

Fundamentos de la Programación Orientada a Objetos (POO) en PHP

Sitio: [Aula Virtual do IES de Teis](#)
Curso: Desarrollo web en entorno servidor 2023-24 (DAW-DUAL-A)
Libro: Fundamentos de la Programación Orientada a Objetos (POO)
en PHP

Impreso por: Joaquín Lafuente Espino
Data: Domingo, 10 de Marzo de 2024, 17:44

Táboa de contidos

1. Fundamentos de la Programación orientada a objetos (POO)

2. Programación orientada a objetos (POO) en PHP

3. Definición de una clase

4. Propiedades tipadas

5. Creación de objetos

6. Visibilidad

7. Constantes

8. Propiedades y métodos estáticos

9. Constructores

10. Asignación y Comparación de objetos

11. Herencia

12. Autocarga de clases

13. Espacios de nombres

14. Interfaces

15. Clases abstractas

16. Traits o rasgos

17. Funciones para clases y objetos

18. Patrón de arquitectura Model-View-Controller (MVC)

18.1. Ejemplo básico de MVC con división en capas

1. Fundamentos de la Programación orientada a objetos (POO)

La POO es una metodología de programación basada en clases y objetos.

- Una **clase** es una "plantilla" o una estructura común con información estructurada que une datos (atributos) y tipo de comportamiento (métodos).
 - Un **objeto** es un ejemplar concreto de una clase (esa plantilla o estructura común) con unos datos concretos. Podéis consultar una imagen con esta distinción entre clase y objeto [aquí](#).
- **Métodos.** Son los miembros de la clase que contienen el código de la misma. Un método es como una función. Puede recibir parámetros y devolver valores.
 - **Atributos o propiedades.** Almacenan información acerca del estado del objeto al que pertenecen (y por tanto, su valor puede ser distinto para cada uno de los objetos de la misma clase)

Hasta ahora, hemos usado programación estructurada (salvo para acceder a bases de datos). En la programación estructurada los datos y las funciones no están directamente relacionados, simplemente se procesan datos de entrada para obtener otros de salida.

A diferencia de la programación estructurada, en la POO se aúnan datos y comportamiento en una misma estructura: el objeto. Para manipular los datos de un objeto (sus atributos), hay que hacer uso de sus métodos (o funciones).

Las **características** principales de la POO son:

- **Herencia.** Es el proceso de crear una clase a partir de otra, heredando su comportamiento, características y permitiendo a su vez redefinirlos y/o ampliarlos en las clases heredadas.
- **Abstracción.** Hace referencia a que cada clase oculta en su interior las peculiaridades de su implementación, y presenta al exterior una serie de métodos (interfaz pública) cuyo comportamiento está bien definido. Visto desde el exterior, cada objeto es un ente abstracto que realiza un trabajo, pero la implementación no tiene por qué ser conocida.
- **Polimorfismo.** Un mismo método puede tener comportamientos distintos en función del objeto con que se utilice. Esto es habitual en la herencia.
- **Encapsulación.** En la POO se juntan en un mismo lugar los datos y el código que los manipula.

2. Programación orientada a objetos (POO) en PHP

PHP nació como un lenguaje de script y originalmente carecía de las características de la POO.

A partir de la versión 4 y sobretodo 5, se implementaron características de soporte de la POO.

Vamos a seguir la documentación de [PHP online](https://www.php.net/manual/es/language.oop5.basic.php) comenzando por: <https://www.php.net/manual/es/language.oop5.basic.php>

3. Definición de una clase

- Comienza con la palabra reservada **class**, seguido de un nombre de clase, y continuando con un par de llaves que encierran las definiciones de las propiedades y métodos pertenecientes a dicha clase.
- Los nombres de clase no son case sensitive y siguen la convención **PascalCase**. Se recomienda que se utilice una sola clase por fichero y que el fichero tenga el mismo nombre de la clase.
- La pseudovariable **\$this** está disponible cuando un método es invocado dentro del contexto de un objeto.
- Consulta la URL de PHP.net para más información y ver ejemplos: <https://www.php.net/manual/es/language.oop5.basic.php>

```
<?php
class ClaseSencilla
{
    // Declaración de una propiedad
    public $var = 'un valor predeterminado';

    // Declaración de un método
    public function mostrarVar() {
        echo $this->var;
    }
}
?>
```

4. Propiedades tipadas

Las variables pertenecientes a una clase se llaman *propiedades*

A partir de PHP 7.4.0, las definiciones de propiedades pueden incluirse Declaraciones de tipo, originando **propiedades tipadas**:

```
class Shape
{
    public int $numberOfSides;
    public string $name;
}
```

Las propiedades tipadas **deben ser inicializadas antes de acceder a ellas**, de lo contrario se produce un [Error](#).

```
$circle = new Shape();

$circle->getName(); //causará Error
```

A partir de PHP 8.1.0, una **propiedad tipada** se puede declarar con el modificador **readonly** (de solo lectura), lo que impide la modificación de la propiedad después de la inicialización

```
class Test1 {
    public readonly string $prop;
}
```

Fuente: <https://www.php.net/manual/es/language.oop5.properties.php>

5. Creación de objetos

- Para crear una instancia de una clase, se suele emplear la palabra reservada **new** y a continuación el nombre de la clase.
- Se usarán paréntesis obligatoriamente si hay que pasarle parámetros (al constructor) para inicializar el objeto.

```
<?php
$instancia = new ClaseSencilla();

// Esto también se puede hacer con una variable:
$nombreClase = 'ClaseSencilla';
$instancia = new $nombreClase(); // new ClaseSencilla()
?>
```

- Un objeto se creará siempre, a menos que el objeto tenga un constructor que lance una excepción.
- Las propiedades y métodos de una clase viven en «espacios de nombres» diferentes, por tanto, es posible tener una propiedad y un método con el mismo nombre.
- Se accede con **\$objeto->campo** y con **\$objeto->método()**

6. Visibilidad

Se pueden añadir los modificadores *public*, *protected* o *private* a una propiedad, un método o una constante.

Consulta sus significados en la URL <https://www.php.net/manual/es/language.oop5.visibility.php>

- **public:** accesible desde cualquier parte
- **protected:** accesible desde la misma clase, mediante clases heredadas o desde la clase padre
- **private:** accesible solo desde la la clase que los definió.

7. Constantes

Las constantes pueden ser declaradas en una clase con diferentes tipos de visibilidad

- Si se hace referencia a las constantes dentro de la propia clase se utiliza la palabra clave `self` seguida del operador [resolución de ámbito ::](#).
- Si se hace referencia a las constantes fuera de la propia clase se utiliza el nombre de la clase seguida del operador [resolución de ámbito ::](#).

```
<?php
/**
 * Definir MiClase
 */
class MiClase
{
    // Declarar una constante pública
    public const MY_PUBLIC = 'public';

    // Declarar una constante protegida
    protected const MY_PROTECTED = 'protected';

    // Declarar una constante privada
    private const MY_PRIVATE = 'private';

    public function foo()
    {
        echo self::MY_PUBLIC;
        echo self::MY_PROTECTED;
        echo self::MY_PRIVATE;
    }
}

$myclass = new MiClase();
MiClase::MY_PUBLIC; // Funciona
MiClase::MY_PROTECTED; // Error fatal
MiClase::MY_PRIVATE; // Error fatal
$myclass->foo(); // Funcionan Public, Protected y Private
```

8. Propiedades y métodos estáticos

- Se puede aplicar a propiedades o métodos de clases
- Son métodos o propiedades compartidos por todos los objetos de la clase (no hay una propiedad/método por cada objeto)
- Son accesibles sin la necesidad de instanciar la clase. Desde dentro de la clase con `self::` y desde fuera con el `nombreClase::`
- Una propiedad declarada como `static` no puede ser accedida con un objeto de clase instanciado (aunque un método estático sí lo puede hacer).
- Si no se usa ninguna declaración de visibilidad, se tratará a las propiedades o métodos como si hubiesen sido definidos como `public`.

```
<?php
class Foo
{
    public static $mi_static = 'foo';

    public function valorStatic() {
        return self::$mi_static;
    }
}
```

Desde fuera de la clase Foo se llamaría a la función con el operador resolución de ámbito:
`Foo::$mi_static`

Ejemplo #1 Ejemplo de método estático

```
<?php
class Foo {
    public static function unMetodoEstatico() {
        // ...
    }
}

Foo::unMetodoEstatico();
```

Más información sobre static: <https://www.php.net/manual/es/language.oop5.static.php>

9. Constructores

- Las clases que tengan un método constructor lo invocarán en cada nuevo objeto creado, por lo que se utilizará para inicializarlo antes de ser usado.
- Se declara con function `__construct`, y no se debe utilizar la manera antigua de creación de constructores con el nombre de la clase.
- Para ejecutar un constructor padre desde el constructor hijo se requiere invocar a **`parent::__construct()`**.

Consulta la URL: <https://www.php.net/manual/es/language.oop5.decon.php>

10. Asignación y Comparación de objetos

El operador de **asignación** en PHP (**=**) funciona de forma distinta para objetos que para el resto de elementos (variables o arrays). Cuando se crea un nuevo objeto y se asigna a una variable, esta se añade a una tabla de símbolos que apunta al identificador del objeto, que se almacena en otra parte.

Así como la asignación (**=**) generalmente copia el valor del segundo operando en el primero, la asignación de objetos se hace añadiendo una nueva variable a la tabla de símbolos que apunta al mismo identificador que la variable anterior. En el caso de querer crear un nuevo objeto con los mismos valores que otro, hay que utilizar la palabra clave [clone](#).

```
$obj2 = clone $obj;
```

Tenéis un ejemplo de este comportamiento en el **Ejemplo #4 Asignación de objetos de la [URL](#)**

	<u>Nombre variables</u>	<u>Id objetos</u>
<u>Son alias</u>	<u>\$instancia,</u> <u>\$referencia</u>	<u>id objeto ClaseSencilla</u>
	<u>\$asignada</u>	<u>id objeto ClaseSencilla</u>

<u>Id objetos</u>	<u>Datos y métodos objetos</u>
<u>id objeto ClaseSencilla</u>	<u>\$var, mostrarValor(), etc.</u>

Al utilizar el operador de comparación (**==**), se comparan de una forma sencilla las variables de cada objeto, es decir: Dos instancias de un objeto son iguales si tienen los mismos atributos y valores, y son instancias de la misma clase.

Pero el comportamiento del operador identidad (**===**) no funciona como podría parecer si tenemos en cuenta como funciona con variables (no con referencias a objetos), que comprueba el valor y el tipo de la variable. Con objetos, las variables de un objeto son idénticas sí y sólo sí hacen referencia a la misma instancia de la misma clase.

Enlaces relacionados:

- <https://www.php.net/manual/es/language.oo5.object-comparison.php>
- <https://www.php.net/manual/es/language.oo5.references.php>

11. Herencia

- Una clase puede heredar los métodos y propiedades de otra clase empleando la palabra reservada **extends** en la declaración de la clase.
- No es posible la extensión de múltiples clases, una clase solo puede heredar de una clase base.
- Los métodos y propiedades heredados pueden ser sobrescritos con la redeclaración de éstos utilizando el mismo nombre que en la clase madre. Sin embargo, si la clase madre definió un método como **final**, éste no podrá ser sobrescrito.
- Es posible acceder a los métodos sobrescritos o a las propiedades estáticas haciendo referencia a ellos con **parent::**
- Más información: <https://www.php.net/manual/es/language.oop5.inheritance.php>

12. Autocarga de clases

Como hemos visto, lo habitual es crear un fichero por cada clase con su mismo nombre.

Para evitar tener que incluir todas las clases que se utilicen con **include** o **require** o sus versiones terminadas en **_once**, es posible habilitar la autocarga de clases.

Cada vez que se intenta inicializar una clase y la clase no existe en el código anterior, el nombre de esta clase se pasa a una función de autocarga que la buscará en los directorios que le indiquemos. Se puede usar más de una función de autocarga y formarán parte de una cola de funciones.

Se llamará a la función `spl_autoload_register` que registra una función en una cola. Una cola es una estructura de datos que se evalúa siguiendo FIFO (first in, first out), como las colas en la carnicería, por orden de llegada.

Esa cola de funciones será evaluada por la Standard PHP Library (SPL). Cada función registrada recibirá por parámetro de entrada el nombre de la clase. El cuerpo de la función se encarga de indicar el/los directorios donde se encuentran las clases y las requiere/include a partir del nombre de la clase.

```
spl_autoload_register(function ($nombre_clase) {  
    include $nombre_clase . '.php';  
});
```

Consultar la URL: <https://www.php.net/manual/es/language.oop5.autoload.php>

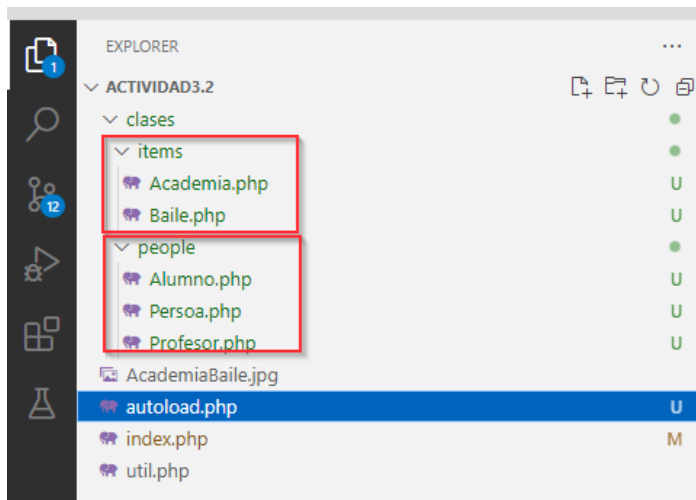
Ejemplo con autoload

https://github.com/dudwcs/Ejemplo_autoload_AcademiaBaile.git

Sobre la propuesta de solución de la Actividad 6.2 Academia de baile, se han movido las clases a los subdirectorios:

- clases/people
- clases/items

tal y como indica la siguiente imagen:



Se ha creado un fichero de `autoload.php`

En él se utiliza `spl_autoload_register` con una función anónima que va a buscar en los directorios de un array, hasta encontrar el archivo con el mismo nombre que `$nombre_clase.php`. Cuando lo encuentra lo requiere con `require_once` y sale del bucle y de la función.

Si no se encuentra el fichero, no se podrá crear el objeto de esa clase y se obtendrá un error fatal **Fatal error**: Uncaught Error: Class "X" not found

13. Espacios de nombres

Declaración de espacios de nombres

- Los espacios de nombres de PHP proporcionan una manera para agrupar clases, interfaces, funciones y constantes relacionadas (de una forma similar a los packages de Java)
- Solamente se ven afectados por espacios de nombres: clases (incluyendo abstractas y traits, que veremos en próximas secciones), interfaces, funciones y constantes. Los espacios de nombres permiten crear clases (y demás elementos antes mencionados) con el mismo nombre, pero en distintos namespaces sin originar conflictos entre ellos.
- Un fichero que contenga un espacio de nombres debe declararlo al inicio del mismo, antes que cualquier otro código con la palabra clave **namespace**, únicamente con una excepción: la palabra reservada `declare` para declarar la codificación de un fichero fuente. Además, todo lo que no sea código de PHP no puede preceder a la declaración del espacio de nombres, incluyendo espacios en blanco extra:

```
<?php
```

```
namespace MiProyecto;
```

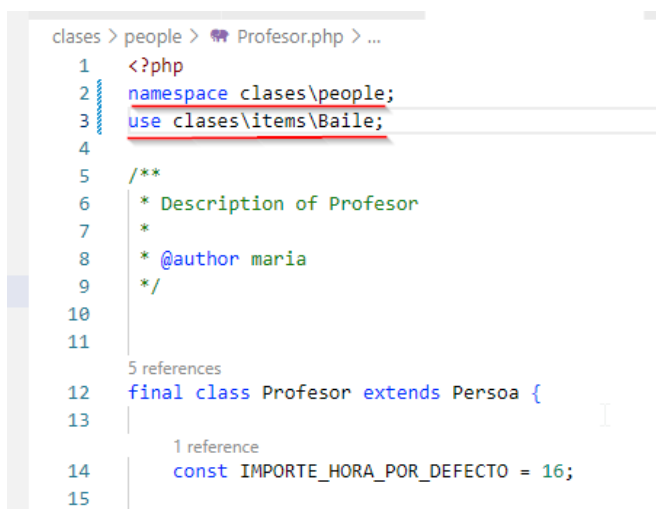
- Aunque es posible, se desaconseja completamente, como práctica de código, la combinación de varios espacios de nombres en un mismo fichero.
- Un espacio de nombres se puede definir con subniveles: `namespace MiProyecto\Sub\Nivel;`
- Por convención, se suelen hacer corresponder los niveles con la jerarquía de subdirectorios.
- Para acceder a cualquier clase, función o constante **globales**, se puede utilizar un nombre completamente cualificado con una barra invertida inicial, como `\strlen()` o `\Exception` o `\INI_ALL`.
- Se puede hacer referencia a un nombre de una clase de tres maneras: <https://www.php.net/manual/es/language.namespaces.basics.php>
 - **Nombre no cualificado**, o nombre de clase sin prefijo. Por ejemplo: `Persoa`, se interpreta en función del namespace del fichero actual. Con
`namespace clases\items;`

se resolverá a `clases\items\Persoa`

- **Nombre cualificado**, o un nombre de clase con prefijo `people\Persoa`, se interpreta en función del namespace del fichero actual. Con
`namespace clases\items;`
se resolverá a `clases\items\people\Persoa`
- **Nombre completamente cualificado**, comenzando por barra invertida `\clases\people\Persoa`. No importa el namespace del fichero actual.
- El valor de la constante mágica `__NAMESPACE__` es una cadena que contiene el nombre del espacio de nombres actual. En código global, que no es de espacio de nombres, contiene una cadena vacía.
- La palabra reservada `namespace` se puede utilizar para solicitar explícitamente un elemento **del espacio de nombres o subespacio de nombres actual**. Es el equivalente del operador `self` de las clases para espacios de nombres

Importación de espacios de nombres

- Para usar clases que pertenecen a un namespace diferente al actual se debe usar el operador **use**:



```

clases > people > Profesor.php > ...
1  <?php
2  namespace clases\people;
3  use clases\items\Baile;
4
5  /**
6   * Description of Profesor
7   *
8   * @author maria
9   */
10
11
12  5 references
13  final class Profesor extends Persoa {
14
15      1 reference
16      const IMPORTE_HORA_POR_DEFECTO = 16;

```

- `use clases\items\Academia;` //y ahora podremos usar `$obj = new Academia();`
- Si no se usa `use clases\items\Academia;` habría que crear `$obj = new \clases\items\Academia();` usando el namespace completamente cualificado.
- Al contrario de lo que ocurre en Java con los packages `import java.util.*;` no se pueden importar todas las clases de un paquete con `.*;`, pero a partir de PHP 7+ se permiten agrupaciones de clases, funciones y constantes:

- `use un\espacioDeNombres\{ClaseA, ClaseB, ClaseC};`
`use function un\espacioDeNombres\{fn_a, fn_b, fn_c};`
`use const un\espacioDeNombres\{ConstA, ConstB, ConstC};`
- Es posible utilizar un alias para una clase o función:
`use clases\items\Baile as Dance;`

`public function engadir(Dance $baile): bool {...}`
- Es posible agrupar varios import en un único operador use, separados por comas:
`use clases\people\Profesor, clases\people\Alumno;`
- Ver también **[Uso de los espacios de nombres: apodar/importar](#)**

<https://www.php.net/manual/es/language.namespaces.php>

14. Interfaces

- Las interfaces definen qué métodos deben ser implementados por una clase.
- No aportan la implementación concreta de los métodos, solo indican el nombre de métodos y sus parámetros.
- Todos los métodos de la interfaz deben ser públicos
- Se definen de la misma manera que una clase, aunque reemplazando la palabra reservada `class` por la palabra reservada `interface`
- La clase que implementa una interfaz debe usar la palabra clave `implements`
- Se puede ver un ejemplo de interfaz e implementación en <https://www.php.net/manual/es/language.oop5.interfaces.php>
- Las interfaces se pueden extender al igual que las clases utilizando el operador `extends`.
- Es posible tener constantes dentro de las interfaces. Se accede con el operador de resolución de ámbito:: (Ejemplo #4 Interfaces con constantes)

¿Para qué se usan las interfaces?

-Para establecer un **contrato de comunicación entre dos partes** sin depender de la implementación exacta. Por ejemplo, dos equipos que desarrollan un front-end y un back-end se ponen de acuerdo en los métodos que proveerá el back-end y que consumirá el front-end estableciendo una interfaz. De esta forma, pueden desarrollar en paralelo el software sabiendo de antemano la firma de los métodos que comunicará ambas partes.

-Para favorecer la **interoperabilidad**. Por ejemplo, PDO define una interfaz conceptual ([no es una interfaz en PHP](#), sino una clase con métodos vacíos que se puede extender, pero la idea es básicamente la misma) para poder acceder a bases de datos en PHP. PDO proporciona una capa de abstracción de *acceso a datos*, lo que significa que, independientemente de la base de datos que se esté utilizando, se emplean las mismas funciones para realizar consultas y obtener datos. Teóricamente, si se usa PDO con MySQL, se podría cambiar el Sistema Gestor de Base de Datos a SQL Server, sin afectar a las capas software no relacionadas con la BD. La implementación de la interfaz la proporciona cada fabricante con su driver específico (la implementación de acceso al SGBD).

-Agrupar comportamiento que podría tener cualquier clase: Serialización en formato JSON (o cualquier otro formato)

15. Clases abstractas

- Las clases definidas como abstractas no se pueden instanciar. No podemos hacer un `new ClaseAbstracta()`;
- Cualquier clase que contiene al menos un método abstracto debe ser definida como clase abstracta
- Los métodos definidos como abstractos simplemente declaran la firma del método, pero no pueden definir la implementación.
- Cuando se hereda de una clase abstracta, todos los métodos definidos como abstractos en la declaración de la clase madre deben ser definidos en la clase hija
- Las firmas de los métodos tienen que coincidir, es decir, la declaración de tipos y el número de argumentos **requeridos** deben ser los mismos. Por ejemplo, si la clase derivada define un argumento opcional y la firma del método abstracto no lo hace, no habría conflicto con la firma.
- Podemos ver ejemplos en la documentación: <https://www.php.net/manual/es/language.oop5.abstract.php>

Diferencias entre clases abstractas e interfaces

- En las clases abstractas, los métodos pueden contener código. Si van a existir varias subclases con un comportamiento común, se podría programar en los métodos de la clase abstracta. Si se opta por una interfaz, habría que repetir el código en todas las clases que lo implemente.
- Las clases abstractas pueden contener atributos, y las interfaces no.
- No se puede crear una clase que herede de dos clases abstractas, pero sí se puede crear una clase que implemente varias interfaces.

16. Traits o rasgos

- Mecanismo de reutilización de código en lenguajes de herencia simple (una clase hija no puede extender de más de una clase). Permite imitar la herencia múltiple.
- Un trait es similar a una clase, pero solo agrupa funcionalidades muy específicas de forma coherente

```
trait DecirMundo {
    public function decirHola() {
        echo 'Mundo!';
    }
}
```

- No se puede instanciar directamente un Trait
- Para insertar un Trait en una clase, se usa `use Nombre_Trait1`; Es posible insertar más de un trait: `use Nombre_Trait1, Nombre_Trait2`;

```
class Base {
    use DecirMundo;
    public function decirHola() {
        echo '¡Hola ';
    }
}
```

- Un objeto de una clase que usa un trait, puede llamar a los métodos del trait como si fueran definidos en la propia clase con ->
- En caso de usar métodos homónimos en un Trait y en la clase que lo usa, existe un orden de precedencia: los métodos de la clase actual sobrescriben los métodos del Trait, a la vez que el Trait sobrescribe los métodos de la clase base (padre).

```
$object = new Base();
$object->decirHola(); // Mostrará ¡Hola
```

- Si dentro de una clase, se importan dos Traits y tienen métodos homónimos, se produce un error fatal salvo que se resuelva el conflicto con `insteadof`
- Es posible cambiar la visibilidad de un Trait dentro de la clase que lo inserta con `as`
- Un Trait puede hacer uso de otros Traits
- Un Trait puede tener métodos abstractos e impone que se reescriban en la clase que los inserta
- Un Trait puede tener miembros y métodos static. (Serán estáticos a nivel de la clase, no del trait)
- Un Trait puede definir propiedades. La clase que lo inserte y el Trait pueden tener propiedades homónimas si tienen misma visibilidad y valor inicial. Si no, se producirá un error fatal.

Es posible consultar la documentación de PHP sobre Traits en la URL: <https://www.php.net/manual/es/language.oop5.traits.php>

17. Funciones para clases y objetos

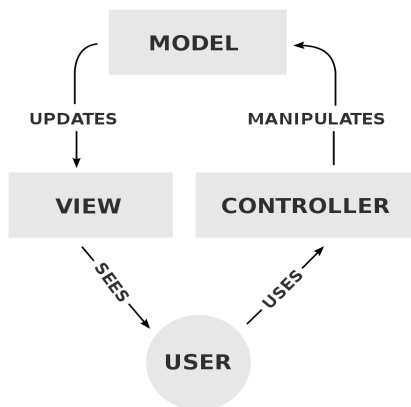
<https://www.php.net/manual/es/ref.classobj.php>

[instanceof](#)

18. Patrón de arquitectura Model-View-Controller (MVC)

Modelo-vista-controlador (MVC) es un patrón de arquitectura de software. MVC propone la construcción de tres componentes distintos que son el **modelo**, la **vista** y el **controlador**

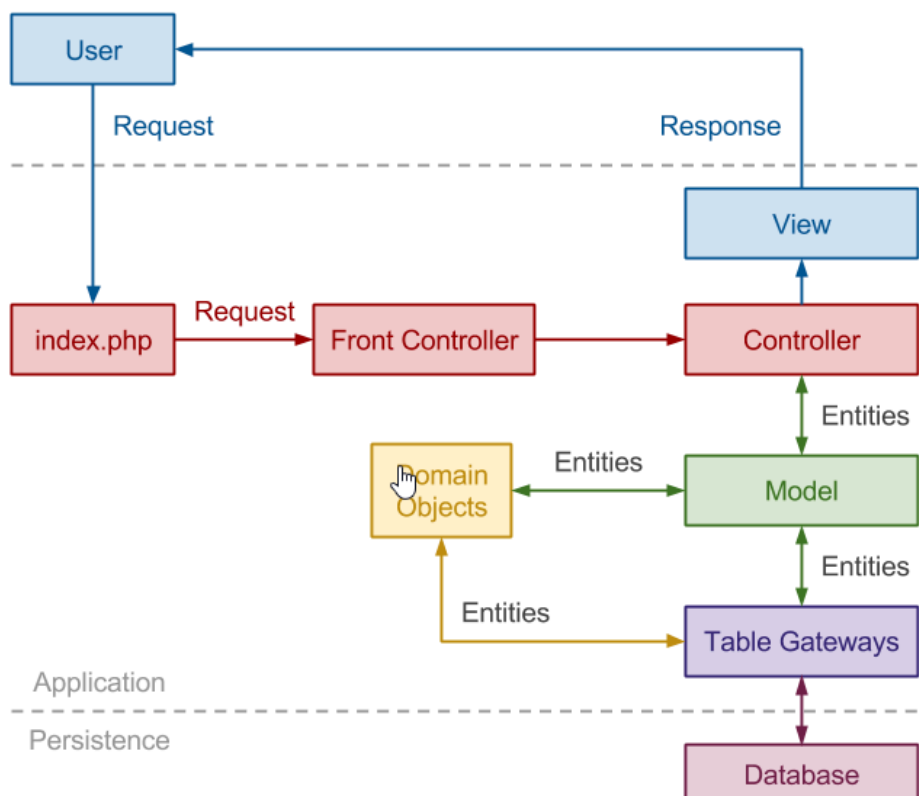
- El **Modelo**: Representa la información o los datos que maneja la aplicación. Gestiona todos los accesos a dicha información, tanto consultas como actualizaciones. Las peticiones de acceso o manipulación de información llegan al **modelo** a través del **controlador**. Si la aplicación utiliza algún tipo de almacenamiento para su información (como un SGBD), tendrá que encargarse de almacenarla y recuperarla.
- El **Controlador**: Responde a eventos o acciones del usuario e invoca peticiones al **modelo** cuando se hace alguna solicitud sobre la información (por ejemplo, editar un documento o un registro en una base de datos). El **controlador** hace de intermediario entre la **vista** y el **modelo**.
- La **Vista**: Presenta el **modelo** (información y *lógica de negocio*) en un formato adecuado para interactuar con el usuario. En esta parte se encuentra el código necesario para generar el interfaz de usuario (en nuestro caso en HTML), según la información obtenida del modelo.



Fuente: <https://upload.wikimedia.org/wikipedia/commons/thumb/a/a0/MVC-Process.svg/931px-MVC-Process.svg.png>

En aplicaciones web también es habitual el uso de un FrontController, un controlador que recibe todas las peticiones HTTP y en función de la URL, es capaz de redirigir una petición a uno u otro controlador. Podéis ver una imagen del funcionamiento de un FrontController en la URL:

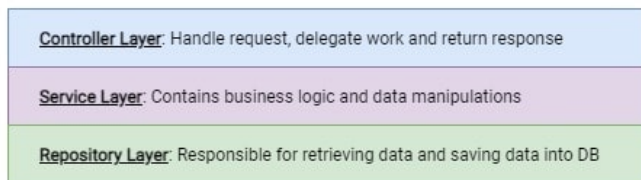
<https://github.com/joefallon/KissMVC>



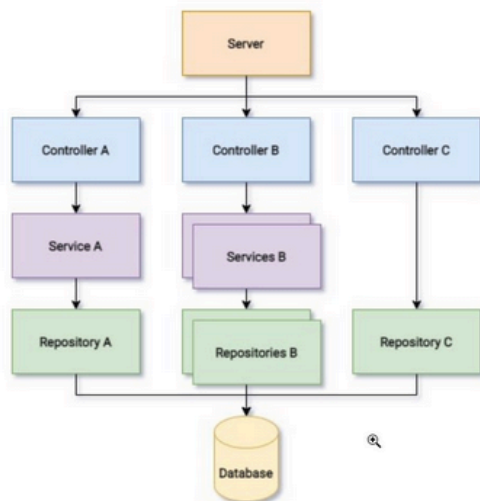
La imagen superior no es más que un ejemplo de una posible implementación del patrón MVC. Nosotros vamos a ver una implementación similar con el siguiente comportamiento:

- Todas las peticiones llegan al **FrontController**. Este, en función de la URL sabrá a qué controlador específico enviar la petición y qué método de ese controlador invocar. El controlador se comunica con el modelo para obtener datos o para insertar/actualizar/borrar datos (operaciones CRUD). El controlador se comunica con las vistas para enviarle los datos devueltos por el modelo o para recuperar los datos que introduce el usuario y que serán enviados al modelo. Normalmente existe un controlador específico por cada entidad susceptible de realizar operaciones CRUD: UsuarioController, ProductoController, etc.
- La **vista** se encargará de mostrar y recoger los datos introducidos por el usuario. Si tiene algo de lógica será siempre relacionada con cómo se mostrará la información.
- El **modelo** trabajará con entidades (son objetos persistentes, es decir, que tienen correspondencia en el sistema de almacenamiento utilizado, normalmente bases de datos, pero también podrían ser ficheros). El modelo será el encargado de realizar las operaciones CRUD contra el sistema de persistencia y creará los objetos en memoria que podrá enviar al controlador o recibir los objetos del propio controlador. En el modelo residirá también la complejidad de procesamiento que requiera nuestra aplicación (la lógica del dominio o del negocio). En nuestra implementación de MVC, veremos que el modelo se va a dividir, a su vez, en 2 capas:
 - **Capa de Repositorios:** Utiliza el [patrón Repository](#) que se encarga de abstraer el sistema de persistencia y media entre el dominio y las capas de mapeo de datos.
 - **Capa de servicios:** Clases que contienen un método por cada caso de uso funcional de la aplicación. Habrá una clase servicio siguiendo criterios coherentes. Por ejemplo, UserService (para login, listar usuarios, operaciones CRUD, etc), ProductService (para gestionar stock con operaciones CRUD), ClientService, etc. La capa de servicios será invocada por el controlador y, la capa de servicios será quien invoque a los métodos de la capa repositorio

Esta división por capas en el modelo de las arquitecturas web MVC es bastante frecuente, independientemente de la tecnología en Java (SpringBoot, SpringMVC) o en PHP (Laravel, Symfony):



Fuente: <https://dev.to/blindkai/backend-layered-architecture-514h>



Fuente: <https://dev.to/blindkai/backend-layered-architecture-514h>

18.1. Ejemplo básico de MVC con división en capas

Vamos a ver un ejemplo de implementación de MVC muy sencillo, que está basado en la URL: <https://www.adaweb.es/modelo-vista-controlador-mvc-en-php-actualizado-2022/> sobre el que haremos algunas modificaciones para que, **de momento se almacenen los datos en ficheros JSON**.

Este ejemplo está disponible en el repositorio <https://github.com/dudwcs/Ejemplo-b-sico-mvc-con-json.git>