43
```

No obstante, si se intenta realizar una operación aritmética con cadenas de texto, que además de números, incluyan otro carácter, PHP, en vez de fallar y avisarnos del error, pasará por alto la variable conflictiva:

```
__eugenia_1978_esAR__@mydream:~$ php -r '$a = "E33"; $b = 10; echo $a + $b;'  
10
```

Existe una función para **convertir el tipo** de una variable `settype($variable, "nuevo_tipo")`:

```
__eugenia_1978_esAR__@mydream:~$ php -a
```

Interactive shell

```
php > $a = "33 manzanas";  
php > settype($a, "integer");  
php > var_dump($a);  
int(33)  
php >
```

Utilizando **settype**, "podríamos" asegurarnos realizar operaciones aritméticas seguras:

```
<?php  
$a = "33 manzanas";  
$b = 10;  
settype($a, "integer");  
echo $a + $b;  
# salida: 43  
?>
```

Sin embargo **¿prueba que sucede al ejecutar este código?**

```
$a = "manzanas 33";  
$b = 10;  
settype($a, "integer");  
echo $a + $b;
```

AVISO:

No confíes en **settype()** para efectuar operaciones aritméticas. Es preferible evitar su uso para estos casos.

HTML y PHP en un mismo archivo

Como se comentó anteriormente, es posible "mezclar" código HTML con PHP, sin necesidad de imprimir etiquetas HTML mediante PHP. Es decir, conservar la independencia de ambos lenguajes en el mismo archivo.

El mejor procedimiento para hacer esto, es comenzar escribiendo el código HTML y utilizando comentarios para recordarnos dónde debemos insertar el código PHP. En la programación funcional o estructurada, éste, es el mejor método. Puesto que nos asegura, la mayor legibilidad posible.

Vamos a ver un ejemplo de aquello que **no debe hacerse** y sugerir una **mejor práctica**.

Una mala práctica: colocar código HTML embebido dentro de variables de PHP

```
<?php
$alicuota_iva = 21;
$codigo_de_producto = 1284;
$nombre_producto = "Agua Mineral Manantial x 500 ml";
$precio_bruto = 3.75;
$iva = 3.75 * 21 / 100;
$precio_netto = $precio_bruto + $iva;

$producto = "<p><b>Producto:</b> ($codigo_de_producto) $nombre_producto<br/>
<b>Precio:</b> USD $precio_netto.- (IVA incluido)</p>";
?>
<!doctype html>
<html>
<head>
    <title>Detalles del producto <?php echo $nombre_producto; ?></title>
</head>

<body>
    <?php echo $producto; ?>
</body>
</html>
```

Una buena práctica para evitar lo anterior

```
<?php
$alicuota_iva = 21;
$codigo_de_producto = 1284;
$nombre_producto = "Agua Mineral Manantial x 500 ml";
$precio_bruto = 3.75;
$iva = 3.75 * 21 / 100;
$precio_netto = $precio_bruto + $iva;
?>
<!doctype html>
<html>
<head>
    <title>Detalles del producto <?php echo $nombre_producto; ?></title>
</head>

<body>
    <p><b>Producto:</b> (<?php echo $codigo_de_producto; ?>)
    <?php echo $nombre_producto; ?><br/>
    <b>Precio:</b> USD <?php echo $precio_netto; ?>.- (IVA incluido)</p>
</body>
</html>
```

AVISO:

Nótese que la instrucción **`<?=$foo; ?>`** es una abreviación de **`<?php echo $foo; ?>`** la cual solo se ejecutará de forma satisfactoria desde la versión 5.4 de PHP, aunque el valor de **`short_open_tag`** sea **`Off`**. En versiones anteriores, se debe utilizar **`<?php echo $foo; ?>`** o en su defecto, establecer en **`On`**, el valor de **`short_open_tag`** en **`php.ini`**

Comentando y documentando el código fuente

Una de las prácticas más recomendadas, consiste en "comentar" el código fuente. Comentar el código, significa "escribir referencias sobre el código fuente que nos ayuden a entenderlo".

En PHP, existen dos tipos de comentarios:

1. Los comentarios de una sola línea
2. Los comentarios de varias líneas (o docstrings)

Los **comentarios de una sola línea**, pueden escribirse anteceditos de una doble barra diagonal (//) o una almohadilla (#):

```
<?php
// alícuota del IVA (en porcentaje)
$alicuota_iva = 21;

// Datos del producto
$codigo_de_producto = 1284;
$nombre_producto = "Agua Mineral Manantial x 500 ml";
$precio_bruto = 3.75; # precio sin IVA

// Cálculos relacionados al IVA
$iva = 3.75 * 21 / 100;
$precio_netto = $precio_bruto + $iva; # Precio con IVA incluido
?>
```

Otra utilidad de comentar el código fuente, es recomendarnos aquello que nos queda por hacer. Para ello, se utiliza la palabra **TOD** (del inglés "to do" que en español significa "por hacer") al comentario:

```
<?php
// alícuota del IVA (en porcentaje)
$alicuota_iva = 21;

// Datos del producto
$codigo_de_producto = 1284;
$nombre_producto = "Agua Mineral Manantial x 500 ml";
```

```
$precio_bruto = 3.75; # precio sin IVA

// Cálculos relacionados al IVA
$iva = 3.75 * 21 / 100;
$precio_netto = $precio_bruto + $iva; # Precio con IVA incluido

# TODO calcular descuentos por cantidad
?>
```

En cambio, los **comentarios de varias líneas** se encierran entre `/*` y `*/` (como en CSS, por ejemplo):

```
<?php
// alícuota del IVA (en porcentaje)
$alícuota_iva = 21;

// Datos del producto
$código_de_producto = 1284;
$nombre_producto = "Agua Mineral Manantial x 500 ml";
$precio_bruto = 3.75; # precio sin IVA

// Cálculos relacionados al IVA
$iva = 3.75 * 21 / 100;
$precio_netto = $precio_bruto + $iva; # Precio con IVA incluido

/*
  TODO calcular descuentos por cantidad
  para ello, tener en cuenta los siguientes porcentajes
  de descuento:
  1 a 5 productos: 0%
  6 a 10 productos: 2.5%
  11 a 25 productos: 6.2%
  Más de 25: 10%
*/
?>
```

Inclusión de archivos en PHP

PHP nos permite insertar cualquier tipo de archivos con formato de texto, dentro de un archivo PHP.

Entre los tipos de archivos que podemos insertar dentro de un fichero .php, se encuentran aquellos con las siguientes extensiones: **.php**, **.txt**, **.htm**, **.html**, entre otros con formato de texto.

Para insertar archivos, PHP dispone de cuatro funciones:

1. **include:**
<http://www.php.net/manual/es/function.include.php>
2. **include_once:**
<http://www.php.net/manual/es/function.include-once.php>
3. **require:**
<http://www.php.net/manual/es/function.require.php> y
4. **require_once:**
<http://www.php.net/manual/es/function.require-once.php>

Estas cuatro funciones, necesitan recibir como parámetro², la ruta local o remota del archivo a ser incluido.

Ejemplos de inclusión de archivos locales:

```
<?php
include("archivo.php");
include_once("archivo.txt");
require("archivo.html");
require_once("archivo.htm");
```

² Más adelante, cuando hablemos de funciones, veremos detenidamente qué es y para qué sirve un parámetro. Por el momento, cuando hablemos de parámetro, entenderemos que se trata de un "valor" que debemos pasar/enviar a una función.

```
?>
```

Ejemplos de inclusión de archivos remotos:

```
<?php
include("http://www.miweb.com/archivo.php?foo=bar");
include_once("http://www.miweb.com/archivo.php");
require("http://www.miweb.com/archivo.html");
require_once("http://www.miweb.com/archivo.txt");
?>
```

Diferencia entre inclusión de archivos remotos y locales

Una diferencia fundamental, entre incluir archivos remotos y archivos locales, es que los archivos PHP remotos, serán interpretados previamente en el servidor de origen y "servidos" al servidor de destino (el que los incluye), ya interpretados. Sin embargo, cuando un archivo PHP local es incluido, no será previamente interpretado, sino que de eso, se encargará el archivo que lo incluyó.

Es decir, que si queremos incluir el archivo `mi_fichero.php` ya interpretado conforme el valor del parámetro "foo", si utilizamos:

```
include("mi_fichero.php?foo=15");
```

PHP arrojará un error, ya que buscará un archivo llamado `mi_fichero.php?foo=15` en vez de interpretarlo.

Sin embargo, podremos incluir remotamente para que se nos devuelva el archivo interpretado, mediante:

```
include("http://miweb.com/mi_fichero.php?foo=15");
```

No obstante, PHP dispone de una función alternativa SOLO para

estos casos (inclusión de archivos remotos ya interpretados) que incluso permite almacenar los datos recibidos en una variable:

```
$contenido = file_get_contents("http://miweb.com/mi_fichero.php?foo=15");
```

Diferencia entre include y require

Si bien las funciones `require()` e `include()` de PHP realizan una acción similar (importar un archivo), no son iguales.

include() intenta importar al archivo indicado y en caso de no poder hacerlo, **arroja un error y continúa** ejecutando el resto del script.

Sin embargo, la función **require()**, cuando no logra importar el archivo indicado, **arroja un error y finaliza** sin permitir que el resto del script continúe ejecutándose.

Include y require "_once"

La única diferencia que existe entre `include` e `include_once` y `require` y `require_once`, es que si el archivo indicado con `"_once"` ya ha sido incluido no volverá a importarse.

Estructuras de Control - Parte I (condicionales)

Definición

Una estructura de control es un **bloque de código que permite tomar decisiones de manera dinámica**, sobre código existente.

Condicionales if, else y else if

El condicional if, al igual que otras estructuras de control, permite tomar decisiones, partiendo de la base de evaluar si una determinada condición se cumple. El razonamiento de condicionales puede representarse como sigue:

```
si condicion X se cumple {  
    hacer esto  
} sino, si condicion Y se cumple: {  
    hacer esto otro  
} si no se cumple ni X ni Y {  
    hacer tal otra cosa  
}
```

No necesariamente el condicional debe cumplir la estructura anterior. A veces solo es necesario evaluar una única condición y tomar una decisión SOLO sobre la base de si esta condición se cumple:

```
si condición X se cumple {  
    hacer esto;  
    // fin de la evaluación  
}
```

La **sintaxis básica para los condicionales** en PHP, se resume en:

```

if (condición A) {
    // algoritmo si se cumple condición A
} else if (condición B) {
    // algoritmo si se cumple condición B
} else {
    // algoritmo si no se cumplen las condiciones anteriores
}

```

Operadores lógicos y de comparación

Para evaluar condiciones, no solo podemos recurrir a si "A es igual a B". Existen otros operadores que nos permiten evaluar diferentes condiciones. Estos operadores se denominan **operadores lógicos** y son aquellos que **nos permiten evaluar múltiples condiciones en un mismo proceso**, mientras que se denominan **operadores de comparación**, a aquellos que **utilizamos para evaluar (comparar) la relación existente entre elementos**.

Operadores de comparación

Ejemplo	Nombre	Resultado
<code>\$a == \$b</code>	Igual	TRUE si <code>\$a</code> es igual a <code>\$b</code> después de la manipulación de tipos.
<code>\$a === \$b</code>	Idéntico	TRUE si <code>\$a</code> es igual a <code>\$b</code> , y son del mismo tipo.
<code>\$a != \$b</code>	Diferente	TRUE si <code>\$a</code> no es igual a <code>\$b</code> después de la manipulación de tipos.
<code>\$a <> \$b</code>	Diferente	TRUE si <code>\$a</code> no es igual a <code>\$b</code> después de la manipulación de tipos.
<code>\$a !== \$b</code>	No idéntico	TRUE si <code>\$a</code> no es igual a <code>\$b</code> , o si no son del mismo tipo.
<code>\$a < \$b</code>	Menor que	TRUE si <code>\$a</code> es estrictamente menor que <code>\$b</code> .
<code>\$a > \$b</code>	Mayor que	TRUE si <code>\$a</code> es estrictamente mayor que <code>\$b</code> .
<code>\$a <= \$b</code>	Menor o igual que	TRUE si <code>\$a</code> es menor o igual que <code>\$b</code> .
<code>\$a >= \$b</code>	Mayor o igual que	TRUE si <code>\$a</code> es mayor o igual que <code>\$b</code> .

Diferencia entre igualdad e idéntico en la comparación

AVISO:

Nótese que la **diferencia** principal entre **igualdad (==)** e **idéntico (===)**, es que el primero convierte ambos elementos al mismo tipo (anulando así, la comparación de tipos, y arrojando que 10 será igual que "10"), mientras que el segundo, compara además si ambos elementos pertenecen al mismo tipo.

Lo anterior, **también aplica a != y !==**. Veamos algunos ejemplos:

```
$a = 10;
$b = "10";
$c = 11;
$d = 0;
$e = False;
```

```
$a == $b    True
$a === $b   False
$a != $b    False
$a !== $b   True
```

```
$d == $e    True
$d === $e   False
$d != $e    False
$d !== $e   True
```

Operadores lógicos

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

En el manual oficial de PHP

<http://www.php.net/manual/es/language.operators.logical.php>, podemos ver una leyenda que dice:

"La razón para tener las dos variaciones diferentes de los operadores "and" y "or" es que ellos operan con precedencias diferentes."

Lo cierto, es que tanto el operador **and** como el operador **&&** poseen una precedencia izquierda (es decir, que la condición es evaluada de izquierda a derecha) y lo mismo sucede con **or** y **||**.

Creando condicionales

Evaluando una única condición:

```
$a = 10;
if ($a == 10) {
    echo "\$a es igual a 10";
}
```

Iniciar A en 10
si (A es igual a 10), entonces
imprimir 'A es igual a 10'

O también, tomar una decisión si la condición se cumple y otra si no se cumple:

```
$a = 10;
if ($a == 10) {
    echo "\$a es igual a 10";
} else {
    echo "\$a NO es igual a 10";
}
```

También podemos combinar **else** e **if**, para tomar tantas decisiones como condiciones quieran evaluarse:

```
$a = 15;
if ($a == 10) {
```

```
    echo "\$a es igual a 10";  
} else if ($a == 12) {  
    echo "\$a es igual a 12";  
} else if ($a == 15) {  
    echo "\$a es igual a 15";  
}
```

Y si a lo anterior le agregamos else estaríamos cubriendo todas las posibilidades:

```
$a = 15;  
if ($a == 10) {  
    echo "\$a es igual a 10";  
} else if ($a == 12) {  
    echo "\$a es igual a 12";  
} else if ($a == 15) {  
    echo "\$a es igual a 15";  
} else {  
    echo "\$a NO es ni 10 ni 12 ni 15";  
}
```

AVISO:

Se debe tener especial cuidado en la comparación de números reales (flotantes). Veremos esto con detenimiento más adelante. Para mayor información, visitar la documentación oficial en:

<http://www.php.net/manual/es/language.types.float.php#language.types.float.comparison>

Estructuras de control con Switch

Switch es una de las estructuras de control, que mejor se disponen para programar la toma de decisiones basadas en la comparación de un único elemento.

Switch se asemeja sutilmente a la evaluación de condiciones mediante "else if", pero diferenciándose notablemente por su aplicación en la práctica, ya que es verdaderamente útil en aquellos casos donde la utilización de else if resulte redundante.

Básicamente, con Switch, lo que hacemos es **evaluar una única expresión o variable, y tomar diversas decisiones en base a los diferentes posibles valores de la misma**:

```
$numero_dia = date('N');  
/*  
la función nativa date() de PHP, permite dar formato a la fecha local  
N, retorna un número de 1 a 7, que representa el número de día de la  
semana, siendo 1 Lunes y 7 domingo.  
Esta función, será vista con detenimiento, más adelante.  
Para mayor información, visitar la documentación oficial en  
http://www.php.net/manual/es/function.date.php  
*/  
  
$nombre_dia = '';  
  
switch ($numero_dia) {  
    case 1:  
        $nombre_dia = "Lunes";  
        break;  
  
    case 2:  
        $nombre_dia = "Martes";  
        break;  
  
    case 3:  
        $nombre_dia = "Miércoles";  
        break;  
  
    case 4:  
        $nombre_dia = "Jueves";  
        break;  
  
    case 5:  
        $nombre_dia = "Viernes";  
        break;
```

```
case 6:
    $nombre_dia = "Sábado";
    break;

case 7:
    $nombre_dia = "Domingo";
    break;

default:
    $nombre = "No sabemos que día es";
}
```

El funcionamiento de **switch** puede resultar complejo de entender en un comienzo. Pero va a ir paso a paso.

Switch tiene una **sintaxis básica** que se compone de:

```
switch ($variable) {
    case "posible valor 1":
        // algoritmo a ejecutar si $variable == "posible valor 1"
        break;

    case "posible valor 3":
        // algoritmo a ejecutar si $variable == "posible valor 3"
        break;

    default:
        // algoritmo a ejecutar si valor no ha sido contemplado en
        // ninguno de los «case» anteriores
}
```

Cada **case** representa un "caso" (el posible valor que pueda retornar la variable evaluada).

La palabra clave reservada **break**, "rompe" la ejecución de la estructura. Es decir, que cuando un case es evaluado como verdadero, tras ejecutar el algoritmo de ese case, la palabra clave reservada **break** indica la finalización de toda la estructura (es decir, no se continúan evaluando los siguientes "case").

La palabra clave reservada **default**, contendrá el algoritmo a ser ejecutado, cuando la evaluación de todos los "case" anteriores, haya sido False.

A fin de evitar la redundancia en el código (por ejemplo, si se debiera aplicar el mismo algoritmo a dos o más valores distintos), switch nos permite agrupar los casos:

```
switch ($variable) {
    case "posible valor 1":
    case "posible valor 2":
    case "posible valor 3":
        /* algoritmo a ejecutar si el valor de $variable es
           posible valor 1, posible valor 2 o posible valor 3
        */
        break;

    case "posible valor 4":
        /* algoritmo a ejecutar si el valor de $variable es
           posible valor 4
        */
        break;

    default:
        // algoritmo a ejecutar si valor no ha sido contemplado en
        // ninguno de los «case» anteriores
}
```

¿Cuándo utilizar if y cuando switch?

Si bien el uso de una u otra estructura, dependerá de la decisión de cada programador, existe un sentido lógico que debe priorizarse para decidir su uso.

Para ello, hay que tener en cuenta que:

if nos sirve para evaluar "condiciones" y comparar múltiples elementos, mientras que switch, solo permite la evaluación de un único elemento o expresión. Por lo tanto, toda vez que solo se requiera la evaluación de un único elemento o expresión, se mayormente (con ciertas excepciones), se utilizará switch y se optará por if, cuando el uso de switch no sea posible.

Tipos de Datos complejos: Matrices simples y multidimensionales

Matrices en PHP

Una matriz (array) es un mapa de datos ordenado que asocia "claves" a sus valores correspondientes. Es así, que estas matrices, nos son de gran utilidad para crear desde diccionarios de datos hasta árboles de múltiples diccionarios.

Sintaxis básica

La **sintaxis básica** se compone de:

```
array(clave => valor, );
```

Donde **clave**, puede ser un **entero**:

```
$nombres_de_mujer = array(0 => 'Ana', 1 => 'Gabriela', 2 => 'Cecilia', );
```

O una **cadena de texto**:

```
$telefonos_de_amigos = array('Juan' => '15 4017-2530',  
                             'Javier' => '4921 - 1200',);
```

Y **valor**, cualquier tipo de dato:

```
$datos_de_juan = array('apellido' => 'Pérez',  
                      'Fecha de nacimiento' => '23-11-1970',  
                      'Teléfonos' => array('Casa' => '4310-9030',
```

```

2530',
9000'),
                                'Móvil' => '15 4017-
                                'Trabajo' => '4604-
                                'Casado' => True,
                                'Pasaporte' => False,
                                );

```

La forma para declarar un array, es simplemente asignarlo a una variable, teniendo en cuenta, que un array, puede estar inicialmente vacío:

```
$mi_array = array();
```

Imprimir en pantalla con print_r

Para **imprimir una matriz completa en pantalla**, se puede utilizar la función **print_r(\$array)**:

```

php > $array = array(0 => 'Ana', 1 => 'Gabriela', 2 => 'Noelia',);
php > print_r($array);
Array
(
    [0] => Ana
    [1] => Gabriela
    [2] => Noelia
)
php >

```

Acceso a los ítems de un array

Para **acceder a un ítem del array**, se puede realizar haciendo una llamada a su **clave**, o por su **número de índice** (número de almacenamiento interno), siendo **0** (cero) el primero:

```

$apellidos = array('Ana' => 'Rodriguez', 'Marcos' => 'Gómez',);
echo $apellidos['Ana']; // imprime Rodriguez
echo $apellidos[1]; // imprime Gómez

```

AVISO:

La sintaxis básica para acceder a un array es `$array[indice]` o `$array['clave']`

Los valores de un array, **pueden no tener una clave explícitamente asociada:**

```
$mi_array = array('Ana', 'Gabriela', 'Julia', 'Noelia');
```

En ese caso, siempre se accederá a ellos por su número de índice:

```
$mi_array = array('Ana', 'Gabriela', 'Julia', 'Noelia');  
echo $mi_array[2]; // imprimirá Julia
```

Modificando elementos

Para modificar un elemento, basta con acceder a éste y asignarle un nuevo valor:

```
$mi_array = array('Ana', 'Gabriela', 'Julia', 'Noelia');  
$mi_array[2] = 'Ximena';  
echo $mi_array[2]; // imprimirá Ximena
```

Agregar elementos

Para agregar un valor a un array existente, se asigna éste a un índice vacío:

```
$mi_array = array('Ana', 'Gabriela', 'Julia', 'Noelia');  
$mi_array[] = 'Cecilia';  
print_r($mi_array);
```

El nuevo valor, será agregado al final de la lista:


```
php > print_r($mi_array);
Array
(
    [0] => Ana
    [1] => Gabriela
    [2] => Julia
    [3] => Noelia
    [4] => Cecilia
)
```

Pero **si se desea asociar dicho valor a una clave**, ésta debe indicarse:

```
$telefonos_de_amigos = array('Juan' => '15 4017-2530',
                             'Javier' => '4921 - 1200',);

$telefonos_de_amigos['Luis'] = '4321-5012';
$telefonos_de_amigos['Carlos'] = '15 3239-0432';
```

ADVERTENCIA sobre claves de matrices:

Al usar **TRUE** como clave, el valor será evaluado al integer **1**. Al usar **FALSE** como clave, el valor será evaluado al integer **0**. Al usar NULL como clave, el valor será evaluado a un **string vacío**.

El uso de un string vacío como clave, creará (o reemplazará) una clave con un string vacío y su valor; **no es lo mismo que usar corchetes vacíos**.

Vale aclarar además, que si **por error** intentamos agregar un nuevo elemento, usando como clave o como índice, alguna clave o índice existente, estaríamos MODIFICANDO dicho elemento en vez de estar agregando uno nuevo.

Estructuras de Control - Parte II (bucles I)

Recorriendo matrices dinámicamente con foreach

Foreach es un constructor nativo de PHP, que permite realizar operaciones iterativas (*cíclicas*) con matrices, recorriendo uno a uno los elementos de una matriz, comenzando por el primer elemento.

Sintaxis básica del constructor foreach

```
foreach($array as $valor_del_elemento) {  
    // algoritmo a realizar con cada uno de los elementos  
}
```

Dónde **\$array** será el nombre de la matriz a ser iterada y **\$valor_del_elemento**, un nombre que utilizaremos como identificador del elemento, el cual retornará su valor:

```
$nombres_propios = array('Ana', 'Julia', 'Luisa', 'Alberto', 'Cecilia',  
    'Carlos',);  
  
foreach($nombres_propios as $nombre) {  
    echo $nombre . chr(10);  
}  
/*  
Salida:  
Ana  
Julia  
Luisa  
Alberto  
Cecilia  
Carlos  
*/
```

Es posible también, **iterar obteniendo las claves de cada elemento**, además de su valor. Para ello, se utiliza la siguiente sintaxis:

```
foreach($array as $clave => $valor) {  
    // algoritmo a ejecutar en cada iteración  
}
```

Un ejemplo de iteración compleja con foreach

```
$datos_de_juan = array('Apellido' => 'Pérez',  
                      'Fecha de nacimiento' => '23-11-1970',  
                      'Teléfonos' => array('Casa' => '4310-9030',  
                                           'Móvil' => '15 4017-2530',  
                                           'Trabajo' => '4604-9000'),  
                      'Casado' => True,  
                      'Pasaporte' => False,  
                      );  
  
foreach($datos_de_juan as $titulo => $dato) {  
    if(!is_array($dato)) {  
        if($dato === True) {  
            $dato = 'SI';  
        } else if ($dato === False) {  
            $dato = 'NO';  
        }  
        echo "{$titulo}: {$dato}" . chr(10);  
    } else {  
        foreach($dato as $tipo_telefono => $numero) {  
            echo "Teléfono {$tipo_telefono}: {$numero}" . chr(10);  
        }  
    }  
}  
/* Apellido: Prez  
Fecha de nacimiento: 23-11-1970  
Teléfono Casa: 4310-9030  
Teléfono Móvil: 15 4017-2530  
Teléfono Trabajo: 4604-9000  
Casado: SI  
Pasaporte: NO  
*/
```

AVISO:

La función `is_array($array)` nos permite evaluar una variable y conocer si su tipo es "array". Devuelve **TRUE** si efectivamente es un array y **FALSE** en caso contrario.

Modificando matrices dinámicamente con foreach

En el ejemplo anterior, evaluábamos si el dato recibido era True o False, asignando un nuevo valor a éste (SI para True y NO para False).

Pero, si tras finalizar el bucle, hiciéramos un `print_r()` a `$datos_de_juan`, el valor de la clave "casado" continuaría siendo True, mientras que el de "Pasaporte", False:

```
php > print_r($datos_de_juan);
Array
(
    [Apellido] => Prez
    [Fecha de nacimiento] => 23-11-1970
    [Telfonos] => Array
        (
            [Casa] => 4310-9030
            [Mvil] => 15 4017-2530
            [Trabajo] => 4604-9000
        )
    [Casado] => 1
    [Pasaporte] =>
)
```

Es decir, que ese dato, solo fue modificado en un ámbito local, el cual aplica solo a esa estructura de control.

Pero ¿qué sucede si queremos modificar globalmente los valores de los elementos de una matriz?

PHP, nos facilita esa opción, **asignando el valor por referencia**. Una asignación por referencia, se realiza

antecediendo el signo & al valor:

```
$datos_de_juan = array('Apellido' => 'Pérez',
                      'Fecha de nacimiento' => '23-11-1970',
                      'Teléfonos' => array('Casa' => '4310-9030',
                                           'Móvil' => '15 4017-2530',
                                           'Trabajo' => '4604-9000'),
                      'Casado' => True,
                      'Pasaporte' => False,
                      );

foreach($datos_de_juan as $titulo => &$dato) {
    if(!is_array($dato)) {
        if($dato === True) {
            $dato = 'SI';
        } else if ($dato === False) {
            $dato = 'NO';
        }
        echo "{$titulo}: {$dato}" . chr(10);
    } else {
        foreach($dato as $tipo_telefono => $numero) {
            echo "Teléfono {$tipo_telefono}: {$numero}" . chr(10);
        }
    }
}
```

Si tras el caso anterior, hiciéramos un `print_r()` notaríamos que los valores de "Casado" y "Pasaporte" han modificado su valor y su tipo:

```
php > print_r($datos_de_juan);
Array
(
    [Apellido] => Prez
    [Fecha de nacimiento] => 23-11-1970
    [Telfonos] => Array
        (
            [Casa] => 4310-9030
            [Mvil] => 15 4017-2530
            [Trabajo] => 4604-9000
        )
    [Casado] => SI
    [Pasaporte] => NO
)
```

La **asignación por referencia**, suele ser muy útil, cuando por ejemplo, se necesita aplicar una misma función, a todos los elementos de un array (por ejemplo, convertir a mayúsculas

todos los valores, con la función **strtoupper()**, nativa de PHP):

```
$nombres = array('Ana', 'Julia', 'Luisa', 'Alberto', 'Cecilia', 'Carlos',);

foreach($nombres as &$nombre) {
    $nombre = strtoupper($nombre);
}

print_r($nombres);
/*
Array
(
    [0] => ANA
    [1] => JULIA
    [2] => LUISA
    [3] => ALBERTO
    [4] => CECILIA
    [5] => CARLOS
)
*/
```

Estructuras de Control - Parte III (bucles II)

While, un bucle simple

Así como **foreach** puede parecer uno de los bucles más complejos, **while**, resulta ser el más simple de todos.

while, simplemente evaluará de forma booleana (true o false) una expresión de iterativamente, hasta que la expresión evaluada retorne **False**, y parará.

Su **sintaxis** es la siguiente:

```
while (expresión) {  
    // algoritmo a ejecutar hasta expresión retorne False  
}
```

O dicho de forma humanamente legible:

```
mientras que (esta condición se cumpla) {  
    hacer esto  
}
```

Un ejemplo sencillo

```
$n = 0;  
while ($n <= 5) {  
    echo $n . chr(10);  
    $n++; // incremento el valor de $n en 1. Equivale a $n = $ + 1;  
}
```

Lectura humana:

iniciar N en cero

```
mientras que (N sea menor o igual a 5) {  
    imprimir N  
    incrementar N  
}
```

Un ejemplo práctico

```
$years = array();  
$year = 1990;  
  
while ($year <= 2000) {  
    $years[] = $year;  
    $year++;  
}  
  
print_r($years);  
  
/*  
Array  
(  
    [0] => 1990  
    [1] => 1991  
    [2] => 1992  
    [3] => 1993  
    [4] => 1994  
    [5] => 1995  
    [6] => 1996  
    [7] => 1997  
    [8] => 1998  
    [9] => 1999  
    [10] => 2000  
)  
*/
```

Vale la pena hacer notar, que si al iniciar una iteración con `while`, la primera expresión es falsa, no se continuará ejecutando el bucle:

```
$years = array();  
$year = 1990;  
  
while ($year < 1990) {  
    $years[] = $year;  
    $year++;  
}  
  
print_r($years);  
  
/*
```



```
Array  
(  
)  
*/
```

Do while, tan simple como while pero con una ventaja

El bucle do-while es, como indica el título, tan simple como while y funciona de manera prácticamente idéntica. La única diferencia, el algoritmo iterativo se ejecutará sí o sí, una vez al comienzo y luego, evaluará la expresión, y volverá a ejecutar el algoritmo si la expresión es verdadera.

Su **sintaxis** es la siguiente:

```
do {  
    /* algoritmo a ejecutarse al principio de la iteración  
       y toda vez que expresión sea verdadera  
    */  
} while ($expresion);
```

Lectura humana:

```
hacer {  
    esto, la primera y vez y luego hacerlo...  
} mientras que (esta condición se cumpla);
```

Un ejemplo simple

```
$years = array();  
$year = 1990;  
  
do {  
    $years[] = $year;  
    $year++;  
} while ($year < 1990);
```

```
print_r($years);

/*
Array
(
    [0] => 1990
)
*/

print $year;
// 1991
```

Bucles for, los más complejos de PHP

Los bucles for son los ciclos iterativos más complejos de los cuáles se dispone en PHP. Éstos, evalúan 3 expresiones en cada iteración, siguiendo este esquema:

- **Primera expresión:** se ejecuta incondicionalmente al comienzo del bucle;
- **Segunda expresión:** se evalúa como verdadera o falsa al inicio de cada iteración. El bucle continúa, solo si es verdadera.
- **Tercera expresión:** se ejecuta al final de cada iteración

Sintaxis:

```
for (expresion1; expresion2; expresion3) {  
    // algoritmo que se ejecuta cuando expresion2 es verdadera  
}
```

Un ejemplo:

```
for ($i = 0; $i <= 3; $i++) {  
    echo $i . chr(10);  
}  
  
/*  
0  
1  
2  
3  
*/
```

Cada una de las expresiones, puede contener **múltiples expresiones**, las cuales deberán ir **separadas por una coma**:

```

for ($i = 1, $prefijo = "Codigo N° 000"; $i <= 4; $sufijo = chr(10), $i++) {
    echo "{$prefijo}{$i}{$sufijo}";
}

/*
Codigo N° 0001
Codigo N° 0002
Codigo N° 0003
Codigo N° 0004
*/

```

AVISO:

Al utilizar expresiones múltiples, se debe tener en cuenta, que las **expresiones múltiples en la segunda expresión** son evaluadas, pero el resultado solo se toma de la última parte.

Un ejemplo absurdo, pero que grafica el aviso anterior:

```

for ($i = 1, $prefijo = "Codigo N° 000"; $i <= 4, $i < 2; $sufijo = chr(10), $i++) {
    echo "{$prefijo}{$i}{$sufijo}";
}

/*
Codigo N° 0001
*/

```

No obstante, es posible que puedan coexistir **expresiones vacías**:

```

for ($i = 1, $prefijo = "Codigo N° 000", $sufijo = chr(10); ; $i++) {
    if ($i > 4) {
        break;
    }
    echo "{$prefijo}{$i}{$sufijo}";
}

/*
Codigo N° 0001
Codigo N° 0002
Codigo N° 0003
Codigo N° 0004
*/

```

* /

AVISO:

Si la **segunda expresión está vacía**, el bucle será corrido de forma indefinida, motivo por el cual, debe utilizarse en algún punto, la expresión break (vista en switch). De lo contrario, podría producirse un desbordamiento de la memoria. Esto, es debido a que PHP, considera la expresión vacía como True (al igual que C).

Curiosidades sintácticas de la bipolaridad no diagnosticada de PHP

Queremos a PHP y por eso lo usamos ¿cierto? Pero aquello de que "el amor es ciego", es bastante relativo, cuando lo que nos guía como programadores, es la lógica. Y es allí, cuando nos encontramos con que PHP, pareciera ser "bipolar" y, según su ánimo, la sintaxis de las estructuras de control, pueden escribirse de dos formas: la que ya conocemos y... esta de aquí:

```
if($PHP == "es bipolar"):
    echo "No te aconsejo utilizar esta sintaxis :D";
endif;
```

Y a veces, está un poco deprimido:

```
if($PHP == "es bipolar"):
    echo "No te aconsejo utilizar esta sintaxis :D";
else if ($PHP == "está en su fase depresiva"):
    echo "Oh! No! Debiste utilizar elseif sin separación! Ahora fallará!";
endif;
```

Por curiosidad, el ejemplo anterior pero sin fallas:

```
if($PHP == "es bipolar"):
    echo "No te aconsejo utilizar esta sintaxis :D";
elseif ($PHP == "está en su fase depresiva"):
    echo "Ok. Usemos elseif sin separación así no se enoja";
else:
    echo "Ya podemos volver a utilizar llaves";
endif;
```

¡Oh, no! Con while también sucede!

```
while ($i < 10):
    echo "Y aquí tenemos un PHP maniaco!";
endwhile;
```

Y tal cual lo imaginas, efectivamente, con for y foreach, PHP puede ser “bipolar”. ¿No me crees? Mira esto!

```
for ($i = 0; $i < 3; $i++):  
    echo $i . chr(10); endfor;  
$array = array(1, 2, 3);  
foreach($array as $a):  
    echo $a . chr(10); endforeach;
```

Goto, si lo usas... es tu elección!

Si todo lo anterior, te dejó la frase “*no quería saber tanto*”, rondando en tu cabeza, sabrás disculparme, pero debo contarte también sobre **goto** (aunque entre nos, a veces puede ser útil).

goto es un “operador” al que podríamos llamar “*operador sintáctico*”, que **se utiliza para “saltar” a una zona específica del programa.** El punto de destino es especificado mediante una etiqueta seguida de dos puntos y la instrucción es dada como goto seguida de la etiqueta del destino deseado.

Goto podría reemplazar en ocasiones, el uso de break, puesto que solo puede ser utilizado dentro de una misma estructura y con varias restricciones.

Un ejemplo no-práctico para entender goto

```
<?php  
  
echo "Hola Mundo!";  
  
goto mi_etiqueta;  
  
echo "Esto no se mostrará, ya que goto lo saltará";  
  
mi_etiqueta:
```

```
    echo chr(10) . "Esto sí será mostrado" . chr(10);  
    echo "Y esta también!" . chr(10);  
    ?>
```

Lo anterior, generará la siguiente salida:

```
eugenia@cocochito:~/cursophpbasico$ php -f file.php  
Hola Mundo!  
Esto sí será mostrado  
Y esta también!  
eugenia@cocochito:~/cursophpbasico$
```

Sin embargo, al no tener las etiquetas definidas, un identificador de cierre y, PHP, ser un lenguaje que no requiere de indentación para la definición de estructuras, todo el código escrito debajo de una etiqueta, será ejecutado cuando goto sea llamado:

```
<?php  
echo "Hola Mundo!";  
goto mi_etiqueta;  
  
echo "Esto no se mostrará, ya que goto lo saltará";  
  
mi_etiqueta:  
    echo chr(10) . "Esto sí será mostrado" . chr(10);  
  
otra_etiqueta:  
    echo "mmm... esta también se muestra :/" . chr(10);  
    ?>
```

Y aquí, la prueba:

```
eugenia@cocochito:~/cursophpbasico$ php -f file.php  
Hola Mundo!  
Esto sí será mostrado  
mmm... esta también se muestra :/  
eugenia@cocochito:~/cursophpbasico$
```


Por lo tanto, utilizar goto, puede ser engorroso ya que para evitar lo anterior, deberían especificarse más y más etiquetas y más y más gotos:

```
<?php
echo "Hola Mundo!";
goto mi_etiqueta;

echo "Esto no se mostrará, ya que goto lo saltará";

mi_etiqueta:
    echo chr(10) . "Esto sí será mostrado" . chr(10);
    goto ya_basta_de_ejecutarse;

otra_etiqueta:
    echo "Ahora ya no voy a imprimirme!" . chr(10);

ya_basta_de_ejecutarse:
    exit();
?>
```

Como se ve a continuación, esta vez, lo logramos:

```
eugenia@cocochito:~/cursophpbasico$ php -f file.php
Hola Mundo!
Esto sí será mostrado
eugenia@cocochito:~/cursophpbasico$
```

Puedes ver más sobre goto, leyendo la documentación oficial:
<http://www.php.net/manual/es/control-structures.goto.php>

Funciones definidas por el usuario

Definición

Una función, es una forma de agrupar expresiones y sentencias (algoritmos) que realicen determinadas acciones, pero que éstas, solo se ejecuten cuando son llamadas (al igual que las funciones nativas de PHP).

Declarando Funciones

Sintaxis básica

La sintaxis básica de una función, es verdaderamente sencilla:

```
function nombre_de_la_funcion(parametros) {  
    // algoritmo  
}
```

Sobre el nombre de las funciones

Para el nombre de las funciones, aplica todo lo dicho para el nombre de variables.

Sobre los parámetros

Un parámetro es un valor, que la función espera a fin de ejecutar acciones en base al mismo. Una función puede esperar uno o más parámetros (que irán separados por una coma) o ninguno.

```
function nombre_de_la_funcion(parametro1, parametro2) {  
    // algoritmo  
}  
  
function otra_funcion() {  
    // algoritmo  
}
```

Los parámetros, se indican entre los paréntesis, a modo de variables, a fin de poder utilizarlos como tal, dentro de la misma función:

```
function nombre_de_la_funcion($parametro1, $parametro2) {  
    // algoritmo  
}
```

Además, a cada parámetro, se le puede asignar un valor por defecto de cualquier tipo:

```
function nombre_de_la_funcion($nombre, $edad=25, $sexo='F') {  
    // algoritmo  
}
```

Llamando a una función

Una función no será ejecutada nunca, hasta que no se la llame:

```
<?php  
function hola_mundo() {  
    echo "Hola Mundo!" . chr(10);  
}  
?>
```

Ninguna salida será obtenida de lo anterior, puesto que la función, no ha sido llamada:

```
eugenia@cocochito:~$ php -f file.php  
eugenia@cocochito:~$
```

Para **llamar a una función**, simplemente debe especificarse su nombre, en el lugar preciso donde se desea que ésta se ejecute. Siempre deben incluirse los paréntesis en las llamadas, incluso aunque no requieran que un parámetro les sea pasado:

```
<?php
function hola_mundo() {
    echo "Hola Mundo!" . chr(10);
}

hola_mundo();
?>
```

Ahora sí, se obtendrá el resultado de la ejecución de la función:

```
eugenia@cocochito:~$ php -f file.php
Hola Mundo!
eugenia@cocochito:~$
```

Sobre la finalidad de las funciones

Una función, puede tener cualquier tipo de algoritmo y cualquier cantidad de ellos y, utilizar cualquiera de las características vistas hasta ahora.

No obstante ello, **una buena práctica, indica que la finalidad de una función, debe ser realizar una única acción, reutilizable y por lo tanto, tan genérica como sea posible.**

Paso de variables por referencia en funciones

Al igual que en el bucle foreach, es posible pasar variables por referencia a una función.

Para ello, al definirse la función, debe colocarse el signo “&” antecediendo a aquellos parámetros que hagan referencia a una variable global:

```
// definimos una variable de ámbito global
$mi_variable_global = 10;

// definimos una función que modificará la variable global
function modificar_variable_global(&$variable, $otro_parametro) {
    $variable = $variable * $otro_parametro;
}

// llamamos a la función pasando como referencia la variable global
modificar_variable_global($mi_variable_global, 2);

// imprimimos la variable global
echo $mi_variable_global; // salida: 20
```

Modificando variables globales mediante el uso de global

En PHP, es posible también, modificar una variable de ámbito global, sin necesidad de pasarla como referencia.

Para ello, dentro de la función, se hará referencia a la variable global a ser utilizada, mediante el uso de la palabra clave “global”. El siguiente ejemplo, es sinónimo del anterior:

```
// definimos una variable de ámbito global
$mi_variable_global = 10;

// definimos la función que hará referencia a la variable global
function modificar_variable_global($otro_parametro) {
    global $mi_variable_global;
    $mi_variable_global = $mi_variable_global * $otro_parametro;
}

// llamamos a la función
modificar_variable_global(2);

// imprimimos la variable global
echo $mi_variable_global; // salida: 20
```

AVISO

Nótese que si la variable global a la cual se hace referencia dentro de la función, no ha sido declarada previamente, **global \$mi_variable** creará la variable global \$mi_variable.

Llamadas de retorno

En PHP, es posible (al igual que en la gran mayoría de los lenguajes de programación), llamar a una función dentro de otra, de forma fija y de la misma manera que se la llamaría, desde fuera de dicha función:

```
function mi_funcion($parametro) {  
    mi_otra_funcion();  
    $una_variable = otra_funcion_mas($parametro);  
}
```

Sin embargo, es posible que se desee realizar dicha llamada, de manera dinámica, es decir, desconociendo el nombre de la función a la que se deseará llamar. A este tipo de acciones, se las denomina **llamadas de retorno**.

En una llamada de retorno, **el nombre de la función** a la cual se desea llamar, **es pasado como una cadena de texto** y para ello, se utiliza la función nativa de PHP, **call_user_func('nombre_de_la_funcion_a_llamar')**.

```
// Función que llamaré desde otra función  
function decir_hola() {  
    return "Hola Mundo!";  
}  
  
// Función que hará la llamada de retorno  
function llamar_a_otra($funcion) {  
    echo call_user_func($funcion);  
    echo chr(10);  
    // continuación del algoritmo
```

```
}  
  
llamar_a_otra('decir_hola');
```

Pasar argumentos en una llamada de retorno

¿Qué sucede si la función a la cual se desea llamar, necesita recibir uno o más argumentos? En este caso, tenemos dos opciones:

1) Pasar los argumentos a continuación del nombre de la función:

```
call_user_func('nombre_de_la_funcion', $parametro1, $parametro2);
```

Ejemplo:

```
// Función que llamaré con call_user_func  
function sumar_dos_numeros($a, $b) {  
    return $a + $b;  
}  
  
$numero_1 = 5;  
$numero_2 = 10;  
  
$resultado = call_user_func('sumar_dos_numeros',  
                            $numero_1,  
                            $numero_2);  
  
echo $resultado;
```

2) Definir un array con todos los argumentos necesarios, y hacer la llamada de retorno ampliada:

```
call_user_func_array('nombre_de_la_funcion', $array_con_argumentos);
```

Ejemplo:

```
// Función que llamaré con call_user_func_array  
function sumar_dos_numeros($a, $b) {
```

```
        return $a + $b;
    }

    $args = array(5, 10);
    $resultado = call_user_func_array('sumar_dos_numeros', $args);
    echo $resultado;
```

Argumentos no conocidos

Cuando dentro de una función, realizamos una llamada de retorno, así como la función puede desconocer el nombre de aquella a la cual deberá llamar, también es probable que si esa función requiere de argumentos, se desconozca también la cantidad de argumentos.

Para resolver este planteo, contamos con varias funciones nativas que nos ayudarán a lograr un mejor tratamiento de las llamadas de retornos. Veamos algunas de ellas.

Conocer la cantidad de argumentos

Con la **func_num_args** podemos conocer exactamente, la cantidad de argumentos recibidos en una función:

```
function foo() {
    $cantidad_de_argumentos = func_num_args();
    echo "Recibimos {$cantidad_de_argumentos} argumentos";
}

foo('argumento 1', 'otro_argumento');
```

Como podemos notar, la función `foo()` en realidad, no esperaba ningún argumento. Sin embargo, al hacer la llamada a `foo()`, hemos pasado dos argumentos.

Obtener una lista completa de todos los argumentos

Es posible obtener una matriz (array) con todos los argumentos recibidos. Para ello, disponemos de la función **func_get_args**:

```
function foo() {
    $argumentos = func_get_args();
    print_r($argumentos);
    /*
        Retornará un array con todos los argumentos:
        Array
        (
            [0] => argumento 1
            [1] => otro argumento
        )
    */
}

foo('argumento 1', 'otro argumento');
```

Obtener un argumento específico

Puede ser muy útil además, obtener un argumento determinado. Para ello, disponemos de la función **func_get_arg(index)**, donde index, será el número de índice del argumento en la matriz:

```
function foo3() {
    echo func_get_arg(1);
    // salida: otro argumento
}

foo3('argumento 1', 'otro argumento');
```

Saber si una función puede ser llamada (callable)

*Cuando decimos **callable** nos referimos a si la función existe y además, puede ser llamada.*

Cuando trabajamos con llamadas de retorno, asumimos que nuestro script, desconoce el nombre de la función a la que se desea llamar y por lo tanto, no debemos confiar en que el nombre de la función pasada como cadena de texto, sea efectivamente el nombre de una función callable.

Para sortear este obstáculo, disponemos de la función **is_callable**, la cual nos retornará TRUE en caso de ser una función callable. De lo contrario, retornará FALSE.

```
function funcion_callable() {
    echo "Llamada correcta";
}

function llamar_a_funcion_callable($funcion) {
    if(is_callable($funcion)) {
        call_user_func($funcion);
    } else {
        echo "La función no es callable";
    }
}

llamar_a_funcion_callable('funcion_callable');
// salida: Llamada correcta

llamar_a_funcion_callable('funcion_inexistente');
// salida: La función no es callable
```

Material de lectura adicional

- Sobre **call_user_func**
<http://www.php.net/manual/es/function.call-user-func.php>
- Sobre **call_user_func_array**
<http://www.php.net/manual/es/function.call-user-func-array.php>
- Sobre **func_num_args**

<http://www.php.net/manual/es/function.func-num-args.php>

- Sobre **func_get_args**

<http://www.php.net/manual/es/function.func-get-args.php>

- Sobre **func_get_arg**

<http://www.php.net/manual/es/function.func-get-arg.php>

- Sobre **is_callable**

<http://www.php.net/manual/es/function.is-callable.php>

- Sobre **function_exists** (alternativa a `is_callable` que solo comprueba si la función existe o no, pero no verifica que ésta, pueda ser llamada)

<http://www.php.net/manual/es/function.function-exists.php>

Diferentes formas de recoger argumentos para hacer una llamada de retorno

Veremos aquí, dos formas de crear funciones para hacer llamadas de retorno que requieran de argumentos.

Forma 1: recibir argumentos en un array

```
// función callable
function callable_func_1($arg1, $arg2, $arg3) {
    $result = ($arg1 + $arg2) * $arg3;
    return $result;
}

// función que hará la llamada de retorno
function forma_1($funcion, $argumentos=array()) {
    $result = NULL;

    if(is_callable($funcion)) {
        $result = call_user_func_array($funcion, $argumentos);
    }
}
```

```
        return $result;
    }

    // implemenatción
    $args = array(10, 5, 2);
    $resultado = forma_1('callable_func_1', $args);
    echo $resultado;
```

Forma 2: recibir argumentos 1 a 1

```
// función callable
function callable_func_1($arg1, $arg2, $arg3) {
    $result = ($arg1 + $arg2) * $arg3;
    return $result;
}

// función que hará la llamada de retorno
function forma_2() {
    $num_args = func_num_args();
    $args = func_get_args();
    $result = NULL;

    // verifico que al menos se reciba 1 argumento
    if($num_args >= 1) {
        // obtengo el nombre de la función (asumo que es el 1er arg.)
        $function = func_get_arg(0);

        // elimino el nombre de la función de los argumentos
        array_shift($args); // elimino el índice 0

        // verifico que sea una función callable y la llamo
        if(is_callable($function)) {
            $result = call_user_func_array($function, $args);
        }
    }

    return $result;
}

// implementación
$function = 'callable_func_1';

$args1 = 10;
$args2 = 5;
$args3 = 2;

$resultado = forma_2($function, $arg1, $arg2, $arg3);
echo $resultado;
```

Llamadas recursivas

Se denomina llamada recursiva (o recursividad), a aquellas funciones que en su algoritmo, hacen referencia sí misma.

Las llamadas recursivas suelen ser muy útiles en casos muy puntuales, pero debido a su gran factibilidad de caer en iteraciones infinitas, deben extremarse las medidas precautivas necesarias y solo utilizarse cuando sea estrictamente necesario y no exista una forma alternativa viable, que resuelva el problema, evitando la recursividad.

PHP admite las llamadas recursivas, permitiendo a una función, llamarse a sí misma, de igual forma que lo hace cuando llama a otra función.

```
function funcion_recursiva() {  
    //algoritmo...  
    funcion_recursiva();  
}
```

Veamos como funciona. En el siguiente ejemplo:

```
function funcion_recursiva($a=0) {  
    if($a == 0) {  
        $a = 1;  
        $a = funcion_recursiva($a);  
    } else {  
        $a = $a*2;  
    }  
    return $a;  
}
```

si llamo a `funcion_recursiva()` sin pasar ningún parámetro, `funcion_recursiva` tomará el valor de `$a` definido por defecto (0). Ejecutará entonces el `if`, se llamará a sí misma, y en esta segunda ejecución, actuará el `else`.

si en cambio, llamara a `funcion_recursiva(5)` pasando un entero como parámetro, se ejecutará el `else` directamente. Lo

mismo sucedería si en vez de un entero, pasara una cadena como parámetro.

Helpers

En programación, un **helper** es una **función** o **conjunto de funciones** genéricas, de uso común, destinadas a servir de ayuda a otros procesos dentro de un mismo sistema.

Un helper que retorna la fecha actual

```
/*
    Retorna la fecha actual en formato largo, corto o ISO (canónico)

    Argumentos:
    $formato -- largo, retorna la fecha actual en formato
                Lunes, 2 de Agosto de 2011

                corto, retorna la fecha en formato 02/08/2011

                ISO, retorna la fecha en formato 2011-08-02
*/
function get_fecha_actual($formato) {
    // defino un array con los patrones de formato
    $formato_fecha = array(
        "largo" => "l, j \d\e F \d\e Y",
        "corto" => "d/m/Y",
        "ISO" => "Y-m-d",
    );

    // inicializo la variable $fecha
    $fecha = NULL;

    // compruebo que $formato sea un formato válido
    if(array_key_exists($formato, $formato_fecha)) {
        // si el formato es válido, reasigno el valor a $fecha
        $fecha = date($formato_fecha[$formato]);
    }

    // retorno la fecha formateada
    return $fecha;
}
```

Un helper que modifica una variable global, haciendo una llamada de retorno

```
/*  
    Llama a la función indicada y reasigna el valor de una variable  
    global, formateado por la función indicada  
  
    Argumentos:  
    $variable -- variable global a ser modificada  
    $funcion -- función a la cual debe llamarse para dar formato a $variable  
    $argumentos -- (opcional) parámetros que eventualmente puedan  
                  ser requeridos por $funcion  
*/  
function set_variable_global(&$variable, $funcion, $argumentos=array()) {  
    // compruebo que $funcion sea una función callable  
    if(is_callable($funcion)) {  
        $variable = call_user_func_array($funcion, $argumentos);  
    }  
}
```


Taller de Funciones

En este taller, veremos como, utilizando buenas prácticas de programación, nos ayudaremos de funciones definidas por el usuario, para lograr:

- Un sistema dinámico, seguro, fácilmente mantenible y escalable
- Lograr una completa abstracción de código HTML, evitando embeberlo y/o fusionarlo con PHP

Archivos necesarios para el taller

Descarga los archivos que utilizaremos en el taller, pulsando el siguiente enlace:

<http://taller-de-php.eugeniabahit.com/taller-de-funciones.tar.gz>

Trabajando con el Sistema de Archivos

PHP dispone de un conjunto de funciones nativas, que nos permiten trabajar holgadamente todo el sistema de archivos, como si lo manejaríamos desde el propio sistema operativo.

En este capítulo, veremos las principales funciones del sistema de archivos, que nos será útiles para la mayor parte de las aplicaciones que requieran manipular tanto archivos como directorios.

Recorrido rápido por las principales funciones

De forma rápida, veremos aquí como abrir archivos, leer su contenido, manipularlo, crear archivos y escribir en ellos. Comencemos!

Apertura de archivos

Sin dudas, cuando solo se requiere abrir un archivo para leer su contenido, el modo más práctico de hacerlo es con la función **file_get_contents** vista al comienzo (y a lo largo) de este curso.

No obstante, PHP dispone de la función **fopen()** que permite, no solo abrir el archivo para leerlo, sino también, para escribir en él y manipular sus datos.

```
fopen($archivo, $modo[, $include_path]);
```

AVISO:

Los corchetes [y] indican que el parámetro es opcional.

fopen abrirá el archivo en el modo indicado y creando un **puntero** en el mismo.

***Puntero:** lugar del el archivo en el cual se coloca el cursor al ser abierto.*

Toda vez que un archivo sea abierto, debe cerrarse a fin de liberarlo de la memoria. Para ello, utilizamos **fclose(\$cursor_creado_con_fopen):**

```
$cursor = fopen('archivo.txt', 'r');  
fclose($cursor);
```

Modos de apertura

Los modos posibles de apertura, los siguientes:

Modo	Descripción	Puntero
r	Lectura	Al inicio del archivo
r+	Lectura y escritura	Al inicio del archivo
	Escritura	
w	Si e archivo no existe, intenta crearlo. Si existe, lo sobrescribe.	Al inicio del archivo
	Lectura y escritura	
w+	Si e archivo no existe, intenta crearlo. Si existe, lo sobrescribe.	Al inicio del archivo, truncándolo
	Escritura	
a	Si el archivo no existe, intenta crearlo.	Al final del archivo
a+	Lectura y escritura	
	Si e archivo no existe, intenta crearlo.	Al final del archivo
	Escritura	
x	Crea un nuevo archivo para escribir en él. Si el archivo ya existe, falla.	Al inicio del archivo
x+	Lectura y Escritura	Al inicio del archivo

Crea un **nuevo archivo** para escribir en él y luego poder leerlo. Si el archivo ya existe, falla.

- | | | |
|-----------|--|-----------------------|
| c | Escritura
Si e archivo no existe, intenta crearlo. | Al inicio del archivo |
| c+ | Lectura y escritura
Si e archivo no existe, intenta crearlo. | Al inicio del archivo |

Ruta hacia el archivo

La ruta especificada hacia el archivo, debe seguir la forma **protocolo://ruta_al_archivo**.

De esta forma, si quisiéramos abrir una URL, deberíamos utilizar: `http://www.dominio.com/archivo.txt`

Pero si quisiéramos **abrirlo localmente**, deberíamos indicar: `/ruta_a_mi_dominio/public_html/archivo.txt`

ADVERTENCIA

Nótese que en **Windows** deberá utilizarse el siguiente formato: `c:\\ruta_a\\archivo.txt`

Utilizar o no `include_path`

El tercer parámetro (opcional), permite indicar si se desea buscar el archivo en el `include_path` seteado en el archivo `php.ini`. En dicho caso, debe pasarse `TRUE` (o `1`):

```
$file = fopen('file.txt', 'r', TRUE);
```

Generalmente, no utilizaremos este parámetro, a no ser que sea extrictamente necesario.

Lectura de Archivos

Una vez abierto un archivo, podremos leer su contenido

utilizando la función de lectura en modo binario seguro, **fread()**.

Utilizando la siguiente sintaxis:

```
fread($recurso, $bytes);
```

podremos **leer el contenido de un archivo en modo binario seguro**, necesitando de un recurso (obtenido mediante **fopen**) e indicando la cantidad de bytes a leer (1 carácter = 1 byte).

Para leer el contenido completo del archivo, podemos ayudarnos de la función **fsize(\$archivo)**, donde **\$archivo**, será la ruta completa al archivo que se quiere leer:

```
$archivo = "archivo.txt"; // nombre del archivo
$bytes = filesize($archivo); // tamaño del archivo
$cursor = fopen($archivo, "r"); // abrir archivo
$contenido = fread($cursor, $bytes); // leer contenido
fclose($cursor); // cerrar el cursor (liberar memoria)
```

Escribir en un archivo

Para escribir en un archivo, nos valemos de la función **fwrite()** la cuál escribirá en modo binario seguro.

Su sintaxis es la siguiente:

```
fwrite($recurso, $contenido_a_escribir[, $cantidad_de_bytes_a_escribir]);
```

El contenido, puede ser cualquier variable de tipo **string**, mientras que la cantidad de bytes a escribir, es opcional. **Si se indica la cantidad de bytes**, se dejará de escribir cuando la cantidad de bytes se haya alcanzado o cuando la cadena termine (lo que suceda primero).

```
$archivo = "archivo.txt";
$recurso = fopen($archivo, "a+");
$nuevo_contenido = "nuevo contenido";
fwrite($recurso, $nuevo_contenido);
$bytes = filesize($archivo);
$contenido = fread($recurso, $bytes);
fclose($recurso);
```

Moviendo el puntero dentro del archivo

Cuando tenemos que escribir un archivo, es muy útil saber en qué lugar se encuentra el puntero, y moverlo a la posición indicada.

Podemos obtener la posición actual del puntero, con la función **ftell(\$recurso)** y movernos hacia el byte indicado, con **fseek(\$recurso, \$byte)**.

Un contador de visitas sencillo

```
function contador_de_visitas() {
    $archivo = "contador.txt";
    $recurso = fopen($archivo, "r+");
    $bytes_totales = filesize($archivo);
    $contador = fread($recurso, $bytes_totales);
    $nuevo_contenido = $contador + 1;
    $posicion_actual = ftell($recurso);

    if($posicion_actual == $bytes_totales) {
        // me muevo al byte 0 para sobrescribir el archivo
        fseek($recurso, 0);
    }

    fwrite($recurso, $nuevo_contenido);
    fclose($recurso);

    return $nuevo_contenido;
}

// Actualizar el número de visitas y mostrarlo
echo contador_de_visitas();
```

¡Cuidado con los permisos!

Como es lógico de esperar, para poder crear un archivo o escribir sobre un archivo existente, éste, debe tener permisos de escritura para el usuario www-data.

En un servidor Web, tener archivos o directorios servidos con permisos de escritura, es una puerta que se está abriendo hasta para el más novato de los delincuentes informáticos que transitan por la red.

La **mejor alternativa**, es tener un directorio **NO SERVIDO** con permisos de escritura (es decir, un directorio con permisos de escritura, fuera del directorio de publicación Web).

En este caso, bastará con utilizar como ruta del archivo, la ruta absoluta.

Trabajando con directorios

Como hemos comentado antes, PHP permite trabajar el sistema de archivos, como podríamos hacerlo desde las aplicaciones del propio sistema operativo. Y esto, incluye también, funciones relacionadas a los directorios.

Creando el gestor

Al igual que con los archivos, para acceder a un directorio, debe crearse primero un recurso (gestor de directorio). Para ello, al igual que fopen abre un archivo, tenemos una función para abrir los directorios.

Tal vez, tomando como base lógica el nombre de la función fopen (que proviene de ***FileOpen***), estés esperando una función llamada *dopen*, pero lamentablemente, no existe una función llamada *dopen*, ya que PHP, no tiene estandarizado el estilo para nombres de funciones. A diferencia de lo que esperamos, para abrir un directorio, la función que debemos utilizar, se denomina **opendir**.

```
$recurso = opendir('nombre_del_directorio');
```

Como nombre de directorio, es posible utilizar también, cualquier **ruta absoluta**:

```
$recurso = opendir('/var/www/ dominio.com/public_html/archivos/pdf');
```

o una **ruta relativa**:

```
$recurso = opendir('../archivos/pdf');  
$otro_recurso = opendir('archivos/pdf');
```

Al igual que cuando abrimos un archivo, cuando abrimos un directorio, es necesario cerrarlo para liberarlo de memoria:

```
closedir($recurso);
```

Explorando el contenido de un directorio

Explorar el contenido de un directorio, es sumamente sencillo, ya que disponemos de una función para hacerlo: **readdir(\$recurso)**. Sin embargo, la exploración de directorios puede ser compleja, debido a que:

- **readdir** no devuelve el contenido completo de un directorio en su primera ejecución, sino que va leyendo cada elemento de a uno por vez y por lo tanto, **readdir** debe ejecutarse iterativamente;
- **readdir** retorna el nombre del elemento (archivo o directorio) pero en caso de error, puede devolver tanto **False** como un valor no booleano que pueda ser evaluado como **False**, por lo cual, antes de ejecutar alguna acción, debe verificarse el retorno.
- En sistemas basados en UNIX, todo directorio contiene a

la vez dos subdirectorios ocultos cuyos nombres son `.` (punto) y `..` (doble punto), que deben ser validados previamente a fin de evitar listarlos.

```
// abro el directorio
$dir = opendir('../taller-de-funciones');

// itero solo si readdir NO devuelve False
while(($selemento = readdir($dir)) !== False) {
    // imprimo el nombre del archivo o directorio
    echo $selemento . chr(10);
}

// cierro el directorio
closedir($dir);

/*
    Salida:

    index.php
    template.html
    funciones.php
    ..
    files
*/
```

Nótese que en el ejemplo anterior, se están listando los dos directorios ocultos típicos de todo sistema UNIX-Like. Para evitar eso, será necesario filtrarlos:

```
// abro el directorio
$dir = opendir('../taller-de-funciones');

// inicializo un array donde guardaré cada elemento
$contenido = array();

// itero solo si readdir NO devuelve False
while(($selemento = readdir($dir)) !== False) {
    // evito que liste los directorios ocultos . y ..
    if($selemento != "." and $selemento != "..") {
        // agrego cada elemento en el array $contenido
        $contenido[] = $selemento;
    }
}

// cierro el directorio
closedir($dir);
```

```
// imprimo la salida
print_r($contenido);

/* Salida:

    Array
    (
        [0] => index.php
        [1] => template.html
        [2] => funciones.php
        [3] => files
    )
*/
```

Filtrando el tipo de elemento

Como vimos en el ejemplo anterior, `readdir` retorna tanto archivos como directorios. Es posible filtrar el tipo de elemento, para poder manipularlos de forma más apropiada. Para ello disponemos de cuatro funciones muy útiles:

`is_dir($elemento)`

Nos indica si el elemento evaluado es un directorio (True) o no (False)

`is_file($elemento)`

Nos indica si el elemento evaluado es un archivo (True) o no (False).

`is_link($elemento)`

Nos indica si el elemento evaluado es un elace simbólico (True) o no (False).

Nótese que en Windows, los enlaces simbólicos son denominados “accesos directos”.

filetype(\$elemento)

Nos retorna el tipo de elemento siendo los valores de retorno posibles: fifo, char, dir, block, link, file, socket y unknown.

ADVERTENCIA

Nótese que filetype podrá devolver False si no pudo ejecutarse con éxito pero también podría devolver un error, si el tipo de archivo es desconocido.

```
$dir = opendir('../taller-de-funciones');
$archivos = array();
$directorios = array();
$symlinks = array();

while(($elemento = readdir($dir)) !== False) {
    if($elemento != "." and $elemento != "..") {
        $path_elemento = "../taller-de-funciones/{$elemento}";
        if(is_dir($path_elemento)) {
            $directorios[] = $elemento;
        } elseif(is_file($path_elemento)) {
            $archivos[] = $elemento;
        } elseif(is_link($path_elemento)) {
            $symlinks[] = $elemento;
        }
    }
}

closedir($dir);

$contenido = array('Directorios' => $directorios,
                  'Archivos' => $archivos,
                  'Enlaces simbólicos' => $symlinks);

print_r($contenido);
```

Lo anterior, producirá la siguiente salida:

```
eugenia@cocochito:~/borradores$ php -f file.php
Array
(
```

```
[Directorios] => Array
(
    [0] => files
)

[Archivos] => Array
(
    [0] => index.php
    [1] => template.html
    [2] => funciones.php
)

[Enlaces simbólicos] => Array
(
)

)
```

Nótese que alternativamente a las tres funciones utilizadas en el ejemplo (`is_file`, `is_dir` e `is_link`) se podría comprobar mediante `filetype($path_elemento)`. Sin embargo, la forma segura de chequear el tipo de elemento, es con las funciones usadas en el código anterior.

Otras funciones que necesitarás con frecuencia

Muchas veces, será necesario saber si el directorio o archivos que intentamos abrir, existe, conocer si puede ser leído y/o escrito.

Estas acciones serán muy frecuentes, y para resolver el dilema, disponemos de las funciones necesarias.

Comprobar la existencia de un archivo o directorio

```
file_exists('archivo_o_directorio')
```

Comprueba si un archivo o directorio existe (True) o no (False):

```
// validando si un archivo existe
$archivo = 'ruta_a/mi_archivo.txt';
if(file_exists($archivo)) {
    echo "El archivo {$archivo} existe";
} else {
    echo "El archivo {$archivo} no pudo localizarse";
}

// ahora, verificando si un directorio existe
$directorio = 'ruta/a/mi/carpeta';
if(file_exists($directorio)) {
    echo "El directorio {$directorio} existe";
} else {
    echo "El directorio {$directorio} no pudo localizarse";
}
```

Comprobar si un archivo o directorio es legible

```
is_readable('archivo_o_directorio')
```

Comprueba si un archivo o directorio es legible (True) o no (False):

```
// validando si un archivo es legible
$archivo = 'ruta_a/mi_archivo.txt';
if(is_readable($archivo)) {
    echo "El archivo {$archivo} puede ser leído";
} else {
    echo "El archivo {$archivo} no puede ser leído";
}

// ahora, verificando si un directorio es legible
$directorio = 'ruta/a/mi/carpeta';
if(is_readable($directorio)) {
    echo "El directorio {$directorio} puede ser leído";
} else {
    echo "El directorio {$directorio} no puede ser leído";
}
```

Comprobar si un archivo o directorio puede escribirse

```
is_writable('archivo_o_directorio')
```

Comprueba si un archivo o directorio es legible (True) o no (False):

```
// validando si un archivo puede escribirse
$archivo = 'ruta_a/mi_archivo.txt';
if(is_writable($archivo)) {
    echo "El archivo {$archivo} puede ser escrito";
} else {
    echo "El archivo {$archivo} no puede ser escrito";
}

// ahora, verificando si un directorio puede escribirse
$directorio = 'ruta/a/mi/carpeta';
if(is_writable($directorio)) {
    echo "El directorio {$directorio} puede ser escrito";
} else {
    echo "El directorio {$directorio} no puede ser escrito";
}
```

Más funciones sobre el sistema de archivos

Más funciones sobre el sistema de archivos, pueden encontrarse en

<http://www.php.net/manual/es/ref.filesystem.php>

Procesamiento de texto y manipulación de strings

En este capítulo, veremos las principales funciones y formas básicas, de procesar texto y manipular strings en PHP, las cuales serán la base fundamental de las expresiones y algoritmos integrantes de todo sitio Web y aplicación.

Ampliando la definición de variables de tipo string

Como hemos visto anteriormente, una cadena de texto en PHP, puede encerrarse tanto entre **comillas simples** como entre **comillas dobles**:

```
$var_1 = 'Esta es una cadena de texto';  
$var_2 = "Esta es otra cadena de texto";
```

También disponemos de la posibilidad de delimitar cadenas de texto de gran extensión, mediante **heredoc** (del inglés “here document” - documento aquí), típica de una gran parte de lenguajes de programación y shells de Sistemas Operativos basados UNIX, cuya sintaxis es:

```
$variable = <<<IDENTIFICADOR  
contenido de heredoc  
IDENTIFICADOR;
```

Donde IDENTIFICADOR, podrá ser cualquier nombre que respete las reglas para nombres de variables y el identificador de cierre, no podrá estar sangrado.

Veamos un ejemplo:

```
$documento = <<<NOTA_SOBRE_HEREDOC
```

Es muy importante señalar que la línea con el identificador de cierre no debe contener ningún carácter, excepto posiblemente un punto y coma (;). Esto significa en particular que el identificador no debe usar sangría, y que no deben existir ningún espacio ni tabulación antes o después del punto y coma. Es muy importante darse cuenta que el primer carácter antes del identificador de cierre debe ser un salto de línea definida por el sistema operativo local. En los sistemas UNIX sería `\n`, al igual que en Mac OS X. El delimitador de cierre (posiblemente seguido de un punto y coma) también debe ser seguido de un salto de línea.

Si se rompe esta regla y el identificador de cierre no está "limpio", no será considerado como un identificador de cierre, y PHP continuará buscando uno. Si no se encuentra ningún identificador de cierre antes del final del fichero, se producirá un error de análisis en la última línea.

```
NOTA_SOBRE_HEREDOC;
```

Se recomienda el uso de **heredoc**, para definir cadenas de texto de grandes extensiones.

Escapando caracteres

Muchas veces es necesario imprimir ciertos caracteres que no pueden simplemente indicarse. Un ejemplo de ello, es cuando en una cadena de texto delimitada por comillas dobles, se desea imprimir el literal de comillas dobles.

Hacer esto:

```
$var = "Las comillas dobles (") deben escaparse";
```

generará un error, puesto que PHP considerará el final de la cadena de texto en la segunda comilla doble:

```
$var = "Las comillas dobles ("
```

y encontrará un error de sintaxis a continuación:

```
) deben escaparse";
```

Para solucionar este problema, ciertos caracteres deben **escaparse, mediante** el uso de una **barra diagonal invertida **

```
$var = "Las comillas dobles (\") deben escaparse";
```

Caracteres de escape

Algunos caracteres de escape pueden representarse como se muestra en la siguiente table:

Caracter	Significado
<code>\n</code>	avance de línea (LF o 0x0A (10) en ASCII)
<code>\r</code>	retorno de carro (CR o 0x0D (13) en ASCII)
<code>\t</code>	tabulador horizontal (HT o 0x09 (9) en ASCII)
<code>\v</code>	tabulador vertical (VT o 0x0B (11) en ASCII) (desde PHP 5.2.5)
<code>\e</code>	escape (ESC o 0x1B (27) en ASCII) (desde PHP 5.4.0)
<code>\f</code>	avance de página (FF o 0x0C (12) en ASCII) (desde PHP 5.2.5)
<code>\\</code>	barra invertida
<code>\\$</code>	signo del dólar
<code>\"</code>	comillas dobles
<code>\'</code>	Comilla simple
	Imprimir las llaves alrededor del contenido de \$var
<code>\{\$var}</code>	<pre>\$a = "Hola Mundo"; \$b = "Yo digo \{\$a}"; echo \$b; // salida: Yo digo {Hola Mundo}</pre>

Funciones para manipulación de strings

A continuación, se mostrarán las funciones de uso más frecuente para la manipulación de cadenas de texto. Para obtener un listado completo de funciones de string, visita el manual oficial en <http://www.php.net/manual/es/ref.strings.php>

Funciones de escape

addslashes(\$cadena) escapa una cadena de texto añadiendo barras invertidas a las comillas dobles, simples, barras invertidas y bytes nulos.

escapar cadenas de texto que deban insertarse en bases de datos, y hayan sido recibidas mediante HTTP POST.

quotemeta(\$cadena) escapa una cadena de texto añadiendo barras invertidas a los siguientes caracteres: . \ + * ? [^] (\$)

```
$doc = "Si se realiza el cálculo (15*2)+[(12+5)*(4.3+0.45)] obtendremos el
importe en $";
$doc = quotemeta($doc);
echo $doc;
/*
Salida:
Si se realiza el clculo \(15\*2\) \+ \[(12\+5\) \*\ (4\.3\+0\.45\) \] obtendremos
el importe en \$
*/
```

De forma inversa a lo anterior, pueden eliminarse las barras invertidas de una cadena espada, mediante **stripslashes(\$cadena)**

```
$doc = "Si se realiza el cálculo (15*2)+[(12+5)*(4.3+0.45)] obtendremos el
importe en $";
$doc = stripslashes($doc);
```

```

echo $doc;
/*
Salida:
Si se realiza el clculo \((15*2)\)\+[\((12+5)\)*\((4\3\+0\45)\)\] obtendremos
el importe en \$
*/

echo stripslashes($doc);
/*
Salida:
Si se realiza el cálculo (15*2)+[(12+5)*(4.3+0.45)] obtendremos el importe en
$
*/

```

Funciones de conversión

htmlentities(\$cadena) convierte los caracteres aplicables a entidades HTML.

Esta función debe utilizarse siempre que una cadena de texto deba ser impresa en un documento HTML y se desconozca su contenido, para prevenir que código fuente no deseado, sea ejecutado.

```

$cadena = "Las negritas se escriben entre los tags <b> y </b> mientras que el
salto de linea se representa con <br/>";
$cadena = htmlentities($cadena);
echo $cadena;
/*
Las negritas se escriben entre los tags &lt;b&gt; y &lt;/b&gt; mientras que
el salto de linea se representa con &lt;br/&gt;
*/

```

Su opuesto es **html_entity_decode(\$cadena)**

```

$cadena = "Las negritas se escriben entre los tags &lt;b&gt; y &lt;/b&gt;
mientras que el salto de linea se representa con &lt;br/&gt;";
$cadena = html_entity_decode($cadena);
echo $cadena;
/*
Las negritas se escriben entre los tags <b> y </b> mientras que el salto de

```

```
línea se representa con <br/>
*/
```

Cuando solo se deseen convertir a entidades HTML, caracteres especiales tales como & " ' < >, se utilizará la función **htmlspecialchars(\$cadena)** siendo el opuesto de esta última, la función **htmlspecialchars_decode(\$cadena)**.

Evitando ejecución de código no deseado

Una función que deberá utilizarse toda vez que quiera evitarse la ejecución de código PHP y HTML no deseado, es **strip_tags(\$cadena, \$caracteres_permitidos)**. Esta función **eliminará todas las etiquetas PHP y HTML** exceptuando aquellas que se indiquen como caracteres permitidos:

```
$caracteres_permitidos = "<b>";
$cadena = "<p>Hola <b>Mundo</b></p><script
language='javascript'>alert('hola');</script>
<?php
echo $caracteres_permitidos;
?>";
$resultado = strip_tags($cadena, $caracteres_permitidos);
echo $resultado;
// salida: Hola <b>Mundo</b>alert('hola');
```

Eliminar espacios en blanco, también es posible. Disponemos de tres funciones predefinidas:

ltrim(\$cadena): Elimina los espacios en blanco del inicio de la cadena

rtrim(\$cadena): los elimina del final de la cadena

trim(cadena): los elimina del inicio y final de la cadena

Funciones de formato

La función **nl2br(\$cadena)** nos permite **convertir saltos de**

línea en su representación HTML (
):

```
$cadena = "Esto es
una cadena
de texto";
$resultado = nl2br($cadena);
echo $resultado;
/*
    salida:
    Esto es<br />
    una cadena <br />
    de texto
*/
$cadena = "Esto es\nuna cadena\nde texto";
$resultado = nl2br($cadena);
echo $resultado;
/*
    salida:
    Esto es<br />
    una cadena <br />
    de texto
*/
```

Podemos además, **ajustar el ancho de caracteres** de una cadena de texto, utilizando la función **wordwrap(\$cadena, \$ancho, \$salto_de_linea, \$no_cortar_palabras)**.

Esta función, recibirá 1 parámetro obligatorio (\$cadena) y tres parámetros opcionales:

\$ancho cantidad de caracteres

\$salto_de_linea el carácter o patrón que se utilizará para crear el salto de línea. Ejemplo: \n o
.

\$no_cortar_palabras si se establece en TRUE, PHP tendrá cuidado de insertar el salto, sin cortar palabras.

```
$texto = "Lorem ipsum ad his scripta blandit partiendo, eum fastidii accumsan
euripidis in, eum liber hendrerit an. Qui ut wisi vocibus suscipiantur, quo
dicit ridens inciderint id. Quo mundi lobortis reformidans eu, legimus
senserit definiebas an eos. Eu sit tincidunt incorrupte definitionem, vis
mutat affert percipit cu, eirmod consectetur signiferumque eu per. In usu
latine equidem dolores. Quo no falli viris intellegam, ut fugit veritus
placerat per.";

$formato = wordwrap($texto, 60, chr(10), True);

echo $formato;
```

```
/*  
Lorem ipsum ad his scripta blandit partiendo, eum fastidii  
accumsan euripidis in, eum liber hendrerit an. Qui ut wisi  
vocibus suscipiantur, quo dicit ridens inciderint id. Quo  
mundi lobortis reformidans eu, legimus senserit definiebas  
an eos. Eu sit tincidunt incorrupte definitionem, vis mutat  
affert percipit cu, eirmod consectetur signiferumque eu  
per. In usu latine equidem dolores. Quo no falli viris  
intellegam, ut fugit veritus placerat per.  
*/
```

Podemos necesitar **convertir toda una cadena a minúsculas** con `strtolower($cadena)`, o solo **convertir a minúscula, el primer carácter** de una cadena con `lcfirst($cadena)`:

```
$usuario = "AnGgie";  
echo strtolower($usuario); // anggie  
echo lcfirst($usuario); // anGgie
```

Pero también podemos querer **convertir toda una cadena a mayúsculas** con `strtoupper($cadena)`, **convertir solo el primer carácter de la cadena** `ucfirst($cadena)` o **convertir el primer carácter de cada palabra** `ucwords($cadena)`:

```
$cadena = "hola mundo";  
echo strtoupper($cadena); // HOLA MUNDO  
echo ucfirst($cadena); // Hola mundo  
echo ucwords($cadena); // Hola Mundo
```

Un ejemplo práctico de conversión de mayúsculas y minúsculas:

```
$nombre_y_apellido = "Anggie Lopez";  
$username = strtolower($nombre_y_apellido);  
$nombre_visible = ucwords($username);
```

Dar a una cadena, **formato de moneda**, es posible mediante

el uso de `money_format($formato, $cadena):`

```
setlocale(LC_MONETARY, "es_ES.UTF-8");

$bruto = 178.45;
$iva = $bruto * 0.21;
$neto = $bruto * 1.21;

$bruto_txt = money_format('%(#4n', $bruto);
$iva_txt = money_format('%(#4n', $iva);
$neto_txt = money_format('%(#4n', $neto);

echo $bruto_txt . chr(10);
echo $iva_txt . chr(10);
echo $neto_txt . chr(10);

/*
    178,45 €
    37,47 €
    215,92 €
*/
```

AVISO

utilizar `setlocale` antes de dar formato de moneda, asegura la correcta salida de los datos con el símbolo monetario correspondiente al idioma y país.

Para comprender mejor los posibles patrones de formato que pueden ser utilizados con `money_format`, acceder a las referencias oficiales en

<http://www.php.net/manual/es/function.money-format.php#refsect1-function.money-format-parameters>

A veces es preciso formatear un valor numérico, estableciendo decimales y separadores de decimales y miles. Contamos para ello con la función `number_format($numero, $decimales, $separador_decimales, $separador_miles)` que retorna el número formateado como cadena de texto:

```
$precio = 12478.493;
$precio_txt = number_format($precio, 2, ',', '.');
```

```
echo $precio_txt; // 12.478,49
```

Funciones de manipulación

Muchas veces, puede ser muy útil, manipular una cadena de texto, de forma tal, que nos permite operar con diferentes datos. Por ejemplo, es posible **dividir una cadena de texto**, tomando como punto de división, un caracter o patrón, mediante la función **explode(\$delimitador, \$cadena)** y así obtener un array con las fracciones de cadena divididas, que nos permita iterar sobre cada una:

```
$contactos = "Juan Antonio Avila <avila@mail.com>,  
Rodrigo Mancusso <rmancu@mail.com>,  
Silvina D'laggio <dlaggio@mail.com>  
";  
  
$patron = "," . chr(10);  
  
$personas = explode($patron, $contactos);  
  
foreach($personas as $persona) {  
    echo $persona . chr(10);  
}  
  
/*  
Juan Antonio Avila <avila@mail.com>  
Rodrigo Mancusso <rmancu@mail.com>  
Silvina D'laggio <dlaggio@mail.com>  
*/
```

Podemos **contar la cantidad de caracteres** de una cadena de texto, mediante la función **strlen(\$cadena)**:

```
$mensaje = "Lorem ipsum ad his scripta blandit partiendo, eum fastidii  
accumsan euripidis in, eum liber hendrerit an. Qui ut wisi vocibus  
suscipiantur, quo dicit ridens inciderint id. Quo mundi lobortis reformidans  
eu, legimus senserit definiebas an eos. Eu sit tincidunt incorrupte  
definitionem, vis mutat affert percipit cu, eirmod consectetuer signiferumque  
eu per. In usu latine equidem dolores. Quo no falli viris intellegam, ut  
fugit veritus placerat per. Ius id vidit volumus mandamus, vide veritus  
democritum te nec, ei eos debet libris consulatu. No mei ferri graeco dicunt,  
ad cum veri accommodare. Sed at malis omnesque delicata, usu et iusto zrril  
meliore. Dicunt maiorum eloquentiam cum cu, sit summo dolor essent te. Ne  
quodsi nusquam legendos has, ea dicit voluptua eloquentiam pro, ad sit quas
```



```

qualisque. Eos vocibus deserunt quaestio ei. ";

$caracteres = strlen($mensaje);

if($caracteres > 140) {
    echo "Tu mensaje es demasiado largo. Solo se admiten 140 caracteres.";
}

```

Contar la cantidad de palabras en una cadena de texto, e incluso, iterar sobre cada palabra, puede ser algo realmente útil. La función **str_word_count(\$cadena, \$formato)** nos ayudará a hacerlo:

```

$nombre_y_apellido = "Juan P.";
$datos = str_word_count($nombre_y_apellido, 1);

if(count($datos) < 2) {
    echo "{$nombre_y_apellido} no es un nombre y apellido válido";
} else {
    foreach($datos as $dato) {
        if(strlen($dato) < 2) {
            echo "Por favor, no utilices iniciales.";
        }
    }
}

```

Otra función que podremos utilizar muy a menudo, es **str_replace(\$busqueda, \$reemplazo, \$cadena)** que nos permite **buscar un determinado carácter o patrón y reemplazarlo por el indicado**:

```

$email = "juanperez@dominio.com";
$mail_no_spam = str_replace("@", " [AT] ", $email);
echo $mail_no_spam; // juanperez [AT] dominio.com

```

Esta función, admite como parámetros de búsqueda y reemplazo, tanto cadenas de texto, como matrices:

```

$email = "juanperez@dominio.com";
$busqueda = array("@", ".");
$reemplazo = array(" [AT] ", " [DOT] ");
$mail_no_spam = str_replace($busqueda, $reemplazo, $email);
echo $mail_no_spam; // juanperez [AT] dominio [DOT] com

```

Incluso, permite reemplazar todos los elementos de un array de búsqueda, por un único carácter o patrón de reemplazo (muy útil para eliminar espacios en blanco en una cadena, como en el siguiente ejemplo):

```
$username = "    alejo val3nt1n0 ";
$busqueda = array(" ", "\t", "\n", "\r", "\0", "\x0B");
$username = str_replace($busqueda, '', $username);
echo $username; // alejoval3nt1n0
```

Es posible también, realizar reemplazos, haciendo que la búsqueda sea insensible a mayúsculas y minúsculas. Para ello, debemos utilizar la función **str_ireplace(\$busqueda, \$reemplazo, \$cadena)** de la misma forma que lo haríamos con **str_replace()**.

Otra función sumamente útil, es **strpos(\$cadena, \$patron_de_busqueda)**, la cual nos **retornará la posición en la que se encuentra el patrón buscado, dentro de la cadena**:

```
$email = "juanperez@mail.com";
$patron = "@";
$posicion = strpos($email, $patron);
echo $posicion; // 9
```

Si se desea que la búsqueda sea insensible a mayúsculas y minúsculas, deberá utilizarse **stripos(\$cadena, \$patron)**.

Es importante tener en cuenta, que tanto **strpos** como **stripos**, retornarán **False** cuando el patrón de búsqueda no sea encontrado. Por lo tanto, toda condición debe ser comparada por exactitud de valor y tipo de dato:

```
$var1 = "Hola Mundo";
$var2 = "adios mundo";
$patron = "hola";
```

```
if(strpos($var1, $patron) === 0) {
    echo "Está al comienzo de la cadena" . chr(10);
}

# INCORRECTO
if(strpos($var2, $patron) == 0) {
    echo "Está al comienzo de la cadena" . chr(10);
}

if(strpos($var2, $patron) === False) {
    echo "No se encontró" . chr(10);
}
```

Manipulando subcadenas en cadenas

Hay tres funciones muy útiles que nos permiten manipular subcadenas de texto dentro de una cadena.

La función **substr(\$cadena, \$inicio, \$longitud)** nos retornará la longitud de la cadena desde el inicio indicado:

```
$cadena = "Lorem ipsum ad his scripta blandit partiendo, eum fastidii
accumsan euripidis in, eum liber hendrerit an. Qui ut wisi vocibus
suscipiantur, quo dicit ridens inciderint id. Quo mundi lobortis reformidans
eu, legimus senserit definiebas an eos. Eu sit tincidunt incorrupte
definitionem, vis mutat affert percipit cu, eirmod consectetuer signiferumque
eu per. In usu latine equidem dolores. Quo no falli viris intellegam, ut
fugit veritus placerat per.";

$resumen = substr($cadena, 0, 100);

echo "{$resumen}[...]";

/*
Lorem ipsum ad his scripta blandit partiendo, eum fastidii accumsan euripidis
in, eum liber hendrerit[...]
*/
```

Un ejemplo un poco más complejo, puede darse con el uso combinado de varias funciones:

```
$patron = "dicit";
$inicio_patron = strpos($cadena, $patron);
```

```
if($inicio_patron != False) {  
    echo substr($cadena, $inicio_patron, strlen($cadena));  
}  
  
/*  
dicit ridens inciderint id. Quo mundi lobortis reformidans eu, legimus  
senserit definiebas an eos. Eu sit tincidunt incorrupte definitionem, vis  
mutat affert percipit cu, eirmod consectetur signiferumque eu per. In usu  
latine equidem dolores. Quo no falli viris intellegam, ut fugit veritus  
placerat per.  
*/
```

Con **substr_count(\$cadena, \$patron)** podremos obtener la cantidad de veces que el patrón es encontrado en la cadena:

```
$cadena = "Lorem ipsum ad his scripta blandit partiendo, eum fastidii  
accumsan euripidis in, eum liber hendrerit an. Qui ut wisi vocibus  
suscipiantur, quo dicit ridens inciderint id. Quo mundi lobortis reformidans  
eu, legimus senserit definiebas an eos. Eu sit tincidunt incorrupte  
definitionem, vis mutat affert percipit cu, eirmod consectetur signiferumque  
eu per. In usu latine equidem dolores. Quo no falli viris intellegam, ut  
fugit veritus placerat per.";  
  
$patron = "ut";  
$apariciones = substr_count($cadena, $patron);  
echo $apariciones; // 3
```

Es posible también, reemplazar una subcadena dentro de una cadena con **substr_replace(\$cadena, \$reemplazo, \$inicio, \$longitud)**:

```
$cadena = "Ayer recorrí las calles de Liniers con mi primo";  
  
$patron = "las calles de Liniers";  
$ini = strpos($cadena, $patron);  
  
$nueva_cadena = substr_replace($cadena, "los alrededores de Belgrano",  
                                $ini, strlen($patron));  
  
echo $nueva_cadena;  
  
// Ayer recorrí los alrededores de Belgrano con mi primo
```

Funciones de encriptación

Podemos obtener el **hash MD5** con la función **md5(\$cadena)**:

```
$clave = "pepe_grillo-12_14";  
$hash_clave = md5($clave);  
echo $hash_clave; // 917f2e75f261ba6df7b36a80e1f38241
```

ADVERTENCIA

Nunca utilices conversores MD5 online. Estos conversores, suelen almacenar las cadenas ingresadas asociadas al hash MD5 resultante, lo cual directa o indirectamente, permite realizar una pseudo ingeniería inversa sobre los hashes MD5, haciendo vulnerables las contraseñas. **Siempre que necesites obtener el hash MD5 de alguna cadena, utiliza PHP-CLI:**

```
php -r 'echo md5("cadena a hashear");'
```

PHP dispone de otras funciones de cifrado, para varios algoritmos como SHA1 y CRC32. Sin embargo y a pesar de contar con la función de cifrado MD5, disponemos de una función que engloba todo lo anterior, llamada **hash(\$algoritmo_cifrado, \$cadena)** que nos facilita **cifrar una cadena en varios formatos**.

Los **posibles algoritmos de cifrado**, pueden obtenerse con la función **hash_algos()** la cual retorna un array con todos los algoritmos disponibles:

```
php > print_r(hash_algos());  
Array  
(  
    [0] => md2  
    [1] => md4  
    [2] => md5  
    [3] => sha1  
    [4] => sha224  
    [5] => sha256
```

```
[6] => sha384
[7] => sha512
[8] => ripemd128
[9] => ripemd160
[10] => ripemd256
[11] => ripemd320
[12] => whirlpool
[13] => tiger128,3
[14] => tiger160,3
[15] => tiger192,3
[16] => tiger128,4
[17] => tiger160,4
[18] => tiger192,4
[19] => snefru
[20] => snefru256
[21] => gost
[22] => adler32
[23] => crc32
[24] => crc32b
[25] => salsa10
[26] => salsa20
[27] => haval128,3
[28] => haval160,3
[29] => haval192,3
[30] => haval224,3
[31] => haval256,3
[32] => haval128,4
[33] => haval160,4
[34] => haval192,4
[35] => haval224,4
[36] => haval256,4
[37] => haval128,5
[38] => haval160,5
[39] => haval192,5
[40] => haval224,5
[41] => haval256,5
```

)

Ejemplo (correr este script para ver los resultados):

```
$clave = "tRxc6348-bR129";
$shashes = array();
foreach(hash_algos() as $hash) {
    $hash_clave = hash($hash, $clave);
    $shashes[$hash] = $hash_clave;
}
```

Resumen de las principales funciones de string

Tipo de función	Función	Descripción
Escape	addslashes	Escapa una cadena añadiendo barras invertidas a " ' \ y bytes nulos
	quotemeta	Añade barras invertidas delante de . \ + * ? [^] (\$)
	stripslashes	Elimina las barras invertidas de una cadena escapada
Conversión	htmlentities	Convierte los caracteres a entidades HTML
	html_entity_decode	Inversa a htmlentities
	htmlspecialchars	Convierte a entidades HTML los siguientes caracteres: & " ' < >
	htmlspecialchars_decode	Inversa a htmlspecialchars
	strip_tags	Elimina todos los tags HTML y PHP
	ltrim	Elimina espacios en blanco del comienzo de la cadena
	rtrim	Elimina espacios en blanco del final de la cadena
	trim	Elimina espacios en blanco del comienzo y final de la cadena
Formato	nl2br	Convierte saltos de línea en su equivalente HTML
	wordwrap	Ajusta el ancho de caracteres de una cadena
	strtolower	Convierte toda la cadena a minúsculas
	lcfirst	Convierte a minúscula el primer carácter de una cadena
	strtoupper	Convierte toda la cadena a mayúsculas
	ucfirst	Convierte el primer carácter de una cadena a mayúscula
	ucwords	Convierte el primer carácter de cada palabra de una cadena a mayúsculas
	money_format	Formatea un número con el símbolo de moneda correspondiente
	number_format	Formatea un número con el separador de miles y decimales correspondiente

Manipulación	Explode	Divide una cadena generando un array
	strlen	Retorna la longitud de una cadena
	str_word_count	Cuenta la cantidad de palabras
	str_replace str_ireplace	Reemplaza iterativamente un patrón
	strpos stripos	Retorna la posición del patrón buscado en una cadena
	substr	Retorna una porción de la cadena
	substr_count	Retorna la cantidad de apariciones de un patrón en la cadena
	substr_replace substr_ireplace	Reemplaza iterativamente una porción de la cadena
Cifrado	md5	Retorna el hash MD5 de una cadena
	hash	Retorna el hash de una cadena, cifrado con el algoritmo indicado

Taller de Archivos y Procesamiento de Formularios

En este taller, crearemos un libro de visitas basado en el sistema de archivos (sin bases de datos) y utilizaremos a la vez, las librería de funciones de string, para el procesamiento de los textos.

Archivos necesarios para el taller

Descarga los archivos que utilizaremos en el taller, desde el siguiente enlace:

<http://taller-de-php.eugeniabahit.com/taller-de-archivos-y-webforms.tar.gz>

Constantes, variables variables y variables superglobales

Durante el taller, hemos introducido cuatro nuevos conceptos, de los cuales, trataremos aquí, dos de ellos: **constantes** y **variables variables**.

Para el procesamiento del formulario, también hemos utilizado la **variable superglobal** `$_POST` de PHP y hablaremos de ello.

Constantes

PHP, a diferencia de otros lenguajes, introduce el concepto de constante. Una constante, para PHP, es un identificador que se utiliza para almacenar datos fijos simples.

Se diferencian de las variables, en dos aspectos:

- Almacenan datos simples (aunque esto, es un punto discutible) como una cadena de texto, un entero, un flotante, etc.
- Una vez definidos no pueden modificarse.

Definición clásica de constantes en PHP

Originalmente, PHP requiere del uso de la función `define('NOMBRE_DE_LA_CONSTANTE', 'valor')` para declarar y definir una constante:

```
define('PRECIO', 25.78);  
define('PRODUCTO', 'Short de baño para niño');
```

```
define('HAY_STOCK', False);
```

Por convención, **el nombre de las constantes se define en letras mayúsculas**. No obstante, aplican las reglas de nombre para la definición de variables.

Este tipo de constante, **puede definirse en cualquier ámbito de la aplicación**, ya sea dentro de una función como fuera de ella.

Para **llamar a una constante**, simplemente se hace referencia a ella, por el nombre:

```
echo PRECIO; // imprime 25.78
```

Este tipo de constantes, **admiten como valor**, cualquier tipo de dato simple, incluso, **una variable**:

```
$nombre = strip_tags($_GET['nombre']);  
define('NOMBRE', $nombre);
```

Definición de constantes en PHP 5.3

Desde la versión 5.3 de PHP, se introdujo el uso de la palabra clave **const** para definir constantes en PHP:

```
const PRECIO = 25.78;  
const PRODUCTO = 'Short de baño para niños';
```

Este tipo de constantes, son las que utilizaremos en nuestros códigos, puesto que las mismas, **introducen un concepto más preciso y exacto de lo que es una constante**:

- Solo pueden **declararse en el ámbito global de la aplicación**;
- Admiten cualquier tipo de dato simple, pero **no admiten variables**;
- **El valor** de estas constantes, **no puede formarse dinámicamente** (es “constante” en todo sentido);
- **No pueden ser redeclaradas**;

Finalidad de las constantes

Si bien en los ejemplos anteriores, hemos utilizado un precio y producto para demostrar como definir constantes, la finalidad de éstas, debe ser **definir datos no variables inherentes al núcleo de una aplicación**. Para ver un uso práctico y preciso, referirse al taller de archivos y web forms.

Variables variables

Leer la frase “variables variables” no solo parece redundante, sino además, inexacto y bastante confuso. Lo cierto, es que no existe otra forma de poder llamar a las **variables cuyos nombres se forman dinámicamente y pueden ser modificados**.

Es decir, que son “variables” porque aceptan datos que pueden ser modificados y a la vez, **vuelven a ser “variables” porque además de sus datos, podemos modificar sus nombre**:

```
$nombre_de_variable = 'precio';  
$$nombre_de_variable = 25.78;  
echo $nombre_de_variable; // imprime precio  
echo $$nombre_de_variable; // imprime 25.78
```

Esto significa, que el nombre de la variable que almacena el valor 25.78 será “precio”. Es decir, que estamos creando una variable, cuyo nombre es dinámico y por tanto, desconocemos, pero podemos acceder a ella, ya que el nombre otorgado, es el valor de otra variable:

```
$a = "mi_variable";  
$$a = 75;  
  
echo "El nombre de \$$a es \${$a}";  
// salida: El nombre de $$a es $mi_variable
```

Variables superglobales

Como hemos podido ver, PHP dispone de variables globales a las cuales se accede mediante el uso de la palabra clave **global**.

Así como existen las variables globales, también podemos encontrar **variables superglobales**.

Estas variables superglobales, suelen ser **arrays asociativos** desde los cuales PHP, de forma nativa, nos facilita su **acceso desde cualquier parte de la aplicación** sin necesidad de utilizar la palabra clave `global`, ya que son variables internas.

A lo largo del curso, hemos utilizado dos variables superglobales:

\$_GET Un array asociativo tipo clave => valor, de los parámetros pasados al script mediante el método HTTP GET, es decir, parámetros pasados por URL.

\$_POST Al igual que el anterior, es un array asociativo formado por clave => valor, pero que almacena los datos pasados al script, mediante el método HTTP POST, generalmente, a través de un formulario.

Además de `$_GET` y `$_POST`, existen otras variables superglobales, que veremos más adelante. Una de las más importantes, es la variable superglobal **\$_SERVER** que contiene información del entorno del servidor y de la ejecución.

Entre la lista de índices (claves) de este array asociativo superglobal, podemos encontrar algunos de uso frecuente como `REQUEST_METHOD` que nos retorna el método de petición HTTP del script en ejecución (POST, GET, PUT o HEAD) o

REQUEST_URI que nos devuelve la URI completa que se utilizó para acceder al script, entre otros.

```
$metodo = $_SERVER['REQUEST_METHOD'];
$uri = $_SERVER['REQUEST_URI'];

foreach($_SERVER as $clave=>$valor) {
    echo "\$_SERVER['$clave'] = $valor<br/>";
}
```

El foreach anterior, generará una salida similar a la siguiente (se resaltan los keys más usuales):

```
$_SERVER['HTTP_HOST'] = localhost
$_SERVER['HTTP_CONNECTION'] = keep-alive
$_SERVER['HTTP_CACHE_CONTROL'] = max-age=0
$_SERVER['HTTP_USER_AGENT'] = Mozilla/5.0 (X11; Linux i686) AppleWebKit/535.11
(KHTML, like Gecko) Chrome/17.0.963.79 Safari/535.11
$_SERVER['HTTP_ACCEPT'] = text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
$_SERVER['HTTP_ACCEPT_ENCODING'] = gzip,deflate,sdch
$_SERVER['HTTP_ACCEPT_LANGUAGE'] = es-419,es;q=0.8
$_SERVER['HTTP_ACCEPT_CHARSET'] = ISO-8859-1,utf-8;q=0.7,*;q=0.3
$_SERVER['PATH'] = /usr/local/bin:/usr/bin:/bin
$_SERVER['SERVER_SIGNATURE'] =
$_SERVER['SERVER_SOFTWARE'] = Apache
$_SERVER['SERVER_NAME'] = localhost
$_SERVER['SERVER_ADDR'] = ::1
$_SERVER['SERVER_PORT'] = 80
$_SERVER['REMOTE_ADDR'] = ::1
$_SERVER['DOCUMENT_ROOT'] = /var/www
$_SERVER['SERVER_ADMIN'] = webmaster@localhost
$_SERVER['SCRIPT_FILENAME'] = /var/www/euge/file.php
$_SERVER['REMOTE_PORT'] = 47578
$_SERVER['GATEWAY_INTERFACE'] = CGI/1.1
$_SERVER['SERVER_PROTOCOL'] = HTTP/1.1
$_SERVER['REQUEST_METHOD'] = GET
$_SERVER['QUERY_STRING'] = parametro=valor
$_SERVER['REQUEST_URI'] = /euge/file.php?parametro=valor
$_SERVER['SCRIPT_NAME'] = /euge/file.php
$_SERVER['PHP_SELF'] = /euge/file.php
$_SERVER['REQUEST_TIME'] = 1331772401
```

Más información sobre la superglobal \$_SERVER puede obtenerse en:

<http://www.php.net/manual/es/reserved.variables.server.php>.

Envío de correo electrónico con PHP

PHP, dispone de una función llamada `mail()` que permite enviar correos electrónicos tanto en texto plano como HTML, a través del servidor Web, utilizando la librería `sendmail` (generalmente, instalada por defecto).

La función `mail()` y su sintaxis

La función `mail()` requiere que mínimamente le sean pasados 3 parámetros: destinatario, asunto y mensaje:

```
$destinatario = "user@mail.com";  
$asunto = "Correo electrónico enviado desde PHP";  
$mensaje = "Esta es una prueba de envío.";  
  
mail($destinatario, $asunto, $mensaje);
```

El parámetro “destinatario”: formatos admitidos

La función `mail()` admite como destinatario, una o más direcciones de correo electrónico, debiendo mantener alguno de los siguientes formatos:

Único destinatario:

`user@mail.com`

Varios destinatarios:

`user2@mail.com, user2@mail.com, user5@mail.com`

Destinatario con nombre e e-mail:

Juan Pérez <mail@dominio.com>

Varios destinatarios con nombre e e-mail:

Juan Pérez <mail@dominio.com>, Ana Gómez <mail2@dominio.com>

Y lógicamente, cualquier combinación de las anteriores:

user2@mail.com, Ana Gómez <mail2@dominio.com>

Cabeceras adicionales como parámetro extra

Adicionalmente, pueden sumarse a la función `mail()` cabeceras adicionales a ser enviadas. Estas cabeceras, pueden utilizarse para agregar destinatarios con copia, con copia oculta, dirección de respuesta, remitente, tipo de contenido, etc.

```
$destinatario = "user@mail.com";
$sasunto = "Correo electrónico enviado desde PHP";
$mensaje = "Esta es una prueba de envío.";

$cabeceras_adicionales = "From: Ana María López <anita@mail.com>\r\n";
$cabeceras_adicionales .= "Reply-to: Rocío Irao <rocio@mail.com>\r\n";
$cabeceras_adicionales .= "Cc: Ariel Domingo <ariel@mail.com>\r\n";
$cabeceras_adicionales .= "Bcc: Supervisor <admin@mail.com>\r\n";

mail($destinatario, $sasunto, $mensaje, $cabeceras_adicionales);
```

Comprobando que el e-mail pudo enviarse

La función `mail()` retornará **TRUE** cuando el envío del mensaje haya podido concretarse. De lo contrario, retornará **FALSE**:

```
if(mail($destinatario, $sasunto, $mensaje, $cabeceras)) {
    echo "El e-mail se ha enviado satisfactoriamente.";
}
```

```
} else {  
    echo "Se ha producido un error al intentar enviar el e-mail";  
}
```

ADVERTENCIA

La función `mail()` **abre un socket SMTP en cada llamada**. Si bien puede utilizarse esta función, para realizar envíos iterativos (mediante un bucle `for`, por ejemplo), **se desaconseja iterar sobre esta función** en envíos masivos.

Enviando mensajes en formato HTML

Para poder enviar un mensaje con formato HTML desde PHP, solo será necesario, agregar en las cabeceras del correo electrónico, el **Content-type** correspondiente:

```
$destinatario = "user@mail.com";
$asunto = "Correo electrónico enviado desde PHP";

$mensaje = "<p><a href='http://es.wikipedia.org/wiki/Lorem_ipsum'>Lorem ipsum</a> ad his scripta blandit partiendo, eum fastidii accumsan euripidis in, eum liber hendrerit an. Qui ut wisi vocibus suscipiantur, <b>quo dicit ridens inciderint id</b>. Quo mundi lobortis reformidans eu, legimus senserit definiebas an eos. Eu sit tincidunt incorrupte definitionem, vis mutat affert percipit cu, eirmod consectetur signiferumque eu per. In usu latine equidem dolores. Quo no falli viris intellegam, ut fugit veritus placerat per.</p>";

$cabeceras_adicionales = "MIME-Version: 1.0\r\n";
$cabeceras_adicionales .= "Content-type: text/html; charset=utf-8\r\n";
$cabeceras_adicionales .= "From: Ana María López <anita@mail.com>\r\n";
$cabeceras_adicionales .= "Reply-to: Rocío Irao <rocio@mail.com>\r\n";
$cabeceras_adicionales .= "Cc: Ariel Domingo <ariel@mail.com>\r\n";
$cabeceras_adicionales .= "Bcc: Supervisor <admin@mail.com>\r\n";

if(mail($destinatario, $asunto, $mensaje, $cabeceras_adicionales)) {
    echo "El e-mail se ha enviado satisfactoriamente.";
} else {
    echo "Se ha producido un error al intentar enviar el e-mail";
}
```

TIP

El mensaje a enviar por correo electrónico puede ser cualquier tipo de contenido almacenado en una variable. Por lo tanto, puede utilizarse un sistema de plantillas HTML, con `file_get_contents()`, formatearse y manipularse el contenido e incluso, reemplazar datos dinámicamente a través de un formulario.

Funciones para el manejo de Fecha y Hora

La librería de funciones para la manipulación de fechas y horas de PHP, es lo suficientemente amplia, para permitirnos un control absoluto en el manejo de las mismas.

Veremos aquí, aquellas que utilizaremos con mayor frecuencia. Sin embargo, una guía completa de referencias de funciones para fecha y hora, puede obtenerse, visitando la documentación oficial en <http://nc.php.net/manual/es/ref.datetime.php>

Funciones simples de fecha y hora

Obtener la fecha y hora actual en un array asociativo

getdate() es la función indicada para obtener la información relativa a la fecha y hora actual, en un array asociativo:

```
$datos_fecha_hora = getdate();
print_r($datos_fecha_hora);
/*
Array
(
    [seconds] => 3
    [minutes] => 53
    [hours] => 16
    [mday] => 15
    [wday] => 4
    [mon] => 3
    [year] => 2012
    [yday] => 74
    [weekday] => Thursday
    [month] => March
    [0] => 1331841183
)
```

* /

El **array asociativo retornado por `getdate()`**, como bien se indica en el manual oficial³, devolverá las siguientes claves:

Clave	Descripción	Ejemplo de valores devueltos
seconds	Representacion numérica de los segundos	0 a 59
minutes	Representacion numérica de los minutos	0 a 59
hours	Representacion numérica de las horas	0 a 23
mday	Representacion numérica del día del mes	1 a 31
wday	Representacion numérica del día de la semana	0 (para Domingo) hasta 6 (para Sábado)
mon	Representacion numérica de un mes	1 hasta 12
year	Una representacion numérica completa de una año, 4 dígitos	Ejemplos: 1999 o 2003
yday	Representacion numérica del día del año	0 hasta 365
weekday	Una representación textual completa del día de la semana	Sunday hasta Saturday
month	Una representación textual completa de un mes, como January o March	January hasta December
0	Los segundos desde la Época Unix	Dependiente del Sistema, típicamente -2147483648 hasta 2147483647 .

Obtener fecha y hora actual con formato en una cadena de texto

Una función que hemos utilizado mucho, es **`date()`**. Esta función nos permite obtener datos relacionados a la fecha y hora actual, con un formato específico. Este formato, se especifica como parámetro tipo string:

```
echo date('Y-m-d'); // 2012-03-15
```

Dentro del parámetro tipo string, algunos de los **formatos**

³ <http://nc.php.net/manual/es/function.getdate.php#refsect1-function.getdate-returnvalues>

combinables de los cuales disponemos, son los siguientes:

Carácter de formato	Descripción	Ejemplo de valores devueltos
Día		
<i>d</i>	Día del mes, 2 dígitos con ceros iniciales	01 a 31
<i>D</i>	Una representación textual de un día, tres letras	Mon hasta Sun
<i>j</i>	Día del mes sin ceros iniciales	1 a 31
<i>J</i> ('L' minúscula)	Una representación textual completa del día de la semana	Sunday hasta Saturday
<i>N</i>	Representación numérica ISO-8601 del día de la semana (añadido en PHP 5.1.0)	1 (para lunes) hasta 7 (para domingo)
<i>w</i>	Representación numérica del día de la semana	0 (para domingo) hasta 6 (para sábado)
<i>z</i>	El día del año (comenzando por 0)	0 hasta 365
Semana		
<i>W</i>	Número de la semana del año ISO-8601, las semanas comienzan en lunes (añadido en PHP 4.1.0)	Ejemplo: 42 (la 42ª semana del año)
Mes		
<i>F</i>	Una representación textual completa de un mes, como January o March	January hasta December
<i>m</i>	Representación numérica de una mes, con ceros iniciales	01 hasta 12
<i>M</i>	Una representación textual corta de un mes, tres letras	Jan hasta Dec
<i>n</i>	Representación numérica de un mes, sin ceros iniciales	1 hasta 12
<i>t</i>	Número de días del mes dado	28 hasta 31
Año		
<i>L</i>	Si es un año bisiesto	1 si es bisiesto, 0 si no.
<i>Y</i>	Una representación numérica completa de un año, 4 dígitos	Ejemplos: 1999 o 2003
<i>y</i>	Una representación de dos dígitos de un año	Ejemplos: 99 o 03
Hora		
<i>a</i>	Ante meridiem y Post meridiem en minúsculas	am o pm
<i>A</i>	Ante meridiem y Post meridiem en mayúsculas	AM o PM
<i>g</i>	Formato de 12 horas de una hora sin ceros iniciales	1 hasta 12
<i>G</i>	Formato de 24 horas de una hora sin ceros iniciales	0 hasta 23
<i>h</i>	Formato de 12 horas de una hora con ceros iniciales	01 hasta 12

Carácter de formato	Descripción	Ejemplo de valores devueltos
H	Formato de 24 horas de una hora con ceros iniciales	00 hasta 23
i	Minutos, con ceros iniciales	00 hasta 59
s	Segundos, con ceros iniciales	00 hasta 59
Zona Horaria		
e	Identificador de zona horaria (añadido en PHP 5.1.0)	Ejemplos: UTC , GMT , Atlantic/Azores
I (i mayúscula)	Si la fecha está en horario de verano o no	1 si está en horario de verano, 0 si no.
O	Diferencia de la hora de Greenwich (GMT) en horas	Ejemplo: +0200
P	Diferencia con la hora de Greenwich (GMT) con dos puntos entre horas y minutos (añadido en PHP 5.1.3)	Ejemplo: +02:00
T	Abreviatura de la zona horaria	Ejemplos: EST , MDT ...
Z	Índice de la zona horaria en segundos. El índice para zonas horarias al oeste de UTC siempre es negativo, y para aquellas al este de UTC es siempre positivo.	-43200 hasta 50400
Fecha/Hora Completa		
c	Fecha ISO 8601 (añadido en PHP 5)	2004-02-12T15:19:21+00:00
r	Fecha con formato » RFC 2822	Ejemplo: Thu, 21 Dec 2000 16:01:07 +0200
U	Segundos desde la Época Unix (1 de Enero del 1970 00:00:00 GMT)	

Alternativamente, es posible pasar como parámetro a `date()` una **constante de formato predefinida**:

```
echo date(DATE_RSS);
// Thu, 15 Mar 2012 18:16:21 -0300
```

Una **lista completa de las constantes de formato predefinidas**, puede obtenerse en <http://nc.php.net/manual/es/class.datetime.php#datetime.constants.types>

AVISO

Nótese que la hora también puede obtenerse, en formato **hora UNIX**, con la función **time()**:

<http://nc.php.net/manual/es/function.time.php>

Validar una fecha

Podemos validar la veracidad de una fecha, mediante el uso de la función **checkdate(\$mes, \$dia, \$año)** teniendo en cuenta que retornará **TRUE** cuando se trate de una fecha válida, o **FALSE** o en caso contrario:

```
if(checkdate(12, 25, 2011) === True) {  
    echo "Fecha válida";  
} else {  
    echo "Fecha no válida";  
}
```

Cálculo de fecha / hora sencillo

Es posible realizar cálculos sencillos con la fecha y hora, combinando el uso de las funciones **date()** y **time()**:

```
$hoy = date('Y-m-d');  
$manana = date('Y-m-d', (time() + (1 * 24 * 60 * 60)));  
$ayer = date('Y-m-d', (time() - (1 * 24 * 60 * 60)));  
/*  
    (1 * 24 * 60 * 60) equivale a:  
        1 día  
        24 horas  
        60 minutos  
        60 segundos  
*/  
echo $ayer . chr(10) . $hoy . chr(10) . $manana . chr(10);  
/*  
2011-11-08 (ayer)  
2011-11-09 (hoy)  
2011-11-10 (mañana)  
*/
```

Lo anterior, es posible gracias a que la función **date()** puede recibir opcionalmente como parámetro, una marca de tiempo tipo UNIX (**timestamp**). Cuando la marca de tiempo no es pasada como parámetro, **date()** retornará el formato indicado,

teniendo en cuenta la fecha y hora actual. Pero cuando una marca de tiempo le es pasada, formará esa fecha y hora.

Dado que `time()` retorna la fecha/hora en forma UNIX (timestamp) al sumar o restar mediante otra marca de tiempo, será posible obtener el timestamp deseado, que permitirá ser formateado con `date()`.

```
php > echo 1 * 24 * 60 * 60 ;  
86400  
php > echo time();  
1331847837  
php > $a = 1331847837 + 86400;  
php > echo $a;  
1331934237  
php > echo date('Y-m-d', $a);  
2012-03-16
```

Otra forma de **obtener la marca de tiempo de una fecha determinada** es con la función `mktime()`:

```
php > echo mktime();  
1331848266
```

Si `mktime()` no recibe parámetros, retornará la marca de tiempo de la fecha y hora actual. De lo contrario, deberá recibir los parámetros correspondientes a **hora, minuto, segundo, mes, día y año**.

```
echo mktime(0, 0, 0, 12, 25, 2011);  
// obtiene la marca de tiempo del 25 de diciembre de 2011
```

Por lo tanto, podríamos realizar cálculos, utilizando `mktime()` como se muestra a continuación:

```
$dia_hoy = (int)date('d');  
$mes_hoy = (int)date('m');  
$anio_hoy = (int)date('Y');  
$semana_siguiente = mktime(0, 0, 0, $mes_hoy, $dia_hoy+7, $anio_hoy);  
echo date('Y-m-d', $semana_siguiente);
```

Aunque esta última forma, no es la más acertada.

Ejemplos prácticos de cálculos basados en fechas

¿Cuánto tiempo ha pasado?

Problema:

El 15 de marzo de 2011, Natalia le comentó a su madre, que comenzaría a ahorrar dinero para comprar un nuevo ordenador y que a tal fin, todos los días guardaría en una caja de zapatos, \$2,75. ¿Cuánto dinero habrá ahorrado Natalia a la fecha de hoy?

Solución:

```
// obtengo la marca de tiempo para el 15/03/2011
$fecha_inicio = mktime(0, 0, 0, 3, 15, 2011);

// obtengo la marca de tiempo para hoy
$fecha_fin = mktime();

// obtengo la diferencia timestamp entre ambas fechas
$diferencia = ($fecha_fin - $fecha_inicio);

// convierto a días la diferencia timestamp
$dias = $diferencia / (24 * 60 * 60); # días que pasaron entre dos fechas

// dinero ahorrado x día
$dinero = 2.75;

// obtengo el importe total ahorrado,
// multiplicando los días x el importe diario
$ahorro = $dias * $dinero;

// Imprimo el resultado
echo $ahorro;
```

¿Qué edad tiene...?

Problema:

Lucas, nació el 27 de Septiembre de 1978 ¿Qué edad tiene hoy?

Solución:

```
// obtengo la marca de tiempo de la fecha de nacimiento
$fecha_nacimiento = mktime(0, 0, 0, 9, 27, 1978);

// obtengo la marca de tiempo de la fecha actual
$hoy = mktime();

// obtengo la diferencia entre fecha de nacimiento y hoy
$diferencia = $hoy - $fecha_nacimiento;

// obtengo la edad
$edad = $diferencia / (365 * 24 * 60 * 60); # años que pasaron entre 2 fechas

// imprimo la edad
echo (int)$edad;
```

¿En qué fecha nació...?

Problema:

Luciana tiene hoy, 15 años ¿En qué fecha pudo haber nacido Luciana?

Solución:

```
// edad actual de Luciana en años
$edad = 15;

// probable año de nacimiento de luciana
$anio = (int)date('Y') - $edad;

// probable fecha de nacimiento más antigua
$probable_mas_antigua = date('d/m/') . ($anio-1);

// probable fecha de nacimiento más reciente
$probable_mas_reciente = date('d/m/') . $anio;

echo <<<EOT
Luciana tiene que haber nacido después del $probable_mas_antigua y antes
```

```
o durante el $probable_mas_reciente .  
EOT;
```

Funciones matemáticas

PHP dispone de una gran galería de funciones matemáticas predefinidas, que pueden encontrarse en la documentación oficial visitando <http://www.php.net/manual/es/ref.math.php>

Muchas de estas funciones, nos resultarán de gran ayuda, convirtiéndose en **funciones de uso frecuente** para nuestros programas. Veremos las mismas a continuación.

Obtener un número elevado a la potencia

`pow($base, $potencia)`

```
echo pow(2, 3); // 8  
echo pow(5, 2); // 25
```

Obtener el número más alto y el número más bajo

`max($valores) - min($valores)`

```
$precios = array(12.75, 43.90, 106.60, 9, 35.85);  
$producto_mas_caro = max($precios); // 106.6  
$producto_mas_barato = min($precios); // 9
```

```
$mejor_oferta = max(107.75, 109.84); // 109.84  
$peor_oferta = min(107.75, 109.84); // 107.75
```

Redondear un número con N cantidad de decimales

`round($numero, $decimales)`

```
$bruto = 1573.94;  
$alicuota_iva = 10.5;  
$iva = $bruto * $alicuota_iva / 100; // 165.2637  
$iva = round($iva, 2); // 165.26
```

Redondear un número hacia abajo

`floor($numero)`

```
$bruto = 1573.94;  
$alicuota_iva = 10.5;  
$iva = $bruto * $alicuota_iva / 100; // 165.2637  
$iva = floor($iva); // 165
```

Redondear un número hacia arriba

`ceil($numero)`

```
$bruto = 1573.94;  
$alicuota_iva = 10.5;  
$iva = $bruto * $alicuota_iva / 100; // 165.2637  
$iva = ceil($iva); // 166
```

Obtener un número entero aleatorio

```
rand($numero_minimo, $numero_maximo)
```

```
$password = rand(199999, 999999); // 158035
```

Funciones para el manejo de matrices

En capítulos anteriores, cuando hablamos sobre arrays, pudimos ver varias funciones útiles para manejar estos tipos más complejos. A lo largo de los talleres y ejercicios que hemos hecho, también pudimos llevar dichas funciones a la práctica.

Veremos aquí una lista de funciones para el manejo de matrices, de forma más detallada. No obstante, una lista completa puede obtenerse en <http://www.php.net/manual/es/ref.array.php>

Diviando y uniando arrays

Dividir un array en matrices más pequeñas

```
array_chunk($array, $tamaño[, boolean  
$conservar_claves])
```

Nótese que el tercer argumento es opcional. Por defecto, `array_chunk`, creará nuevas claves en los nuevos array. Pero si se indica `TRUE`, conservará estas claves.

```
$personas = array('Juan', 'Emilse', 'Pedro', 'Eliseo', 'Rosa', 'Noelia',  
                 'Raul', 'Esteban', 'Diego');  
  
$grupos = array_chunk($personas, 3);  
  
print_r($grupos); /*  
Array  
(  
    [0] => Array  
        (  
            [0] => Juan  
            [1] => Emilse  
            [2] => Pedro  
        )  
    [1] => Array  
        (  
            [0] => Eliseo  
            [1] => Rosa  
            [2] => Noelia  
        )  
    [2] => Array  
        (  
            [0] => Raul  
            [1] => Esteban  
            [2] => Diego  
        )  
) */
```

Con una iteración, incluso, podríamos asignar los grupos creados a nuevos array:

```
$personas = array('Juan', 'Emilse', 'Pedro', 'Eliseo', 'Rosa', 'Noelia',  
                 'Raul', 'Esteban', 'Diego');  
  
$grupos = array_chunk($personas, 3);  
  
foreach($grupos as $numero=>$grupo) {  
    $nombre_array = "grupo_{$numero}";  
    $$nombre_array = $grupo;  
}
```

Finalmente, obtendríamos 3 nuevos arrays, llamados \$grupo_0, \$grupo_1 y \$grupo_2, respectivamente.

Obtener la porción específica de un array

`array_slice($array, $desde[, $hasta])`

```
$personas = array('Juan', 'Emilse', 'Pedro', 'Eliseo', 'Rosa', 'Noelia',
                  'Raul', 'Esteban', 'Diego');

$primeras_3_personas = array_slice($personas, 0, 3);
print_r($primeras_3_personas);
/*
Array
(
    [0] => Juan
    [1] => Emilse
    [2] => Pedro
)
*/

$personas_restantes = array_slice($personas, 3);
print_r($personas_restantes);
/*
Array
(
    [0] => Eliseo
    [1] => Rosa
    [2] => Noelia
    [3] => Raul
    [4] => Esteban
    [5] => Diego
)
*/
```

Combinar dos arrays, utilizando uno para las claves y otro para los valores

`array_combine($array_claves, $array_valores)`

```
$comodines = array('{TITULO}', '{SUBTITULO}');
$valores = array('Manual de PHP', 'Trabajando con arrays');
$datos = array_combine($comodines, $valores);
print_r($datos);
/*
Array
(
    [{TITULO}] => Manual de PHP
    [{SUBTITULO}] => Trabajando con arrays
)
*/
```

Combinar dos o más arrays

array_merge(\$array_1, \$array_2[, \$mas_arrays])

```
$grupo_a = array('Eliseo', 'Noemi', 'Santiago');
$grupo_b = array('Diego', 'Cecilia', 'Roman');
$personas = array_merge($grupo_a, $grupo_b);
print_r($personas);
/*
Array
(
    [0] => Eliseo
    [1] => Noemi
    [2] => Santiago
    [3] => Diego
    [4] => Cecilia
    [5] => Roman
)
*/
```

Combinar dos o más arrays multidimensionales de manera recursiva

array_merge_recursive(\$array_1, \$array_2[, \$mas_arrays])

```
$persona_a = array('Nombre'=>'Eliseo', 'Edad'=>25);
$persona_b = array('Nombre'=>'Miriam', 'Edad'=>37);
$personas = array_merge_recursive($persona_a, $persona_b);
print_r($personas);
/*
Array
(
    [Nombre] => Array
        (
            [0] => Eliseo
            [1] => Miriam
        )
    [Edad] => Array
        (
            [0] => 25
            [1] => 37
        )
)
```

```
)  
*/
```

Ordenando Arrays por sus valores

Ordenar un array de menor a mayor

sort(\$array)

```
$nombres = array('Noemi', 'Diego', 'Ana', 'Eliseo');  
sort($nombres);  
print_r($nombres);  
/*  
Array  
(  
    [0] => Ana  
    [1] => Diego  
    [2] => Eliseo  
    [3] => Noemi  
)  
*/
```

Ordenar un array de mayor a menor

rsort(\$array)

```
$nombres = array('Noemi', 'Diego', 'Ana', 'Eliseo');  
rsort($nombres);  
print_r($nombres);  
/*  
Array  
(  
    [0] => Noemi  
    [1] => Eliseo  
    [2] => Diego  
    [3] => Ana  
)  
*/
```

Ordenar un array de menor a mayor manteniendo la relación con los índices

asort(\$array)

```
$nombres = array('Noemi', 'Diego', 'Ana', 'Eliseo');
asort($nombres);
print_r($nombres);
/*
Array
(
    [2] => Ana
    [1] => Diego
    [3] => Eliseo
    [0] => Noemi
)
*/
```

Ordenar un array de mayor a menor manteniendo la relación con los índices

arsort(\$array)

```
$nombres = array('Noemi', 'Diego', 'Ana', 'Eliseo');
arsort($nombres);
print_r($nombres);
/*
Array
(
    [0] => Noemi
    [3] => Eliseo
    [1] => Diego
    [2] => Ana
)
*/
```

Ordenando Arrays por su clave

Ordenar un array de menor a mayor por su clave

ksort(\$array)

```
$personas = array(
    'Nombre' => 'Miguel',
    'Apellido' => 'Montero',
);
ksort($personas);
print_r($personas);
/*
Array
(
    [Apellido] => Montero
    [Nombre] => Miguel
)
*/
```

Ordenar un array de mayor a menor por su clave

krsort(\$array)

```
$personas = array(
    'Nombre' => 'Miguel',
    'Apellido' => 'Montero',
    'Talle' => 'XL',
);
krsort($personas);
print_r($personas);
/*
Array
(
    [Talle] => XL
    [Nombre] => Miguel
    [Apellido] => Montero
)
*/
```

Comparando funciones de ordenamiento de arrays

Función	Ordena por	Mantiene las claves asociadas	Orden de clasificación
asort	valor	SI	menor a mayor
arsort	valor	SI	mayor a menor

ksort	clave	SI	menor a mayor
krsort	clave	SI	mayor a menor
sort	valor	NO	menor a mayor
rsort	valor	NO	mayor a menor

Agregar y Eliminar elementos de un array

Agregar elementos al final del array

`array_push($array, $valores)`

```
$personas = array('Juan', 'Emilio');
array_push($personas, 'Miguel', 'Ana', 'Herminio');
print_r($personas);
/*
Array
(
    [0] => Juan
    [1] => Emilio
    [2] => Miguel
    [3] => Ana
    [4] => Herminio
)
*/
```

Agregar elementos al comienzo del array

`array_unshift($array, $valores)`

```
$personas = array('Juan', 'Emilio');
array_unshift($personas, 'Miguel', 'Ana', 'Herminio');
print_r($personas);
/*
Array
(
    [0] => Miguel
    [1] => Ana
    [2] => Herminio
    [3] => Juan
)
```

```
        [4] => Emilio
    )
    */
```

Eliminar el último elemento de un array

`array_pop($array)`

```
$personas = array('Juan', 'Emilio', 'Ana');
array_pop($personas);
print_r($personas);
/*
Array
(
    [0] => Juan
    [1] => Emilio
)
*/
```

Eliminar el primer elemento de un array

`array_shift($array)`

```
$personas = array('Juan', 'Emilio', 'Ana');
array_shift($personas);
print_r($personas);
/*
Array
(
    [0] => Emilio
    [1] => Ana
)
*/
```

Eliminar valores duplicados en un array

`array_unique($array)`

```
$personas = array('Juan', 'Emilio', 'Ana', 'Emilio');
$personas = array_unique($personas);
```

```
print_r($personas);
/*
Array
(
    [0] => Juan
    [1] => Emilio
    [2] => Ana
)
*/
```

Búsquedas y filtros

Contar la cantidad de veces que los elementos aparecen en un array

array_count_values(\$array)

```
$frutas = array('pera', 'manzana', 'pera', 'durazno', 'melón', 'sandía',
                'kiwi', 'manzana', 'melón', 'pera', 'mandarina', 'naranja',
                'limón', 'lima', 'pomelo', 'pera');

$repeticiones = array_count_values($frutas);

foreach($repeticiones as $fruta=>$veces) {
    if($veces > 1) {
        echo "Usted repitió {$fruta} {$veces} veces". Chr(10);
    }
}
/*
Usted repitió pera 4 veces
Usted repitió manzana 2 veces
Usted repitió melón 2 veces
*/
```

Contar la cantidad de elementos de un array

count(\$array)

```
$frutas = array('pera', 'manzana', 'durazno');
```



```
echo count($frutas); // 3
```

Obtener la suma matemática de los valores de un array

array_sum(\$array)

```
$precios = array(75.40, 93.12, 7, 25.18, 173.60);  
$total = array_sum($precios);  
echo $total; // 374.3
```

Obtener las diferencias entre dos o más arrays

array_diff(\$array_1, \$array_2[, \$array_3, ...])

```
$frutas_1 = array('pera', 'manzana', 'durazno', 'melón', 'sandía', 'kiwi',  
                 'mandarina', 'naranja', 'limón', 'lima', 'pomelo');  
  
$frutas_2 = array('pera', 'manzana', 'durazno', 'melón', 'sandía', 'kiwi',  
                 'mandarina', 'lima', 'pomelo');  
  
$diferencias = array_diff($frutas_1, $frutas_2);  
  
echo "Las siguientes frutas no están en los 2 arrays:" . Chr(10);  
  
foreach($diferencias as $fruta_no_repetida) {  
    echo "- {$fruta_no_repetida}" . Chr(10);  
}  
/*  
Las siguientes frutas no están en los 2 arrays:  
- naranja  
- limón  
*/
```

Filtrar datos de un array, utilizando una función de retorno

array_filter(\$array, \$funcion)

```
$datos = array(25, 43.2, 64.98, 33.7, 'luna', 95, 32, 60.05, 'agua', 'sol');

function retornar_enteros($dato) {
    if(is_int($dato)) {
        return $dato;
    }
}

function retornar_otros_datos($dato) {
    if(!is_int($dato)) {
        return $dato;
    }
}

$enteros = array_filter($datos, 'retornar_enteros');
$otros_datos = array_filter($datos, 'retornar_otros_datos');

print_r($enteros);
/*
Array
(
    [0] => 25
    [5] => 95
    [6] => 32
)
*/

print_r($otros_datos);
/*
Array
(
    [1] => 43.2
    [2] => 64.98
    [3] => 33.7
    [4] => luna
    [7] => 60.05
    [8] => agua
    [9] => sol
)
*/
```

Verificar si un array contiene una clave determinada

`array_key_exists($clave, $array)`

```
if(!array_key_exists('password', $_POST)) {  
    echo 'Debe indicar una contraseña';  
}
```

Obtener todas las claves de un array o todos los valores

`array_keys($array) - array_values($array)`

```
$libro = array(  
    'Titulo' => 'Manual de PHP',  
    'Subtitulo' => 'Trabajando con arrays',  
    'Autor' => 'Eugenia Bahit',  
    'Fecha' => '12/10/2011',  
);  
  
$claves = array_keys($libro);  
$valores = array_values($libro);  
  
print_r($claves);  
/*  
Array  
(  
    [0] => Titulo  
    [1] => Subtitulo  
    [2] => Autor  
    [3] => Fecha  
)  
*/  
  
print_r($valores);  
/*  
Array  
(  
    [0] => Manual de PHP  
    [1] => Trabajando con arrays  
    [2] => Eugenia Bahit  
    [3] => 12/10/2011  
)  
*/
```

Verificar si un array contiene una valor determinada

`in_array($valor, $array)`

```
if(in_array(50, $puntaje)) {  
    echo 'Usted ha obtenido el máximo puntaje posible en una respuesta';  
}
```

Buscar un valor detrminado en un array y obtener su clave correspondiente

`array_search($valor, $array)`

```
$personas = array('Juan', 'Ana', 'Emilse', 'Diego');  
  
$persona_buscada = 'Emilse';  
$resultado = array_search($persona_buscada, $personas);  
var_dump($resultado); // int(3)
```

Cookies y Sesiones de usuario

¿Qué es una cookie?

Una cookie es un **archivo de texto plano, que se almacena remotamente -en la máquina del cliente-** a través del navegador.

Cada cookie -archivo- **es un conjunto de datos** que provienen del mismo servidor -más precisamente, del mismo dominio-.

Básicamente, cada cookie tendrá asociado a ella, un nombre que la identifique y un valor.

Los datos que se almacenan remotamente en el ordenador del cliente, pueden ser de cualquier tipo y el objetivo de estos, es:

- Almacenar información relativa al usuario;
- Acceder a esa información, para realizar seguimientos y acciones personalizadas con respecto a cada usuario en particular

De esta forma, podríamos pedirle a un usuario, que ingrese su nombre mediante un Web Form (por ejemplo, ingresa el nombre "Javier"), almacenar ese dato en una cookie, y así, cada vez que el usuario ingrese a nuestra aplicación o Sitio Web, buscaríamos esa cookie, accederíamos a ella, leeríamos los datos y finalmente, podríamos mostrarle al usuario, un mensaje personalizado, que diga "Hola Javier!".

Vale aclarar entonces, que las cookies se pueden **crear, leer, modificar y eliminar**, tanto por nuestra aplicación como por el mismo usuario, si éste es además de curioso, medianamente avezado.

Las cookies no son eternas

Así como **una cookie**, posee un nombre y valor asociado, también **puede tener asociada, una fecha de caducidad o período de validez**. De esta forma, podemos crear una cookie indicando que expire el 12 de febrero de 2015 y otra, que lo haga dentro de 6 días.

Pero **las cookies, pueden desaparecer antes de lo previsto**, ya que al ser archivos pertenecientes al usuario y por tanto, almacenados en su propio ordenador, el usuario podría eliminarlos.

¿Qué son las sesiones de usuario?

Las sesiones de usuario, al igual que las cookies, son una forma de almacenar información relativa al usuario, que permiten que dicha información se propague y mantenga activa, con cada acción del usuario sobre nuestra App o Sitio Web.

Las sesiones, también son almacenadas remotamente mediante cookies, pero a la vez son retenidas localmente en memoria.

A diferencia de las cookies, **las sesiones expiran pasado un período de tiempo preestablecido, de inactividad por parte del usuario**.

Como **diferencias fundamentales entre cookies y sesiones**, podemos mencionar que:

- Las sesiones crean cookies, pero las cookies no crean sesiones;

- Las sesiones expiran automáticamente por inactividad del usuario, tras un período de tiempo predeterminado, mientras que las cookies expiran en la fecha que se les indique o porque son eliminadas por el usuario;

Usos e importancia

Como bien hemos dicho antes, tanto cookies como sesiones se utilizan para personalizar la experiencia del usuario. De esta forma, podremos saber que todos los sistemas Web que restringen su acceso mediante contraseñas, pueden hacerlo gracias al uso de cookies y sesiones. Por ello, es tan importante tener dominio tanto de unas como de otras.

Lo básico

Antes de ver como implementar el uso de cookies y sesiones en una aplicación Web, necesitamos conocer cómo llevar adelante las acciones básicas que podemos realizar con las cookies y sesiones. Estas acciones son: **crearlas, leerlas, modificarlas y eliminarlas**.

Creación, lectura, modificación y eliminación de cookies

Para realizar acciones con cookies, además de un gran número de funciones, PHP nos brinda un **array superglobal** denominado **\$_COOKIE**, el cual nos permitirá acceder en todo momento a los datos del usuario.

Crear una cookie

Para crear una cookie utilizaremos la función **setcookie()** de PHP. En orden de aparición, los parámetros que esta función

recibe, son los siguientes:

Parámetros obligatorios:

1. **Nombre de la cookie.** Ejemplo: username
2. **Valor.** Ejemplo: javier75

Parámetros opcionales:

3. **Momento en el que debe expirar.** Si no se indica, caduca automáticamente. Ejemplo en segundos: `time() + 3600` -el equivalente a 1 hora-
4. **Directorio** en el cuál es válida la cookie. Se debe utilizar `'/'` para que sea válida en todo el dominio. Ejemplo: `'/'`
5. **Dominio.** Ejemplo: `eugeniabahit.com`
6. Solo se transmite por **HTTPS**. Ejemplo: `False`
7. Solo se transmite por **HTTP**. Ejemplo: `True`. Siempre se recomienda indicar `TRUE`, a fin de evitar que la cookie pueda ser accedida mediante JavaScript, y por lo tanto, vulnerable a ataques del tipo XSS.

```
$nombre = "nombre y apellido";  
$valor = "Eugenia Bahit";  
$expira = time() + (3600 * 24 * 365); // 1 año  
$dir = "/";  
$dominio = "desa.eugeniabahit.com"; // no será válida en www.eugeniabahit.com  
$https = FALSE;  
$http = TRUE;  
  
setcookie($nombre, $valor, $expira, $dir, $dominio, $https, $http);
```


Leer una cookie

Para leer una cookie, haremos uso del array superglobal `$_COOKIE`:

```
echo "Hola {$_COOKIE["nombre y apellido"]}!"; // Hola Eugenia Bahit!
```

Modificar una cookie

La forma correcta de modificar una cookie, es sobrescribirla, es decir, volver a crearla:

```
$nombre = "nombre_y_apellido";  
$valor = "Juan Pérez";  
$expira = time() + (3600 * 24 * 365);  
$dir = "/";  
$dominio = "desa.eugeniabahit.com";  
$https = FALSE;  
$http = TRUE;  
  
setcookie($nombre, $valor, $expira, $dir, $dominio, $https, $http);
```

Ahora, la cookie “nombre_y_apellido” tendrá el valor “Juan Pérez”.

Eliminar una cookie

Para eliminar una cookie, el mejor método es volver a crear la cookie, con valor `NULL` haciendo que expire antes de la fecha actual:

```
$nombre = "nombre_y_apellido";  
$valor = NULL;  
$expira = time() - (3600 * 24 * 365); // 1 año antes  
$dir = "/";  
$dominio = "desa.eugeniabahit.com";  
$https = FALSE;  
$http = TRUE;  
  
setcookie($nombre, $valor, $expira, $dir, $dominio, $https, $http);
```

Un ejemplo práctico con Cookies

Vamos a crear un script sencillo, que solicite al usuario, el idioma en el que desea leer un artículo y vamos a guardar su preferencia en una cookie, a fin de que cada vez que visite nuestro sitio Web, podamos decidir en qué idioma le mostraremos el artículo.

Paso a paso:

1. Crear una carpeta llamada `sitio-web-multi-idioma`
2. Dentro de ella, vamos a crear otra carpeta llamada `paginas`
3. Dentro de la carpeta `paginas`, vamos a crear 2 archivos:
`hola_en.html`
`hola_es.html`
4. En el archivo `hola_en.html` vamos a escribir cualquier texto en inglés, y en el archivo `hola_es.html`, cualquier texto en español

Ahora, dentro de la carpeta `sitio-web-multi-idioma` vamos a crear los siguientes archivos:

`template.html`
`cambiar_idioma.php`
`funciones.php`
`index.php`

En los cuales, vamos a colocar el código que se describe a continuación.

Archivo `template.html`

Será nuestra vista HTML, en la cual, mostraremos un formulario que permita al usuario elegir el idioma en el cual ver la página.

```

<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
  <title>Web Site</title>
</head>
<body>
  <header>
    <h1>Web Site</h1>
  </header>
  <nav>
    <form method="POST" action="cambiar_idioma.php" id="frm_idioma">
      <label for="lang">Elija su idioma / choose your language:</label>
      <select id="lang" name="idioma">
        <option value="es">Español</option>
        <option value="en">English</option>
      </select>
      <input type="submit" value="OK"/>
    </form>
  </nav>
  <article>
    {PAGINA}
    <!--
      Aquí se reemplazará el contenido por el del archivo hola_es.html
      u hola_en.html según el idioma elegido por el usuario
    -->
  </article>
</body>
</html>

```

Archivo funciones.php

En este archivo definiremos todas las funciones necesarias para recibir los datos del formulario, crear la cookie, renderizar el HTML y mostrar la página.

```

<?php
# Trae los datos del formulario
function get_idioma() {
  $pagina = "paginas/hola_es.html";
  if(isset($_POST['idioma'])) {
    $idioma = $_POST['idioma'];

    switch ($idioma) {
      case 'es':
        $pagina = "paginas/hola_es.html";
        break;
      case 'en':
        $pagina = "paginas/hola_en.html";
        break;
    }
  }
}

```

```
        return $pagina;
    }

    # Modifica el idioma elegido - crea o modifica la cookie
    function cambiar_idioma() {
        $pagina = get_idioma();
        setcookie("pagina", $pagina, time()+(3600*24*365));
        header('Location: index.php');
    }

    # Trae el contenido de la página según el idioma
    function get_pagina() {
        if(isset($_COOKIE['pagina'])) {
            $pagina = $_COOKIE['pagina'];
        } else {
            $pagina = "paginas/hola_es.html";
        }
        return file_get_contents($pagina);
    }

    # Muestra la página al usuario
    function mostrar_pagina() {
        $plantilla = file_get_contents("template.html");
        $contenido = get_pagina();
        $html = str_replace("{PAGINA}", $contenido, $plantilla);
        echo $html;
    }
?>
```

Archivo cambiar_idioma.php

A este archivo es enviado el formulario y es quien se encarga de llamar a la función que se ocupa de modificar o crear la cookie con la elección del idioma del usuario.

```
<?php
require_once("funciones.php");
cambiar_idioma();
?>
```

Archivo index.php

Este archivo, requerirá también de funciones.php. Será el archivo principal, que se encargue de llamar a la función mostrar_pagina().

```
<?php
require_once("funciones.php");
mostrar_pagina();
?>
```

Descarga los archivos de este ejemplo desde la Web del curso, en:

<http://taller-de-php.eugeniabahit.com>

O mira el **ejemplo en marcha** ingresando en:

<http://taller-de-php.eugeniabahit.com/Ejemplos-En-Marcha/sitio-web-multi-idioma/>

Trabajando con Sesiones

Llegó la hora de introducirnos más a fondo con las sesiones para ir finalizando con la primera parte del curso y, ya adentrarnos en el trabajo con bases de datos. Así que ¡No perdamos tiempo!

Primeros pasos con sesiones

Antes de comenzar a crear sesiones -y manipularlas-, es necesario saber, que para poder trabajar con ellas, a diferencia de las cookies, es necesario **inicializarlas**. Para ello, PHP nos provee de la función `session_start()`, la cual **debe ser llamada siempre, antes de realizar cualquier otra operación relacionada con sesiones**:

```
session_start();
```

AVISO:

Al igual que con las cookies, PHP también nos otorga un **array superglobal** para acceder a las sesiones, llamado **\$_SESSION**

Otra particularidad a tener en cuenta, es que PHP, genera un **identificador único de sesión del usuario**, -que nos permitirá utilizarlo para identificar en cada sesión y de manera inequívoca al usuario- al cual se puede acceder, invocando a la función **session_id()**:

```
session_start();  
echo session_id();
```

Crear una nueva sesión

Mediante el **array superglobal \$_SESSION**, podemos crear, leer y modificar sesiones, de manera simple y directa:

***Recuerda** que tanto para iniciar una nueva sesión como para reanudar una sesión existente siempre tendrás que hacerlo con `session_start()` sin excepción.*

```
session_start();  
$_SESSION['usuario'] = 'javier75'; // creo la sesión 'usuario'
```

Leer una sesión

Si se desea obtener la ID de sesión, habrá que recurrir a la función [session_start\(\)](#) como se indicó [anteriormente](#). Para leer una sesión creada por nosotros, bastará con invocar al array superglobal `$_SESSION['nombre_de_la_sesion']`:

```
session_start();
echo $_SESSION['usuario']; // javier75
```

Modificar la sesión

Si se desea **modificar la ID de sesión**, debe pasarse como parámetro a `session_id()`:

```
session_start();
session_id('nuevoID');
echo session_id(); // nuevoID
```

En cambio, si se desea **modificar cualquier variable de sesión**, creada por nosotros, bastará con modificar el array superglobal `$_SESSION`:

```
session_start();
$_SESSION['usuario'] = 'javier_1975';
```

Eliminar una variable de sesión

Para eliminar una variable de sesión, ésta, puede destruirse mediante `unset($_SESSION['nombre_de_la_variable_de_sesion']`:

```
session_start();  
unset($_SESSION['usuario']);
```

Pero **para destruir la sesión completa del usuario (incluyendo la ID de sesión)**, se debe recurrir a la función `session_destroy()` y eliminar la cookie con el nombre de la sesión, el cual se obtiene mediante `session_name()`:

```
session_start(); // reanudo la sesión  
unset($_SESSION); // destruyo todas las variables de sesión creadas  
  
// obtengo los parámetros de la cookie de sesión  
// los necesitaré para poder destruirla  
$datos_cookie = session_get_cookie_params();  
// sobrescribo la cookie de sesión -la elimino-  
setcookie(session_name(), NULL, time()-999999, $datos_cookie["path"],  
          $datos_cookie["domain"], $datos_cookie["secure"],  
          $datos_cookie["httponly"]);  
  
session_destroy(); // destruyo la sesión
```

Un caso práctico de uso de sesiones

Es el caso de restringir el acceso a ciertas páginas de nuestro sitio Web, solo a usuarios con permiso para hacerlo. Crearemos un programa muy simple, solo a modo de ejemplo.

Lo primero que haremos -ya que no utilizaremos ningún sistema de registro y administración de usuarios-, es crear un **usuario genérico con contraseña única**. Para evitar guardar estos datos en texto plano, lo que haremos, es utilizar PHP-CLI, para obtener el **hash MD5, de la combinación usuario contraseña**:

```
php > $u = "pepegriillo";  
php > $p = "_italia1975_";  
php > echo md5($u . $p);  
85ce93e9490c0fe6a6431f45c8837de8
```


En el ejemplo que coloqué, utilizo como usuario genérico **pepegrillo** y como contraseña **_italia1975_** y convierto a ambos (en el mismo paso) a su correspondiente hash MD5 (sin espacios, ni caracteres adicionales de ningún tipo).

De esta forma, cuando el usuario quiera ingresar a nuestras páginas restringidas, tendrá que utilizar como nombre de usuario **pepegrillo** y como clave **_italia1975_**

Luego, nosotros lo que haremos, será hashear los datos que ingrese el usuario y compararlos con nuestro hash original. A continuación, veremos como lograrlo y de que forma mantenerlo logueado en el sistema, a través de sesiones.

Una vez creado nuestro hash MD5, crearemos un archivo **settings.php** destinado a almacenar variables/constantes de entorno global.

```
<?php
# Dejaremos ya iniciada una sesión
session_start();

# aquí copiaremos nuestro hash MD5 obtenido con PHP-CLI
const HASH_ACCESO = "85ce93e9490c0fe6a6431f45c8837de8";
# formulario.html será el que pida el ingreso de user y pass al usuario
const PAGINA_LOGIN = "formulario.html";

# esta será una página cualquiera, con acceso restringido, a la cual
# redirigir al usuario después de iniciar su sesión en el sistema
const PAGINA_RESTRINGIDA_POR_DEFECTO = "pagina_de_muestra.php";

?>
```

A continuación, crearemos el formulario HTML, necesario para que el usuario inicie sesión en el sistema. Lo llamaremos **formulario.html**. Este formulario, enviará los datos por HTTP POST, a otro archivo llamado **iniciar.php** que crearemos luego.

```
<h1>Ingreso al sistema</h1>
```

```
<form method="POST" action="iniciar.php">
  Usuario: <input type="text" name="user" /><br/>
  Clave: <input type="password" name="pass" /><br/>
  <input type="submit" value="Ingresar" />
</form>
```

Ahora, crearemos el archivo principal de nuestro sistema: **funciones.php**

Aquí almacenaremos todas las funciones necesarias para:

- Iniciar la sesión
- Destruir la sesión (desconectar al usuario)
- Verificar si el usuario tiene sesión iniciada

Funciones necesarias

Funciones de acceso al sistema

```
/*
    Traigo los datos recibidos por HTTP POST
    y retorno el HASH MD5 de ambos
*/
function get_post_data() {
    $hash = "";
    if(isset($_POST['user']) && isset($_POST['pass'])) {
        $hash = md5($_POST['user'] . $_POST['pass']);
    }
    return $hash;
}
```

En la función anterior, primero inicializo un hash vacío. Luego verifico si user y pass han venido a través de HTTP POST con un valor declarado. De ser así, hasheo ambos datos (sin riesgos, ya que al cifrarlos con MD5 directamente, no hay posibilidad de que se inyecte código malicioso de ningún tipo).

Finalmente, retorno ese hash (si no vinieron datos, retornará el hash vacío).

```
/*
    Comparo ambos hashes. Si son idénticos, retorno Verdadero
*/
function validar_user_y_pass() {
    $user_hash = get_post_data();
    $system_hash = HASH_ACCESO;
    if($user_hash == $system_hash) {
        return True;
    }
}

/*
    Esta será la función principal, que será llamada tras enviar el
    formulario. Si los datos ingresados coinciden con los esperados,
    inicio la sesión del usuario.
    Finalmente, redirijo al usuario a la página restringida por defecto
    (posteriormente crearemos una función que se encargue de ello)
*/
function login() {
    $user_valido = validar_user_y_pass();
    if($user_valido) {
        $_SESSION['login_date'] = time();
    }
    goto_page(PAGINA_RESTRINGIDA_POR_DEFECTO);
}
```

La función `login()` genera una variable de sesión llamada **login_date** cuyo valor es la marca de tiempo actual (al momento del logueo).

Utilizaremos luego esa variable de sesión, para verificar la inactividad del usuario.

ADVERTENCIA:

Nótese que no se crean otras variables de sesión, ni **tampoco se almacenan datos privados como usuario o clave**, ni sus hashes MD5.

Funciones para destruir la sesión del usuario

Una sola función será necesario para cumplir este propósito. Esta función, luego será invocada por un archivo al que llamaremos `salir.php` (que luego crearemos).

Esta función, solo se encargará de destruir la sesión del usuario tal cual se indicó cuando hablamos sobre como desconectar a un usuario del sistema, y finalmente, redirigirá al usuario al formulario de login, haciendo uso de una función que crearemos más adelante.

```
# Destruir sesión
function logout() {
    unset($_SESSION);
    $datos_cookie = session_get_cookie_params();
    setcookie(session_name(), NULL, time()-999999, $datos_cookie["path"],
        $datos_cookie["domain"], $datos_cookie["secure"],
        $datos_cookie["httponly"]);
    goto_page(PAGINA_LOGIN);
}
```

Funciones para verificación y validación de sesiones

Primero, me encargaré de obtener los datos del último acceso del usuario. Para eso, voy a recurrir a la variable de sesión llamada `login_date`:

```
/*
    Primero verifico que la variable de sesión login_date, existe. De ser
    así, obtengo su valor y lo retorno.
    Si no existe, retornará el entero 0
*/
function obtener_ultimo_acceso() {
    $ultimo_acceso = 0;
    if(isset($_SESSION['login_date'])) {
        $ultimo_acceso = $_SESSION['login_date'];
    }
    return $ultimo_acceso;
}
```

El siguiente paso, será verificar el tiempo de inactividad de la sesión y actualizarlo:

```
/*
    Esta función, retornará el estado de la sesión:
    sesión inactiva, retornará False mientras que sesión activa,
    retornará True.
    Al mismo tiempo, se encarga de actualizar la variable de sesión
    login_date, cuando la sesión se encuentre activa
*/
```

```
*/

function sesion_activa() {
    $estado_activo = False;
    $ultimo_acceso = obtener_ultimo_acceso();

    /*
        Establezco como límite máximo de inactividad (para mantener la
        sesión activa), media hora (o sea, 1800 segundos).
        De esta manera, sumando 1800 segundos a login_date, estoy definiendo
        cual es la marca de tiempo más alta, que puedo permitir al
        usuario para mantenerle su sesión activa.
    */
    $limite_ultimo_acceso = $ultimo_acceso + 1800;

    /*
        Aquí realizo la comparación. Si el último acceso del usuario,
        más media hora de gracia que le otorgo para mantenerle activa
        la sesión, es mayor a la marca de hora actual, significa entonces
        que su sesión puede seguir activa. Entonces, le actualizo la marca
        de tiempo, renovándole la sesión
    */
    if($limite_ultimo_acceso > time()) {
        $estado_activo = True;
        # actualizo la marca de tiempo renovando la sesión
        $_SESSION['login_date'] = time();
    }
    return $estado_activo;
}
```

Finalmente, crearemos una pequeña función, que llame a la anterior, y en caso de recibir como resultado que la sesión está inactiva, desconectará al usuario del sistema.

```
# Verificar sesión
function validar_sesion() {
    if(!sesion_activa()) {
        logout();
    }
}
```

Esta función, será la que invocaremos desde todas y cada una de las páginas, a las cuales querramos restringir su acceso.

La función que redirige a los usuarios

Como última función, crearemos aquella pendiente, de la cual hemos hablado, que se encargará de redirigir a los usuarios a otra página de nuestro sistema, utilizando la función `header()` de PHP.

```
# redirigir al usuario
function goto_page($pagina) {
    header("Location: $pagina");
}
```

Pasos finales

Con todo esto, tenemos “el alma” de nuestro sistema de logeo. Ahora solo nos resta crear los archivos pendientes:

iniciar.php

Lamará a la función `login()`. Es quien recibe los datos desde el formulario.

```
<?php
require_once("funciones.php");
login();
?>
```

salir.php

Lamará a la función `logout()`. Será llamado cada vez que el usuario elija desconectarse del sistema (tendremos que proveerle del link correspondiente)

```
<?php
require_once("funciones.php");
logout();
?>
```

pagina_de_muestra.php

Es solo a modo de ejemplo. Emula a cualquier página restringida de nuestro sistema, la cual deberá invocar a la función `validar_sesion()`. Es decir, en esta página (así como en cualquier otra página restringida), colocaremos todo el contenido de acceso privado, ya sea puramente PHP, como HTML, una mezcla de ambos o mejor aún, código PHP que invoque y renderize el HTML.

Todo, absolutamente todo el contenido de estas páginas restringidas, solo será visible al usuario si tiene la sesión iniciada y activa. De lo contrario, el contenido estará seguro y no será mostrado a usuarios sin sesión iniciada o con sesión inactiva.

```
<?php
require_once("funciones.php");
validar_sesion();
?>

<!-- contenido de ejemplo -->
<b>Bienvenido usuario registrado!</b> (<a href="salir.php">Desconectarse</a>)
```

Descarga los archivos de este ejemplo desde la Web del curso, en:

<http://taller-de-php.eugeniabahit.com>

O mira el **ejemplo en marcha** ingresando en:

http://taller-de-php.eugeniabahit.com/Ejemplos-En-Marcha/uso-de-sesiones/pagina_de_muestra.php

Tratamiento y control de errores

En algún momento, mientras ejecutábamos algún código PHP, habremos podido notar con bastante frecuencia, que PHP nos arrojaba algún tipo de mensaje, cuando nuestro script, contenía algún tipo falla.

Estos mensajes de error pueden ser diversos tipos (**funciones obsoletas, avisos, advertencias, errores fatales** -entre otros-) y **su visibilidad, se puede configurar tanto desde el archivo de configuración de PHP (php.ini) como en tiempo de ejecución**, mediante la función `ini_set()`.

Sin embargo, **antes de decidir qué tipos de error deben producir mensajes visibles o no, se debe considerar primero, en que entorno estamos trabajando.**

Como hemos hablado en varias ocasiones, por cuestiones de seguridad, cuanto menos información sobre el comportamiento interno de nuestra aplicación, le demos al usuario, más a salvo estará. Por lo tanto, como regla general, **debemos ocultar todos los errores, cuando la aplicación esté corriendo en un entorno de producción.**

Tipos de errores

Como se comentó en párrafos anteriores, PHP puede emitir distintos tipos de errores, que van desde el aviso de funciones obsoletas hasta errores fatales.

Estos tipos de errores, poseen asociadas constantes predefinidas, que podrán ser pasadas posteriormente, como segundo parámetro a la función `ini_set()` a fin de configurar errores en tiempo de ejecución.

Entre las **constantes predifinidas** que más nos ocupan, podemos encontrar las siguientes:

CONSTANTE	DESCRIPCIÓN	Interrumpe el Script
E_ERROR	Errores fatales en tiempo de ejecución.	SI
E_WARNING	Advertencias no fatales en tiempo de ejecución	NO
E_NOTICE	Avisos en tiempo de ejecución, que indican que el script encontró algo que podría ser un error u ocurrir en el curso normal de un script	NO
E_STRICT	Sugerencias de cambios al código para ampliar la compatibilidad con versiones posteriores de PHP	NO
E_DEPRECATED	Avisos en tiempo de ejecución, sobre funciones obsoletas	NO
E_ALL	Todos los anteriores (excepto E_STRICT, que recién es incluido en E_ALL, desde la versión 5.4 de PHP)	SI

Estos niveles de error, pueden utilizarse de forma combinada, mediante los siguientes **operadores**:

OPERADOR	SIGNIFICADO	USO
	"o" (alternativa)	E_NOTICE E_DEPRECATED (E_NOTICE o E_DEPRECATED)
&	"y" (concatenación)	E_NOTICE & E_DEPRECATED (E_NOTICE y E_DEPRECATED)
~	Negación	E_ALL & ~E_NOTICE (E_ALL pero no E_NOTICE)
^	Negación (en tiempo de ejecución)	E_ALL ^ E_NOTICE (E_ALL pero no E_NOTICE)

Configurando errores en tiempo de ejecución

En tiempo de ejecución, mediante la función `ini_set()` de PHP, se pueden establecer ciertas directivas de configuración, relativas a los errores y registro de los mismos.

Entre las **directivas más comunes**, podemos encontrar:

DIRECTIVA	DESCRIPCIÓN / EJEMPLO	VALOR POR DEFECTO
error_reporting	Establece que tipo de errores son reportados <code>ini_set('error_reporting', E_ALL & E_DEPRECATED);</code>	E_ALL & ~E_NOTICE
display_errors	Determina si se deben mostrar o no los errores en pantalla <code>ini_set('display_errors', '0');</code>	String 1
track_errors	Indica si el último error encontrado, estará disponible a través de la variable <code>\$php_errormsg</code> <code>ini_set('track_errors', '0');</code>	String off
error_prepend_string	Cadena a imprimir antes del mensaje de error <code>ini_set('error_prepend_string', 'Error encontrado:');</code>	NULL
error_append_string	Cadena a imprimir después del mensaje de error <code>ini_set('error_prepend_string', '<hr/>');</code>	NULL

Un ejemplo sencillo pero altamente productivo

Como comentamos anteriormente, cuando nuestra aplicación corriese en un entorno de producción, los errores deberían ocultarse. Sin embargo, mientras que se esté trabajando en un entorno de desarrollo, podrían estar visiblemente activos para ayudarnos a depurar nuestro código.

Una forma simple de lograr esto, es crear un archivo de configuración para la aplicación (que deba ser importado por todos los archivos de la aplicación), que decida si mostrar o no los errores, según el valor de una constante creada a tal fin, que llamaremos `PRODUCCION`, estableciendo su valor por defecto en `False` (significará que estamos en entorno de desarrollo) y al subir a producción, la setearemos en `True`:

```
<?php

const PRODUCCION = False; // en entornos de producción, cambiar a True

if(!PRODUCCION) {
    ini_set('error_reporting', E_ALL | E_NOTICE | E_STRICT);
    ini_set('display_errors', '1');
    ini_set('track_errors', 'On');
} else {
    ini_set('display_errors', '0');
}
?>
```

Utilizando el símbolo @ para silenciar errores

En PHP, es posible silenciar errores anteponiendo una arroba (@) a la instrucción que podría generar un error. Es una práctica que debe utilizarse con sumo cuidado y siempre que se quiera capturar el error (a pesar de estar silenciado), deberá establecer `track_errors` en `On` a fin de obtener dicho error mediante la variable `$php_errormsg`.

Veamos algunos ejemplos:

```
<?php
ini_set('error_reporting', E_ALL | E_NOTICE | E_STRICT);
ini_set('display_errors', '0');
ini_set('track_errors', 'On');

$archivo = @fopen('archivo_que_no_existe.txt', 'r');
if(!$archivo) {
```

```
        echo $php_errormsg;
    }
?>
```

En el ejemplo anterior, silenciamos el posible error al intentar abrir un archivo mediante `fopen()`, pero imprimimos en pantalla el mensaje de error capturado, mediante la variable `$php_errormsg` obteniendo como resultado:

```
fopen(archivo_que_no_existe.txt): failed to open stream: No such file or directory
```

Sin embargo, podríamos ocultar esta información al usuario:

```
<?php
ini_set('error_reporting', E_ALL | E_NOTICE | E_STRICT);
ini_set('display_errors', '0');
ini_set('track_errors', '0n');

$archivo = @fopen('archivo_que_no_existe.txt', 'r');
if(!$archivo) {
    echo 'Ha ocurrido un error en el sistema. Disculpe las molestias.';
}
?>
```

De esta forma, el usuario solo verá el siguiente mensaje:

```
Ha ocurrido un error en el sistema. Disculpe las molestias.
```

Trabajando con Bases de Datos MySQL

Con este capítulo, **llegamos al final del curso “PHP para Principiantes”**. Abarcando esta última unidad, ya estaremos en condiciones de crear aplicaciones funcionales de alto nivel, de complejidad media.

Sin dudas, el trabajo con bases de datos, es lo más esperado por cualquier programador que está dando sus primeros pasos, pero entonces ¿por qué dejarlo para el final? Y la respuesta a esta pregunta, es muy simple: porque las bases de datos son el “cristal” de una aplicación. Representan la parte más vulnerable de un sistema informático y de su vulnerabilidad, dependerá la estabilidad o inestabilidad de todo el sistema.

A lo largo del curso, hemos adquirido todas las técnicas, prácticas y herramientas necesarias, para saber como filtrar y securizar datos y **recién ahora, estamos listos para poder comenzar a trabajar con bases de datos, en absoluta libertad y confianza.**

¡Comencemos!

Una base de datos representa un conjunto de datos pertenecientes a un mismo contexto, que son almacenados de forma sistemática para su posterior uso. Para comprender mejor el concepto de Base de Datos, por favor, dirigirse a http://es.wikipedia.org/wiki/Base_de_datos

Acerca de MySQL

MySQL es un **servidor de Bases de Datos SQL** (Structured Query Language) que se distribuye en dos versiones:

- Una versión GPL (Software Libre)
- Otra versión privativa, llamada MySQL AB

En este curso, utilizaremos la versión estandar licenciada bajo la GNU General Public License (GPL).

Instalación y configuración de MySQL

Para instalar MySQL, por línea de comandos, escribe:

```
sudo apt-get install mysql-server mysql-client
```

Durante la instalación, el sistema te pedirá que ingreses una contraseña para la administración de MySQL. Asigna una contraseña que puedas recordar fácilmente y mantenla a salvo ya que deberás utilizarla frecuentemente.

Una vez que finalice la instalación, ejecuta el siguiente comando a fin de securizar el servidor MySQL (esta configuración, es válida también, para servidores de producción):

```
sudo mysql_secure_installation
```

A continuación, el sistema te pedirá que ingreses la contraseña actual para administración de MySQL (la del usuario root de MySQL). Ten en cuenta que la contraseña no será mostrada mientras escribes:

```
Enter current password for root (enter for none):
```

A continuación, te preguntará si deseas modificar esa contraseña. Salvo que desees modificarla, ingresa n:

```
Change the root password? [Y/n] n
```

Ahora la pregunta, será si deseas eliminar usuarios anónimos. Responde que sí:

```
Remove anonymous users? [Y/n] Y
```

Luego, te preguntará si desees desabilitar el acceso remoto al usuario root de MySQL. Por supuesto, responde que sí:

```
Disallow root login remotely? [Y/n] Y
```

La siguiente pregunta será si deseas eliminar la base de datos de prueba y el acceso a ella. También responde que sí:

```
Remove test database and access to it? [Y/n] Y
```

Finalmente, te preguntará si deseas recargar las tablas de privilegios (esto es para asegurar que todos los cambios realizados surjan efecto). Entonces, responde sí, por última vez:

```
Reload privilege tables now? [Y/n] Y
```

Iniciar, reiniciar y detener el servidor MySQL

En ocasiones necesitarás iniciar, reiniciar o detener el servidor de bases de datos, MySQL.

Las **opciones** disponibles son:

stop	detiene el servidor
start	inicia el servidor
restart	reinicia el servidor

Para iniciar, reiniciar o detener el servidor, deberás **ejecutar el siguiente comando**, seguido de la opción deseada:

```
sudo /etc/init.d/mysql opcion_deseada
```

Lógicamente reemplazando `opcion` por `stop`, `start` o `restart` según si deseas parar, iniciar o reiniciar el servidor.

Administración de MySQL

Una vez que comencemos a utilizar bases de datos, necesitarás poder acceder a las opciones de administración de las mismas. Por lo tanto, te recomiendo tener siempre a mano este capítulo, para poder consultarlo con frecuencia.

Conectarse y desconectarse al servidor

Para conectarte deberás ejecutar el siguiente comando:

```
mysql -u root -p
```

A continuación, deberás ingresar la contraseña del root de MySQL (no es la del root del SO. Es la que hemos configurado durante la instalación de MySQL).

Las `-u` y `-p` significan usuario y password respectivamente.

Te aparecerá un shell interactivo para MySQL:

```
mysql>
```

Allí podremos escribir los comandos necesarios para administrar el servidor de bases de datos.

Comandos para administrar MySQL desde el shell interactivo

La siguiente tabla describe los comandos de uso frecuente que necesitarás para administrar el servidor de bases de datos desde el shell interactivo.

Es una buena idea, imprimir esta tabla para tenerla siempre a

mano :)

COMANDO	DESCRIPCIÓN
show databases;	Muestra todas las bases de datos creadas en el servidor
use nombre_de_la_base_de_datos;	Indicar que vas a comenzar a utilizar la base de datos elegida
create database nombre_de_la_db;	Crear una nueva base de datos
quit	Salir del shell interactivo

Sobre el lenguaje SQL

SQL -siglas de *Structured Query Language*-, es el **lenguaje de consultas a bases de datos**, que nos permitirá crear, modificar, consultar y eliminar tanto bases de datos como sus tablas y registros, desde el shell interactivo de MySQL y también desde PHP.

Como todo **lenguaje informático**, posee su propia **sintaxis**, **tipos de datos** y **elementos**.

En este curso, **abordaremos los conceptos básicos sobre SQL** que nos permitan desarrollar aplicaciones de media complejidad, sin profundizar en el lenguaje en sí, sino solo en aquellos **aspectos mínimamente necesarios relacionados con MySQL**.

Tipos de datos más comunes (recomendados)

La siguiente tabla, muestra los tipos de datos más comunes, aceptados por versiones la versión 5.0.3 o superior, de MySQL.

Tipo de dato	Denominación	Especificaciones	Ejemplo
Entero	INT(N)	N = cantidad de dígitos	INT(5)
Número decimal	DECIMAL(N, D)	N = cantidad de dígitos totales D = cantidad de decimales	DECIMAL(10, 2)
Booleano	BOOL		BOOL
Fecha	DATE		DATE
Fecha y hora	DATETIME		DATETIME
Fecha y hora automática	TIMESTAMP		TIMESTAMP
Hora	TIME		TIME
Año	YEAR(D)	D = cantidad de dígitos	YEAR(4)

		(2 o 4)	
Cadena de longitud fija	CHAR(N)	N = longitud de la cadena - entre 0 y 255	CHAR(2)
Cadena de longitud variable	VARCHAR(N)	N = longitud máxima de la cadena - entre 0 y 65532	VARCHAR(100)
Bloque de texto de gran longitud variable	BLOB		BLOB

Sintáxis básica de las sentencias SQL

Una sentencia SQL (denominada “*query*” en la jerga informática), es una **instrucción** escrita en lenguaje SQL. Veremos aquí, el tipo de sentencias más habituales.

Crear tablas en una base de datos

Sintaxis:

```
CREATE TABLE nombre_de_la_tabla(
    nombre_del_campo TIPO_DE_DATO,
    nombre_de_otro_campo TIPO_DE_DATO
);
```

Ejemplo:

```
CREATE TABLE productos(
    producto VARCHAR(125),
    descripcion BLOB,
    precio DECIMAL(6, 2),
    en_stock BOOL
);
```

Explicación:

```
CREATE TABLE productos
```

Crear una nueva tabla llamada “productos”

```
producto VARCHAR(125),
```

Crear un campo llamado producto, de tipo cadena de texto de longitud variable, con una longitud máxima de 125 caracteres

```
descripcion BLOB,
```

Crear un campo llamado descripción, de tipo bloque de texto de gran longitud

```
precio DECIMAL(6, 2),
```

Crear un campo precio de tipo numérico de longitud máxima de 6 dígitos de los cuales, solo 2 pueden ser decimales

```
en_stock BOOL
```

Crear un campo llamado “en_stock” del tipo booleano

Insertar datos en una tabla

Sintaxis:

```
INSERT INTO
    nombre_de_la_tabla(campo1, campo2, campo10..)
VALUES(dato1, dato2, dato10..);
```

Ejemplo:

```
INSERT INTO
    productos(producto, precio, en_stock)
VALUES('Bolsa de dormir para alta montaña', 234.65, TRUE);
```

Explicación:

```
INSERT INTO
    productos(producto, precio, en_stock)
```

Insertar un nuevo registro en los campos producto, precio y en_stock de la tabla productos

```
VALUES('Bolsa de dormir para alta montaña', 234.65, TRUE);
```

Con los valores “Bolsa de dormir para alta montaña”, 234.65 y verdadero, respectivamente en cada uno de los campos indicados

Seleccionar registros

Sintaxis:

```
SELECT    campo1, campo2, campo10
FROM      tabla;
```

Ejemplo:

```
SELECT    producto, precio
```

```
FROM      productos;
```

Explicación:

```
SELECT     producto, precio
```

Seleccionar los campos producto y precio

```
FROM      productos;
```

De la tabla productos

Modificar registros

Sintaxis:

```
UPDATE     tabla
SET        campo1 = valor,
           campo2 = valor,
           campo10 = valor;
```

Ejemplo:

```
UPDATE     productos
SET        en_stock = FALSE,
           precio = 0;
```

Explicación:

```
UPDATE     productos
```

Actualizar la tabla productos

```
SET        en_stock = FALSE,
```

Modificar el campo en_stock por falso

```
precio = 0;
```

y el campo precio a 0

Eliminar registros

Sintaxis:

```
DELETE FROM tabla;
```

Ejemplo:

```
DELETE FROM productos;
```

Explicación:

```
DELETE FROM productos;
```

Eliminar todos los registros de la tabla productos

Consultas avanzadas

Si bien no veremos aquí consultas realmente complejas, ya que el curso se basa en el lenguaje de programación PHP y no, en el lenguaje de consulta SQL, haremos un rápido paseo, por las opciones disponibles en SQL para sentencias más complejas que las anteriores.

La cláusula WHERE

Las sentencias en SQL, se componen de **cláusulas**. Y WHERE es una de ellas. La **cláusula WHERE** nos permite **filtrar registros en una sentencia SQL**.

Esta cláusula, funciona de forma similar a la comparación de expresiones en PHP, utilizando los siguientes **operadores de comparación**:

>	mayor que	<	menor que
=	igual que	<>	distinto que
>=	mayor o igual que	<=	menor o igual que
BETWEEN n1 AND n2		entre n1 y n2	
IS NULL TRUE FALSE		es nulo es verdadero es falso	
IN(valor1, valor2, va...)		contiene	

Por supuesto, también admite **operadores lógicos**:

AND (y)	NOT (negación)	OR (o)
---------	----------------	--------

Veamos algunos ejemplos:

Seleccionar productos donde precio sea menor que 1000:

```
SELECT    producto,  
          precio  
FROM      productos  
WHERE     precio < 1000;
```

Aumentar el 10% del precio de los productos, que actualmente se encuentren entre 150 y 200:

```
UPDATE    productos  
SET       precio = (precio * 1.10)  
WHERE     precio BETWEEN 150 AND 200;
```

Seleccionar productos donde en_stock no sea falso

```
SELECT    producto,  
          precio  
FROM      productos  
WHERE     en_stock IS NOT FALSE;
```

Eliminar productos cuyos precios sean 100, 200 y/o 300 y además, en_stock sea falso o producto sea nulo:

```
DELETE  
FROM      productos  
WHERE     precio IN(100, 200, 300)  
AND       (en_stock IS FALSE  
OR        producto IS NULL);
```

Modificar en_stock a verdadero donde precio sea menor que 50 y producto no sea nulo:

```
UPDATE    productos  
SET       en_stock = TRUE  
WHERE     precio < 50  
AND       en_stock IS NOT NULL;
```

Ordenando consultas: la cláusula ORDER BY

Es posible además, ordenar los resultados de una consulta, en forma **ascendente** (ASC) o **descendente** (DESC):

```
SELECT    producto,
          descripcion,
          precio

FROM      productos

WHERE     precio BETWEEN 1 AND 50
AND       en_stock IS NOT FALSE

ORDER BY  precio DESC;
```

También es posible, ordenar los resultados de la consulta, por más de un campo:

```
SELECT    producto,
          descripcion,
          precio

FROM      productos

WHERE     precio BETWEEN 1 AND 50
AND       en_stock IS NOT FALSE

ORDER BY  precio DESC,
          producto ASC;
```

Alias de tablas y campos

Otra posibilidad que nos da el lenguaje SQL, es utilizar alias para el nombre de los campos y las tablas. Estos alias se asignan mediante la palabra clave reservada, **AS**:

```
SELECT    producto      AS 'Nombre del Producto',
          descripcion    AS Detalles,
          precio         AS Importe

FROM      productos     AS p

WHERE     precio BETWEEN 1 AND 50
AND       en_stock IS NOT FALSE

ORDER BY  precio DESC,
```

```
producto ASC;
```

Nótese que los **alias** que contengan caracteres **extraños**, deben ser **encerrados entre comillas simples**

Funciones del lenguaje SQL de MySQL

Es posible también, utilizar diversas funciones propias del lenguaje SQL -ya sea estandar o de MySQL- a fin de poder obtener los datos con cierto formato. Veremos aquellas de uso más frecuente.

Contar la cantidad de registros: COUNT()

```
SELECT      COUNT(producto)  AS Cantidad
FROM        productos;
```

Sumar totales: SUM()

```
SELECT      SUM(precio)      AS Total
FROM        productos;
```

Concatenar cadenas: CONCAT()

```
SELECT      producto,
FROM        CONCAT('USD ', precio, '.-')  AS Precio
productos;
```

Nótese que las cadenas de caracteres deben encerrarse entre comillas simples y que el operador de concatenación para esta función, es la coma.

Convertir a minúsculas y mayúsculas: LCASE() y UCASE()

```
SELECT    UCASE(producto),
FROM      LCASE(descripcion)
productos;
```

Reemplazar datos: REPLACE()

```
SELECT    REPLACE(descripcion, '\n', '<br/>')    AS Descripcion
FROM      productos;
```

Reemplaza '\n' por '
'

Obtener los primeros o últimos caracteres: LEFT() y RIGHT()

```
SELECT    LEFT(producto, 50)
FROM      productos;
```

Redondear números: ROUND()

```
SELECT    ROUND(precio, 2)
FROM      productos;
```

Retornará los precios con 2 decimales

Obtener solo la fecha de un campo DATETIME o TIMESTAMP: DATE()

```
SELECT    DATE(campo_datetime)
FROM      tabla;
```

Obtener una fecha formateada: DATE_FORMAT()

```
SELECT    DATE_FORMAT(campo_fecha, '%d/%m/%Y')
FROM      tabla;
```

Aplican los mismos patrones de formato de fecha que en PHP

Obtener el registro con el valor máximo y mínimo: MAX() y MIN()

```
SELECT    MAX(precio)
FROM      productos;
```

Retorna el producto con el precio más caro

```
SELECT    MIN(precio)
FROM      productos;
```

Retorna el producto con el precio más barato

Optimización de bases de Datos

A continuación, encontrarás una lista de consejos que SIEMPRE debes seguir, al momento de crear nuevas tablas y escribir sentencias SQL.

Todos los registros deben tener un ID único

Cuando crees tablas, asígnales un campo `id` de tipo autonumérico incremental y establéclo como índice primario. Cuando agregues registros, este campo se completará automáticamente, con un número incremental, que te servirá para optimizar tus consultas y contar con un campo que te permita reconocer el registro como único.

```
CREATE TABLE productos(
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
    producto VARCHAR(125)
);
```

El campo `id`, será como cualquier otro y lo podrás seleccionar en un `SELECT` o utilizarlo en cualquier cláusula `WHERE`.

Crear índices en las tablas

Todas las tablas deben tener un índice. El índice se asigna a uno o más campos, y es utilizado por SQL para filtrar registros de forma más rápida. Debes crear índices con precaución, ya que de la misma forma que se aceleran las consultas, se retrasa la inserción y actualización de registros, puesto que la base de datos, deberá actualizar los índices cada vez que se agreguen o modifiquen datos.

Cuando una consulta es ejecutada, MySQL tratará de encontrar primero la respuesta en los campos índice, y lo hará en el orden que los índices hayan sido creados.

¿Cuándo agregar índices? Cuando vayas a utilizar una combinación de campos en la cláusula WHERE. Por ejemplo, si filtrarás a menudo, los datos de la tabla producto por su campo precio y en_stock, que precio y en_stock sean un índice de múltiples campos:

```
CREATE TABLE productos(  
  id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,  
  producto VARCHAR(125),  
  precio DECIMAL(10, 2),  
  en_stock BOOL,  
  descripcion BLOB,  
  INDEX(precio, en_stock)  
);
```

Indica cuáles campos no pueden ser nulos

SQL te da la posibilidad de indicar qué campos no pueden estar nulos. Indicar que un campo no debe estar nulo, te ayudará a no almacenar registros defectuosos en tu base de datos.

```
CREATE TABLE productos(  
  id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,  
  producto VARCHAR(125) NOT NULL,
```

```
precio DECIMAL(10, 2) NOT NULL,  
en_stock BOOL,  
descripcion BLOB NOT NULL,  
INDEX(precio, en_stock)  
);
```

Utiliza el motor InnoDB

El motor de bases de datos InnoDB, te permitirá crear tablas relaciones optimizando su rendimiento. Al momento de crear tus tablas, indica que utilizarás el motor InnoDB:

```
CREATE TABLE productos(  
    id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    producto VARCHAR(125) NOT NULL,  
    precio DECIMAL(10, 2) NOT NULL,  
    en_stock BOOL,  
    descripcion BLOB NOT NULL,  
    INDEX(precio, en_stock)  
) ENGINE=InnoDB;
```

Obtener mayor información

- [Descarga el Manual de MySQL](#)
- [Aprende sobre el lenguaje SQL](#), gratis en [1KeyData](#)

Trabajando con MySQL desde PHP

Antes de comenzar, deseo aclarar que -por cuestiones de seguridad- en nuestros archivos PHP, solo nos conectaremos a una base de datos, para realizar consultas de selección, modificación, inserción y eliminación de registros en tablas y bases de datos existentes.

No crearemos tablas ni bases de datos desde nuestros archivos PHP, sino que lo haremos desde el administrador de MySQL por línea de comandos.

A fin de poder trabajar con los ejemplos de este capítulo, comenzaremos creando una nueva base de datos, con la tabla que necesitaremos.

Ejecuta las siguientes sentencias, desde el Shell interactivo de MySQL:

```
CREATE DATABASE curso_php;
USE curso_php;

CREATE TABLE usuarios(
  id INT(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(24) NOT NULL,
  email VARCHAR(100) NOT NULL,
  password VARCHAR(40) NOT NULL,
  suspendido BOOL,
  fecha_registro TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  INDEX(username, password, suspendido)
) ENGINE=InnoDB;
```

AVISO:

PHP cuenta con **dos tipos de conectores** MySQL: **mysql** y **mysqli**, disponible desde la versión 5. Si bien mysqli y más potente, seguro y estable que mysql, en este capítulo, veremos como utilizar ambos.

Antes de continuar, vamos a instalar el módulo de MySQL para PHP:

```
sudo apt-get install php5-mysql
```

MySQL desde PHP con el conector mysql

Veremos como utilizar el conector `mysql`, **a solo efecto educativo**, pero para nuestras aplicaciones, utilizaremos `mysqli`.

Para trabajar con bases de datos desde PHP con el conector `mysql`, el procedimiento, consiste en los siguientes pasos:

1. Conectarse a la base de datos
2. Seleccionar la base de datos a utilizar
3. Ejecutar una consulta
4. Capturar los resultados
5. Liberar los resultados
6. Cerrar la conexión

Conectarse a la base de datos

```
$host = 'localhost';  
$usuario = 'root';  
$clave = 'contraseña';  
  
$conn = mysql_connect($host, $usuario, $clave)  
        or die('No me pude conectar a la base de datos');
```

Seleccionar una base de datos

```
$db = 'curso_php';  
mysql_select_db($db) or die('No pude seleccionar la base de datos');
```

Ejecutar una consulta simple

```
$sql = "  
    INSERT INTO    usuarios  
                  (username, email, password)  
    VALUES        ('juanperez',  
                    'jperez@algundominio.ext',  
                    '26ec07ef61f135494b79a13674a9a4ae')  
";  
$result = mysql_query($sql) or die('No pude ejecutar la consulta');
```

Ejecutar una consulta de selección múltiple y capturar sus resultados

```
$sql = "  
    SELECT id, username, email  
    FROM    usuarios  
";  
$result = mysql_query($sql) or die('No pude ejecutar la consulta');
```

Capturamos el array con los resultados

```
while($registros[] = mysql_fetch_array($result));
```

El array devuelto (\$registros) será del tipo multidimensional, con un formato como el siguiente:

```
$registros = array(  
    array('campo1'=>'valor', 'campo2'=>'valor', 'campo7'=>'valor'),  
    array('campo1'=>'valor', 'campo2'=>'valor', 'campo7'=>'valor'),  
);
```

Es decir, que será un array, conteniendo otro array por cada registro encontrado.

Liberar los resultados

```
mysql_free_result($result);
```

Cerrar la conexión

```
mysql_close($conn);
```

Algunos ejemplos concretos

Consulta de selección

```
# Preparo los datos para conectarme a la DB
$host = 'localhost';
$usuario = 'root';
$clave = 'contraseña';
$db = 'curso_php';

# me conecto a la DB
$conn = mysql_connect($host, $usuario, $clave)
    or die('No me pude conectar a la base de datos');

# Selecciono la DB a utilizar
mysql_select_db($db) or die('No pude seleccionar la base de datos');

# Preparo la sentencia SQL
$sql = "
    SELECT id, username, email
    FROM    usuarios
    WHERE   suspendido IS NOT TRUE
";
# Ejecuto la consulta
$result = mysql_query($sql) or die('No pude ejecutar la consulta');

# Capturo los resultados
while($registros[] = mysql_fetch_array($result));

# Libero los resultados
mysql_free_result($result);

# Cierro la conexión
mysql_close($conn);

# Imprimo los resultados
print_r($registros);
```

Insertar varios registros en un solo paso

```
# Preparo los datos para conectarme a la DB
$host = 'localhost';
$usuario = 'root';
$clave = 'contraseña';
$db = 'curso_php';

# me conecto a la DB
$conn = mysql_connect($host, $usuario, $clave)
```

```
        or die('No me pude conectar a la base de datos');

# Selecciono la DB a utilizar
mysql_select_db($db) or die('No pude seleccionar la base de datos');

# Preparo la sentencia SQL
$sql = "
    INSERT INTO    usuarios (username, email, password)
    VALUES        ('javier75',
                    'javi75@algundominio.ext',
                    '26ec07ef61f135494b79a13674a9a4ae'),

                    ('noelia',
                    'noe@algundominio.ext',
                    '26ec07ef61f135494b79a13674a9a4ae'),

                    ('ana_AR',
                    'anita@algundominio.ext',
                    '26ec07ef61f135494b79a13674a9a4ae')
";
# Ejecuto la consulta
mysql_query($sql) or die('No pude ejecutar la consulta');

# Cierro la conexión
mysql_close($conn);
```

MySQL desde PHP con el conector mysqli

Como comentamos anteriormente, `mysqli` es un conector mucho más seguro y potente que `mysql`. Si bien la forma más acertada de implementar este conector, es a través de *objetos* (es decir, utilizando el paradigma de la programación orientada a objetos), nos concentraremos solo en estilo por procedimientos, ya que ésta, es la técnica de programación utilizada en este curso inicial.

VENTAJA DE UTILIZAR EL CONECTOR MYSQLI

En el caso de `mysqli`, contamos con una gran ventaja al momento de preparar nuestras sentencias SQL: podemos utilizar “comodines”, para indicar al conector, qué datos deben ser reemplazados dinámicamente. De esta forma, **el propio conector se encargará de filtrar los datos dinámicos, obteniendo consultas mucho más seguras.**

Por consiguiente, para trabajar con el conector `mysqli`, seguiremos estos pasos:

- 1. Conectarse a la base de datos y seleccionar la DB a utilizar**
- 2. Preparar la consulta**
3. Ejecutar una consulta
4. Capturar los resultados
- 5. Cerrar consulta**
6. Cerrar la conexión

Abrir una conexión mediante mysqli

```
# Preparar las variables con los datos de conexión
$host = 'localhost';
$usuario = 'root';
$password = 'contraseña';
$db = 'curso_php';

# Conectarse a la base de datos
$conn = mysqli_connect($host, $usuario, $password, $db);
```

Preparar la consulta

Preparar una consulta para trabajar mediante mysqli, requerirá de algunos cuantos pasos que debemos seguir, cuidada y ordenadamente.

Primero, preparamos la sentencia. Pero a diferencia de la vez anterior (con mysql), los datos que relacionados con los registros (ya sean datos a insertar o coincidencias establecidas en la cláusula where), no los pondremos. En su lugar, utilizaremos el **comodín ?**:

```
$sql = "
    INSERT INTO    usuarios (username, email, password)
    VALUES        (?, ?, ?)
";
```

Luego, definiremos los datos dinámicos mediante variables:

```
$username = 'juan-perez';
$email = 'juan_perez@algundominio.ext';
$password = '26ec07ef61f135494b79a13674a9a4ae';
```

Luego, le indicamos a mysqli que inicie la preparación para la consulta:


```
$pre = mysqli_prepare($conn, $sql);
```

Y finalmente, le decimos que una los parámetros a la consulta, indicándole el tipo de datos correspondiente:

```
mysqli_stmt_bind_param($pre, "sss", $username, $email, $password);
```

El segundo parámetro de la función `mysqli_stmt_bind_param` (sss) indica el tipo de datos correspondiente a los parámetros (nuestras variables). La “s” significa string. Tres “s”, significan que los tres datos son de tipo string.

Otros **tipos de datos soportados** son:

(s) string

(i) entero

(d) doble/decimal

(b) blob

El equivalente para otros tipos de datos no listados arriba, es:

Fechas y horas
Booleanos

son cadenas de texto. Por lo tanto, se utiliza
es un entero (0: False, 1: True). Utilizar

s
i

Ejecutar la consulta

```
mysqli_stmt_execute($pre);
```

Cerrar la consulta

```
mysqli_stmt_close($pre);
```

Cerrar la conexión

```
mysqli_close($conn);
```

Ejemplo de inserción completo

```
# Preparar las variables con los datos de conexión
$host = 'localhost';
$usuario = 'root';
$sclave = 'contraseña';
$db = 'curso_php';

# Conectarse a la base de datos
$conn = mysqli_connect($host, $usuario, $sclave, $db);

# Preparo la sentencia con los comodines ?
$sql = "
    INSERT INTO    usuarios (username, email, password)
    VALUES       (?, ?, ?)
";

# Preparo los datos que voy a insertar
$username = 'juan-perez';
$email = 'juan_perez@algundominio.ext';
$password = '26ec07ef61f135494b79a13674a9a4ae';

# Preparo la consulta
$pre = mysqli_prepare($conn, $sql);

# indico los datos a reemplazar con su tipo
mysqli_stmt_bind_param($pre, "sss", $username, $email, $password);

# Ejecuto la consulta
mysqli_stmt_execute($pre);

# PASO OPCIONAL (SOLO PARA CONSULTAS DE INSERCIÓN):
# Obtener el ID del registro insertado
$nuevo_id = mysqli_insert_id($conn);

# Cierro la consulta y la conexión
mysqli_stmt_close($pre);
mysqli_close($conn);
```

Nótese que el ejemplo de inserción, **aplica también a consultas de actualización, modificación y eliminación.**

Capturar resultados de una consulta de selección

En una consulta de selección (donde necesito capturar los

resultados devueltos), después de ejecutar la consulta (y antes de cerrarla), al igual que hicimos con el conector `mysql`, vamos a capturar los resultados, pero de forma diferente.

Primero, vamos a asociar la salida a variables. Esto es, **asociar los nombres de los campos devueltos a nombres de variables** y se logra así:

```
mysqli_stmt_bind_result($pre, $id, $username, $email);
```

Claramente, le estamos indicando al conector, que nos referiremos a los campos `id`, `username` e `email` como variables `$id`, `$username` e `$email` respectivamente.

Como a nosotros nos interesa almacenar los datos en un array, a fin de continuar manteniendo nuestra arquitectura de aislación del código PHP y HTML, nuevamente, vamos a iterar sobre los resultados, para generar un array que almacene todos los registros devueltos:

```
while(mysqli_stmt_fetch($pre)) {  
    $registros[] = array('id'=>$id,  
                        'username'=>$username,  
                        'email'=>$email);  
}
```

Como se puede ver, la diferencia es notable. Esta vez, tuvimos que encargarnos de crear el array prácticamente “a mano”.

Ejemplo completo de consultas de selección

```
# Preparar las variables con los datos de conexión  
$host = 'localhost';  
$usuario = 'root';  
$clave = 'contraseña';  
$db = 'curso_php';
```

```
# Conectarse a la base de datos
$conn = mysqli_connect($host, $usuario, $clave, $db);

# Preparo la sentencia con los comodines ?
$sql = "
    SELECT id, email
    FROM usuarios
    WHERE username = ?
    AND password = ?
";

# Preparo los datos que voy a insertar
$username = 'juan-perez';
$password = '26ec07ef61f135494b79a13674a9a4ae';

# Preparo la consulta
$pre = mysqli_prepare($conn, $sql);

# indico los datos a reemplazar con su tipo
mysqli_stmt_bind_param($pre, "ss", $username, $password);

# Ejecuto la consulta
mysqli_stmt_execute($pre);

# asocio los nombres de campo a nombres de variables
mysqli_stmt_bind_result($pre, $id, $email);

# Capturo los resultados y los guardo en un array
while(mysqli_stmt_fetch($pre)) {
    $registros[] = array('id'=>$id,
                        'email'=>$email);
}

# Cierro la consulta y la conexión
mysqli_stmt_close($pre);
mysqli_close($conn);
```

Introducción al Paradigma de la programación orientada a objetos

La orientación a objetos es un paradigma de programación que puede resultar complejo, si no se lo interpreta de forma correcta desde el inicio. Por eso, en esta primera parte, nos enfocaremos primero, en cuestiones de conceptos básicos, para luego, ir introduciéndonos de a poco, en principios teóricos elementalmente necesarios, para implementar la orientación a objetos en la práctica.

Pensar en objetos

Pensar en objetos, puede resultar -al inicio- una tarea difícil. Sin embargo, difícil no significa complejo. Por el contrario, pensar en objetos representa la mayor simplicidad que uno podría esperar del mundo de la programación. **Pensar en objetos, es simple...** aunque lo simple, no necesariamente signifique sencillo.

Y ¿qué es un objeto?

Pues, como dije antes, es “simple”. Olvidemos los formalismos, la informática y todo lo que nos rodea. Simplemente, olvida todo y concéntrate en lo que sigue. Lo explicaré de manera “simple”:

Un objeto es “una cosa”. Y, si una cosa es un sustantivo, entonces **un objeto es un sustantivo.**

Mira a tu alrededor y encontrarás decenas, cientos de objetos. Tu ordenador, es un objeto. Tú, eres un objeto. Tu llave es un objeto. El cenicero (ese que tienes frente a ti cargado de colillas de cigarrillo), es otro objeto. Tu mascota también es un objeto.

Cuando pensamos en “objetos”, **todos los sustantivos son objetos.**

Sencillo ¿cierto? Entonces, de ahora en más, solo concéntrate en pensar la vida en objetos (al menos, hasta terminar de leer este documento).

Ahora ¿qué me dices si describimos las cualidades de un objeto?

Describir un objeto, es simplemente mencionar sus cualidades. **Las cualidades son adjetivos.** Si no sabes que es un adjetivo, estamos jodidos (y mucho). Pero, podemos decir que **un adjetivo es una cualidad del sustantivo.**

Entonces, para describir “la manera de ser” de un objeto, debemos preguntarnos **¿cómo es el objeto?** Toda respuesta que comience por “el objeto es”, seguida de un adjetivo, será una cualidad del objeto.

Algunos ejemplos:

- **El objeto es** verde
- **El objeto es** grande
- **El objeto es** feo

Ahora, imagina que te encuentras frente a un niño de 2 años (niño: objeto que pregunta cosas que tú das por entendidas de forma implícita). Y cada vez que le dices las cualidades de un objeto al molesto niño-objeto, éste te pregunta: -“¿Qué es...?”, seguido del adjetivo con el cuál finalizaste tu frase. Entonces, tu le respondes diciendo “es un/una” seguido de un sustantivo. Te lo muestro con un ejemplo:

- El objeto es verde. **¿Qué es** verde? Un color.
- El objeto es grande. **¿Qué es** grande? Un tamaño.
- El objeto es feo. **¿Qué es** feo? Un aspecto.

Estos sustantivos que responden a la pregunta del niño,

pueden pasar a formar parte de una **locución adjetiva** que especifique con mayor precisión, las descripciones anteriores:

- El objeto es **de color** verde.
- El objeto es **de tamaño** grande.
- El objeto es **de aspecto** feo.

Podemos decir entonces -y todo esto, gracias al molesto niño-objeto-, que una cualidad, es un atributo (derivado de “cualidad atribuible a un objeto”) y que entonces, **un objeto es un sustantivo que posee atributos, cuyas cualidades lo describen.**

Veámoslo más gráficamente:

OBJETO (sustantivo)	ATRIBUTO (locución adjetiva)	CUALIDAD DEL ATRIBUTO (adjetivo)
(el) Objeto	(es de) color	Verde
	(es de) tamaño	Grande
	(es de) aspecto	Feo

Pero algunos objetos, también se componen de otros objetos...

Además de cualidades (locución adjetiva seguida de un adjetivo), **los objetos “tienen otras cosas”**. Estas “otras cosas”, son aquellas “pseudo-cualidades” que en vez de responder a ¿cómo es el objeto? responden a **“¿cómo está compuesto el objeto?”** o incluso, aún más simple **“¿Qué tiene el objeto?”**.

La respuesta a esta pregunta, estará dada por la frase “el objeto tiene...”, seguida de un adverbio de cantidad (uno, varios, muchos, algunos, unas cuantas) y un sustantivo.

Algunos ejemplos:

- El objeto **tiene algunas** antenas
- El objeto **tiene un** ojo
- El objeto **tiene unos cuantos** pelos

Los componentes de un objeto, también integran los atributos de ese objeto. Solo que **estos atributos**, son algo particulares: **son otros objetos que poseen sus propias cualidades**. Es decir, que estos “atributos-objeto” también responderán a la pregunta “¿Cómo es/son ese/esos/esas?” seguido del atributo-objeto (sustantivo).

Ampliemos el ejemplo para que se entienda mejor:

- El objeto tiene algunas antenas. **¿Cómo son esas** antenas?
 - Las antenas son de color violeta
 - Las antenas son de longitud extensa
- El objeto tiene un ojo. **¿Cómo es ese** ojo?
 - El ojo es de forma oval
 - El ojo es de color azul
 - El ojo es de tamaño grande
- El objeto tiene unos cuantos pelos. **¿Cómo son esos** pelos?
 - Los pelos son de color fucsia
 - Los pelos son de textura rugosa

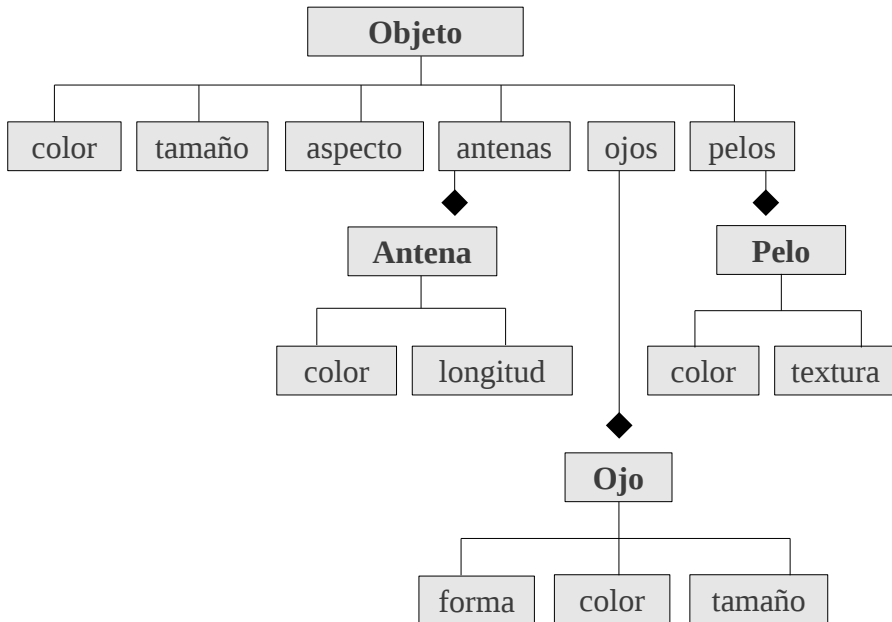
Pongámoslo más gráfico:

OBJETO (sustantivo)	ATRIBUTO-OBJETO (sustantivo)	ATRIBUTOS (locución adjetiva)	CUALIDADES DE LOS ATRIBUTOS (adjetivo)
(el) Objeto	(tiene algunas) antenas	(de) color (de) longitud	Violeta extensa
	(tiene un) ojo	(de) forma (de) color (de) tamaño	Oval azul grande
	(tiene unos cuantos) pelos	(de) color (de) textura	Fucsia rugosa

Entonces, podemos deducir que **un objeto puede tener dos tipos de atributos:**

- 1) Los que responden a la pregunta ***“¿Cómo es el objeto?”*** con la frase ***“El objeto es...”*** + adjetivo (atributos definidos por cualidades)
- 2) Los que responden a la pregunta ***“¿Qué tiene el objeto?”*** con la frase ***“El objeto tiene...”*** + sustantivo (atributos definidos por las cualidades de otro objeto)

Veámoslo aún, más gráficamente:



Viendo el gráfico anterior, tenemos lo siguiente: Un **objeto** (sustantivo) al cual hemos descrito con tres **atributos** (adjetivos) y otros tres **atributos-objeto** (sustantivos) los cuáles son a la vez, otros tres objetos (sustantivos) con sus atributos (adjetivos) correspondientes. ¿Simple, no? Ahora, compliquemos todo un poco.

Y también hay objetos que comparten características con otros objetos

Resulta ser, que nuestro Objeto, es prácticamente igual a un nuevo objeto. Es decir, que el nuevo objeto que estamos viendo, tiene absolutamente todas las características que nuestro primer objeto, es decir, tiene los mismos atributos. Pero también, tiene algunas más. Por ejemplo, este **nuevo objeto**, además de los atributos de nuestro primer objeto, **tiene un pie**. Es decir, que las características de nuestro

nuevo objeto, serán todas las del objeto original, más una nueva: pie.

Repasemos las características de nuestro nuevo objeto:

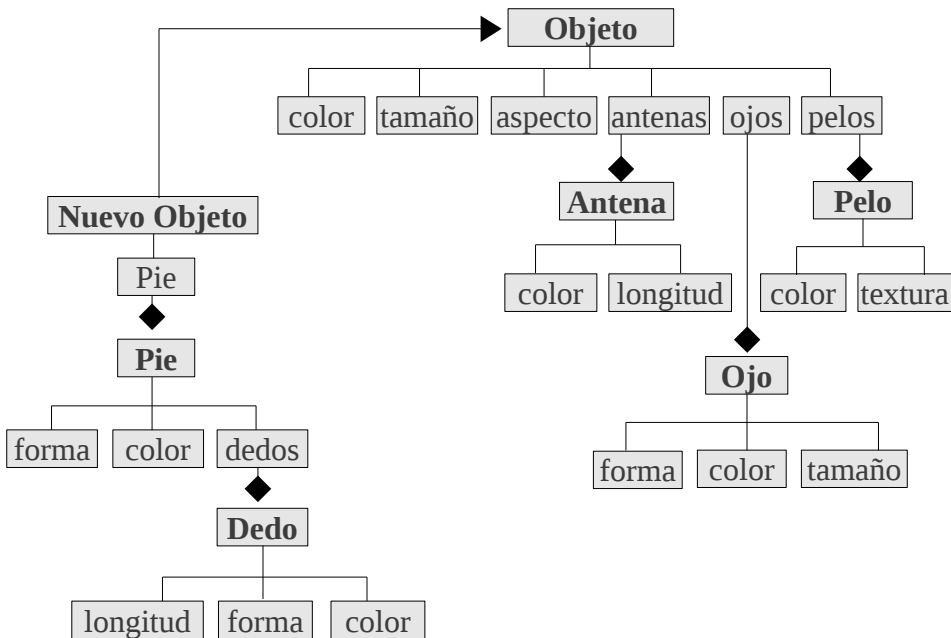
- El nuevo objeto es de color verde.
- El nuevo objeto es de tamaño grande.
- El nuevo objeto es de aspecto feo.
- El nuevo objeto tiene algunas antenas. ¿Cómo son esas antenas?
 - Las antenas son de color violeta
 - Las antenas son de longitud extensa
- El nuevo objeto tiene un ojo. ¿Cómo es ese ojo?
 - El ojo es de forma oval
 - El ojo es de color azul
 - El ojo es de tamaño grande
- El nuevo objeto tiene unos cuantos pelos. ¿Cómo son esos pelos?
 - Los pelos son de color fucsia
 - Los pelos son de textura rugosa

(nuevas características)

- El nuevo objeto tiene un pie. ¿Cómo es ese pie?
 - El pie es de forma rectangular

- El pie es de color amarillo
- El pie tiene 3 dedos. ¿Cómo son esos dedos?
 - Los dedos son de longitud mediana
 - Los dedos son de forma alargada
 - Los dedos son de color amarillo

Veamos todas las características de este nuevo, en un gráfico como lo hicimos antes.



Con mucha facilidad, podemos observar como nuestro nuevo objeto es una especie de “objeto original ampliado”. Es decir

que el nuevo objeto, es exactamente igual al objeto original (comparte todos sus atributos) pero posee nuevas características.

Está claro además, que el objeto original y el nuevo objeto, son dos objetos diferentes ¿cierto? No obstante, **el nuevo objeto es un sub-tipo del objeto original.**

Ahora sí, a complicarnos aún más.

Los objetos, también tienen la capacidad de “hacer cosas”

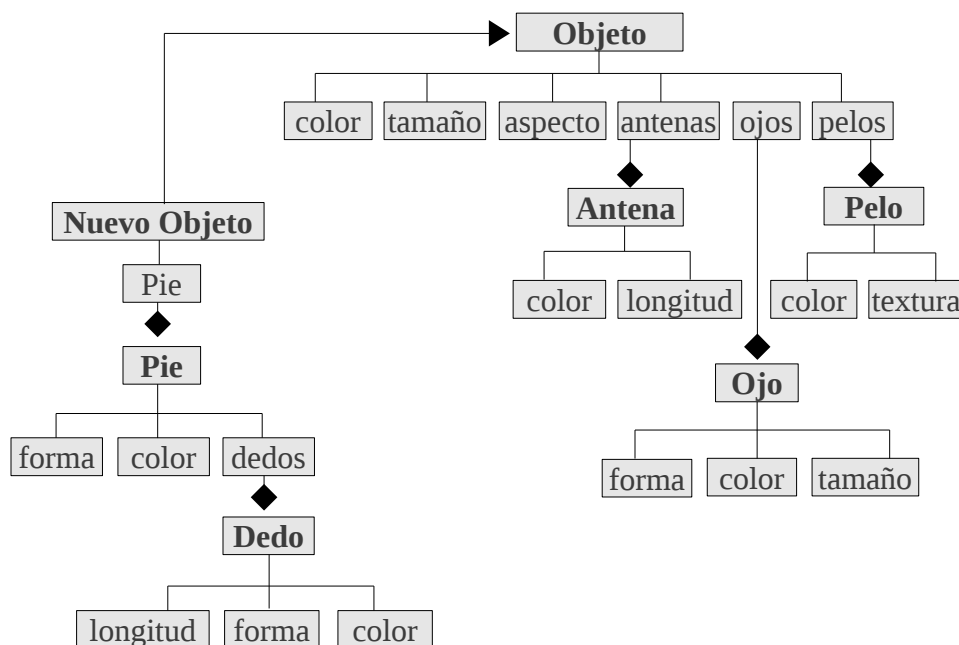
Ya describimos las cualidades de nuestros objetos. Pero de lo que no hemos hablado, es de aquellas cosas que los objetos “pueden hacer”, es decir, “cuáles son sus capacidades”.

Los objetos tiene la capacidad de realizar acciones. Las acciones, son verbos. Es decir, que para conocer las capacidades de un objeto, debes preguntarte **“¿Qué puede hacer el objeto?”** y la respuesta a esta pregunta, estará dada por todas aquellas que comiencen por la frase “el objeto puede” seguida de un verbo en infinitivo.

Algunos ejemplos:

- El objeto original **puede** flotar
- El nuevo objeto (además) **puede** saltar

Si completamos el gráfico anterior con las acciones, obtendremos lo siguiente:



Si observas el gráfico anterior, notarás que el nuevo objeto, no solo tiene los mismos atributos que el objeto original, sino que además, también puede realizar las mismas acciones que éste. Sencillo, cierto?

Ahora sí, compliquémonos del todo :)

Objetos y más objetos: la parte difícil

Si entendiste todo lo anterior, ahora viene la parte difícil. ¿Viste que esto de “pensando en objetos” viene a colación de la programación orientada a objetos? Bueno, la parte difícil es

que en la programación, todo lo que acabamos de ver, se denomina de una forma particular. Pero, la explicación es la misma que te di antes.

Al pan, pan. Y al vino, vino. Las cosas por su nombre

Cuando en el documento...	En la programación se denomina...	Y con respecto a la programación orientada a objetos es...
Hablamos de “objeto”	Objeto	Un elemento
Hablamos de “atributos” (o cualidades)	Propiedades	Un elemento
Hablamos de “acciones” que puede realizar el objeto	Métodos	Un elemento
Hablamos de “atributos-objeto”	Composición	Una técnica
Vemos que los objetos relacionados entre sí, tienen nombres de atributos iguales (por ejemplo: color y tamaño) y sin embargo, pueden tener valores diferentes	Polimorfismo	Una característica
Hablamos de objetos que son sub-tipos (o ampliación) de otros	Herencia	Una característica

Ahora, pasemos a un marco un poco más “académico”.

Programación Orientada a Objetos

La Programación Orientada a Objetos (POO u OOP por sus siglas en inglés), es un **paradigma de programación**.

Paradigma:

“Teoría cuyo núcleo central [...] suministra la **base y modelo para resolver problemas** [...] “

Definición de la Real Academia Española, vigésimo tercera edición

Cómo tal, nos enseña un método -probado y estudiado- el cual se basa en las interacciones de objetos (todo lo descrito en el título anterior, “Pensar en objetos”) para resolver las necesidades de un sistema informático.

Básicamente, este paradigma se compone de 6 elementos y 7 características que veremos a continuación.

Elementos y Características de la POO

Los **elementos** de la POO, pueden entenderse como los **“materiales”** que necesitamos para diseñar y programar un sistema, mientras que las **características**, podrían asumirse como las **“herramientas”** de las cuáles disponemos para construir el sistema con esos materiales.

Entre los **elementos principales** de la POO, podremos

encontrar a:

Clases

Las clases son los modelos sobre los cuáles se construirán nuestros objetos. Podemos tomar como ejemplo de clases, el gráfico que hicimos en la página 8 de este documento.

En PHP, una clase se define con la instrucción `class` seguida de un nombre genérico para el objeto.

```
class Objeto { }

class Antena { }

class Pelo { }

class Ojo { }
```

Sugerencia de estilos

El nombre de las clases se define en singular, utilizando CamelCase.

Propiedades

Las propiedades, como hemos visto antes, son las características intrínsecas del objeto. Éstas, se representan a modo de variables, solo que técnicamente, pasan a denominarse “propiedades”:

```
class Antena {
    public $color = "";
    public $longitud = "";
}

class Pelo {
    public $color = "";
    public $textura = "";
}
```

```
}

class Ojo {
    public $forma = "";
    public $color = "";
    public $tamanio = "";
}

class Objeto {
    public $color = "";
    public $tamanio = "";
    public $aspecto = "";
    public $antenas = array(); # será otro objeto
    public $ojos = array(); # será otro objeto
    public $pelos = array(); # será otro objeto
}
```

Estilos:

Las propiedades se definen de la misma forma que las variables (aplican las mismas reglas de estilo).

Métodos

Los métodos son “funciones” (como las que utilizamos en la programación estructurada), solo que técnicamente se denominan métodos, y representan acciones propias que puede realizar el objeto (y no otro):

```
class Objeto {
    public $color = "";
    public $tamanio = "";
    public $aspecto = "";
    public $antenas = array(); # será otro objeto
    public $ojos = array(); # será otro objeto
    public $pelos = array(); # será otro objeto

    function flotar() { }
}
```

Objeto

Las clases por sí mismas, no son más que modelos que nos servirán para crear objetos en concreto. Podemos decir que una clase, es el razonamiento abstracto de un objeto, mientras que el objeto, es su materialización. A la acción de crear objetos, se la denomina “instanciar una clase” y dicha instancia, consiste en asignar la clase, como valor a una variable:

```
class Objeto {
    public $color = "";
    public $tamanio = "";
    public $aspecto = "";
    public $antenas = array(); # será otro objeto
    public $ojos = array(); # será otro objeto
    public $pelos = array(); # será otro objeto

    function flotar() { }
}

$et = new Objeto()
print $et->color;
print $et->tamanio;
print $et->aspecto;
$et->color = "rosa";
print $et->color;
```

Herencia: característica principal de la POO

Como comentamos en el título anterior, algunos objetos comparten las mismas propiedades y métodos que otro objeto, y además agregan nuevas propiedades y métodos. A esto se lo denomina herencia: una clase que hereda de otra.

```
# NuevoObjeto hereda de otra clase: Objeto
class NuevoObjeto extends Objeto {
    public $pie = array();

    function saltar() { }
}
```

Accediendo a los métodos y propiedades de un objeto

Una vez creado un objeto, es decir, una vez hecha la instancia de clase, es posible acceder a sus métodos y propiedades. Para ello, PHP utiliza una sintaxis muy simple: el nombre del objeto, seguido de “->” y la propiedad o método al cuál se desea acceder:

```
$objeto = new MiClase()  
print $objeto->propiedad;  
$objeto->otra_propiedad = "Nuevo valor";  
$variable = $objeto->metodo();  
print $variable  
print $objeto->otro_metodo();
```

Acceder a las propiedades de un objeto, dentro de la clase

Para poder acceder a las propiedades del objeto, dentro de la clase, se utiliza la pseudo variable \$this y lo mismo aplica para acceder a los métodos:

```
class A {  
  
    public $foo = '';  
  
    function bar() {  
        $this->foo = 'Hola Mundo';  
        $this->foobar();  
    }  
  
    function foobar() {  
        print $this->foo;  
    }  
}
```

Composición

Como comentamos anteriormente, algunos objetos se componen de las propiedades de otro (lo cual, no significa que las hereden, sino simplemente eso: “se componen de”).

Cuando la propiedad de un objeto, se compone de las características de otro objeto, dicha propiedad se transforma en una especie de “propiedad-objeto”. Es decir, que el tipo de datos de esta propiedad, pasa a ser de tipo objeto. Esto significa, que dicha propiedad, estará formada por sub-propiedades.

Para lograrlo, debemos recurrir a la composición. Una técnica que en PHP, se logra creando un método que asigne a dicha propiedad, el objeto correspondiente, pasado como parámetro:

```
class Objeto {  
    public $color = "";  
    public $tamano = "";  
    public $aspecto = "";  
    public $antenas = array(); # será otro objeto  
    public $ojos = array(); # será otro objeto  
    public $pelos = array(); # será otro objeto  
  
    function set_antena(Antena $antena) {  
        $this->antenas[] = $antena;  
    }  
  
    function set_ojo(Ojo $ojo) {  
        $this->ojos[] = $ojo;  
    }  
  
    function set_pelo(Pelo $pelo) {  
        $this->pelos[] = $pelo;  
    }  
}
```

Notar que al parámetro se le indica previamente que deberá ser un objeto de tipo Antena, Ojo y Pelo respectivamente.

Ejemplo práctico de Herencia vs. Composición

Como comentamos anteriormente, la diferencia entre la herencia y la composición es radical. Mientras que en el primer caso, un objeto pasa a ser un subtipo del objeto del cuál hereda, en el segundo, ambos objetos pueden ser sumamente diferentes entre sí y no guardar ninguna característica en común. El siguiente ejemplo lo demuestra claramente:

```
<?php

class Categoria {
    public $nombre = "";
}

class Producto {
    public $descripcion = "";
    public $categoria = array();
    public $precio = 0.0;

    public function __construct(Categoria $categoria) {
        $this->categoria = $categoria;
    }
}

class PrendaDeVestir extends Producto {
    public $marca = "";
    public $talle = "";
    public $color = "";
    public $material = "";
}

class Libro extends Producto {
    public $titulo = "";
    public $autor = "";
    public $editorial = "";
    public $paginas = 0;
    public $encuadernacion = "";
    public $anio_edicion = 0;
}

$categoria = new Categoria();
$categoria->nombre = "Vestimenta";

$pantalon = new PrendaDeVestir($categoria);
$pantalon->descripcion = "Pantalón clásico recto";
$pantalon->marca = "Christian Dior";
$pantalon->talle = "XS";
$pantalon->color = "Negro";
```



```
$pantalon->material = "Lino";
$pantalon->precio = 978.50;

print <<<EOT
=====
                        DATOS DEL PRODUCTO
=====
    CATEGORÍA: {$pantalon->categoria->nombre}
    PRODUCTO: {$pantalon->descripcion}
      MARCA: {$pantalon->marca}
      TALLE: {$pantalon->talle}
      COLOR: {$pantalon->color}
    MATERIAL: {$pantalon->material}
      PRECIO: \$ {$pantalon->precio}.-

EOT;
?>
```

Programando con Historias de Usuario

Una **historia de usuario** es aquella que puede escribirse con la siguiente frase:

Como [un usuario], **puedo** [acción/funcionalidad] **para** [beneficio]

Por ejemplo: *Como administrador del sistema, puedo agregar productos al catálogo para ser visualizados por los clientes.*

Muchas veces, puede resultar redundante o hasta incluso carecer de sentido, indicar el beneficio. Por ello, es frecuente describir las historias de usuario, sin incorporar este tercer elemento: Como administrador del sistema, puedo agregar productos al catálogo.

Vale aclarar, que es frecuente encontrar términos como “quiero” o “necesito” en reemplazo de “puedo” cuando se describen las historias de usuario:

Prioridad	Historia de Usuario	Valor	Esfuerzo estimado
1	como administrador del sistema necesito agregar productos al catálogo	10	
2	como usuario del sitio web quiero recorrer el catálogo de productos	10	
3	como cliente de la empresa quiero agregar productos a un pedido	10	
4	como cliente necesito enviar el pedido una vez que haya agregado todos los productos deseados	10	
5	como administrador de pedidos necesito ver la lista de pedidos efectuados por los clientes	7	
6	como administrador de pedidos necesito ver el detalle de los productos solicitados por el cliente en cada uno de los pedidos	10	
7	como administrador de pedidos necesito modificar el estado de cada pedido	3	
8	como cliente quiero poder visualizar el estado de mis pedidos	8	
9	como administrador del sistema necesito clasificar los productos por categorías	3	
10	como administrador del sistema necesito eliminar productos del catálogo	3	
11	como administrador del sistema necesito indicar cuáles productos y cuáles no, son visibles en el catálogo de productos que ven los usuarios	3	
12	como administrador del sistema necesito modificar productos del catálogo	3	
13	como cliente quiero poder modificar la cantidad de ítems de los productos de mi pedido	5	
14	como cliente de la empresa quiero poder cancelar mi pedido que aún no ha sido despachado	5	
15	como empleado de depósito necesito ver el detalle de los pedidos que aún no han sido enviados a empaque	1	
16	como empleado de depósito necesito indicar que un pedido ya ha sido enviado a empaque	5	
17	como cliente de la empresa necesito poder recuperar mi contraseña cuando la olvido	10	

Imagen 1: Backlog de Producto

Criterios de Aceptación

Es recomendable que cada Historia de Usuario, especifique cuales son los criterios de aceptación, para considerar cumplido el requisito.

Los criterios de aceptación, entonces, no son más que “pautas” o pequeñas “reglas” que una historia de usuario debe respetar para considerarla cumplida. Por ejemplo, para la historia de usuario **«Como administrador del sistema necesito agregar productos al catálogo»** los **criterios de aceptación**, podrían ser:

- Cada producto debe contener:
 - código de producto (opcional),
 - descripción de hasta 500 caracteres (opcional)
 - precio (obligatorio)
 - stock inicial (opcional)
 - un nombre (obligatorio),
 - una foto (opcional)
- No pueden existir dos productos con el mismo nombre de producto o código de producto
- El nombre del producto jamás puede estar vacío o tener menos de 3 caracteres
- Cuando no se asigne un stock inicial al producto, éste debe asignarse automáticamente en cero (sin stock)

Dividiendo Historias de Usuario en Tareas

La estrategia consiste en desmembrar el item a la mínima expresión posible, encuadrada en un mismo tipo de actividad.

El desmembramiento debe hacerse "de lo general a lo particular, y de lo particular al detalle".

Historia de Usuario # 123	
<i>Como usuario puedo ingresar mi e-mail y contraseña para acceder al sistema</i>	Prioridad 5
	Valor

Criterios de aceptación:	100
	Esfuerzo
	21

Ficha típica de Historia de Usuario

Análisis General:

Es aquel que responde a la pregunta ¿qué es?

- un sistema de validación de usuarios registrados

Análisis Particular:

Es el que responde a la pregunta ¿cómo hacerlo?

- Arquitectura MVC (requiere hacer el modelo, la lógica y la GUI de la vista y el controlador)

Análisis detallado: Es el que responde a la pregunta general ¿qué tareas se necesitan hacer para lograrlo?

Los detalles, son aquellas restricciones que deberán considerarse para todo lo anterior. Por ejemplo, la creación del modelo, repercutirá en la base de datos. Por lo cual, tras crear los nuevos modelos, habrá que correr el ORM para que modifique las tablas.

Otro detalle a considerar, es el tiempo que demanda cada tarea. Por ejemplo, correr un ORM lleva solo algunos minutos, pues no puede ser considerado una única tarea. Entonces, puede "sumarse como detalle" a la tarea "crear modelos". De manera contraria, documentar en el manual del usuario, llevará todo un día de trabajo. Por lo cual, debe asignarse a una única tarea.

- Crear el modelo Usuario y correr el ORM para modificar las tablas
 - Tag: programación
 - Esfuerzo: 2 h
- Diseñar un formulario HTML para insertar usuario y contraseña
 - Tag: diseño
 - Esfuerzo: 4 h
- Desarrollar la lógica de la vista del formulario de logueo
 - Tag: programación
 - Esfuerzo: 4 h
- Crear el controlador para el modelo
 - Tag: programación
 - Esfuerzo: 6 h
- Correr los test e integrar
 - Tag: testing
 - Esfuerzo: 1 h

Introducción a la Programación eXtrema

eXtreme Programming (programación extrema) también llamado **XP**, es una metodología de trabajo que tiene su origen en 1996, de la mano de **Kent Beck, Ward Cunningham y Ron Jeffries**. XP propone un **conjunto de 12 prácticas técnicas**, que **aplicadas de manera simultánea**, pretenden enfatizar los efectos positivos de en un proyecto de desarrollo de Software. En este curso, nos enfocaremos en tres de ellas:

1. TDD
2. Integración continua
3. Refactoring

TDD – Test-Driven Development

Entre las prácticas técnicas sugeridas por XP, nos encontramos con la técnica de programación TDD, del inglés *Test-Driven Development* (Desarrollo Guiado por Pruebas).

A muchos asusta esta práctica por el simple hecho de ser “desconocida” y resultar su descripción, algo confusa: ¿Qué es a caso, aquello de hacer un test antes de programar?

Pero no dejes que el miedo a lo desconocido te gane, que aprender a programar guiándote por test, es algo realmente simple, divertido y sobre todo, muy productivo.

¿Qué es el desarrollo -o programación- guiado por pruebas?

TDD es una técnica de programación que consiste en guiar el desarrollo de una aplicación, por medio de **Test Unitarios**.

Los **Test Unitarios** (Unit Testing) no son más que algoritmos que emulan lo que la aplicación se supone debería hacer, convirtiéndose así, en un modo simple de probar que lo que “piensas” programar, realmente funciona.

Para verlo de forma sencilla, imagina que estás desarrollando una aplicación, que necesita un algoritmo que sume dos números. Sabes perfectamente como hacerlo:

```
function sumar_dos_numeros($a, $b) {  
    return $a + $b;  
}  
  
$a = 10;  
$b = 25;  
  
$suma = sumar_dos_numeros($a, $b);  
  
echo $suma; // salida: 35
```

¿Sencillo verdad? ¿Para qué necesitarías complicarte la vida con eso de los test?

```
function sumar_dos_numeros($a, $b) {  
    return $a + $b;  
}  
  
$a = 'Zanahorias: 10';  
$b = '25 hinojos';  
  
$suma = sumar_dos_numeros($a, $b);  
  
echo $suma; // salida: 25
```

¿a caso 10 zanahorias más 25 hinojos no debería dar como resultado, 35 vegetales? Hasta que no lo pruebas, tal vez no sepas que suceda. Y hasta no saber que sucede, no sabrás como solucionarlo.

Y para esto, sirven los Test, ya que **te guiarán para saber qué, cómo, cuáles y cuántos algoritmos necesitarás desarrollar para que tu aplicación haga lo que se supone debe hacer.**

Es entonces, que en el caso anterior, los test que se desarrollen previamente, irán guiando tu desarrollo hasta encontrar la solución a todos los problemas. Y siguiendo el ejemplo, sabrás

que evidentemente, necesitarás de algún algoritmo extra: validas los parámetros antes de sumarlos y, o los conviertes a enteros o retornas un mensaje de error. En definitiva, **los Test Unitarios serán una guía para entender como funciona el código, ayudándote a organizarlo de manera clara, legible y simple.**

Carlos Blé Jurado⁴ en su libro **Diseño Ágil con TDD⁵** nos define al TDD como:

“[...] la respuesta a las grandes preguntas: ¿Cómo lo hago? ¿Por dónde empiezo? ¿Cómo se qué es lo que hay que implementar y lo que no? ¿Cómo escribir un código que se pueda modificar sin romper funcionalidad existente? [...]”

Según **Kent Beck** -uno de los principales fundadores de la metodología XP-, **implementar TDD nos otorga grandes ventajas:**

1. **La calidad del software aumenta** disminuyendo prácticamente a cero, la cantidad de *bugs* en la aplicación;
2. **Conseguimos código altamente reutilizable** puesto que los test nos obligan a desarrollar algoritmos genéricos;
3. **El trabajo en equipo se hace más fácil, une a las personas**, ya que al desarrollar con test, nos

4 Carlos Blé es un desarrollador de Software y emprendedor español, con una gran trayectoria en la ingeniería de sistemas ágiles. Visitar su blog: www.carlosble.com

5 Diseño Ágil con TDD. ISBN 978-1-4452-6471-4. Creative Commons-ND

aseguramos no romper funcionalidades existentes de la aplicación;

4. **Nos permite confiar en nuestros compañeros aunque tengan menos experiencia.** Esto es, debido a que el hecho de tener que desarrollar test antes de programar el algoritmo definitivo, nos asegura -independientemente del grado de conocimiento y experiencia del desarrollador- que el algoritmo efectivamente, hará lo que se supone, debe hacer y sin fallos;
5. **Escribir el ejemplo (test) antes que el código nos obliga a escribir el mínimo de funcionalidad necesaria, evitando sobrediseñar,** puesto que desarrollando lo mínimamente indispensable, se obtiene un panorama más certero de lo que la aplicación hace y cuál y cómo es su comportamiento interno;
6. **Los tests son la mejor documentación técnica que podemos consultar a la hora de entender qué misión cumple cada pieza del rompecabezas,** ya que cada test, no es más que un “caso de uso” traducido en idioma informático.

Test Unitarios

Los Test Unitarios (o Unit Testing), representan el alma de la programación dirigida por pruebas. Son test que se encargan de verificar -de manera simple y rápida- el comportamiento de una parte mínima de código, de forma independiente y sin alterar el funcionamiento de otras partes de la aplicación.

Características de los Test Unitarios

Un Test Unitario posee cuatro características particulares que debe guardar para considerarse “unitario”. Estas cualidades son:

1. Atómico:

Prueba una parte mínima de código.

Dicho de manera simple, cada test unitario debe probar una -y solo una- “acción” realizada por un método.

Por ejemplo, para un método que retorna el neto de un monto bruto más el IVA correspondiente, deberá haber un test que verifique recibir en forma correcta el importe bruto, otro que verifique el cálculo del IVA sobre un importe bruto y finalmente, un tercer test unitario que verifique el cálculo de un importe bruto más su IVA correspondiente.

2. Independiente:

Cada Test Unitario DEBE ser independiente de otro.

Por ejemplo, siguiendo el caso anterior, el test que verifique la suma de un importe bruto más su IVA correspondiente, no debe depender del test que verifica el cálculo del IVA.

3. Inocuo:

Podría decirse que cada test unitario debe ser inofensivo para el Sistema. Un test unitario DEBE poder correrse sin alterar ningún elemento del sistema, es decir, que no debe, por ejemplo, agregar, editar o eliminar registros de una base de datos.

4. Rápido:

La velocidad de ejecución de un test unitario cumple un papel fundamental e ineludible en el desarrollo guiado por pruebas, ya que de la velocidad de ejecución de un test, dependerá de manera proporcional, la velocidad con la que una funcionalidad se desarrolle.

Anatomía

Los Test Unitarios se realizan, en cualquier lenguaje de programación, mediante herramientas -Frameworks- con un **formato** determinado, conocido como **xUnit**.

De allí, que los frameworks para Unit Testing que cumplen con dicho formato, suelen tener nombres compuestos por una abreviatura del lenguaje de programación, seguida del término “unit”: **PyUnit** (Python), **PHPUnit** (PHP), **ShUnit** (Shell Scripting), **CppUnit** (C++), etc.

Exceptuando el caso de Shell Scripting, **los frameworks xUnit, utilizan el paradigma de programación orientada a objetos** (OOP) tanto en su anatomía de desarrollo como para su implementación (creación de test unitarios).

Por lo tanto, los Test Unitarios se agrupan en **clases**, denominadas **Test Case**, que heredan de una clase del framework xUnit, llamada **xTestCase**:

```
class BalanceContableTest extends PHPUnit_Framework_TestCase {  
    # Métodos  
}
```

*Creación de una clase Test Case con **PHPUnit** (PHP). La clase **hereda de** PHPUnit_Framework_TestCase*

Los métodos contenidos en una clase Test Case, pueden o no, ser Test Unitarios. **Los Test Unitarios** contenidos en una clase Test Case, **deben contener el prefijo test_ en el nombre del método** a fin de que el framework los identifique como tales.

```
ini_set('include_path', '.:usr/share/php:usr/share/pear');

class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function test_calcular_iva() {
        # Algoritmo
    }

}
```

Definición de un método “test” con PHPUnit. Nótese que los métodos de una clase Test Case DEBEN ser métodos públicos.

Otra ventaja que los frameworks xUnit nos proveen, es la facilidad de poder crear **dos métodos especiales** dentro de una clase Test Case **-que no son test-**, los cuales están destinados a **preparar el escenario** necesario para correr los test de esa clase y **eliminar aquello que se desee liberar**, una vez que el test finalice. Estos métodos, son los denominados **setUp()** y **tearDown()** respectivamente:

```
class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function setUp() {
        $this->importe_bruto = 100;
        $this->alicuota_iva = 21;
    }

    public function tearDown() {
        $this->importe_bruto = 0;
        $this->alicuota_iva = 0;
    }

    public function test_calcular_iva() {
        # Algoritmo
    }

}
```

Los métodos setUp() y tearDown() en PHPUnit

Esta anatomía dual -por un lado, la del Framework y por otro, la de utilización o implementación de éste-, se logrará finalmente, dividiendo a cada Test Case -imaginariamente- en **tres partes**, identificadas por las siglas **AAA** las cuáles representan a las tres “acciones” que son necesarias llevar a cabo, para dar forma a los Tests:

Arrange, Act and Assert

(preparar, actuar y afirmar)

Preparar los test consiste en establecer -setear o configurar- todo aquello que sea necesario para que cada uno de los métodos “test” pueda ejecutarse. Esto puede ser, la declaración de propiedades comunes a todos los test, la instancia a objetos, etc.

Cuando estas “preparaciones previas” sean comunes a todos los test, deberán valerse del método setUp() para ser creadas. De lo contrario, se crearán métodos -a modo de helpers- dentro de la clase Test Case, a los cuáles cada uno de los test, recurra cuando los necesite. Vale aclarar que estos “helpers” NO podrán contener el prefijo “test” en su nombre.

Actuar, se refiere a hacer la llamada al código del Sistema cubierto por Test (SUT) que se desea probar. Esto se conoce como “cobertura de código” o Code Coverage:

```
class BalanceContableTest extends PHPUnit_Framework_TestCase {  
  
    # Arrange (preparar)  
    public function setUp() {  
        // importar la clase a ser testeada  
        require_once('/contabilidad/models/balance_contable.php');  
    }  
}
```

```

        // setear propiedades comunes
        $this->importe_bruto = 100;
        $this->alicuota_iva = 21;
    }

    public function tearDown() {
        $this->importe_bruto = 0;
        $this->alicuota_iva = 0;
    }

    public function test_calcular_iva() {
        # Act (actuar)
        // Instanciar al objeto que será probado
        $this->coverage = new BalanceContable();

        // modificar las propiedades del objeto
        $this->coverage->importe_bruto = $this->importe_bruto;
        $this->coverage->alicuota_iva = $this->alicuota_iva;

        // invocar al método que se está testeando
        $result = $this->coverage->calcular_iva();

        # Assert (afirmar)
        // sentencias
    }
}

```

Finalmente, **afirmar el resultado de un test**, se refiere a **invocar a los métodos assert del Framework xUnit**, que sean necesarios para afirmar que el resultado obtenido durante la actuación, es el esperado (más adelante, veremos cuáles son estos métodos assert de los cuales disponemos):

```

class BalanceContableTest extends PHPUnit_Framework_TestCase {

    # Arrange (preparar)
    public function setUp() {
        // importar la clase a ser testeada
        require_once('/contabilidad/models/balance_contable.php');

        // setear propiedades comunes
        $this->importe_bruto = 100;
        $this->alicuota_iva = 21;
    }

    public function tearDown() {

```



```
        $this->importe_bruto = 0;
        $this->alicuota_iva = 0;
    }

    public function test_calcular_iva() {
        # Act (actuar)
        // Instanciar al objeto que será probado
        $this->coverage = new BalanceContable();

        // modificar las propiedades del objeto
        $this->coverage->importe_bruto = $this->importe_bruto;
        $this->coverage->alicuota_iva = $this->alicuota_iva;

        // invocar al método que se está testeando
        $result = $this->coverage->calcular_iva();

        # Assert (afirmar)
        $this->assertEquals(121, $result);
    }
}
```

Algoritmo para escribir pruebas unitarias

Existe un algoritmo (o por qué no, “una fórmula”), para escribir Test Unitarios bajo TDD, el cual consiste en:

PRIMER PASO: Escribir el Test y hacer que falle

Para ello, lo primero que haremos será crear nuestra clase Test Case (nótese que ningún otro código ha sido escrito al momento, para esta aplicación).

Nos baseremos en el mismo ejemplo que venimos siguiendo, donde lo que se necesita es calcular el IVA a un monto bruto dado.

```
<?php
require_once('contabilidad/models/BalanceContable.php');

class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function test_calcular_iva() {
```

```
$this->coverage = new BalanceContable();
$this->coverage->importe_bruto = 1500;
$this->coverage->alicuota_iva = 21;
$result = $this->coverage->calcular_iva();
$this->assertEquals(315, $result);
}
}
?>
```

archivo: /MyApp/Tests/BalanceContableTest.php

Nuestro test, nos está guiando el desarrollo, diciéndonos que:

1. Debemos crear una clase llamada BalanceContable que la guardaremos en el archivo BalanceContable.php
2. Esta clase debe tener dos propiedades: importe_bruto y alicuota_iva
3. La clase debe contener un método llamado calcular_iva()

¿Qué hace nuestro test? Verificará que el método calcular_iva() retorne el valor 315. Este valor, debe corresponder al 21% (alicuota_iva) de 1500 (importe_bruto).

Entonces, el siguiente paso, será crear aquello que nuestro Test, nos guió a hacer:

```
<?php
class BalanceContable {

    public $importe_bruto;
    public $alicuota_iva;

    public function calcular_iva() {
```

```
}  
}  
?>
```

archivo: /MyApp/contabilidad/models/BalanceContable.php

Finalmente, **correremos nuestro Test, y éste, deberá fallar:**

```
eugenia@cocochito:~/borrador/MyApp$ phpunit Tests  
PHPUnit 3.4.5 by Sebastian Bergmann.
```

F

Time: 0 seconds, Memory: 4.00Mb

There was 1 failure:

1) BalanceContableTest::test_calcular_iva
Failed asserting that <null> matches expected <integer:315>.

/home/eugenia/borrador/MyApp/Tests/BalanceContableTest.php:11

FAILURES!

Tests: 1, Assertions: 1, Failures: 1.

En la muestra de consola anterior, corrimos el Test creado mediante el comando `phpunit` al cual le pasamos como parámetro la carpeta `Tests` que contiene (y contendrá) todos nuestros test. Se resaltan en color naranja, las explicaciones arrojadas por PHPUnit, donde nos dice que el método `test_calcular_iva` del Test Case `BalanceContableTest` ha fallado, en afirmar que `NULL` coincidió con el valor `315` esperado de tipo entero.

Con lo anterior, **nos aseguramos que el Test falle, cuando se espera que así sea.**

SEGUNDO PASO: Escribir la mínima cantidad de código para que el test pase.

De la misma forma que nos aseguramos que el test fallará cuando deba hacerlo, ahora debemos asegurarnos de que el test NO falle cuando no lo esperamos.

Para ello, debemos evitar escribir -de buenas a primeras-, el algoritmo de nuestro método. Por el contrario, debemos encontrar la expresión mínima que nos retorne el resultado que esperamos. Es decir, que nos retorne el entero 315. Entonces, editamos nuestra clase `BalanceContable` y agregamos la mínima expresión necesaria al método `calcular_iva()`:

```
<?php
class BalanceContable {

    public $importe_bruto;
    public $alicuota_iva;

    public function calcular_iva() {
        return 315;
    }
}
```

Volvemos a correr el test para asegurarnos de que esta vez, NO falle:

```
eugenia@cocochito:~/borrador/MyApp$ phpunit Tests
PHPUnit 3.4.5 by Sebastian Bergmann.

.

Time: 0 seconds, Memory: 4.00Mb

OK (1 test, 1 assertion)
```

TERCER PASO: Escribir un nuevo test y hacer que falle

Hemos escrito la cantidad mínima de código para que nuestro primer test, pase. Lo hemos hecho fallar, cuando aún no existía algoritmo o instrucción en el método probado. Es hora de crear un nuevo test que falle, cuando ya existe una mínima porción de código escrita (o modificar el existe, cambiando los parámetros de éste). Agregaremos entonces, un nuevo test y modificaremos el nombre del anterior, de manera tal que

nuestro Test Case se vea como sigue:

```
<?php
require_once('contabilidad/models/BalanceContable.php');

class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function test_calcular_iva_con_1500_esperando_315() {
        $this->coverage = new BalanceContable();
        $this->coverage->importe_bruto = 1500;
        $this->coverage->alicuota_iva = 21;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(315, $result);
    }

    public function test_calcular_iva_con_2800_esperando_588() {
        $this->coverage = new BalanceContable();
        $this->coverage->importe_bruto = 2800;
        $this->coverage->alicuota_iva = 21;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(588, $result);
    }
}

?>
```

Ahora nuestro segundo test, debe fallar:

```
eugenia@cocochito:~/borrador/MyApp$ phpunit Tests
PHPUnit 3.4.5 by Sebastian Bergmann.

.F

Time: 0 seconds, Memory: 4.00Mb

There was 1 failure:

1) BalanceContableTest::test_calcular_iva_con_2800_esperando_588
Failed asserting that <integer:315> matches expected <integer:588>.

/home/eugenia/borrador/MyApp/Tests/BalanceContableTest.php:19

FAILURES!
Tests: 2, Assertions: 2, Failures: 1.
```

CUARTO PASO: Escribir el algoritmo necesario para hacer pasar el test

Es hora de programar lo que nuestro método, realmente necesita y sin *hardcodear* el retorno de resultados. Modificaremos nuestro método `calcular_iva()` de la clase

BalanceContable a fin de escribir el algoritmo que efectivamente se encargue de calcular el valor de retorno:

```
<?php
class BalanceContable {

    public $importe_bruto;
    public $alicuota_iva;

    public function calcular_iva() {
        $iva = $this->alicuota_iva / 100;
        $neto = $this->importe_bruto * $iva;
        return $neto;
    }
}

?>
```

Nuestros test, ahora pasarán:

```
eugenia@cocochito:~/borrador/MyApp$ phpunit Tests
PHPUnit 3.4.5 by Sebastian Bergmann.

..

Time: 0 seconds, Memory: 4.00Mb

OK (2 tests, 2 assertions)
```

Es válido hacer notar, que a medida que se van escribiendo los test, éstos, no solo guiarán nuestro desarrollo en cuanto a “lo nuevo que se debe escribir”, sino que además, **nos irán obligando a refactorizar el código constantemente**, en principio, mostrándonos el camino para eliminar redundancias y **crear un diseño más simple**:

```
<?php
require_once('contabilidad/models/BalanceContable.php');

class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function test_calcular_iva_con_1500_esperando_315() {
        $this->coverage = new BalanceContable();
    }
}
```

```

        $this->coverage->importe_bruto = 1500;
        $this->coverage->aliquota_iva = 21;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(315, $result);
    }

    public function test_calcular_iva_con_2800_esperando_588() {
        $this->coverage = new BalanceContable();
        $this->coverage->importe_bruto = 2800;
        $this->coverage->aliquota_iva = 21;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(588, $result);
    }
}

?>

```

La redundancia anterior, nos está diciendo que existen **elementos de preparación (arrange) comunes para nuestra Test Case:**

```

<?php
require_once('contabilidad/models/BalanceContable.php');

class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function setUp() {
        $this->coverage = new BalanceContable();
        $this->coverage->aliquota_iva = 21;
    }

    public function test_calcular_iva_con_1500_esperando_315() {
        $this->coverage->importe_bruto = 1500;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(315, $result);
    }

    public function test_calcular_iva_con_2800_esperando_588() {
        $this->coverage->importe_bruto = 2800;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(588, $result);
    }
}

?>

```

RECUERDA: cada cambio que se haga al código tanto del test como del SUT (código de la aplicación cubierta por test),

requiere que se vuelva a correr el test.

Unit Testing con PHPUnit

Existen varios frameworks xUnit para Unit Testing en PHP, pero sin dudas, el único que ha demostrado contar con una gran cobertura de código, estabilidad y buena documentación, es PHPUnit.

El **manual oficial de PHPUnit (en inglés)** puede encontrarse en: <http://www.phpunit.de/manual/3.6/en/>.

Se puede **instalar PHPUnit** en sistemas operativos **GNU/Linux**, ejecutando los siguientes comandos:

```
sudo pear upgrade PEAR
pear config-set auto_discover 1
pear install pear.phpunit.de/PHPUnit
```

Aunque en **distribuciones basadas en Debian**, puede hacerse directamente mediante la instalación del paquete `phpunit` con `apt-get`:

```
sudo apt-get install phpunit
```

Métodos Assert de PHPUnit

PHPUnit provee una gran cantidad de métodos assert cuyas referencias, podemos encontrar en el **Capítulo 4 del manual oficial**:

<http://www.phpunit.de/manual/3.6/en/writing-tests-for-phpunit.html>

Algunas **características comunes** de los métodos assert, son:

- Generalmente, por **cada método assert existe su opuesto**: `assertContains()` y `assertNotContains()`.
- A la vez, cada método assert deberá recibir mínimamente **un parámetro** que será el **resultado** de ejecutar el código del SUT.
- Adicionalmente, a cada método assert, se le puede pasar como **parámetro opcional, un mensaje personalizado** para ser arrojado en caso de error (generalmente, será el último parámetro).
- Los métodos assert que requieren el paso de dos parámetros obligatorios (valores que deben compararse entre sí), generalmente guardan el siguiente orden:

`metodoAssert($valor_esperado, $valor_recibido)`

Es decir, que en esos casos, siempre **el primer parámetro será el valor esperado** y el **segundo parámetro, el valor recibido** por la ejecución del código SUT.

Veamos algunos ejemplos puntuales:

```
<?php
require_once('contabilidad/models/BalanceContable.php');

class BalanceContableTest extends PHPUnit_Framework_TestCase {

    public function setUp() {
        $this->coverage = new BalanceContable();
        $this->coverage->alicuota_iva = 21;
    }

    // AssertEquals($valor_esperado, $valor_recibido)
    public function test_calcular_iva() {
        $this->coverage->importe_bruto = 1500;
        $result = $this->coverage->calcular_iva();
        $this->assertEquals(315, $result);
    }

    // AssertTrue($valor_recibido)
```

```

    public function test_alcanzado_por_impuesto_de_importacion_con_160() {
        $this->coverage->importe_bruto = 160;
        $result = $this->coverage->alcanzado_por_impuesto_de_importacion();
        $this->assertTrue($result);
    }

    // AssertNull($valor_recibido)
    public function test_alcanzado_por_impuesto_de_importacion_con_143() {
        $this->coverage->importe_bruto = 143;
        $result = $this->coverage->alcanzado_por_impuesto_de_importacion();
        $this->assertNull($result);
    }
}
?>

```

Código fuente de la clase Test Case

```

<?php
class BalanceContable {

    public $importe_bruto;
    public $alicuota_iva;

    # Calcular IVA sobre un importe bruto
    public function calcular_iva() {
        $iva = $this->alicuota_iva / 100;
        $neto = $this->importe_bruto * $iva;
        return $neto;
    }

    # Determinar si un importe paga impuesto de importación
    public function alcanzado_por_impuesto_de_importacion() {
        // importes mayores a 150 USD pagan impuesto
        if($this->importe_bruto > 150) {
            return True;
        }
    }
}
?>

```

Código fuente del SUT

Ejercicio

Escribir el código SUT del siguiente Test Case:

```
<?php
require_once('Usuario.php');

class UsuarioTest extends PHPUnit_Framework_TestCase {

    public function setUp() {
        $this->coverage = new Usuario();
        $this->coverage->username = "juanperez75";
        $this->coverage->password = md5("pablito: clavo_1_palito");
    }

    public function test_set_usuario_esperando_key() {
        $result = $this->coverage->set_usuario();
        $this->assertArrayHasKey('Usuario', $result);
    }

    public function test_set_usuario_esperando_juanperez75() {
        $result = $this->coverage->set_usuario();
        $this->assertEquals('juanperez75', $result['Usuario']);
    }

    public function test_set_usuario_esperando_pass() {
        $result = $this->coverage->set_usuario();
        $this->assertEquals(md5('pablito: clavo_1_palito'),
        $result['Clave']);
    }
}

?>
```

Unificación del código en Repositorios

Un Repositorio es un espacio destinado a almacenar información digital. En nuestro caso, lo que se almacenará en ese repositorio, serán los archivos -código fuente, *tarballs*, binarios, etc- de nuestra aplicación.

Sus principales características son:

- Espacio de **almacenamiento centralizado** de, principalmente, el código fuente de la aplicación así como scripts de construcción -en el caso de aplicaciones que requieran ser compiladas o simplemente, necesiten realizar configuraciones especiales, ya sea tanto para continuar desarrollándolas como para ejecutarlas-.
- Para ser efectivos, deben llevar un **control histórico de cambios** que se vayan efectuando en los archivos -preferentemente automático-, permitir el establecimiento de *tags* -etiquetas- que ayuden a identificar diferentes *releases* -versiones-.

Y todas esta característica, son aquellas que nos brindan los programas de control de versiones⁶.

Sobre los Sistemas de Control de Versiones

⁶ http://es.wikipedia.org/wiki/Programas_para_control_de_versiones

Los Sistemas de Control de Versiones (SCV) pueden agruparse en dos tipos:

- **Centralizados:**

un único repositorio centralizado administrado por un solo responsable.

- **Distribuidos (recomendados):**

donde existe un repositorio central que cada usuario podrá clonar para obtener su propio repositorio -local- e interactuar con con otros repositorios locales.

Entre los **SCV distribuidos** podemos destacar excelentes alternativas **GPL** (Software Libre), como es el caso de -entre otros-, **Git** (de Linus Torvalds, creador del Kernel Linux en el que se basa el Sistema Operativo GNU/Linux), **Mercurial** (desarrollado en Python y C) o el magnífico **Bazaar**, nacido a partir de GNUArch y desarrollado íntegramente en Python por Martin Pool, con el patrocinio de Canonical y **elegido en este curso**.

Una gran ventaja de los SCV es que permiten a varios programadores trabajar simultáneamente sobre los mismos archivos, impidiendo que el trabajo de uno, pise al trabajo de otro.

Los SCV pueden utilizarse tanto a través de línea de comandos, como de aplicaciones gráficas. En este curso, nos centraremos en el uso por medio de línea de comandos.

Los SCV, en su mayoría -y a rasgos generales- cuentan con un conjunto de funcionalidades, las cuales, para cada una, existe

un determinado comando (generalmente, similar en la mayoría de los SCV).

Integración continua con Bazaar

Bazaar, cuenta con una estructura que puede dividirse en:

1. Repositorio:

conjunto de revisiones

2. Árbol de trabajo:

un directorio que contiene las revisiones con sus correspondientes ramas

3. Ramas:

un conjunto ordenado de las diferentes revisiones con sus archivos correspondientes

4. Revisiones:

es una vista espontánea del estado de cada uno de los archivos, en un momento determinado

Un repositorio, entonces, contiene un árbol de trabajo, quien a la vez, puede estar integrado por varias ramas, las cuales almacenarán las distintas versiones (revisiones) por las que ha ido transitando el Software.

Instalación de Bazaar

Nótese que Bazaar deberá instalarse en cada una de las máquinas donde se desee usar. Esto es, en los ordenadores que contarán con repositorios locales y en el ordenador destinado a actuar como repositorio central.

Para instalar Bazaar, por favor, visita

<http://wiki.bazaar.canonical.com/Download> para obtener las instrucciones necesarias.

Bazaar por línea de comandos

Más adelante veremos todo lo que puede hacerse con Bazaar. Aquí, nos limitaremos a conocer la sintaxis básica.

```
bzr [-h|-v|-q] comando-interno-de-bazaar [argumentos]
```

Las **opciones globales** (opcionales) **-h**, **-v** y **-q**, significan **ayuda** (sobre el comando interno), **modo verboso** (despliega mayor información sobre lo que sucede mientras se ejecuta el comando en cuestión) y **silenciar** (solo desplegará errores y advertencias), respectivamente.

Veremos los comandos internos de bazaar, en el siguiente título. No obstante, citaré algunos comandos útiles a la propia aplicación, que pueden servirnos para entender mejor de que se trata:

```
bzr help commands
```

Despliega una lista completa de los comandos internos y su descripción.

```
bzr version
```

Despliega información sobre la versión de Bazaar.

```
bzr help comando-interno
```

Despliega información de ayuda sobre el comando indicado.

Presentarse ante Bazaar

Cada vez que enviemos cambios al repositorio, Bazaar tendrá que identificarnos a fin de poder ofrecer la información correcta sobre quien ha efectuado alguna revisión. Para ello, el primer paso, es presentarnos:

```
bzr whoami "Juan Perez <juanperez@dominio.ext>"
```

Iniciar un nuevo proyecto

El siguiente paso, será crear un nuevo proyecto. Para ello, haremos lo siguiente:

Primero inicializamos el **repositorio central** en el ordenador destinado a tal fin:

```
__eugenia_1978_esAR__@mydream:/srv/repos$ bzr init-repo app-curso-xp
Shared repository with trees (format: 2a)
Location:
  shared repository: app-curso-xp
```

Luego, allí mismo, creamos nuestro **branch** (al que llamaremos trunk):

```
__eugenia_1978_esAR__@mydream:/srv/repos$ bzr init app-curso-xp/trunk
Created a repository tree (format: 2a)
Using shared repository: /srv/repos/app-curso-xp/
```

Clonar el repositorio central: crear los repositorios locales

Es hora de que cada miembro del equipo, se traiga el branch desde el repositorio central. Esta actividad deberá realizarse en cada una de las máquinas de cada uno de los miembros del equipo (previamente, deberán presentarse ante Bazaar mediante bzr whoami).

Para clonar el repositorio, cada uno de los miembros del equipo, hará lo siguiente:

```
eugenia@cocochito:~/example$ bzr branch  
bzr+ssh://user@x.x.x.x/srv/repos/app-curso-xp/trunk  
user@x.x.x.x's password:  
Branched 0 revision(s).
```

Nótese que la ruta hacia el branch debe formarse por:

```
protocolo://usuario@host/ruta/al/branch/central
```

Nótese que **en el caso del protocolo SSH**, debe anteponerse el prefijo **bzr+**

Si listamos el directorio donde clonamos el repo, podremos ver que ya tenemos nuestro branch local:

```
eugenia@cocochito:~/example$ ls -lha  
total 12K  
drwxr-xr-x  3 eugenia eugenia 4,0K 2012-04-21 19:59 .  
drwxr-xr-x 65 eugenia eugenia 4,0K 2012-04-21 20:04 ..  
drwxr-xr-x  3 eugenia eugenia 4,0K 2012-04-21 19:59 trunk
```

Podremos comprobar también, que efectivamente estamos “conectando” nuestro repo local con el central:

```
eugenia@cocochito:~/example$ cd trunk/  
eugenia@cocochito:~/example/trunk$ bzr info  
Standalone tree (format: 2a)  
Location:  
  branch root: .  
  
Related branches:  
  parent branch: bzr+ssh://__eugenia_1978_esAR__@66.228.52.93/srv/repos/app-  
curso-xp/trunk/
```

Nociones básicas para integrar código de forma continua

Crearemos ahora nuestro primer archivo (de prueba):

```
eugenia@cocochito:~/example/trunk$ echo "Hola Mundo" > prueba.txt
```

Y verificaremos cual es el **estado** actual de nuestro repo:

```
eugenia@cocochito:~/example/trunk$ bzz st
unknown:
prueba.txt
```

puede utilizarse tambien, bzz status

Nos indica que hay un archivo desconocido. ¡Es el archivo que acabamos de crear! No te preocupes. El siguiente paso tras crear nuevos archivos y/o directorios, es avisarle a Bazaar, para lo cual, hay que **agregar el archivo** o directorio:

```
eugenia@cocochito:~/example/trunk$ bzz add prueba.txt
adding prueba.txt
```

Volvemos a comprobar el estado:

```
eugenia@cocochito:~/example/trunk$ bzz st
added:
prueba.txt
```

Bazaar no está informando que se ha realizado un cambio en el repositorio. Y **todo cambio, debe ser informado Bazaar** con un mensaje que lo describa:

```
eugenia@cocochito:~/example/trunk$ bzz ci -m "Agregado archivo de prueba"
Committing to: /home/eugenia/example/trunk/
added prueba.txt
Committed revision 1.
```

¡Ya tenemos la primera revisión!!! Pero el *commit*, fue realizado

localmente. Nuestro repo central, aún no se ha enterado y tenemos que **enviar los cambios al repositorio central**:

```
eugenia@cocochito:~/example/trunk$ bzip push
bzip+ssh://__eugenia_1978_esAR__@66.228.52.93/srv/repos/app-curso-xp/trunk/
__eugenia_1978_esAR__@66.228.52.93's password:
This transport does not update the working tree of:
bzip+ssh://__eugenia_1978_esAR__@66.228.52.93/srv/repos/app-curso-xp/trunk/.
See 'bzip help working-trees' for more information.
Pushed up to revision 1.
```

Pero ¿por qué nos dice que el árbol de trabajo no se ha actualizado? Debemos actualizarlo en el repositorio central:

```
__eugenia_1978_esAR__@mydream:/srv/repos/app-curso-xp/trunk$ bzip st
working tree is out of date, run 'bzip update'
__eugenia_1978_esAR__@mydream:/srv/repos/app-curso-xp/trunk$ bzip update
+N prueba.txt
All changes applied successfully.
Updated to revision 1 of branch /srv/repos/app-curso-xp/trunk
```

Diariamente, todos los miembros del equipo, deberán **traer los cambios** desde el repo central:

```
eugenia@cocochito:~/example/trunk$ bzip pull
bzip+ssh://__eugenia_1978_esAR__@66.228.52.93/srv/repos/app-curso-xp/trunk/
__eugenia_1978_esAR__@66.228.52.93's password:
No revisions to pull.
```

Guardando el path del repo central

Puede ser extremadamente molesto, tener que estar indicando la dirección del repo central con cada pull y push que hagamos. Podemos evitar esto, editando el **archivo de configuración de nuestro repo local**:

```
eugenia@cocochito:~/example/trunk$ vim .bzip/branch/branch.conf
```

y configuramos las variables `pull_location` y `push_location` (modificar la ruta por la que corresponda):

```
push_location = bzd+ssh://user@host/srv/repos/app-curso-xp/trunk/  
pull_location = bzd+ssh://user@host/srv/repos/app-curso-xp/trunk/
```

Guardamos los cambios y ya podremos hacer pull y push, solo con el comando respectivo:

```
eugenia@cocochito:~/example/trunk$ bzr pull  
Using saved parent location:  
bzd+ssh://__eugenia_1978_esAR__@66.228.52.93/srv/repos/app-curso-xp/trunk/  
__eugenia_1978_esAR__@66.228.52.93's password:  
No revisions to pull.  
eugenia@cocochito:~/example/trunk$ bzr push  
Using saved push location:  
bzd+ssh://__eugenia_1978_esAR__@66.228.52.93/srv/repos/app-curso-xp/trunk/  
__eugenia_1978_esAR__@66.228.52.93's password:  
This transport does not update the working tree of:  
bzd+ssh://__eugenia_1978_esAR__@66.228.52.93/srv/repos/app-curso-xp/trunk/.  
See 'bzr help working-trees' for more information.  
No new revisions to push.
```

Integración continua avanzada con Bazaar

A la hora de trabajar con un SCV, aparece una larga lista de cuestiones, que ameritan especial cuidado. Por ejemplo ¿qué sucede si por accidente, elimino un archivo en mi repositorio local? ¿Cómo lo recupero? O ¿qué sucede si modifiqué un archivo en mi repo local, sobre una versión distinta a la que está en el repo central? Y estas, son solo dos de las decenas de preguntas que nos pueden surgir.

Veremos aquí, como solucionar cada una de las problemáticas más frecuentes que se presentan en la vida diaria de la integración continua.

Problema	Descripción	solución
Ignorar archivo	Se desea evitar que un archivo o directorio sea enviado al repo central	bzr ignore archivo bzr ignore archivo.txt
Recuperar archivo	Recuperar una versión anterior de un archivo o directorio	bzr revert nro_revision bzr revert 2
Dejar de lado un archivo temporalmente	Se desea evitar temporalmente, enviar los cambios de un archivo, al repo central	bzr shelve archivo bzr shelve archivo.txt
Recuperar cambios	Se desea recuperar los cambios de un archivo, previamente salvado mediante shelve	bzr unshelve archivo bzr unshelve archivo.txt
Encontradas versiones diferentes de un mismo archivo	Al traer los cambios desde el repo central, las modificaciones hechas sobre un archivo, interfieren con las efectuadas localmente al mismo archivo	bzr merge -i archivo la opción -i es opcional y permite seleccionar los cambios de forma interactiva
Merge no resolvió el conflicto	Tras combinar diferentes versiones, Bazaar informa de conflictos existentes	Bzr conflict-diff archivo (generalmente requerirá solucionar el conflicto de forma manual. Conflict-diff mostrará las diferencias encontradas que no pudieron resolverse con el merge)

Nótese que toda vez que se indica la palabra “archivo”, se hace referencia no solo a archivos sino también a directorios.

Resumen de comandos de uso frecuente

Comando	Descripción
add	Agregar archivo o directorio
check	Verifica la consistencia del árbol de trabajo
ci	Envía mensajes al repo informando cambios (debe ejecutarse siempre después de realizar cambios y antes de enviar cambios al repo central con push)
conflict-diff	Muestra las diferencias que generan conflictos en un archivo
deleted	Muestra la lista de archivos eliminados
ignore	Ignora un archivo
ignored	Muestra la lista de archivos ignorados

info	Muestra información sobre el árbol de trabajo
log	Muestra los cambios históricos (revisiones) de un branch
merge	Combina los cambios centrales con los locales en un archivo (tras el merge, siempre se debe hacer commit – generalmente: <code>bzr ci -m “merge”</code>)
mv	Mueve o renombra un archivo o directorio
pull	Enviar cambios al repo central
push	Traer cambios desde el repo central
remerge	Elimina un merge anterior
remove	Elimina archivos o directorios
renames	Muestra la lista de archivos que han sido renombrados
resolve	Marca resuleto un conflicto
revert	Recupera un archivo a una revisión anterior
revno	Muestra el número de la revisión actual
shelve	Deja temporalmente a un lado, un determinado archivo que no se desee enviar aún al repo central
st	Muestra el estado del repo (localmente) – cuando no muestra nada, significa que todo está en orden y se pueden enviar cambios al repo central sin conflictos. Cualquier otro mensaje, requerirá intervenir para resolverlo, antes de enviar cualquier cambio al repo central.
tag	Taguea (etiqueta) una revisión (la remueve o modifica según las opciones pasadas como argumentos)
tags	Retorna la lista completa de tags
uncommit	Revierte un commit efectuado anteriormente
unshelve	Recupera los cambios de un archivo salvado mediante shelve
update	Actualiza el árbol de trabajo trayendo el último commit enviado
whoami	Presentarse ante Bazaar (u obtener los datos de presentación aportados con anterioridad)

Resumen para uso diario de Bazaar

Esta, es una breve síntesis, de **los pasos diarios a seguir**, cuando se trabaja con repositorios.

Al comienzo, podría ser muy útil, imprimir esta planilla e ir marcando los casilleros vacíos a medida que se completa cada paso.

¿Cuándo?	¿Qué?	¿Cómo?	
Al comenzar el día	Actualizar el repo local	bzr pull	
	Mergear archivos con diferencias (si aplica)	bzr merge bzr ci -m "merge"	
	Resolver conflictos en archivos, no resueltos con un merge: nombre_archivo (archivo conflictivo mergeado localmente) nombre_archivo. BASE (original del repo central) nombre_archivo. OTHER (archivo con las modificaciones hechas por otro miembro del equipo) nombre_archivo. THIS (archivo modificado por uno mismo, localmente)	visualizar los archivos en conflicto y corregirlos.	
	Eliminar el conflicto	bzr resolve	
Luego de	Trabajar libremente sobre tus archivos		

actualizar el repo local			
Al concluir un task (tarea)	Verificar el estado de los cambios	bzr st	
	Agregar archivos nuevos (reportados como unknow)	bzr add nombre_archivo	
	Eliminar archivos obsoletos	bzr remove nombre_archivo	
	Ignorar archivos que no deban actualizarse en el repo central	bzr ignore nombre_archivo	
	Comitear los cambios	bzr ci -m "breve descripción de la tarea terminada"	
	Correr todos los test y verificar que pasen	php -f test python -m unittest discover -v	
	Enviar cambios al repo central	bzr push	
Al finalizar el día	Actualizar el repo CENTRAL	bzr update	

Los pasos que se encuentran sombreados en tono más claro, son opcionales, y dependerán de cada caso en particular.

Ver el historial de revisiones: **bzr log** | Obtener el nro. de revisión actual: **bzr revno**

Refactoring

Veremos aquí, la quinta práctica sugerida por eXtreme Programming, con mayores detalles y algunos ejemplos.

Refactoring, como comentamos anteriormente, es una técnica que consiste en mejorar el código fuente de una aplicación (limpiarlo), sin que dichas modificaciones, afecten el comportamiento externo del sistema.

Existen diferentes tipos de refactorizaciones que pueden ser necesarias implementar al código de nuestra aplicación. Cada tipo, representa una técnica diferente de refactorización. Por ejemplo, eliminar código redundante, requiere de una técnica diferente a dividir los algoritmos de un método para crear métodos derivados.

Sin embargo, hablar de técnicas de refactorización puede resultar confuso, ya que **la refactorización en sí misma es una técnica, que ofrece diferentes soluciones a cada tipo de problema**. Por lo tanto, es preferible pensar la refactorización como una única técnica que propone diferentes soluciones a cada tipo de problema.

El problema

En principio, habría que diferenciar el término “problema” de la palabra “error”, para no generar confusiones. El error en sí, es una falla en el código fuente, que impide el correcto comportamiento del sistema. Mientras que el problema, puede

definirse como “algo que huele mal en el código fuente”⁷ pero sin embargo, no impide el correcto funcionamiento de la aplicación.

Los problemas que se pueden presentar en el código fuente de una aplicación, dependen de muchísimos factores, que en gran parte de los casos, encuentran una relación directa con el paradigma de programación empleado así como en el lenguaje que se utilice.

Si se intentara abarcar todos los problemas posibles, la lista podría tornarse infinita, tediosa y hasta inútil o muy confusa. Es por ello, que solo abarcaremos los problemas más frecuentes, que puedan considerarse generales, independientes al lenguaje pero más cercanos al paradigma de la programación orientada a objetos.

La solución

Indefectiblemente, la solución a cada problema será la refactorización y aunque resulte redundante, la solución, dependerá de cada problema. Sin embargo, como regla general, **la solución deberá comenzar por identificar el momento en el cual llevarla a cabo.**

⁷ Kent Beck, uno de los creadores de eXtreme Programming, es quien introdujo el término “bad smells” (malos olores) para referirse de manera global, a aquellas expresiones y algoritmos poco claros que generan confusión en el código fuente de un sistema, tornándolo más complejo de lo que debería ser.

Cuándo y cómo tomar la decisión de refactorizar

Tres strikes y ¡Refactoriza!

En el mundo de la refactorización, haciendo una analogía con el béisbol, suele utilizarse la regla “Tres Strike⁸ y ¡refactoriza!”. Esto puede describirse análogamente como: “la primera vez que hagas algo, solo hazlo. La segunda vez que hagas algo similar, notarás que estás duplicando código, pero lo harás de todas formas. La tercera vez que te enfrentes al mismo caso, refactoriza”.

Cuando se está programando una aplicación con TDD, como hemos visto anteriormente, el proceso de desarrollo se está dividiendo en dos acciones concretas: **programar** y **refactorizar**. Esto es, a medida que vamos creando nuevos métodos, vamos refactorizando el código para eliminar redundancias y en definitiva, hacer el código -del test- más legible y así obtener un mejor rendimiento. Pero no estamos refactorizando el SUT constantemente, puesto que éste, tiene un momento y lugar para ser refactorizado.

La refactorización del SUT, implica que lo primero que debemos hacer, es **cumplir el objetivo** (programar aquello que se necesita) **y luego refactorizar el código del SUT**, cada vez

⁸ En el béisbol, un strike es una anotación negativa para el bateador ofensivo, cuando la pelota no es lanzada hacia el diamante. Al tercer strike anotado, termina el turno del bateador.

que:

- Se agregue un nuevo método
- Se corrija un bug
- Se haga una revisión de código

Pero siempre, respetando la regla de “los tres strikes”. Una vez identificado el momento, solo será cuestión de identificar el problema a fin de poder elegir la solución indicada.

Una solución a cada problema

Como comentamos anteriormente, no haremos una extensa lista de problemas, sino que nos centraremos en problemas generales. Muchas de las soluciones sugeridas en este capítulo, han sido extraídas de *SourceMaking.com*⁹, sitio donde se puede encontrar una completa clasificación de problemas¹⁰ y sus respectivas soluciones¹¹. Como hemos hecho a lo largo del curso, iremos de lo general a lo particular y de lo particular al detalle.

Variables de uso temporal mal implementadas

En principio, definiremos a las variables de uso temporal, como aquellas variables que son asignadas en el ámbito local de un método de clase y son necesarias temporalmente, solo en ese método, sin ser llamadas o requeridas por otros métodos.

⁹ <http://sourcemaking.com/refactoring>. Nótese que algunas de las técnicas expuestas en el sitio Web referido, no se mencionan en este curso, por considerarlas poco apropiadas. Esto es debido a que algunas prácticas son más específicas de lenguajes como Java, mientras que a otras, las considero contrarias a las buenas prácticas de la programación orientada a objetos y por lo tanto, contraproducentes.

¹⁰ “Bad Smells in Code” <http://sourcemaking.com/refactoring/bad-smells-in-code>

¹¹ Diferentes técnicas de refactorización: <http://sourcemaking.com/refactoring>

Generalmente representan un problema en los siguientes casos:

1) Variables de uso temporal que definen una acción concreta:

```
$var = ($a * $b ) / (int)$c;
```

En el ejemplo anterior, vemos una variable de uso temporal, que define una acción concreta: dividir el producto de dos factores. Esto representa un problema, ya que las acciones son responsabilidad de los métodos y no de las variables. En estos casos, la solución, es **transferir la responsabilidad de la acción a un método**:

```
$var = dividir_producto($a, $b, $c);  
  
function dividir_producto($a, $b, $c) {  
    return ($a * $b ) / (int)$c;  
}
```

Nótese que variables de uso temporal que definen un valor directo: `$var = 15;` o por el retorno de la llamada a una función: `$var = strlen($variable);` no necesitan transferir su responsabilidad a otro método.

2) Variables de uso temporal son requeridas por más de un método:

```
function metodo_a() {  
    $a = 15;  
    $b = 100;  
    $c = 2;  
    $var = self::dividir_producto($a, $b, $c);  
    // continuar...
```

```
}

private static function dividir_producto($a, $b, $c) {
    return ($a * $b ) / $c;
}
```

En el ejemplo, anterior, las variables temporales \$a, \$b y \$c, son requeridas por dos métodos y se están definiendo como tales en un método, requiriendo ser pasadas como parámetros. Aquí, la solución, será **convertir las variables temporales, en propiedades de clase**:

```
function metodo_a() {
    self::$a = 15;
    self::$b = 100;
    self::$c = 2;
    $var = self::dividir_producto();
    // continuar...
}

private static function dividir_producto() {
    return (self::$a * self::$b ) / self::$c;
}
```

3) Variables de uso temporal que reasignan parámetros:

```
function foo($a) {
    $a = strtoupper($a);
    // continuar ...
}
```

En casos como éste, la confusión puede ser grande: un parámetro es un parámetro y una variable temporal, una variable temporal. Es entonces, cuando **variables temporales no deben tener el mismo nombre que los parámetros**:

```
function foo($a) {
    $b = strtoupper($a);
    // continuar ...
}
```

Métodos que reciben parámetros

Aquí debe hacerse una notable distinción entre parámetros, variables de uso temporal y propiedades de clase. Y esta distinción, está dada por la finalidad de cada una:

- Las **variables de uso temporal**, como hemos visto antes, están destinadas a definir un valor concreto al cual se hará referencia solo en el ámbito donde se haya definido.
- Las **propiedades de clase**, son características inherentes al objeto a las cuales se hará referencia desde diversos ámbitos.
- Y finalmente, los **parámetros**, serán valores adicionales, que no pueden ser considerados propiedades del objeto pero que sin embargo, son requeridos para que una acción, modifique las propiedades de un objeto.

```
Class Usuario {  
    function validar_usuario($username, $pass) {  
        if($username == 'pepe' && $pass == '123') {  
            return True;  
        }  
    }  
}
```

En el ejemplo anterior, claramente los parámetros \$username y \$pass, deberían ser propiedades del objeto Usuario puesto que son características intrínsecas al objeto. Como regla general, **los parámetros deben ser evitados toda vez que sea posible, reemplazándolos por propiedades de clase:**

```
Class Usuario {
```



```
function validar_usuario() {  
    if($this->username == 'pepe' && $this->pass == '123') {  
        return True;  
    }  
}  
}
```

Expresiones extensas

Muchas veces, podremos encontrarnos con expresiones que debido a su extensión, se hacen difíciles de leer y cuando no, confusas:

```
return ((in_array('abc', $array) || in_array('bcd', $array)) &&  
(in_array('cde', $array) || in_array('def', $array))) ? 'OK' : 'ERROR';
```

Cuando estamos en presencia de expresiones tan extensas, lo mejor es -aquí sí- **utilizar variables de uso temporal para simplificar dichas expresiones:**

```
$a = in_array('abc', $array);  
$b = in_array('bcd', $array);  
$c = in_array('cde', $array);  
$d = in_array('def', $array);  
$ab = ($a || $b);  
$cd = ($c || $d);  
  
return ($ab && $cd) ? 'OK' : 'ERROR';
```

Métodos extensos

No solo una expresión puede ser extensa. Muchas veces, nos encontraremos con métodos con extensos algoritmos que realizan varias acciones:

```
function renderizar_plantilla($data=array(), $pattern, $template) {  
    $ini_pattern = "[[INI-PATTERN-{$pattern}]]";  
    $end_pattern = "[[END-PATTERN-{$pattern}]]";  
    $plantilla = file_get_contents($template);
```

```

$pos_ini = strpos($plantilla, $ini_pattern);
$pos_fin = strpos($plantilla, $end_pattern);
$longitud_cadena = $pos_fin - $pos_ini;
$cadena = substr($plantilla, $pos_ini, $longitud_cadena);
$reemplazos = '';
foreach($data as $identificador=>$valor) {
    $reemplazos .= str_replace("[{$identificador}]", $valor, $cadena);
}
$resultado = str_replace($cadena, '[[NUEVO-CONTENIDO]]', $plantilla);
$resultado = str_replace('[[NUEVO-CONTENIDO]]', $reemplazos, $plantilla);
return $resultado;
}

```

Cuando existen métodos tan extensos, probablemente, la solución consista en la combinación de diversas técnicas, que van desde **agrupar expresiones en una misma línea** hasta evitar la asignación de variables temporales (como vimos al comienzo) y **extraer código llevándolo a diferentes métodos**:

```

function renderizar_plantilla($data=array()) {
    self::set_patterns();
    self::set_contenido_plantilla();
    self::get_pattern();
    self::reemplazar_datos($data);
    return str_replace('[[NEW]]', self::$reemplazos,
self::set_new_pattern());
}

// extracción de código para crear nuevo método
// y sustitución de parámetros por propiedades de clase
static function set_patterns() {
    self::$ini_pattern = "[[INI-PATTERN-{self::$pattern}]]";
    self::$end_pattern = "[[END-PATTERN-{self::$pattern}]]";
}

// extracción de código para crear nuevo método
// y sustitución de parámetros por propiedades de clase
static function set_contenido_plantilla() {
    self::$contenido = file_get_contents(self::$template);
}

// extracción de código para crear nuevo método
// sustitución de parámetros por propiedades de clase
// y sustitución de expresiones en línea
static function get_pattern() {
    self::$cadena = substr(self::$contenido, strpos(self::$ini_pattern),
        (self::$pos_fin - self::$pos_ini));
}

// extracción de código para crear nuevo método

```

```
// y sustitución de parámetros por propiedades de clase
static function reemplazar_datos($data=array()) {
    self::$reemplazos = '';
    foreach($data as $identificador=>$valor) {
        self::$reemplazos .= str_replace("[{$identificador}]", $valor, self::$scadena);
    }
}

// extracción de código para crear nuevo método
static function set_new_pattern() {
    return str_replace(self::$scadena, '[[NEW]]', self::$contenido);
}
```

Código duplicado en una misma clase

Es frecuente -y de lo más común-, que las mismas expresiones, comiencen a duplicarse en diferentes métodos de una misma clase:

```
function metodo_1() {
    $a = strip_tags(self::$propiedad);
    $a = htmlentities(self::$propiedad);
    return self::metodo_a() . self::$propiedad;
}

function metodo_2() {
    $a = strip_tags(self::$propiedad);
    $a = htmlentities(self::$propiedad);
    return self::$propiedad . self::metodo_b() . self::metodo_c();
}
```

Las expresiones duplicadas en el código de los diferentes métodos de una misma clase, se solucionan **extrayendo el código duplicado de los métodos, y colocándolo en un nuevo método de clase**:

```
function metodo_1() {
    self::metodo_3();
    return self::metodo_a() . self::$propiedad;
}

function metodo_2() {
    self::metodo_3();
    return self::$propiedad . self::metodo_b() . self::metodo_c();
}
```

```
static function metodo_3() {
    self::$propiedad = strip_tags(self::$propiedad);
    self::$propiedad = htmlentities(self::$propiedad);
}
```

Código duplicado en varias clases con la misma herencia

El caso anterior puede darse también, cuando el código se encuentra duplicado en diferentes métodos de clases con la misma herencia:

```
class B extends A {
    function metodo_1() {
        $a = strip_tags(self::$propiedad);
        $a = htmlentities(self::$propiedad);
        return self::metodo_a() . self::$propiedad;
    }
}

class C extends A {
    function metodo_2() {
        $a = strip_tags(self::$propiedad);
        $a = htmlentities(self::$propiedad);
        return self::$propiedad . self::metodo_b() . self::metodo_c();
    }
}
```

En estos casos, en los cuáles existen dos o más clases que heredan de la misma clase, **se extrae el código duplicado en los métodos de las clases hijas, y con éste, se crea un nuevo método de en la clase madre:**

```
class A {
    static function metodo_3() {
        self::$propiedad = strip_tags(self::$propiedad);
        self::$propiedad = htmlentities(self::$propiedad);
    }
}
```

```
class B extends A {  
    function metodo_1() {  
        self::metodo_3();  
        return self::metodo_a() . self::$propiedad;  
    }  
}  
  
class C extends A {  
    function metodo_2() {  
        self::metodo_3();  
        return self::$propiedad . self::metodo_b() . self::metodo_c();  
    }  
}
```

Código duplicado en varias clases sin la misma herencia

Como era de esperarse, el código también podrá aparecer duplicado en diferentes clases pero que no tienen la misma herencia:

```
class B {  
    function metodo_1() {  
        $a = strip_tags(self::$propiedad);  
        $a = htmlentities(self::$propiedad);  
        return self::metodo_a() . self::$propiedad;  
    }  
}  
  
class C {  
    function metodo_2() {  
        $a = strip_tags(self::$propiedad);  
        $a = htmlentities(self::$propiedad);  
        return self::$propiedad . self::metodo_b() . self::metodo_c();  
    }  
}
```

En estos casos, la solución es **extraer el código duplicado, crear una nueva clase y con el código extraído, crear un**

método para esta nueva clase que podrá ser heredada por las anteriores o simplemente, instanciada:

```
class A {  
    static function metodo_3($parametro) {  
        return htmlentities(strip_tags($parametro));  
    }  
}  
  
class B {  
    function metodo_1() {  
        return self::metodo_a() . A::metodo_3(self::$propiedad);  
    }  
}  
  
class C {  
    function metodo_2() {  
        return A::metodo_3(self::$propiedad) . self::metodo_b() .  
        self::metodo_c();  
    }  
}
```

Introducción a la Arquitectura de Software

¿Qué es la arquitectura de software?

Es necesario aclarar, que no existe una definición única, exacta, abarcadora e inequívoca de “arquitectura de software”. La bibliografía sobre el tema es tan extensa como la cantidad de definiciones que en ella se puede encontrar. Por lo tanto trataré, no de definir la arquitectura de software, sino más bien, de introducir a un concepto simple y sencillo que permita comprender el punto de vista desde el cual, este libro abarca a la arquitectura de software pero, sin ánimo de que ello represente “una definición más”.

A grandes rasgos, puede decirse que “la Arquitectura de Software es la forma en la que se organizan los componentes de un sistema, interactúan y se relacionan entre sí y con el contexto, aplicando normas y principios de diseño y calidad, que fortalezcan y fomenten la usabilidad a la vez que dejan preparado el sistema, para su propia evolución”.

Atributos de calidad

La Calidad del Software puede definirse como los atributos implícitamente requeridos en un sistema que deben ser satisfechos. Cuando estos atributos son satisfechos, puede decirse (aunque en forma objetable), que la calidad del software es satisfactoria. Estos atributos, se gestan desde la

arquitectura de software que se emplea, ya sea cumpliendo con aquellos requeridos durante la ejecución del software, como con aquellos que forman parte del proceso de desarrollo de éste.

Atributos de calidad que pueden observarse durante la ejecución del software

1. Disponibilidad de uso
2. Confidencialidad, puesto que se debe evitar el acceso no autorizado al sistema
3. Cumplimiento de la Funcionalidad requerida
4. Desempeño del sistema con respecto a factores tales como la capacidad de respuesta
5. Confiabilidad dada por la constancia operativa y permanente del sistema
6. Seguridad externa evitando la pérdida de información debido a errores del sistema
7. Seguridad interna siendo capaz de impedir ataques, usos no autorizados, etc.

Atributos de calidad inherentes al proceso de desarrollo del software

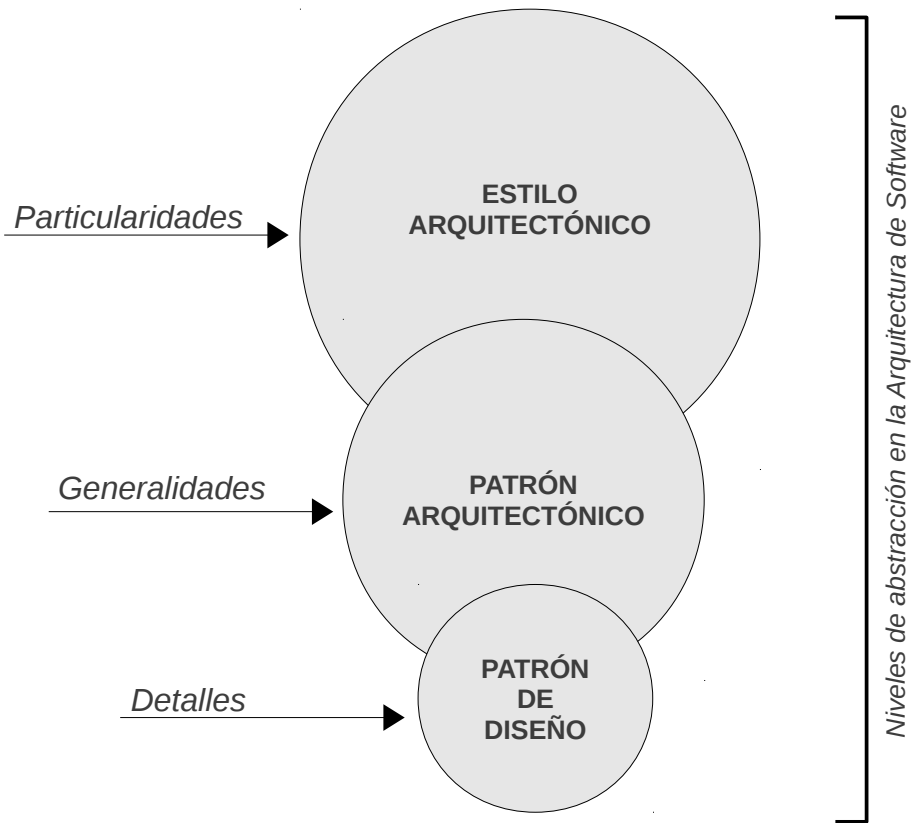
1. Capacidad de Configurabilidad que el sistema otorga al usuario a fin de realizar ciertos cambios
2. Integrabilidad de los módulos independientes del sistema
3. Integridad de la información asociada
4. Capacidad de Interoperar con otros sistemas (interoperabilidad)
5. Capacidad de permitir ser modificable a futuro (modificabilidad)

6. Ser fácilmente Mantenable (mantenibilidad)
7. Capacidad de Portabilidad, es decir que pueda ser ejecutado en diversos ambientes tanto de software como de hardware
8. Tener una estructura que facilite la Reusabilidad de la misma en futuros sistemas
9. Mantener un diseño arquitectónico Escalable que permita su ampliación (escalabilidad)
10. Facilidad de ser Sometido a Pruebas que aseguren que el sistema falla cuando es lo que se espera (testeabilidad)

Niveles de abstracción

Podemos decir que la AS se compone de tres niveles de abstracción bien diferenciados: Estilo Arquitectónico, Patrón Arquitectónico y Patrón de Diseño. Existe una diferencia radical entre estos tres elementos, que debe marcarse a fin de evitar las grandes confusiones que inevitablemente, concluyen en el mal entendimiento y en los resultados poco satisfactorios. Éstos, son los que en definitiva, aportarán “calidad” al sistema resultante. En lo sucesivo, trataremos de establecer la diferencia entre estos tres conceptos, viendo como los mismos, se relacionan entre sí, formando parte de un todo: la arquitectura de software.

Estilo Arquitectónico, Patrón Arquitectónico y Patrón de Diseño, representan -de lo general a lo particular- los tres niveles de abstracción que componen la Arquitectura de Software.



Estilo Arquitectónico

El estilo arquitectónico define a niveles generales, la estructura de un sistema y cómo éste, va a comportarse. Mary Shaw y David Garlan, en su libro “Software Architecture” (Prentice Hall, 1996), definen los estilos arquitectónicos como la forma de determinar el los componentes y conectores de un sistema, que pueden ser utilizados a instancias del estilo elegido, conjuntamente con un grupo de restricciones sobre como éstos pueden ser combinados:

“[...] an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined [...]”

Mary Shaw y David Garlan -en el mismo libro-, hacen una distinción de estilos arquitectónicos comunes, citando como tales a:

1. Pipes and filters (filtros y tuberías)
2. Data Abstraction and Object-Oriented Organization (Abstracción de datos y organización orientada a objetos)
3. Event-based (estilo basado en eventos)
4. Layered Systems (Sistemas en capas)
5. Repositories (Repositorios)
6. Table Driven Interpreters

Viendo la clasificación anterior, es muy frecuente que se encuentren relaciones entre los estilos arquitectónicos y los paradigmas de programación. Sin embargo, debe evitarse relacionarlos en forma directa.

Patrón Arquitectónico

Un patrón arquitectónico, definirá entonces, una plantilla para construir el Software, siendo una particularidad del estilo arquitectónico elegido.

En esta definición, es donde se incluye a MVC, patrón que a la vez, puede ser enmarcado dentro del estilo arquitectónico orientado a objetos (estilo arquitectónico basado en el paradigma de programación orientada a objetos).

Patrón de Diseño

Dentro de niveles de abstracción de la arquitectura de Software, los patrones de diseño representan el nivel de abstracción más detallado. A nivel general, nos encontramos con el Estilo Arquitectónico. En lo particular, hallamos al Patrón Arquitectónico y, finalmente, el Patrón de Diseño es “el detalle”.

Matt Zandstra en su libro “PHP Objects, Patterns and Practice” (Apress, 2010) define los patrones de diseño como:

“[...] is a problem analyzed with good practice for its solution explained [...]” (un problema analizado con buenas prácticas para su solución explicada)

Un patrón de diseño, entonces, es un análisis mucho más detallado, preciso y minucioso de una parte más pequeña del sistema, que puede incluso, trabajar en interacción con otros patrones de diseño. Por ejemplo, un *Singleton* puede coexistir con un *Factory* y éstos, a la vez, con un *Abstract Factory*.

En este sentido, un Patrón Arquitectónico como MVC, podría utilizar diversos patrones de diseño en perfecta coexistencia, para la creación de sus componentes.

Introducción al Patrón Arquitectónico MVC

MVC -por sus siglas en inglés, model-view-controller (modelo-vista-controlador)- es un patrón arquitectónico que nos permite desarrollar aplicaciones, manteniendo separada la lógica de negocios de las vistas, utilizando un “controlador” como conector (o intermediario) entre ambas.

Entendiendo el funcionamiento de MVC

En MVC, todo comienza con una petición del usuario. En una aplicación Web, la petición del usuario podría ser, por ejemplo, “agregar un nuevo registro”.

- **¿Cómo realiza esta petición el usuario?** A través del navegador.
- **¿Cómo se identifica la petición?** Por medio de la URL ingresada por el usuario.

Un ejemplo:

Tenemos una aplicación Web, cuyos módulos son:

Módulo de Usuarios

Módulo de Proyectos

Cada módulo, a la vez, se encontrará dividido en “modelos” (objetos):

Módulo de Usuarios

- Modelo Usuario
- Modelo Permiso

Y estos modelos, ofrecerán diversos recursos (funcionalidades de cada modelo):

Módulo de Usuarios

- Modelo Usuario
 - Recursos:
 - Agregar Usuario
 - Modificar Usuario
 - Eliminar Usuario
 - Obtener Usuario

Las peticiones del usuario, entonces, se realizarán vía navegador, siendo descifradas por la URL ingresada, la cual, guardará un formato sugerido como el siguiente:

dominio/modulo/modelo/recurso[/atributos]

Esto significa, que si el usuario desea agregar un nuevo usuario, su petición, debería ser:

```
http://app.dominio.com/usuarios/usuario/agregar-usuario/
```

- **¿A quién efectúa la petición el usuario?** Al controlador.
- **¿Cómo maneja el controlador, la petición del usuario?** A través de *Handler*.

Un *handler* (o “manejador”) de peticiones, es un objeto encargado de gestionar las peticiones del usuario a nivel de la aplicación. Este *handler*, descifrará dichas peticiones, realizando un trabajo de pseudo ingeniería inversa, sobre la URI.

Por ejemplo, para la URI anterior, el handler podría realizar los siguientes pasos:

1) Identificar el dominio, eliminarlo del contexto y así solo obtener el último tramo de la URI:

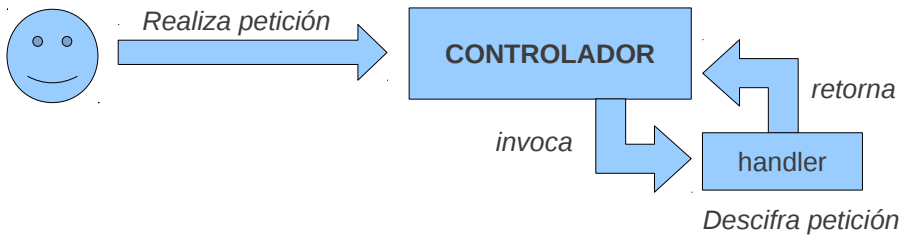
```
$dominio = "http://{$_SERVER['SERVER_NAME']}";  
$uri = "{$dominio}{$_SERVER['REQUEST_URI']}";  
$ultimo_tramo = str_replace "{$dominio}/", NULL, $uri);
```

2) Hacer un explode del último tramo de la URI, para obtener

un array de 3 elementos (módulo, modelo y recurso respectivamente):

```
$partes = explode("/", $ultimo_tramo);
```

3) Finalmente, el objeto handler, retornará al controlador, el módulo, el modelo y el recurso solicitados por el usuarios, y será el controlador, quien prosiga con el resto.



- **¿Qué hace el controlador una vez que recibe la información retornada por el handler?** La analiza para saber a que modelo deberá instanciar.

Una vez que el controlador recibe la información retornada por el *handler*, procede a analizarla en conjunto.

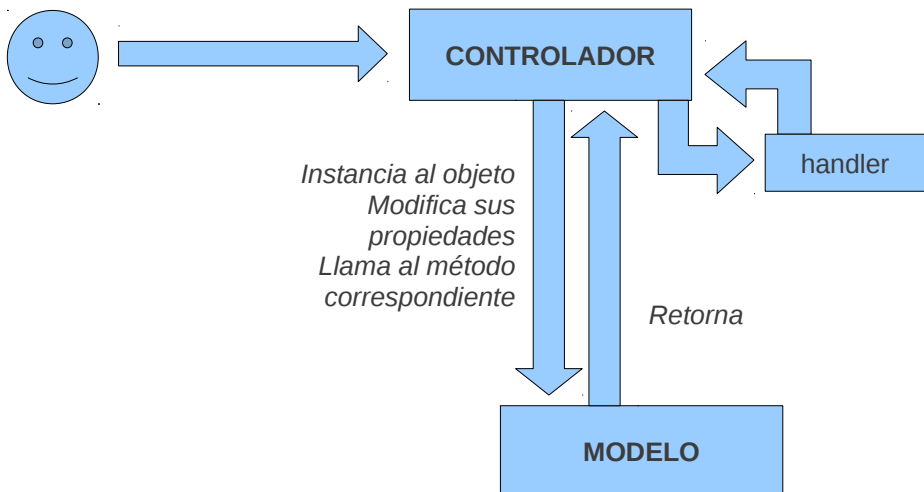
El handler me envió la siguiente información:

Módulo: usuarios

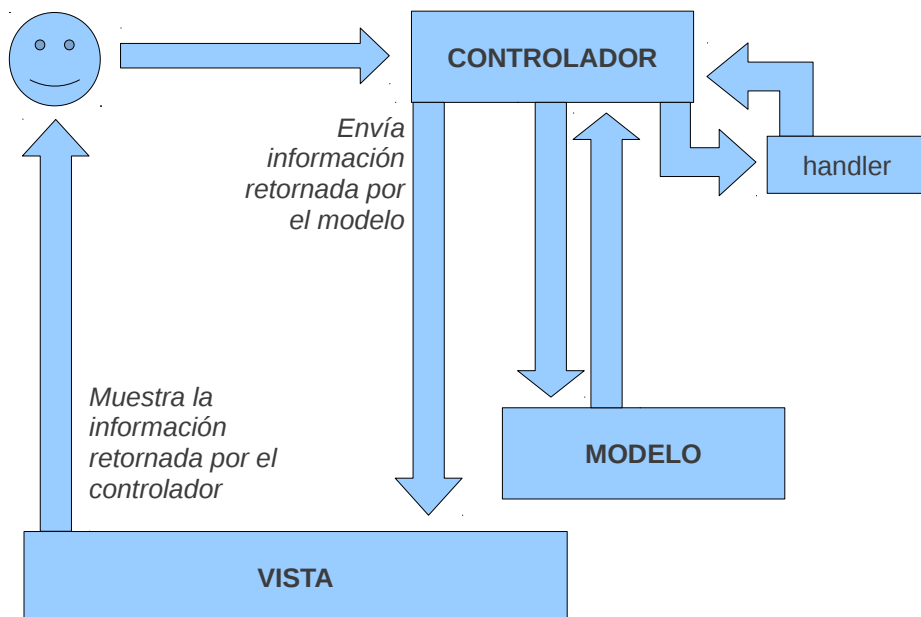
Modelo: usuario

Recurso: agregar-usuario

Entonces, debo agregar un nuevo usuario. Para ello, voy a instanciar el objeto, modificar sus propiedades y llamar al método correspondiente.



- **¿Qué hace el controlador con la información retornada por el modelo?** La entrega a la vista.
- **¿Para qué el controlador le entrega la información a la vista?** Para que ésta, se la muestre al usuario.



Modelos en MVC

Un modelo, en MVC, es una clase. Como patrón que guía nuestras arquitecturas, MVC nos obliga a escribir clases “puras” que respeten el verdadero espíritu de la orientación a objetos.

Objetos puros: características de un modelo

1. Una clase, debe representar solo y únicamente, un “modelo” para crear un objeto único.
2. Las propiedades de un objeto, siempre deberán ser -solo y únicamente- “sustantivos” cuyo valor, pueda ser definido por una cualidad o más de una.
3. Cuando una propiedad, sea definida por más de una cualidad, deberá considerarse la opción de componer el objeto: dicha propiedad, adquirirá las cualidades de otro objeto.
4. Las propiedades no compuestas del objeto -mayormente-, deben poder modificarse de forma directa, sin necesidad de requerir la llamada a un método.
5. Las propiedades compuestas -mayormente-, deberán contener un método -en la clase- del tipo `set_propiedad(Objeto $objeto)` para ser modificadas.
6. Los métodos de un objeto, deberán ser “acciones” intrínsecas del objeto y jamás, podrán representar acciones genéricas, que puedan ser consideradas ajenas al objeto. Por ejemplo, un objeto jamás podría contener entre sus métodos, una acción destinada a filtrar datos para prevenir inyecciones SQL.

7. Solo los modelos, podrán interactuar con la base de datos. No se podrá acceder a una base de datos, desde un ámbito externo a los modelos.
8. Preferentemente, cada modelo accederá a la base de datos, mediante otro objeto- conector (capa de abstracción que pertenecerá al núcleo de la aplicación).

Creando modelos bajo MVC en PHP

Herencia y Composición

Piensa en una camisa como un objeto. ¿Qué propiedades podemos encontrar? Una camisa tendrá un tipo de tela determinado con un color específico, botones, un talla, etc. ¿cierto? Podríamos decir que la clase para crear una camisa, podría tener las siguientes propiedades:

```
class Camisa {  
    public $tela = "seda";  
    public $botones = 8;  
    public $color = "blanca";  
    public $talle = 42;  
}
```

Imagínate que trabajas en el salón de ventas de una camisería y un cliente te pregunta ¿De qué material son los botones de la camisa? ¿Son botones ciegos o con ojales? ¿qué color son? Podríamos deducir, que cada uno de los 8 botones, tiene características particulares:

```
boton.material = "nylon"
```

```
boton.color = "marfil"  
boton.ojales = 2
```

Cuando hablamos de una propiedad con múltiples cualidades, como Arquitectos, debemos pensar -en principio- en dos opciones: **herencia** o **composición**.

Un botón ¿no es acaso un objeto con atributos propios? Es allí, donde probablemente, comiencen a surgir dudas. La primera pregunta que deberás hacerte, entonces, es: ¿Qué relación existe entre Camisa y Botón? Para responder a esta pregunta, deberás hacerlo formulando nuevas incógnitas:

- ¿Es Objeto A una extensión ampliada de Objeto B? Si la respuesta es sí, pensarás en herencia. Si la respuesta es no, te harás la siguiente pregunta:
- Objeto B ¿forma parte integrante de Objeto A? Si la respuesta es sí, pensarás entonces en composición.

Entonces, la pregunta es: ¿Es Camisa una extensión ampliada de Botón? La respuesta clara es "No". Pues Camisa y Botón, nada tienen en común. Sin embargo, Botón, forma parte integrante de Camisa:

```
# archivo: models/camisa.php  
  
class Camisa {  
    public $tela = "seda";  
    public $botones = array();  
    public $color = "blanca";  
    public $talle = 42;  
  
    public function set_botones(Boton $boton, $cuantos_botones) {  
        $iterar = 0;  
        while($iterar < $cuantos_botones) {
```

```

        $this->botones[] = $boton;
        $iterar++;
    }
}

# archivo: models/boton.php

class Boton {
    public $material = "nylon";
    public $color = "marfil";
    public $ojales = 2;
}

# archivo: crear_camisa.php

$camisa = new Camisa();
$camisa->set_botones(new Boton(), 8);

$botones = count($camisa->botones);

if($camisa->botones[0]->ojales < 1) {
    $tipo_boton = "ciego";
} else {
    $tipo_boton = "ojalado";
}

print "La camisa de {$camisa->tela} color {$camisa->color}
tiene {$botones} botones de {$camisa->botones[0]->material} {$tipo_boton}s
en color {$camisa->botones[0]->color}";

/*
Salida:
La camisa de seda color blanca tiene 8 botones de nylon ojalados en color
marfil
*/

```

Acceso a bases de datos

Un “objeto-conector” a nivel del *core*, que actúe como intermediario entre los modelos y la base de datos, será muy recomendable para respetar el verdadero espíritu del patrón arquitectónico MVC.

Este objeto, será una clase preferentemente estática. Es decir, que necesitará ser instanciada para hacer uso de los métodos destinados a acceder a las bases de datos.

Código fuente de una capa de abstracción a nivel del core

```
class DBObject {

    protected static $conn;
    protected static $stmt;
    protected static $reflection;
    protected static $sql;
    protected static $data;
    public static $results;

    protected static function conectar() {
        self::$conn = new mysqli(DB_HOST, DB_USER, DB_PASS, DB_NAME);
        self::$conn->autocommit(False);
    }

    protected static function preparar() {
        self::$stmt = self::$conn->prepare(self::$sql);
        self::$reflection = new ReflectionClass('mysqli_stmt');
    }

    protected static function set_params() {
        $method = self::$reflection->getMethod('bind_param');
        $method->invokeArgs(self::$stmt, self::$data);
    }

    protected static function get_data($fields) {
        $method = self::$reflection->getMethod('bind_result');
        $method->invokeArgs(self::$stmt, $fields);
        while(self::$stmt->fetch()) {
            self::$results[] = unserialize(serialize($fields));
        }
    }

    protected static function finalizar() {
        self::$stmt->close();
        self::$conn->close();
    }

    public static function ejecutar($sql, $data, $fields=False) {
        self::$sql = $sql;
        self::$data = $data;
        self::conectar();
        self::preparar();
        self::set_params();
        self::$stmt->execute();
        if($fields) {
            self::get_data($fields);
        } else {
            self::$conn->commit();
        }
        self::finalizar();
    }
}
```

```
}

```

Implementación desde el modelo:

```
require_once('/app/core/db_object.php');

class Boton {

    public $material = NULL;
    public $color = NULL;
    public $ojales = NULL;
    public $boton_id = 0;

    public function save() {
        $sql = "INSERT INTO boton (material, color, ojales) VALUES
(?, ?, ?)";

        $data = array("ssi",
                        "{$this->material}",
                        "{$this->color}",
                        "{$this->ojales}");

        return DBObject::ejecutar($sql, $data);
    }

    public function get() {
        $sql = "SELECT material, color, ojales FROM boton WHERE boton_id
> ?";

        $data = array("i", "{$this->boton_id}");

        $fields = array("Material"=>"",
                        "Color"=>"",
                        "Ojales"=>"");

        DBObject::ejecutar($sql, $data, $fields);
        return DBObject::$results;
    }
}
```

Object Relational Mapping (ORM)

Si observamos la clase Boton, notaremos que el método `save()`, declara él mismo la sentencia SQL que será utilizada por `DBObject` para agregar registros en la base de datos. Pero una capa de abstracción a bases de datos, podría -y debería- otorgar una abstracción mucho más completa.

El **Mapeo Relacional de Objetos** (Object Relational Mapping) es una técnica de programación que nos permite vincular los objetos (modelos) de nuestra aplicación, con una base de datos relacional, otorgándonos de manera colateral, una abstracción completa entre los modelos y el motor de la base de datos.

Ventajas y desventajas del utilizar ORMs

Sin dudas, las principales **ventajas del ORM** -en orden de prioridades-, son:

1. **Independizar los modelos de las bases de datos**, lo cual nos obliga a razonar nuestra aplicación, 100% orientada a objetos (ya no pensaremos en como crear las tablas, solo crearemos nuestros modelos -modelar la app-).
2. **Independencia del motor de base de datos:** este punto es fundamental, ya que un buen ORM debe ser capaz de mapear los modelos de forma tal que las llamadas a sus métodos, sean capaces de conectarse a cualquier tipo de base de datos y general consultas con el lenguaje SQL de la misma. Esta característica, es la que permite una mayor portabilidad de la aplicación.
3. **Acelera el proceso de desarrollo**, puesto que se ahorra tiempo tanto en la creación y diseño de las bases de datos como en la escritura de las consultas.

Sin embargo, la utilización de ORM, trae aparejadas ciertas **desventajas** que no pueden ser obviadas al momento de tomar una decisión:

- **Lenguaje de consulta propio:** la mayoría de las

librerías ORM disponibles en el mercado, poseen su propio lenguaje de consulta a bases de datos. Esta característica, que erróneamente es considerada una ventaja por muchos autores (debido al escaso conocimiento del lenguaje SQL) genera una pseudo-abstracción (o abstracción falaz) que induce a errores en el modelado de objetos, puesto que estos pseudo-lenguajes que ofrecen los ORM, no son más que una especie de “lenguaje SQL resumido”.

- **Reduce el rendimiento de la aplicación**, puesto que el proceso de conversión desde el pseudo-lenguaje al lenguaje SQL de la base de datos y el de ésta a objetos, demanda un mayor consumo de recursos.
- **Baja escalabilidad**: también derivado del uso de pseudo-lenguajes de consulta propio, cuanto más robusta y compleja se va haciendo una aplicación, mayor necesidad tendrá el programador, de escribir consultas en lenguaje SQL “crudo”.

ORM Frameworks para PHP

En PHP, existen prestigiosos frameworks ORM, que podríamos utilizar en nuestra aplicaciones. Entre los más destacados, podremos encontrar:

- **Doctrine**: <http://www.doctrine-project.org/>
- **Propel**: <http://www.propelorm.org/>
- Entre otros.

Bibliografía recomendada

Para **complementar la lectura sobre modelos en PHP**, se recomienda leer bibliografía sobre:

- Programación orientada a objetos
- Patrones de diseño

Libros recomendados:

- PHP Objects, Patterns and Practice (Third Edition) – Matt Zandstra (Apress, 2010)
- Pro PHP – Patterns, Frameworks, Testing and More (Part I: OOP and Patterns) – Kevin McArthur (Apress, 2008)
- Pro PHP Programming (Chapter I:Object Orientation) – Peter MacIntyre, Brian Danchilla & Mladen Gogala (Apress, 2011)

Las vistas

Embeber código HTML, CSS y JavaScript en el código PHP, con MVC queda en el olvido. Sin embargo, muchos Frameworks MVC para PHP, aún conservan esa mala costumbre que los programadores, solemos arrastrar de la programación estructurada.

Basaremos este curso, en el espíritu original y más desarrollado de MVC, el cual nos propone una completa abstracción de las vistas, subdividiendo éstas, en la parte lógica (código PHP que se encargará de hacer un *render* de la parte gráfica) y en la GUI (parte gráfica inherente al diseño gráfico), incorporando algunos *Katas* para no volver a embeber código y lograr mantener un verdadero diseño orientado a objetos.

¿Por dónde empezar a desarrollar las vistas?

Lo primero que debemos tener en cuenta, es la completa abstracción de la lógica de su correspondiente GUI (Grafical User Interface).

El diseño gráfico es un verdadero arte, que abarca múltiples disciplinas y que, como programadores, el peor error que podemos cometer, es querer convertir al diseño gráfico en un “producto científico”, ya que **la lógica utilizada en el arte, debe indefectiblemente estar sometida al ingenio y creatividad**, puesto que **de lo contrario, se desaprovecharían todas las virtudes que el arte**, tiene para ofrecernos a fin de hacer nuestro Software “más humano”.

La GUI de nuestras aplicaciones, debe -sí y solo sí- basarse en cuestiones inherentes al arte y la creatividad, al servicio de la usabilidad y experiencia del usuario.

El primer paso, entonces, consistirá en crear las interfaces gráficas de la aplicación. Es sumamente importante, considerar la posibilidad de contar con artistas expertos en el Diseño Gráfico y experiencia del usuario, puesto que son los únicos que cuentan con la capacidad necesaria y la autoridad profesional suficiente, para crear verdaderas GUI.

En este curso, no haremos demasiado énfasis -ni ahondaremos- en temas relativos al diseño gráfico, puesto que el curso, está basado en la AS. Los ejemplos que utilizaremos, lejos están de ser tomados como parámetro de arte aplicado al diseño gráfico. Nos enfocaremos entonces, solo y únicamente, en aquellos aspectos que como programadores y/o arquitectos, debemos considerar transmitir a los profesionales encargados de diseñar la GUI de nuestras aplicaciones.

Desarrollando la GUI

Componentes de la GUI

La GUI deberá estar compuesta, por todos aquellos archivos -y datos- estáticos, que son ejecutados del lado del cliente. Entre ellos, nos encontramos con archivos **HTML, CSS, JS, imágenes, archivos de audio y video y cualquier otro tipo de documento estático** (como PDFs, ODTs, etc...).

Arquitectura

Refiriéndonos a Arquitectura como “Arquitectura de Software” (AS) y no, como Arquitectura de la Información (AI), la GUI de nuestra aplicación, deberá estar concentrada en un solo lugar. Esto puede ser:

- Un directorio exclusivo dentro de la App

```
myapp/
└─ site_media /
```

- Un servidor independiente del servidor de la aplicación

La **estructura más recomendada** que puede tener la GUI, es aquella que siga una organización como la que sigue:

```
myapp/
└─ site_media /
    │
    │ └─ css/
    │   │
    │   │ └─ core/           Archivos CSS aplicables a toda
    │   │   │ la app
    │   │   │
    │   │   │ └─ module-a/   Archivos CSS pertenecientes
    │   │   │   │ solo al módulo A
    │   │   │   │
    │   │   │   └─ module-b/ Archivos CSS pertenecientes
    │   │   │       │ solo al módulo B
    │   │   └─ html/
    │   │       │
    │   │       │ └─ core /
    │   │       │
    │   │       │ └─ module-a /
    │   │       │
    │   │       │ └─ module-b /
    │   └─ img/
    │       │
    │       │ └─ core /
```

```
|   ├── module-a /  
|   └── module-b /  
└── js/  
    ├── core /  
    ├── module-a /  
    └── module-b/
```

Esta estructura de directorios, si bien **es la más escalable** de todas, lógicamente podrá variar de acuerdo a las necesidades de cada app. Por ejemplo, también podrá incluir otros directorios destinados a almacenar archivos de audio, de video, PDFs, etc.

Preparando la GUI para interactuar con la lógica

Los archivos de la GUI (más específicamente, aquellos destinados como maqueta de nuestra app -generalmente, archivos HTML), solo podrán contener lenguaje de marcado| diseño exclusivamente. Pero ¿Qué sucede con aquellos datos que deberán sustituirse dinámicamente?

```
<!doctype html>  
<head>  
  <meta charset="utf-8">  
  <title>Listado de Camisas</title>  
</head>  
  
<body>  
  <header>  
    <h1>MyApp: Camisas</h1>  
    <nav>  
      <ul>  
        <li><a href="menu-option-1"  
          title="Menu Option 1">Ver camisas</a></li>  
        <li><a href="menu-option-2"  
          title="Menu Option 2">Agregar camisa nueva</a></li>  
      </ul>  
    </nav>
```

```
</header>
<section>
  <header>
    <h2>Camisas creadas</h2>
  </header>
  <table>
    <tr>
      <th>ID</th>
      <th>Camisa</th>
      <th>Tela</th>
    </tr>
    <tr>
      <td>1</td>
      <td>Tombolini Home</td>
      <td>Seda italiana blanca</td>
    </tr>
  </table>
</section>
</body>

</html>
```

Si miramos el código anterior, observando el texto en **negritas**, podremos notar que se trata de un *template*, esos datos, deberán ser plasmados dinámicamente. Es en estos casos, donde el diseñador, deberá utilizar “comodines” que luego, la lógica de esa vista, se encargará de renderizar.

Comodines ¿Qué son y cómo implementarlos?

Un comodín, puede ser cualquier texto plano que nos ayude a identificar -como programadores, en nuestra lógica de negocios- aquellos datos que necesitan ser sustituidos dinámicamente.

Estos comodines, deberán seguir un patrón que, por un lado, permita al diseñador tener una vista previa real de su diseño y por otro, permita a nuestra lógica de negocios, identificarlos con facilidad.

Algunos ejemplos:

[TEXT0 IDENTIFICADOR]

`{TEXTO IDENTIFICADOR}`

Lo importante, es que en toda la GUI, **siempre** se utilice el mismo patrón:

```
<!doctype html>
<head>
  <meta charset="utf-8">
  <title>{TITULO DE PAGINA}</title>
</head>

<body>
  <header>
    <h1>MyApp: {MODULO}</h1>
    <nav>
      {MENU}
    </nav>
  </header>
  <section>
    <header>
      <h2>{SUBTITULOS}</h2>
    </header>
    {TABLA}
  </section>
</body>

</html>
```

{TABLA} y **{MENU}** serán a la vez, dos nuevos archivos HTML. En el segundo caso (menú), será sencillo:

```
<ul>
  <li><a href="ver-camisas">Ver camisas</a></li>
  <li><a href="crear-camisa">Crear camisa</a></li>
</ul>
```

Pero en el caso de la tabla **¿cómo haremos para obtener identificadores iterativos?** Si miramos el código de la tabla que listará las camisas, podremos notar que la fila destinada a la descripción de cada registro, deberá repetirse tantas veces como registros se encuentren:

```

<table>
  <tr>
    <th>ID</th>
    <th>Camisa</th>
    <th>Tela</th>
  </tr>
  <!-- a partir de aquí los datos deben iterar -->
  <tr>
    <td>{CAMISA ID}</td>
    <td>{CAMISA DESCRIPCION}</td>
    <td>{CAMISA DATOS TELA}</td>
  </tr>
  <!-- desde aquí, ya no iteran -->
</table>

```

De la misma forma que en el código anterior, un comentario explica que esa fila debe ser iterada, será como los diseñadores, deberán especificárnoslo a nosotros, solo que de forma más simple y siguiendo algún patrón, como por ejemplo:

Para identificar dónde inicia una iteración:

```

<!-- iniciar-loop: NOMBRE DE IDENTIFICADOR DEL LOOP
-->

```

Para identificar dónde finaliza:

```

<!-- finalizar-loop: NOMBRE DE IDENTIFICADOR DEL
LOOP -->

```

Por ejemplo, el diseñador podría hacer lo siguiente:

```

<!-- iniciar-loop: CAMISA -->
<tr>
  <td>{CAMISA ID}</td>
  <td>{CAMISA DESCRIPCION}</td>
  <td>{CAMISA DATOS TELA}</td>
</tr>
<!-- finalizar-loop: CAMISA -->

```

Diseñando la lógica de negocios

La lógica de negocios de las vistas, es aquella que se encargará de traer el contenido estático y sustituir los “comodines” dinámicamente, con los datos que le sean entregados por el controlador.

La lógica de negocios, tendrá un **diseño distribuido**:

- Una parte -reutilizable y común a toda la aplicación- a nivel del *core*
- Otra parte, a nivel del módulo, mediante la cual, cada modelo tendrá su propia lógica

```
myapp/  
├─ core  
│   └─ view.php  
├─ module  
│   ├── models  
│   │   ├── boton.php  
│   │   └─ camisa.php  
│   └─ views  
│       ├── boton.php  
│       └─ camisa.php  
└─ site_media
```

La lógica principal, será la del *core*. Ésta, consistirá en una librería que nos provea de los métodos necesarios para:

- Traer los *templates*

- Identificar comodines e iteraciones
- Sustituir los comodines dinámicamente
- Retornar el *render* del *template*

La lógica de cada modelo, será una extensión de la lógica principal, con métodos propios que requieran de los métodos heredados de la clase principal, para que cada *render*, sea específico y particular.

Lógica principal

Como comentamos anteriormente, la lógica principal será una librería -una clase- con métodos que se encarguen, a niveles generales, de traer *templates* y renderizarlos. Un ejemplo de ello, sería la siguiente clase (a nivel del *core*), llamada **RenderTemplate**:

```
# archivo /myapp/core/render_template.php

class RenderTemplate {

    public $file = "";
    public $data = array();
    protected $comodines = array();
    protected $values = array();
    public $html = "";

    # Traer contenido HTML de una plantilla
    protected function get_html($str=NULL) {
        return isset($str) ? $str : file_get_contents($this->file);
    }

    # Setear comodines y valores
    public function set_data($data=array()) {
        $this->comodines = array_keys($data);
        $this->values = array_values($data);
        $this->set_comodines();
    }

    # Modificar comodines (envolver entre llaves)
    private function set_comodines() {
        foreach($this->comodines as &$comodin) {
            $comodin = "{" . $comodin . "}";
        }
    }
}
```

```
}

# Renderizar plantilla
public function render_template($str=NULL) {
    $this->html .= str_replace($this->comodines,
                             $this->values,
                             $this->get_html($str));
}

}
```

Esta clase, podrá ser heredada, por cualquier vista, pudiendo acceder y modificar cualquier propiedad o método, público o protegido.

Veamos un ejemplo de implementación en detalle:

Tenemos la siguiente plantilla HTML:

```
<!-- archivo: /myapp/site_media/html/template.html -->
<!doctype html>
<head>
    <meta charset="utf-8">
    <title>{TITULO}</title>
</head>

<body>
    <header>
        <h1>MyApp: {MODULO}</h1>
    </header>
    <section>
        <header>
            <h2>{SUBTITULO}</h2>
        </header>
        {CONTENIDO}
    </section>
</body>

</html>
```

De la plantilla anterior, necesitamos reemplazar 4 comodines: TITULO, MODULO, SUBTITULO y CONTENIDO. Hasta ahora, solo contamos con un modelo (Boton) y la lógica de la vista a nivel del core, RenderTemplate. No tenemos ni un controlador, ni la lógica de la vista para el modelo, así que probaremos

implementar la clase `RenderTemplate` desde un archivo de pruebas ubicado en `/myapp/module/prueba.php`.

Lo primero que necesitaremos, será importar la clase `RenderTemplate`:

```
require_once("../core/render_template.php");
```

El paso siguiente, será definir un *array* asociativo, donde las claves sean los comodines y los valores, los datos por los cuales esos comodines, deberán ser reemplazados:

```
$data = array("TITULO" => "Administración de Camisas",  
             "MODULO" => "Camisas",  
             "SUBTITULO" => "Listado de Botones",  
             "CONTENIDO" => "Aquí debo mostrar el listado de botones");
```

Finalmente, tendremos que:

```
# Crear una instancia de RenderTemplate  
$view = new RenderTemplate();  
  
# Preparar los datos a ser renderizados  
$view->set_data($data);
```

El método `set_data` de la clase `RenderTemplate`, se encargará de:

- Dividir el *array* asociativo en dos. Un *array* lo destinará a los comodines (en la propiedad `$comodines` y el segundo, será almacenado en la propiedad `$values`
- A los comodines, les agregará las llaves de apertura y cierre, para generar el patrón que los identifique en la plantilla como tales.

```
# Modificar la propiedad $file indicando el path del template HTML a ser
renderizado
$view->file = "../site_media/html/template.html";

# Llamar al método render_template
$view->render_template();

# Imprimir el render resultante
print $view->html;
```

RenderTemplate, almacena el resultado del *render* en la propiedad pública \$html.

Pero una cuestión muy importante es **¿qué sucede si necesitamos hacer un render iterativamente?** Seguramente, podríamos utilizar esta misma clase, almacenando “mini-plantillas” con el código HTML de las iteraciones, y luego, hacer un bucle en la llamada a set_data y render_template. Pero esto, sería poco escalable, redundante y hasta tedioso.

Supongamos que CONTENIDO, a veces pueda ser un texto, otras un layer con imágenes o una tabla (como vimos al principio) que deba completarse dinámicamente, dependiendo de la cantidad de registros devueltos por una consulta a la DB.

Colocar la tabla, texto o layer en el template.html, sería contraproducente. Pues ese contenido es dinámico. Pero a la vez, ese mismo contenido que podrá variar, también variará de forma dinámica. Por ejemplo:

```
<table>
  <tr>
    <th>Material</th>
    <th>Color</th>
    <th>Ojales</th>
```

```

        </tr>
        <!-- aquí comienza un loop con reemplazos dinámicos -->
        <tr>
            <td>{Material}</td>
            <td>{Color}</td>
            <td>{Ojales}</td>
        </tr>
        <!-- aquí finaliza el loop -->
    </table>

```

Lo más conveniente entonces, será que cada uno de esos contenidos (tablas, layers, textos, etc), se almacenen en una nueva plantilla HTML, identificando cada fragmento a ser reemplazado iterativamente, como vimos al comienzo:

```

<!-- archivo: /myapp/site_media/html/lista_botones.html -->

<table>
    <tr>
        <th>Material</th>
        <th>Color</th>
        <th>Ojales</th>
    </tr>
    <!-- ini-loop: BOTONES -->
    <tr>
        <td>{Material}</td>
        <td>{Color}</td>
        <td>{Ojales}</td>
    </tr>
    <!-- end-loop: BOTONES -->
</table>

```

Hasta aquí, logramos resolver el tema de las plantillas HTML (GUI de la vista). Para obtener el reemplazo del comodín CONTENIDO, solo necesitaríamos hacer un `file_get_contents` de esta nueva plantilla:

```

$data = array("TITULO" => "Administración de Camisas",
              "MODULO" => "Camisas",
              "SUBTITULO" => "Listado de Botones",
              "CONTENIDO" =>
file_get_contents('../site_media/html/lista_botones.html');

```

Sin embargo **¿Cómo le decimos a RenderTemplate, que además de reemplazar todos los comodines anteriores,**

deberá reemplazar los del loop “BOTONES” de nuestra tabla?

La solución, será crear una nueva clase reutilizable (a nivel del *core*), para renderizar datos dinámicamente, que a la vez, herede de *RenderTemplate*. Veamos como sería:

```
# Archivo: /myapp/core/render_data.php

class RenderData extends RenderTemplate {

    protected $pattern_tags = array();
    protected $pattern = "";

    # Setear pattern tags
    protected function set_tag($tag) {
        $this->pattern_tags = array("<!-- ini-loop: {$tag} -->",
                                   "<!-- end-loop: {$tag} -->");
    }

    # Obtener posición de los pattern tags
    private function get_position($tag) {
        return strpos($this->get_html(), $this->pattern_tags[$tag]);
    }

    # Obtener longitud total del pattern
    private function get_longitud() {
        $longitud = $this->get_position(1) - $this->get_position(0);
        return $longitud + strlen($this->pattern_tags[1]);
    }

    # Setear el contenido del pattern
    protected function set_pattern_content() {
        $this->pattern = substr($this->get_html(),
                               $this->get_position(0),
                               $this->get_longitud());
    }

    # Eliminar el patrón HTML y sustituirlo por el render
    private function delete_pattern() {
        $str_final = str_replace($this->pattern_tags, "", $this->html);
        return str_replace($this->pattern, $str_final, $this->get_html());
    }

    # Renderizar datos
    public function render_data($tag, $data) {
        $this->set_tag($tag);
        $this->set_pattern_content();

        foreach($data as $array) {
            $this->set_data($array);
            $this->render_template($this->pattern);
        }
    }
}
```

```
        $this->html = $this->delete_pattern();  
    }  
}
```

Lo primero que hará esta nueva clase, es setear el patrón que identifica los tags de inicio y finalización del fragmento HTML a renderizar dinámicamente:

```
protected function set_tag($tag) {  
    $this->pattern_tags = array("<!-- ini-loop: {$tag} -->",  
                               "<!-- end-loop: {$tag} -->");  
}
```

Este método, necesitará que se le indique el nombre del identificador (tag) que en nuestro ejemplo, será “BOTONES”.

El siguiente paso que realiza, es obtener el contenido HTML de todo el fragmento. De eso se encarga el método `set_pattern_content`. Veamos como actúa este método:

```
protected function set_pattern_content() {  
    $this->pattern = substr($this->get_html(), # método de la clase principal  
                           $this->get_position(0),  
                           $this->get_longitud());  
}
```

Este método, se vale de la función `substr` de PHP, para obtener un fragmento de código de una cadena de texto. La cadena de texto que almacena el código fuente HTML, será retornada por el método `get_html` de la clase principal:

```
protected function get_html($str=NULL) {  
    return isset($str) ? $str : file_get_contents($this->file);  
}
```

Si no se le pasa una string, retornará el contenido HTML del

template seteado en la propiedad \$file.

La posición de inicio para extraer una “sub-cadena”, la obtiene a través del método `get_position()`:

```
private function get_position($tag) {  
    return strpos($this->get_html(), $this->pattern_tags[$tag]);  
}
```

Este método, recibe el índice del tag correspondiente (recordemos que se setean solo 2) y simplemente se vale de `strpos` para obtener la posición en la cual dicho tag se encuentra.

Y finalmente, para conocer la longitud total de la cadena, llama al método `get_longitud()`:

```
private function get_longitud() {  
    $longitud = $this->get_position(1) - $this->get_position(0);  
    return $longitud + strlen($this->pattern_tags[1]);  
}
```

Este método, se basa simplemente en una lógica matemática. La longitud total del patrón, será igual a la diferencia entre la posición de inicio del tag de apertura del bucle, y de la del tag de cierre del bucle, más la longitud de ese tag. Veamos un pequeño ejemplo, que nos permita entender esto, haciendo una comprobación manual:

```
Soy <b>una frase</b> corta
```

Necesitamos obtener únicamente, el texto envuelto en negritas (pero también necesitamos que las etiquetas “b” nos sean retornadas.

Contemos los caracteres:

S	o	y		<	b	>	u	n	a		f	r	a	s	e	<	/	b	>		c	o	r	t	a
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

¿Cómo obtendremos `una frase`? Haremos lo siguiente:

1. Le diremos a `strpos` que nos retorne la posición de nuestro primer patrón ``. `strpos` nos retornará 4.
2. Luego le pediremos la posición de nuestro segundo patrón `` y nos devolverá 16.
3. Para conocer la longitud entre 4 y 16, necesitamos una simple resta: $16 - 4 = 12$.
4. Ahora, cuenta (en la tabla) desde el 4, 12 casilleros. ¡Llegamos al casillero 15! Aún nos resta obtener nuestro patrón de cierre ``. Para ello, simplemente le pedimos a `strlen` que nos devuelva la longitud de nuestro patrón de cierre. Nos dirá 4. Entonces, solo sumamos esos 4 a 12: $12 + 4 = 16$.
5. Si contamos 16 casilleros desde el 4, obtendremos nuestro patrón completo (desde el 4 al 19 inclusive).

No te dejes confundir por la igualdad de los números:

Da la casualidad que el inicio del primer patrón se produce en la posición 4 y que la longitud del segundo patrón también es 4. Esto es coincidencia. El primer patrón podría iniciar en la posición 7 y la

*longitud del segundo patrón podría ser 19! Sí o sí,
el cálculo debe ser el que hemos hecho.*

Tenemos nuestra lógica a nivel del *core*, totalmente terminada. Contamos con dos clases de las cuales podremos heredar indistintamente. Pero si heredamos de *RenderData*, será como “matar dos pájaros de un tiro”.

Desde el archivo de prueba anterior, veremos como implementar entonces, ambas lógicas, a través de *RenderData*, pero esta vez, utilizaremos también nuestro modelo:

```
require_once('../core/settings.php');
require_once('../core/db_object.php');
require_once("../core/render_template.php");
require_once("../core/render_data.php");
require_once('models/boton.php');

$boton = new Boton();
$boton->boton_id = 10;
$data = $boton->get();

$view = new RenderData();
$view->file = "../site_media/html/lista_botones.html";
$view->render_data('BOTONES', $data);
$contenido = $view->html;

$data = array("TITULO" => "Administracion de Camisas",
              "MODULO" => "Camisas",
              "SUBTITULO" => "Listado de Botones",
              "CONTENIDO" => $contenido);

$view = new RenderTemplate();
$view->set_data($data);
$view->file = "../site_media/html/template.html";
$view->render_template();
print $view->html;
```

Lógica a nivel de módulo

Llegó la hora de desarrollar la lógica de las vistas para nuestros modelos. Recordemos tres cosas:

1. La lógica de la vista para cada modelo, será la encargada de mostrar (hacer un echo|print) el render del HTML al usuario;
2. Los datos que debe renderizar (datos dinámicos), se los deberá entregar el controlador (la lógica de la vista no puede conectarse con el modelo. Solo el controlador puede hacerlo).
3. Contamos con los métodos necesarios para renderizar una plantilla HTML y datos iterativamente. Entonces, “por descarte”, **nuestras vistas solo deberán encargarse de:**
 1. Definir los comodines a ser reemplazados
 2. Solo cuando sea necesario, asociar los comodines con los datos recibidos desde el controlador
 3. Modificar las propiedades de la lógica principal, definiendo los archivos a ser renderizados y haciendo la llamada a los métodos de esa lógica, que apliquen a cada caso
 4. Finalmente, deberá mostrarle esos renders al usuario

Características mínimas que debe tener la lógica

Debe ser una clase que herede de la lógica principal:

```
class NombreDeLaVista extends RenderData { }
```

Cada vista, debe estar asociada al modelo:

```
class BotonView extends RenderData { }
```

Mínimamente, cada vista deberá tener un método público (que pueda ser llamado desde el controlador), para cada uno de los métodos públicos del modelo al que pertenezca:

```
class BotonView extends RenderData {  
    public function save_view() { }  
    public function get_view() { }  
}
```

Deberá estar preparada para recibir datos desde el controlador. Ya sea, contener una propiedad pública que el controlador pueda modificar:

```
class BotonView extends RenderData {  
    public $data = NULL;  
    public function save_view() { }  
    public function get_view() { }  
}
```

O sino, definir en cada método, el/los parámetro/s necesario/s para recibir estos datos:

```
class BotonView extends RenderData {  
    public function save_view($mensaje='') { }  
    public function get_view($registros=array()) { }  
}
```

Los datos que la vista reciba desde el controlador,

generalmente serán de alguno de los siguientes tipos:

- Mensajes retornados por el modelo al controlador
- Datos retornados al controlador por el modelo

Cuando no haya sido necesario que el controlador contactase con el modelo (por ejemplo, cuando la petición del usuario sea ver un formulario para agregar un nuevo registro), luego de hacer el switch, el controlador contactará directamente con la vista. Para ello, la vista debe estar preparada, para recibir un llamado de petición. En estos casos, el tipo de datos recibido, será una petición que generalmente, estará representada por la llamada a un método:

```
class BotonView extends RenderData {  
    public function save_view($mensaje='') { }  
    public function get_view($registros=array()) { }  
    public function mostrar_formulario_alta_boton() { }  
}
```

En otros casos, la vista podrá tener un método público, que simplemente switchee la petición y ella misma, haga la llamada a un método de sí misma:

```
class BotonView extends RenderData {  
    public function save_view($mensaje='') { }  
    public function get_view($registros=array()) { }  
    public function recibir_peticion($peticion) {  
        switch ($peticion) {  
            case 'form-nuevo-boton':  
                $this->mostrar_formulario_alta_boton();  
                break;  
        }  
    }  
}
```



```
private function mostrar_formulario_alta_boton() { }  
}
```

Creando la lógica de la vista para un modelo

A continuación, basándonos en las características que definimos en el punto anterior y, en nuestro archivo de pruebas, crearemos la lógica de la vista encargada de mostrar el listado de botones. Esta clase, tendrá por el momento, solo tres métodos:

- Un método constructor, encargado de setear el diccionario de datos inicial (array asociativo donde las claves serán los comodines)
- Otro método público, para ser llamado por el controlador (será quien se encargue de hacer el render interactivo)
- Y un tercer método privado para renderizar y mostrar el *template* general

Nuestro código, se verá así:

```
# Archivo: /myapp/module/views/boton.php  
  
class BotonView extends RenderData {  
    public function __construct() {  
        $this->dict = array("TITULO" => "Administracion de Camisas",  
                           "MODULO" => "Camisas",  
                           "SUBTITULO" => "",  
                           "MENU" => "",  
                           "CONTENIDO" => "");  
    }  
  
    public function get_boton($registros=array()) {  
        $this->file = "../site_media/html/lista_botones.html";  
        $this->render_data('BOTONES', $registros);  
        $this->dict['SUBTITULO'] = "Listado de Botones";  
        $this->dict['CONTENIDO'] = $this->html;  
        $this->html = NULL;  
        $this->mostrar();  
    }  
}
```

```
private function mostrar() {
    $this->set_data($this->dict);
    $this->file = "../site_media/html/template.html";
    $this->render_template();
    print $this->html;
}

}
```

Finalmente, modificaremos nuestro archivo de pruebas, para simular la implementación de lo anterior:

```
require_once('../core/settings.php');
require_once('../core/db_object.php');
require_once("../core/render_template.php");
require_once("../core/render_data.php");
require_once('models/boton.php');
require_once('views/boton.php');

$boton = new Boton();
$data = $boton->get();

$view = new BotonView();
$view->get_boton($data);
```

El resultado final, será el *template* y la tabla del listado de botones, renderizada:

```
<!doctype html>
<head>
    <meta charset="utf-8">
    <title>Administracion de Camisas</title>
</head>

<body>
    <header>
        <h1>MyApp: Camisas</h1>
        <nav>

        </nav>
    </header>
    <section>
        <header>
            <h2>Listado de Botones</h2>
        </header>
        <table>
```

```
<tr>
  <th>Material</th>
  <th>Color</th>
  <th>Ojales</th>
</tr>

<tr>
  <td>nylon</td>
  <td>verde</td>
  <td>4</td>
</tr>

<tr>
  <td>madera</td>
  <td>caoba</td>
  <td>0</td>
</tr>

<tr>
  <td>madera</td>
  <td>verde</td>
  <td>0</td>
</tr>

<tr>
  <td>metal</td>
  <td>oro</td>
  <td>0</td>
</tr>

</table>

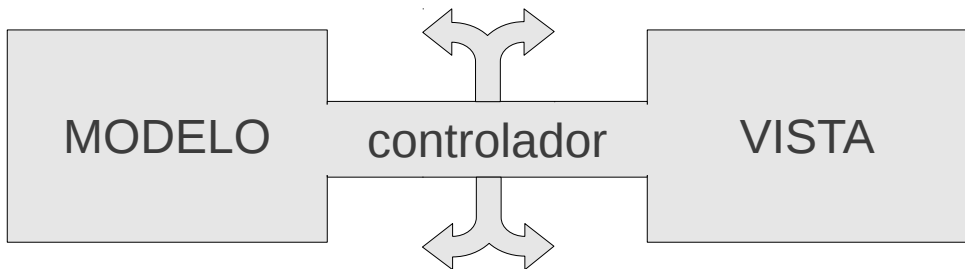
</section>
</body>

</html>
```

El controlador: el alma de MVC

En MVC, el controlador es parte esencial de la arquitectura. El controlador es una especie de “intermediario” entre el modelo y la vista. Es quien los conecta.

Podemos imaginar al controlador, como un túnel conector entre el modelo y la vista:

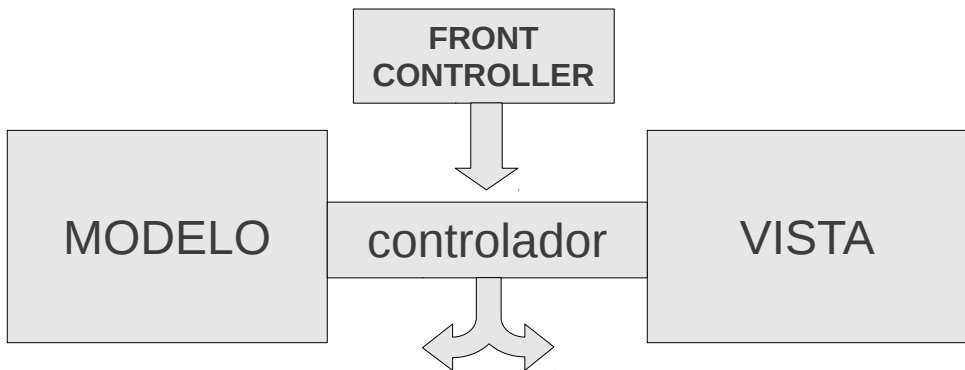


Cada módulo, contendrá entonces, un controlador para conectar cada uno de los modelos del módulo, con la vista correspondiente.

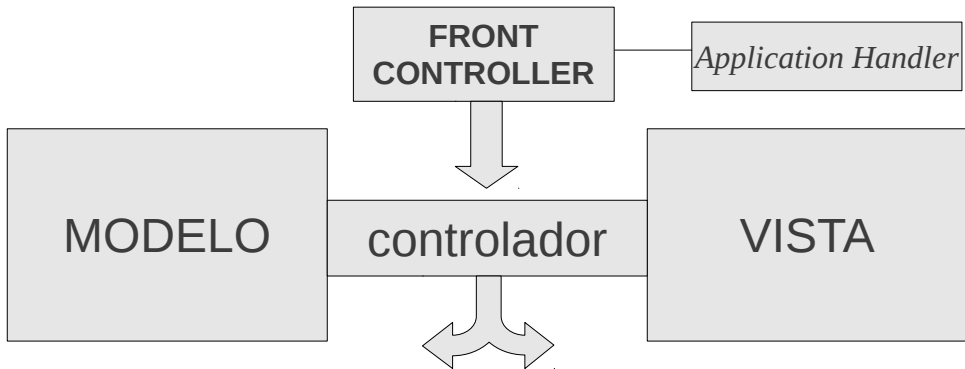
```
myapp/  
├─ core  
├─ module  
│   └─ models  
│       └─ boton.php  
│       └─ camisa.php
```

```
|   └─ views
|   │   └─ boton.php
|   │   └─ camisa.php
|   └─ controllers
|       └─ boton.php
|       └─ camisa.php
└─ site_media
```

A la vez, el usuario llegará a cada controlador, por media de un controlador general, a nivel del núcleo de la aplicación:



Y este controlador general, manejará las peticiones del usuario, a través de un *handler* (*Application Handler*):



Front Controller: controlando la aplicación a nivel del core

El **Front Controller** será el encargado de manejar las peticiones del usuario a nivel de la aplicación por medio de un **Application Handler** y hacer la llamada al controlador correspondiente, el que dependerá, de cuál haya sido la petición del usuario.

Toda la aplicación, entonces, se manejará desde el *Front Controller*, significando este hecho, que cada petición del usuario, será recibida, en primer término, por este controlador.

Para ello, el **Front Controller**, necesitará conocer:

- A qué módulo de la aplicación desea acceder el usuario
- A qué modelo hace referencia
- Qué recurso se está solicitando
- y, si se le están enviando parámetros (argumentos) a

dicho recurso

Todos estos datos, *Front Controller* los recibirá invocando al ***Application Handler*** y éste, a la vez, los obtendrá analizando la URI ingresada por el usuario.

*El **Application Handler** obtiene el módulo, el modelo, el recurso y sus argumentos, analizando la URI a través de la cual, ha ingresado el usuario.*

Para que el *Application Handler*, pueda realizar un análisis simple de la URI, ésta, en consecuencia, también debería simplificarse. Para ello, utilizaremos las llamadas ***Friendly URL*** (o URL amigables).

Configurando URLs amigables para nuestra aplicación

Hasta ahora, accedíamos a nuestra aplicación, realizando las peticiones mediante HTTP GET. Con las URL amigables, reemplazamos dicho método, tomando el valor de cada argumento -anteriormente enviado por HTTP GET- y convirtiéndolo en parte de la URL. Veámoslo con un ejemplo.

Por HTTP GET, pasaríamos los argumentos así:

```
archivo.php?  
modulo=camisas&modelo=boton&recurso=get&boton_id=32
```

Pero ahora, convertiremos el valor de cada argumento, en parte de la URI:

```
/camisas/boton/get/32
```

Es decir, que todas nuestras URIs, ahora estarán conformadas siguiendo el patrón:

```
/
modulo/modelo/recurso/argumento1/argumento2/argument
o9/
```

Para poder trabajar de esta forma, vamos a necesitar:

1. Configurar Apache, habilitando el módulo `rewrite`
2. Modificar nuestro *virtual host*, permitiendo sustituir las URL
3. Crear un archivo `.htaccess`

Configuración de Apache

La configuración de Apache es sumamente sencilla. Lo único que tendremos que hacer es habilitar el módulo `rewrite`:

```
sudo a2enmod rewrite
```

Y luego, reiniciar Apache para que el cambio surta efecto:


```
sudo service apache2 restart
```

Modificar el VirtualHost

Habr  que editar el archivo VirtualHost de nuestro Web Site. Por defecto, en distribuciones basadas en Debian, este archivo se encuentra en

/etc/apache2/sites-available/default

En otras distribuciones, este archivo puede localizarse dentro de /etc/httpd/conf.d/

Para modificarlo, dentro de <VirtualHost> tendremos que localizar (o crear si no existe), las etiquetas <Directory /> y </Directory> y all  dentro, colocar el valor de la variable AllowOverride en All, para permitir que las URL sean sustituidas:

```
<VirtualHost>
    # directivas....
    <Directory />
        AllowOverride All
    </Directory>
</VirtualHost>
```

A continuaci n, recargaremos Apache:

```
sudo service apache2 reload
```

Creando el archivo .htaccess

El archivo .htaccess lo crearemos en el directorio principal de la aplicación. En él, nos encargaremos de habilitar la re-escritura con el módulo rewrite:

```
RewriteEngine On
```

E indicarle que todo tipo de peticiones, deberán ser devueltas a un archivo llamado init_core.php que crearemos a continuación:

```
RewriteRule ^ init_core.php
```

Creando un Application Handler

Nuestro Application Handler, será una clase que tendrá a su cargo:

- Analizar la URI
- Setear un array con la petición del usuario (módulo, modelo y recurso)
- Setear un array con todos los argumentos que le hayan sido pasados al recurso

Para ello, lo que tendremos que hacer es, declarar métodos

que se encarguen de:

1. Obtener la URI
2. Dividirla en fragmentos
3. Asignar cada fragmento a la petición módulo/modelo/recurso correspondiente
4. Almacenar los argumentos

La clase AppHandler

```
# Archivo: /app/core/app_handler.php
class AppHandler {

    private $petition_names;
    private $uri;
    private $uri_data;
    public $peticiones = array();
    public $args = array();

    function __construct() {
        $this->petition_names = array("modulo", "modelo", "recurso");
        $this->uri = $this->get_uri();
        $this->uri_data = $this->set_peticiones();
        $this->peticiones = $this->get_peticiones();
        $this->args = $this->get_parametros();
    }

    private function get_uri() {
        return str_replace(APP_PATH, "", $_SERVER['REQUEST_URI']);
    }

    private function set_peticiones() {
        return explode("/", $this->uri);
    }

    private function get_long() {
        $partes = count($this->uri_data);
        return ($partes > 3) ? 3 : $partes;
    }

    private function get_peticiones() {
        $long = $this->get_long();
        $sarr = array_combine(array_slice($this->petition_names, 0, $long),
                             array_slice($this->uri_data, 0, $long));

        foreach($sarr as $clave=>$valor) {
            if(empty($valor)) unset($sarr[$clave]);
        }

        return $sarr;
    }
}
```

```
}

private function get_parametros() {
    return array_slice($this->uri_data,
                      $this->get_long(),
                      count($this->uri_data));
}
}
```

Analicemos la clase AppHandler:

El método `get_uri()` retorna el valor de la clave `REQUEST_URI` del array superglobal `$_SERVER`. Lo que hace es quitar el contenido de la constante `APP_DIR` (declarada en el settings) de la URI. Esto es necesario, si nuestra app, no está corriendo en el directorio raíz de nuestro servidor. Entonces, declaramos el path hasta el directorio de nuestra aplicación, en la constante `APP_DIR` y obtendremos la parte de la URI que nos interesa. El retorno de este método, será almacenado desde el constructor en la propiedad `$uri`.

```
private function get_uri() {
    return str_replace(APP_PATH, "", $_SERVER['REQUEST_URI']);
}
```

El método `set_peticiones()`, se encarga de dividir la URI anterior, tomando como referencia, la barra diagonal. Los valores retornados por este método, serán almacenados desde el constructor, en la propiedad `$uri_data`.

```
private function set_peticiones() {
    return explode("/", $this->uri);
}
```

El método `get_peticiones()` realiza un análisis más complejo de los resultados almacenados en la propiedad `$uri_data`. Primero, llama `get_long()` para calcular la longitud de las peticiones. Esto es:

Si hay 3 valores, estaremos en condiciones de asignar un valor a módulo, otro a modelo y el tercero, a recurso. Si hay más, será señal de que estamos recibiendo argumentos. Pero si hay menos, la longitud será la cantidad de elementos reales de `uri_data`:

```
private function get_long() {  
    $partes = count($this->uri_data);  
    return ($partes > 3) ? 3 : $partes;  
}
```

`get_peticiones`, necesita de `get_long`, para poder extraer de `uri_data`, la cantidad correspondiente a los tres primeros elementos (o si hay menos, los que tenga). Mediante `array_slice`, extrae el módulo, modelo y recurso de `uri_data` (son los tres primeros elementos) y hace lo propio con el array que define los nombres (claves) para estos tres pedidos.

```
$arr = array_combine(array_slice($this->petition_names, 0, $long),  
                    array_slice($this->uri_data, 0, $long));  
}
```

Finalmente, recorre el array resultante en busca de elementos vacíos. Elementos vacíos pueden suceder cuando la URI real, finalizaba en una `"/`. Si encuentra elementos vacíos, elimina la clave del array resultante y finalmente, retorna ese array. El resultado, será almacenado desde el constructor, en la propiedad pública `$peticiones`.

```
foreach($arr as $clave=>$valor) {  
    if(empty($valor)) unset($arr[$clave]);  
}  
  
return $arr;
```

Finalmente, el método `get_parametros` se encargará de extraer de `uri_data`, aquellos elementos “sobrantes” (es decir, cualquier elemento que se encuentre del tercero). Esos

elementos, serán los argumentos que almacene la propiedad pública `args` desde el constructor:

```
private function get_parametros() {  
    return array_slice($this->uri_data,  
                      $this->get_long(),  
                      count($this->uri_data));  
}
```

Conclusión

Podemos deducir entonces, que la clase `AppHandler` será la encargada de analizar toda la URL, colocando en disposición pública un array asociativo con las peticiones módulo-modelo-recurso y otro, con todos los argumentos.

Esta clase, entonces, será utilizada por el `FrontController`, quien solo la instanciará y, el método constructor de ésta, le proveerá a `FrontController` de un objeto `AppHandler`, con dos propiedades públicas `AppHandler->peticiones` y `AppHandler->args`, para que `FrontController` se encargue del resto, es decir, instancie al controlador correspondiente, enviándole la petición (recurso) con todos los argumentos que este recurso necesite.

La clase `FrontController`

Ahora sí, ya estamos en condiciones de crear nuestro **controlador general de la aplicación**.

La clase `FrontController` se encargará entonces, de:

- Llamar a `AppHandler` para obtener las peticiones solicitadas por el usuario y los argumentos enviados
- Buscar el controlador necesario e importarlo
- Llamar al controlador y enviarle la petición (recurso) y los argumentos

Veamos como será la clase FrontController:

```
# Archivo: /app/core/app_handler.php

class FrontController {

    private static $app_data;
    public static $error = NULL;

    public static function run() {
        self::$app_data = new AppHandler();
        if(!self::call_controller()) {
            self::$error = "El recurso solicitado no se encuentra
disponible";
        }
    }

    private static function call_controller() {
        extract(self::$app_data->peticiones);
        if(isset($modulo) && isset($modelo) && isset($recurso)) {
            require_once("$modulo/controllers/$modelo.php");
            $controller = Helper::set_controller_name($modelo);
            $method = Helper::set_method_name($recurso);
            $c = new $controller($method, self::$app_data->args);
            return True;
        }
    }
}
```

Cuando FrontController deba ser “activado”, será llamado de forma estática, su método run. Por ejemplo:

```
FronController::run();
```

Será el encargado de manejar todas las peticiones a nivel de la aplicación.

Para obtener las peticiones y argumentos, FrontController simplemente crea un objeto AppHandler que será almacenado en una propiedad estática (privada) \$app_data (línea 9 de app_handler.php)

```
self::$app_data = new AppHandler();
```

Para buscar e importar el controlador necesario, primero verificará que las peticiones módulo, modelo y recurso se encuentren seteadas en el objeto AppHandler y de ser así, incluirá el archivo del controlador:

```
# convierte en variables temporales todas las peticiones
extract(self::$app_data->peticiones);
# si modulo, modelo y recurso han sido seteadas...
if(isset($modulo) && isset($modelo) && isset($recurso)) {
    # Importa el archivo del controlador
    require_once("$modulo/controllers/$modelo.php");
}
```

Si nos ponemos detallistas, podremos observar que FrontController, de manera indirecta nos está forzando a mantener una estructura de directorios y una arquitectura, conforme lo que hemos visto hasta ahora sobre MVC.

Finalmente, para llamar al controlador y pasarle el recurso (petición) y sus argumentos, hace lo siguiente:

Se vale de un Helper para obtener el nombre adecuado del controlador. El mismo debe ser el nombre de la petición, sin guiones medios, con formato CamelCase seguido de la palabra Controller. De esto se encarga el método Helper::set_controller_name().

```
$controller = Helper::set_controller_name($modelo);
```

Luego, hace lo propio con el recurso. Para obtener el nombre del método-controlador del recurso, se vale de otro helper, el cual considerará las reglas para establecer dicho nombre: será el nombre del recurso, reemplazando guiones medios por bajos, todo en minúsculas y seguido del sufijo _controller.


```
$method = Helper::set_method_name($recurso);
```

Finalmente, instancia al controlador, pasándole dos parámetros: el recurso y los argumentos:

```
$c = new $controller($method, self::$app_data->args);
```

Conclusión

Si observamos el código anterior, concluiremos en que FrontController, nos ha guiado el camino para crear nuestros controladores. Los mismos deberá, contar sí o sí, con:

- Un método constructor preparado para recibir dos parámetros: el recurso (nombre del método que deberá invocar) y los argumentos (parámetros que deberá pasarle a ese método)
- Un método para cada recurso, donde el nombre del mismo, deberá ser `nombre_del_recurso_controller`
- cada uno de esos métodos, deberá estar preparado para recibir una cantidad incierta de parámetros

Y sí. Si estás pensando en `call_user_func_array()` y `func_get_args()`, estás en el camino correcto :)

*Vale aclarar además, que **todos los archivos que***

*integran el core de la aplicación que fuimos creando a lo largo de este taller, **son reutilizables en cualquier tipo de aplicación modular MVC.** Esto significa, que para crear aplicaciones MVC modulares, podrás reutilizar este core y solo concentrarte en crear los módulos, con sus correspondientes modelos, vistas y controladores.*

*Es decir, que **todos estos archivos del core, no son meros ejemplos que deban ser interpretados y trasladados a los objetivos de tu aplicación,** sino que por el contrario, **son el núcleo de tu aplicación, sea cual fuere su objetivo.***

A continuación, veremos como crear el controlador para nuestros modelos.

Creando controladores para nuestros modelos

Crear el controlador para nuestros modelos, será una tarea sumamente simple.

Por un lado, ya conocemos **qué es lo que debe hacer un método-controlador:**

- Instanciar al modelo
- Modificar sus propiedades (cuando sea necesario)
- Llamar a uno de sus métodos (el cual, nos retornará algún dato)
- Enviar los datos retornados por el modelo, a la vista

Por otro lado, nuestro FrontController, nos ha definido las características que tendrán que tener nuestros controladores:

- El nombre de la clase será el nombre del modelo, seguido del sufijo Controller:

```
class ModeloController { }
```

- Deberá tener un método constructor, preparado para recibir dos parámetros. El primer parámetro, será el nombre del recurso (método-controlador) y el segundo, un array de argumentos para enviar a ese recurso:

```
class ModeloController {  
    public function __construct($recurso='',  
    $argumentos=array()) { }  
}
```

- La clase ModeloController, deberá contar entonces, con un método por cada recurso, donde el nombre de éste se conforme por el nombre del recurso en minúsculas, palabras separadas por guiones medios, sucedido del sufijo _controller:

```
class ModeloController {  
  
    public function __construct($recurso='',  
    $argumentos=array()) { }  
  
    protected function  
    nombre_del_recurso_controller() { }  
}
```

- El método constructor, deberá ser quien invoque al método-controlador pasándole los argumentos

necesarios. Para ello, utilizará la función nativa `call_user_func_array`:

```
class ModeloController {

    public function __construct($recurso='',
    $argumentos=array()) {

        call_user_func_array(array($this, $recurso),
        $argumentos);

    }

    protected function
    nombre_del_recurso_controller() { }

}
```

- Y a la vez, los métodos-controladores que requieran parámetros, deberán estar preparados, para recibir una cantidad de argumentos incierta:

```
class ModeloController {

    public function __construct($recurso='',
    $argumentos=array()) {

        call_user_func_array(array($this, $recurso),
        $argumentos);

    }

    protected function
    nombre_del_recurso_controller() {

        $argumentos = func_get_args();

    }

}
```

- Y finalmente, nuestros controladores, deberán almacenarse en un directorio controllers dentro del módulo al cual pertenezcan. El nombre, deberá ser el mismo que el delo modelo.

Características extras que debemos tener en cuenta para crear nuestros controladores son:

- Sabemos que todos los métodos-controlares (o casi todos), necesitarán instanciar al modelo. Pues entonces, lo haremos en el método constructor, asignando el objeto a una propiedad:

```
class ModeloController {  
  
    public function __construct($recurso='',  
$argumentos=array()) {  
        call_user_func_array(array($this, $recurso),  
$argumentos);  
        $this->modelo = new Modelo();  
    }  
  
    protected function  
nombre_del_recurso_controller() {  
        $argumentos = func_get_args();  
    }  
}
```

- También sabemos que todos los métodos-controlares, deberán instanciar a la vista. Así que haremos lo propio:

```
class ModeloController {
```

```
public function __construct($recurso='',
$argumentos=array()) {
    call_user_func_array(array($this, $recurso),
$argumentos);
    $this->modelo = new Modelo();
    $this->view = new ModeloView();
}

protected function
nombre_del_recurso_controller() {
    $argumentos = func_get_args();
}
}
```

- Ya que el controlador será el único autorizado a conectar el modelo con la vista, será a la vez, el único encargado de importar los archivos correspondientes a su modelo y su vista:

```
require_once('modulo/models/modelo.php');
require_once('modulo/views/modelo.php');

class ModeloController {

    public function __construct($recurso='',
$argumentos=array()) {
        call_user_func_array(array($this, $recurso),
$argumentos);
        $this->modelo = new Modelo();
        $this->view = new ModeloView();
    }
}
```

```
        protected function
nombre_del_recurso_controller() {
            $argumentos = func_get_args();
        }
    }
}
```

Lo anterior, entonces, será nuestro **template para crear controladores**. Entonces, con este *template*, crearemos -ahora sí-, nuestro controlador BotonController:

```
# Archivo: /app/module/controllers/boton.php

require_once('module/models/boton.php');
require_once('module/views/boton.php');

class BotonController {

    public function __construct($recurso='', $args=array()) {
        $this->boton = new Boton();
        $this->view = new BotonView();
        call_user_func_array(array($this, $recurso), $args);
    }

    private function get_controller() {
        $args = func_get_args();
        $this->boton->boton_id = (count($args) > 0) ? (int)$args[0] : 0;
        $data = $this->boton->get();
        $this->view->get_boton($data);
    }

}
```

Notas adicionales sobre el ejemplo

A fin de hacer más entretenido|productivo|real nuestra aplicación, haremos una pequeña modificación en el query del método get del modelo Boton, de forma tal que pueda traer un

solo resultado que coincida con la ID seteada o todos, en caso de que la ID sea igual a cero. El modelo Boton, entonces, se verá así:

```
class Boton {

    public $material = NULL;
    public $color = NULL;
    public $ojales = NULL;
    public $boton_id = 0;

    public function save() {
        $sql = "INSERT INTO boton (material, color, ojales) VALUES
        (?, ?, ?)";

        $data = array("ssi",
            "{$this->material}",
            "{$this->color}",
            "{$this->ojales}");

        return DBObject::ejecutar($sql, $data);
    }

    public function get() {
        $sql = "SELECT material, color, ojales FROM boton WHERE boton_id ";
        $sql .= ($this->boton_id > 0) ? "= ?" : "> ?";

        $data = array("i", "{$this->boton_id}");

        $fields = array("Material"=>"",
            "Color"=>"",
            "Ojales"=>"");

        DBObject::ejecutar($sql, $data, $fields);
        return DBObject::$results;
    }
}
```

Agregamos una línea que completa el *query*, diciendo que:

Si boton_id es mayor que cero, la cláusula WHERE se completará con = ?, sino, se completará con > ?

Completando la aplicación

Ahora sí, es hora de ir “redondeando” nuestra aplicación. En principio, haremos lo siguiente:

1. Mover el archivo settings.php al directorio raíz de la aplicación
2. Crear en el mismo directorio, un archivo llamado init_core.php que -como su nombre lo indica- se encargará de inicializar el núcleo de la aplicación.

De esta forma, la estructura de nuestra aplicación, deberá verse así:

```
app/  
├─ core/  
│   ├── app_handler.php  
│   ├── db_object.php  
│   ├── front_controller.php  
│   ├── helper.php  
│   ├── render_data.php  
│   └─ render_template.php  
├─ init_core.php  
├─ module/  
│   ├── controllers/  
│   │   └─ boton.php  
│   ├── models/  
│   │   └─ boton.php  
│   └─ views/  
│       └─ boton.php  
├─ settings.php  
└─ site_media/  
    └─ html/  
        └─ lista_botones.html
```

```
└─ template.html
```

A continuación, haremos algunas modificaciones a nuestro archivo settings.php

Por un lado, agregaremos 4 constantes, encargadas de definir:

- El path donde corre nuestra aplicación. Esto es, desde el directorio raíz del servidor, hasta el directorio raíz de la aplicación inclusive.
- El directorio que almacenará los archivos estáticos. Si es otro servidor, deberá definirse antecediendo el protocolo correspondiente. Por ejemplo:
`http://xxx.xxx.xx.xx/app/`
- El directorio que almacena los archivos HTML dentro del directorio estático.
- El directorio del núcleo (en nuestro caso, core/)

```
const APP_PATH = "/mvcapp/";  
const STATIC_PATH = "site_media/";  
const STATIC_DIR = "html/";  
const CORE_DIR = "core/";
```

Por otro lado, modificaremos -en tiempo de ejecución- el `include_path` del `php.ini`, a fin de poder incluir los archivos de forma más cómoda. Asignaremos el valor de la constante superglobal `__DIR__` a esta directiva:

```
ini_set("include_path", __DIR__);
```

Finalmente, agregaremos un pequeño script, para que nos muestre todos los errores, avisos y advertencias de PHP, siempre y cuando, no estemos en producción (si haz realizado el curso PHP para Principiantes o haz leído el libro del curso, el

script que utilizaremos es el mismo que usamos en el ejemplo del capítulo “Tratamiento y Control de Errores”):

```
const PRODUCCION = False;

if(!PRODUCCION) {
    ini_set('error_reporting', E_ALL | E_NOTICE | E_STRICT);
    ini_set('display_errors', '1');
    ini_set('track_errors', 'On');
} else {
    ini_set('display_errors', '0');
}
```

Inicializador del núcleo

Finalmente, vamos a crear nuestro archivo inicializador del núcleo, `init_core.php`

Este archivo cumplirá dos funciones:

- 1) Incluir (importar) todos los archivos del núcleo en el orden preciso
- 2) Iniciar el `FronController`

```
require_once('settings.php');
require_once(CORE_DIR . 'app_handler.php');
require_once(CORE_DIR . 'front_controller.php');
require_once(CORE_DIR . 'db_object.php');
require_once(CORE_DIR . 'helper.php');
require_once(CORE_DIR . 'render_template.php');
require_once(CORE_DIR . 'render_data.php');

FrontController::run();
```

Para este ejemplo, agregaremos una tercer función que en

realidad, es opcional. Ésta, será la de verificar si FrontController ha fallado, para imprimir un mensaje que lo indique.

```
if(!is_null(FrontController::$error)) {  
    print FrontController::$error;  
}
```

¡Ya tenemos nuestra aplicación funcionando!

Para probarla, podremos ingresar en nuestro navegador la URI correspondiente:

`http://dominio/aplicacion/module/boton/get`



O podremos también, pasar una ID de botón como argumento:

`http://dominio/aplicacion/module/boton/get/32`



Los patrones de diseño Front Controller y Application Handler en MVC

Es posible crear una vista general, que sea mostrada al usuario cada vez que un recurso no sea especificado. Existen varios métodos mediante los cuales podremos lograrlo.

El más simple de todos, es crear una clase `FrontView` a nivel del core. Para ello necesitaremos:

1. Una plantilla HTML
2. Una clase `FrontView` que herede de `RenderTemplate`
3. Agregar un método de comprobación a `FrontController` que se encargue de llamar a nuestra nueva clase

Veamos como sería en la práctica.

Plantilla HTML (`site_media/html/default_template.html`):

```
<!doctype html>
<head>
  <meta charset="utf-8">
  <title>MVCAApp</title>
</head>

<body>
  <header>
    <h1>MVC App</h1>
  </header>
  <section>
    <header>
      <h2>Módulos Disponibles en la aplicación</h2>
```

```

        </header>
        <section>
            <p><b>{FRONT-CONTROLLER-MESSAGE}</b><br/>
            Por favor, seleccione la opción de su interés</p>
            <h3>Módulo de Camisas</h3>
            <nav>
                <b>Botones:</b>
                <a href="{APP_PATH}module/boton/get">Listar Botones<a>
            </nav>
        </section>
    </section>
</body>
</html>

```

La clase FrontView (core/front_view.php):

```

class FrontView extends RenderData {

    public function __construct($msg='') {
        $this->dict = array("APP_PATH" => APP_PATH,
                           "FRONT-CONTROLLER-MESSAGE" => $msg);
    }

    public function show_default_view() {
        $this->set_data($this->dict);
        $this->file = STATIC_PATH . STATIC_DIR . "default_template.html";
        $this->render_template();
        print $this->html;
    }

}

```

Método de validación en FrontController:

Método para setear el mensaje que será transmitido a la vista:

```

    private static function set_error() {
        if(isset(self::$app_data->peticiones['modulo'])) {
            self::$error = "El recurso solicitado no se encuentra
disponible";
        }
    }
}

```

Este método, seteará siempre el mismo error, excepto cuando el usuario acceda a la carpeta raíz (index) de la aplicación.

Método que ejecuta un comando “mostrar vista por defecto”:

```
private static function execute_command() {  
    $command = new FrontView(self::$error);  
    $command->show_default_view();  
}
```

Ambos métodos, implicarán modificar el método run de FrontController:

```
if(!self::call_controller()) {  
    self::set_error();  
    self::execute_command();  
}
```

Eliminar la siguiente validación del init_core.php

```
if(!is_null(FrontController::$error)) {  
    print FrontController::$error;  
}
```

Y lógicamente, agregar la importación de front_view.php en el init_core.php

```
require_once(CORE_DIR . 'front_view.php');
```


Web Services: creación de una API REST

Conceptos básicos

Web Services

Un **Web Service** (o WS) -Servicio Web- es una **tecnología que permite a dos aplicaciones** diferentes, **comunicarse entre sí, por medio de diferentes protocolos** como HTTP, FTP, SMTP, SOAP, entre otros.

API

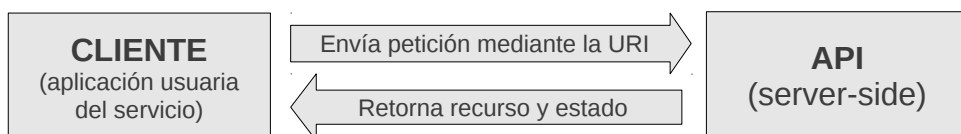
Una **API** (Application Programming Interface) -o Interfaz de programación de aplicaciones-, es el **conjunto de métodos de una capa de abstracción**, que **permite a otra aplicación, acceder a funcionalidades que le son ajenas**. La API, pertenece a la aplicación que permitirá a otras, hacer uso de una o más de sus funcionalidades.

REST

REST (Representational State Transfer) -Transferencia de estado representacional- es un **patrón arquitectónico** que **basa su estructura en recursos** (todo es un recurso), permitiendo crear servicios Web que solo utilicen el **protocolo HTTP**, de manera clara y simple, donde **las solicitudes** (al igual que en MVC), **son manejadas a través de la URI**.

Entendiendo las Arquitecturas REST

A diferencia de otras arquitecturas para servicios Web, como SOAP, por ejemplo, REST propone manejar, tanto los métodos (PUT, POST, GET y DELETE) como los estados de los mismos (resultado de la petición), en el mismo recurso, manejando las peticiones solo a través de las URIs, significando esto, que solo será necesaria una única petición, para acceder a cualquiera de los métodos.



Realmente, una arquitectura REST, no tiene ninguna complejidad. Solo se requiere de una API que maneje las peticiones del cliente y le retorne el recurso solicitado con su estado correspondiente.

Características de una API-REST

Para manejar las peticiones del cliente y retornar el recurso solicitado con su correspondiente estado, una API solo requiere:

1. Analizar la URI (de la misma forma que lo hace el `ApplicationHandler`)
2. Actuar como una capa de abstracción, entre el recurso solicitado y el cliente (de la misma forma que lo hace un `FrontController`)
3. Entregar al cliente el recurso solicitado, en formato de

texto plano (no se utilizan GUIs).

Los requerimientos 1 y 2, se diseñan con un patrón FrontController (la API en sí, ES un FrontController). Y el requerimiento 3, solo necesita un método que imprima los datos entregados -en nuestro caso, por el controlador de nuestra MVCApp-en un formato válido. Por ejemplo, XML o mejor aún, algo mucho más liviano y legible como JSON¹².

¹² <http://es.wikipedia.org/wiki/Json>

Funcionamiento de la API REST de nuestra MVC App

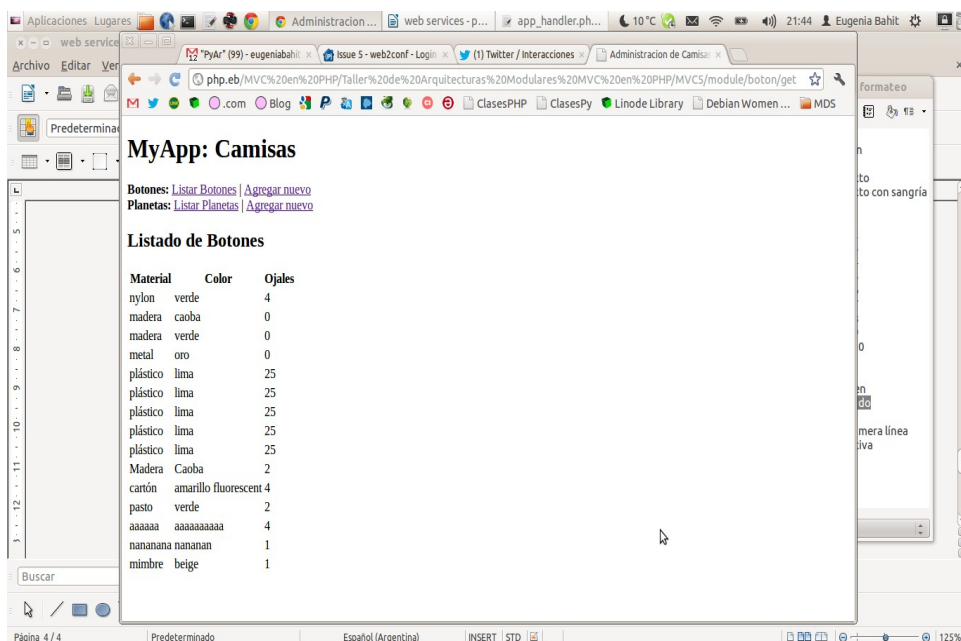
En capítulos anteriores, hemos creado una completa aplicación modular, basada en el patrón arquitectónico MVC. Veremos ahora, como crear un Web Service para nuestra aplicación, basado en una arquitectura REST.

Antes de continuar analizando el código fuente que acompaña el módulo, es importante saber con exactitud, qué es lo que hará nuestro Web Service y cuál será su finalidad.

Objetivo del Web Service

El objetivo de nuestro Web Service, será el de permitir a otras aplicaciones, obtener los mismos datos que nuestra app provee, pero en un formato que la aplicación cliente, para utilizar para sí.

Por ejemplo, cuando nosotros accedemos al listado de botones, obtenemos la lista de registros almacenados en nuestra DB, con una GUI que los formatea:



Imaginemos que somos fabricantes de botones, y que uno de nuestros clientes, desea disponer de nuestra base de datos de botones, actualizada y en todo momento.

Permitirle a nuestro cliente, una conexión externa a nuestra base de datos, sería una locura, puesto que no solo significa un inmensurable riesgo de seguridad, sino que además, con cada cambio que hiciéramos a la base de datos, deberíamos estar notificándolo al cliente. Hoy, es un cliente. Pero mañana ¿cómo notificaremos a diez mil clientes?

La solución, entonces, es crear un Web Service. Que nuestro cliente, pueda hacer una simple petición mediante el protocolo HTTP, y así, disponer en todo momento, los datos actualizados. De esta forma, nuestro WS, le entregará al cliente, un archivo JSON, que éste, podrá manipular al igual que un array multidimensional retornado por una consulta SQL de selección:

```
[
  {"Material":"nylon","Color":"verde","Ojales":4},
  {"Material":"madera","Color":"caoba","Ojales":0},
  {"Material":"madera","Color":"verde","Ojales":0},
  {"Material":"metal","Color":"oro","Ojales":0},
  {"Material":"pl\u00e9stico","Color":"lima","Ojales":25},
  {"Material":"pl\u00e9stico","Color":"lima","Ojales":25},
  {"Material":"pl\u00e9stico","Color":"lima","Ojales":25},
  {"Material":"pl\u00e9stico","Color":"lima","Ojales":25},
  {"Material":"Madera","Color":"Caoba","Ojales":2},
  {"Material":"cart\u00f3n","Color":"amarillo fluorescent","Ojales":4},
  {"Material":"pasto","Color":"verde","Ojales":2},
  {"Material":"aaaaaa","Color":"aaaaaaaaaa","Ojales":4},
  {"Material":"nananana","Color":"nananan","Ojales":1},
  {"Material":"mimbre","Color":"beige","Ojales":1}
]
```

Es así entonces, como nuestro cliente, podrá iterar con los datos de este archivo (incluso, en cualquier otro lenguaje que no sea PHP, ya que JSON, es y puede ser manipulado desde cualquier lenguaje), para presentarlos a sus propios usuarios, de la manera que desee. Por ejemplo, en un formulario:



Formato de la URI

Al igual que con MVC, la URI de nuestro WS, se conformará por:

```
modulo/modelo/recurso/parametros
```

La única diferencia con MVC, es que todo lo anterior, será precedido por la palabra “api” a fin de identificar que se trata de una petición de tipo servicio Web:

```
api/modulo/modelo/recurso/parametros
```

Vale aclarar, que los parámetros, son opcionales.

Diseño de la API

Recordemos, como bien se comentó al inicio, que la API utilizará un patrón de diseño FrontController, es decir, que deberá ser un objeto (tipo ApiController), que utilizando un ApplicationHandler para analizar la URI, se encargue de instanciar al controlador del modelo solicitado y llamar al método correspondiente al recurso solicitado.

Finalmente, esta ApiController, deberá entregar los datos retornados por el método del controlador del modelo, en formato JSON al cliente. Ahora sí, analicemos el código que acompaña al módulo.