cipsa.net

PHP & MySQL

Anexo 10.- MVC y Web Services

DISTRIBUIDO POR:

CENTRO DE INFORMÁTICA PROFESIONAL S.L.

C/ URGELL, 100
08011 BARCELONA
TFNO: 93 426 50 87

C/ RAFAELA YBARRA, 10
48014 BILBAO
TFNO: 94 448 31 33

www.cipsa.net

Patrón MVC

MVC son las siglas de *Modelo-Vista-Controlador*. MVC es un patrón que determina la estructura interna de una aplicación web de modo que sea más fácil su mantenimiento y escalabilidad.

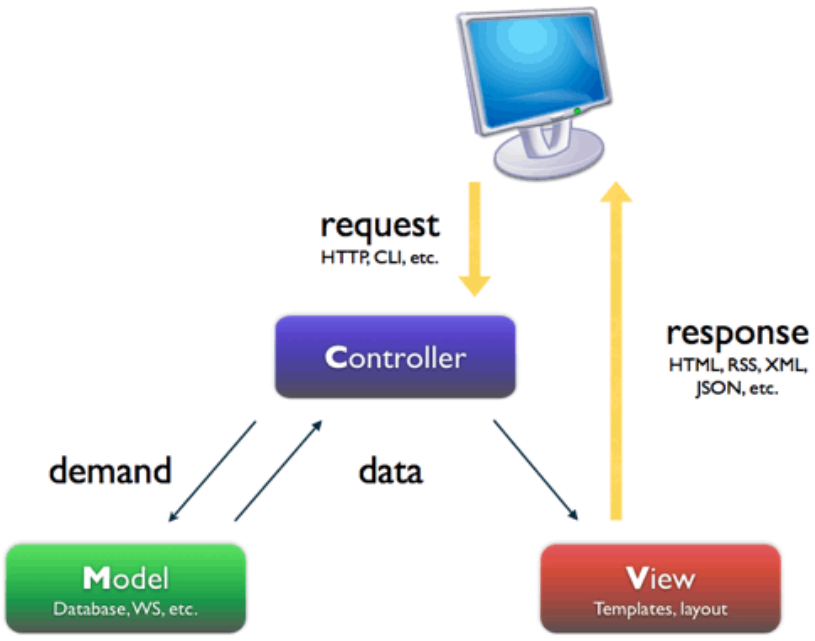
La separación del código en capas permite crear aplicaciones Web evitando la mezcla de código de servidor (*PHP*) y cliente (*HTML*, *Javascript*), así como aislar el impacto de las modificaciones del código de una capa en el resto.

El patrón MVC estructura el código de una aplicación en *tres capas*:

- ***El modelo*** → Esta es la capa de *información* responsable de la obtención de la información empleada por la aplicación desde los orígenes de datos (ficheros, bases de datos, servicios web…, etc). Esta capa contiene las clases que representan los tipos de datos a manejar (*entidades*) y las que gestionan la obtención y actualización de los mismos (*modelos*) desde/hacia los orígenes de datos. El resto de las capas manejan los datos aportados por la capa modelo sin conocimiento de su procedencia.

- ***La vista*** → Esta es la capa de *interfaz con el usuario* responsable de mostrar y solicitar valores al usuario. Esta parte implementa la generación de las páginas web empleando *PHP* para el contenido dinámico y plantillas con *HTML*, *CSS*, *Javascript*… etc, para la maquetación y el código a ejecutar en el lado del cliente.

- ***El controlador*** → Esta es la capa intermediaria entre la vista y el modelo. Se ocupa de recibir las solicitudes procedentes del navegador del usuario, recupera y actualizar los datos requeridos del modelo, y presentar los resultados generando mediante la vista la página web de respuesta.



Esquema funcional de una aplicación modelo vista controlador básica.

Ciclo de atención de peticiones en patrón MVC

El ciclo de vida de una solicitud dirigida a una aplicación Web MVC sigue los siguientes pasos:

1. El usuario realiza la solicitud dirigida al dominio del servidor web que es recogida y atendida por la capa ***controlador***. En función del tipo de petición, la URL, y los parámetros contenidos en la misma se ejecuta el código de controlador correspondiente.

2. La capa ***modelo*** (*la única que tiene acceso directo a las bases de datos*) recupera la información necesaria por la capa *controlador*.

3. La capa ***controlador*** procesa los datos recibidos de la capa *modelo* y envía los resultados a la capa *vista* para generar la página de respuesta al usuario.

4. La capa ***vista*** recibe los resultados enviados desde la capa controlador y genera la página de respuesta para el usuario empleando una plantilla (*layout*) con la maquetación de la página.

5. El usuario recibe la página web compuesta como respuesta a su petición.

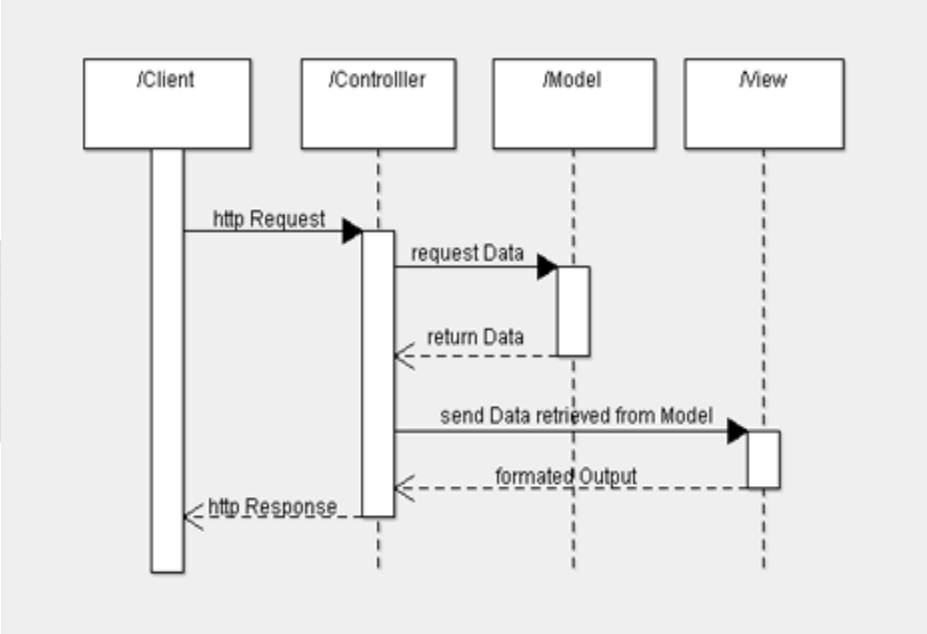


Diagrama de secuencia del procesado de una petición en una aplicación MVC

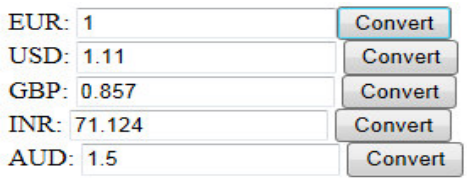
Objetivos

La finalidad del uso del patrón MVC es crear aplicaciones modulares cuyas partes puedan modificarse y ampliarse evitando la modificación de las demás. Esto facilita el mantenimiento (detección y corrección de problemas) y escalabilidad (facilidad de ampliación y actualización) de la aplicación.

Ejemplo de implementación sencilla MVC

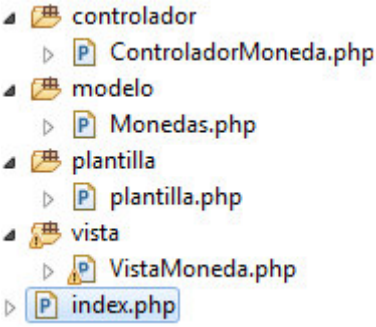
Supóngase que se desea crear una pequeña aplicación web que permita calcular el cambio de moneda entre las diferentes divisas: Euros (*EUR*), Dólares americanos (*USD*), Dólares australianos (*AUD*), Libras (*GBP*) y Rupias (*INR*).

Para ello la aplicación mostrará la siguiente página:



Al introducir el valor de una divisa y pulsar el botón "*CONVERT*" correspondiente debe mostrarse el cambio en el resto de monedas.

Para la implementación de esta aplicación vamos a emplear el patrón MVC, por lo que dividiremos el código entre modelo, vista y controlador. La estructura de ficheros del proyecto se divide en las siguientes carpetas:



- ***controlador*** → Carpeta con las clases controladoras.
- ***modelo*** → Carpeta con las clases que implementan el modelo para el acceso y tratamiento de los datos.
- ***vista*** → Carpeta con las clases que implementan las vistas para generar las páginas web de respuesta para el usuario
- ***plantilla*** → Carpeta con los archivos que sirven de plantilla para la generación de las páginas de respuesta.

El fichero ***index.php*** es el *punto de entrada a la aplicación* que atiende todas las peticiones de los usuarios, las cuales deben ir dirigidas contra su URL.

Implementación del modelo

La capa modelo es la encargada de obtener, actualizar y procesar la información manejada por la aplicación web. Para ello implementamos en este caso la clase ***modelo/Monedas*** que calcula el cambio de una cantidad monetaria de una divisa a otras mediante sus métodos:

***Código**: Modelo/Monedas*

```php
<?php
namespace Modelo;
class Monedas {
    private $baseValue = 0;     // Valor de referencia en Euros para conversión
    private $rates = [
        'GBP' => 1.16670,       // Libra
        'USD' => 0.90123,       // Dolares americanos
        'EUR' => 1.0,           // Euros
        'INR' => 0.01406,       // Rupias
        'AUD' => 0.66687        // Dolares australianos
    ];
```

```
    /**
     * Devuelve valor de conversion en la divisa indicada
     * @param unknown $currency
     * @return number
     */
    public function get($currency) {
        if (isset($this->rates[$currency])) {
            $rate = 1/$this->rates[$currency];
            return round($this->baseValue * $rate, 3);
        }
        else return 0;
    }
    /**
     * Devuelve una matriz asociativa con el cambio en cada una de las divisas
     * @return number[]
     */
    public function getAll() {
        $monedas = array();
        foreach( array_keys($this->rates) as $moneda ) {
                $monedas[$moneda] = $this->get($moneda);
        }
        return $monedas;
    }
    /**
     * Fija valor monetario de origen y el tipo de divisa
     * @param unknown $amount
     * @param string $currency
     */
    public function set($amount, $currency = 'EUR') {
        if (isset($this->rates[$currency])) {
            $this->baseValue = $amount * $this->rates[$currency];
        }
    }
}
```

- El método ***set(cantidad, moneda)*** recibe la cantidad y el tipo de moneda que se quiere convertir, y lo pasa a Euros como valor de referencia almacenándolo en el atributo *$this->baseValue*.

- El método ***get(moneda)*** convierte la cantidad en euros como referencia a la moneda indicada y retorna el resultado.

- El método ***getAll()*** convierte la cantidad en euros como referencia a todas las monedas almacenando los resultados en una matriz asociativa que se retorna como resultado. El formato de la matriz resultante es la siguiente:

```
array (size=5)
  'GBP' => float 0.857
  'USD' => float 1.11
  'EUR' => float 1
  'INR' => float 71.124
  'AUD' => float 1.5
```

Ejemplo de uso:

El script "*index.php*" es el *punto de entrada a la aplicación que recibe las peticiones de los usuarios*. Por ello, incluye la función de autocarga (*autload_classes*) necesaria para la importación automática de todas las clases requeridas por el resto de capas.

```php
<?php

define('PROJECTPATH', dirname(__FILE__));
function autoload_classes($class_name)
{
        $filename = PROJECTPATH . '/' . str_replace('\\', '/', $class_name) .'.php';
        if(is_file($filename))
        {
                include_once $filename;
        }
}
//registramos el autoload autoload_classes
spl_autoload_register('autoload_classes');

use Modelo\Monedas;

$modelo = new Monedas();
$modelo->set(1, 'EUR');                         // Fijado de valor inicial a convertir 1€
echo "1 Euro = ";
echo $modelo->get("USD")." USD, ";              // Obtención de cambio en otras monedas
echo $modelo->get("GBP")." GBP, ";
echo $modelo->get("INR")." INR, ";
echo $modelo->get("AUD")." AUD, ";
```

El código anterior muestra el manejo del modelo de la aplicación creando un objeto de la clase ***Modelo/Monedas***, simulando la inserción de 1 Euro como valor de partida, y la obtención del cambio al resto de monedas.

El resultado mostrado por pantalla será el siguiente:

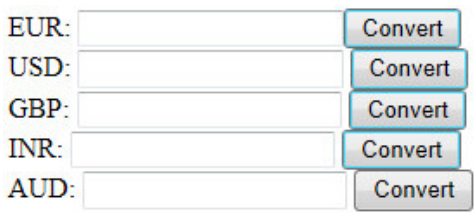
1 Euro = 1.11 USD, 0.857 GBP, 71.124 INR, 1.5 AUD,

Implementación de la vista

La capa vista es la encargada de visualizar y obtener los datos para el usuario. En el caso de una aplicación web esto se traduce en generar las páginas web. Este proceso se basa normalmente en el uso de *plantillas*.

Una *plantilla* contiene la maquetación, el diseño y las partes estáticas de una página web, e incluye pequeñas inserciones de código que se sustituyen por el contenido dinámico al enviar la página final al usuario. Este es el trabajo de las clases que conforman la vista.

Supongamos que la página que deseamos mostrar al usuario es la siguiente:



La plantilla debe contener el contenido estático, es decir; el que no cambia. En este caso eso son los formularios con las etiquetas de cada moneda y su botón “CONVERT”. La parte dinámica son los valores de las cajas de texto que van incluidos mediante inserciones de PHP cuyos valores serán proporcionados por la clase vista:

***Código**: plantilla/plantilla.php*

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>Valores</title>
    </head>
    <body>
        <form action="?action=convert" method="post">
        <input name="currency" type="hidden" value="EUR"/><label>EUR:</label>
        <input name="amount" type="text" value="<?=$this->cantidades['EUR']?>"/>
        <input type="submit" value="Convert"/>
        </form>
        <form action="?action=convert" method="post">
        <input name="currency" type="hidden" value="USD"/><label>USD:</label>
        <input name="amount" type="text" value="<?=$this->cantidades['USD']?>"/>
        <input type="submit" value="Convert"/>
        </form>
        <form action="?action=convert" method="post">
        <input name="currency" type="hidden" value="GBP"/><label>GBP:</label>
        <input name="amount" type="text" value="<?=$this->cantidades['GBP']?>"/>
        <input type="submit" value="Convert"/>
        </form>
        <form action="?action=convert" method="post">
        <input name="currency" type="hidden" value="INR"/><label>INR:</label>
        <input name="amount" type="text" value="<?=$this->cantidades['INR']?>"/>
        <input type="submit" value="Convert"/>
        </form>
        <form action="?action=convert" method="post">
        <input name="currency" type="hidden" value="AUD"/><label>AUD:</label>
        <input name="amount" type="text" value="<?=$this->cantidades['AUD']?>"/>
        <input type="submit" value="Convert"/>
        </form>
    </body>
</html>
```

La clase ***Vista/VistaMoneda*** es la responsable de recibir los valores a mostrar en la plantilla y generar la página para el usuario.

***Código**: Vista/VistaMoneda*

```
<?php
namespace Vista;
use Modelo\Monedas;

class VistaMoneda {
        private $cantidades;
        /**
         * Reciba los valores a mostrar
         * @param unknown $_cantidades
         */
        public function __construct($_cantidades = NULL) {
                if ( $_cantidades != NULL ) {
                        $this->cantidades = $_cantidades;
                } else {
                        // Matriz vacía para no mostrar ningún valor en las cajas de texto
                        $this->cantidades = [ 'EUR' => '',
                                              'USD' => '',
                                              'GBP' => '',
                                              'INR' => '',
                                              'AUD' => ''];
                }
        }
```

```
        /**
         * Carga y envia el fichero plantilla correspondiente.
         */
        public function output() {
                include PROJECTPATH. "/plantilla/plantilla.php";
        }
}
```

Los métodos de la clase son los siguientes.

- El constructor que recibe una matriz asociativa con todas las monedas y cantidad a mostrarse en la plantilla. Puede invocarse al constructor sin dar ninguna matriz para que se muestre la página sin valores.
- El método ***output()*** que importa el fichero plantilla correspondiente de la carpeta plantillas y lo envía al usuario.

Ejemplo de uso:

El código siguiente en el fichero "*index.php*" muestra el uso del modelo y la vista para mostrar una página al usuario en la que se simula la inserción del valor 1 € por parte del usuario y se visualiza el cambio a cada una de las monedas.

La página resultante enviada al usuario debe ser:

Código prueba en "*index.php*"

```
<?php

define('PROJECTPATH', dirname(__FILE__));
function autoload_classes($class_name)
{
        $filename = PROJECTPATH . '/' . str_replace('\\', '/', $class_name) .'.php';
        if(is_file($filename))
        {
                include_once $filename;
        }
}
//registramos el autoload autoload_classes
spl_autoload_register('autoload_classes');

use Modelo\Monedas;
use Vista\VistaMoneda;

// Creación del objeto modelo.
$modelo = new Monedas();
// Fijado de valor inicial 1 Euro
$modelo->set(1, 'EUR');
// Obtención de la matriz asociativa con el cambio a todas las monedas.
$cambios = $modelo->getAll();

// Creación del objeto vista asignando la matriz con los cambios a todas las monedas.
$vista = new VistaMoneda($cambios);
// generación de la página resultante al usuario
$vista->output();
```

Implementación del controlador

El controlador es el encargado de atender las peticiones del usuario, obtener los datos del modelo, y generar la pagina de respuesta con la vista. En este caso creamos la clase controlador "***Controlador/ControladorMoneda***".

En una aplicación MVC todas las peticiones son atendidas por el controlador. Por ello, todas deben ir dirigidas a la URL del punto de entrada a la aplicación web (*"index.php"*). Este script debe pasar el control a la clase controlador:

Código: *index.html.*

```php
<?php

define('PROJECTPATH', dirname(__FILE__));
function autoload_classes($class_name)
{
        $filename = PROJECTPATH . '/' . str_replace('\\', '/', $class_name) .'.php';
        if(is_file($filename))
        {
                include_once $filename;
        }
}
//registramos el autoload autoload_classes
spl_autoload_register('autoload_classes');

use Controlador\ControladorMoneda;
// Creación del objeto controlador
$controlador = new ControladorMoneda();

// Enrutado
if (isset($_GET['action'])) {
        $controlador->{$_GET['action']}($_POST); // Llamada al método del objeto controlador.
} else {
        $controlador->home();              // Llamada a método de acción predeterminado home.
}
```

La parte más importante del script "*index.php*" es lo que se denomina "*enrutado*".

Las aplicaciones web tradicionales se componían de múltiples páginas cada una de las cuales hacía una determinada cosa, y el usuario navegaba hacia ellas para realizar cada tarea. En las aplicaciones MVC sin embargo, todas las peticiones van dirigidas a la misma URL (*punto de entrada de la aplicación*) y añaden un parámetro de acción que indique al controlador lo que debe hacer. Ese parámetro es "*action*" y normalmente va concatenado en la propia URL.

En aplicación web que estamos haciendo de ejemplo tiene dos acciones:

- "***home***" → Es la acción que muestra la página sin ningún valor. Es la acción predeterminada cuando el usuario accede por primera vez a la aplicación.
- "***convert***" → Es la acción que solicita la conversión de una cantidad monetaria introducida al resto de monedas. Esta es invocada por todos los formularios de la vista enviando los datos a la URL: "*?action=convert*"

El valor del parámetro "*action*" concatenado en la URL, se emplea para invocar al método de mismo nombre en la clase controladora.

La clase "***Controlador/MonedaControlador***" consta por tanto de dos métodos:

- ***home()*** → Método que únicamente visualiza la página sin valores.
- ***convert(request)*** → Método que obtiene los valores de los campos del formulario, obtiene los cambios mediante la clase del modelo (*Modelo/Moneda*), y muestra los resultados mediante la clase de la vista (*Vista/VistaMoneda*).

***Código**: Controlador/ControladorMoneda.*

```php
<?php
namespace Controlador;

use Modelo\Monedas;
use Vista\VistaMoneda;

class ControladorMoneda {

	private $modelo;
	public function __construct() {
		$this->modelo = new Monedas();
	}
	/**
	 * Método de accion 'home' -> Muestra la página sin valores
	 */
	public function home() {
		$vista = new VistaMoneda();
		$vista->output();
	}
	/**
	 * Método de acción 'convert' -> Muestra la página con las conversiones
	 * @param unknown $request
	 */
	public function convert($request) {
		if (isset($request['currency']) && isset($request['amount'])) {
			// Obtención de valores introducidos
			$cantidad_introducida = $request['amount'];
			$moneda_introducida = $request['currency'];
			// Inserción de cantidad de entrada al modelo
			$this->modelo->set($cantidad_introducida, $moneda_introducida);
			// Obtencion de matriz de cambios
			$matriz_cambios = $this->modelo->getAll();
			// Visualizacion de cambios resultantes
			$vista = new VistaMoneda($matriz_cambios);
			$vista->output();
		}
	}
}
```

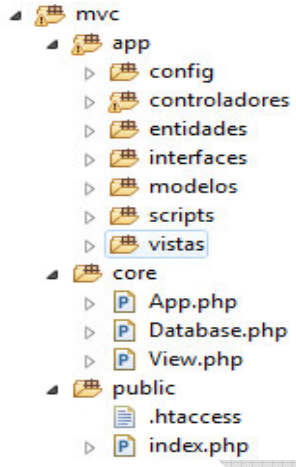
Implementación de aplicación CRUD con MVC

El siguiente ejemplo será una aplicación MVC de mantenimiento de usuarios almacenados en una base de datos. Para ello emplearemos la tabla "*usuarios*" contenida en la base de datos "*mensajería*" con el siguiente diseño:

Columna	Tipo	Nulo	Predeterminado
usuario *(Primaria)*	varchar(10)	No	
clave	varchar(10)	No	
administrador	bit(1)	No	
activo	bit(1)	No	b'1'

Diseño tabla "usuarios" de BD "mensajería"

La estructura del proyecto que vamos a emplear es la siguiente:



- La carpeta ***"/app"*** contiene las carpetas que conforman los elementos de la propia aplicación así como otras carpetas con interfaces, scripts de javascript, imágenes…, etc.
- La carpeta ***"/core"*** contiene clases reutilizables no dependientes de la aplicación web, y que son reutilizables.
- La carpeta ***"/public"*** contiene el punto de entrada a la aplicación web.

URLs amigables

El concepto de URLs amigables consiste en hacer que las aplicaciones web funcionen empleando URLs semánticamente comprensibles evitando el uso parámetros y valores concatenados. Por ejemplo: una URL no amigable podría ser la siguiente:

http://www.dominio.es/index.php?accion=detalle&userid=10293

Esta es una URL compleja entender, escribir y recordar. Una URL amigable debe emplear únicamente palabras relacionadas con el contenido que se solicita.

http://www.dominio.es/usuarios/detalle/10293

El uso de las URLs amigables está muy relacionado con las técnicas de posicionamiento (SEO), ya que facilitan la indexación de las mismas y la valoración de los enlaces, mejorando así su posicionamiento en los buscadores.

Reescritura de URLs (*mod_rewrite*)

El uso de URLs amigables requiere la configuración de reescritura de URL en el servidor web. Esto se realiza mediante la configuración módulo *mod_rewrite* mediante un fichero *.htaccess* en el caso de los servidores Apache.

Ejemplo: El siguiente archivo "*.htaccess*" situado en la carpeta de publicación raíz de un sitio web, reescribe cualquier URL dirigida al dominio:

```
Options -Multiviews
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.+)$ index.php?url=$1 [QSA]
```

Una petición con la siguiente URL:

www.dominio.es/usuarios/detalle/10293

Es convertida en:

www.dominio.com/index.php?url=usuarios/detalle/10293

De este modo todas las peticiones resultan atendidas por el script "*index.php*" convirtiéndose en el *punto de entrada* de la aplicación web. Este es el tipo de reescritura que emplearemos en nuestra aplicación web.

Para definir otros tipos de reescritura puede consultarse la documentación del módulo *mod_rewrite* en el sitio web oficial de Apache:

http://httpd.apache.org/docs/current/mod/mod_rewrite.html

El punto de entrada

El punto de entrada es el script al que se dirigen todas las peticiones web mediante la reescritura de URLs. En esta aplicación web el punto de entrada es el fichero "*public/index.php*":

```
<?php
define("DOMAIN", "/prueba/mvc/public");
define("JS_SCRIPTS", "/prueba/mvc/app/scripts");

//directorio del proyecto
define("PROJECTPATH", dirname(__DIR__));
//directorio app
define("APPPATH", PROJECTPATH . '/App');

//función de autocarga
function autoload_classes($class_name)
{
        $filename = PROJECTPATH . '/' . str_replace('\\', '/', $class_name) .'.php';
        if(is_file($filename)) {
                include_once $filename;
        }
}
spl_autoload_register('autoload_classes');

//creación del objeto enrutador y ejecución del controlador.
$app = new \Core\App;
$app->render();
```

La clase enrutadora (Core/App)

La clase *Core/App* hace de enrutadora. Su finalidad es inspeccionar la URL reescrita separando tres valores: controlador, métodos y parámetros.

El formato de las URLs esperadas es el siguiente:

http://localhost/mvc/public/<controlador>/<metodo>/<param_1>/<param_2>/....

Ejemplo: Sea la URL: http://localhost/mvc/public/controlusuarios/ver/petrov

La URL es reescrita por el *mod_rewrite* del siguiente modo:

http://localhost/mvc/index.php?url=controlusuarios/ver/petrov

El *constructor* de la clase ***Core/App*** obtiene el valor del parámetro "url" y lo descompone en sus tres atributos:

- Nombre del controlador al que va dirigida la petición (*$_controller*) = "*controlusuarios*"
- Nombre del método del controlador solicitado (*$_method*) = "*ver*"
- Parámetros para el método solicitado (*$_parameters*) = "*petrov*"

Mediante estos atributos se llama al método del controlador correspondiente pasando como parámetro una matriz con los valores recogidos mediante el método ***render()***.

Todo éste proceso está implementado en el constructor de la clase ***Core/App***.

```php
<?php
namespace Core;
defined("APPPATH") OR die("Access denied");
class App
{
        private $_controller;
        private $_method = "index";
        private $_params = [];
        public function getController(){
                return $this->_controller;
        }
        public function getMethod(){
                return $this->_method;
        }
        public function getParams() {
                return $this->_params;
        }

        const NAMESPACE_CONTROLLERS = "\App\Controladores\\";
        const CONTROLLERS_PATH = "../App/controladores/";

        /**
         * [getConfig Obtenemos la configuración de la app]
         * @return [Array] [Array con la config]
         */
        public static function getConfig()
        {
                return parse_ini_file(APPPATH . '/config/config.ini');
        }

        /**
         * [parseUrl Parseamos la url en trozos]
         * @return [type] [description]
         */
        public function parseUrl()
        {
                if(isset($_GET["url"]))
                {
                        return explode("/", filter_var(rtrim($_GET["url"], "/"), FILTER_SANITIZE_URL));
                }
        }
```

```
        public function __construct()
        {
                //obtenemos la url parseada
                $url = $this->parseUrl();
                 // controlador y accion por defecto
                 if ( $url == null) {
                          $url[0] = "controlusuario";
                          $url[1] = "listar";
                 }
                //comprobamos que exista el archivo en el directorio controllers
                if(file_exists(self::CONTROLLERS_PATH.ucfirst($url[0]) . ".php"))
                {
                        //nombre del archivo a llamar
                        $this->_controller = ucfirst($url[0]);
                        //eliminamos el controlador de url, así sólo nos quedaran el metodo
                        unset($url[0]);
                }else{
                        include APPPATH . "/vistas/errores/404.php";
                        exit;
                }
                //obtenemos la clase con su espacio de nombres
                $fullClass = self::NAMESPACE_CONTROLLERS.$this->_controller;

                //asociamos la instancia a $this->_controller
                $this->_controller = new $fullClass;

                //si existe el segundo segmento comprobamos que el método exista en esa clase
                if(isset($url[1]))
                {
                        //aquí tenemos el método
                        $this->_method = $url[1];
                        if(method_exists($this->_controller, $url[1]))
                        {
                               //eliminamos el método de url, así sólo quedan los parámetros del
método
                               unset($url[1]);
                        }else{
        throw new \Exception("Controlador: {$fullClass} Metodo: {$this->_method} Desconocido", 1);
                        }
                }
                //asociamos el resto de segmentos a $this->_params para pasarlos al método llamado.
                $this->_params = $url ? array_values($url) : [];
        }

        /**
         * [render  lanzamos el controlador/método que se ha llamado con los parámetros]
         */
        public function render()
        {
                call_user_func_array([$this->_controller, $this->_method], $this->_params);
        }
}
```

Si la URL no lleva información de controlador y métodos:

http://localhost/mvc/public/

La clase enrutadora ejecuta de manera predeterminada el método "*listar*" del controlador "*controlusuarios*" como si fuera:

http://localhost/mvc/public/controlusuarios/listar

El método ***getConfig()***devuelve una matriz asociativa con los valores contenidos en el archivo de configuración "*app/config/config.ini*". En este caso, se trata de los parámetros de conexión a la base de datos.

Código: */app/config/config.ini*

```
[database]
host     = 192.168.9.180
user     = alumno
password = cipsa
database = mensajeria
```

Implementación del modelo

El modelo en esta aplicación se basa en tres clases:

- ***Core/Database*** → Esta es la clase principal que establece comunicación con la base de datos empleando la librería PDO de PHP.

- ***App/Modelos/Usuarios*** → Esta clase implementa todas las operaciones a realizar sobre la tabla de usuarios de la base de datos incluyendo métodos para obtener, actualizar, insertar y eliminar usuarios.

- ***App/Entidades/Usuario*** → Esta clase representa la entidad de información usuario incluyendo todos los datos almacenados por cada usuario como atributos.

Core/Database

Esta es la única clase que establece contacto directo con la base de datos. Su función es aislar a la aplicación de la base de datos que se está empleando.

El método estático ***instance()*** se emplea para obtener siempre el mismo objeto de esta clase en vez de instanciar un nuevo objeto cada vez. Esto se hace para emplear la misma conexión al servidor de base de datos.

- El *constructor* crea el objeto *PDOConnection*.

- El método ***run(consulta, parámetros)*** ejecuta la consulta con los parámetros indicados retornando como resultado el objeto *PDOStatement* con el conjunto de resultados.

***Código**: Core/Database*

```php
<?php
namespace Core;
defined("APPPATH") OR die("Access denied");
use \Core\App;

class Database
{
	private $_dbUser;
	private $_dbPassword;
	private $_dbHost;
	protected $_dbName;

	private $_connection;
	/**
	 * @desc instancia de la base de datos
	 * @var $_instance
	 * @access private
	 */
	private static $_instance;
	 /**
	  * [instance singleton]
	  * @return [object] [class database]
	  */
	 public static function instance()
	 {
		if (!isset(self::$_instance)){
			$class = __CLASS__;
			self::$_instance = new $class;
		}
		return self::$_instance;
	 }
```

```
        /**
         * [__construct]
         */
        private function __construct()
        {
                try {
                        //load from config/config.ini
                        $config = App::getConfig();
                        $this->_dbHost = $config["host"];
                        $this->_dbUser = $config["user"];
                        $this->_dbPassword = $config["password"];
                        $this->_dbName = $config["database"];
                        $this->_connection = new \PDO('mysql:host='.$this->_dbHost.'; dbname='.$this-
>_dbName, $this->_dbUser, $this->_dbPassword);
                        $this->_connection->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION);
                        $this->_connection->exec("SET CHARACTER SET utf8");
                }catch (\PDOException $e){
                        print "Error!: " . $e->getMessage();
                        die();
                }
        }
        /**
         * [run]
         * @param [type] $sql [descripcion]
         * @param array $args parametros de sustitucion para consulta
         * @return PDOStatement
         */
        public function run($sql, $args = [])
        {
                $stmt = $this->_connection->prepare($sql);
                $stmt->execute($args);
                // Retorno de objeto PDOStatement.
                return $stmt;
        }

       /**
        *
        * @param unknown $method
        * @param unknown $args
        * @return mixed
        */
       public static function __callStatic($method, $args)
       {
               // Ejecuta el método estático invocado con los argumentos dados
               // sobre el objeto PDO existente.
               return call_user_func_array(array($this->_connection, $method), $args);
       }
}
```

App/Entidades/Usuario

Esta clase representa un usuario mediante sus atributos: *$nombre*, *$clave*, *$administrador* y *$activo*. Sus objetos se emplean para llevar los datos de cada usuario de la capa modelo a la capa vista y viceversa a través de la capa controlador:

***Código**: App/Entidades/Usuario*

```
<?php
namespace App\Entidades;
defined("APPPATH") OR die("Access denied");

class Usuario {
        public $nombre;
        public $clave;
        public $administrador;
        public $activo;

        public function __construct( $ nombre, $ clave, $ administrador, $ activo ) {
                $this->nombre = $ nombre;
                $this->clave = $ clave;
                $this->administrador = $ administrador;
                $this->activo = $ activo;
        }
}
```

App/Modelos/Usuarios

Esta clase es estática e implementa con sus métodos todas las operaciones relacionadas con la obtención, búsqueda, inserción, modificación y eliminación de usuarios en el origen de datos.

(*) El uso de la clase Usuarios permite aislar al controlador y al resto de la aplicación del origen de datos empleado (base de datos, fichero, servicio web…, etc)

Sus métodos básicos son:

- ***getAll()*** → Devuelve una matriz de objetos *App/Entidades/Usuario* con todos los usuarios registrados en el origen de datos. En este caso todos los registros contenidos en la tabla *USUARIOS* de la base de datos.

- ***getById($id)*** → Devuelve el objeto *App/Entidades/Usuario* correspondiente al usuario almacenado en el origen de datos con el identificador indicado. En caso retorna el objeto $usuario correspondiente al registro en la tabla *USUARIOS* cuyo nombre coincida con el indicado. En caso no encontrarse retorna NULL.

- ***insert($usuario)*** → Inserta en el origen de datos los datos del objeto *App/Entidades/Usuario* indicado como argumento. En este caso, inserta en la tabla *USUARIOS* un nuevo registro con los datos del objeto $*usuario* indicado.

- ***update($usuario)*** → Actualiza en el origen de datos los datos del objeto *App/Entidades/Usuario* indicado como argumento. En este caso, modifica aquel registro en la tabla *USUARIOS* correspondiente al objeto $*usuario* indicado.

- ***delete($id)*** → Elimina en el origen de datos el usuario con el identificador dado como argumento. En este caso, aquel registro en la tabla *USUARIOS* cuyo nombre sea el indicado.

Cada método obtiene el objeto de la clase Core/Database que representa la conexión a la base de datos y ejecuta el comando SQL correspondiente. La obtención del objeto conexión se realiza mediante el método estático *instance()* para asegurar el uso de la misma conexión y no varias en el caso de ejecutarse varios métodos de acceso a datos.

Estos métodos conforman un patrón con las operaciones básicas que debería implementar cualquier clase de gestión de datos de la capa modelo. Por ello se define el interfaz *App/Interfaces/Crud*.

***Código**: App/Interfaces/Crud*

```php
<?php
namespace App\Interfaces;
defined("APPPATH") OR die("Access denied");
interface Crud
{
        public static function getAll();
        public static function getById($id);
        public static function insert($data);
        public static function update($data);
        public static function delete($id);
}
```

Ese interfaz debe ser implementarse en todas las clases de gestión de la capa modelo.

***Código**: App/Modelo/Usuarios*

```php
<?php
namespace App\Modelos;
defined("APPPATH") OR die("Access denied");
use \Core\Database;
use \App\Entidades\Usuario;
use \App\Interfaces\Crud;

class Usuarios implements Crud
{
        public static function getAll()
        {
                $usuarios = array();
                try {
                        $db = Database::instance();
                        $sql = "SELECT * from usuarios";
                        $query = $db->run($sql);
                        // Bucle de obtención de resultados
                        while ( $reg = $query->fetch() ) {
                                // Creacion de objeto Usuario por cada registro y agregado
                                // a matriz de resultados
                                array_push($usuarios,
                                                new Usuario($reg['usuario'],
                                                        $reg['clave'],
                                                        $reg['administrador'],
                                                        $reg['activo']));
                        }
                        return $usuarios;
                } catch(\PDOException $e)  {
                        print "Error!: " . $e->getMessage();
                }
        }
                public static function getById($id)
        {
                try {
                        $db = Database::instance();
                        $sql = "SELECT * from usuarios WHERE usuario LIKE :usuario";
                        $query = $db->run($sql, [':usuario' => $id]);
                        $reg = $query->fetch();
                        return ( ($reg)?
                                        // Retorno del objeto Usuario recuperado
                                        new Usuario($reg['usuario'],
                                                    $reg['clave'],
                                                    $reg['administrador'],
                                                    $reg['activo'])
                                         // Retorno NULL si no se recuperó ningún registro
                                        :NULL );
                } catch(\PDOException $e)  {
                        print "Error!: " . $e->getMessage();
                }
        }
                public static function insert($user)
        {
                try {
                        $db = Database::instance();
                        $sql = "INSERT INTO usuarios( usuario, clave, administrador, activo )
                                        VALUES ( :usuario, :clave, :admin, :activo )";
                        $query = $db->run($sql, [':usuario' => $user->nombre,
                                                 ':clave' => $user->clave,
                                                 ':admin' => $user->administrador,
                                                 ':activo' => $user->activo ]);
                } catch(\PDOException $e)  {
                        print "Error!: " . $e->getMessage();
                }
        }
                public static function update($user)
        {
                try {
                        $db = Database::instance();
                        $sql = "UPDATE usuarios SET clave = :clave,
                                                    administrador = :admin,
                                                    activo = :activo
                                                    WHERE usuario = :usuario";
                        $query = $db->run($sql, [':usuario' => $user->nombre,
                                                 ':clave' => $user->clave,
                                                 ':admin' => $user->administrador,
                                                 ':activo' => $user->activo ]);
                }catch(\PDOException $e) {
                        print "Error!: " . $e->getMessage();
                }
        }
```

```
        public static function delete($id)
        {
                try {
                        $db = Database::instance();
                        $sql = "DELETE FROM usuarios WHERE usuario = :usuario";
                        $query = $db->run($sql, [':usuario' => $id]);
                }
                catch(\PDOException $e)
                {
                        print "Error!: " . $e->getMessage();
                }
        }

}
```

Implementación de la vista

La capa vista se encarga de generar las páginas que se envían al usuario como respuesta a sus peticiones. Para ello se emplean ***plantillas***.

Las *plantillas* son ficheros *.php* que contiene el diseño de una página web junto con inserciones de PHP donde debe aparecer el contenido dinámico. *Estas ficheros NO SON solicitados directamente por el usuario, sino que son servidos por la clase* ***vista***.

Ejemplo: Sea el siguiente fichero como plantilla de ejemplo:

```
<!DOCTYPE html>
<html>
        <head>
        <meta charset="utf-8">
        <title></title>
        </head>
        <body>
                <h1>AUTENTIFICACION</h1>
                <h2 style='background-color: #ff0000'><?= $error['mensaje']?></h2>
        <form action='<?= DOMAIN?>/controlsesion/autentificar' method='post'>
                Usuario: <input type='text' name='nombre' value='<?= $error['nombre']?>'>
                Clave: <input type='text' name='clave' value='<?= $error['clave']?>'><br/>
                <input type='submit' value='registrar'>
        </form>
    </body>
</html>
```

Las inserciones de código PHP van destinadas a mostrar los diferentes contenidos dinámicos de la página. En este caso, los valores son diferentes elementos de una matriz asociativa $error, que deberá definirse mediante la clase ***View*** (*Core/View*).

La clase Vista (*Core/View* ***)***

Esta clase estática se encarga de cargar una plantilla, ejecutar el código de PHP que contiene, y enviar el código HTML resultante al navegador del usuario.

La clase vista posee los siguientes métodos estáticos:

- ***set($name, $value)*** → Registra un valor para ser mostrado como parte del contenido dinámico de una plantilla.
- ***render($template)*** → Carga el fichero plantilla con el nombre indicado para su ejecución y envío al usuario como página resultado.

Para mostrar la plantilla anterior (***vistas/login.php***) mostrando los valores correspondientes a las inserciones de PHP, el código sería el siguiente:

```
        // Clave incorrecta.
        View::set("error", ['mensaje' => 'Contraseña incorrecta',
                            'nombre' => $nombre,
                            'clave' => $clave]);
        View::render("login");
```

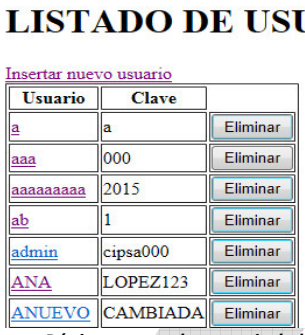
Los valores indicados en el método ***set()*** darán lugar a una matriz asociativa *$error* con valores asociados a las claves '*mensaje*', '*nombre*', y '*clave*'.

***Código**: Core/View*

```php
<?php
namespace Core;
defined("APPPATH") OR die("Access denied");

class View
{
	/**
	 * @var
	 */
	protected static $data;
	/**
	 * @var
	 */
	const VIEWS_PATH = "../App/vistas/";
	/**
	 * @var
	 */
	const EXTENSION_TEMPLATES = "php";

	/**
	 * Proyecta la pagina con el fichero de plantilla indicado y los datos registrados de previo.
	 * @param  [String]  [template name]
	 * @return [html]    [render html]
	 */
	public static function render($template)
	{
		if(!file_exists(self::VIEWS_PATH . $template . "." . self::EXTENSION_TEMPLATES))
		{
			throw new \Exception("Error: El archivo " . self::VIEWS_PATH . $template . "." .
				self::EXTENSION_TEMPLATES . " no existe", 1);
		}
		ob_start();
		if ( self::$data != NULL ) extract(self::$data);
		include(self::VIEWS_PATH . $template . "." . self::EXTENSION_TEMPLATES);
		$str = ob_get_contents();
		ob_end_clean();
		echo $str;
	}

	/**
	 * Registra datos para ser proyectados por una plantilla.
	 * @param [string] $name  [key]
	 * @param [mixed] $value [value]
	 */
	public static function set($name, $value)
	{
		self::$data[$name] = $value;
	}

}
```

Plantillas

- ***Lista*** → Plantilla que muestra una tabla con todos los usuarios registrados en el sistema y un enlace al método *"nuevo" del controlador (controlusuarios/nuevo)*. Por cada usuario se indica su nombre y clave. El nombre es un hipervínculo al método "*ver*" incluyendo el nombre del usuario en la URL, (*controlusuarios/ver/ANA*) además de un botón Eliminar que envía el nombre del usuario al método "*eliminar*" (*controlusuarios/eliminar*).



LISTADO DE USUARIOS

Insertar nuevo usuario

Usuario	Clave	
a	a	Eliminar
aaa	000	Eliminar
aaaaaaaaa	2015	Eliminar
ab	1	Eliminar
admin	cipsa000	Eliminar
ANA	LOPEZ123	Eliminar
ANUEVO	CAMBIADA	Eliminar

Página generada a partir de la vista Lista

***Código**: app/vistas/lista.php*

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title></title>
        <script src="<?= JS_SCRIPTS?>/jquery-3.0.0.min.js"></script>
                  <script>
                           $().ready(function() {
                                    $('form').submit(function(e){
                                             var user = $(this).find('input:hidden').val();
                                             var response = confirm("Desea eliminar usuario: " + user );
                                             if ( !response) e.preventDefault();
                                    });
                           });
                  </script>
    </head>
    <body>
         <h1>LISTADO DE USUARIOS</h1>
         <a href='<?=DOMAIN?>/controlusuarios/nuevo'>Insertar nuevo usuario</a><br/>
        <table border='1' class="table">
            <thead>
                <tr>
                    <th>Usuario</th>
                    <th>Clave</th>
                </tr>
            </thead>
            <tbody>
                 <?php
                 foreach ($users as $user) { ?>
                <tr>
                    <td><a href='<?= DOMAIN?>/controlusuarios/ver/<?=$user->nombre?>'><?=$user->nombre ?>
                        </a></td>
                    <td><?= $user->clave ?></td>
                    <td>
                           <form action='<?=DOMAIN?>/controlusuarios/eliminar' method='post'>
                                    <input type='hidden' name='usuario' value='<?= $user->nombre?>'>
                                    <input type='submit' value='Eliminar'>
                           </form>
                    </td>
                </tr>
                 <?php } ?>
            </tbody>
        </table>
    </body>

</html>
```

- ***Detalle*** → Plantilla que muestra un formulario con los datos de un usuario seleccionado en la lista de usuarios con la intención de modificar su clave, o su estado de activo o administrador. El nombre del usuario (que no puede modificarse) va en un campo oculto. Los valores son enviados al método "*actualizar*" del controlador (*controlusuarios/actualizar*).

  La plantilla incluye un vínculo al método "*listar*" (*controlusuarios/listar*) para retornar a la lista de usuarios.

Página generada a partir de la vista Detalle

***Código**: app/vistas/lista.php*

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title></title>
    </head>
    <body>
        <h1>DETALLE USUARIO <?= $user->nombre ?></h1>
        <form action='<?= DOMAIN?>/controlusuarios/actualizar/' method='post'>
                <input type='hidden' name='usuario' value='<?= $user->nombre?>'>
                Clave: <input type='text' name='clave' value='<?= $user->clave?>'><br/>
                Administrador: <input type='checkbox' name='administrador'
                                    <?= ($user->administrador==1)?'checked':'') ?>><br/>
                Activo: <input type='checkbox' name='activo'
                                    <?= ($user->activo==1)?'checked':'') ?>><br/>
                <input type='submit' value='modificar'>
        </form>
        <a href='<?= DOMAIN?>/controlusuarios/listar'>Volver a lista usuarios</a>
    </body>
</html>
```

- ***Nuevo*** → Plantilla que muestra un formulario con campos para introducir el nombre, clave, y estado de activo y administración de un usuario para registrarlo. El formulario envía los datos al método "*insertar*" del controlador (*controlusuarios/insertar*).

  La plantilla incluye un vínculo al método "*listar*" (*controlusuarios/listar*) para retornar a la lista de usuarios.

Página generada a partir de la vista Detalle

***Código**: app/vistas/nuevo.php*

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title></title>
    </head>
    <body>
        <h1>NUEVO USUARIO</h1>
        <h2 style='background-color: #ff0000'><?= $error['mensaje']?></h2>
        <form action='<?= DOMAIN?>/controlusuarios/insertar/' method='post'>
                Usuario: <input type='text' name='usuario' value='<?= $error['nombre']?>'>
                Clave: <input type='text' name='clave' value='<?= $error['clave']?>'><br/>
                Administrador: <input type='checkbox' name='administrador'
                                    <?=($error['administrador']==1)?'checked':'')?>><br/>
                Activo: <input type='checkbox' name='activo'
                                    <?=($error['activo']==1)?'checked':'')?>><br/>
                <input type='submit' value='registrar'>
        </form>
        <a href='<?= DOMAIN?>/controlusuarios/listar'>Volver a lista usuarios</a>
    </body>

</html>
```

Implementación del controlador

La capa controlador es la encargada de atender las peticiones de los usuarios realizando la tarea solicitada a través del modelo y los datos enviados, y emitir la página de respuesta correspondiente mediante la vista.

Recordemos que la clase enrutadora (*Core/App*) decodifica la URL de cada solicitud del usuario obteniendo los siguientes parámetros:

http://localhost/mvc/index.php?url=controlusuarios/ver/petrov

- Nombre del controlador al que va dirigida la petición (*$_controller*) = "*controlusuarios*"
- Nombre del método del controlador solicitado (*$_method*) = "*ver*"
- Parámetros para el método solicitado (*$_parameters*) = "*petrov*"

La clase controlador a la que va dirigida la petición (*Controlusuarios*) debe tener en este caso un método llamado "ver" con un parámetro de entrada (el cual en esta petición de ejemplo recibiría el valor "petrov").

Hay que entender que *las peticiones a una aplicación web MVC no van dirigidas a páginas, sino a la solicitud de tareas dirigidas a un controlador. La clase que implementa cada controlador debe definir un método para cada una de estas tareas*.

App/Controladores/ControlUsuarios

La capa controlador puede componerse de una o controladores según la complejidad de la aplicación. En este caso, empleamos un único controlador implementado en la clase *App/Controladores/ControlUsuarios*.

Esta clase dispone de los siguientes métodos:

listar() → Este método obtiene del modelo todos los usuarios registrados y muestra la tabla con sus datos al usuario empleando la plantilla *lista*.

La plantilla *lista* requiere un parámetro *$users* donde recibe una matriz de objetos *entidades/Usuario* con todos los usuarios presentes en la base de datos:

```
/**
 * Solicitud de listado de todos los usuarios
 * URL: controladorusuarios/listar
 */
public function listar()
{
        $users = Usuarios::getAll();
        View::set("users", $users);
        View::render("lista");
}
```

ver($nombre) → Este método obtiene los datos del usuario del nombre indicado como parámetro en la URL y los muestra en el formulario de edición de la plantilla *edición*.La plantilla edición requiere como parámetro *$user* el objeto *App/Entidades/Usuario* con los datos del usuario del nombre indicado:

```
/**
 * Solicitud de visualización de formulario de edicion
 * del usuario con el nombre indicado.
 * URL: controladorusuarios/ver/<nombre>
 * @param unknown $nombre
 */
public function ver( $nombre )
{
        $user = Usuarios::getById($nombre);
        View::set("user", $user);
        View::render("edicion");
}
```

nuevo() → Este método muestra el formulario de alta de usuario de la plantilla *nuevo*:

```
/**
 * Solicitud de visualización de formulario de alta
 * de usuario.
 * URL: controladorusuarios/nuevo
 */
public function nuevo() {
        View::set("error",[
                        'mensaje'=> '',
                        'nombre' => '',
                        'clave' => '',
                        'administrador' => 0,
                        'activo' => 0 ]);
        View::render("nuevo");
}
```

actualizar() → Este método recibe los valores indicados en el formulario de la plantilla *edición* mediante POST, crea el objeto *App/Entidades/Usuario* modificado y solicita su actualización en la base de datos mediante el método *update()* del modelo. Finalmente muestra al usuario la lista de usuarios actualizada llamando al método *listar():*

```
/**
 * Solicitud de actualización de usuario por los datos
 * enviados mediante POST, y muestra al usuario lista
 * actualizada de usuarios.
 * URL: controladorusuarios/actualizar
 */
public function actualizar() {
        $nombre = filter_input(INPUT_POST, "usuario");
        $clave = filter_input(INPUT_POST, "clave");
        $administrador = isset($_POST['administrador'])?1:0;
        $activo = isset($_POST['activo'])?1:0;
        $usuario = new Usuario($nombre, $clave, $administrador, $activo);
        Usuarios::update($usuario);
        $this->listar();
}
```

eliminar() → Este método elimina el usuario con el nombre pasado mediante POST. Finalmente muestra la lista de usuarios actualizada llamando al método *listar():*

```
/**
 * Solicitud de eliminacion de usuario con el nombre
 * indicado mediante POST
 * URL: controladorusuarios/eliminar
 */
public function eliminar() {
        $nombre = filter_input(INPUT_POST, "usuario");
        Usuarios::delete($nombre);
        $this->listar();

}
```

Insertar() → Este método recibe los valores indicados en el formulario de la plantilla *nuevo* mediante POST, crea el objeto *App/Entidades/Usuario* del nuevo usuario, y lo inserta en la base de datos mediante el método *insert()* del modelo. Finalmente se muestra la lista de usuarios actualizada llamando al método *listar()*.´

Si ya existe un usuario registrado con el nombre indicado *se vuelve a mostrar el formulario de la plantilla nuevo con los valores introducidos y un mensaje de error*. Para ello, la plantilla *nuevo* requiere un parámetro ***$error*** con una matriz asociativa y los siguientes elementos:

- ***mensaje*** → Mensaje de error a mostrar al usuario
- ***nombre*** → Valor introducido anteriormente en la caja de texto "nombre"
- ***clave*** → Valor introducido anteriormente en la caja de texto "clave".
- ***administrador*** → Estado anterior de la casilla "administrador"
- ***activo*** → Estado anterior de la casilla "activo"

```
        /**
         * Solicitud de insercion de usuario con los datos
         * enviuados mediante POST, y se muestra lista actualizada
         * de usuarios.
         * En caso de usuario con nombre ya existente, se muestra
         * formulario con mensaje de error.
         * URL: controladorusuarios/insertar
         */
        public function insertar() {
                $nombre = filter_input(INPUT_POST, "usuario");
                $clave = filter_input(INPUT_POST, "clave");
                $administrador = isset($_POST['administrador'])?1:0;
                $activo = isset($_POST['activo'])?1:0;
                // Comprobacion de usuario existente?
                $usuario_existente = Usuarios::getById($nombre);
                 if ( $usuario_existente == NULL ) {
                                // Usuario no existe
                                $nuevo_usuario = new Usuario($nombre, $clave, $administrador, $activo);
                                Usuarios::insert($nuevo_usuario);
                                $this->listar();
                 } else {
                                // Usuario ya existe -> ERROR.
                                View::set("error",[
                                                'mensaje'=> "Usuario $nombre ya existe",
                                                'nombre' => $nombre,
                                                'clave' => $clave,
                                                'administrador' => $administrador,
                                                'activo' => $activo ]);
                                View::render("nuevo");
                 }
        }
```

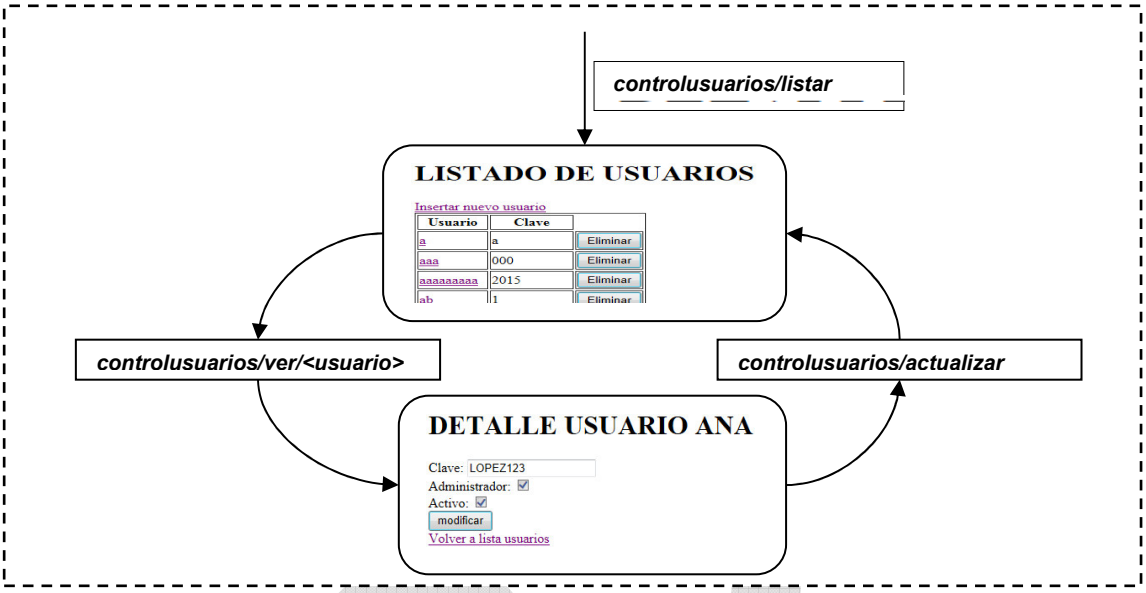
Código: *App/Controladores/ControlUsuarios*

```
<?php
namespace App\Controladores;
defined("APPPATH") OR die("Access denied");
use \Core\View;
use \App\Modelos\Usuarios;
use \App\Entidades\Usuario;

class ControlUsuarios
{
        /**
         * Solicitud de listado de todos los usuarios
         * URL: controladorusuarios/listar
         */
        public function listar()
        {
                $users = Usuarios::getAll();
                View::set("users", $users);
                View::render("lista");
        }
```

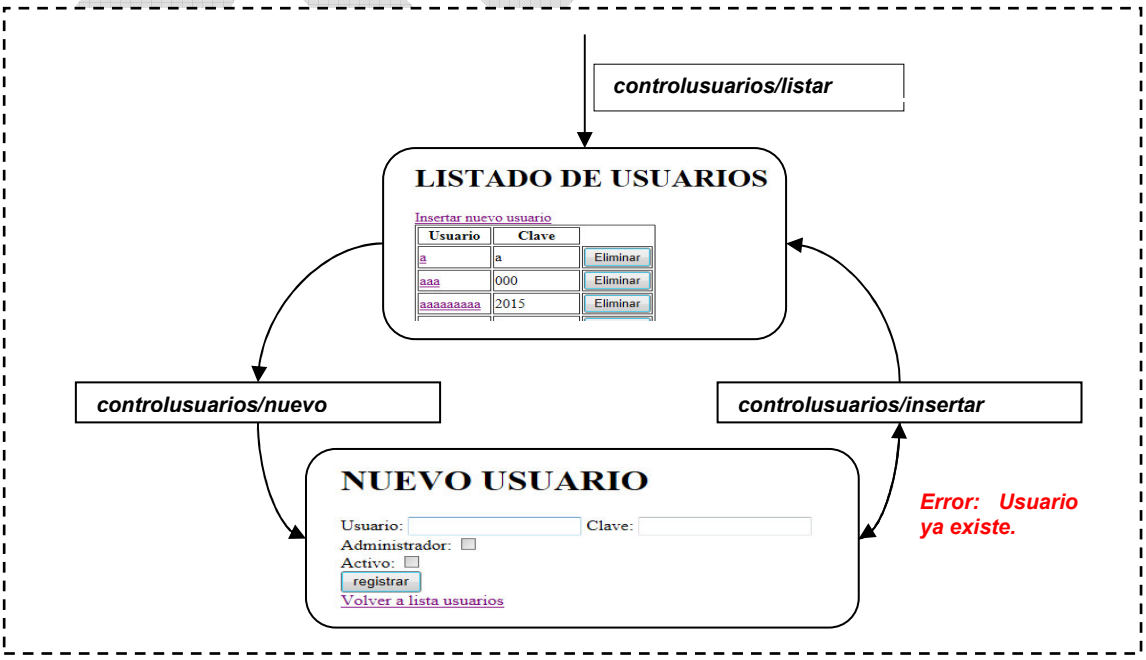
```php
        /**
         * Solicitud de visualización de formulario de edicion
         * del usuario con el nombre indicado.
         * URL: controladorusuarios/ver/<nombre>
         * @param unknown $nombre
         */
        public function ver( $nombre )
        {
                $user = Usuarios::getById($nombre);
                View::set("user", $user);
                View::render("edicion");
        }
        /**
         * Solicitud de visualización de formulario de alta
         * de usuario.
         * URL: controladorusuarios/nuevo
         */
        public function nuevo() {
                View::set("error",[
                                'mensaje'=> '',
                                'nombre' => '',
                                'clave' => '',
                                'administrador' => 0,
                                'activo' => 0 ]);
                View::render("nuevo");
        }
        /**
         * Solicitud de actualización de usuario por los datos
         * enviados mediante POST, y muestra al usuario lista
         * actualizada de usuarios.
         * URL: controladorusuarios/actualizar
         */
        public function actualizar() {
                $nombre = filter_input(INPUT_POST, "usuario");
                $clave = filter_input(INPUT_POST, "clave");
                $administrador = isset($_POST['administrador'])?1:0;
                $activo = isset($_POST['activo'])?1:0;
                $usuario = new Usuario($nombre, $clave, $administrador, $activo);
                Usuarios::update($usuario);
                $this->listar();
        }
        /**
         * Solicitud de eliminacion de usuario con el nombre
         * indicado mediante POST
         * URL: controladorusuarios/eliminar
         */
        public function eliminar() {
                $nombre = filter_input(INPUT_POST, "usuario");
                Usuarios::delete($nombre);
                $this->listar();
        }
        /**
         * Solicitud de insercion de usuario con los datos
         * enviuados mediante POST, y se muestra lista actualizada
         * de usuarios.
         * En caso de usuario con nombre ya existente, se muestra
         * formulario con mensaje de error.
         * URL: controladorusuarios/insertar
         */
        public function insertar() {
                $nombre = filter_input(INPUT_POST, "usuario");
                $clave = filter_input(INPUT_POST, "clave");
                $administrador = isset($_POST['administrador'])?1:0;
                $activo = isset($_POST['activo'])?1:0;
                // Comprobacion de usuario existente?
                $usuario_existente = Usuarios::getById($nombre);
                if ( $usuario_existente == NULL ) {
                        // Usuario no existe
                        $nuevo_usuario = new Usuario($nombre, $clave, $administrador, $activo);
                        Usuarios::insert($nuevo_usuario);
                        $this->listar();
                } else {
                        // Usuario ya existe -> ERROR.
                        View::set("error",[
                                        'mensaje'=> "Usuario $nombre ya existe",
                                        'nombre' => $nombre,
                                        'clave' => $clave,
                                        'administrador' => $administrador,
                                        'activo' => $activo ]);
                        View::render("nuevo");
                }
        }
}
```

Flujo de la aplicación

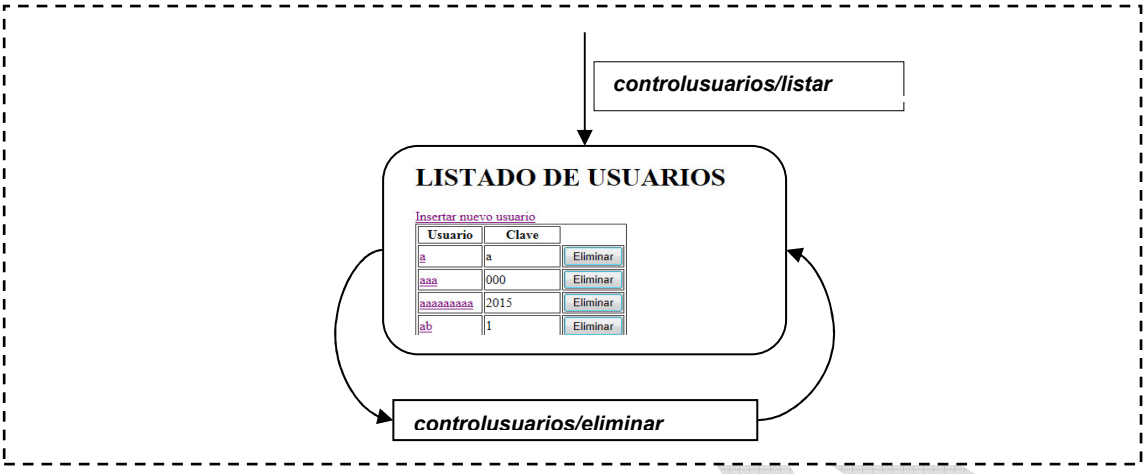
Flujo de modificación de usuario: Desde la pantalla listado de usuarios, el nombre de cada usuario es un enlace a URL: (*controlusuarios/ver/<nombre_usuario>*). Este enlace muestra el formulario de modificación de datos de usuario. El botón "*modificar*" envía los datos a URL: (*controladorusuarios/actualizar*) mediante POST, que actualiza datos y vuelve a mostrar lista usuarios. El enlace *"<Volver a lista usuarios>"* está dirigido a URL: (*controlusuarios/listar*).



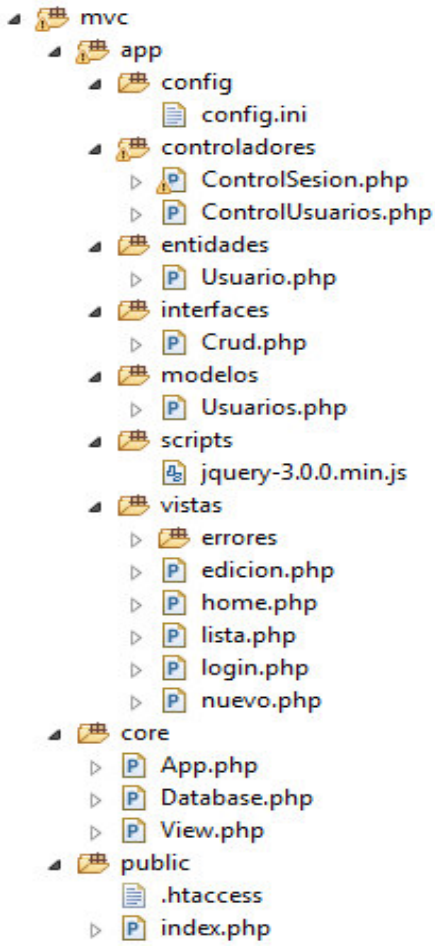
Flujo de inserción de usuario: Desde la pantalla listado de usuarios, el enlace "Insertar nuevo usuario" está dirigido a URL: (*controlusuario/nuevo*), que muestra el formulario de inserción de datos de nuevo usuario. El botón "Registrar" envía datos a URL: (*controlusuarios/insertar*) mediante POST. Si ya existe usuario con el nombre indicado se muestra nuevamente el formulario de alta con los datos introducidos previamente y el mensaje de error. En caso contrario, se almacena y se muestra la lista de usuarios actualizada.



Flujo de eliminación de usuario: Desde la pantalla listado de usuarios, el botón "*Eliminar*" a la derecha de cada usuario envía el nombre del usuario a la URL: (*controlusuarios/eliminar*) mediante POST. El usuario se elimina de la base de datos y se muestra la lista de usuarios actualizada.



Estructura final del proyecto



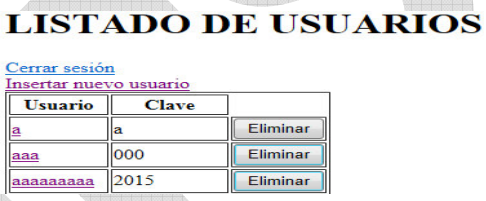
Ejercicio:

Complementa la aplicación añadiendo un control de usuarios.

Al entrar en la aplicación si el usuario no ha iniciado sesión debe mostrarse un formulario de registro:

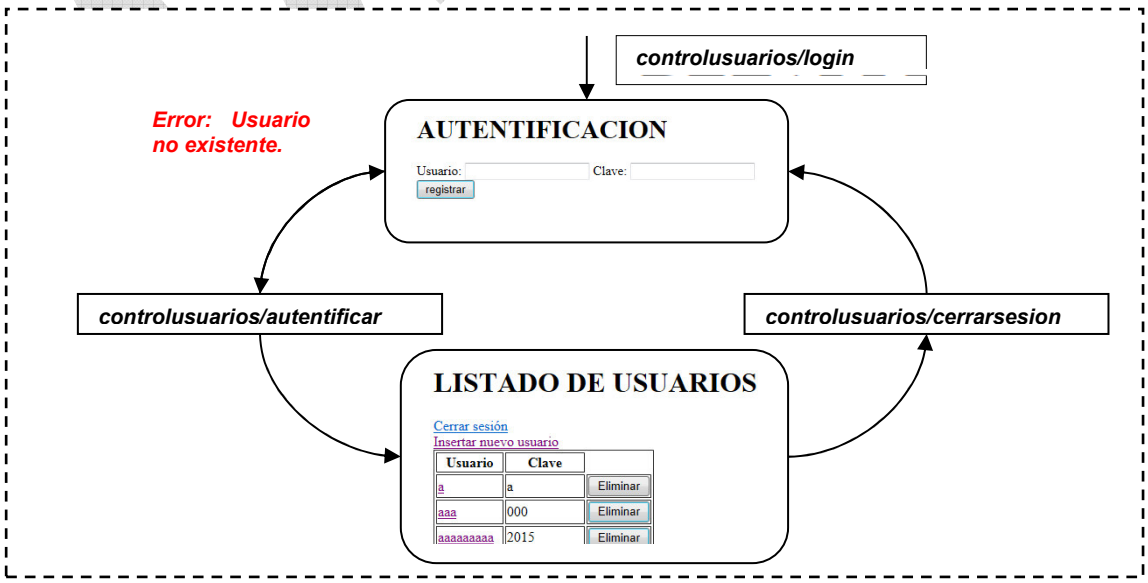
Si el usuario introduce un nombre y contraseña registrados en la base de datos, se inicia sesión y se le muestra la lista de usuarios. En caso contrario se le vuelve a mostrar el formulario con los valores anteriores y un mensaje de error:

En la página de listado de usuarios añadir un vínculo “Cerrar sesión” para que el usuario pueda cerrar sesión:



Debe asegurarse que la solicitud de cualquier URL por un usuario sin haber iniciado sesión, muestre como el formulario de inicio de sesión.

Flujo de ejecución con inicio de sesión



Especificaciones:

Será preciso crear una nueva plantilla "login" con el formulario de inicio de sesión y porciones de PHP para mostrar valores en las cajas de texto y un mensaje de error.

Será preciso añadir nuevos métodos a la clase (*App/Controladores/ControlUsuarios*):

- ***login()*** → Muestra el formulario de inicio de sesión mediante la plantilla *login*.
- ***autentificar()*** → Recibe el nombre de usuario y contraseña del formulario de inicio de sesión y comprueba está registrado en la tabla de usuarios de la base de datos. Si está registrado, almacena el objeto *App/Entidades/Usuario* resultante en la sesión y muestra el listado de usuarios. Si no, vuelve a mostrar el formulario de la plantilla *login* añadiendo los valores introducidos y el mensaje de error "*Usuario desconocido*."
- ***cerrarsesion*** → Elimina la sesión del usuario y muestra el formulario de inicio de sesión mediante la plantilla *login*.

En la clase enrutador (*Core/App*) deberá redirigirse a *controlusuarios/login* todas las peticiones sin sesión.