

Módulo 4. Funcionalidades para bases de datos y formularios

Introducción

En este módulo, trataremos uno de los temas más importantes de la programación: la programación orientada a objetos. La trabajaremos en relación con ejemplos y usos para una base de datos.

Los sistemas informáticos se basan en la recopilación de datos que, al ser procesados, devuelven información vital para los usuarios. Por ejemplo, podemos crear estadísticas como la edad promedio de los usuarios que acuden a nuestro sistema web, información que luego nos servirá, por ejemplo, para el envío de un *newsletter* (correos electrónicos que enviamos para promocionar nuestros productos o servicios) o para conocer la hora de mayor tráfico en nuestra red del sistema. Por supuesto, los datos que se recopilen y la información que se genere dependerán de cada necesidad.

La utilización de clases para guardar, actualizar, eliminar y manipular los datos del sistema de bases de datos nos permitirá reutilizar el código, teniendo toda la lógica de conexión centrada en un único archivo al cual accederemos en todo el sistema.

Además de lo mencionado, incursionamos en el envío de correos electrónicos (por ejemplo, para *newsletters*), viendo las diferentes opciones disponibles en este lenguaje.

Video de inmersión

Unidad 1. Clases y bases de datos

Una de las principales ventajas de los sistemas es poder guardar, modificar y obtener datos, ya sea mediante el uso de bases de datos (ampliamente difundido en los sistemas actuales) o mediante *webs services* (sistemas a los que podemos conectar nuestro desarrollo con otros sistemas o diferentes dispositivos).

Las bases de datos, como su palabra refiere, son un almacén de datos que pertenece a un

el mismo contexto (o sistema), aunque un sistema puede requerir varias bases de datos —de las cuales algunas pueden ser compartidas por otros sistemas—. Este es el caso de los webs services, que almacenan sus datos en una base de datos, de forma tal que, al ser consultados, nos retornan la información o los datos solicitados.

Con los fines prácticos de esta materia, vamos a considerar que los datos, a su vez, se dispondrán en tablas, que tienen sus propiedades y representan una entidad (datos afines) o vinculación de entidades.

Por ejemplo, en un sistema, podremos tener las entidades, usuarios, productos, categoría (de los productos). Entonces, tendremos estas tres tablas con las propiedades específicas de cada entidad.

A su vez, las tablas pueden estar relacionadas entre sí. En el ejemplo citado, cada producto pertenece a una categoría; por lo tanto, debe existir una vinculación entre ambas tablas.

Dentro de los sistemas de bases de datos, contamos con diferentes motores. Muchos de ellos trabajan con lenguajes similares (SQL), y existen motores gratuitos o pagos.

Los más difundidos entre los gratuitos son los siguientes:

- MySQL (el que utilizaremos durante todo nuestro desarrollo).
- PostgreSQL.
- SQLite.
- MongoDB.

Entre los pagos, podemos contar con los siguientes:

- Microsoft SQL Server.
- Microsoft Access.

PHP permite realizar conexiones para cualquiera de estos motores; ahora bien, al contratar un servidor para nuestros sistemas, mayormente vienen con MySQL, por lo cual es el que más se difunde actualmente en sistemas webs.

Los webs services, que también utilizan bases de datos para recuperar información, se llaman

desde una URL común.

Muchas veces, crearemos nuestros propios *webs services*. Imaginemos, por ejemplo, que tenemos nuestro sistema web y también una aplicación de celular. Esta última no puede tener una base de datos para grandes volúmenes de información y, por este motivo, recurriremos a esta técnica.

En este sentido, en nuestro sistema web, podremos crear un *web service* que se consumirá por la *app*. Esta hace una llamada al servidor indicando ciertos parámetros, nuestro servidor lo procesa y le retorna la información solicitada.

Existen empresas que desarrollan sus propios *webs services* (generalmente pagos), en los que se puede consultar, entre otras cosas, datos de la temperatura actual, cotizaciones de monedas, etc.

Pregunta multiple choice

¿Por qué debemos crear clases en nuestro sistema?

- Podemos reutilizar código, sin la necesidad de que debamos reescribir muchas funcionalidades en nuestro sistema.
- Ayuda a la lectura de código, ya que queda más prolíjo y entendible a todos los programadores.
- Nuestro sistema se ejecuta de manera más rápida, dado que está cargado en memoria para todo el proyecto.
- No siempre nos conviene utilizar clases, porque su recarga nos produce un error y el sistema se detiene en ejecución.

Justificación

Tema 1. Archivos de proyecto (*include* y *require*)

Los archivos de un proyecto son propiamente archivos que se alojan dentro del disco duro del servidor, en los que escribiremos nuestro código. Podemos tener archivos con diferentes extensiones. A saber, los siguientes:

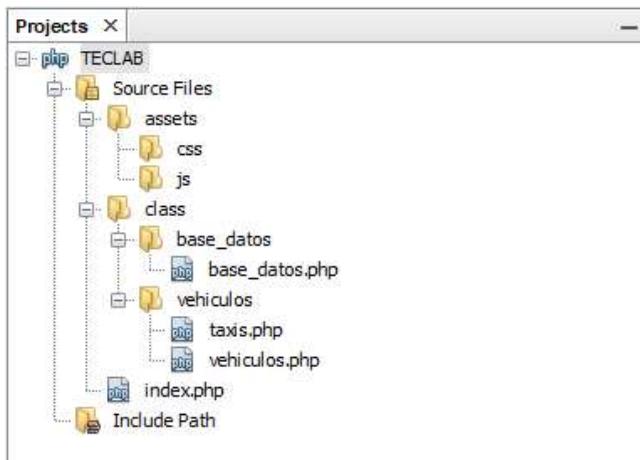
Infografía 1: Archivos de proyecto

Fuente: elaboración propia

Recordemos nuestra clase de acceso a bases de datos (`base_datos`) abordada en los módulos anteriores. Se sugiere que las clases que se vayan creando en nuestro sistema se almacenen siempre en archivos diferentes. Supongamos que tenemos nuestra clase `base_datos`, otra llamada «Vehículos», otra llamada «Taxis».

El estándar en programación nos sugiere mantener una estructura de proyecto, como la que se observa en la siguiente figura:

Figura 1: Ejemplo de estructura de proyecto



Fuente: elaboración propia

Como puede observarse, las carpetas principales son `assets` (que podemos considerar los bienes o activos de nuestro sistema), en los que guardaremos los archivos de estilos y `javascripts`. También, de ser necesario, se guardarán imágenes y demás recursos visuales que lleguemos a necesitar.

Luego, tenemos la carpeta «Class», que a su vez se divide en tantas carpetas como clases afines podamos tener. En cada archivo dentro de «Class», solo debe desarrollarse una única clase para facilitar el mantenimiento. Un ejemplo es que, si quisiéramos modificar la clase «Taxis», podemos identificarla solamente buscando el propio archivo.

Luego, para la ejecución del proyecto, podemos tener, en la base de directorios, un archivo llamado `index.php` (en el que arranca nuestro sistema).

Ahora bien, la pregunta radica en cómo incluir nuestras clases dentro de este archivo, para poder interactuar con ellas.

La idea de incluir archivos, es decir, de ser llamados y cargados desde otros, es lograr la

reutilización de código, ya que tenemos un archivo con funcionalidades que podemos reutilizar desde varios otros.

PHP cuenta con las sentencias *include* y *require*.

En index.php, podemos tener cualquiera de las siguientes líneas, pero solo una de ellas, puesto que, al incluir dos veces un archivo de clase, el sistema genera un error y se termina la ejecución del programa. La regla general —y muy importante— es que la carga de un archivo de clases solo puede realizarse una vez en el ciclo de vida del sistema. La misma regla puede aplicarse a nunca tener dos clases con el mismo nombre:

```
include './class/base_datos/base_datos.php';
```

allí /class/base_datos/base_datos.php es la ruta al archivo de la clase base de datos.

Otra opción sería con lo siguiente:

```
require './class/base_datos/base_datos.php';
```

Infografía 2: Diferencia entre include y require

Fuente: elaboración propia

Esto puede hacernos definir que utilizamos uno u otro según la importancia que tenga el archivo a incluir.

Por ejemplo, supongamos que queremos agregar un archivo, pero que no es crítico para la resolución del sistema; utilizaremos include, ya que no termina la ejecución del sistema. Ahora bien, si el archivo es crítico, utilizaremos require.

Figura 2: Código

```
require './class/base_datos/base_datos.php';

include './class/mostrar_imagen.php';
```

Fuente: elaboración propia

En el ejemplo anterior, es vital para el sistema cargar la clase base_datos, ya que sin ella nuestro sistema perdería toda utilidad.

Ahora bien, no es tan necesario cargar la clase `mostrar_imagen` (una clase de ayuda que permite mostrar ciertas imágenes en el navegador), puesto que, a lo sumo, no veríamos del todo correcto el desarrollo en el navegador.

Tema 2. Autocarga de clases (autoload)

Ahora bien, supongamos que, erróneamente, intentamos cargar dos veces la misma clase; el sistema nos mostrará un *fatal error*, y se terminará completamente su ejecución.

PHP define una función como mejor práctica para la carga de clases. Esta función es `spl_autoload_register()`. Permite tener, en un solo archivo, la definición de todas las clases que utilizaremos durante el sistema.

Esta funcionalidad revisa automáticamente que la clase no se haya cargado nunca. De ocurrir esto, carga la clase correspondiente a utilizar, caso contrario, directamente ignora la carga.

Valiosamente, nos quita la necesidad de agregar infinitos `includes` a cada archivo de clase necesario. Solo debemos incluir la clase de autocarga (un solo `include`) como veremos en el siguiente ejemplo.

Veamos cómo la ejecutaremos en nuestro código.

Primeramente, es conveniente tener en la base del directorio «Class» un archivo llamado `autocarga.php` y que este sea una clase:

Figura 3: Código

```
<?php

class autocarga {

    static public function cargar_clase($clase) {
        $arrayClases = array();
        $arrayClases['base_datos'] = './class/base_datos/base_datos.php';
        $arrayClases['vehiculos'] = './class/vehiculos/vehiculos.php';
        $arrayClases['taxis'] = './class/vehiculos/taxis.php';

        if (isset($arrayClases[$clase])) {
            if (file_exists($arrayClases[$clase])) {
                include $arrayClases[$clase];
            } else {
                throw new Exception("Archivo de clase no encontrada [{$_arrayClases[$clase]}]");
            }
        }
    }

    spl_autoload_register('autocarga::cargar_clase');
}
```

Fuente: elaboración propia

Lo que iremos realizando es completar la clase como índice del array `$arrClases` y le asignamos su *path* (ubicación del archivo) en el sistema.

Luego, lo único que necesitamos es agregar en index.php una sola línea:

```
include './class/autocarga.php';
```

Esta línea realizará la autocarga de la clase (el propio include a ella) a medida que la vayamos necesitando.

Veamos algunas consideraciones importantes:

- Con autocarga, nunca podremos cargar dos veces la misma clase, ya que PHP entiende si se ha cargado con antelación, y no volverá a hacerlo.
- Las clases se autocargan en el preciso momento en el que las necesitamos; esto es que, si tenemos clases que nunca se utilizan en el sistema, nunca se cargarán, lo cual evita la sobrecarga de la memoria del servidor.

Tema 3. Relacionando objetos con la base de datos

Objetos instanciados desde un registro de una tabla en la base de datos

Imaginemos nuestra clase anterior «Empleados». Podemos instanciar (concepto que refiere a crear un objeto de la clase) perfectamente un empleado desde la base de datos (llamando a la clase «Empleados»).

Utilizando nuestra clase común (base_datos.php), podemos acceder a la tabla y buscar un registro por su ID. En este ejemplo, también sería válido buscarlo por su e-mail, siempre que hayamos definido que el e-mail es único por empleado.

Veamos lo que se presenta a continuación y saquemos algunas conclusiones:

Figura 4: Código

```

<?php

class empleados {

    protected $id;
    public $nombre;
    public $apellido;
    public $fecha_nacimiento;
    public $email;
    public $domicilio;
    private $_exists = false;

    function __construct($id) {
        $db = new base_datos("mysql", "test", "127.0.0.1", "aquiles248", "aquiles248");
        $resp = $db->select("empleados", "id=?", array($id));
        if (count($resp) == 1){
            $this->_exists = true;
            $this->id = $resp[0]['id'];
            $this->nombre = $resp[0]['nombre'];
            $this->apellido = $resp[0]['apellido'];
            $this->fecha_nacimiento = $resp[0]['fecha_nacimiento'];
            $this->domicilio = $resp[0]['domicilio'];
            $this->email = $resp[0]['email'];
        }
    }

    public function mostrar(){
        echo "<pre>";
        print_r($this);
        echo "</pre>";
    }
}

```

Fuente: elaboración propia

En la clase «Personas», definimos un constructor como hemos visto en el módulo anterior (lectura 2, módulo 2. 2. 4.). Este constructor admite un parámetro que, en este caso, es el ID (identificador dentro de la tabla).

Además, definimos todas sus propiedades junto a una extra privada llamada `$_exists` (que veremos un poco más adelante).

Realizamos la conexión a la base de datos con nuestra clase creada (`base_datos.php`), y ejecutamos una consulta de selección para buscar el registro perteneciente a ese identificador (`$id`).

Como hemos visto, esta consulta retorna un `array` con todos los elementos encontrados, pero ahora bien, sabemos de antemano que un identificador único solo retornará un único registro y es la razón por la cual instanciamos todas las propiedades con el índice primario (índice primario responde a la posición cero del `array`) `$resp[0]`:

Al finalizar la clase, creamos una función pública (función que es visible o accesible desde cualquier parte del proyecto) para mostrar la correcta instancia del objeto desde la base de datos.

Nuestro archivo `index.php` ahora queda de esta forma:

Figura 5: Código

```

<?php

include './class/autocarga.php';

$empleado = new empleados(1);

$empleado->mostrar();|

```

Fuente: elaboración propia

Utilizamos la función autocarga, como hemos convenido por ser una de las mejores prácticas. Luego, instanciamos el objeto «Empleados», cuyo ID es 1, y la salida en el navegador debe ser la siguiente:

Figura 6: Código

```

empleados Object
(
    [id:protected] => 1
    [nombre] => Pedro
    [apellido] => Picapiedras
    [fecha_nacimiento] => 1974-12-10
    [email] => pedro@picapiedras.com
    [domicilio] => Rocallosas 50
    [_exists:empleados:private] => 1
)

```

Fuente: elaboración propia

Ahora bien, veamos la funcionalidad de la propiedad `$_exists` antes de continuar.

¿Qué pasaría si intentáramos instanciar un objeto que en la base de datos no existe? (Supongamos que al ID 2 no lo tenemos registrado).

En nuestro archivo index.php, tendríamos lo siguiente:

Figura 7: Código

```

include './class/autocarga.php';

$empleado = new empleados(2);

$empleado->mostrar();

```

Fuente: elaboración propia

Y la salida en el navegador será la siguiente:

Figura 8: Código

```

empleados Object
(
    [id:protected] =>
    [nombre] =>
    [apellido] =>
    [fecha_nacimiento] =>
    [email] =>
    [domicilio] =>
    [_exists:empleados:private] =>
)

```

Fuente: elaboración propia

Es decir, el objeto se crea vacío y, además, tenemos la variable `$_exists` como vacío (FALSE).

Quiere decir que hemos creado un objeto vacío que no existe en la base de datos. Al momento de realizar su guardado, se agrega un nuevo registro en nuestra tabla, puesto que entendemos que no existía anteriormente y que no hay forma de actualizarlo, solo de crearlo.

Objetos íntegramente relacionados con la base de datos

Ahora bien, sabiendo el valor de la variable `$_exists`, podemos anticipar si debemos realizar un *insert* en la tabla (`$_exists == false`) o bien hacer un *update* (`$_exists == true`).

Podemos ampliar nuestra clase «Empleados» con el siguiente código:

Figura 9: Código

```

public function guardar(){
    if ($this->$_exists){
        return $this->actualizar();
    } else {
        return $this->insertar();
    }
}

public function eliminar(){
    $db = new base_datos("mysql", "test", "127.0.0.1", "aquiles248", "aquiles248");
    return $db->delete("empleados", "id = ".$this->id);
}

private function insertar(){
    $db = new base_datos("mysql", "test", "127.0.0.1", "aquiles248", "aquiles248");
    return $db->insert("empleados", "nombre=?, apellido=?, fecha_nacimiento=?, email=?, domicilio=?",
        null, array(0 => $this->nombre, 1 => $this->apellido,
        2 => $this->fecha_nacimiento, 4 => $this->email, 5 => $this->domicilio));
}

private function actualizar(){
    $db = new base_datos("mysql", "test", "127.0.0.1", "aquiles248", "aquiles248");
    return $db->update("empleados", "nombre=?, apellido=?, fecha_nacimiento=?, email=?, domicilio=?",
        null, array("id = ".$this->id), array(0 => $this->nombre, 1 => $this->apellido,
        2 => $this->fecha_nacimiento, 4 => $this->email, 5 => $this->domicilio));
}

```

Fuente: elaboración propia

Vemos que tenemos un sistema de guardado automático, esto es, que puede definir si hacer un *insert* o un *update*.

Con guardado automático, nos referimos a que el objeto decide si debe crear un nuevo registro en la tabla o actualizarse.

Como mencionamos anteriormente, al momento de instanciarlo, le asignamos un valor a la propiedad `$_exists`, que nos indica si el objeto es nuevo (agregaremos el registro en la base de datos) o si el objeto ya se encontraba registrado (lo actualizaremos).

Además, hemos agregado la función «Eliminar».

Veamos el porqué de la visibilidad de cada función nueva.

Hemos definido «Insertar» y «Actualizar» como privada, es decir, que no podemos llamarla desde el sistema, solo desde el objeto, ya que es él quien tiene que definir si inserta o actualiza el registro (según la propiedad `$_exists`).

«Guardar» es pública, visible desde cualquier parte del sistema, puesto que es el propio sistema quien debe decidir en qué momento debe guardarse (por ejemplo, por alguna acción realizada por el usuario).

Lo mismo sucede con la función «Eliminar», que hemos declarado pública.

Unidad 2. Funciones para formularios HTML

Tema 1. Funciones de formulario en HTML

PHP, junto con el lenguaje HTML, nos permiten realizar formularios webs. Veremos, en este capítulo, cómo estos lenguajes nos permiten realizar los formularios, lo cual permite el ingreso de datos de usuarios, las validaciones y los diferentes métodos disponibles para su creación.

Por formularios web, podemos imaginarnos las famosas páginas de contacto que muchas veces nos encontramos al navegar por diferentes sitios, como, por ejemplo, <https://www.grupoorono.com.ar/contacto>, donde se nos solicita nuestro nombre, e-mail, teléfono y los motivos de nuestra consulta.

Otros formularios pueden pedirnos solo nuestro e-mail para el registro a un *newsletter* (novedades del sitio que nos llegarán por e-mail), o diferentes datos según las necesidades del sistema que estemos utilizando.

Los formularios son la herramienta que nos permite interactuar con los usuarios. Por ejemplo, en

una web de venta de productos, los formularios representan el punto en el que pedimos a los usuarios sus datos, sus métodos de pago, la forma de envío.

Por este motivo, una buena comprensión de esto nos llevará a tener páginas más amigables tanto para el cliente como para el registro de estos datos en el sistema.

Un ejemplo de formularios complejos se da en las famosas páginas de compra online (el punto de entrada para una venta de productos o servicios). En estas páginas, luego de seleccionar el producto que queremos comprar, nos muestra un paso a paso de los formularios.

Por ejemplo, podemos tener un primer formulario en el que se nos pide nuestros datos personales; en un segundo paso, aparece otro formulario en el que se nos pide el método que vamos a utilizar como medio de pago. De acuerdo con el método seleccionado, se nos presenta otro formulario (por ejemplo, si seleccionamos método de pago con tarjeta de crédito, se nos pedirán los datos de la tarjeta). Por último, podemos tener un formulario en el que se nos pide que seleccionemos la forma de envío.

Tema 2. Cómo acceder y validar formularios (métodos get y post/accediendo desde PHP/variables \$_POST, \$_GET y \$_REQUEST)

Veamos con un ejemplo el desarrollo de un formulario. En este caso, el formulario es un contacto de los clientes con nuestra empresa (se conoce como formulario de contacto). En él, pediremos nombre y apellido, *e-mail*, número de teléfono y el mensaje que nos quiere enviar:

Figura 10: Código

```
<!DOCTYPE html>
<html lang="es">
    <head>
        <title>Formulario de contactos</title>
        <meta name="Contactos" content="Contactos">
        <meta charset="utf-8">
        <link rel="stylesheet" type="text/css" href="assets/css/main.css">
    </head>
    <body>
        <div class="formulario">
            <h1>Formulario de Contactos</h1>
            <form name="contactos" method="POST">
                <label for="nombre">Nombre</label>
                <input type="text" name="nombre" id="nombre" required="">
                <label for="email">Email</label>
                <input type="email" name="email" id="email" required="">
                <label for="telefono">Teléfono</label>
                <input type="text" name="telefono" id="telefono" required="">
                <label for="mensaje">Mensaje</label>
                <textarea name="mensaje" id="mensaje" required=""></textarea>
                <div class="boton">
                    <button class="boton-enviar" type="submit">Enviar</button>
                </div>
            </form>
        </div>
    </body>
</html>
```

En la figura 11, se puede observar la salida del navegador.

Figura 11: Salida del navegador

La imagen muestra un formulario titulado "Formulario de Contactos". El formulario consiste en los siguientes campos:

- Campo de texto para "Nombre".
- Campo de texto para "Email".
- Campo de texto para "Teléfono".
- Campo de texto grande para "Mensaje".
- Botón verde con el texto "Enviar" en la parte inferior derecha.

Fuente: elaboración propia

Veamos las propiedades de los componentes que se utilizaron hasta aquí:

```
<form name="contactos" method="POST">
```

form nos indica que estamos construyendo un formulario, es decir, un espacio en el que pediremos datos al usuario. Este elemento tiene una propiedad llamada **method** que indica la forma en la que desde el navegador web se enviarán los datos al servidor.

Los valores posibles de **method** son los siguientes:

- GET: indica que los datos del usuario viajarán al servidor en la propia URL. Utilizar este método es muy desaconsejable, ya que los datos del usuario, que pueden ser muy sensibles en muchas ocasiones, viajan visibles por Internet. Por otra parte, según la cantidad de datos que pidamos, esta URL se transforma en una ruta muy grande (muchos caracteres), y debemos considerar que los servidores mayormente se configuran para cantidad limitada de caracteres.

Veamos qué sucede con la URL al enviar el formulario, para comprender mejor este concepto.

Antes de completar el formulario, tenemos la URL del formulario de la siguiente manera:

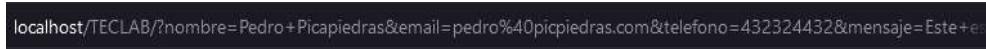
Figura 12: URL antes de completar el formulario



Fuente: elaboración propia

Al completar el formulario (con método get) y enviarlo, la URL se transforma en la siguiente:

Figura 13: URL después de completar el formulario



localhost/TECLAB/?nombre=Pedro+Picapiedras&email=pedro%40picapiedras.com&telefono=432324432&mensaje=Este+es+un+mensaje+de+prueba

Fuente: elaboración propia.

Como podemos ver, el nombre, *e-mail*, teléfono y mensaje forman parte de la URL.

A estas partes, las conocemos como parámetros; luego, veremos cómo se recuperan en el servidor.

- POST: el más difundido y utilizable. En este método, los datos no viajan en la URL, es decir, esta se mantiene inalterable. Si navegamos por https, los datos irán con encriptación punto a punto; esto quiere decir que solo el cliente los conoce, los encripta, y llega al servidor en el que se desencriptan. Son invisibles dentro de internet.

La etiqueta form puede tener una propiedad llamada **enctype** que puede tener los siguientes valores:

- application/x-www-form-urlencoded. Es el valor que toma por defecto si no se declara la propiedad dentro de la etiqueta form. Indica que los datos se encodean (codifican) al momento del envío del formulario.
- multipart/form-data: es necesario indicar este valor cuando vamos a enviar archivos al servidor (desde un `<input type="file">`).
- text/plain: envía los datos en formato de texto plano (es decir, en la red viajan totalmente visibles, por lo que no es recomendable utilizarlo).

Tenemos también elementos **inputs** y **textarea**. Ellos tienen definidas las siguientes propiedades:

- **name**. Es el nombre del elemento. En los formularios, es vital contar con esta propiedad, ya que el valor de dicha propiedad será utilizado por el servidor para recuperar los valores que el usuario ingresó.
- **required**: indica que el elemento es requerido. Todos los elementos con esta propiedad son validados por el navegador web antes de ser enviados al servidor. Esto refiere que, si ingresamos un *e-mail* inválido, en un *input* del tipo *e-mail*, por ejemplo, el navegador no

realizará el envío al servidor y nos mostrará un mensaje en pantalla pidiendo que lo corrijamos.

- **ID:** identificador único del componente. Como regla general, podemos indicar que pueden existir dos o más componentes con el mismo valor en esta propiedad. Esta propiedad puede existir o no, pero se recomienda utilizarlo en conjunto con la propiedad for de la etiqueta label que lo representa (los valores de for e id deben ser exactos) y se utilizan por cuestiones de accesibilidad. Por ejemplo, existen complementos para los navegadores para personas invidentes, y esta etiqueta permite la lectura y transformación en audio del valor.

Accediendo a los datos de un formulario desde PHP

Una vez que desarrollamos el formulario con el cual interactúa el cliente, necesitamos procesar sus datos en nuestro servidor web.

El navegador web envía los datos que se han completado en el formulario a través de la red (internet), por lo que debemos recordar la importancia de utilizar el protocolo https (visto en la lectura 1 del módulo 1.1.1), esto es, bajo este protocolo, los datos irán encriptados y serán ilegibles entre los puntos de comunicación (cliente y servidor).

Al llegar a nuestro servidor, PHP almacena estos datos en un *array* asociativo (en la memoria del servidor), que se denomina superglobal, ya que está disponible en cualquier parte de nuestro sistema, ya sea dentro de funciones, clases, funciones dentro de clases y demás.

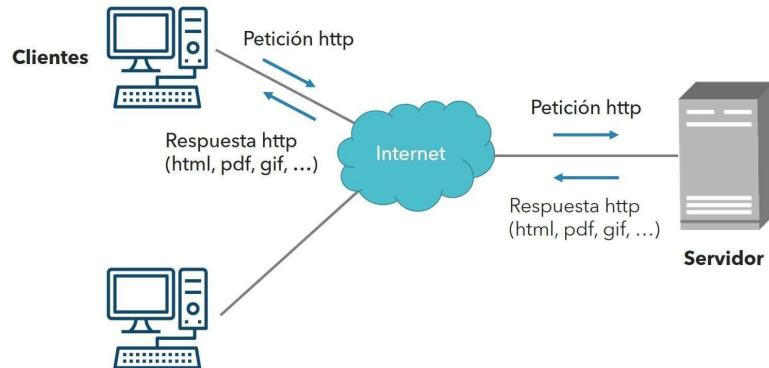
Como veremos en la siguiente unidad, estos *arrays* asociativos pueden ser tres diferentes:

- `$_POST`.
- `$_GET`.
- `$_REQUEST`.

Cada uno de ellos tiene particularidades diferentes, dado que dependen del método de envío del formulario, post o get.

Figura 14: Modelo cliente/servidor

Modelo Cliente / Servidor



Fuente: elaboración propia

Pregunta:

¿qué tipo de variable es `$_POST`?

• Array asociativo global.

• Array asociativo superglobal.

• Array de un nivel accesible solo para controladores.

• Dependiendo de los datos enviados, será string, int, float, etc.

Justificación

Variables `$_POST`, `$_GET` y `$_REQUEST`

Estas variables superglobales se completan con los datos que llegan al servidor y según el método que hemos indicado en formulario (post o get).

`$_POST`: se utiliza para recoger los datos de formularios creados con `method="POST"`. Nos retornará un array asociativo en el que los índices corresponden a las etiquetas `name` de los elementos de entrada (`input`, `textarea`, `select`) y sus valores son los datos que ingresó el cliente.

En el ejemplo de nuestro formulario anterior, podemos agregar las siguientes líneas para ver los datos del usuario:

Figura 15: Código

```
echo "<pre>";
print_r($_POST);
echo "</pre>";
```

Fuente: elaboración propia

Podremos ver esta variable superglobal de la siguiente forma:

Figura 16: Código

```
Array
(
    [nombre] => Pedro Picapiedras
    [email] => pedro@picpiedras.com
    [telefono] => 432324432
    [mensaje] => Este es un mensaje de pruebas del formulario
)
```

Fuente: elaboración propia

\$_GET: de forma análoga a la anterior, genera un asociativo superglobal, pero solo toma los datos cuando creamos el formulario con el método get (**method="GET"**).

Si se utiliza el siguiente código para mostrarlo, veremos la misma salida como la anterior:

Figura 17: Código

```
echo "<pre>";
print_r($_GET);
echo "</pre>";
```

Fuente: elaboración propia

Veremos lo siguiente:

Figura 18: Código

```
Array
(
    [nombre] => Pedro Picapiedras
    [email] => pedro@picpiedras.com
    [telefono] => 432324432
    [mensaje] => Este es un mensaje de pruebas del formulario
)
```

Fuente: elaboración propia

\$_REQUEST: es una variable superglobal como las anteriores, pero con una particularidad; podemos utilizarla tanto para formularios get (**method="GET"**) como formularios post (**method="POST"**), puesto que con cualquier método recupera los datos del formulario:

Figura 19: Código

```
echo "<pre>";
print_r($_REQUEST);
echo "</pre>";
```

Fuente: elaboración propia

Obtendremos la misma salida como en los dos casos anteriores:

Figura 20: Código

```
Array
(
    [nombre] => Pedro Picapiedras
    [email] => pedro@picpiedras.com
    [telefono] => 432324432
    [mensaje] => Este es un mensaje de pruebas del formulario
)
```

Fuente: elaboración propia

Como mencionamos anteriormente, siempre nos conviene utilizar formularios post, ya que los datos no viajan por la URL, y viajan por la red encriptados bajo navegación https.

Entonces, podemos definir que no intentaremos recoger los datos con la variable superglobal `$_GET`, y nos quedan `$_POST` o `$_REQUEST`.

Ahora bien, ¿cuál de las dos nos conviene utilizar?

Si consideramos que un usuario puede modificar la URL para enviar parámetros get desde su navegador, algo que no podemos impedir, entonces tendremos valores `$_GET` en conjunto con los `$_POST` del formulario.

Supongamos que la URL de nuestro formulario es la siguiente:

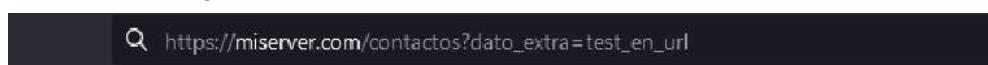
Figura 21: Ejemplo de URL de formulario



Fuente: elaboración propia

El usuario, con toda facilidad, podrá modificar la URL y enviarnos un valor por GET:

Figura 22: Enviar un valor por GET



Fuente: elaboración propia

Ahora, el usuario agregó un parámetro get llamado dato_extra. cuyo valor es test_en_url.

Cuando ejecutemos las siguientes líneas en nuestro servidor, el código se ve del siguiente modo:

Figura 23: Código

```
echo "<pre>";
print_r($_REQUEST);
echo "</pre>";
```

Fuente: elaboración propia

Veremos como salida lo siguiente:

Figura 24: Código

```
Array
(
    [dato_extra] => test_en_url
    [nombre] => Pedro Picapiedras
    [email] => pedro@picpiedras.com
    [telefono] => 432324432
    [mensaje] => Este es un mensaje de prueba del formulario
)
```

Fuente: elaboración propia

Podemos observar que, con una simple acción del usuario, nos ha enviado un nuevo parámetro GET en conjunto con los POST del formulario. Esto podemos considerarlo una falla de seguridad; por esta razón, se desaconseja el uso de \$_REQUEST.

Podemos llegar a la conclusión de que siempre nos conviene manejar los datos con \$_POST en el servidor y construir nuestros formularios con el método post.

Validación del formulario del lado del servidor

Anteriormente, hicimos mención al atributo required, el cual permite dejar las validaciones para el navegador web. Pero esta práctica no es del todo efectiva.

Veamos un ejemplo clásico. En el campo «Nombre» de nuestro formulario, podemos indicar varios espacios en blanco, sin ningún otro carácter; damos clic al botón «Enviar» y vemos que el formulario envía los datos al servidor. Imprimamos los datos mediante \$_POST.

Veremos la siguiente salida:

Figura 25: Código

```
Array
(
    [nombre] =>
    [email] => pedro@picpiedras.com
    [telefono] => 432324432
    [mensaje] => Este es un mensaje de pruebas del formulario
)
```

Fuente: elaboración propia

El nombre está vacío (o en realidad es un conjunto de caracteres espacios que no representan un nombre alguno).

¿Cómo lo validamos del lado del servidor?

Podemos incluir unas simples líneas:

Figura 26: Código

```
if (trim($_POST['nombre']) == "") {
    echo "Debe completar su Nombre y Apellido";
}
```

Fuente: elaboración propia

En este ejemplo, utilizamos la utilidad propia de PHP llamada trim, cuya funcionalidad es quitar espacios en blanco delante y detrás de los valores de variables.

Si al quitar todos los espacios quedamos con una cadena vacía de caracteres, estamos seguros de que el usuario no completará su nombre y lo informamos para que lo complete.

filter_var

Es una utilidad propia de PHP que nos permitirá aplicar diferentes validaciones sobre las variables.

Un caso muy utilizado es en validaciones de direcciones de e-mail:

Figura 27: Código

```
$email = "email@incorrecto";  
  
if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {  
    echo "el email ingresado no es correcto";  
}
```

Fuente: elaboración propia

Como observamos, filter_var nos solicita dos parámetros. El primero es la propia variable a validar y el segundo es una constante definida por el lenguaje PHP para indicar qué queremos validar.

Algunas de las posibles constantes son las siguientes:

FILTER_VALIDATE_EMAIL: valida que la variable sea una dirección de correo electrónico correcta.

FILTER_VALIDATE_BOOLEAN: la variable debe ser booleana (*true* o *false*).

FILTER_VALIDATE_FLOAT: la variable es un número flotante.

FILTER_VALIDATE_INT: la variable debe ser un número entero.

FILTER_VALIDATE_URL: la variable debe ser una dirección URL válida.

Tema 3. Sesiones

Muchas veces, hemos visto que ciertas páginas web nos solicitan «loguearnos» para interactuar con ellas. Una vez «logueados», tanto el cliente (navegador) como el servidor deben estar en conocimiento de ello, y guardar nuestro usuario para identificarnos durante toda nuestra interacción con el sistema.

En los sistemas populares de compra de productos, muchas veces, esta acción es requerida para, por ejemplo, permitirnos armar nuestro carro de compras, puesto que se debe conocer el usuario y los productos que va seleccionando.

Como esto es una especie de contrato entre cliente y el servidor, estos datos se guardan de

forma especial, es decir, no en una base de datos, y en cada iteración el cliente envía los datos de la sesión que son validados por el servidor que también los conoce.

PHP utiliza una variable superglobal llamada `$_SESSION` para almacenar los datos de la sesión (por ejemplo, el usuario y los productos que vamos agregando al carro de compras). Esta variable superglobal en el servidor se guarda en un archivo propio de él.

En el lado del cliente, se guardan en la memoria temporal del navegador web.

Una aclaración importante: cuando cerramos el navegador, su memoria temporal se limpia (elimina), por lo que la sesión se pierde y este contrato cliente-servidor deja de existir.

`$_SESSION` es un *array*, y puede o no ser asociativo según las necesidades de nuestro sistema.

Para iniciar una sesión en el servidor y asignarle datos, debemos comenzar con la sentencia `session_start();`

Figura 28: Código

```
<?php  
  
session_start();  
  
$_SESSION['usuario']['nombre'] = "Pedro Picapiedras";  
$_SESSION['usuario']['id'] = 1;  
$_SESSION['carro']['articulos'][1]['id'] = 11442;  
$_SESSION['carro']['articulos'][1]['nombre'] = "TV Led 29 pulgadas";  
$_SESSION['carro']['articulos'][2]['id'] = 11442;  
$_SESSION['carro']['articulos'][2]['nombre'] = "Disco HHD 1T";
```

Fuente: elaboración propia

De igual forma, si queremos recuperar sus valores, en este ejemplo, mostraremos los datos en el navegador:

Figura 29: Código

```
<?php
```

```
session_start();  
  
echo "<pre>";  
print_r($_SESSION);  
echo "</pre>";
```

Fuente: elaboración propia

La salida en pantalla será la siguiente:

Figura 30: Código

```
Array  
(  
    [usuario] => Array  
        (  
            [nombre] => Pedro Picapiedras  
            [id] => 1  
        )  
  
    [carro] => Array  
        (  
            [articulos] => Array  
                (  
                    [1] => Array  
                        (  
                            [id] => 11442  
                            [nombre] => TV Led 29 pulgadas  
                        )  
  
                    [2] => Array  
                        (  
                            [id] => 11442  
                            [nombre] => Disco HHD !T  
                        )  
                )  
        )  
)
```

Fuente: elaboración propia

Tema 4. Graficando código HTML

Desde bucles de PHP

Como mencionamos en lecturas anteriores, la construcción del lenguaje **echo** nos permite producir salidas en pantalla (navegador web), donde podremos mezclar código HTML y PHP.

Veamos un ejemplo en el que definimos variables en PHP, y luego las vamos mostrando intercambiando los lenguajes:

Figura 31: Código

```

<?php
$nombre1 = "Pedro";
$apellido1 = "Picapiedras";
$email1 = "pedro@picapiedras.com";
$nombre2 = "Vilma";
$apellido2 = "Picapiedras";
$email2 = "vilma@picapiedras.com"; ?>
<!DOCTYPE html>
<html lang="es">
    <head>
        <title>Contactos</title>
        <meta name="Contactos" content="Contactos">
        <meta charset="utf-8">
        <link rel="stylesheet" type="text/css" href="assets/css/main.css">
    </head>
    <body>
        <div class="contactos">
            <table>
                <thead>
                    <tr>
                        <th>Nombre</th>
                        <th>Apellido</th>
                        <th>Email</th>
                    </tr>
                </thead>

```

Fuente: elaboración propia

Figura 32: Código

```

                <tbody>
                    <tr>
                        <td><?php echo $nombre1 ?></td>
                        <td><?php echo $apellido1 ?></td>
                        <td><?php echo $email1 ?></td>
                    </tr>
                    <tr>
                        <td><?php echo $nombre2 ?></td>
                        <td><?php echo $apellido2 ?></td>
                        <td><?php echo $email2 ?></td>
                    </tr>
                </tbody>
            </table>
        </div>
    </body>
</html>

```

Fuente: elaboración propia

Como podemos observar, primeramente, definimos nuestras variables PHP; luego, realizamos el código HTML mezclando PHP para mostrarlas.

La salida en el navegador sería la siguiente:

Tabla 1: Salida en el navegador

| Nombre | Apellido | Email |
|--------|-------------|-----------------------|
| Pedro | Picapiedraa | pedro@picapiedras.com |
| Vilma | Picapiedraa | pedro@picapiedras.com |

Fuente: elaboración propia.

Podremos utilizar estructuras de repetición, condicionales, realizar operaciones matemáticas y todas las funcionalidades propias del lenguaje intercalando ambos lenguajes, recordando que siempre el código PHP debe ir entre las etiquetas de apertura `<?php` y de cierre `?>`.

Cabe aclarar que estas etiquetas son PHP y no HTML, puesto que desde la apertura hasta el cierre le estamos indicando al procesador de PHP que realice sus funciones.

Siguiendo el ejemplo anterior, podemos hacer una mejora, esto es, utilizar estructuras de repetición para graficar un número indeterminado de contactos.

En este ejemplo, utilizaremos un bucle **foreach**, pero podríamos hacer con un bucle **while** o **for**, según la necesidad del caso.

Figura 33: Código

```
<?php $contactos = array(
    0 => array("nombre" => "Pedro", "apellido" => "Picapiedras", "email" => "pedro@picapiedras.com"),
    1 => array("nombre" => "Vilma", "apellido" => "Picapiedras", "email" => "vilma@picapiedras.com"),
    2 => array("nombre" => "Pablo", "apellido" => "Marmol", "email" => "pablo@marmol.com")
); ?>
<!DOCTYPE html>
<html lang="es">
    <head>
        <title>Contactos</title>
        <meta name="Contactos" content="Contactos">
        <meta charset="utf-8">
        <link rel="stylesheet" type="text/css" href="assets/css/main.css">
    </head>
    <body>
        <div class="contactos">
            <table>
                <thead>
                    <tr>
                        <th>Nombre</th>
                        <th>Apellido</th>
                        <th>Email</th>
                    </tr>
                </thead>
                <tbody>
                    <?php foreach ($contactos as $contacto){ ?>
                    <tr>
                        <td><?php echo $contacto['nombre']; ?></td>
                        <td><?php echo $contacto['apellido']; ?></td>
                        <td><?php echo $contacto['email']; ?></td>
                    </tr>
                <?php } ?>
            </tbody>
        </table>
    </div>
</body>
</html>
```

Fuente: elaboración propia

En esta ocasión, hemos creado el *array* \$contactos y le hemos agregado 3 elementos, cada uno de ellos con su correspondiente nombre, apellido y *e-mail*.

Luego, en un bucle *foreach*, recorremos todos los elementos de este arreglo y graficamos la línea correspondiente de la tabla.

La salida del navegador sería la siguiente:

Tabla 2: Salida en el navegador

| Nombre | Apellido | Email |
|--------|-------------|-----------------------|
| Pedro | Picapiedras | pedro@picapiedras.com |
| Vilma | Picapiedras | vilma@picapiedras.com |
| Pablo | Marmol | pablo@marmol.com |

Fuente: elaboración propia

Podemos notar la principal ventaja de utilizar estructuras de control frente a la generación de las líneas de la tabla 1 por 1; solo agregando más líneas al arreglo, la tabla se graficará automáticamente sin la necesidad de que modifiquemos nuestro código HTML.

Desde PHP con información recuperada de la base de datos

Sin lugar a dudas, la máxima potencia de ambos lenguajes se logra al agregar un nuevo elemento, recupero de información desde una base de datos.

Siguiendo con el ejemplo anterior, imaginemos que en nuestra base de datos «test», contamos con la tabla contactos, la cual vamos llenando de registros a medida que nuestros clientes completan el formulario de contacto, indicando sus datos.

Utilizando la clase `base_datos`, desarrollada en lecturas anteriores (módulo 2, unidad 2. 2. 4.), podemos mostrar una tabla de todos los contactos que fuimos registrando en nuestro sistema.

- Nuestro código:

Figura 34: Código

```
<?php include './class/autocarga.php';
$dbm = new base_datos('mysql', "test", "127.0.0.1", "aquiles248", "aquiles248");
$contactos = $dbm->select("contactos"); ?>
<!DOCTYPE html>
<html lang="es">
    <head>
        <title>Contactos</title>
        <meta name="Contactos" content="Contactos">
        <meta charset="utf-8">
        <link rel="stylesheet" type="text/css" href="assets/css/main.css">
    </head>
    <body>
        <div class="contactos">
            <table>
                <thead>
                    <tr>
                        <th>Nombre</th>
                        <th>Apellido</th>
                        <th>Email</th>
                    </tr>
                </thead>
                <tbody>
                    <?php foreach ($contactos as $contacto) { ?>
                    <tr>
                        <td><?php echo $contacto['nombre']; ?></td>
                        <td><?php echo $contacto['apellido']; ?></td>
                        <td><?php echo $contacto['email']; ?></td>
                    </tr>
                    <?php } ?>
                </tbody>
            </table>
        </div>
    </body>
</html>
```

Fuente: elaboración propia

- Nuestra tabla mySQL;

Figura 35: Código

```
CREATE TABLE `contactos` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `nombre` varchar(200) NOT NULL,
  `apellido` varchar(200) NOT NULL,
  `email` varchar(200) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8
```

Fuente: elaboración propia

- El contenido de la tabla contiene los registros de clientes que fueron completando el formulario:

Tabla 3: Clientes que han ido completando el formulario

| id | nombre | apellido | email |
|----|--------|-------------|-----------------------|
| 1 | Pedro | Picapiedras | pedro@picapiedras.com |
| 2 | Vilma | Picapiedras | vilma@picapiedras.com |
| 3 | Pablo | Marmol | pablo@marmol.com |
| 4 | Betty | Marmol | betty@marmol.com |

Fuente: elaboración propia

- La salida en el navegador web:

Tabla 4: Salida en el navegador web

| Nombre | Apellido | Email |
|--------|-------------|-----------------------|
| Pedro | Picapiedras | pedro@picapiedras.com |
| Vilma | Picapiedras | vilma@picapiedras.com |
| Pablo | Marmol | pablo@marmol.com |
| Betty | Marmol | betty@marmol.com |

Fuente: elaboración propia

Como podemos observar, encontramos un cierto grado de dinamismo, puesto que, a medida que nuestros clientes se van registrando, la tabla en el navegador irá mostrando los nuevos resultados.

Tema 5. Vista modelo controlador (MVC)

Como vimos anteriormente en la primera unidad, el MVC es una técnica de programación que busca lograr una estructura de independencia entre las diferentes partes de un sistema. Estas partes son las siguientes:

- **Modelo:** se encarga de la lógica del sistema, esto es, acceso a base de datos o webs services para obtener datos y transformarlos en información que luego se presentará a la vista.
- **Vista:** la parte visual que se presenta tanto en el *back-end* como en el *front-end* (es decir, lo que mostramos en la pantalla del navegador). La vista interactúa principalmente con el controlador, pero a veces puede solicitar datos al modelo.
- **Controlador:** responde a eventos (acciones que realiza el usuario sobre nuestro sistema) e interactúa con el modelo. Podemos considerar un controlador como nexo entre la vista y el modelo.

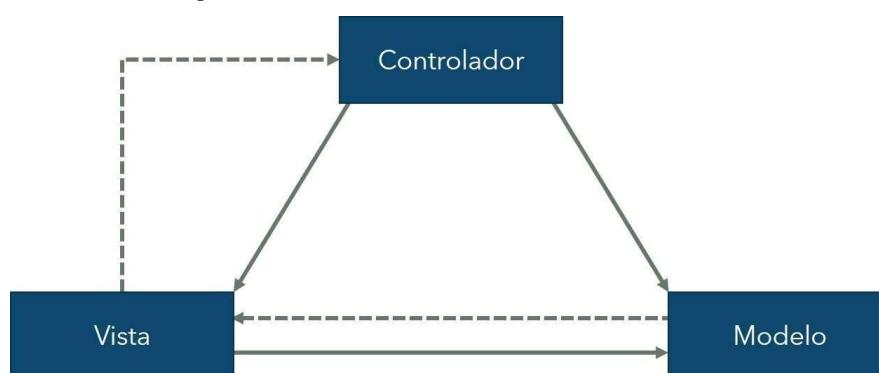
Podemos exemplificar el mecanismo de funcionamiento siguiendo nuestro formulario de clientes. Primeramente, presentamos en el navegador el formulario para que sea completado (vista).

Una vez que el formulario se envía a nuestro servidor, lo capturamos en el controlador que realiza las validaciones sobre los datos enviados.

En este punto, si los datos verificados son correctos, invocamos al modelo para que realice las acciones pertinentes (guardado de datos, actualización, eliminación, según sea el caso, en nuestra base de datos).

En caso de que la validación de datos no sea correcta (por ejemplo, el *e-mail* ingresado no tiene el formato correcto), el controlador vuelve a invocar la vista enviando un mensaje al usuario (ejemplo «el *e-mail* ingresado no es correcto, modifíquelo y vuelva a intentar»).

Figura 36: Controlador, vista y modelo



Fuente: elaboración propia

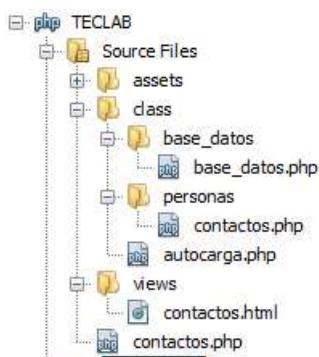
Partiendo de un ejemplo diferente, podemos ver cómo la vista puede interactuar con el modelo.

Supongamos que debemos realizar el listado de estos clientes registrados. La vista solicita la lista de clientes al modelo, el cual los recupera de la base de datos y se los entrega a la vista. Luego, de forma análoga a lo visto en esta lectura, graficamos el listado de clientes.

Como regla general, podemos tener una estructura de proyecto con una carpeta llamada «Class» en la que guardamos todos los modelos, otra llamada «Views», donde guardamos nuestras vistas y los archivos de controladores en el primer nivel de la estructura del proyecto.

Nuestro proyecto TECLAB tendría la siguiente estructura:

Figura 37: Estructura del proyecto TECLAB



Fuente: elaboración propia

Allí, dentro de class/personas/contactos.php, tenemos nuestro controlador de contactos.

En views/contactos.html, alojamos nuestra vista de contactos.

En el primer nivel del proyecto (root del sitio), tenemos nuestro controlador de contactos.

Si la vista es el ejemplo del listado de contactos, al invocar en el navegador <http://localhost/TECLAB/contactos.php>, estaríamos llamando al controlador de contactos; allí, podemos tener el siguiente código:

Figura 38: Código

```

<?php

include './class/autocarga.php';

include './views/contactos.html';

```

Fuente: elaboración propia.

Cargamos nuestra clase de autocarga y luego llamamos a la vista de contactos.

Figura 39. Código

```

<?php $contactos = contactos::listar(); ?>
<!DOCTYPE html>
<html lang="es">
    <head>
        <title>Formulario de contactos</title>
        <meta name="Contactos" content="Contactos">
        <meta charset="utf-8">
        <link rel="stylesheet" type="text/css" href="assets/css/main.css">
    </head>
    <body>
        <div class="contactos">
            <h1>Formulario de Contactos</h1>
            <table>
                <thead>
                    <tr>
                        <th>Nombre</th>
                        <th>Apellido</th>
                        <th>Email</th>
                    </tr>
                </thead>
                <tbody>
                    <?php foreach ($contactos as $contacto){ ?>
                    <tr>
                        <td><?php echo $contacto['nombre'] ?></td>
                        <td><?php echo $contacto['apellido'] ?></td>
                        <td><?php echo $contacto['email'] ?></td>
                    </tr>
                    <?php } ?>
                </tbody>
            </table>
        </div>
    </body>
</html>

```

Fuente: elaboración propia

En la primera línea de la vista:

```
<?php $contactos = contactos::listar(); ?>
```

Estamos invocando el modelo (clase) contactos para recuperar todos los registros de la base de datos.

La salida en el navegador será la siguiente:

Tabla 5: Salida en el navegador

Formulario de Contactos

| Nombre | Apellido | Email |
|--------|-------------|-----------------------|
| Pedro | Picapiedras | pedro@picapiedras.com |
| Vilma | Picapiedras | vilma@picapiedras.com |
| Pablo | Marmol | pablo@marmol.com |
| Betty | Marmol | betty@marmol.com |

Fuente: elaboración propia

Como podemos observar, interactuamos con controlador -> vista -> modelo.

Podemos considerar la estructura de trabajo que hemos propuesto en nuestro *framework* (plataforma para crear aplicaciones web) simplificado, el cual podemos utilizar en otros desarrollos e ir ampliando a medida que realicemos proyectos más grandes.

Existen muchos *frameworks* que pueden descargarse de internet que ya vienen con la estructura necesaria para trabajar con el formato VMC; entre ellos, podemos nombrar los siguientes:

- Laravel.
- Codeigniter.
- Symfony.
- CakePHP.
- YI.
- Zend Framework.

Microactividades

¡A practicar!

Vamos a seguir trabajando en la creación de un sistema de organización para la clase de Química Orgánica de una escuela. Seguimos poniendo en práctica lo aprendido sobre Clases de PHP.

1. En la clase alumnos que utilizamos en la actividad anterior creamos una función listar para mostrar los registros de alumnos de la tabla.

Modifica esta función para mostrar el listado dentro de una tabla html. Todo debe estar dentro de la función listar, es decir, mezclando código html y php dentro de la misma.

- 2. Modifica la función mostrar para que, en lugar de mostrar la tabla dentro de la misma, retorne los resultados.

Ahora, creá un archivo index.php donde llames a la función listar de alumnos y con la respuesta de la misma gráfica de la tabla, debes llamar a una función de autocarga para cargar la clase alumnos.

-
- 3. Si decidimos utilizar la estructura VISTA - MODELO - CONTROLADOR ¿cómo quedaría la estructura de nuestro proyecto?

Entrega: una captura de pantalla de la estructura de directorios desplegada, con sus archivos dentro. No olvides crear el directorio assets aunque no lo hayamos utilizado en esta actividad.

En el siguiente video podrás ver las respuestas correctas:

Video de habilidades

1- ¿Cuál es el protocolo que se utiliza para el envío de emails?

SMTP

FTP

HTTP

TELNET

2- Utilizando PHPMailer ¿es factible conectarse a una cuenta de gmail
conociendo solamente la dirección de correo electrónico para enviar
correos a otras cuentas?

Verdadero

Falso

Cierre

La combinación entre lenguajes de programación es vital a la hora de construir mejores sistemas.

La utilización de PHP, muy fácil de aprender por su amplia difusión en internet (*manual online* — <https://php.net>— donde encontramos toda la difusión oficial del lenguaje), junto con HTML (que nos permite graficar en el navegador), CSS (que habilita tener páginas amigables a los usuarios) y base de datos (MySQL —en estas lecturas—, que permite con toda facilidad guardar contenido permanente en nuestras tablas), hacen que nuestros sistemas sean robustos, dinámicos, adaptables y mantenibles (concepto que refiere a la facilidad de resolución de errores o de rápida adaptación a nuevos requerimientos) a todas las necesidades de sistemas.

En su conjunto con las técnicas de programación orientada a objetos y el modelo VMC, nos permiten crear proyectos de diferentes dimensiones y satisfacer todas sus necesidades.

Glosario