Lógica de negocio y presentación © ADR Infor SL

fundacionuniriioja.adrformacion.com © ADR Infor SL
Héctor García González
fundacionuniriioja.adrformacion.com © ADR Infor SL
Héctor García González

Indice

Lógica de negocio y presentación **3**
- Separando la lógica de negocio de la presentación 5
  - Segmentar las plantillas 8
  - Repetir secciones de código 10
  - Trabajar con idiomas 12
- Motor de plantillas predesarrollado: Smarty 15
  - Descargar y configurar Smarty 16
  - Utilizar las plantillas 17
  - Segmentar las plantillas 21
  - Repetir secciones de código 22
  - Trabajar con idiomas 27
  - Funcionalidades adicionales del código Smarty 29
- Hemos aprendido 33

Ejercicios **35**
- Ejercicio: Separar lógica de negocio y presentación de una aplicación web 35
  - Lo necesario para comenzar 35
  - Pasos a seguir 35
  - Solución 35

Recursos **37**
- Enlaces de Interés 37
- Preguntas Frecuentes 37
- Glosario. 37

Lógica de negocio y presentación

Al finalizar esta unidad el alumno será capaz de separar por completo la lógica de negocio de la presentación en sus aplicaciones web, utilizando tanto soluciones propias como motores de plantillas ya existentes.

Hasta ahora, en nuestros scripts de PHP **hemos mezclado código PHP con HTML**. Cuando comenzamos a desarrollar aplicaciones web, ésta nos suele parece la forma más sencilla y comprensible de programar, pero a medida que vayamos adquiriendo experiencia, nos daremos cuenta de que **ésta mezcla de códigos es un problema**.

En toda aplicación web, existen dos capas muy bien diferenciadas:

Lógica de negocio

Es la parte de la aplicación web encargada del manejo de la información y la realización de los procesos.

En reglas generales, las acciones realizadas por instrucciones de **PHP** conformarán esta capa.

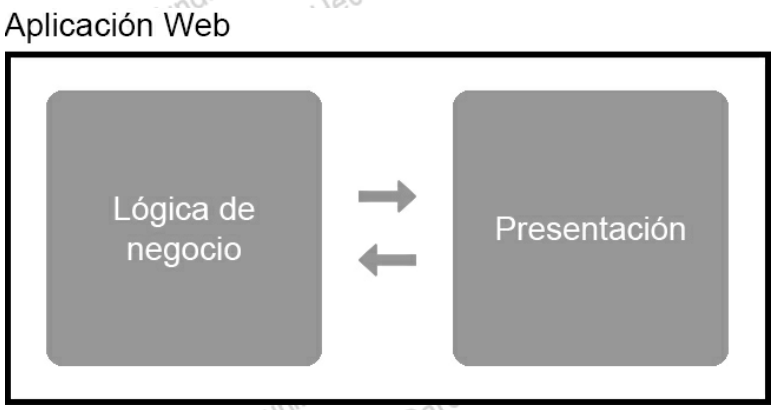
Presentación

Es la parte de la aplicación web encargada de establecer cómo se deben mostrar los datos al usuario final.

Normalmente está compuesta por código **HTML** que es completado con los datos suministrados por la lógica de negocio.

Cuando desarrollemos aplicaciones web, **debemos separar por completo la lógica de negocio de la presentación**, de forma que puedan ser desarrolladas de forma independiente la una de la otra.

El objetivo debe ser que podamos **reemplazar por completo cualquiera de las dos capas sin modificar en absoluto la otra**.



Las capa de lógica de negocio y de presentación **se comunicarán entre sí para componer el resultado final** Separar estas capas y establecer la comunicación entre ellas se convertirá en la base de la estructuración de nuestro código.

Esta forma de programar nos proporcionará gran cantidad de **ventajas**. Algunas de ellas son:

1

Si cambiamos el diseño de la aplicación, **no necesitamos reprogramar** nada relativo al funcionamiento de la misma.

2

Podemos **cambiar entre varios diseños** de forma rápida utilizando "skins" diferentes para nuestra web.

3

La **organización del trabajo** es más sencilla. Los programadores sólo necesitan desarrollar en la capa de lógica de negocio mientras que los diseñadores maquetarán el HTML en la de presentación.

4

Nos brinda la posibilidad de **reaprovechar diferentes partes de la lógica de negocio** que, aplicándoles una presentación diferente, puedan ser reutilizadas en otros proyectos.

Tenemos una sección de noticias **ya programada** para un sitio web.

Vamos a desarrollar un nuevo sitio web en el que también tendremos una sección de noticias que **tiene un funcionamiento idéntico**, lo que es muy habitual.

Si tenemos separada la lógica de negocio de la presentación, podemos simplemente **copiar el módulo encargado del funcionamiento** de la sección de noticias y aplicarle **una presentación diferente** para que encaje con el diseño del nuevo sitio web.

Separando la lógica de negocio de la presentación

Uno de los principales objetivos, a la hora de organizar el código de nuestra aplicación web, es **mantener la lógica de negocio completamente separada de la presentación**.

Para ello, separaremos ambas parte en archivos diferentes:

- Guardaremos el **código HTML** de nuestra aplicación web en un **archivo .html**, como si de una **página estática** se tratase.
- Mantendremos exclusivamente el **código PHP** en el **archivo .php**.

Vamos a ver un ejemplo:

Tenemos el siguiente código en cliente.php:

```
<?php
    $GLOBALS['conexion'] = mysqli_connect('localhost','root','','bbdd');
    $GLOBALS['conexion'] -> query('SET NAMES utf8');
?>
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo</title>
    <link href="estilos.css" rel="stylesheet" type="text/css">
</head>
<body>
    <div id="main">
<?php
    $consulta = 'SELECT * FROM clientes WHERE id = ' . (int)$_GET['id'];
    $resultado = $GLOBALS['conexion'] -> query($consulta);
    $cliente = $resultado -> fetch_array();
?>
        <table>
        <tr>
            <td>Nombre:</td>
            <td><?php echo htmlentities($cliente['nombre']); ?></td>
        </tr>
        <tr>
            <td>Apellidos:</td>
            <td><?php echo htmlentities($cliente['apellidos']); ?></td>
        </tr>
        </table>
    <div>
</body>
</html>
```

Si separamos la lógica de negocio de la presentación, tendremos el archivo clientes.html que tendrá este contenido:

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo</title>
    <link href="estilos.css" rel="stylesheet" type="text/css">
</head>
<body>
    <div id="main">
        <table>
        <tr>
            <td>Nombre:</td>
            <td>###nombre###</td>
        </tr>
        <tr>
            <td>Apellidos:</td>
            <td>###apellidos###</td>
        </tr>
        </table>
    <div>
</body>
</html>
```

Y el archivo clientes.php pasará a tener únicamente este contenido:

```
<?php
    $GLOBALS['conexion'] = mysqli_connect('localhost','root','','bbdd');
    $GLOBALS['conexion'] -> query('SET NAMES utf8');
    $consulta = 'SELECT * FROM clientes WHERE id = ' . (int)$_GET['id'];
    $resultado = $GLOBALS['conexion'] -> query($consulta);
    $cliente = $resultado -> fetch_array();
?>
```

Si te fijas en este ejemplo, podrás comprobar cómo el archivo**clientes.html**, es muy fácil de interpretar ya que sólo contiene el diseño y el archivo **clientes.php** también ya que sólo se ocupa de las acciones.

Manteniendo el código HTML aparte, estamos generando un**sistema de plantillas**, que pueden intercambiarse fácilmente y modificarse sin conocimientos de programación.

Estas plantillas tienen indicadas ciertas **variables**, expresadas con una nomenclatura especial, que serán **sustituidas por los valores generados por la capa de lógica de negocio**

En el ejemplo, hemos elegido la nomenclatura###nombre### para identificar estos espacios donde debemos sustituir valores, pero esto no es una norma.

Podemos elegir la sintaxis que nos resulte más clara en nuestras plantillas.

Una vez separados los códigos, necesitamos que nuestra aplicación web, sea capaz de conformar el resultado final.

Para ello programaremos un **motor de plantillas**.

El motor de plantillas será un fragmento de código que debe realizar las siguientes acciones:

1
Recuperar el código HTML contenido en la plantilla correspondiente.
2
Sustituir las variables del código HTML por sus valores correspondientes.
3
Mostrar el código HTML resultante.

La última acción que realizaremos siempre en nuestra capa de lógica de negocio es **llamar al motor de plantillas** para que genere y muestre el HTML final.

Anotación: Rentabilidad de programar un motor de plantillas

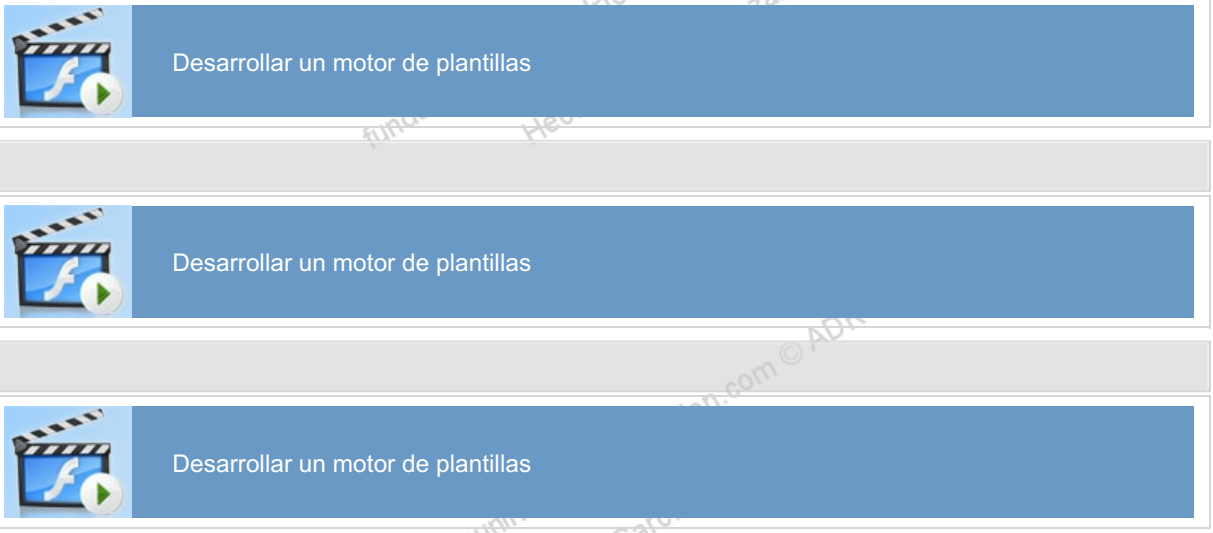
Aunque pueda parecer mucho trabajo inicialmente, piensa que una vez desarrollado el motor de plantillas, **podemos utilizarlo para la totalidad de nuestros desarrollos**.

Gracias a la sencillez de trabajo que nos proporciona separar la lógica de negocio de la presentación, merecerá la pena el tiempo invertido en su desarrollo desde el primer proyecto.

Para generar el motor de plantillas, solo tenemos que:

- Recuperar el código HTML de la plantilla con `file_get_contents`
- Ir realizando los reemplazos necesarios con `str_replace`.

Vamos a ver cómo hacerlo:



Segmentar las plantillas

Nuestro motor de plantillas, debe permitirnos trabajar de forma **cómoda y flexible**. El motivo de su existencia es organizar y simplificar nuestro código, por lo que tendremos que implementar las opciones necesarias para que nuestro trabajo con él sea **lógico y eficiente**.

En el momento en el que comenzamos a trabajar con plantillas, nos daremos cuenta de una realidad: Hay **porciones de código HTML que se repiten** en muchas de nuestra páginas. Repetir código, nunca es una buena idea, ya que cuando debemos realizar una modificación, tenemos que hacerlo en todos los lugares en los que se encuentra.

Lo adecuado es **no tener nunca código HTML duplicado** y utilizar la misma porción de código en todos los lugares en los que debamos mostrarla.

De esta forma, el mantenimiento es más simple, ya que **todas las modificaciones se realizan en un único lugar**.

Todas las páginas de nuestro sitio web, tienen **el mismo diseño de cabecera**.

No tiene sentido que el código HTML de la cabecera esté presente en **todas y cada una de las páginas del sitio**. Si tuviéramos que modificar su diseño, **deberíamos modificar todas ellas**, aumentando las posibilidad de error y de producir diseños diferentes.

Almacenando esta cabecera **en un único archivo**, tendremos que cambiar el código **en un único lugar**. De esta forma conseguimos una estructura mucho más robusta.

La solución a este problema es **segmentar las plantillas**. Generaremos nuevas plantillas con estas porciones de código que se repiten y, de igual forma que hacemos con las variables, idearemos una nomenclatura para especificar su posición.

Vamos a ver un ejemplo de cómo segmentar las plantillas:

Tenemos el archivo plantilla.html que contiene una cabecera que se repite en todas las páginas del sitio web:

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo</title>
    <link href="estilos.css" rel="stylesheet" type="text/css">
</head>
<body>
    <header>
        <h1>Mi página web</h1>
        <nav>
            <ul>
            <li><a href="index.php" title="Inicio">Inicio</a></li>
            <li><a href="noticias.php" title="Noticias">Noticias</a></li>
            <li><a href="contacto.php" title="Contacto">Contacto</a></li>
            </ul>
        </nav>
    </header>
    ...
    ...
    ...
    Contenido de la página
    ...
    ...
    ...
</body>
</html>
```

Generamos un nuevo archivo llamado cabecera.html que contiene exclusivamente el código de la cabecera:

```
<header>
    <h1>Mi página web</h1>
    <nav>
        <ul>
        <li><a href="index.php" title="Inicio">Inicio</a></li>
        <li><a href="noticias.php" title="Noticias">Noticias</a></li>
        <li><a href="contacto.php" title="Contacto">Contacto</a></li>
        </ul>
    </nav>
</header>
```

En el archivo plantilla.html indicaremos la posición en la que debemos introducir esta porción de código.

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo</title>
    <link href="estilos.css" rel="stylesheet" type="text/css">
</head>
<body>
    @@@cabecera.html@@@
    ...
    ...
    ...
    Contenido de la página
    ...
    ...
    ...
</body>
</html>
```

Como ves, hemos ideado una nueva nomenclatura para indicar la posición en la que deberá ser introducido el código de otra plantilla. Ahora debemos implementar en nuestro motor de plantillas la funcionalidad de **sustituir las subplantillas en los lugares indicados**.

Segmentar plantillas

Repetir secciones de código

A la hora de realizar el diseño de una página, nos encontraremos con que**habrá fragmentos de código HTML que se repetirán** en función de la información que deba mostrarse.

Si accedemos al catálogo de productos de nuestra tienda on-line, la página nos mostrará todos los productos disponibles.

Cada producto, **es un fragmento de HTML similar** al resto de productos, pero mostrando los datos concretos del producto.

Nuestra plantilla deberá mostrar **tantos bloques HTML de producto, como productos tengamos a la venta** en este momento.

Para que nuestro motor de plantillas sea plenamente funcional, debemos implementar en él un mecanismo para realizar repeticiones, que se asemeje a un **bucle**.

Tendremos que idear una nomenclatura que exprese estas repeticiones. Vamos a ver un ejemplo de cómo podríamos indicar un bucle en una plantilla:

Dentro de nuestra lógica de negocio, tenemos un array con la siguiente estructura:

```
$productos[0]['nombre'] = 'Kit de manicura';
$productos[0]['precio'] = 27.49;
$productos[1]['nombre'] = 'Shampoo 250 ml.';
$productos[1]['precio'] = 17.33;
$productos[2]['nombre'] = 'Eau de toilette vaporizador';
$productos[2]['precio'] = 49.99;
```

En la siguiente plantilla, indicaremos que los datos del array deben ser repetidos de este modo:

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo</title>
    <link href="estilos.css" rel="stylesheet" type="text/css">
</head>
<body>
    $$$solucion$$$
    <div class="producto">
        <div class="nombre">#$#nombre#$#</div>
        <div class="precio">#$#precio#$#</div>
    </div>
    $$$/solucion$$$
</body>
</html>
```

Tras ser procesado por el motor de plantillas, el resultado final es el siguiente:

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo</title>
    <link href="estilos.css" rel="stylesheet" type="text/css">
</head>
<body>
    <div class="producto">
        <div class="nombre">Kit de manicura</div>
        <div class="precio">27.49</div>
    </div>
    <div class="producto">
        <div class="nombre">Shampoo 250 ml.</div>
        <div class="precio">17.33</div>
    </div>
    <div class="producto">
        <div class="nombre">Eau de toilette vaporizador</div>
        <div class="precio">49.99</div>
    </div>
</body>
</html>
```

Puedes ver, que hemos ideado una nomenclatura para expresar **fragmentos de HTML que se repitan** en función del número de componentes de un array. También hemos establecido una nomenclatura para indicar dónde irán colocados los diferentes datos de cada componente del array.

Lo único que nos falta por hacer, es dotar de la funcionalidad necesaria a nuestro motor de plantillas, para que sea capaz de interpretar esta nomenclatura y componer el HTML final.

Repetir secciones de código

Trabajar con idiomas

Uno de las situaciones más frecuentes cuando desarrollamos un sitio web, es la necesidad de **mostrarlo en diferentes idiomas**. Cuando una web cambia de idioma, su diseño no varía, sólo cambian los textos.

Debemos dotar a nuestro motor de plantillas de la capacidad de **mostrar los textos en un idioma u otro, sin variar el diseño**. Para ello generaremos un **archivo de recursos de texto**, por cada uno de los idiomas en los que se mostrará nuestra página.

Este archivo de recursos de texto, contendrá todos los textos de la página, en el idioma correspondiente. **Cada texto, vendrá referenciado por un identificador**, que luego podremos colocar en el lugar deseado de nuestra plantilla.

Vamos a ver un ejemplo de cómo podríamos organizar esta estructura:

Tenemos un archivo de recursos de texto llamado textos_es.txt, con los textos en castellano:

```
ejemplo;Ejemplo
nombre;Nombre
precio;Precio
```

Igualmente, tenemos un archivo de recursos de texto llamado textos_en.txt, con los textos en inglés:

```
ejemplo;Example
nombre;Name
precio;Price
```

Nuestra plantilla contiene el siguiente código:

```
<!DOCTYPE html>
<html>
<head>
    <title>***ejemplo***</title>
    <link href="estilos.css" rel="stylesheet" type="text/css">
</head>
<body>
    <div class="producto">
        <div class="nombre">***nombre***: ###nombre###</div>
        <div class="precio">***precio***: ###precio###</div>
    </div>
</body>
</html>
```

Cuando la lógica de negocio indique al motor de plantillas que muestre los textos en castellano, mostrará lo siguiente:

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo</title>
    <link href="estilos.css" rel="stylesheet" type="text/css">
</head>
<body>
    <div class="producto">
        <div class="nombre">Nombre: Shampoo 250 ml.</div>
        <div class="precio">Precio: 17.33</div>
    </div>
</body>
</html>
```

Cuando la lógica de negocio indique al motor de plantillas que muestre los textos en inglés, mostrará lo siguiente:

```
<!DOCTYPE html>
<html>
<head>
    <title>Example</title>
    <link href="estilos.css" rel="stylesheet" type="text/css">
</head>
<body>
    <div class="producto">
        <div class="nombre">Name: Shampoo 250 ml.</div>
        <div class="precio">Price: 17.33</div>
    </div>
</body>
</html>
```

Organizar los idiomas en la web

La forma adecuada de estructurar los diferentes idiomas en una web, es tener **URLs diferentes** para cada uno de ellos.

Esto suele hacerse mediante subdominios:

- http://es.example.com/
- http://en.example.com/
- http://fr.example.com/

O directamente mediante directorios:

- http://www.example.com/es/
- http://www.example.com/en/
- http://www.example.com/fr/

Nunca hagas depender el idioma de la página de una variable de sesión o una cookie ya que mantener una única URL para distintos idiomas, provocará fallos de indexación de tu sitio en los motores de búsqueda penalizando el posicionamiento de tu web.

Puedes utilizar la **reescritura de URLs** para obtener el idioma en el que estás trabajando y no tener que repetir tu código por cada uno de los idiomas.

Hemos ideado una nueva nomenclatura para indicar recursos de texto dentro de nuestras plantillas. Ahora debemos implementar en el motor de plantillas las acciones necesarias para **sustituir estas nomenclaturas por sus valores, dependiendo del idioma** que la lógica de negocio nos ha indicado que debemos mostrar:

Trabajar con idiomas

Descarga: Motor de plantillas

Puedes descargar el motor de plantillas que acabamos de desarrollar, haciendo clic aquí.

Motor de plantillas predesarrollado: Smarty

Ya hemos visto que podemos desarrollar un motor de plantillas por nosotros mismos. No obstante **existen motores de plantillas predesarrollados**, que podemos utilizar para separar la lógica de negocio de la presentación, sin necesidad de realizar desarrollos específicos para ello.

Ya que debemos separar la lógica de negocio de la presentación en **todos nuestros proyectos**, es crítico que el módulo encargado de realizar esta acción este **libre de errores**, nos proporcione **todas las funcionalidades que necesitamos** y además esté **lo más optimizado posible**.

La forma más rápida de conseguir estas premisas es **utilizar un motor de plantillas predesarrollado**, que se encuentre lo suficientemente testado e implantado como para asegurarnos una robustez y fiabilidad completas.

En esta sección veremos cómo trabajar con el **motor de plantillas Smarty**.

Smarty es uno de los motores de plantillas para PHP más extendidos en la comunidad gracias a su gran **fiabilidad** y **facilidad de uso**.

Alguna de las características que lo hacen una de las mejores alternativas para separar la lógica de negocio de la presentación son:

1
Facilidad de instalación, configuración y utilización.
2
Permite **separar la lógica de negocio de la presentación** de forma eficaz.
3
Permite **segmentar las plantillas** y trabajar con **archivos de recursos de texto** para implementar rápidamente varios idiomas.

4
Dispone de una **nomenclatura sencilla** y potente que nos permite hacer **bucles**, **condicionales** y **cálculos básicos**.
5
Utiliza **almacenamiento en caché** para optimizar la velocidad de generación de código.
6
Es un proyecto de código abierto y su uso es **gratuito**.

Descargar y configurar Smarty

Puedes descargar Smarty, de forma gratuita, desde su página web oficial: https://www.smarty.net/

Sólo debes pulsar sobre **Download** y seleccionar la última versión disponible del motor de plantillas.

Descarga: Smarty 3.1.30

Si lo prefieres, puedes descargar la versión 3.1.30 de Smarty desde el propio curso, pulsando aquí.

Una vez descargado Smarty, la instalación es muy sencilla, sólo debes seguir los siguientes pasos:

1. Crea una carpeta llamada **smarty** dentro de la carpeta pública de tu servidor.
2. Descomprime el archivo descargado dentro de dicha carpeta.
3. Crea una carpeta llamada **smarty** en una ubicación **fuera de la carpeta pública de tu servidor**.
4. Dentro de la carpeta smarty, ubicada fuera de la carpeta pública, crea las carpetas, **template**, **config**, **compile** y **cache**.
5. Añade el siguiente fragmento de código a tus scripts:

```
include('smarty/libs/Smarty.class.php');
$smarty = new Smarty();
$smarty -> template_dir = 'RUTA_TEMPLATE';
$smarty -> config_dir = 'RUTA_CONFIG';
$smarty -> compile_dir = 'RUTA_COMPILE';
$smarty -> cache_dir = 'RUTA_CACHE';
```

RUTA_TEMPLATE, RUTA_CONFIG, RUTA_COMPILE y RUTA_CACHE son las rutas absolutas a las cuatro carpetas creadas dentro de la carpeta smarty que se encuentra fuera de la carpeta pública.

Hemos creado nuestra carpeta smarty, no pública en **C:\xampp\smarty**

El fragmento de código que deberemos incluir en nuestros scripts será:

```
include('smarty/libs/Smarty.class.php');
$smarty = new Smarty();
$smarty -> template_dir = 'C:\\xampp\\smarty\\template\\';
$smarty -> config_dir = 'C:\\xampp\\smarty\config\\';
$smarty -> compile_dir = 'C:\\xampp\\smarty\compile\\';
$smarty -> cache_dir = 'C:\\xampp\\smarty\\cache\\';
```

Descargar y configurar Smarty

Utilizar las plantillas

Como en cualquier motor de plantillas, en Smarty las plantillas son **archivos de texto plano que contienen la presentación** de la aplicación. Esta presentación está compuesta por **código HTML** y **código Smarty**.

Anotación: Código Smarty

El código Smarty está compuesto por **una serie de nomenclaturas sencillas** para indicar ciertas **funcionalidades relacionadas con la presentación** (mostrar una variable, repetir segmentos de código...)

Todas las plantillas estarán incluidas en el directorio que hemos especificado en $smarty -> template_dir y tendrán extensión **.tpl**

Una vez creada la plantilla, la mostraremos desde la lógica de negocio utilizando la siguiente sintaxis:

```
$smarty -> display('archivo.tpl');
```

archivo.tpl es el nombre de la plantilla que queremos mostrar.

Vamos a ver un ejemplo:

La plantilla ubicada en C:\xampp\smarty\template\holamundo.tpl tiene el siguiente contenido:

```
<!DOCTYPE html>
<html>
<head>
    <title>Hola mundo</title>
</head>
<body>
    Hola mundo
</body>
</html>
```

En la carpeta publica de nuestro servidor tenemos el archivo holamundo.php con el siguiente contenido:

```
<?php
    include('smarty/libs/Smarty.class.php');
    $smarty = new Smarty();
    $smarty -> template_dir = 'C:\\xampp\\smarty\\template\\';
    $smarty -> config_dir = 'C:\\xampp\\smarty\config\\';
    $smarty -> compile_dir = 'C:\\xampp\\smarty\compile\\';
    $smarty -> cache_dir = 'C:\\xampp\\smarty\\cache\\';
    $smarty -> display('holamundo.tpl');
?>
```

Al abrir http://localhost/holamundo.php, el navegador mostrará por pantalla lo siguiente:

Hola mundo

Igual que en el motor de plantillas que desarrollamos anteriormente, con Smarty podemos**pasar una serie de variables desde la lógica de negocio a la presentación**. De esta forma añadiremos los datos a la plantilla.

Para ello utilizaremos la siguiente sintaxis:

$smarty -> assign('nombre_variable',VALOR);

'nombre_variable' es el nombre de la variable que crearemos en la plantilla.

VALOR es el valor que asignaremos a la variable.

Una vez pasada la variable a la plantilla, ubicaremos dónde será mostrada indicando la siguiente nomenclatura:

{$nombre_variable}

nombre_variable es el nombre de la variable que hemos pasado a la plantilla.

Vamos a ver un ejemplo:

La plantilla ubicada en C:\xampp\smarty\template\suma.tpl tiene el siguiente contenido:

```
<!DOCTYPE html>
<html>
<head>
    <title>Hola mundo</title>
</head>
<body>
    La suma de {$a} + {$b} es {$resultado}
</body>
</html>
```

En la carpeta publica de nuestro servidor tenemos el archivo suma.php con el siguiente contenido:

```
<?php
    include('smarty/libs/Smarty.class.php');
    $smarty = new Smarty();
    $smarty -> template_dir = 'C:\\xampp\\smarty\\template\\';
    $smarty -> config_dir = 'C:\\xampp\\smarty\config\\';
    $smarty -> compile_dir = 'C:\\xampp\\smarty\compile\\';
    $smarty -> cache_dir = 'C:\\xampp\\smarty\\cache\\';
    $valor1 = 75;
    $valor2 = 48;
    $smarty -> assign('a',$valor1);
    $smarty -> assign('b',$valor2);
    $smarty -> assign('resultado',$valor1 + $valor2);
    $smarty -> display('suma.tpl');
?>
```

Al abrir http://localhost/suma.php, el navegador mostrará por pantalla lo siguiente:

La suma de 75 + 45 es 123

Utilizar las plantillas

Truco: Guardar HTML en una variable

Smarty nos da la posibilidad de **guardar el código HTML generado en una variable**, en vez de mostrarlo por pantalla. Para ello utilizaremos la siguiente sintaxis:

```
$variable = $smarty -> fetch('archivo.tpl');
```

Esto nos será de gran utilidad cuando queremos generar código HTML para enviarlo por email, guardarlo en un archivo o procesos similares.

Segmentar las plantillas

Una de las funcionalidades básicas, que debe tener todo motor de plantillas, para poder establecer una estructura lógica en la presentación, es la posibilidad de **segmentar las plantillas para incluir una dentro de otra**.

De esta forma, podemos mantener estructuras que se repiten, como cabeceras, menús o pies, **en un único lugar** y modificarlas fácilmente de forma global.

Para incluir una plantilla Smarty dentro de otra, utilizaremos la siguiente nomenclatura:

```
{include file="archivo.tpl"}
```

archivo.tpl es el nombre de la plantilla que queremos insertar.

Vamos a ver un ejemplo:

La plantilla pagina.tpl tiene el siguiente contenido:

```
<!DOCTYPE html>
<html>
<head>
    <title>Hola mundo</title>
</head>
<body>
    {include file="cabecera.tpl"}
    <main>Cuerpo de la página</main>
</body>
</html>
```

La plantilla cabecera.tpl tiene el siguiente contenido:

```
<header>Cabecera de la página</header>
```

Cuando el motor Smarty produzca el código final de la plantilla pagina.tpl generará el siguiente HTML:

```
<!DOCTYPE html>
<html>
<head>
    <title>Hola mundo</title>
</head>
<body>
    <header>Cabecera de la página</header>
    <main>Cuerpo de la página</main>
</body>
</html>
```

Segmentar las plantillas

Repetir secciones de código

Ya hemos comprobado como, en muchas ocasiones, **será necesario repetir un bloque de código HTML**

Páginas de productos, noticias o listados en general repetirán el fragmento correspondiente, por cada uno de los datos existentes. Para ello Smarty, nos provee de varias nomenclaturas para expresar bucles.

foreach

La primera de ellas es un bucle similar a **foreach**, que recorre los elementos de un array dado.

Su nomenclatura es la siguiente:

```
{foreach from=$miarray item=$componente key=$clave}
    Fragmento HTML a repetir.
{/foreach}
```

$miarray es el array a recorrer. Podemos asignarlo desde la lógica de negocio, como cualquier otra variable.

$componente se rellenara con los datos del elemento actual del array.

$clave se rellenara con el índice actual del array. Este parámetro es opcional.

Vamos a ver un ejemplo:

La plantilla pagina.tpl tiene el siguiente contenido:

```
<!DOCTYPE html>
<html>
<head>
    <title>Hola mundo</title>
</head>
<body>
    <table>
    <tr>
        <th>Nombre</th>
        <th>Apellidos</th>
        <th>Edad</th>
    </tr>
{foreach from=$clientes item=$persona}
    <tr>
        <td>{$persona.nombre}</td>
        <td>{$persona.apellidos}</td>
        <td>{$persona.edad}</td>
    </tr>
{/foreach}
    </table>
</body>
</html>
```

En la carpeta publica de nuestro servidor tenemos el archivo pagina.php con el siguiente contenido:

```
<?php
    include('smarty/libs/Smarty.class.php');
    $smarty = new Smarty();
    $smarty -> template_dir = 'C:\\xampp\\smarty\\template\\';
    $smarty -> config_dir = 'C:\\xampp\\smarty\config\\';
    $smarty -> compile_dir = 'C:\\xampp\\smarty\compile\\';
    $smarty -> cache_dir = 'C:\\xampp\\smarty\\cache\\';
    $clientes[0]['nombre'] = 'Pedro';
    $clientes[0]['apellidos'] = 'López del Río';
    $clientes[0]['edad'] = 45;
    $clientes[1]['nombre'] = 'Rosa';
    $clientes[1]['apellidos'] = 'Hita Pozo';
    $clientes[1]['edad'] = 34;
    $clientes[2]['nombre'] = 'Laura';
    $clientes[2]['apellidos'] = 'Hernández Gutiérrez';
    $clientes[2]['edad'] = 19;
    $smarty -> assign('clientes',$clientes);
    $smarty -> display('pagina.tpl');
?>
```

Cuando llamemos al script pagina.php generará el siguiente HTML:

```
<!DOCTYPE html>
<html>
<head>
    <title>Hola mundo</title>
</head>
<body>
    <table>
    <tr>
        <th>Nombre</th>
        <th>Apellidos</th>
        <th>Edad</th>
    </tr>
    <tr>
        <td>Pedro</td>
        <td>López del río</td>
        <td>45</td>
    </tr>
    <tr>
        <td>Rosa</td>
        <td>Hita Pozo</td>
        <td>34</td>
    </tr>
    <tr>
        <td>Laura</td>
        <td>Hernández Gutiérrez</td>
        <td>19</td>
    </tr>
    </table>
</body>
</html>
```

Acceder a componentes de un array en Smarty

Si te fijas, hemos utilizado la siguiente nomenclatura para acceder a la componente de un array:

$nombre_array.componente

Para acceder a la componente de un array desde Smarty, solo debes indicarla con el nombre del array, un punto y el nombre de la componente.

for

Smarty también nos ofrece la posibilidad de realizar bucles de tipo**for**, en los que damos una serie de pasos determinados.

Su nomenclatura es la siguiente:

```
{section start=VALOR_INICIO loop=VALOR_FIN step=VALOR_PASO name='nombre'}
    Fragmento HTML a repetir.
{/section}
```

VALOR_INICIO es el valor desde el que comenzará el bucle.

VALOR_FIN es el valor hasta el que llegará el bucle.

VALOR_PASO es el valor que se añadirá al contador interno en cada vuelta.

'nombre' es un parámetro obligatorio que debemos indicar para nombrar a la sección.

Vamos a ver un ejemplo:

La plantilla pagina.tpl tiene el siguiente contenido:

```
<!DOCTYPE html>
<html>
<head>
    <title>Hola mundo</title>
</head>
<body>
{section start=0 loop=3 step=1 name='hm'}
    Hola mundo<br>
{/section}
</body>
</html>
```

Cuando el motor Smarty produzca el código final de la plantilla pagina.tpl generará el siguiente HTML:

```
<!DOCTYPE html>
<html>
<head>
    <title>Hola mundo</title>
</head>
<body>
    Hola mundo<br>
    Hola mundo<br>
    Hola mundo<br>
</body>
</html>
```

Repetir secciones de código

Trabajar con idiomas

Smarty también nos provee de la funcionalidad necesaria para generar**archivos de recursos de texto**. Igual que hicimos en el motor de plantillas que desarrollamos anteriormente, utilizaremos estos archivos para **guardar las traducciones de los textos** de nuestro sitio web. De esta forma, podremos cambiar el idioma de la página fácilmente.

Todos los archivos de recursos de texto estarán incluidas en el directorio que hemos especificado en $smarty -> config_dir y tendrán extensión **.conf**

La sintaxis de un archivo de recursos será similar a la siguiente:

```
[es]
id_texto1=Cadena de texto 1
id_texto2=Cadena de texto 2

[en]
id_texto1=Text string 1
id_texto2=Text string 2
```

Como puedes ver, en el mismo archivo de recursos crearemos **diferentes secciones, una por cada idioma**, en las que mostraremos la traducción de cada cadena de texto.

Una vez creado el archivo de recursos, necesitaremos incluirlo en la plantilla que vaya a utilizarlo. Para ello utilizaremos la siguiente nomenclatura:

{config_load file='archivo.conf' section=IDIOMA}

archivo.conf es el nombre del archivo de recursos a incluir.

IDIOMA es el código del idioma, tal como lo hemos indicado en el archivo de recursos.

Por último, utilizaremos la siguiente nomenclatura para indicar en qué lugar de la plantilla se incluirá el texto:

{#id_texto1#}

id_texto1 será el identificador del texto.

Vamos a ver un ejemplo:

El archivo de recursos ubicado en C:\xampp\smarty\config\holamundo.conf tiene el siguiente contenido:

```
[es]
holamundo=Hola mundo

[en]
holamundo=Hello world
```

La plantilla ubicada en C:\xampp\smarty\template\holamundo.tpl tiene el siguiente contenido:

```
{config_load file='holamundo.conf' section='en'}
<!DOCTYPE html>
<html>
<head>
    <title>{#holamundo#}</title>
</head>
<body>
    {#holamundo#}
</body>
</html>
```

Cuando el motor Smarty produzca el código final de la plantilla holamundo.tpl generará el siguiente HTML:

```
<!DOCTYPE html>
<html>
<head>
    <title>Hello world</title>
</head>
<body>
    Hello world
</body>
</html>
```

Truco: Especificar idioma desde la lógica de negocio

Lo más habitual es indicar el idioma que utilizará la plantilla**desde la lógica de negocio**.

Para esto sólo debemos pasar una variable a la plantilla de la forma habitual:

```
$smarty -> assign('idioma','en');
```

Y en la plantilla, utilizar la variable para indicar la sección del archivo de recursos a utilizar:

```
{config_load file='archivo.conf' section=$idioma}
```

Trabajar con idiomas

Funcionalidades adicionales del código Smarty

El código Smarty dispone de gran cantidad de funcionalidades que lo asemejan a un lenguaje de programación completo.

Aunque esto nos brinda una gran potencia, no debemos caer en la tentación de utilizarlo indebidamente. Todas las acciones que realicemos con Smarty **deben ser exclusivamente para la presentación** de la aplicación y nunca para la lógica de negocio.

Vamos a ver algunas de las funcionalidades adicionales que nos permite el código Smarty:

Variables

Con Smarty podemos crear variables directamente en la plantilla, o modificar el valor de las variables pasadas por la lógica de negocio gracias a su nomenclatura de **asignación de variables**.

Su sintaxis es la siguiente:

{assign var='VARIABLE' value=VALOR}

VARIABLE es el nombre de la variable a asignar. Puede ser tanto una variable ya existente, como una nueva variable.

VALOR es el nuevo valor que contendrá la variable.

Tenemos el siguiente código en la plantilla Smarty:

{assign var='mivariable' value=45}
El valor de la variable es {$mivariable}

Este fragmento generaría el siguiente código HTML:

El valor de la variable es 45

Operaciones básicas

Smarty nos permite realizar operaciones básicas como la **suma**, la **resta**, la **multiplicación** o la **división** con **la misma sintaxis que utilizamos en PHP**.

Desde la lógica de negocio, hemos pasado a la plantilla la variable $mivariable, con el valor **100**.

En la plantilla Smarty tenemos el siguiente código:

{assign var='mivariable' value=$mivariable + 50}
El valor de la variable es {$mivariable}

Este fragmento generaría el siguiente código HTML:

El valor de la variable es 150

Comentarios

Como en cualquier lenguaje de programación, podemos realizar comentarios en el código que nos servirán para **explicar** su funcionamiento, o **desactivar porciones de código**.

Para indicar un comentario en Smarty, debemos incluirlo entre las nomenclaturas{** y **}

Tenemos el siguiente código en la plantilla Smarty:

```
Lorem ipsum dolor
{**
Esto es un comentario
**}
Lorem ipsum dolor
```

Este fragmento generaría el siguiente código HTML:

```
Lorem ipsum dolor

Lorem ipsum dolor
```

Literales

En muchas ocasiones, el código Smarty**puede entrar en conflicto** con el contenido que realmente deseamos mostrar.

Esto suele suceder cuando utilizamos en nuestra presentación llaves, u otros caracteres que**podrían ser interpretados por Smarty como parte de su nomenclatura**.

Para indicar a Smarty que un fragmento de código, no debe ser procesado sino**mostrado literalmente**, utilizaremos la siguiente nomenclatura:

```
{literal}
Fragmento de código literal
{/literal}
```

Tenemos el siguiente código en la plantilla Smarty:

```
{assign var='mivariable' value=45}
{literal}
El valor es: {$mivariable}
{/literal}
```

Este fragmento generaría el siguiente código HTML:

```
El valor es: {$mivariable}
```

Condicionales

Smarty nos ofrece la posibilidad de crear condicionales dentro de las plantillas para mostrar un fragmento de código u otro, en función de una **condición**.

La nomenclatura de un condicional en Smarty es la siguiente:

```
{if condicion}
    Bloque si la condición es verdadera
{else}
    Bloque si la condición es falsa
{/if}
```

condicion es una expresión condicional, con **la misma sintaxis que PHP**, que puede valer true o false.

El bloque else es opcional. Lo incluiremos únicamente cuando lo necesitemos.

Desde la lógica de negocio, hemos pasado a la plantilla la variable $mivariable, con el valor **100**.

Tenemos el siguiente código en la plantilla Smarty:

```
{if $mivariable >= 200}
La variable es mayor o igual a 200
{else}
La variable es menor a 200
{/if}
```

Este fragmento generaría el siguiente código HTML:

La variable es menor a 200

Funciones

Una de las funcionalidades más potentes del código Smarty, es la capacidad de **utilizar funciones de PHP** para formatear ciertos valores. Podemos utilizar cualquier **función predefinida** en PHP o incluso **funciones propias**.

Para pasar una función a un valor utilizaremos la siguiente sintaxis:

```
$mivariable|funcion
```

$mivariable será el valor que se le pase como parámetro a la función.

funcion será el nombre de la función.

Desde la lógica de negocio, hemos pasado a la plantilla la variable $mivariable, con el valor **'Murciélago'**.

Tenemos el siguiente código en la plantilla Smarty:

El animal es: {$mivariable|htmlentities}

Este fragmento generaría el siguiente código HTML:

El animal es: Murciélago

Importante: Qué funciones debo utilizar y cuáles no

Sólo debes utilizar funciones de PHP **para dar formato**.

Nunca utilices funciones que realicen acciones ya que estarías mezclando la lógica de negocio con la presentación.

También podemos pasar parámetros adicionales a la función utilizando la siguiente sintaxis:

$mivariable|funcion:$parametro1:$parametro2

Desde la lógica de negocio, hemos pasado a la plantilla la variable $mivariable, con el valor **1587.4**.

Tenemos el siguiente código en la plantilla Smarty:

El número es: {$mivariable|number_format:2:',':'.'}

Este fragmento generaría el siguiente código HTML:

El número es: 1.587,40

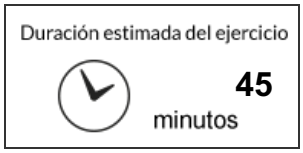
Funcionalidades del lenguaje Smarty

Hemos aprendido

- La **lógica de negocio** es la parte de la aplicación web que se encarga de extraer datos y realizar acciones. Por lo general estará constituida por nuestro código PHP.
- La **presentación** es la parte de la aplicación web que se encarga de modelar el aspecto final que el usuario verá. Por lo general estará constituida por nuestro código HTML.
- **Siempre debemos separar la lógica de negocio de la presentación** para lograr una organización del código óptima.
- Para separar la lógica de negocio de la presentación, utilizaremos un **motor de plantillas**.
- Podemos construir **nuestro propio motor de plantillas** o utilizar uno predesarrollado como **Smarty**.
- Las funcionalidades habituales de un motor de plantillas son:
  - **Generar el código HTML** correspondiente de la plantilla.
  - **Pasar valores** desde la lógica de negocio a la plantilla.
  - **Segmentar las plantillas** para evitar repetir porciones de código.
  - **Repetir bloques de código** para representar datos en forma de lista.
  - Manejar **archivos de recursos de texto** para poder trabajar con diferentes idiomas de forma sencilla.

Ejercicios

Ejercicio: Separar lógica de negocio y presentación de una aplicación web



Para finalizar correctamente el ejercicio, deberás separar la lógica de negocio de la presentación en una aplicación web.

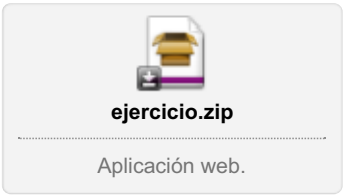
Esta aplicación web se encuentra actualmente programada con ambas capas mezcladas.

Lo necesario para comenzar

Descarga el archivo **ejercicio.zip** y descomprímelo en el directorio público de tu servidor. Contiene una aplicación web que realiza la gestión de una tabla de productos.

Entre los archivos descomprimidos encontrarás el archivo **tienda.sql** con la estructura y datos de la base de datos que utiliza la aplicación. Crea una nueva base de datos con esta estructura.

Asegúrate que los datos de conexión a la base de datos son correctos. Puedes ver la conexión que utiliza la aplicación en **funciones.php**.



Pasos a seguir

1. Configura un motor de plantillas en la aplicación web. Puedes usar el motor que prefieras, aunque te recomendamos utilizar Smarty.
2. Crea las plantillas necesarias e incluye en ellas el código HTML de la aplicación. Recuerda segmentar las plantillas para no repetir código si es posible.
3. Dota de funcionalidad a las plantillas, añadiendo las variables, bucles o condicionales que necesites.
4. Elimina todo el código HTML de la lógica de negocio y haz que ésta muestre la plantilla correspondiente.
5. Asegúrate de que la aplicación sigue funcionando correctamente y de que la lógica de negocio y la presentación están completamente separadas.

Solución

Solución al ejercicio

Recursos

Enlaces de Interés

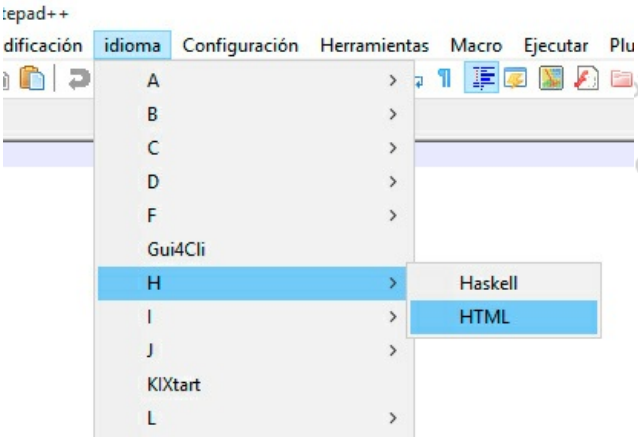
https://www.smarty.net/
https://www.smarty.net/
Página web oficial de Smarty

Preguntas Frecuentes

1. **He abierto un fichero con extensión TPL en Notepad++, pero no aparece resaltado el HTML ¿Cómo puedo hacer que se resalte el código con color?**

   Notepad++ no identifica los ficheros TPL como archivos HTML y por ello no los resalta con color.

   No obstante, puedes indicar en qué lenguaje quieres resaltar el código del archivo actual desde el menú **idioma** de Notepad++.



2. **Cuando tengo un sitio web en diferentes idiomas ¿sólo debo cambiar los textos correspondientes? ¿nunca cambiará el diseño?**

   Por lo general, sólo debemos cambiar los textos. No obstante, hay idiomas que nos exigirán pequeños cambios de diseño.

   Por ejemplo, el árabe o el hebreo se leen de derecha a izquierda, lo que nos obligará a cambiar todas las alineaciones de texto de la página.

Glosario.

- **Lógica de negocio**: Es la parte de la aplicación web encargada del manejo de la información y la realización de los procesos.

- **Motor de plantillas**: Módulo que permite que las plantillas sean interpretadas, generando en código HTML final a mostrar. De esta forma podemos separar la lógica de negocio de la presentación.

- **Presentación**: Es la parte de la aplicación web encargada de establecer cómo se deben mostrar los datos al usuario final.

- **Skins**: También llamadas pieles o temas, son distintas apariencias gráficas que pueden configurarse para una misma aplicación, modificando por completo su diseño, pero no su funcionalidad.