Estándares, COnvenciones y buenas prácticas de desarrollo



Tabla de contenido

[Tabla de Contenido 1](#_Toc447684180)

[Tabla de Modificaciones y correcciones 3](#_Toc447684181)

[Introducción 4](#_Toc447684182)

[Estándares de análisis y diseño de Base de Datos 5](#_Toc447684183)

# Tabla de Modificaciones y correcciones

|  |  |  |
| --- | --- | --- |
| Persona que realiza el cambio | Fecha | Resumen del cambio |
|  |  |  |

# Introducción

#### Este documento con los estándares para el diseño de Bases de Datos para el sistema de Gestión para la gasolinera “Servicio San Fermín”, pretende normar el análisis, diseño, implementación y mantenimiento de esta importante área en un mundo donde la información es poder y posee un gran valor, tanto financiero como estratégico.

#### Los estándares aquí incluidos pretenden formalizar la manera en que se codifican o estructuran las diversas partes que conforman las bases de datos de la empresa tanto a nivel lógico como físico, en aspectos fundamentales tales como la seguridad y ambiente. De esta manera se logra que los miembros del equipo de trabajo se comuniquen mejor y logren entender claramente los objetos de la base de datos y lo que se ha pretendido procesar o almacenar en cada uno de ellos.

# Estándares de análisis y diseño de Base de Datos

#### **Objetivo**

#### La especificación de los estándares para el análisis y diseño de la Base de Datos tiene como fin principal asegurar la legibilidad del modelo de datos de tal manera que sea entendible para toda aquella persona que incluso no tenga experiencia en la informática, además de facilitar la portabilidad entre los diferentes motores de bases de datos, plataformas y aplicaciones y como un objetivo principal es que para los desarrolladores les sea más fácil el desarrollo al identificar los objetos.

#### **Alcance**

#### La implementación de los estándares de este manual es de carácter obligatorio para todas las Bases de Datos utilizadas en la gasolinera “Servicio San Fermín”.

#### **Estándar de Análisis de Bases de Datos**

#### Siempre deberá existir un análisis de la base de datos antes de la implementación de cualquier Base de Datos y deberá contar tanto en forma impresa como electrónica, al menos con los siguientes elementos:

#### **Diagrama Entidad Relación**

#### Definición con todas las tablas, llaves y las relaciones correspondientes. Este diagrama debe cumplir con al menos, las tres primeras Formas Normales, en caso de no ser posible debe documentarse las excepciones y los motivos por los que no se aplicaron.

#### **Diccionario de Datos**

#### Es un conjunto de metadatos que contiene las características lógicas y puntuales de los datos que se van a utilizar en el sistema que se programa, incluyendo nombre, descripción, alias, contenido y organización.

#### En un diccionario de datos se encuentra la lista de todos los elementos que forman parte del flujo de datos de todo el sistema. El diccionario de datos guarda los detalles y descripción de todos estos elementos.

#### Es obligación de quien realice el análisis y diseño de los datos a almacenar, especifique cada uno de los elementos que componen su respectivo diseño mediante el diccionario de datos.

#### **Estándar de Diseño de Bases de Datos**

#### Con este se busca estandarizar aspectos relevantes de los objetos como el uso de nombres, documentación, seguridad y rendimiento.

#### **Tablas**

#### Las tablas identifican una entidad del sistema con un **nombre completo**.

#### Ejemplo: usuarios

#### El nombre de la tabla debe ser **descriptivo**, en **plural** y en **minúsculas**.

#### Una tabla hija debe llevar al inicio el nombre de la tabla padre.

#### Ejemplo: usuario\_contratos

#### Si la tabla tiene más de una palabra se deberá colocar cada una de ellas en singular y deben ser separadas por un "\_".

#### No usar prefijos o sufijos innecesarios (como TabEmpleado, EmpleadoTabla, EntEmpleado etc. Para hacer referencia a la tabla Empleado)

#### Los nombres de las tablas no deben exceder de 128 caracteres.

#### No usar acentos ni “ñ”. En caso de requerirse sustituir por “ni”. Ejemplo anios

#### Cuando se requiera almacenar bitácoras de alguna tabla, guardar todos los datos en una tabla copia y agregándole la fecha de inserción y el id de usuario en sesión.

#### **Columnas**

#### Los nombres de los campos deben ser **descriptivos**, en **singular** y en **minúsculas**. Entre más simple, mejor.

#### Si el nombre de la columna contiene varias palabras, deben ser separadas por un “\_”.

#### Toda tabla deberá tener como llave primaria el campo “**id**” autoincrementable(1) BIGINT. Una columna con tipo varchar puede causar problemas de rendimiento. Se deberán definir los índices necesarios de búsqueda para agilizar las consultas.

#### Para nombrar columnas de tipo llave foránea usar “**nombre\_tabla\_id**”. Ejemplo: “empleado\_id”, “usuario\_contratos\_id”.

#### Los nombres de las columnas no deben exceder de 30 caracteres.

#### No usar acentos ni caracteres especiales como la “ñ”. En caso de requerirse sustituir por “ni”. Ejemplo “anio”

#### Cuando se requiera almacenar valores monetarios será obligatorio usar el tipo de dato **MONEY**.

#### Para almacenar valores tipo fecha:hora utilizar el tipo de dato DATETIME. Si se requiere almacenar el campo de control fecha de registro, el nombre deberá ser **fecha\_registro**.

#### Para almacenar valores tipo **estatus** utilizar el tipo de dato **tinyint**. Con los posibles valores 1: activo, 0: inactivo.

#### Para almacenar valores en los que haya la certeza que no se excederá 1 byte, utilizar el tipo de dato **tinyint**.

#### Ejemplos de nombrado de columnas: id, nombre, paterno, folio, fecha\_ingreso

#### **4.3 Restricciones (Constraints)**

#### Se recomienda usar restricciones para la integridad de datos. No dejar todo el control al código de la aplicación.

#### Llaves primarias PK\_NombreTabla

#### Llaves foráneas FK\_NombreTablaOrigen\_NombreTablaReferenciada

#### Restricción Unique UQ\_NombreTabla\_NombreUnique

#### Default DF\_NombreTabla\_NombreColumna

#### Check CK\_NombreTabla\_NombreCheck

#### **4.4 Índices**

#### Nomenclatura: IXX\_YYYY… XX : Tipo de índice (Unique, Clustered, NonClustered) YYYY : Nombre del índice

#### **4.5 Triggers**

#### TR\_NOMBRETABLA\_D (Cuando se realiza una eliminación en la tabla).

#### TR\_NOMBRETABLA\_U (Cuando se realiza una actualización en la tabla).

#### TR\_NOMBRETABLA\_I (Cuando se realiza una inserción en la tabla).

#### TR\_NOMBRETABLA\_IU (Cuando se realiza una inserción o actualización en la tabla).

#### **4.6 Funciones de Usuario (functions)**

#### Debemos nombrarla de acuerdo a su funcionalidad, como éstos no van dirigidos a una tabla en específica pueden nombrarse de acuerdo a lo que hacen.

#### Ejemplo:

#### CalcularIva

#### **4.7 Procedimientos Almacenados (Stored Procedures)**

#### Se recomienda sólo utilizarlos para consultas.

#### Nomenclatura: PA\_YYYY

#### YYY : Nombre del procedimiento almacenado

#### Los nombres de los Stored Procedures NO deben comenzar con SP, esto porque generalmente el SQL piensa que son system procedures y los busca primero en la Base de Datos master

#### SET NOCOUNT ON (elimina la notificación del nro. de registros afectados por cada sentencia SQL lo cual incrementa el performance

#### Estructura del Stored Procedure: • Identificador, nombre del stored procedure, y parámetros • Comentarios: • Descripción: “funcionalidad del stored procedure” • Fecha Creación: • Fecha Modificación: • Descripción de Parámetros de entrada y salida • Autor: • Versión: • Cambios Importantes • Declaración Variables locales • Sentencias SQL • Palabras del lenguaje SQL y funciones en MAYUSCULAS, columnas y otras variables en minúsculas. • Sentencias legibles e indentadas (cada clausula SQL en una línea nueva)

#### • Ejemplo:

#### /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*Descripcion: Genera el cierre contable mensual o anual \*Fecha Creación: 15/02/2001 \*Fecha Modificación: 19/02/2001 \*Parametros: @ANIOEJERCICIOFISCAL : Año del ejercicio \* @TIPO: Tipo del cierre Mensual / Anual \* … \*Autor: jperez \*Versión: Final (Beta|Final) \*Cambios Importantes: Inclusión de la condición de consulta (19/02/2001) \*/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#### CREATE PROCEDURE PA\_GENERACIERRE( @ ANIOEJERCICIOFISCAL int, @ NOMBRE varchar(50), …) AS <Declaración de variables>

#### --COMENTAR Sentencia SQL <Sentencias SQL>

#### 3.8 Vistas (views)

#### Nomenclatura: VW\_NombreVista

#### **Mejores prácticas**

#### Los procedimientos almacenados, funciones y las vistas transversales (de uso común) deberán ser definidos por todos los involucrados.

#### Proveer siempre de autenticación para el acceso a base de datos. No colocar el rol de administrador a cada usuario.

#### Usa campos de tipo bit para almacenar valores booleanos. Usar enteros o varchar repercute en un consumo innecesario de almacenamiento.

#### Imágenes y columnas de tipo blob no deben estar definidas en tablas frecuentemente requeridas para evitar problemas de rendimiento. Estos datos deben ser puestos en tablas separadas relacionadas por un identificador.

#### Los passwords almacenados en la BD se tendrán encriptados por seguridad. Desencriptarlos en la aplicación si es necesario.

#### No usar querys del tipo "select \* " a menos de que sea necesario, extraer sólo las columnas a utilizar.

#### Usar un framework o marco de trabajo ORM (Mapeo Relacional de Objetos) como Eloquent, si el código de la aplicación es lo suficientemente grande. Los problemas de rendimiento de los ORMs pueden manejarse detallando sus parámetros de configuración.

#### Particionar la base de datos separando las tablas que se usan mucho de las que no se usan tanto para un mejor desempeño.

#### Para bases de datos grandes, sensibles y sistemas de misión crítica, usar los servicios de recuperación de desastres y servicios de seguridad como el failover clustering, respaldos automáticos, replicación, etc.

#### La falta de documentación en una base de datos es mala idea. Documentar el diseño de base de datos con esquemas de entidad relacionales (ER) e instrucciones. Incluso escribe líneas de comentarios en triggers, procedimientos almacenados y otros scripts.

#### Usar índices para scripts frecuentemente usados en tablas grandes. Hay herramientas de análisis que puede ser usadas para determinar donde pueden estar definidos los indices. Para queries que extraen un rango de registros, índices agrupados usualmente son mejores. Para queries de punto los índices no agrupados son la mejor opción.

#### Un servidor de base de datos y un servidor web deben estar en máquinas diferentes. Esto provee de mayor seguridad y separan la carga de trabajo en dos CPUs y memoria diferentes.

#### La normalización debe ser usada cuando sea requerida para optimizar el performance. Una baja normalización puede repercutir en una repetición de datos, una sobre normalización puede tener efectos en el rendimiento a causa de las excesivas uniones entre tablas para extraer datos. Se debe mantener un equilibrio.

#### Auditar las sentencias update’s y delete’s mediante triggers que se activan cuando alguien elimina y/o actualiza campos de la BD.

#### Cuando se van hacer cambios a la base, avisar a todos los involucrados en el proyecto ya que pueden afectar a la aplicación y tomar las medidas pertinentes.

#### **Modelos en Laravel:**

#### Los modelos son las clases que tienen el código donde se realizará el acceso a los datos, por tanto, los accesos a las bases de datos son responsabilidad de los modelos. En Laravel se han diseñado para que funcionen con muy poco código y para ello el programador debe respetar una serie de convenciones.

#### Esas convenciones son las que permiten que podamos ahorrar código. Si no las respetamos simplemente nos veríamos obligados a escribir algo de código para adaptar los modelos a nuestras circunstancias. Estas reglas son importantes para comenzar:

#### El nombre del modelo tiene que ser en singular y mayúscula y la tabla a la que acceden estará en plural y minúscula. Por ejemplo: modelo "Usuario" para tabla "usuarios". Modelo "Producto" para tabla "productos".

#### Por defecto busca siempre la llave primaria con el nombre de campo "id".

#### *Eloquent* usa las columnas created\_at y updated\_at para actualizar automáticamente esos valores de tiempo.

#### Ejemplo: usuarios\_contratos (Ha de ser en plural y en minúsculas)

#### El modelo tiene que ser: UsuarioContrato (Ha de ser en Singular y en Mayúsculas)

#### Si por lo que sea no podemos cambiar el nombre a las tablas y su nombre no casa con las convenciones de Laravel, podemos introducir una variable $table indicando el nombre de la tabla de nuestra base de datos.

#### Para Laravel poder definir que tabla es la que esta ligada a este modelo le es suficiente con realizar la conversion a notacion underscore y plural, dando como resultado la tabla: users.

#### Y esto aplica para cuando queremos crear nuestros modelos, si tenemos una tabla en la base de datos con la que queremos trabajar que se llama user\_profiles, vemos que se encuentra con las convenciones para tablas de bases de datos (plural y underscore), entonces el modelo para esta tabla cambiando las convenciones seria: UserProfile (singular y UpperCamelCase).

# Convenciones y buenas prácticas de programación

#### Tener buenas practicas a la hora de programar nos puede evitar muchos dolores de cabeza, la mayoría de los programadores se enfocan en el código, no en el formato del mismo, esto podría ser un error, mejorar las prácticas de programación puede traer muchos beneficios, es posible generar código más consistente, mejoramos la lectura del mismo y facilitamos la colaboración con otros programadores. Una de las mejores maneras de lograr esto, es mediante el uso de estandartes, para ello se adoptará el siguiente:

#### **1. Activar el reporte de errores**

#### error\_reporting(E\_ALL) es tu amigo. Utilízalo, junto con ini\_set(‘display\_errors’, ‘On’), para ver todas las advertencias del compilador de PHP, saber qué métodos están depreciados, índices inexistentes, etc.

#### **2. No emplear etiquetas cortas**

#### Para que el intérprete de PHP ejecute un código, éste ha de estar delimitado por las etiquetas de apertura y cierre de PHP: <?php ?>. Si en la configuración de PHP de nuestro servidor (archivo php.ini) se permite el uso de etiquetas cortas (<? ?>) el script se ejecutará exactamente igual. Pero, ¿qué ocurre si cambiamos a otro servidor en el que no estén habilitadas? Nuestras páginas dejarán de funcionar como esperamos y, lo que es más grave, puede resultar en que todo nuestro código fuente se imprima por pantalla como texto plano (ya que no habrá sido interpretado), quedando a disposición de cualquier personaje.

#### **3. Operadores de comparación: === vs ==**

#### == (o !=) compara el valor, mientras que === (o !==) compara el valor y el tipo.

#### Dado que en PHP las variables no tienen un tipo asignado y éste puede cambiar “al vuelo” en tiempo de ejecución, conviene tener en cuenta cómo serán evaluadas las comparaciones en PHP:

#### <?php

#### $a = 0;

#### $b = "0"

#### $c = false;

#### var\_dump($a == $b); // 0 == 0 -> true

#### var\_dump($a === $b); // 0 == 0 && (integer) == (string) -> false

#### var\_dump($a == $c); // 0 == false -> true

#### Por otro lado, existen funciones que retornan un número entero. Como el valor lógico de cualquier entero es true, exceptuando el cero, que es false, deberemos tenerlo en cuenta a la hora de evaluar nuestras condiciones. Por ejemplo, la función strpos() busca la posición de la primera ocurrencia de una subcadena dentro de otra:

#### <php

#### $cadena = 'formación online';

#### $buscamos = 'm'; // presente en $cadena

#### $posicion = strpos($buscamos, $cadena); // $posicion = 3

#### $buscamos = 'z'; // ausente en $cadena

#### $posicion = strpos($buscamos, $cadena); // $posicion = false

#### $buscamos = 'f'; // presente en $cadena

#### $posicion = strpos($buscamos, $cadena); // $posicion = 0

#### Por lo tanto, si queremos comprobar la presencia o no de una cadena con strpos() no debemos comparar por valor (==) sino por tipo y valor (===).

#### **4. echo vs print**

#### Estas funciones realizan la misma tarea. Sin embargo, *echo* es sensiblemente más rápida que print.

#### **5. Concatenación de cadenas, comillas simples (‘) vs comillas dobles (“)**

#### Cuando trabajes con cadenas, evita siempre el uso de comillas dobles. La razón es que PHP analiza el contenido de las comillas dobles en búsqueda de variables que deban ser interpretadas, resultando en un tiempo de ejecución mayor.

#### Emplea siempre la función echo y concatena las cadenas con comas: echo ‘Hola’, $nombre, ‘, ¿qué te trae por aquí? requerirá menos tiempo al compilador que echo ‘Hola’ . $nombre . ‘, ¿qué te trae por aquí?’. Por lo visto en el punto anterior, el “peor caso posible” sería print "Hola $nombre, ¿qué te trae por aquí?"

#### **6. Búsqueda de cadenas y patrones case insensitive**

#### La búsqueda de una cadena sin importar mayúsculas/minúsculas (stripos()) es entre un 400% y un 600% más lenta que su equivalente case sensitive, strpos()).

#### En cuanto a las expresiones regulares, las búsquedas sensibles, preg\_match(“/$patron/”, $cadena), son, como norma, ligeramente más eficaces que su equivalente no sensible: preg\_match(“/$patron/i”, $cadena).

#### Si las coincidencias se realizan de modo iterativo (dentro de un bucle for, while, foreach), es recomendable convertir a lowercase o uppercase antes y realizar las operaciones en su versión case sensitive.

#### **7. Convenciones**

#### **Nombres de Clases:** en MixedCase. . Es una buena práctica usar el “estilo PEAR” para manejar la jerarquía de nuestras clases, esto también ayuda a ubicar los archivos de forma jerárquica y ordenada.

#### Ejemplo: ElNombreDeMiClase

#### **Nombres de Métodos:** en camelCase. Ejempo: nombreDelMetodo()

#### Los métodos llamados “Accessors” utilizaran los prefijos set y get, usando la regla anterior, por ejemplo setVar1 y getVar1.

#### **Constantes:** siempre en ALL\_CAPS. Ejemplo: COLOR\_DEFINIDO\_PARA\_MI

#### **Variables, propiedades, funciones y métodos:** en camelCase. Ejemplo: $variableEnLaQueAlmacenoAlgo

#### **Propiedades y métodos privados**: precedidos por un guión bajo (underscore-prefixed). Ejemplo: $\_miPalabraSecreta

#### Veámoslo en un ejemplo para ilustrar:

#### <?php

#### class MyNewClass{

#### const LIMIT\_OF\_ITERATIONS = 10;

#### public $myProperty;

#### public $myOtherProperty;

#### protected $\_myProtectedProperty; // observa guión bajo por no ser public

#### // constructor

#### function \_\_construct(){

#### }

#### public function hacerAlgoConMiClase($parametroRecibido){

#### for ($i = 0; $i < self::LIMIT\_OF\_ITERATIONS; $i++){

#### // lo que sea

#### }

#### }

#### public function getMyProtectedProperty(){

#### return $this->\_myProtectedProperty;

#### }

#### }

#### **8. Emplea un Entorno de Desarrollo Integrado (IDE)**

#### Un IDE es un editor de código que ofrece la posibilidad de depurar, autocompletar e identar el código. Existen multitud de alternativas, gratuitas y de pago. Entre las opciones gratuitas destacan Eclipse y Netbeans; por su parte, Zend Studio, PHP Designer y PHPStorm destacan como opciones de pago.

#### **9. Separa y reutiliza tu código**

#### Evita tener archivos con cientos o miles de líneas. Agrupa las funciones que vayas a emplear con frecuencia en archivos e inclúyelos para su posterior reutilización. Ejemplo:

#### 

#### <?php

#### class Ipl\_Util\_Date{

#### ...

#### public static function timestampToSpanishFormat($date, $dateSeparator,

#### $timeSeparator = ' a las '){

#### if ($date == '') return null;

#### $temp = explode(' ', $date);

#### if (sizeof($temp) != 2) return null;

#### $dates = explode('-', $temp[0]);

#### return $dates[2] . $dateSeparator . $dates[1]

#### . $dateSeparator . $dates[0]

#### . $timeSeparator . $temp[1];

#### }

#### }

#### Así, si alguna vez queremos convertir una fecha en formato timestamp a un formato con el que estamos más familiarizados, sólo tenemos que hacer <?php echo Ipl\_Util\_Date::timestampToSpanishFormat($myDate, ‘/’) ?>.

#### **10. Al iterar arrays, fija el valor máximo fuera del bucle**

#### Cada llamada al método count() aumenta hasta en un 50% el tiempo de ejecución, según el tamaño del array.

#### // mal

#### for ($i = 0, $i < count($miArray); $i++){

#### ...

#### }

#### // bien

#### $limite = count($miArray);

#### for ($i = 0; $i < $limite; $i++){

#### ...

#### }

#### **11. Emplea el buffer de salida**

#### En lugar de utilizar echo ‘texto que sea‘, utiliza ob\_start() para empezar a almacenar el texto en el búffer de salida. Para terminar la captura, puedes emplear ob\_get\_contents() y ob\_get\_clean(), o ob\_end\_clean().

#### **12. Mantén las opciones de configuración en un archivo**

#### A medida que tu aplicación vaya creciendo necesitarás acceder en distintos lugares a determinados valores. Almacenándolos todos en un único lugar, evitarás tener que modificar todos los archivos cada vez que haya un cambio. Imagina que programas un carrito de la compra y en un determinado momento hay que cambiar el IVA.

#### **13. Incluyendo código**

#### include, require y sus derivados (include\_once, require\_once) son sentencias, no métodos. No debe ponerse la ruta al archivo entre paréntesis.

#### Emplea siempre include o require en vez de include\_once o require\_once para que el cacheado de código sea más efectivo.

#### **14. Codificación de caracteres**

#### Emplea UTF-8 (sin BOM) en lugar de ANSI. Lo que en ANSI u otra codificación puede ser un determinado caracter (una eñe, por ejemplo), en otra codificación puede ser algo completamente distinto. Si tu público emplea tu misma codificación no hay problema pero, si en un momento dado alguien de, por ejemplo, Rusia entra en tu página, verá la pantalla llena de caracteres sin sentido.

#### Si tu web va a estar disponible en más de un idioma, es indispensable que tu codificación sea UTF-8 (incluyendo la codificación y colación de la base de datos).

#### **15. Minimiza el número de variables globales**

#### Operar con variables globales es en torno a un 200% más costoso que hacerlo con locales.

#### **16. Directiva AllowOverride**

#### Si tu servidor web es Apache, emplea la directiva AllowOverride None donde sea posible.

#### Imagina una estructura de directorios como la siguiente: media/images/2013/03/my\_image.jpg. Si AllowOverride está en All, antes de que Apache pueda servir el recurso my\_image.jpg, debe recorrer todos los directorios desde la raíz hasta “/03” en búsqueda los archivos .htaccess por si en alguno de ellos se especificase alguna directiva sobre el objeto de la petición.

0

#### **17. No implementes todas las estructuras de datos como objetos**

#### Los array son más rápidos y consumen menos memoria que instanciar un objeto.

#### **18. Incrementar o Decrementar variables: ++$i vs $i++**

#### Pre-incrementar (++$i) es en torno a un 10% más rápido que post-incrementar ($i++). La razón es que cuando hacemos post-incremento, PHP necesita crear una variable temporal en la que almacenar el valor a ser incrementado.

#### **19. Minimiza el número de consultas a la base de datos**

#### Realizar una consulta es costosa en términos temporales. En todas aquellas ocasiones en que los datos no vayan a cambiar con mucha frecuencia, es interesante realizar una única vez la consulta y almacenar el resultado de ésta en un archivo de texto plano. Cuando se produzcan cambios en la consulta, se vuelve a generar el archivo a ser incluido. Por ejemplo: imagina que tienes una web de películas, y en algún sitio de la página se ve el listado de películas mejor valoradas. Para obtener las diez películas mejor valoradas tendrías que hacer una consulta similar a la siguiente:

#### $sqlVotos = 'SELECT p.id, p.titulo, SUM(v.voto) AS suma\_votos,

#### COUNT(v.id) AS numero\_votos,

#### (SUM(v.voto)/COUNT(v.id)) AS valoracion

#### FROM peliculas p

#### INNER JOIN votos v ON p.id=v.id

#### GROUP BY v.id\_pelicula

#### ORDER BY valoracion DESC

#### LIMIT 0,10';

#### $resVotos = mysql\_query($sql);

#### $votos = mysql\_fetch\_array($res, MYSQL\_ASSOC);

#### echo "<h3>Mejor valoradas</h3>n<ul class="list">";

#### foreach ($votos as $voto){

#### echo '<li><a href="....">' . $voto['titulo']

#### . ' (' . $voto['valoracion'] . ')</a></li>' . "n";

#### }

#### echo '</ul>';

#### Este script debería ejecutarse para todas y cada una de las peticiones de página que tuviera tu web. Lo que podemos hacer es ejecutar el siguiente código cada vez que se produzca una votación, o bien un Cron que se ejecute cada 30 minutos o el intervalo que consideremos:

#### $filename = PATH\_TO\_VOTACIONES\_FILE . 'votos.txt'; // ruta al archivo

#### $handle = fopen($filename, 'w+'); // abrimos el archivo

#### // ejecutamos la consulta anterior ($sqlVotos / $resVotos)

#### foreach ($votos as $voto){

#### fwrite($handle, '<li><a href="...">'. $voto['titulo']

#### . ' (' . $voto['valoracion'] . ')</a></li>' . "n";

#### }

#### fclose($handle);vo

#### Entonces, sólo tendríamos que hacer un require PATH\_TO\_VOTACIONES\_FILE . ‘votos.txt’; para mostrar los votos.

#### **20. isset()**

#### La función isset() (isSet) resulta de una utilidad tremenda. Nos sirve tanto para saber si una determinada variable ha sido inicializada como para comprobar que un índice existe e incluso para trabajar con longitudes de cadenas de un modo más eficiente: Si en un determinado momento necesitamos comprobar que una cadena tiene al menos una determinada longitud, por ejemplo que el nombre de usuario tenga más de 4 caracteres, lo más inmediato que se nos puede ocurrir es utilizar strlen()

#### if (strlen($username) < 5){

#### ....

#### }

#### Pero, en tanto que strlen es una función, PHP necesita realizar un trabajo previo para ejecutar la llamada (convertirla a lowercase y buscarla en la tabla hash de funciones), además de ejecutar la propia función.

#### Por contra, isset no es una función sino una construcción del lenguaje. Esto significa que PHP no tendrá que hacer ninguna operación previa ni habrá sobrecarga.

#### Así, si queremos comprobar que una variable tiene más de una determinada longitud, podemos hacerlo de un modo más efectivo de la siguiente manera:

#### if (isset($username{5})){

#### ....

#### }

#### **21. No utilizar código redundante.**

#### El concepto DRY (Don’t repeat yourself) es aplicable para todos los lenguajes de programación, intenta parametrizar todo lo que puedas. Además combinado con un buen IDE nos ayudará a evitar errores absurdos.

#### **22. Utiliza una nomenclatura comprensible.**

#### Los nombres deben ser descriptivos, no hay nada peor como vérselas con un código en el que no encontremos lo que buscamos. Tu nombre puede ser muy bonito pero no es aconsejable para una variable (verídico).

#### **23. Comenta todo lo que puedas.**

#### Comentar entre líneas a veces puede resultar una tarea tediosa, pero siempre tenemos que intentar dedicarle unos minutos.

#### Comenta las partes de tu código que sean relevantes, evitar poner antes de tu ciclo: “Recorro el arreglo…”, los comentarios son de gran ayuda pero podrían llegar a estorbar si no se usan bien

#### **24. Valores boleanos y null**

#### En los lenguajes donde sea indiferente el usar “null”, “false” y “true” con mayúsculas o minúsculas, siempre serán usados en minúsculas.

#### **25. Indentación…**

#### Se recomienda usar solamente espacios no tabs, 4 espacios por nivel de indentación, la mayoría de los editores pueden ser configurados para que conviertan nuestros tabs en espacios ayudándonos a cumplir este requisito.

#### **EL MANUAL OFICIAL DE PHP SERÁ TU MEJOR AMIGO, SOBRETODO SUS COMENTARIOS.**