Manual de ZWF

ZERFREX™ Web Framework, en adelante ZWF, es un framework muy ligero escrito en PHP. Proporciona los servicios básicos de enrutamiento y vistas y una estructura de aplicación.

En esta Guía del Usuario se recoge la información básica sobre la instalación, configuración y uso de ZWF.

1 Instalación

Requisitos

Los siguientes requisitos definen el entorno donde el funcionamiento de ZWF está probado.

- Sistema operativo CentOS 6, Debian 7, Debian 8, Ubuntu LTS 12.04, Fedora 19/20.
- Servidor Apache 2 (`apache2` en caso de Debian/Ubuntu; `httpd` en caso de CentOS/Fedora).
- PHP 5.3 instalado como módulo de Apache.
- Extensión de PHP: `pgsql` si se va usar las funciones de ZWF de acceso a BD con este motor.
- Extensión de PHP: `mysqli` si se va a usar las funciones de ZWF de acceso a BD con este motor.
- Extensión de PHP: `mbstring`.
- Opcional: base de datos PostgreSQL 8.4 o superior o bien MySQL 5.1 o superior.

Incompatibilidades

En algunos casos el módulo `mod_negotiation`, en concreto la funcionalidad `MultiViews`, puede provocar problemas si ZWF va a coexistir con otros ficheros en el mismo directorio y esos ficheros tienen el mismo nombre que algún controlador de ZWF. En este caso puede ser necesario añadir

```
Options -MultiViews
```

al fichero de configuración de Apache en la sección `<VirtualHost>` o análoga correspondiente donde vaya a estar alojado ZWF.

Instalación

Descomprimir y copiar en su caso **el contenido del directorio src** del paquete ZWF en el directorio raíz del sitio web donde se va a ejecutar la aplicación. Este directorio suele ser el establecido por `DocumentRoot` en la configuración de Apache, pero también puede ser un subdirectorio si es que la aplicación se alojará allí.

ZWF no se ejecuta desde ninguna subcarpeta. Si al descomprimirlo se creó alguna carpeta (muchos descompresores actúan así), es el contenido de dicha carpeta el que debe ser copiado al directorio raíz del sitio web.

El resultado final será que tanto `index.php` como `.htaccess` quedarán en el directorio raíz y, a ese mismo nivel, el resto de directorios.

El fichero `.htaccess` es un fichero oculto. A la hora de copiarlo al directorio raíz del sitio web, podría ser omitido por el programa que hace la copia (clientes FTP, SFTP, exploradores de archivos, etc). Hay que asegurarse de que se copia correctamente.

Configuración de Apache

PHP

El módulo PHP para Apache es necesario que esté instalado y activo. Se requiere la versión 5.3 ó superior. El módulo debe estar compilado, o bien el paquete instalado, con la extensión **mbstring** (Multi-Byte String) y, si se va a usar PostgreSQL usando los módulos de acceso a datos de ZWF, la extensión **pgsql**. Análogamente, en caso de usar MySQL, instalar la extensión **mysqli**.

Rewrite

ZWF usa directivas del módulo `mod_rewrite` de Apache. Están definidas en el fichero `.htaccess`. Por lo tanto es necesario que `mod_rewrite` esté instalado y activo en Apache. En la sección `<VirtualHost>` o análoga de Apache es necesario al menos la siguiente directiva:

```
AllowOverride FileInfo
```

También funcionará:

```
AllowOverride All
```

Esto es importante, porque a veces se tiene lo siguiente:

```
AllowOverride None
```

y en tal caso las directivas de `mod_rewrite` que hay en `.htaccess` no funcionarán y, por lo tanto, ZWF tampoco.

Permisos de los ficheros

No se requieren permisos especiales. Como toda aplicación web, puede ser conveniente:

- Crear un directorio con permisos de lectura y escritura para guardar logs, ficheros subidos, etc.
- Los ficheros de configuración pueden tener contraseñas. En el caso de ZWF, estos ficheros se encuentran en el directorio `cfg/`. Estos ficheros deberían tener permisos únicamente de lectura y ser accesibles solo por el usuario (y grupo, dependiendo de la configuración que usemos) de Apache.

Problemas habituales

Si se encuentra un error *file not found* o *index not found*, asegúrese de que el fichero `.htaccess` existe y es el que venía con el paquete ZWF.

Si el error es *access denied*, compruebe los permisos y propietarios de los ficheros. La instalación de Apache en Linux suele contener algún fichero de muestra `index.html` con los permisos y propietarios correctos.

El módulo de Apache `mod_rewrite` debe estar activo. Normalmente al activarlo es necesario reiniciar el servidor (`apache2` o `httpd`).

2 Actualización

Si se quiere actualizar a una versión superior de ZWF, solo hay que descomprimir y copiar el contenido del directorio src del paquete de distribución de ZWF encima de nuestra instalación la nueva versión, sobreescribiendo los ficheros.

La distribución de ZWF no interfiere con una instalación existente salvo que, obviamente, se hayan modificado los directorios y ubicaciones predeterminadas.

3 Estructura

La función de los ficheros y directorios instalados con ZWF es la siguiente:

- `base/` Contiene las clases del framework. No es necesario modificarlo.
- `cfg/` Directorio donde colocar los ficheros de configuración.
- `controllers/` Directorio donde colocar los controladores de la aplicación.
- `lib/` Directorio de libre disposición.
- `models/` Directorio donde colocar el modelo de datos de la aplicación.
- `res/` Directorio donde colocar JS, CSS, imágenes y otros recursos públicos. Accesible desde fuera.
- `views/` Directorio donde colocar las vistas de la aplicación.
- `.htaccess` Directivas de Apache 2
- `index.php` Controlador frontal

Módulos y espacios de nombres

Las diferentes clases y ficheros de configuración de ZWF están agrupados por módulos. **Un módulo, pues, es un conjunto de clases opcionalmente unido a un fichero de configuración**.

ZWF en su instalación predeterminada tiene tres módulos:

- El módulo **core** contiene las clases básicas de ZWF.
- El módulo **dev** contiene las clases de ayuda al desarrollo.
- El módulo **data-access** contiene las clases de acceso a la base de datos.

Los módulos, a su vez, siempre se agrupan por **espacios de nombres**. El único espacio de nombres que se usa en ZWF es `zfx` y los tres módulos suministrados están bajo dicho espacio.

El directorio `base/` contiene tantos subdirectorios como espacios de nombres haya disponibles. En la instalación predeterminada solo encontraremos `base/zfx/`.

En el interior de cada directorio correspodiente a un espacio de nombres hay tantos subdirectorios como módulos haya definidos. En la instalación predeterminada de ZWF, dentro de `base/zfx/`, encontraremos los directorios correspondientes a los tres módulos disponibles:

```
base/zfx/core/
base/zfx/dev/
base/zfx/data-access/
```

Todas las clases de un módulo están definidas bajo el espacio de nombres al que pertenece dicho módulo. Cuando vayamos a invocar una clase de ZWF, en la mayoría de los casos, será necesario anteponer el espacio de nombres o usar `use`.

Es posible añadir nuevos espacios de nombres y módulos en función de las necesidades del proyecto.

Los módulos son fácimente desactivables. Un módulo desactivado es ignorado por el sistema y no se carga su configuración ni se incluye en el sistema de autocarga de clases. El módulo **core** no se puede desactivar y su carga es obligatoria.

4 Configuración

La configuración se realiza creando ficheros de código PHP en el directorio `cfg/` con código similar a:

```
$cfg['opcion1'] = 'valor1';
$cfg['opcion2'] = 'valor2';
$cfg['opcion3'] = 'valor3';
// ...
$cfg['opcionN'] = 'valorN';
```

Siendo `opcion1`, `opcion2`, ... claves de configuración y `valor1`, `valor2`, ... sus valores. El nombre del array `cfg` es necesario.

Mediante este sistema se puede cambiar la configuración predeterminada (como se verá más adelante) o crear configuración propia según las necesidades de nuestro proyecto.

Por ejemplo: definiremos, para una supuesta aplicación, la clave de configuración `defaultEmailAddr`:

```
$cfg['defaultEmailAddr'] = 'webmaster@mydomain.com';
```

En cualquier punto de nuestra aplicación podemos recuperar esta clave usando la función `\zfx\Config::get()`. Ejemplo:

```
function contactFormSent($text)
{
    //...
    $to = \zfx\Config::get('defaultEmailAddr');
    writeMail($to, $text);
    // ...
}
```

Al ser los ficheros de configuración ficheros de código PHP se puede elaborar esquemas de configuración sofisticados. Por ejemplo:

```
$cfg['max_usuarios'] = 100;
$cfg['max_grupos'] = $cfg['max_usuarios'] / 2;
if ($cfg['max_grupos'] > 25) $cfg['max_grupos'] = 25;
```

Configuración predeterminada

Cada **módulo** de ZWF puede tener un fichero de configuración. Es lo que se llama **la configuración predeterminada del módulo**. Estos ficheros no están en `cfg/`, sino en el directorio de cada módulo.

El fichero de configuración del módulo **core** es `base/zfx/core/core-config.php`.

El fichero de configuración de **cualquier otro módulo siempre se llama** `module-config.php` y *está en su directorio correspondiente*. Todos los ficheros de configuración predeterminados de los módulos considerados activos son cargados al inicio.

Cambiar la configuración. Orden de carga

Para configurar la aplicación, sobreescribiendo los valores predeterminados, es necesario colocar en el directorio `cfg/` ficheros de configuración con los nuevos valores. Sin embargo, es necesario conocer el orden de carga y los nombres de fichero a utilizar para tener un control preciso del sistema.

En cualquier petición, la configuración es cargada en este orden:

1. Se lee la configuración predeterminada del módulo core, o sea, `base/zfx/core/core-config.php`.
2. Se lee la configuración personalizada por el usuario para el módulo core, si es que existe, o sea: `cfg/core-config.php`

   En dicho fichero tenemos la oportunidad de activar/desactivar módulos y también de especificar *una lista personalizada de ficheros de configuración.*
3. Se lee secuencialmente la configuración de los módulos activos. Por defecto todos los módulos (dev, data-access) están activos, así que se leerían los siguientes ficheros:

base/zfx/dev/module-config.php
base/zfx/data-access/module-config.php

4. Si se especificó la lista personalizada de ficheros de configuración en el paso 2, entonces se cargarán secuencialmente. Podemos aprovechar en este punto para sobreescribir la configuración predeterminada de los módulos que se cargó en el paso 3.

Esquema de configuración habitual

El esquema más sencillo pero completamente funcional es establecer dos ficheros de configuración: `cfg/core-config.php`, ya que siempre se intenta cargar este fichero, y otro (que llamaremos `cfg/my-config.php`) para configurar el resto de módulos (por ejemplo el acceso a la base de datos) y cualquier necesidad de nuestra aplicación.

El fichero `cfg/core-config.php` debería tener al menos el siguiente contenido:

```
$cfg['rootUrl']        = 'http://www.miaplicacion.com/';
$cfg['showErrors']     = false;
$cfg['autoLoadConfig'] = array('my-config');
```

La clave `rootUrl`

ZWF necesita saber su propia URL. Se especifica en la clave `rootUrl`.

Por ejemplo, supongamos que hemos instalado ZWF en el directorio `/var/www/test` de nuestro servidor y es accesible en la URL `http://www.example.com/test`.

Entonces en el fichero `cfg/core-config.php` introduciremos lo siguiente:

```
$cfg['rootUrl'] = 'http://www.example.com/test/';
```

Nótese la barra al final de la URL. **Por convenio, todas las URLs que se configuren en ZWF terminan con la barra /.**

También se puede especificar una ruta relativa; los navegadores web la suelen interpretar correctamente:

```
$cfg['rootUrl'] = '/test/';
```

Supongamos que tenemos una intranet y queremos acceder por una determinada IP y puerto:

```
$cfg['rootUrl'] = 'http://192.168.1.23:8080/testapp/';
```

Nótese la barra siempre al final de la URL.

La clave `showErrors`

Indica si se deben mostrar, o no, los errores de PHP. A menudo se combina con la detección automática del directorio o dirección del servidor para saber si estamos en producción o no. Ejemplo:

```
if ($_SERVER['DOCUMENT_ROOT'] == '/var/www/html')
{
    // Estamos en producción
    $cfg['rootUrl']    = 'http://www.miaplicacion.com/';
    $cfg['showErrors'] = false;
}
else if ($_SERVER['DOCUMENT_ROOT'] == '/var/www/test')
{
    // Estamos en desarrollo
    $cfg['rootUrl']    = 'http://dev.miempresa.com/';
    $cfg['showErrors'] = true;
}
```

La clave `autoLoadConfig`

Esta clave permite la carga de ficheros adicionales de configuración. En nuestro ejemplo contiene un único elemento cuyo valor es `my-config`.

5 Flujo de una petición. Controladores

ZWF implementa el patrón llamado «Controlador Frontal» (Front Controller). En ZWF, todas las peticiones son canalizadas mediante el fichero `/index.php`.

Por otro lado se fomenta el uso del patrón MVC (Model-View-Controller) en su aplicación más sencilla, que es una implementación web. Para ello facilita la creación y separación de:

- El modelo de datos en un conjunto separado de clases.
- Vistas gestionadas y encapsuladas.
- Biblioteca de clases adicionales en tierra de nadie.
- Controladores que canalizan el flujo, aglutinan y comunican todo lo anterior.

Sin embargo, todo esto es opcional, salvo la creación de, al menos, un **controlador**. Un controlador **es una clase derivada** de la clase `\zfx\Controller` (proporcionada por ZWF y que se encuentra en el fichero `base/zfx/Controller.php`) y que se almacena en el directorio `controllers/`.

Autocarga de clases

En una aplicación que use ZWF no es necesario hacer `include(...)` de ficheros `.php` para cargar clases.

Al usar una clase definida por ZWF (que se encuentran en el directorio `base/zfx/`), así como los modelos y clases de biblioteca (ubicadas en `models/` y `lib/` respectivamente), se busca el fichero **cuyo nombre coincide con el nombre de la clase** (y extensión `.php`) y se hace `include()` de dicho fichero.

La excepción a esto son los controladores. El directorio `controllers/` no está incluído en la autocarga de clases y para incluir un fichero desde este directorio sí es necesario usar `include()` o similar.

¿Para qué sirve el controlador?

Supondremos que la aplicación va a tener la siguiente dirección raíz:

`http://www.example.com`

Cuando visitamos la dirección:

`http://www.example.com/clientes`

El framework va a ser buscar el controlador llamado `Clientes` que esperará encontrar dentro del fichero `controllers/Clientes.php`. Si no lo encuentra, mostrará un mensaje de error. Pero si lo encuentra, **cargará el fichero y creará una instancia de esa clase**.

Si se visita la dirección raíz, o sea:

`http://www.example.com`

Lo que se buscará es el controlador predeterminado, que por defecto se llama `Index`.

De forma predeterminada se ha establecido que, una vez que se instancia la clase controlador, **si en la clase controlador se encuentran las funciones públicas `_init()` y `_main()`, se ejecutarán**. Por lo tanto, si se visita

`http://www.example.com`

Ocurrirá lo siguiente:

1. Se busca `controllers/Index.php`
2. Se instancia la clase `Index`.
3. Se ejecuta la función `_init()` de `Index`.
4. Se ejecuta la función `_main()` de `Index`.

Y si se visita

`http://www.example.com/clientes`

Ocurrirá lo siguiente:

1. Se busca `controllers/Clientes.php`
2. Se instancia la clase `Clientes`.
3. Se ejecuta la función `_init()` de `Clientes`.
4. Se ejecuta la función `_main()` de `Clientes`.

Proporcionar las funciones `_init()` y `_main()` es una forma de disponer de funciones que se sabe que siempre se van a ejecutar. De este modo podemos preservar el constructor de la clase para su inicialización y los mecanismos de herencia en caso necesario.

Reglas de creación de un controlador

Como se ha dicho, al menos hace falta un controlador en la aplicación. Salvo que se haya configurado otra cosa, su nombre debe ser `Index` y su implementación debería ser similar a

```
class Ctrl_Index extends \zfx\Controller
{
    //...
}
```

El nombre de los ficheros deben coincidir con el de la clase exactamente. Por ejemplo la clase `Clientes` debe ubicarse en el fichero `Clientes.php` y la clase `Ctrl_ListaProductos` debe ir en el fichero `Ctrl_ListaProductos.php`.

Los controladores derivan de la clase `Controller`.

Los controladores **deben nombrarse** según las siguientes reglas:

- Comenzar con un prefijo. De forma predeterminada es `Ctrl_`.
- A continuación, una letra mayúscula (ejemplo: `Ctrl_Index`).
- Seguir con letras o números, pero ningún otro símbolo (especialmente el signo de subrayado: **no** se puede usar). Ejemplo: `Ctrl_I18n`.
- Admite mayúsculas en el interior, lo que permite usar la «notación de camello» para nombres compuestos (ejemplo: `Ctrl_ExportadorProductos`).

Nombres de controladores inválidos:

- `Index`. No comienza por el prefijo `Ctrl_`.
- `Ctrl_clientes`. No comienza por mayúsculas tras el prefijo `Ctrl_`.
- `Ctrl_Lista_productos`. Contiene signo de subrayado.

La relación entre URL y el nombre del controlador

Básicamente la forma de acceder a la aplicación que usa ZWF es la siguiente:

`rooturl/controlador/segmento1/segmento2/.../segmentoN`

Siendo:

- `rooturl`: la dirección raíz (ejemplo `https://www.example.com`)
- `controlador`: el nombre en minúsculas del controlador;
- `segmento1`, `segmento2`,... Datos que se le pasan a la aplicación.

En la URL, el nombre del controlador siempre es en minúsculas. Las mayúsculas están prohibidas.

Si el controlador es un nombre en notación de camello, en la URL cada palabra debe separarse por un guión.

Ejemplo: la dirección

`http://www.example.com/lista-productos`

Invocará al controlador `Ctrl_ListaProductos`.

Lo que sigue al controlador:

`segmento1/segmento2/.../segmentoN`

Se denominan **segmentos**. Son datos que se le pasan al controlador. Más adelante se verá cómo leerlos y usarlos.

Por lo tanto, y en resumen:

- Después de la URL raíz va el nombre del controlador. No simboliza ningún subdirectorio.

- Después del nombre del controlador siguen los segmentos. Tampoco simbolizan subdirectorios.
- Si el controlador es `Index`, no es necesario escribirlo, salvo que se configure lo contrario.

Hay dos consecuencias.

- **Consecuencia 1.** Como el directorio **res/** es accesible desde el exterior (pues se espera que contenga las imágenes, CSS, javascript y demás), no puede existir un controlador con el nombre de clase `Ctrl_Res`.
- **Consecuencia 2.** El controlador `Index` no puede recibir datos (segmentos) porque no hay forma de distinguir si el primer segmento es en realidad un nombre de controlador:

Ejemplo:

`http://www.example.com/dato1/dato2/dato3`

`dato1` ¿Es un controlador?

Pero sí puede recibir datos si se especifica su nombre explícitamente:

`http://www.example.com/index/dato1/dato2/dato3`

¿Por qué toda esta rigidez?

En realidad esta rigidez no es tal, pues se puede cambiar el comportamiento de ZWF; sin embargo ésta es la configuración predeterminada.

Lo que se persigue es facilitar el uso de URL fácilmente escribibles, recordables y que no induzcan a error; que funcionen bien con robots, buscadores e indizadores; que no permitan la introducción de datos que provoquen problemas de seguridad, inyecciones o errores del sistema. Ejemplos:

`https://www.mitienda.com/linea-marron/televisores/philips`
`https://www.miapp.com/clientes/editar/id/3209`
`https://www.miapp.com/listados/productos-y-servicios`

Son más fáciles de entender y de indizar que

`https://www.mitienda.com/listar?cat=34&subcat=4&brand=PHILIPS`
`https://www.miapp.com/CRUD/action_edit?id=3209`
`https://www.miapp.com/list?type=2`

Nota: El sistema permite el uso de datos GET de forma transparente, aunque no se recomienda:

`https://www.example.com/productos?id=32`

Prestaciones comunes de los controladores

Un controlador tiene funciones para recuperar datos de la URL o devolver su propia URL, entre otras muchas. En las siguientes secciones se usará `$this->` para enfatizar que se usa desde el propio controlador.

Los segmentos del URL

En un controlador, invocar `$this->_segment()` devuelve un array con todos los segmentos que se pasaron en la URL.

Ejemplo: Si se visita

`http://www.example.com/clientes/listar/32`

Entonces llamar a `$this->_segment()` devuelve el array

```
(
    0 => 'listar',
    1 => 32
)
```

Si se llama a `_segment()` con un parámetro numérico, se obtiene solo ese segmento. Por ejemplo `$this->_segment(0)` en el ejemplo anterior devolvería la cadena `listar`.

Nótese cómo el segmento que corresponde al controlador no se incluye.

Ejecutar una función llamada como el primer segmento

En otros frameworks de forma predeterminada se ejecuta una función en el controlador que se llama como el primer segmento. Son las denominadas *acciones*.

En ZWF este comportamiento no ocurre, pero puede hacerse con la función `$this->_autoexec()`.

Supongamos el controlador Clientes con estas dos funciones:

```
public function _main()
{
    $this->_autoexec()
}

public function listar()
{
    // ...Aquí hacemos algo
}
```

Si visitamos la URL:

`http://www.example.com/clientes/listar/32`

Ocurrirá que `_autoexec()` leerá el primer segmento (`listar`), buscará una función con el mismo nombre y la ejecutará. Después, `_autoexec()` retornará `TRUE` indicando de este modo que la encontró.

Si no hubiera una función pública en la clase con ese nombre, retornaría `FALSE`. Esto se puede aprovechar para hacer *fallbacks*:

```
public function _main()
{
    if (!$this->_autoexec())
    {
        // Mostrar el índice
    }
}

public function _capitulo1()
{
    // Mostrar el capítulo 1
}
```

La función `_autoexec()` se ocupa de obtener los segmentos y pasárselos a las funciones a las que intenta llamar como parámetros. Esto es un proceso automático.

Los segmentos están limitados a los caracteres `[a-z0-9]` y el guión. Sin embargo, no se puede usar guiones para los nombres de funciones PHP. Si se va a usar `_autoexec()` debe tenerse en cuenta.

Generar URLs

El controlador sabe dónde está:

- La función `_urlController()` devuelve su propia URL, pero sin segmentos (o sea, solo devuelve hasta el controlador).
- La función `_urlPath()` devuelve su propia URL con los segmentos que en ese momento se recibieron.

Ejemplo:

Sea el controlador `Ctrl_Clientes` y visitamos la dirección

`https://www.example.com/clientes/borrar/56`

- La función `_urlController()` devuelve `https://www.example.com/clientes/`.
- La función `_urlPath()` devuelve `https://www.example.com/clientes/borrar/56`.

Redirecciones

En cualquier momento podemos emitir una cabecera de redirección:

```
$this->_redirect("http://www.example.com/asuntos");
```

Esta cabecera solo funcionará antes de emitir alguna salida al navegador.

Controladores personalizados

Es muy habitual y conveniente crear un controlador abstracto que herede de los controladores básicos proporcionados por ZWF y usarlo como base para los controladores reales de nuestra aplicación.

Este controlador abstracto proporcionará todo el código e inicialización que necesiten nuestros controladores, como por ejemplo creación de menús, carga de plantillas, autenticación, etc.

Por claridad, se recomienda usar un prefijo (como por ejemplo `Abs_`) para nombrarlos. Al no usar el prefijo `Ctrl_` el sistema no intentará servirlos, aunque lógicamente al ser controladores abstractos no pueden ser instanciados.

6 Vistas

Las vistas son ficheros PHP que se envían al cliente web. Antes de enviarlos, se hace una expansión de variables a partir de un array. Es así como se le pasan los datos a la vista.

Vistas directas

Los ficheros vista se ubican en `/views` y se pueden nombrar como se quiera.

Por ejemplo si tenemos una vista llamada `listado.php`, para mostrarla usaremos la siguiente función estática de la clase `\zfx\View`:

```
\zfx\View::direct("listado");
```

No se debe incluir la extensión `.php`.

Si tenemos una variable llamada `$lista` que queremos usar en la vista, se pasa formando parte de un array. La clave del array será el nombre de la variable resultante en la vista.

```
$datosVista = array('lista' => $lista);
\zfx\View::direct("listado", $datosVista);
```

En el ámbito de ejecución PHP del fichero vista tendremos la variable `$lista` con los mismos datos que tenía en el controlador.

Se puede establecer una jerarquía de directorios en el directorio `/views` para tener las vistas bien organizadas. Esto es útil si tenemos muchas vistas, diferentes secciones o bien diferentes temas. Ejemplo:

```
views
|
+----- zona-publica
|      |
|      +------- index.php
|      |
|      +------- listado.php
|
+----- zona-privada
|
+----- index.php
|
+----- listado.php
```

Para mostrar las vistas así organizadas llamamos:

```
\zfx\View::direct("zona-privada/listado");
```

Vistas anidadas

Es posible anidar vistas con el fin de crear plantillas que a su vez contienen vistas o bien otras plantillas (y anidarlas nuevamente). Las plantillas (o vistas) anidadas son muy importantes porque permiten la fácil estructuración de las vistas de nuestra aplicación y mejoran la reutilización de código.

Supongamos el caso de que tenemos una plantilla de página genérica que carga todo el javascript y estilo que precisamos y solo cambia el contenido dentro del elemento BODY. La llamaremos `paginabase.php` y el contenido será similar a este:

```
<html>
<head><title>...</title><style>...</style><script>...</script>
<body>
<?php $this->section('cuerpo'); ?>
</body>
</html>
```

Sea `contenido1.php` una vista en nuestra aplicación que queremos cargar justo en el lugar donde hemos ubicado `<?php $this->section('bloque'); ?>`.

Supongamos que dicha vista contiene por ejemplo:

```
<p>Esto es un ejemplo de párrafo</p>
```

Necesitaremos hacer (en el controlador):

```
$v = new \\zfx\\View('paginabase.php');
$v->addSection('cuerpo', 'contenido1.php');
$v->show();
```

El resultado será:

```
<html>
<head><title>...</title><style>...</style><script>...</script>
<body>
<p>Esto es un ejemplo de párrafo</p>
</body>
</html>
```

Nótese cómo hemos instanciado un objeto de la clase `\zfx\View` en lugar de invocar una función estática como se hace con las vistas directas. Para construir y mostrar la vista se usa la función `show()`.

Expansión de datos en vistas anidadas

Tanto en el momento de añadir una sección con addSection() como al mostrar todo con show() se puede especificar listas de variables a expandir:

```
$datosSeccion = array(...);
$datosVista = array(...);
$v = new View('paginabase.php');
$v->addSection('cuerpo', 'contenido1.php', $datosSeccion);
$v->show($datosVista);
```

Anidamiento múltiple

Este mecanismo admite anidamiento de secciones. Supongamos que la vista `contenido2.php` tiene el siguiente código:

```
<div>Esto es un div... <?php $this->section('bloque-interno'); ?></div>
```

Entonces, al ejecutar:

```
$v = new View('paginabase.php');
$v->addSection('cuerpo', 'contenido2.php');
$v->addSection('bloque-interno', 'contenido1.php');
$v->show();
```

El resultado será:

```
<html>
<head><title>...</title><style>...</style><script>...</script>
<body>
<div>Esto es un div... <p>Esto es un ejemplo de párrafo</p></div>
</body>
</html>
```

Encolamiento de secciones

Una característica muy importante del mecanismo de vistas es el encolamiento de secciones. Consiste en añadir a una misma sección múltiples vistas, que serán mostradas secuencialmente.

Supongamos el ejemplo `paginabase.php` anterior y las siguientes vistas `bloque1.php`, `bloque2.php`, `bloque3.php`:

bloque1.php:

```
<div>Esto es el bloque 1.</div>
```

bloque2.php:

```
<div>Esto es el bloque 2.</div>
```

bloque3.php:

```
<div>Esto es el bloque 3.</div>
```

Añadamos todos los bloques a la misma sección:

```
$v = new View('paginabase.php');
$v->addSection('cuerpo', 'bloque1.php');
$v->addSection('cuerpo', 'bloque2.php');
$v->addSection('cuerpo', 'bloque3.php');
$v->show();
```

El resultado será:

```
<html>
<head><title>...</title><style>...</style><script>...</script>
<body>
<div>Esto es el bloque 1.</div>
<div>Esto es el bloque 2.</div>
<div>Esto es el bloque 3.</div>
</body>
</html>
```

El encolamiento de secciones es muy útil, por ejemplo, para añadir múltiples ficheros JS y CSS a una plantilla.

7 Modelo de datos

Se proporciona (y se recomienda su uso) el directorio `models/` para almacenar aquí las clases PHP correspondiente al modelo de datos de nuestra aplicación.

No hay ninguna norma o restricción respecto a esto, ya que cada aplicación requiere un modelo y patrones diferentes.

Se proporcionan clases de conexión y consulta a base de datos que hacen cómodo el trabajo con la BD, pero son completamente opcionales; puede usarse directamente el conjunto de funciones de PHP para MySQL, SQLite, PostgreSQL, etc. tal cual y por supuesto bibliotecas de terceros como RedBeans.

No se proporciona un ORM pues en el mercado hay diferentes soluciones PHP listas para usar.

Acceso a bases de datos

Configuración

En ZWF hay soporte para MySQL y PostgreSQL. Se representan por las cadenas `my` y `pg` respectivamente.

El sistema elegido se almacena en la clave de configuración `dbSys`. Por defecto es `pg`. Por ejemplo para cambiarlo a MySQL:

```
$cfg['dbSys'] = 'my';
```

Esta clave determina, en el sistema de autocarga de clases, si se va a buscar un fichero en el subdirectorio `pg/` o `my/` en cualquier módulo y también en el directorio de modelos (`models/`).

Los parámetros de conexión se almacenan en un array anidado. La conexión preterminada se denomina `default`, pero se pueden establecer tantas conexiones como se necesiten. En caso de MySQL será:

```
$cfg['my'] = array(
    'default' => array(
        'dbHost' => 'hostname',
        'dbUser' => 'username',
        'dbPass' => 'password',
        'dbDatabase' => 'databasename',
        'dbPort' => '3306'
    )
);
```

Análogamente para PostgreSQL será (en este ejemplo se usará dos perfiles):

```
$cfg['pg'] = array(
    'default' => array(
        'dbHost' => 'hostname',
        'dbUser' => 'username',
        'dbPass' => 'password',
        'dbDatabase' => 'databasename',
        'dbPort' => '5432'
    ),
    'production' => array(
        'dbHost' => 'hostname2',
        'dbUser' => 'username2',
        'dbPass' => 'password2',
        'dbDatabase' => 'databasename2',
        'dbPort' => '5432'
    )
);
```

Si en lugar del perfil `default` quisiéramos usar otro de forma predeterminada (por ejemplo production), estableceremos esta clave:

```
$cfg['dbProfile'] = 'production';
```

Conexion

La conexión a la base de datos se realiza instanciando un objeto de la clase `MyDB` o `PgDB` dependiendo del motor que se quiera usar.

También puede usarse la clase `DB` para usar el motor predeterminado.

```
$db = new \zfx\DB();
```

Se puede especificar una conexión diferente a la predeterminada para esa instancia concreta:

```
$db = new \zfx\DB('production');
```

Consultas básicas

Una vez que está configurado el acceso a la BD, y hecha la conexion, se puede acceder directamente sin usar otras librerías ni modelos. Ejemplo:

```
$bd->q("UPDATE clientes SET deuda = 0");
```

Las clases `PgDB` y `MyDB` tienen funciones similares para realizar consultas. Estas son las tareas más habituales para realizar con un objeto de la clase `PgBD` o `MyDB`:

- La función `escape()` prepara una cadena para ser usada dentro de una consulta y prevenir inyecciones de código.
- La función `q()` ejecuta una consulta y no devuelve datos.
- La función `qr()` ejecuta una consulta y devuelve la primera (o única) fila como un array asociativo.
- La función `qa()` ejecuta una consulta y devuelve todos los resultados como un array de arrays asociativos.
- La función `qs()` ejecuta una consulta pero no devuelve datos. Para obtener la siguiente fila (como un array asociativo) se usa la función `next()`.

Ejemplo:

```
$bd = new PgBD();
$bd->qs("SELECT nombre FROM clientes");
$fila = $bd->next();
while ($fila)
{
    print_r($fila);
    $fila = $bd->next();
}
```

8 Herramientas de desarrollo

Registros

La clase `Log` permite controlar la ejecución del programa registrando información en ficheros de texto similares a los generados en `/var/log` en un sistema UNIX.

La función estática `Log::log()` escribe una cadena en el fichero log especificado acompañada de la fecha y hora.

El parámetro de configuración `logPath` establece la ruta donde los ficheros log se escriben. Por defecto `logPath` está establecido a `/tmp/`. Lo habitual es habilitar un directorio con los permisos adecuados.

Inspección de datos

En muchas ocasiones no se tiene acceso a un sistema completo de depuración, pero es necesario mostrar el contenido de variables durante alguna prueba y asegurar que se visualiza correctamente el contenido sin verse afectado por capas como CSS.

`\zfx\Debug::show()` muestra un dato de cualquier tipo de forma legible para un humano.

La clase Debug ofrece otras funciones estáticas para mostrar información de depuración:

- `\zfx\Debug::analyze()` muestra un dato de cualquier tipo de forma legible para un humano, presentando información muy detallada. El uso típico de ambas funciones es visualizar el contenido de variables de forma destacada.
- `\zfx\Debug::devError()` muestra un dato de cualquier tipo de forma legible para un humano, lo registra en el LOG error y finaliza la ejecución. Su uso está indicado para mostrar errores que no deberían ocurrir nunca (errores de desarrollo).

Niveles de depuración

Mediante el parámetro de configuración debugLevel se puede establecer de forma global el nivel de depuración numérico en el que se encuentra la aplicación.

Por convenio el **nivel 0** significa la desactivación del sistema de depuración. El **nivel N** mostrará toda la información de depuración cuyo nivel M sea 1 <= M <= N.

Las funciones `Debug::show()` y `Debug::analyze()` son las que se usan para mostrar información de cierto nivel (pasándoselo como parámetro) en algún punto de la aplicación.

Si la opción `debugLog` es `true`, la información de depuración para los diferentes niveles se almacenará en el fichero log `debugN` siendo N el nivel de depuración.

Además, si la opción `debugShow` es `true`, la información de depuración para los diferentes niveles se mostrará en la salida de la aplicación.

Cronómetros

La clase `Profiler` proporciona un medio de cronometrar una parte de la aplicación y almacenar los resultados en un fichero LOG.

El uso típico de la clase es:

1. Instanciar un objeto. En el constructor se le pasa el nombre del fichero log donde se almacenarán los resultados (por defecto es `profile`).
2. Ejecutar `start()` justo antes de comenzar la parte de la aplicación que se va a cronometrar.
3. Ejecutar `stop()` al final o en cada punto intermedio donde se desea almacenar la duración del intervalo que ha transcurrido desde que se hizo `start()`.

9 CSS, JavaScript y otros recursos

El directorio `res/` está configurado en `.htaccess` para ser accesible públicamente. Se recomienda que el CSS, JavaScript, imágenes, vídeos, etc. de la aplicación se coloquen bajo dicho directorio.

10 Biblioteca de clases

A continuación se describen algunas clases suministradas con ZWF y su función. La funcionalidad se ha reducido al mínimo, siguiendo el principio de implementar sólo aquello que realmente se necesita.

La biblioteca de ZWF está documentada siguiendo el convenio PHPDocumentor y por lo tanto se puede extraer la documentación del API mediante este método.

Módulo core

- La clase `HtmlTools` contiene métodos estáticos que facilitan la creación de elementos HTML como tablas o `<select>`
- La clase `Paginator` gestiona el control y generado de código HTML para dibujar un paginador. Es altamente personalizable.
- La clase `StrFilter` contiene numersos métodos de ayuda para operar con cadenas UTF-8. **Es muy recomendable su uso.**
- La clase `StrValidator` contiene métodos de validación de tipos de datos comunes. Algunos son un stub, como el del email, que es elemental y necesitaría ser desarrollado en profundidad para cumplir con el RFC.
- La clases `Mat` y `Sys` son stubs con la intención de ser desarrolladas en un futuro. Actualmente contienen un método estático para el redondeo de euros a dos decimales y un método para obtener una ruta local respectivamente.

Módulo data-access

- La clase `DataTools` contiene herramientas útiles para el trabajo con la BD.

11 Biblioteca de terceros

Además de los controladores y del modelo de datos, la aplicaciones normalmente hacen uso de otras clases. Se recomienda usar el directorio `lib/` para alojar dichas clases.

Dicho directorio está bajo el sistema de carga automática de clases. Para que funcione la clase y el fichero deben tener el mismo nombre.