

## Guion para la Unidad 7: Organizando Código

Sitio: [Centros - Granada](#)  
Curso: Desarrollo web en entorno servidor  
Libro: Guion para la Unidad 7: Organizando Código

Imprimido por: Aguilera Aguilera, Javier  
Día: miércoles, 22 de diciembre de 2021, 12:21

## Tabla de contenidos

- 1. Separa la lógica de la presentación**
- 2. Ejemplos de código reutilizable**
- 3. Patrones de diseño**
  - 3.1. Modelo Vista Controlador
  - 3.2. Patrón de creación Factory Method
  - 3.3. Patrón de comportamiento Observer
  - 3.4. Patrón estructural Decorator
- 4. Composer / Librerías externas**
  - 4.1. Instalación y configuración
  - 4.2. Instalando dependencias
  - 4.3. Actualización de dependencias
  - 4.4. Cargar las clases automáticamente
  - 4.5. Versionado semántico
- 5. Librerías**
  - 5.1. Generar PDF
  - 5.2. Paginación
  - 5.3. Manipular imágenes
  - 5.4. Depurar código
- 6. Herramientas de depuración de código**
  - 6.1. Instalación de Xdebug
  - 6.2. Depuración de código
- 7. Git**
- 8. URLs amigables o limpias**

## 1. Separa la lógica de la presentación

**Lógica de negocio:** lógica específica de la aplicación, donde realmente se desarrolla la funcionalidad de ésta. Procesa la información que introduce el usuario y maneja la base de datos.

**Lógica de presentación.** Parte de la aplicación que se ocupa de mostrar la información del usuario)

Uno de los temas principales en el diseño de aplicaciones es la reusabilidad del código. Para conseguirlo, las aplicaciones se dividen en partes que se relacionan entre sí, pero que también tienen sentido de manera independiente.

A pequeña escala, se crean funciones y clases que, si están bien diseñadas, se pueden reutilizar en otras aplicaciones. A nivel de diseño de aplicaciones, se plantea una arquitectura con varios componentes desacoplados que se ocupan de las diferentes partes de la aplicación.

En los ejemplos realizados hasta ahora, la lógica de negocio y la de presentación están completamente mezcladas.

Desacoplar la lógica de negocio de la de presentación tiene varias ventajas:

- El código es más fácil de mantener y modificar, al no estar mezclado.
- El código del servidor se puede reutilizar con otros clientes.
- Se puede trabajar en la parte del cliente y en la del servidor en paralelo.

La reutilización de código nos permite ahorrar gran cantidad de horas en el desarrollo de aplicaciones, ya que cuando tenemos un código probado y funcionando no hay que repetirlo.

Una solución para reutilizar código la hemos visto en las unidades anteriores y consiste en colocar el fragmento que será común en otro script, al que invocaremos cuando sea necesario.

Para ello, en PHP tenemos cuatro sentencias: **include()**, **require()**, **include\_once()** y **require\_once()**.

Si se desea se pueden crear plantillas base para nuestros proyectos con PHP y Bootstrap.

## 2. Ejemplos de código reutilizable

- Archivo donde se declaran los parámetros de la conexión como constantes para tenerlas disponibles en toda la aplicación
- Crear una clase reutilizable que sirva para conectarse a una base de datos con PDO
- Crear una clase reutilizable que sirva para conectarse a una base de datos con MySQLi

En estos ejemplos se parte de que tenemos una carpeta con nombre lib en la que se crea un fichero con el nombre BaseDatos.php con este código.

- Cargador de clases

NOTA: Estos ejemplos pertenecen al libro Curso de PHP8 y MySQL 8 . Editorial: Anaya

### 3. Patrones de diseño

Los patrones de diseño son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software .

Un patrón de diseño resulta ser una solución a un problema de diseño. Esta solución se habrá comprobado su efectividad para problemas similares y debe ser reutilizable en distintas circunstancias.

El uso de patrones no es obligatorio, pero sí recomendable ( sin abusar de ellos o forzar su uso)

Libro recomendado

### 3.1. Modelo Vista Controlador

El patrón de diseño de software MVC se encarga de separar la lógica de negocio de la lógica de presentación. Este patrón facilita la funcionalidad, mantenibilidad, y escalabilidad del sistema, de forma cómoda y sencilla.

Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador (o lo que es igual, dividir la aplicación en tres capas o niveles de abstracción).

Al desacoplar los elementos de la aplicación, se consigue código reusable, Además, permite el desarrollo en paralelo, con equipos independientes para cada capa.

El patrón MVC es uno de los más extendidos para el desarrollo de aplicaciones. Actualmente muchos frameworks utilizan este patrón o alguna de las muchas variantes que han surgido.

Las tres capas de la aplicación son:

- **Modelo:** en el modelo está la lógica de negocio. Gestiona todo lo relacionado con la información y la interacción con los datos de nuestra aplicación. Todas las peticiones de acceso a la base de datos pasarán por esta capa.
- **Vista:** la interfaz gráfica. Es la encargada de mostrar la información al usuario de forma gráfica y legible. Una vista muestra una parte del modelo al usuario.
- **Controlador:** el intermediario entre la vista y el modelo. Se encarga de controlar las interacciones del usuario en la vista. El usuario solicitará información por medio de la vista y esta hará la petición al controlador. Posteriormente, este, realizará la petición al modelo.



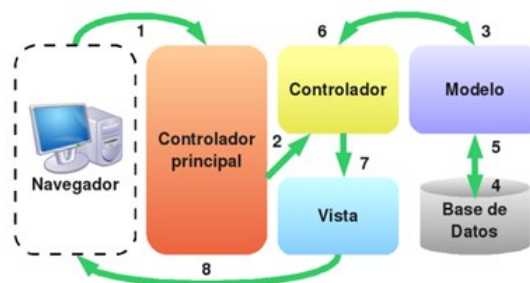
El funcionamiento básico del patrón MVC, puede resumirse en:

- El usuario realiza una petición.
- El controlador captura la petición.
- Hace la llamada al modelo correspondiente.
- El modelo será el encargado de interactuar con la base de datos.
- El controlador recibe la información y la envía a la vista.
- La vista muestra la información.

En la práctica por cada controlador creará una carpeta con las vistas.

**El controlador frontal** (por defecto suele invocarse desde el fichero index.php): Es un patrón de diseño que se basa en usar un controlador como punto inicial para la gestión de las peticiones. Así, usaremos un único punto de entrada en nuestra aplicación redirigiendo las peticiones de los usuarios al/los controladores específicos.

El controlador gestiona estas peticiones, y realiza algunas funciones como: comprobación de restricciones de seguridad, manejo de errores, mapear y delegación de las peticiones al controlador específico que se encargará de generar la vista adecuada para el usuario. La siguiente figura muestra un esquema de ello:



El controlador frontal se encarga de averiguar qué controlador debe cargar y de ejecutar la acción solicitada por el usuario. Es decir, cargará un fichero u otro en función de lo que llega por la URL.

En realidad, el controlador frontal no solo detecta la acción que se tiene que ejecutar, sino que también ejecuta el código común a todas las acciones, por ejemplo:

- Carga la configuración y las librerías
- Decodifica la URL de la petición para determinar la acción a ejecutar y los parámetros de la petición.
- Si la acción no existe, redireccionará a la acción del error 404.
- Activa los filtros (por ejemplo, si la petición necesita autenticación) y los ejecuta
- Ejecuta la acción y produce la vista.
- Muestra la respuesta.



### 3.2. Patrón de creación Factory Method

Sirve para **crear objetos sin tener que especificar su clase exacta**. Esto quiere decir que el objeto creado puede intercambiarse con flexibilidad y facilidad.

- [Ejemplo 1](#)
- [Ejemplo 2](#)



### 3.3. Patrón de comportamiento Observer

Define una dependencia uno-a-muchos entre objetos, de tal forma que, cuando el objeto cambie de estado, todos sus objetos dependientes sean notificados automáticamente.

- [Ejemplo 1](#)
- [Ejemplo 2](#)

### 3.4. Patrón estructural Decorator

Permite añadir funcionalidad a una clase que no es posible extender por cualquier causa. Esto nos permite no tener que crear clases sucesivas que vayan heredando de la primera y añadiendo nueva funcionalidad, sino otras que las implementan y asocian a la primera.

- [Ejemplo 1](#)
- [Ejemplo 2](#)

## 4. Composer / Librerías externas

Composer es un gestor de dependencias en proyectos, para programación en PHP. Eso quiere decir que nos permite gestionar (declarar, descargar y mantener actualizados) los paquetes de software en los que se basa nuestro proyecto PHP.

Es decir, para usar librerías externas (de terceros desarrolladores) instalaremos el gestor de dependencias Composer, que además nos permite mantenerlas siempre actualizadas.

Composer no es un administrador de paquetes. Trabaja con paquetes o librerías, pero los administra por proyectos, instalándolos en un directorio (/vendor) dentro del proyecto. Por defecto no instala nada globalmente, por lo que es un gestor de dependencias. También tiene soporte para proyectos globales a través del comando global.

## 4.1. Instalación y configuración

Composer lo descargamos desde <https://getcomposer.org/download/> y en su instalación elegimos la versión de PHP que estamos usando.

Si trabajamos con Windows simplemente descargamos el instalador y lo ejecutamos.

Una vez instalado abrimos la consola de Windows para comprobar si se ha instalado correctamente con el comando: composer -v.

Ahora ya podemos buscar cualquier librería, crear un fichero con las dependencias e instalarlas y mantenerlas actualizadas.

Para ver la instalación en otras plataformas como Linux o Mac sigue las instrucciones este [enlace](#)

## 4.2. Instalando dependencias

Para localizar librerías disponemos de un repositorio oficial de paquetes instalables con [Composer](#), [Packagist](#) ( <https://packagist.org/>).

Además, en esta web, encuentras información sobre cada paquete y el código necesario para declarar tu dependencia en el JSON de [Composer](#).

Antes de descargar la librería que deseamos instalar, desde la consola de Windows, accedemos a la carpeta de nuestro proyecto donde estará ubicada. Una vez aquí, ejecutaremos el comando **composer require** seguido del nombre del paquete a instalar.

Al ejecutar este comando se genera un archivo JSON ([composer.json](#)), y se descarga las librerías y las dependencias necesarias.

Éste será el archivo de configuración donde indicamos los paquetes que vamos a requerir, el autor de proyecto, las dependencias, etc.

Cuando volvamos a entrar en nuestro proyecto encontraremos:

- una nueva carpeta llamada */vendor* con todas las librerías de terceros que se ha descargado [Composer](#),
- un archivo `autoload.php` que nos carga todas las clases necesarias,
- un archivo [composer.json](#) con el nombre de las dependencias ,al que después podremos añadir todas las que deseemos, y otros metadatos.
- y el archivo [composer.lock](#) que almacena la versión exacta de todas las dependencias que tenemos instaladas. De esta forma, cualquier persona que desee utilizar el proyecto tendrá exactamente las mismas versiones de las dependencias ([Composer](#) descarga la versión que le indique este fichero). Con ello, evitamos errores de compatibilidad.

## 4.3. Actualización de dependencias

Si queremos actualizar nuestras dependencias, a las últimas versiones, podemos hacerlo utilizando el comando: `composer update`

El comando `composer update` lee siempre el fichero `composer.json` e instala las dependencias de ese fichero. Es decir, busca las versiones más recientes de las librerías y las instala.

Después de instalar los paquetes crea, en el directorio donde se ha ejecutado este comando, el fichero `composer.lock` o lo actualiza si ya existe.

El archivo `composer.lock` evita que `Composer` instale automáticamente las últimas versiones de las dependencias.

Si solo queramos instalar o actualizar una dependencia, podemos hacerlo indicando su nombre después del comando update: `composer update nombre-del-paquete`

Debemos de tener cuidado al descargar las dependencias ya que el comando `composer install` descargará las dependencias teniendo en cuenta la versión almacenada en el fichero `composer.lock`. Sin embargo, cuando ejecutamos `composer update` se omite el contenido de este fichero y se descarga la última versión disponible de todas nuestras dependencias.

Para saber más sobre la diferencia entre `composer install` y `composer update` consulta el siguiente [artículo](#).

## 4.4. Cargar las clases automáticamente

A medida que nuestro proyecto crece, generalmente nos encontramos con el problema de tener que incluir todas las clases que vayamos a utilizar en cada uno de nuestros scripts. Para solucionar este inconveniente, PHP cuenta con un sistema de 'Autocarga de clases' (Autoload) que permite usar nuestras clases sin tener que escribir los 'include' correspondientes.

Esta funcionalidad se implementa por medio del archivo `vendor/autoload.php`.

Incluyendo este archivo en nuestro proyecto podemos utilizar cualquier clase de la librería instalada a través de Composer sin tener que incluirla explícitamente en nuestro código.

Cargaremos el autoload con el comando:

```
require 'vendor/autoload.php';
```

 (también se podría usar `include`)

Para usar la librería se hace con *use* y el *namespace* de la clase y ya podré usar cualquier objeto.

## 4.5. Versionado semántico

Versionado semántico (SemVer)



## 5. Librerías

Veremos algunas librerías interesantes que podemos instalar con Composer

## 5.1. Generar PDF

Existen varias librería para generar documentos PDFs desde PHP, por ejemplo: [domPDF](#), [PDFLib](#), [html2pdf](#), etc.

En este caso, instalaremos librería [html2pdf](#) que nos permite maquetar con HTML y CSS.

- La buscamos en el repositorio <https://packagist.org/packages/spipu/html2pdf>
- Accedemos desde la consola a la carpeta de nuestro proyecto.

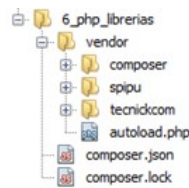
En mi caso, ésta es la ruta de la carpeta donde se va a realizar el ejemplo:

```
C:\wamp64\www\Clases\6_php_librerias>
```

- Ejecuto el comando que se indica en Packagist: `composer require spipu/html2pdf`

```
C:\wamp64\www\Clases\6_php_librerias>composer require spipu/html2pdf
```

Ahora el contenido de la carpeta donde he realizado la instalación será:



En el archivo `composer.json` tendremos las dependencias que por ahora sólo será la de `html2pdf`

```

{
    "require": {
        "spipu/html2pdf": "^5.2"
    }
}

```

Y en el fichero `autoload.php` que nos cargará todas las clases necesarias.

- Para comenzar a utilizar la librería necesitamos un nuevo fichero php para convertir a PDF. Comenzaremos creando carpeta para este ejercicio llamada `generarPDF` y dentro un archivo `index.php` en el que cargamos el `autoload.php` con la sentencia:

```
require "../vendor/autoload.php";
```

*o bien usando la constante predefinida `DIR`:* `require __DIR__."/vendor/autoload.php";`

- Para usar la librería necesito la sentencia `use` seguida del **Namespace**:

```
use Spipu\Html2Pdf\Html2Pdf;
```

- Una forma de crear el PDF es escribiendo directamente en el `index.php` un objeto HTML y después exportarlo al archivo que deseemos:

```

require "../vendor/autoload.php"; // para tener acceso a las clases que hay en vendor

// Para poder utilizar la libreria y cargar el Namespace
use Spipu\Html2Pdf\Html2Pdf;

// Creo un objeto para generar el pdf
$html2pdf = new Html2Pdf();

$html = "<h1>Ejemplo 1: Composer </h1>";
$html .= "<p>Usando una libreria de PHP para hacer PDFs</p>";

$html2pdf->writeHTML($html);

// exportar el objeto HTML a un PDF
$html2pdf->output('pdf_generado.pdf'); // indico el nombre para mi fichero

```

- Pero, lo más normal es que no tengamos que escribir todo el HTML del documento que vas a generar en el `index.php`. Crearé un archivo `.php` donde escribo la estructura del documento que quiere pasar a PDF (una vista). Después recogemos la vista que vamos a imprimir, pasamos la vista a PDF y generamos el documento.

```
<?php
// Ejemplo tomando el HTML desde un archivo

// Cargar el autoload
require "../vendor/autoload.php";

// para usar la libreria
] use Spipu\Html2Pdf\Html2Pdf;

// recoger el contenido de la vista
ob_start();
require_once 'pdf_para_generar.php';
$html = ob_get_clean();

//Pasar la vista a PDF
$html2pdf = new Html2Pdf();

$html2pdf->writeHTML($html); // escribimos el contenido en el PDF
$html2pdf->output('pdf_generado.pdf'); //generamos el PDF
```

## 5.2. Paginación

Cuando tenemos que mostrar muchos datos en nuestra página es aconsejable que lo hagamos en diferentes páginas.

Para ver una posible solución nos descargaremos desde Packagist la librería Zebra\_Pagination y siguiendo las instrucciones lo instalamos en la carpeta que hemos creado para trabajar con librerías, que en mi caso es la siguiente:

```
C:\wamp64\www\Clases\6_php_librerias>composer require stefangabos/zebra_pagination
```

Una vez instalada, podremos comprobar que el archivo `composer.json` tenemos la línea de hace referencia a la nueva librería.

Para trabajar con este ejemplo crearemos una carpeta llamada paginación y dentro un archivo `index.php` donde haremos uso de la citada librería.

Para esta prueba también nos conectaremos a MySQL y mostramos información de la base de datos "empresa" que ya creamos en unidades anteriores.

Y, por último, usaremos la hoja de estilos CSS de esta librería para que quede mejor la presentación.

Recuerda que puedes consultar siempre los ejemplos que vienen incluidos.

## 5.3. Manipular imágenes

Descargamos desde Packagist la librería [phptumb](#) con el comando: `composer require masterexploder/phptumb`.

Con ella, las imágenes se pueden rotar, cortar, añadirles marcas de agua, definir su calidad, poner filtros, redondear esquinas, etc.

De nuevo en el fichero `index.php` cargaré el autoloader para poder trabajar con ella.

## 5.4. Depurar código

Instalaremos la librería FirePHP tal y como se indica en Pakagist <https://packagist.org/packages/firephp/firephp-core> .

Esto nos permitirá depurar código en consola del navegador.

Para que se puedan leer los logs de esta librería se necesita instalar en el navegador un plugin

Por ejemplo, en Chrome <https://chrome.google.com/webstore/detail/firephp-official/ikfbpappjhegehjflebknbhdocbgkdi>

Al tenerlo habilitado este plugin en el navegador, podemos usar la herramienta de desarrolladores para ver en la consola los elementos de nuestro script sin necesidad de usar `var_dump()`.

Video: <https://www.dailymotion.com/video/xa4chg>

## 6. Herramientas de depuración de código

En la unidad anterior hemos usado `var_dump()`, `echo`, `print` y `print_r` para ver lo que fallaba en la ejecución de nuestro script.

Este método de depuración es muy tedioso y requiere hacer cambios constantes en el código de la aplicación. También existe la posibilidad de que, una vez encontrado el fallo, nos olvidemos de eliminar en nuestras páginas las líneas creadas a propósito para la depuración, con los consiguientes problemas.

Por ello, comenzaremos a usar alguna herramienta de depuración específica que se integre con el entorno de desarrollo, permitiéndonos por ejemplo detener la ejecución cuando se llega a cierta línea y ver el contenido de las variables en ese momento.

Cuando se usan estas herramientas se minimiza o elimina por completo la necesidad de introducir líneas específicas de depuración en los programas y se agiliza el procedimiento de búsqueda de errores.

Nosotros usaremos la extensión de código libre Xdebug, que se integra perfectamente con el IDE que estamos usando, NetBeans.

Si la depuración está activada, Xdebug controla la ejecución de los guiones en PHP. Puede pausar, reanudar y detener los programas en cualquier momento. Cuando el programa está pausado, Xdebug puede obtener información del estado de ejecución, incluyendo el valor de las variables (puede incluso cambiar su valor).

Xdebug es un servidor que recibe instrucciones de un cliente, que en nuestro caso será NetBeans. De esta forma, no es necesario que el entorno de desarrollo esté en la misma máquina que el servidor PHP con Xdebug.

La extensión Xdebug suele venir instalada en las pilas de aplicaciones XAMPP, MAMP y WAMP.

Puedes comprobar si está instalada con el script:

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

## 6.1. Instalación de Xdebug

En este ejemplo realizaré la instalación en la máquina virtual donde tengo instalado WampServer

Y en mi archivo php.ini tengo lo siguiente:

```
; XDEBUG Extension
[xdebug]
zend_extension="c:/wamp64/bin/php/php7.4.9/zend_ext/php_xdebug-2.9.6-7.4-vc15-x86_64.dll"
xdebug.remote_enable = off
xdebug.profiler_enable = off
xdebug.profiler_enable_trigger = Off
xdebug.profiler_output_name = cachegrind.out.%t.%p
xdebug.profiler_output_dir = "c:/wamp64/tmp"
xdebug.show_local_vars=0
```

Viene preinstalado, pero no está activado.

Si no está instalado, lo mejor es ir a su página oficial y comprobar antes la versión que debo de instalar en mi máquina virtual.

- Ejecutamos el script con phpinfo()
- Copiamos todo lo que nos muestra (Ctrl-a)
- Y lo pegamos en <https://xdebug.org/wizard>
- Directamente nos dirá los pasos que tenemos que hacer y el archivo que debemos de descargar. En mi caso, tengo que descargar un archivo:

### Summary

- **Xdebug installed:** 2.9.6
- **Server API:** Apache 2.0 Handler
- **Windows:** yes - Compiler: MS VC 15 - Architecture: x64
- **Zend Server:** no
- **PHP Version:** 7.4.9
- **Zend API nr:** 320190902
- **PHP API nr:** 20190902
- **Debug Build:** no
- **Thread Safe Build:** yes
- **OPcache Loaded:** yes
- **Configuration File Path:** C:\Windows
- **Configuration File:** C:\wamp64\bin\apache\apache2.4.46\bin\php.ini
- **Extensions directory:** c:\wamp64\bin\php\php7.4.9\ext

### Instructions

- Download [php\\_xdebug-2.9.6-7.4-vc15-x86\\_64.dll](#)
- Move the downloaded file to c:\wamp64\bin\php\php7.4.9\ext
- Update C:\wamp64\bin\apache\apache2.4.46\bin\php.ini and change the line  

```
zend_extension = c:\wamp64\bin\php\php7.4.9\ext\php_xdebug-2.9.6-7.4-vc15-x86_64.dll
```

 Make sure that  

```
zend_extension = c:\wamp64\bin\php\php7.4.9\ext\php_xdebug-2.9.6-7.4-vc15-x86_64.dll
```

 is **below** the line for OPcache.
- Please also update `php.ini` files in adjacent directories, as your system seems to be configured with a separate `php.ini` file for the web server and command line.
- Restart the webserver

Si ya está instalado debemos modificar el archivo php.ini para que tengamos la siguiente configuración, añadiendo las líneas que sean necesarias.

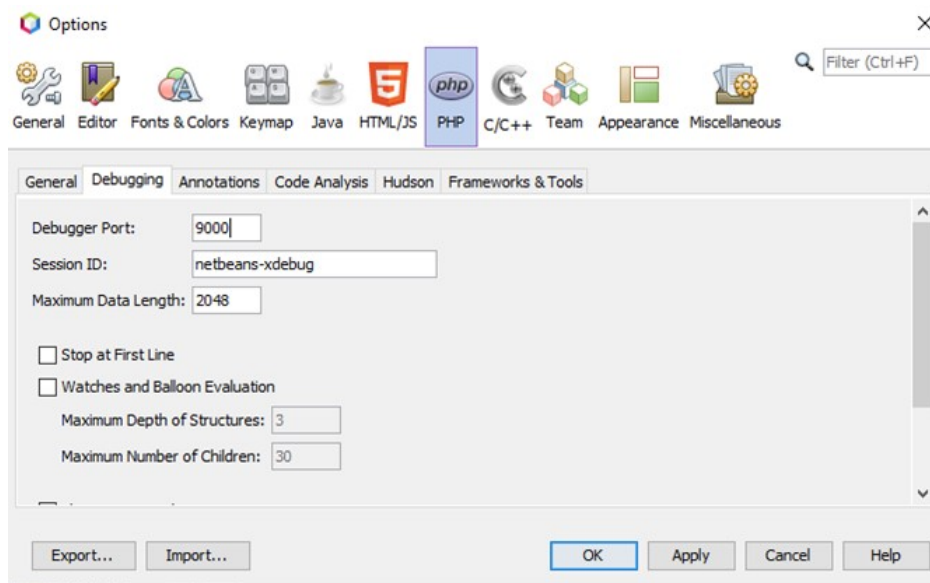
```
xdebug.remote_enable = on
xdebug.remote_handler=dbgp
xdebug.remote_host=localhost
xdebug.remote_port=9000
```

Hay que tener en cuenta que si el puerto 9000 está ocupado deberás usar otro, como por ejemplo el 9001.

Una vez configurado el servidor debemos de configurar nuestro cliente NetBeans.

Comprobamos en Tools > Options > Debugging que el número de puerto del campo Debugger Port se corresponde con el que has configurado en el php.ini (9000 por defecto).



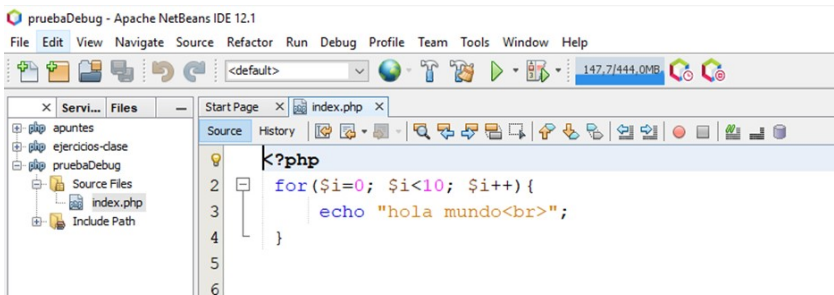


Si quieres más información puedes consultar <http://wiki.netbeans.org/HowToConfigureXDebug>

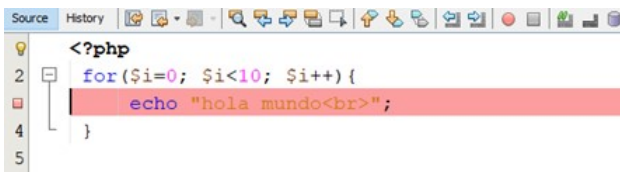
## 6.2. Depuración de código

Tan sólo tienes que hacer un Debug > Debug project, o pulsar la combinación de teclas Ctrl + F5, o clicar en el botón Debug project de la barra.

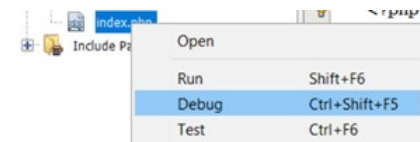
Por ejemplo, supongamos que tenemos el siguiente código:



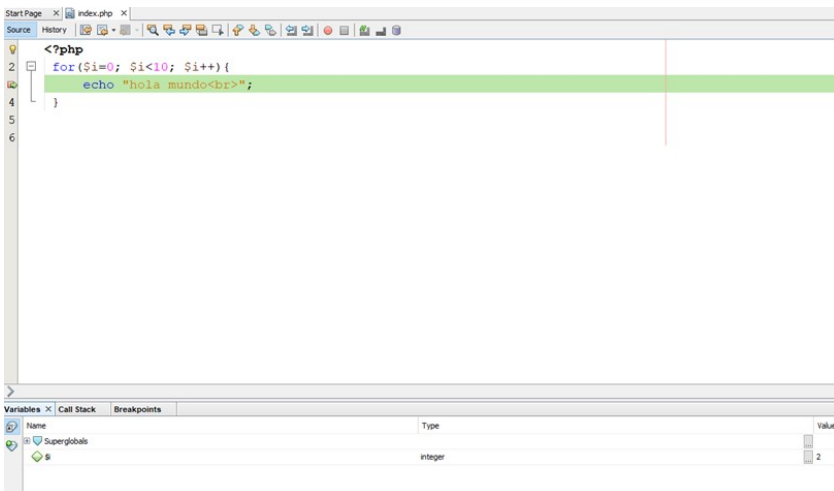
Creamos un punto de interrupción haciendo clic en la línea 3



Al ejecutarlo depurándolo con Debug Project



Y podemos ver cómo van cambiando nuestras variables dentro del bucle



En la página oficial tienes más información de cómo depurar <https://netbeans.org/kb/docs/php/debugging.html>



## 7. Git

- 1.1. Instalación y configuración
- 1.2. Trabajo con repositorios locales
- 1.3. Trabajo con repositorios remotos
- 1.4. Creación repositorio remoto
- 1.5. Trabajo con ramas locales
- 1.6. Trabajo con ramas remotas

## 8. URLs amigables o limpias

Tener URLs amigables, que además reflejen el contenido de nuestro sitio web, es uno de los factores que influyen en el posicionamiento en buscadores.

Ejemplos de URL:

- o No amigable: <http://mitienda.es/empresa.php?empresa=quesos-los-pastores>
- o Amigable: <http://mitienda.es/empresa/quesos-los-pastores>

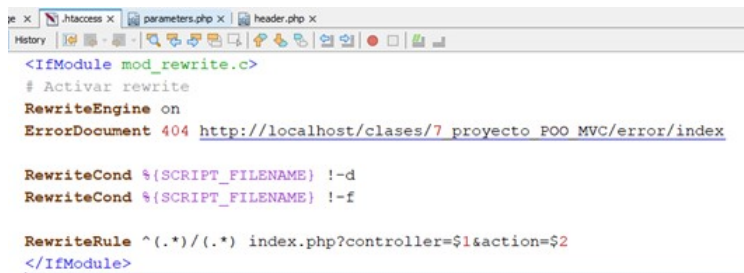
Como puedes comprobar, en una URL amigable no se muestran los caracteres ? ni =, tampoco aparece el nombre del fichero php. Por lo tanto, hacer una URL limpia consiste en usar el nombre de la variable (\$\_GET) como si fuese una carpeta de nuestro directorio y el valor de la variable como si fuese el nombre de una página.

Una de las formas de hacer la URLs amigables en PHP es a través del archivo **.htaccess**. (Fichero especial, que nos permite definir diferentes directivas de configuración para cada directorio sin necesidad de editar el archivo de configuración principal de Apache).

Para que las URLs de mi proyecto sean limpias y amigables necesitamos:

- o El módulo rewrite de Apache instalado y activado
- o Hacer uso de este módulo con un fichero .htaccess en el que reescribimos la URL, es decir, creamos el contenido de este fichero en la raíz y ponemos en él la expresión regular que va a limpiar nuestra URL. (Dentro del fichero .htaccess tendremos un módulo de Apache con reglas de reescritura)
- o Cambiar los <a href=.... de nuestra aplicación y el sitemap para que la nueva configuración de URL sea real.

Ejemplo:



```
<IfModule mod_rewrite.c>
# Activar rewrite
RewriteEngine on
ErrorDocument 404 http://localhost/clases/7 proyecto POO MVC/error/index

RewriteCond %{SCRIPT_FILENAME} !-d
RewriteCond %{SCRIPT_FILENAME} !-f

RewriteRule ^(.*)/(.*) index.php?controller=$1&action=$2
</IfModule>
```

Las líneas de RewriteCond son muy importantes ya que le dicen a Apache que sólo puede crear direcciones amigables si el directorio especificado no existe. La primera línea evita los directorios (I-d) y la segunda línea los archivos (I-f).

En la última línea escribimos la regla con RewriteRule.

- El carácter ^ significa el comienzo de la expresión.
- (.\*?) es la cadena que queremos convertir, es decir, el formato de la URL limpia, lo que el usuario pondrá en barra de direcciones. en este caso se podría leer como todo lo que empiece lo que sea /lo\_que\_sea
- La siguiente parte es el nombre real, es decir, el sucio. Y \$2 es donde se va a guardar el valor de la variable \$\_GET. Coincide con lo que vemos después de la / en la URL limpia.

Veremos con más profundidad este tema en el siguiente proyecto que realizaremos en clase.