Modelo Vista Controlador © ADR Infor SL

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

Indice

Modelo Vista Controlador

Al finalizar esta unidad el alumno será capaz de estructurar el código de sus aplicaciones web utilizando el patrón Modelo Vista Controlador.

PHP nos proporciona gran flexibilidad a la hora de estructurar nuestras aplicaciones. Pero esto nos lleva a plantearnos una importante cuestión **¿Cuál es la forma más adecuada para organizar nuestro código?**

Uno de los patrones de arquitectura de software más famosos y extendidos es el patrón Modelo Vista Controlador (también conocido por su abreviatura **MVC**). Este patrón establece un **estándar** sobre **cómo debemos organizar nuestro código en una aplicación**.

Aunque el patrón MVC esta pensado para aplicaciones de todo tipo, **se ha hecho especialmente útil en aplicaciones web** en las que, debido a la flexibilidad a la hora de estructurar, **es especialmente necesario establecer una organización del código** para mantener las aplicaciones legibles y fáciles de mantener.

El patrón MVC se basa en **segmentar la aplicación en tres capas** totalmente separadas:

Modelo

Esta capa se encarga exclusivamente del **acceso a los datos**.

Las tareas del modelo son:

- Acceder a las bases de datos o cualquier otro almacén de información que la aplicación necesite.
- Recuperar datos.
- Modificar datos.
- Gestionar de privilegios para el acceso a datos.

Vista

Esta capa se encarga exclusivamente de **presentar** el resultado al usuario.

Las tareas de la vista son:

- Generar el HTML con la información a presentar al usuario y la interfaz de la aplicación.
- Enviar el HTML al usuario.

Controlador

Esta capa se encarga de **recibir las solicitudes** del usuario, interpretarlas y gestionarlas.

Las tareas del modelo son:

- Recibir las solicitudes del usuario.
- Implementar la lógica de negocio de la aplicación.
- Comunicarse con el modelo y solicitarle las acciones necesarias.
- Enviar los resultados a la vista.

El controlador es a la vez, **punto de entrada a la aplicación** e **intermediario entre el modelo y la vista**

El objetivo es que cada capa **se encargue exclusivamente de su función** y que **pueda ser reemplazada** sin necesidad de modificar nada en el resto de capas.

Tenemos una aplicación web ya desarrollada con el **patrón MVC**.

Hemos decidido **reestructurar las tablas de la base de datos**, para guardar los datos de una forma más eficiente y lógica.

Lo único que necesitamos hacer es **cambiar la capa Modelo**, indicando la nueva forma de recuperar y guardar estos datos.

Aunque reestructurar la base de datos una vez que la aplicación está desarrollada es una acción muy agresiva, **la capa de Vista y Controlador no necesitan ningún cambio** y el mantenimiento necesario para realizar esta acción es muy sencillo y rápido.

El logro del patrón MVC es conseguir **separar los procesos básicos que debe realizar cualquier aplicación web** de forma que cada uno sea una entidad independiente. Algunas de las ventajas que nos proporciona este modo de trabajo son las siguiente:

1

Podemos cambiar el aspecto gráfico de la aplicación y su interfaz completamente, sin necesidad de cambiar la lógica de negocio.

2

Podemos cambiar el almacenamiento de datos y su estructura por completo sin que ninguna capa, a excepción del modelo, se vea afectada. De esta forma conservamos los procesos intactos.

3
Podemos tener programadores especializados en cada una de las capas. Cada uno de ellos puede trabajar de forma independiente y simultánea en su capa, sin afectar en absoluto al resto.
4
Las aplicaciones se encuentran totalmente organizadas, siendo muy sencillo localizar y modificar los procesos, en base a qué capa pertenecen.
5
Al establecer un estándar de organización, es mas sencillo que nuevos programadores que se unan al proyecto, puedan manejar el código sin grandes explicaciones.

Frameworks PHP

Existen gran cantidad de frameworks para PHP que **añaden un nivel de abstracción adicional** a nuestra programación. El objetivo de estos frameworks es unificar conceptos, prácticas y criterios a la hora de desarrollar aplicaciones.

La mayoría de los frameworks **están basados en el patrón MVC** y nos "obligan" a estructurar nuestras aplicaciones web cumpliendo con las normas del mismo.

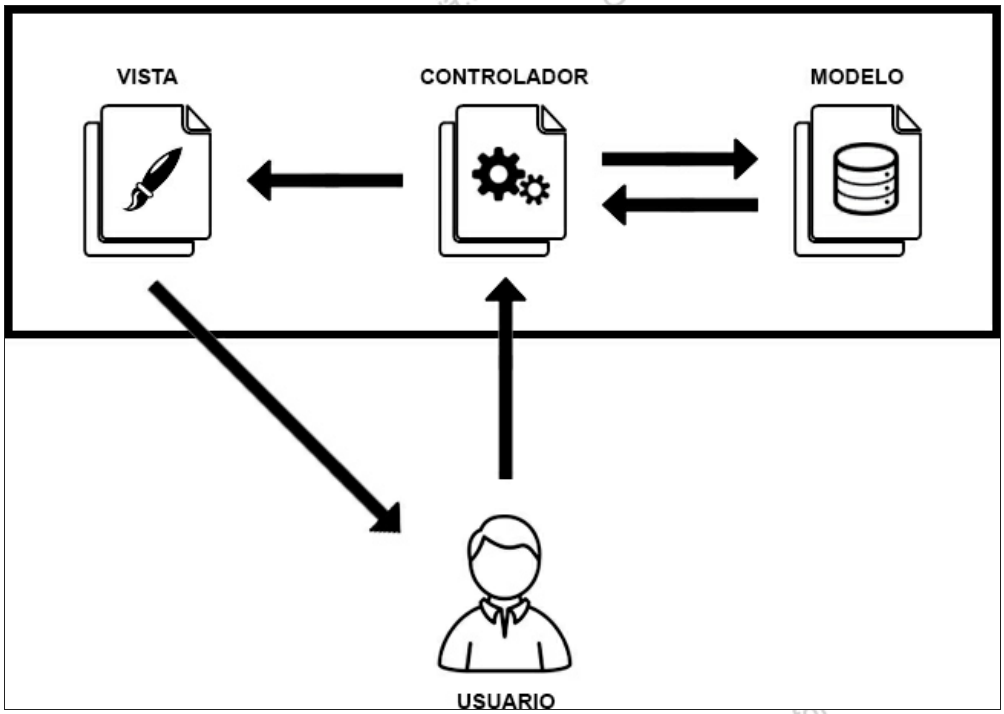
Alguno de las frameworks de PHP más famosos que utilizan el patrón MVC son:

- Laravel
- Zend Framework
- CakePHP
- Symfony
- CodeIgniter

Ciclo de vida del patrón MVC

Lo primero que necesitamos entender para comenzar a estructurar nuestras aplicaciones según el patrón MVC es su ciclo de vida. Es decir, **qué pasos realiza la aplicación** cuando llega una petición HTTP y **cómo interviene cada capa** en el proceso.

El ciclo de vida de cualquier aplicación web que siga el patrón MVC es el siguiente:



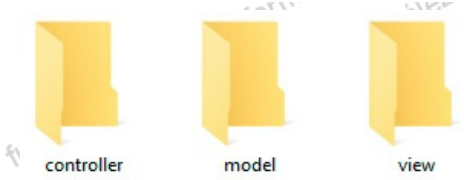
1. El **usuario** realiza una **petición HTTP** a la aplicación web.
2. La petición es **atendida por el controlador**, que interpreta qué está solicitando el usuario.
3. El controlador **aplica la lógica de negocio** correspondiente y realiza las acciones programadas para la solicitud.
4. El controlador **se comunica con el modelo** para solicitarle información o que realice las acciones correspondientes.
5. El modelo **devuelve al controlador el resultado** de sus solicitudes.
6. El controlador **envía a la vista los datos** que deben ser presentados al usuario.
7. La vista **genera el HTML** final con los datos obtenidos del controlador.
8. La vista **envía al usuario el HTML** que ha compuesto.

Ciclo de vida del patrón MVC

Estructura de la aplicación web

Todas las aplicaciones web realizadas según el patrón MVC,**deben cumplir con una estructuración** prefijada.

Comenzaremos creando tres nuevas carpetas dentro de la carpeta pública del servidor con el nombre **controller**, **model** y **view**.



- **controller** contendrá todos los scripts de la capa controlador.
- **model** contendrá todos los scripts de la capa modelo.
- **view** contendrá todos los scripts de la capa vista.

Por lo general, **cada sección de la web** contará con **un script PHP en cada una de las capas** con el nombre de la sección. Casa script se encargará de las acciones de la sección específicas para la capa en la que está ubicado.

Tenemos una web con las secciones **noticias** y **productos**.

Para crear las secciones generaremos los siguientes scripts:

- controller/noticias.php
- controller/productos.php
- model/noticias.php
- model/productos.php
- view/noticias.php
- view/productos.php

Tal y como hemos visto en el ciclo de vida del patrón MVC, **el usuario realiza la petición directamente al controlador**. Sin embargo, el controlador se encuentra siempre dentro de la carpeta **controller**, lo cual hace que **la URL solicitada por el usuario no coincida con la estructura de directorios del servidor**

Siempre que la URL que el usuario debe solicitar no coincida con la estructura de carpetas del servidor, debemos utilizar la **reescritura de URLs** para ajustarlo.

Recuerda: Reescritura de URLs

Para configurar la reescritura de URLs del servidor, debemos utilizar el módulo**mod_rewrite** de Apache.

Utilizaremos un archivo **.htaccess** para establecer las directivas de mod_rewrite de forma que **todas las URLs pasen por un único script** que las controle. El contenido de .htaccess debe ser el siguiente:

```
Options +FollowSymLinks
RewriteEngine on
RewriteCond %{REQUEST_URI} !\.(jpg|css|js|gif|png)$ [NC]
RewriteRule ^(.*) index.php?pantalla=$1 [QSA]
```

El script que controla las peticiones, obtendrá la URL que le está llamando de $_GET['pantalla'].

NOTA: Hemos añadido una línea adicional a nuestro .htaccess que evita que las URLs que contengan archivos de recursos (imágenes, hojas de estilos, scripts de javascript...) sean reescritas y procesadas por el script de PHP.

Mediante la reescritura de URLs, haremos que todas las URLs que el usuario nos envía sean procesadas por el script **index.php** que ubicaremos en la raíz de la carpeta pública del servidor.

Este script **analizará la URL** que el usuario ha utilizado y **llamará al controlador de la sección** correspondiente mediante un **include**.

Estructura de la aplicación web - Parte 1

Estructura de la aplicación web - Parte 2

Controlador

Como hemos visto, el controlador **recibe la solicitud del usuario**.

En este script implementaremos toda la **lógica de negocio** de la sección, **comunicándonos con el modelo** para recuperar y gestionar los datos. Por último, **llamaremos a la vista** para mostrar el resultado.

Qué no debemos hacer nunca en el controlador:

- Nunca debemos **conectarnos a una base de datos** o utilizar cualquier otro tipo de almacén de información. Esto es una competencia exclusiva del modelo.
- Nunca debemos **construir una sentencia SQL**. El modelo es el encargado de establecer cómo son recuperados los datos.
- Nunca debemos **construir u organizar el HTML**. La presentación final es competencia exclusiva de la vista.

Al delegar las operaciones de manejo de datos y presentación en el resto de capas, en el controlador **podremos ver de una forma simple y organizada la lógica de negocio** de la aplicación.

En el controlador de la sección de noticias tenemos el siguiente código:

```
<?php

    include('model/noticias.php');
    $modelo = new model_noticias();

    switch ($_POST['acc']) {
    case 'add_comentario':
        $modelo -> add_comentario($_POST['comentario']);
        break;
    }

    $datos = $modelo -> get_noticias();

    include ('view/noticias.php');
    $vista = new view_noticias();

    $vista -> set_noticias($datos);
    $vista -> mostrar();

?>
```

Como puedes ver, el controlador aplica la lógica de negocio, **delegando al modelo** las tareas de **guardar un comentario** o **recuperar los datos** de las noticias.

Por último **delega en la vista** la tarea de **generar la presentación** final de los datos.

Controlador

Modelo

El modelo es el encargado de **recuperar y modificar la información**.

Aunque el trabajo principal del modelo suele estar ligado al manejo de bases de datos, debemos entender que esta capa debe encargarse de **cualquier tipo de almacenamiento de información**.

EJEMPLO: Un usuario sube su imagen de perfil que se almacenará en una carpeta del servidor. El modelo será el encargado de guardar el archivo de imagen en el lugar correspondiente.

En este script **debemos generar una clase** que se encargue de la gestión de los datos. Dentro de la clase, **implementaremos los métodos** necesarios para realizar las acciones que el controlador nos solicite.

Al encargarnos en esta capa de cómo se guardan los datos sin importar en qué situación son solicitados, **podemos ver de un solo vistazo todos los procesos de tratamiento de información** De esta forma es muy sencillo definir estos procesos y optimizarlos.

En el modelo de la sección de noticias tenemos el siguiente código:

```
<?php

    include_once('db.php');

    class model_noticias {

        public function get_noticias() {
            $consulta = 'SELECT * FROM noticias ORDER BY fecha DESC';
            $resultado = $GLOBALS['conexion'] -> sql($consulta);
            return $resultado;
        }

        public function add_comentario($comentario) {
            $consulta  = 'INSERT INTO comentarios (comentario) VALUES (';
            $consulta .= '\'' . addslashes($comentario) . '\')';
            $GLOBALS['conexion'] -> sql($consulta);
        }

    }

?>
```

Como puedes ver, el modelo **define la forma en la que deben ser recuperadas las noticias y añadidos los comentarios**, de forma que el controlador puede solicitar estas acciones cuando lo considere necesario.

Truco: include_once

Es habitual que todos los scripts del modelo **accedan a una base de datos común**. Por ese motivo, suele generarse un **script adicional con la conexión a la base de datos**, que es incluido en todos los scripts del modelo que lo necesiten.

No obstante, en aplicaciones complejas, es posible que un controlador llame a dos modelos al mismo tiempo. En tal caso, **incluir dos veces el script de conexión a la base de datos provocará un fallo**.

Para solventar el problema, utilizaremos la función include_once, en lugar de include.

La función include_once incluye el archivo del script indicado **sólo la primera vez** que es solicitado. Si volvemos a solicitar incluir el mismo archivo, la función no realizará ninguna acción previniendo el fallo.

Modelo

Vista

La vista es la encargada de **generar la presentación** final de la información, junto con la interfaz y enviarla al usuario.

Aunque lo más habitual es que la vista se encargue de generar el HTML que se enviará al usuario, esta capa debe encargarse de **cualquier acción relacionada con la presentación de los datos en cualquier formato**.

EJEMPLO: Un usuario solicita la ficha en PDF de un producto, que se genera de forma dinámica en el momento de la solicitud. La vista será la encargada de generar y servir dicho PDF con la información suministrada por el controlador.

En este script **debemos generar una clase** que se encargue de la presentación de los datos. Dentro de la clase, **implementaremos los métodos** necesarios para que el controlador pueda suministrarnos los datos a presentar y solicitarnos que generemos la presentación final.

Al encargarnos en esta capa de cómo se muestran los datos sin importar de donde vienen o porqué, **podemos ver de un solo vistazo todos los procesos relacionados con la presentación de datos al usuario**. De esta forma es muy sencillo definir estos procesos y modificarlos si es necesario.

En la vista de la sección de noticias tenemos el siguiente código:

```
<?php

class view_noticias {

    private $datos;

    public function set_datos($datos) {
        $this -> datos = $datos;
    }

    public function mostrar() {
?>
<!DOCTYPE html>
<html>
<head>
    <title>Noticias</title>
</head>
<body>
<?php
        for ($i=0;$i<count($this -> datos);$i++) {
?>
    <h1><?php echo htmlentities($this -> datos[$i]['titular']); ?></h1>
    <div><?php echo htmlentities($this -> datos[$i]['cuerpo']); ?></div>
<?php
        }
?>
</body>
</html>
<?php
    }

}

?>
```

Como puedes ver, la vista **define la forma en la que deben mostrarse las noticias** y provee métodos para que el controlador suministre los datos finales. De esta forma, el controlador sólo debe llamar a estos métodos después de finalizar su proceso para generar la presentación.

Vista

Descarga: Ejemplo de aplicación MVC

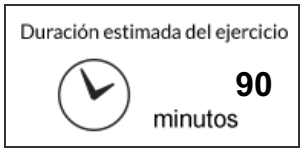
Puedes descargar el ejemplo de aplicación MVC que acabamos de desarrollar, haciendo clic aquí.

Hemos aprendido

- Modelo Vista Controlador (MVC) es un patrón de arquitectura de software que nos indica **cómo debemos estructurar el código** de nuestras aplicaciones web.
- MVC se basa en segmentar nuestras aplicaciones en tres capas:
  - **Modelo**: Se encarga de la recuperación y actualización de los datos.
  - **Vista**: Se encarga de la presentación que se enviará al usuario.
  - **Controlador**: Se encarga de la lógica de negocio y de ser intermediario entre el modelo y la vista.
- Cada sección tendrá un script en cada una de las carpetas de las tres capas.
- Las solicitudes HTTP que realice el usuario las recibirá el **controlador**.
- Como la estructura de URLs no coincide con la estructura de directorios, debemos utilizar la **reescritura de URLs** junto con un script que se encargue de gestionar el controlador adecuado para cada URL.

Ejercicios

Ejercicio: Aplicar el patrón MVC a una aplicación web



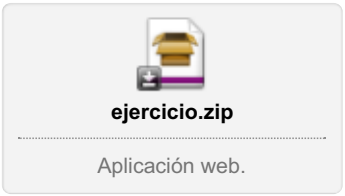
Para finalizar correctamente el ejercicio, deberás convertir una aplicación web que actualmente se encuentra programada con sus capas mezcladas, al patrón Modelo Vista Controlador.

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. Crea los directorios **model**, **view** y **controller**.
2. En cada uno de los directorios crea un script llamado **productos.php**
3. Configura la **reescritura de URLs** para que todas las URLs apunten a **index.php** en el directorio raíz.
4. Crea el script **index.php** en el directorio raíz y haz que llame al controlador adecuado según la URL.
5. Implementa el **controlador**: copia la lógica de negocio que ya existía en la aplicación y adáptala para que delegue en el modelo las operaciones de tratamiento de datos y en la vista la presentación de la página.
6. Implementa el **modelo**: crea la clase y los métodos requeridos por el controlador. Puedes copiar de la aplicación existente la forma de recuperar y guardar los datos.
7. Implementa la **vista**: crea la clase y los métodos requeridos por el controlador. Puedes copiar de la aplicación existente la forma de presentar la información.

Solución

Solución al ejercicio

Recursos

Enlaces de Interés

https://laravel.com/
https://laravel.com/
Framework para PHP basando en Modelo Vista Controlador.

https://framework.zend.com/
https://framework.zend.com/
Framework para PHP basando en Modelo Vista Controlador.

https://cakephp.org/
https://cakephp.org/
Framework para PHP basando en Modelo Vista Controlador.

https://symfony.com/
https://symfony.com/
Framework para PHP basando en Modelo Vista Controlador.

https://codeigniter.com/
https://codeigniter.com/
Framework para PHP basando en Modelo Vista Controlador.

Preguntas Frecuentes

1. **¿Puedo hacer que un controlador acceda a un modelo que no le corresponde para recuperar datos?**

   En aplicaciones complejas puede darse esta situación. No obstante, si nos ceñimos al estándar más purista de MVC, nunca deberías hacerlo.

   Si tienes varios modelos que necesitan ejecutar las mismas acciones, es recomendable crear un repositorio común de funciones donde implementar las acciones comunes a toda la aplicación.

2. **¿Puedo utilizar un motor de plantillas cuando utilizo el patrón MVC?**

   Sí, puedes utilizar cualquier solución para generar la presentación final,**siempre que lo implementes en la vista**. Debes ser capaz de poder reemplazar el motor de plantillas en el futuro sin necesidad de tocar en absoluto el controlador.

Glosario.

- **Ciclo de vida**: Llamamos ciclo de vida de una aplicación, a los diferentes niveles de ejecución por los que pasa desde que inicia hasta que termina. Analizando o fijando un ciclo de vida, podemos estructurar una aplicación de una forma organizada.

- **Framework**: Es un entorno de trabajo que añade un nivel de abstracción a nuestra programación para unificar conceptos, prácticas y criterios a la hora de desarrollar aplicaciones.