# Desarrolla tu propio MVC con PHP y POO

Fuente: http://uno-de-piera.com/desarrolla-tu-propio-mvc-con-php-y-poo/

20 septiembre, 2015 israel965 Php orientado a objetos.

Creo que siempre surge la idea de crear tu propia aplicación utilizando **MVC con PHP y POO** simplemente para saber cómo se puede implementar la lógica de una aplicación **MVC (Modelo Vista Controlador)**, procesar controladores y métodos a través de la url, pasar datos a las vistas, conectar los modelos, y todo esto, sin que al final sea imposible de entender y escalar.

Aquí vamos a crear un ejemplo completo, debe quedar claro que no vamos a desarrollar un framework ni nada parecido, simplemente intentar entender cómo podemos desarrollar aplicaciones sin frameworks utilizando el marco de trabajo MVC con Programación Orientada a objetos (POO).

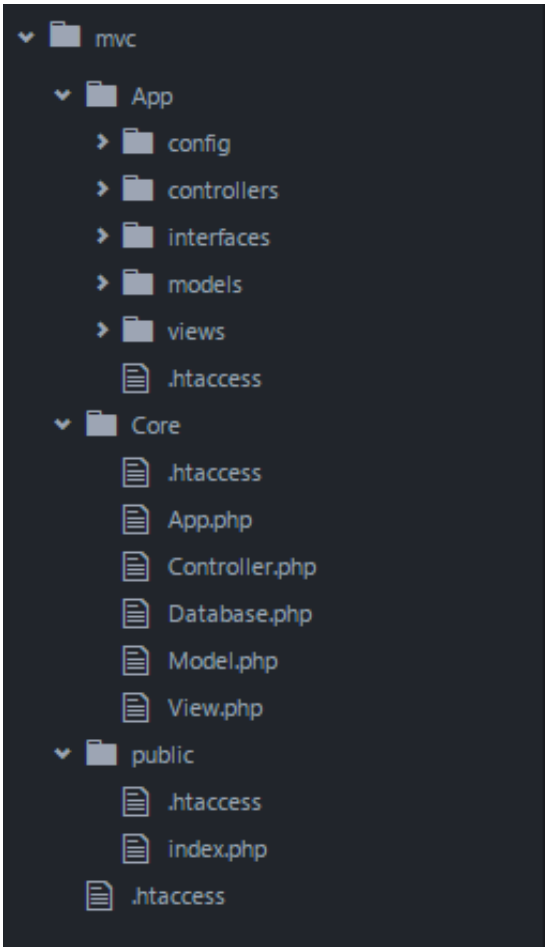
## Características de nuestra aplicación MVC



- Autocarga de clases con la función spl_autoload_register utilizando namespaces.
- Aplicación trabajando con POO utilizando namespaces.
- Urls amigables con un sencillo .htaccess.
- Uso de interfaces para ejemplo de crud.
- Conexión con base de datos utilizando PDO
- Archivo de configuración con extensión .ini.
- Control de acceso a directorios/archivos no permitidos.

## Estructura de nuestra aplicación MVC

A continuación adjunto una imagen de la estructura de nuestro proyecto para que vayas preparando el terreno si te interesa seguir el ejemplo.




# Autocarga de clases con spl_autoload_register

La aplicación será lanzada desde el archivo /public/index.php, aquí será donde vamos a hacer la autocarga de clases y finalmente lanzar la aplicación, de momento, vamos a definir la autocarga con el siguiente código.

PHP

```php
<?php
//directorio del proyecto
define("PROJECTPATH", dirname(__DIR__));

//directorio app
define("APPPATH", PROJECTPATH . '/App');

//autoload con namespaces
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');
```

Con esa sencilla función, todos los archivos que estén dentro del proyecto y cualquier directorio serán autocargados para poder utilizarlos donde necesitemos, y lo más importante, **utilizando namespaces**.

# HTACCESS para tener urls amigables

Ahora abre el .htaccess del directorio public y añade la siguiente lógica para poder trabajar con urls amigables.

Shell

```
Options -Multiviews

RewriteEngine On

RewriteCond %{REQUEST_FILENAME} !-d
RewriteCond %{REQUEST_FILENAME} !-f

RewriteRule ^(.+)$ index.php?url=$1 [QSA]
```

De esta forma tan sencilla, en la variable $_GET[‘url’] tendremos la url segmentada a partir de public, por ejemplo, la url http://localhost:8080/php/mvc/public/home/admin/1 nos devolverá home/admin/1, así podremos saber el controlador, método y parámetros que debemos procesar para lanzar nuestra aplicación.

# Procesar los controladores

Ahora debemos solucionar un componente imprescindible de nuestra aplicación, y es resolver las peticiones, controlador, método y parámetros, para ello, abre el archivo Core/App.php y añade el siguiente código.

PHP

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

class App
{
    /**
    * @var
    */
    private $_controller;

    /**
    * @var
    */
    private $_method = "index";

    /**
    * @var
    */
    private $_params = [];

```

```php
    /**
     * @var
     */
    const NAMESPACE_CONTROLLERS = "\App\Controllers\\";

    /**
     * @var
     */
    const CONTROLLERS_PATH = "../App/controllers/";

    /**
     * [__construct description]
     */
    public function __construct()
    {
        //obtenemos la url parseada
        $url = $this->parseUrl();

        //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 los parámetros del método
            unset($url[0]);
        }
        else
        {
            include APPPATH . "/views/errors/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, $url[1]))
            {
                //eliminamos el método de url, así sólo nos quedaran los parámetros del método
                unset($url[1]);
            }
            else
            {
                throw new \Exception("Error Processing Method {$this->_method}", 1);
            }
        }
        //asociamos el resto de segmentos a $this->_params para pasarlos al método llamado, por c
        $this->_params = $url ? array_values($url) : [];
    }

    /**
     * [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));
        }
    }

    /**
     * [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);
    }

    /**
     * [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');
    }

    /**
     * [getController Devolvemos el controlador actual]
     * @return [type] [String]
     */
    public function getController()
    {
        return $this->_controller;
    }

    /**
     * [getMethod Devolvemos el método actual]
     * @return [type] [String]
     */
    public function getMethod()
    {
        return $this->_method;
    }

    /**
     * [getParams description]
     * @return [type] [Array]
     */
    public function getParams()
    {
        return $this->_params;
    }
}
```

Ahora ya tenemos lista toda la lógica para que nuestra aplicación haga la carga de controladores a través de la url al igual que hacen los actuales frameworks.

Para probar que esto funciona correctamente, vamos a crear un controlador dentro de App/controllers llamado Home.php y añade el siguiente código.

PHP

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

```

```php
class Home
{
    public function saludo($nombre)
    {
        echo "Hola " . $nombre;
    }
}
```

Si te fijas, siempre hacemos uso de namespaces, es muy importante su uso para evitar que nuestras clases colisionen entre ellas, de esta forma, podemos tener clases con el mismo nombre en diferentes namespaces.

Comprobamos si está definida la constante APPPATH, si no es así, es que están tratando de acceder directamente al script, y eso no se puede permitir ya que se rompería la aplicación al completo.

Finalmente, creamos la clase Home y el método saludo con un parámetro $nombre, si accedes a public/home/saludo/tunombre verás que no sucede nada, abre el archivo public/index.php y añade las siguientes dos líneas al final.

PHP

```php
//instanciamos la app
$app = new \Core\App;

//lanzamos la app
$app->render();
```

Si ahora vuelves a visitar la misma url verás que ya pone Hola tunombre, si es así, todo está trabajando correctamente.

# Renderizar vistas con variables

Un factor bien importante dentro del MVC son las vistas, vamos a crear la clase View desde donde tendremos la lógica necesaria para poder renderizar templates con variables, así que crea el archivo Core/View.php y añade el siguiente código.

PHP

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

class View
{
    /**
     * @var
     */
    protected static $data;

    /**
     * @var
     */
    const VIEWS_PATH = "../App/views/";

```

```php
    /**
     * @var
     */
    const EXTENSION_TEMPLATES = "php";

    /**
     * [render views with data]
     * @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::EXTEN
        }

        ob_start();
        extract(self::$data);
        include(self::VIEWS_PATH . $template . "." . self::EXTENSION_TEMPLATES);
        $str = ob_get_contents();
        ob_end_clean();
        echo $str;
    }

    /**
     * [set Set Data form views]
     * @param [string] $name  [key]
     * @param [mixed] $value [value]
     */
    public static function set($name, $value)
    {
        self::$data[$name] = $value;
    }
}
```

Aquí simplemente definimos la lógica necesaria para poder imprimir las vistas pasando variables que nosotros previamente definamos en nuestro controlador, para ello, vamos a modificar el método controlador Home.php.

PHP

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

use \Core\View;

class Home
{
    public function saludo($nombre)
    {
        View::set("name", $nombre);
        View::set("title", "Custom MVC");
        View::render("home");
    }
}
```

Cómo puedes ver, hacemos uso de \Core\View con la sentencia use, finalmente utilizamos los métodos set y render de la clase View para renderizar la vista home, que por defecto está en el directorio views, así que crea el archivo home.php dentro de views con el siguiente código dentro.

PHP

```php
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title><?php echo $title ?></title>
    </head>
    <body>
        Hola <?php echo $name ?>
    </body>
</html>
```

Si visitas de nuevo el navegador verás Hola tunombre y el título que hemos definido en el controlador.

# Conectar con los modelos

Ahora que ya tenemos la V y la C del **MVC** vamos a definir los modelos para tener todos los componentes listos, realmente, los modelos pueden ser alojados en cualquier ubicación, aunque para ser consecuentes los vamos a definir dentro del directorio App/models, así que crea el modelo User.php en dicho directorio con el siguiente contenido.

PHP

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

class User
{
    public static function getAll()
    {
        return ["id" => 1, "nombre" => "Israel"];
    }
}
```

Ahora modifica el controlador Home.php de la siguiente forma.

PHP

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

use \Core\View,
    \App\Models\User;

class Home
{
    public function saludo($nombre)
    {
        View::set("name", $nombre);
        View::set("title", "Custom MVC");
        View::render("home");
    }
```

```
16
17      public function users()
18      {
19          $users = User::getAll();
20          print_r($users);
21      }
22  }
```

Lo primero que hacemos es a través de la sentencia use añadir el modelo User, una vez hecho, ya podemos utilizarlo de la forma en la que lo hacemos en el método users, así de sencillo.

Si visitas la url public/home/users verás cómo se imprimen los datos.

# Conectar con la base de datos

Antes de crear la lógica de la conexión, te dejo el código para crear la base de datos con una tabla usuarios.

PgSQL

```
1  CREATE DATABASE IF NOT EXISTS mvc;
2
3  use mvc;
4
5  -- MySQL dump 10.13  Distrib 5.6.24, for Win64 (x86_64)
6  --
7  -- Host: 127.0.0.1    Database: mvc
8  -- ------------------------------------------------------
9  -- Server version  5.6.25
10
11 /*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
12 /*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
13 /*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
14 /*!40101 SET NAMES utf8 */;
15 /*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
16 /*!40103 SET TIME_ZONE='+00:00' */;
17 /*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
18 /*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_C
19 /*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO
20 /*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;
21
22 --
23 -- Table structure for table `usuarios`
24 --
25
26 DROP TABLE IF EXISTS `usuarios`;
27 /*!40101 SET @saved_cs_client     = @@character_set_client */;
28 /*!40101 SET character_set_client = utf8 */;
29 CREATE TABLE `usuarios` (
30   `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
31   `nombre` varchar(45) DEFAULT NULL,
32   PRIMARY KEY (`id`)
33 ) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=latin1;
34 /*!40101 SET character_set_client = @saved_cs_client */;
35
36 --
37 -- Dumping data for table `usuarios`
38 --
```

```
39
40 LOCK TABLES `usuarios` WRITE;
41 /*!40000 ALTER TABLE `usuarios` DISABLE KEYS */;
42 INSERT INTO `usuarios` VALUES (1,'Israel'),(2,'Juan');
43 /*!40000 ALTER TABLE `usuarios` ENABLE KEYS */;
44 UNLOCK TABLES;
45 /*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;
46
47 /*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
48 /*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
49 /*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
50 /*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
51 /*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
52 /*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
53 /*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;
54
55 -- Dump completed on 2015-09-20 11:03:13
```

Ahora crea el archivo Core/Database.php con el siguiente código, donde simplemente definimos la conexión con PDO utilizando singleton.

PHP

```php
1  <?php
2  namespace Core;
3  defined("APPPATH") OR die("Access denied");
4
5  use \Core\App;
6
7  /**
8   * @class Database
9   */
10 class Database
11 {
12
13     /**
14     * @desc nombre del usuario de la base de datos
15     * @var $_dbUser
16     * @access private
17     */
18     private $_dbUser;
19
20     /**
21     * @desc password de la base de datos
22     * @var $_dbPassword
23     * @access private
24     */
25     private $_dbPassword;
26
27     /**
28     * @desc nombre del host
29     * @var $_dbHost
30     * @access private
31     */
32     private $_dbHost;
33
34     /**
35     * @desc nombre de la base de datos
36     * @var $_dbName
37     * @access protected
38     */
39     protected $_dbName;
40
```

```php
    /**
    * @desc conexión a la base de datos
    * @var $_connection
    * @access private
    */
    private $_connection;

    /**
    * @desc instancia de la base de datos
    * @var $_instance
    * @access private
    */
    private static $_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->_connection->setAttribute(\PDO::ATTR_ERRMODE, \PDO::ERRMODE_EXCEPTION
          $this->_connection->exec("SET CHARACTER SET utf8");
       }
       catch (\PDOException $e)
       {
          print "Error!: " . $e->getMessage();
          die();
       }
    }

    /**
     * [prepare]
     * @param  [type] $sql [description]
     * @return [type]      [description]
     */
    public function prepare($sql)
    {
        return $this->_connection->prepare($sql);
    }

    /**
     * [instance singleton]
     * @return [object] [class database]
     */
    public static function instance()
    {
        if (!isset(self::$_instance))
        {
            $class = __CLASS__;
            self::$_instance = new $class;
        }
        return self::$_instance;
    }

    /**
     * [__clone Evita que el objeto se pueda clonar]
     * @return [type] [message]
```

```
     */
    public function __clone()
    {
        trigger_error('La clonación de este objeto no está permitida', E_USER_ERROR);
    }
}
```

Cómo puedes ver simplemente es una conexión con una base de datos, pero tenemos un detalle en el constructor, y es el uso de un archivo de configuración .ini que consumimos a través de un método de la clase Core/App.php, dicho archivo se llama config.ini y están en el directorio App/config, créalo y añade el siguiente código.

PHP

```
[database]
host     = localhost
user     = root
password = admin
database = mvc
```

# Pasar datos de la base de datos a la vista

Para darle más utilidad a nuestra aplicación, vamos a conectar desde un nuevo modelo que también se va a llamar User, aunque esté estará dentro del directorio App/models/Admin con su respectivo espacio de nombres, así veremos que todo funciona perfectamente.

Crea el modelo User en el directorio App/models/Admin y añade el siguiente código.

PHP

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

use \Core\Database;

class User
{
    public static function getAll()
    {
        try {
            $connection = Database::instance();
            $sql = "SELECT * from usuarios";
            $query = $connection->prepare($sql);
            $query->execute();
            return $query->fetchAll();
        }
        catch(\PDOException $e)
        {
            print "Error!: " . $e->getMessage();
        }
    }
}
```

Para poder utilizar este modelo, vamos al controlador Home y vamos a hacer una pequeña

modificación.

PHP

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

use \Core\View,
    \App\Models\User,
    \App\Models\Admin\User as UserAdmin;

class Home
{
    public function saludo($nombre)
    {
        View::set("name", $nombre);
        View::set("title", "Custom MVC");
        View::render("home");
    }

    public function users()
    {
        $users = UserAdmin::getAll();
        View::set("users", $users);
        View::set("title", "Custom MVC");
        View::render("users");
    }
}
```

Lo primero que hacemos es añadir el espacio de nombres \App\Models\Admin\User y le asignamos el alias UserAdmin para no colisionar con el modelo User, después simplemente modificamos el método users para obtener todos los usuarios de la base de datos y pasarlos a la vista.

Ahora crea el archivo App/views/users.php y añade el siguiente código.

PHP

```php
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title><?php echo $title ?></title>
    </head>
    <body>
        <table class="table">

            <thead>
                <tr>
                    <th>
                        Id
                    </th>
                    <th>
                        Nombre
                    </th>
                </tr>
            </thead>
            <tbody>
                <?php
                foreach ($users as $user)
                {
                ?>
```

```
            <tr>
                <td><?php echo $user["id"] ?></td>
                <td><?php echo $user["nombre"] ?></td>
            </tr>
            <?php
            }
            ?>
        </tbody>
    </table>
</body>
</html>
```

Si accedes a la url public/home/users verás que aparecen todos los usuarios que existen en la base de datos.

# Utilizando interfaces

Ahora crea dentro del directorio interfaces el archivo Crud.php y añade el siguiente código.

PHP

```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);
}
```

Simplemente definimos una interfaz con algunos métodos para poder implementar donde queramos, nosotros vamos a implementar esta interfaz en el modelo App/models/Admin/User.php, así que abre el archivo y modifica su contenido por el siguiente.

PHP

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

use \Core\Database;
use \App\Interfaces\Crud;

class User implements Crud
{
    public static function getAll()
    {
        try {
            $connection = Database::instance();
            $sql = "SELECT * from usuarios";
            $query = $connection->prepare($sql);
            $query->execute();
            return $query->fetchAll();
```

```
18          }
19          catch(\PDOException $e)
20          {
21              print "Error!: " . $e->getMessage();
22          }
23      }
24
25      public static function getById($id)
26      {
27          try {
28              $connection = Database::instance();
29              $sql = "SELECT * from usuarios WHERE id = ?";
30              $query = $connection->prepare($sql);
31              $query->bindParam(1, $id, \PDO::PARAM_INT);
32              $query->execute();
33              return $query->fetch();
34          }
35          catch(\PDOException $e)
36          {
37              print "Error!: " . $e->getMessage();
38          }
39      }
40
41      public static function insert($user)
42      {
43
44      }
45
46      public static function update($user)
47      {
48
49      }
50
51      public static function delete($id)
52      {
53
54      }
55  }
```

Con las líneas use \App\Interfaces\Crud y class User implements Crud ya estamos implementando nuestra interfaz Crud, por ende, debemos definir todos los métodos declarados para no tener errores.

De esta forma tan sencilla podemos crear una completa aplicación utilizando **MVC con PHP** utilizando programación orientada a objetos, el resto ya es cosa de cada uno, todo el código del ejemplo final está en mi repositorio de github, a continuación dejo el enlace.

Código en Github