**Introducción al uso del Framework “SoftTortilla”:**

**Introducción:**

El siguiente documento intenta explicar de manera clara y sencilla, qué es, para qué sirve y cómo se utiliza el Framework “SoftTortilla” desarrollado en PHP bajo el patrón MVC.

**Referencia de abreviaciones y demás:**

FR: Framework

**¿Qué es un Framework?**

La palabra inglesa "framework" (marco de trabajo) define, en términos generales, un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar.

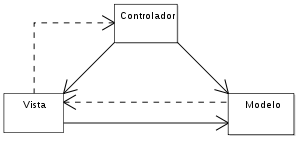
En el desarrollo de software, un framework o infraestructura digital, es una estructura conceptual y tecnológica de soporte definido, normalmente con artefactos o módulos de software concretos, que puede servir de base para la organización y desarrollo de software. Típicamente, puede incluir soporte de programas, bibliotecas, y un lenguaje interpretado, entre otras herramientas, para así ayudar a desarrollar y unir los diferentes componentes de un proyecto.

Representa una arquitectura de software que modela las relaciones generales de las entidades del dominio, y provee una estructura y una especial metodología de trabajo, la cual extiende o utiliza las aplicaciones del dominio.

No es más que una base de programación que atiende a sus descendientes (manejado de una forma estructural y/o en cascada), posibilitando cualquier respuesta ante las necesidades de sus miembros, o en secciones de una aplicación, satisfaciendo así las necesidades más comunes del programador.

**Arquitectura**

Dentro de este aspecto, vamos a basarnos en el modelo MVC (Controlador => Modelo => Vista), ya que debemos fragmentar nuestra programación. Tenemos que contemplar estos aspectos básicos en cuanto a la implementación de nuestro sistema:



**Modelo**

Este miembro del controlador maneja las operaciones lógicas, y de manejo de información (previamente enviada por su ancestro), para resultar de una forma explicable y sin titubeos. Cada miembro debe ser meticulosamente llamado, con su correcto nombre y en principio, con su verdadera naturaleza: el manejo de información, su complementación directa.

**Vista**

Al final, a este miembro de la familia le corresponde dibujar, o expresar la última forma de los datos: la interfaz gráfica que interactúa con el usuario final del programa (GUI). Después de todo, a este miembro le toca evidenciar la información obtenida hasta hacerla llegar al controlador. Solo (e inicialmente), nos espera demostrar la información.

**Controlador**

Con este apartado podemos controlar el acceso (incluso todo) a nuestra aplicación, y esto puede incluir: archivos, scripts, y/o programas; cualquier tipo de información que permita la interfaz. Así, podremos diversificar nuestro contenido de forma dinámica, y estática (a la vez); pues, sólo debemos controlar ciertos aspectos (como se ha mencionado antes).

**Estructura**

Dentro del controlador, modelo o vista podemos manejar lo siguiente: datos. Depende de nosotros como interpretar y manejar estos 'datos'. Ahora, sabemos que el único dato de una dirección estática web es: conseguir un archivo físico en el disco duro o de Internet, etc. e interpretado o no, el servidor responde.

El modelo, al igual que el controlador y la vista, maneja todos los datos que se relacionen consigo (solo es el proceso medio de la separación por capas que ofrece la arquitectura MVC). Y sólo la vista, puede demostrar dicha información. Con lo cual ya hemos generado la jerarquía de nuestro programa: Controlador, Modelo y Vista.

**Sobre SoftTortilla**:

El Framework está desarrollado completamente en lenguajes de tecnologías web (PHP, HTML5, CSS3, JavaScript, etc.), se especificarán más adelante otras cuestiones tales como el lenguaje utilizado para el manejo de plantillas de tipo “Smarty” y demás cuestiones.

En este caso, si bien SoftTortilla está basado en el modelo MVC. Contiene también carpetas que guardan los “Módulos”, los cuales poseen independientemente sus Modelos, sus Controladores y sus Vistas.

**Estructura del Framework:**

La estructura del Framework se basa en las siguientes carpetas y archivos

* app(Carpeta que contiene los archivos “Base” del FW, el corazón o core de la aplicación).
  + Acl.php (Contiene la clase para manejo de acceso multiusuario con privilegios).
  + Autoload.php (Aquí se registran las funciones de la “Autocarga”).
  + Bootstrap.php (Clase que recibe los datos desde el Request.php y los envía al controlador solicitado).
  + Config.php (Archivo de configuración).
  + Database.php (contiene la clase descendiente de PDO (nativa de PHP) para la conexión y manejo de bases de datos)
  + Hash.php (contiene la clase encargada de generar un “hash” para mayor seguridad en contraseñas).
  + Model.php (Contiene una clase abstracta de la que heredan todos los modelos).
  + Registry.php (Contiene la clase encargada de utilizar el patrón singleton, para no duplicar clases innecesarias).
  + Request.php (Contiene la clase que procesa la URL para llamar al módulo, el controlador, el método y sus argumentos).
  + Session.php (Contiene la clase que maneja las sesiones, sus variables, etc.).
  + View.php (Contiene la clase abstracta de la que heredan todas las vistas).
  + Widget.php (Contiene la clase abstracta de la que heredan todos los widget).
* controllers (Carpeta que contiene todos los controladores que no dependen de ningún módulo).
* libs (Carpeta donde se guardan las librerías externas, ya programadas para reutilizar en nuestra aplicación.
* models (Carpeta que contiene todos los modelos que no dependen de ningún módulo).
* modules (Carpeta que contiene todos los módulos).
  + [Nombre de módulo].
    - controllers (Carpeta que contiene los controladores que dependen del módulo al cual pertenecen).
    - models (Carpeta que contiene los controladores que dependen del módulo al cual pertenecen).
    - views (Carpeta que contiene las vistas que dependen del módulo al cual pertenecen).
* public (Carpeta que contiene todos los archivos que pueden compartir distintos módulos o controladores).
  + css
  + js
  + files
  + img
* publice (Carpeta donde se suben todos los archivos desde la computadora).
* tmp (Carpeta que usa el motor de plantillas Smarty, para armar los temporales que se ejecutan al momento de ingresar a una sección).
  + cache (Carpeta que hace de suerte de caché para el motor)
  + template (Carpeta donde se generan los archivos temporales que generan las vistas que se motrará por pantalla)
* views (Carpeta que contiene los archivos propios de cada vista, ya sea de las plantillas por defectos preprogramadas, o las propias de cada sección, más las vistas del contenido del resto de las secciones.
  + layout (Carpeta que contiene los templates preprogramados para las distintas secciones).
    - default (Carpeta que contiene el template por defecto, si no se selecciona ninguno).
      * css
      * img
      * js
      * config.php (Contiene una función que devuelve un array con las posiciones que maneja la plantilla).
      * template.tpl (Mezcla de HTML, con código Smarty que contiene la plantilla que utilizará la vista).
* widget (Carpeta que contiene todos los Widgets, sus modelos y sus vistas).
  + models (Carpeta que contiene todos los modelos que corresponden por lo menos a un Widget).
  + views (Carpeta que contiene todas las vistas que corresponden por lo menos a un Widget).

**Forma de trabajo del Framework:**

Antes de comenzar con la explicación de la forma de trabajo del Framework vamos a explicar qué es el fichero “.htaccess”:

*“Un fichero .htaccess (hypertext access), también conocido como archivo de configuración distribuida, es un fichero especial, popularizado por el Servidor HTTP Apache que permite definir diferentes directivas de configuración para cada directorio (con sus respectivos subdirectorios) sin necesidad de editar el archivo de configuración principal de Apache.”*

El framework trabaja siempre interpretando la URL, la cual debe contener cualquiera de los siguientes formatos:

“http://www.misitio.com/modulo/controlador/metodo/argumento”

“http://www.misitio.com/controlador/metodo/argumento”

Esto se debe a que el archivo .htaccess está configurado de tal manera que, a no ser que exista un archivo físicamente en la ruta indicada, nuestro servidor interprete que estamos llamando al archivo index.php pasándole por get un parámetro llamado url: es decir si nuestra url es http://midominio.algo/controlador/metodo/argumentos, empieza buscando si existen dichos directorios y dentro de ellos el archivo que se solicita, al no ser así llama al archivo index.php?url=controlador/metodo/argumentos.

**El index.php:**

El archivo index.php es el encargado de recibir la url que se pasa por get, y enviarla a la clase Boostrap que se encarga de procesarla, cabe destacar que si el método posee más de un argumento estos se pasan por la misma url separados por “/”; Ej.: http://www.misitio.com/modulo/controlador/metodo/arg1/arg2/arg3.

Para ello comienza haciendo las siguientes acciones. Define 3 constantes que corresponden al separador de directorio (generalmente el símbolo barra “/”), la ruta raíz de la aplicación y el directorio de la carpeta “app”. Abre un bloque try en el que carga dos archivos “Autoload.php” y “Config.php”, inicia la sesión de navegación, genera las instancias perteneciente a la clase “Registry”, la cual utiliza el patrón Singleton, para que no vuelvan a ser instanciadas (ellas son, el mismo Registry, Request, Database y Acl). Luego corre el método “run” de la clase Boostrap enviando el requerimiento (instanciado un par de líneas más arriba). De no lograr todo esto, lanza una Excepción.

**El core de la aplicación (la carpeta “app”):**

Como mencionamos anteriormente, la carpeta “app” contiene todos los archivos correspondientes al corazón del funcionamiento del Framework, con clases y métodos encargados de procesar los distintos requerimientos que van surgiendo al tiempo de la ejecución de una aplicación, por orden alfabético, los archivos son los siguientes.

**Acl.php:**

Contiene la clase homónima encargada de la gestión de usuarios, roles y privilegios, su estructura es la siguiente:

Atributos:

$\_registry: privado, guarda la instancia de la clase Registry, la cual usa el patrón Singleton para no duplicar instancias de otras clases.

$\_db: privado, guarda una instancia Database, para la conexión a la base de datos.

$\_idusuario: privado, guarda el id del usuario que inició sesión, si es que existe, sino, guarda 0.

$\_idrole: privado, guarda el id del role del usuario, cada role posee una serie de privilegios predefinidos.

$\_permisos: guarda un array con los permisos del usuario específico, más allá del role que tenga, ya que se pueden generar excepciones.

Métodos:

constructor(): inicializa todos los atributos de la clase y ejecuta el método compilarAcl().

compilarAcl(): actualiza el atributo $\_permisos el cual combina los privilegios del role que corresponde al usuario, con los propios del mismo.

getRole(): devuelve el id del role correspondiente al usuario que inició sesión, se ejecuta en el constructor para inicializar el atributo $\_idrole.

gerPermisosRoleId(): devuelve un array con los id de todos los permisos existentes.

getPermisosRole(): devuelve un array que contiene de cada permiso, un key(llave que lo identifica de tipo string), un permiso (nombre del permiso), un valor (si es 1, es porque está habilitado para el role del usuario que inició sesión, 0 por lo contrario), un heredado (boolean, por si el valor del permiso corresponde al role, o a un permiso excepción), y el idpermiso.

getPermisoKey(int $idpermiso): recibe como parámetro el id de un permiso y devuelve su “llave”, un string sin espacios ni caracteres especiales que identifica de manera única al permiso.

getPermisosUsuario(): toma el id del usuario que inició sesión y devuelve en un array todos los permisos excepción configurados.

getPermisos(): devuelve todos los permisos que se encuentran en el atributo $\_permisos.

permiso(string $key): recibe como parámetro un string correspondiente a una llave de permiso, y si esta existe y si el valor correspondiente a ese permiso, con respecto al usuario que inició sesión, es 1, devuelve “true”, sino, devuelve “false”. Se utiliza en las vistas o controladores actuar dependiendo lo que devuelve la función.

acceso(string $key): si existe la llave que se recibe como parámetro se actualiza el tiempo de sesión, y se continúa con la ejecución, sino, se redirecciona a una página de error.

**Autoload.php:**

Cuando se intenta instanciar una clase que no se encuentra declarada previamente, PHP realiza un último esfuerzo volviendo a buscar la clase. Para administrar eso, existe el archivo Autoload.php en el cual se declaran y se “registran” las funciones a ejecutar cuando suceden estas cosas. Las mismas son:

autoloadCore($class): la variable $class toma el valor de la clase que se busca y la implementación de la función corresponde a consultar si existe esa clase dentro de la carpeta “app”, si es así, se ejecuta.

autoloadLibs($class): la variable $class toma el valor de la clase que se busca y la implementación de la función corresponde a consultar si existe esa clase dentro de la carpeta “libs”, si es así, se ejecuta.

Luego se ejecuta por cada una la función spl\_autoload\_register(funcion) que las registra.

Como estas sí tienen que estar declaradas de antemano, se agregan al index.php para tenerlas siempre implementadas.

**Bootstrap.php:**

Recibe los datos del Request.php y los envía al controlador indicado. Posee una única función, estática que es la que se corre en el index.php, la cual se llama run y recibe un objeto tipo Request como parámetro.

La misma recibe del Request las variables que corresponden al módulo, controlador, método y argumentos, las asigna a variables, las busca en el las carpetas correspondientes, y si existen los ejecuta.

**Config.php:**

Establece las configuraciones básicas la cual consiste en darle valor a una serie de constantes que se utilizarán durante toda la programación de la aplicación, las mismas son.

BASE\_URL (La URL base de la aplicación).

DEFAULT\_CONTROLLER (Controlador que se llama por defecto, cuando no se ingresa ninguno a la URL, o el ingresado no existe). Valor: ‘index’.

DEFAULT\_LAYOUT (Vista por defecto, si en el controlador no se designa ninguna, se llama a esa, la cual se encuentra en la carpeta “layout” de las vistas). Valor: ‘default’.

APP\_NAME (Nombre de la aplicación).

APP\_SLOGAN (Slogan o descripción de la aplicación).

APP\_COMPANY (Autores de la aplicación).

SESSION\_TIME (Tiempo de sesión). Valor: 10 min.

HASH\_KEY (Un código Hash, para utilizar en contraseñas).

DB\_HOST (Nombre del host para la conexión a la Base de Datos).

DB\_USER (Nombre del usuario para la conexión a la Base de Datos).

DB\_PASS (Contraseña para la conexión a la Base de Datos).

DB\_NAME (Nombre de la base para la conexión a la Base de Datos).

DB\_CHAR (Codificación de la conexión a la Base de Datos).

**Controller.php:**

Contiene la clase abstracta de la cual heredan todos los controladores, en la misma existen atributos y funciones útiles al momento de programar un controlador propio.

Atributos:

$\_registry: privado. Es el encargado de hacer funcionar a la clase Registry, la que implementa el patrón Singleton para instanciar ciertas clases que no requieren ser objetos más de una vez.

$\_view: contiene la clase View, de la cual heredan todas las vistas, donde se podrán ejecutar métodos correspondientes a la misma.

$\_acl: contiene todos los datos relacionados a los usuarios y sus privilegios.

$\_request: es el que procesa el requerimiento del Index.

Métodos:

\_\_construct(): publico. Inicializa todos los atributos con sus valores correspondientes.

index(): publico y abstracto. Es un método vacío que obligan a los controladores a redefinir el mismo para ser tomado como método de entrada al controlador.

loadModel(String $modelo, $modulo = false): protegida . Carga el modelo a utilizar por el controlador, el primer parámetro, de tipo String es el nombre del modelo a cargar y el segundo es el módulo del cuál debe buscarlo, si este no existe, toma como valor predeterminado “falso” y lo busca en la carpeta de modelos estándar.

getLibrary(String $libreria): protegida. Carga una librería que se encuentra en la carpeta “libs” del Framework, si bien, disponemos de función de autocarga que buscan librerías en esa misma carpeta, estas son pura y exclusivamente para clases, pueden existir librerías que no sean clases y requieran importarse, todas tienen que estar sí o sí programadas en PHP.

getTexto(String $clave): protegida. Filtra variables cadena recibidas por POST y devuelve la cadena resultante, recibe como parámetro el nombre del input del cual debe tomar y filtrar la variable.

getInt(String $clave): protegida. Filtra variables entera recibidas por POST y devuelve el entero resultante, recibe como parámetro el nombre del input del cual debe tomar y filtrar la variable.

redireccionar(String $ruta = false): protegida. Redirecciona a la ruta especificada, si esta no existe va al controlador Index.

filtrarInt(Int $int): protegida. Filtra los enteros que van por GET y los devuelve.

getPostParam(String $clave): protegida. Recibe como parámetro el nombre del input del cual debe tomar una variable, si este existe, lo devuelve.

getTexto(String $clave): protegida. Filtra variables cadena recibidas por POST y devuelve la cadena resultante, a diferencia del getTexto, este se utiliza para limpiar los tags html y las inyecciones SQL, recibe como parámetro el nombre del input del cual debe tomar y filtrar la variable.  
  
getAlphaNum(String $clave): protegida. Filtra variables cadena recibidas por POST y devuelve la cadena resultante, a diferencia del getTexto, solo acepta valores entre a y z, 0 y 9 y el \_, recibe como parámetro el nombre del input del cual debe tomar y filtrar la variable.

validarEmail($email): pública. Función que valida formato de emails.

**Database.php:**

Contiene la clase encargada de hacer la conexión a la base de datos, la misma extiende de PDO, una clase nativa de PHP que realiza múltiples operaciones relacionadas con la gestión de BBDD.

Métodos:

\_\_contruct(String $host, $dbname, $usr, $pass, $char): pública. Recibe los parámetros necesarios para la conexión a la base de datos, la cual la realiza al momento de la inicialización, los mismos son $host, el host de la BD, $dbname, nombre de la BD a conectar, $usr, usuario, $pass, contraseña, $char, el charset, o cotejamiento.

**Hash.php:**

Contiene la clase encargada de generar una clave Hash, para mayor seguridad de contraseñas.

Métodos:

getHash($algoritmo, $data, $key): pública y estática. devuelve el código hash final necesario para utilizar en las contraseñas.

**Model.php:**

Clase de la cual heredan todos los modelos.

Atributos:

$\_registry: posee la instancia de la clase Registry, la cual implementa el patrón Singleton para no reinstanciar clases que no deben hacerlo.

$\_db: posee una instancia de la clase Database, para realizar las consultas a la BD.

Métodos:

\_\_construct(): público. Inicializa los atributos de la clase.

**Registry.php:**

Contiene la clase encargada de utilizar el patrón Singleton para no instanciar más de una vez distintas clases que no deben hacerlo.

Atributos:

$\_instancia: privada y estática. Contiene la instancia de sí misma, es la única que se va a utilizar.

$\_data: privada. Serán las otras clases que se instanciarán de manera única.

Métodos:

\_\_construct(): privado. Constructor vacío.

getInstancia(): pública y estática. Verifica si la instancia de Registry ya se generó, si no es así, la crea y la devuelve, sino, devuelve la ya generada anteriormente.

\_\_set($name, $value): pública. Redefinición del método por defecto, agrega al array $\_data(el cual contiene las clases que deben ser instanciadas solo una vez), como índice el valor de $name (identificador para recibir la clase posteriormente), y el contenido el de $value (clase de la cual se habla).

\_\_get($name): pública. Redefinición del método por defecto, obtiene del array $\_data(el cual contiene las clases que deben ser instanciadas solo una vez), la instancia de la clase que se desea utilizar, se envía como parámetro el identificador de la misma, que corresponde al índice dentro del array.

**Request.php:**

Contiene la clase encargada de procesar la URL, y llamar al módulo (si existe), al controlador, al método y a los argumentos correspondientes. Cada vez que se ingresa una URL distinta, se genera una nueva instancia del Request, por eso en su constructor se encuentra el procesamiento de la misma.

Atributos:

$\_modulo: privado. Contendrá el valor del módulo a cargar.

$\_modules: privado. Contendrá todos los módulos instalados.

$\_controlador: privado. Contendrá el valor del controlador a cargar.

$\_metodo: privado. Contendrá el valor del método a cargar.

$\_argumentos: privado. Contendrá los valores de los argumentos a cargar.

Métodos:

\_\_construct(): público. Encargado de procesar la URL, comienza filtrando y separando los elementos de la misma, genera el array con los módulos instalados, y verifica que el primer valor enviado por la URL sea uno de ellos, si no es así, verifica que sea un controlador. Si no existe, se asigna el controlador por defecto, que es el index, el método y los argumentos, si estos existen.

getModulo(): público. Devuelve el módulo obtenido de la URL.

getControlador(): público. Devuelve el controlador obtenido de la URL.

getMetodo(): público. Devuelve el método obtenido de la URL.

getArgumentos(): público. Devuelve los argumentos obtenidos de la URL.

**Session.php:**

Contiene la clase encargada de gestionar la sesión de navegación.

Métodos:

init(): público y estático. Inicia la sesión.

destroy($clave = false): público y estático. Si recibe una clave como parámetro la anula dentro de la sesión, sino, destruye la sesión iniciada.

set($clave, $valor): público y estático. Si existe la clave en la sesión, actualiza con el valor enviado por parámetro.

get($clave): público y estático. Obtiene y devuelve el valor de la clave enviada por parámetro.

acceso($level): público y estático. Verifica que el acceso al contenido esté disponible para el nivel que se le envía por parámetro, de no ser así, se redirecciona a una página de error.

accesoView($level): público y estático. Verifica que el acceso a la vista esté disponible para el nivel que se le envía por parámetro, de no ser así, se redirecciona a una página de error.

getLevel($level): público y estático. Devuelve la llave del role dependiendo el id que se le envíe.

accesoEstricto(array $level, $noAdmin = false): público y estático. Es un control más estricto del acceso a las secciones, dependiendo un grupo de roles, incluso el de administrador, el cual puede no tener acceso a cierta información.

accesoEstrictoView(array $level, $noAdmin = false): público y estático. Es un control más estricto del acceso a las vistas, dependiendo un grupo de roles, incluso el de administrador, el cual puede no tener acceso a cierta información.

tiempo(): público y estático. Verifica el tiempo de sesión, si el mismo caduca, cierra la sesión.

**Views.php:**

Contiene la clase abstracta de la cual heredan todas las vistas, relacionada también con el motor de plantillas Smarty.

Atributos:

$\_request: privado. Contiene la clase que realiza la petición a partir de la URL recibida.

$\_js: privado. Contiene un array en el que se almacenan todos los archivos .js que requieren los controladores y se encuentran dentro de la carpeta de la vista.

$\_acl: privado. Contiene las configuraciones de usuarios a través del gestor ACL para saber si mostrar o no cierto contenido.

$\_rutas: privado. Array que contiene rutas específicas donde se almacenan los archivos necesarios para hacer funcionar los módulos, los controladores y las vistas.

$\_jsPlugin: privado. Array que almacena todos los archivos .js que requieren los controladores y se encuentran en la carpeta public/js.

$\_template: privado. Almacena el dato del template a utilizar, los mismos se encuentran en la carpeta views/layout.

$\_item: privado. Contiene el dato del ítem en el que se uno encuentra, este corresponde al método del controlador que se está llamando a través de la URL, con un identificador especial del mismo.

$\_widget: contiene los datos necesarios de los widgets y los métodos del mismo.

Métodos:

\_\_construct(Request $peticion, Acl $\_acl): público. Inicializa las variables y establece las rutas dónde se encuentran los archivos correspondientes a la vista y los archivos .js.

getViewId(): público y estático. Devuelve el id del ítem en el que se encuentra el usuario.

render($vista, $item = false, $noLayout =false): público. “Dibuja” la vista del controlador utilizado, recibe el id del ítem, la vista que se desea dibujar, establecen los parámetros y las rutas necesarias de archivos, se asigna las variable que necesita el controlador, tales como título, descripción, etc.

setJs(array $js): público. Se establece en un array, todos los archivos .js que se necesitan para utilizar de manera correcta el controlador.

setPluginJs(array $js): público. Se establece en un array, todos los archivos .js que se necesitan para utilizar de manera correcta el controlador, pero que se encuentran en la carpeta “public”.

setTemplate($template): público. Se establece el template a utilizar, de acuerdo al que recibe como parámetro.

widget($widget, $method, $options = array()): público. Devuelve toda la información del widget necesaria, recibiendo como parámetros el nombre del mismo, el método a ejecutar y un array con los argumentos necesarios.

widgetMethod($widget, $method, $options = array()): público. Devuelve la información del método del widget enviados anteriormente.

getLayoutPosition(): público. Devuelve la info relacionada a la configuración de posiciones donde se mostrarán los distintos módulos o Widgets dependiendo cada template utilizado.

getWidgets(): público. Devuelve todos los widgets habilitados para esa sección y ese template, del conjunto de Widgets existentes.

getWidgetContent(array $content): público. Devuelve los valores necesarios para hacer andar un Widget en un determinado lugar de la plantilla.

setWidgetOptions($key, $option): público. Devuelve las opciones propias del Plugin.

**Widget.php:**

Contiene la clase abstracta de la que dependen todos los Widgets, trabaja de forma muy parecida a los controladores y las vistas, pero solo para módulos más chicos.

Métodos:

loadModel(String $model): protegido. Busca y devuelve el modelo enviado por parámetro en la carpeta de los widgets.

render(String $view, $data = array(), $ext = ‘.php’): protegido. Dibuja la vista del widget que se recibe como primer parámetro, con algunos datos que se mandan en un array y si es necesario, con la extención del archivo de la vista.

**Consideraciones al momento de implementar un controlador:**

La implementación de un controlador, hay que tener en consideración las siguientes cuestiones.

Primero, el mismo debe alojarse en la carpeta “controllers”, y su notación debe ser de la forma “nombreController.php”.

La clase debe mantener la misma forma de escritura y depender de la clase abstracta “Controler”. Como constructor debe llamar al constructor del padre y sí o sí se debe implementar el método index(). Si el controlador requiere ejecutar las funciones de un modelo, el controlador necesita poseer una variable privada que lo guarde, y que debe ser inicializada en el constructor. A continuación se ejemplificará con un Controlador que maneje los post de una aplicación a través de un modelo llamado “postModel”.

class postController extends Controller {

private $\_postModel;

public function \_\_construct()

{

parent::\_\_construct();

$this->\_postModel = $this->loadModel('postModel');

}

public function index(){

}

}

Al ser una subclase de Controller, cada controlador, hereda el atributo $\_view, el cual ejecuta todas las funciones correspondiente a la clase View, el cual también hereda funciones del motor de plantillas Smarty.

Para comenzar a entender el uso de esta variable, hay dos funciones principales que siempre (o casi siempre) se utilizan en cada método del controlador, ellas son, assign(variable, valor), la cual asigna a la variable enviada como primer parámetro (de tipo String), el valor enviado como segundo (puede ser tanto un valor específico como una función), para luego ser referenciada en la plantilla, y por otro lado, render(‘vista’, ‘controlador’), encargado de “dibujar” el contenido del método con la vista y template correspondiente, no es necesario poner el nombre del controlador si hace referencia al mismo.

class postController extends Controller {

private $\_postModel;

public function \_\_construct()

{

parent::\_\_construct();

$this->\_postModel = $this->loadModel('postModel');

}

public function index(){

$this->\_view->assign(‘titulo’,’Portada de página’);

$this->\_view->assign(‘post’, $this->\_postModel->getPost());

$this->\_view->render(‘index’);

}

}

Si se requiere cambiar de template se utiliza la función setTemplate($template), enviando como parámetro el que se encuentra en la carpeta “layout” dentro de “views”.

class postController extends Controller {

private $\_postModel;

public function \_\_construct()

{

parent::\_\_construct();

$this->\_postModel = $this->loadModel('postModel');

}

public function index(){

$this->\_view->assign(‘titulo’,’Portada de página’);

$this->\_view->assign(‘post’, $this->\_postModel->getPost());

$this->\_view->setTemplate(‘postTemplate’);

$this->\_view->render(‘index’);

}

}

Si se requieren el uso de archivos js, propio del template a utilizar, se debe utilizar la función setJs(array $js), donde se especifican todos los archivos que se desean usar.

class postController extends Controller {

private $\_postModel;

public function \_\_construct()

{

parent::\_\_construct();

$this->\_postModel = $this->loadModel('postModel');

}

public function index(){

$this->\_view->assign(‘titulo’,’Portada de página’);

$this->\_view->assign(‘post’, $this->\_postModel->getPost());

$this->\_view->setTemplate(‘postTemplate’);

$this->\_view->setJs(array(‘post’));

$this->\_view->render(‘index’);

}

}

Así mismo, si se requieren archivos js, ubicados en la carpeta “public” la cual comparte archivos para todos los módulos y controladores, se utiliza la función setJsPlugin(array $js).

class postController extends Controller {

private $\_postModel;

public function \_\_construct()

{

parent::\_\_construct();

$this->\_postModel = $this->loadModel('postModel');

}

public function index(){

$this->\_view->assign(‘titulo’,’Portada de página’);

$this->\_view->assign(‘post’, $this->\_postModel->getPost());

$this->\_view->setTemplate(‘postTemplate’);

$this->\_view->setJs(array(‘post’));

$this->\_view->setJsPlugin(array(‘validate.js’));

$this->\_view->render(‘index’);

}

}

Si bien el Framework cuenta con funciones de autocarga que llaman a las clases que se requieren instanciar. En la carpeta “libs” existen librerías que nos son clases, por lo cual deben ser llamadas por la función getLibrary($libreria).

class postController extends Controller {

private $\_postModel;

public function \_\_construct()

{

parent::\_\_construct();

$this->\_postModel = $this->loadModel('postModel');

}

public function index(){

$this->\_view->assign(‘titulo’,’Portada de página’);

$this->\_view->assign(‘post’, $this->\_postModel->getPost());

$this->\_view->setTemplate(‘postTemplate’);

$this->\_view->setJs(array(‘post’));

$this->\_view->setJsPlugin(array(‘validate.js’));

$this->lib = $this->getLibrary(‘librería’);

$this->\_view->render(‘index’);

}

}

Manejo de formularios en un controlador:

Muchas veces el controlador requiere manejar datos ingresados por un formulario, para actualizar la Base de datos, para segmentar los resultados de una consulta, etc. Siguiendo el ejemplo anterior, se mostrará un método encargado de dar de alta los post, con título, descripción e imagen, el mismo se llamará nuevo().

class postController extends Controller {

private $\_postModel;

public function \_\_construct()

{

parent::\_\_construct();

$this->\_postModel = $this->loadModel('postModel');

}

public function index(){

$this->\_view->assign(‘titulo’,’Portada de página’);

$this->\_view->assign(‘post’, $this->\_postModel->getPost());

$this->\_view->setTemplate(‘postTemplate’);

$this->\_view->setJs(array(‘post’));

$this->\_view->setJsPlugin(array(‘validate.js’));

$this->lib = $this->getLibrary(‘librería’);

$this->\_view->render(‘index’);

}

public function nuevo(){

$this->\_view->assign('titulo', "Nuevo Post");

$this->\_view->setJsPlugin(array('jquery.validate'));

$this->\_view->setJs(array('nuevo'));

}

}

Como la idea es trabajar siempre en el mismo método, el formulario contendrá un input del tipo hidden con el valor 1 y nombre “guardar” una vez enviado los datos por el formulario, el método revisa si el input guardar posee ese valor y de ser así, comienza a trabajar. Luego se le asignan a la variable “datos” los que se encuentran en el $\_POST (que son los enviados por el formulario), y se procede a verificar si todos hayan sido ingresados, de no ser así, se asigna a una variable el mensaje de error y se vuelve a dibujar la vista del método.

class postController extends Controller {

private $\_postModel;

public function \_\_construct()

{

parent::\_\_construct();

$this->\_postModel = $this->loadModel('postModel');

}

public function index(){

$this->\_view->assign(‘titulo’,’Portada de página’);

$this->\_view->assign(‘post’, $this->\_postModel->getPost());

$this->\_view->setTemplate(‘postTemplate’);

$this->\_view->setJs(array(‘post’));

$this->\_view->setJsPlugin(array(‘validate.js’));

$this->lib = $this->getLibrary(‘librería’);

$this->\_view->render(‘index’);

}

public function nuevo(){

$this->\_view->assign('titulo', "Nuevo Post");

$this->\_view->setJsPlugin(array('jquery.validate'));

$this->\_view->setJs(array('nuevo'));

if ($this->getInt('guardar') == 1) {

$this->\_view->assign('datos', $\_POST);

if(!$this->getTexto('titulo')){

$this->\_view->assign('\_error', 'Debe introducir el título del post');

$this->\_view->render('nuevo', 'post');

exit;

}

if(!$this->getTexto('cuerpo')){

$this->\_view->assign('\_error', 'Debe introducir el cuerpo del post');

$this->\_view->render('nuevo', 'post');

exit;

}

} //If

} //nuevo

} // Fin de clase.

Dentro del mismo condicional que verifica si se enviaron los datos por el formulario se procesa la imagen subida y se asigna la ruta a una variable para que, luego con el modelo, se inserte en la base de datos el post correspondiente.

class postController extends Controller {

private $\_postModel;

public function \_\_construct()

{

parent::\_\_construct();

$this->\_postModel = $this->loadModel('postModel');

}

public function index(){

$this->\_view->assign(‘titulo’,’Portada de página’);

$this->\_view->assign(‘post’, $this->\_postModel->getPost());

$this->\_view->setTemplate(‘postTemplate’);

$this->\_view->setJs(array(‘post’));

$this->\_view->setJsPlugin(array(‘validate.js’));

$this->lib = $this->getLibrary(‘librería’);

$this->\_view->render(‘index’);

}

public function nuevo(){

$this->\_view->assign('titulo', "Nuevo Post");

$this->\_view->setJsPlugin(array('jquery.validate'));

$this->\_view->setJs(array('nuevo'));

if ($this->getInt('guardar') == 1) {

$this->\_view->assign('datos', $\_POST);

if(!$this->getTexto('titulo')){

$this->\_view->assign('\_error', 'Debe introducir el título del post');

$this->\_view->render('nuevo', 'post');

exit;

}

if(!$this->getTexto('cuerpo')){

$this->\_view->assign('\_error', 'Debe introducir el cuerpo del post');

$this->\_view->render('nuevo', 'post');

exit;

}

$imagen = '';

if(isset($\_FILES['imagen']['name'])){

//Código que inserta la imágen

}

$this->\_postModel->insertarPost(

$this->getPostParam('titulo'),

$this->getPostParam('cuerpo'),

$imagen

);

$this->redireccionar('post');

} //If

$this->\_view->render('nuevo', 'post');

} //nuevo

} // Fin de clase.

**Consideraciones al momento de implementar un modelo:**

En la implementación de un modelo, hay que tener en consideración las siguientes cuestiones.

Primero, el mismo debe alojarse en la carpeta “models”, y su notación debe ser de la forma “nombreModel.php”.

La clase debe mantener la misma forma de escritura y depender de la clase abstracta “Model”. Como constructor debe llamar al constructor del padre. A continuación se ejemplificará con el Modelo que maneje los post de una aplicación siguiendo el ejemplo del controlador.

class postModel extends Model{

public function \_\_construct(){

parent::\_\_construct();

}

}

Para mostrar los posts en el index del controlador se llama a la función getPosts() del modelo, el cual los busca en la base de datos y lo devuelve en forma de Array asociativo. Al extender de la clase Model, hereda un atributo denominado $\_db el cual posee la clase Database, que desciende de PDO, por lo tanto hereda todos sus métodos. En este caso se comienza usando query(), que realiza la consulta a la base de datos y la devuelve como un Objeto PDOStatment, el cual utiliza el método fetchall() que devuelve los datos de la consulta en forma de Array asociativo.

class postModel extends Model{

public function \_\_construct(){

parent::\_\_construct();

}

public function getPosts(){

$post = $this->\_db->query("SELECT \* FROM posts");

return $post->fetchall();

}

}

Si se desea insertar un registro en la BD, como en el método nuevo() se utiliza el método insertarPost(), el cual, utiliza el mismo atributo con el método prepare() y execute().

class postModel extends Model{

public function \_\_construct(){

parent::\_\_construct();

}

public function getPosts(){

$post = $this->\_db->query("SELECT \* FROM posts");

return $post->fetchall();

}

public function insertarPost($titulo, $cuerpo, $imagen){

$this->\_db->prepare("INSERT INTO posts (titulo,cuerpo,imagen) values (:titulo, :cuerpo, :imagen)")

->execute(array(

':titulo' => $titulo,

':cuerpo' => $cuerpo,

':imagen' => $imagen

));

}

}

**Consideraciones al momento de implementar una vista:**

En la implementación de una vista, hay que tener en consideración las siguientes cuestiones.

Primero, el mismo debe alojarse en una carpeta que se encuentra en “views” y que se llame cómo el controlador, y su notación debe ser de la forma “nombre.tpl”.

La misma debe utilizar el código propio del motor de plantillas Smarty, mezclado con código html.

Como cada vista solo contiene el “contenido” que devuelve el módulo, y éste forma parte de plantillas que ya existen y que lo llaman de manera externa, solo se debe programar el código propio de ese contenido.

Algunas cuestiones básicas de la programación con Smarty:

* Las variables asignadas en el controlador, se muestran en la vista entre llaves {} y con el signo $ como antecesor, si se trata de un array se hace referencia al índice después de un punto (.).
* Para recorrer un array se utiliza la estructura Foreach con un ítem que representará cada registro del array y un “from” en el que se especifica cuál es el array a recorrer.
* La estructura if se hace también entre llaves, sin utilizar paréntesis y las llamadas a funciones se realiza mediante la flecha ->.

Vista del index.tpl del controlador:

<h2>Bienvenido Piyo - Posts</h2>

{if isset($posts) && count($posts)}

<table>

{foreach item=datos from=$posts}

<tr>

<td>{$datos.id}</td>

<td>{$datos.titulo}</td>

<td>{$datos.cuerpo}</td>

<td>

{if isset($datos.imagen)}

<a href="{$\_layoutParams.root}public/img/post/{$datos.imagen}" target="\_blank">

<img src="{$\_layoutParams.root}public/img/post/thumb/thumb\_{$datos.imagen}">

</a>

{/if}

</td>

<td><a href="{$\_layoutParams.root}post/editar/{$datos.idpost}">Editar Post</a></td>

<td><a href="{$\_layoutParams.root}post/eliminar/{$datos.idpost}">Eliminar Post</a></td>

</tr>

{/foreach}

</table>

{else}

<p>No hay posts</p>

{/if}

<a href="{$\_layoutParams.root}post/nuevo">Agregar Post</a> <!-- BASE\_URL para usar el mismo controlador pero con otra función -->

<br>

**Paginación:**

Muchas veces se requiere paginar los resultados de la consulta a una base de datos, por ejemplo mostrando los post que hay en la base de datos, para eso el Framework dispone de una clase “Paginador” encargada de este tipo de acciones. Para ello, al momento de asignarle a la variable los registros que queremos paginar, instanciamos la clase y utilizamos su función “paginar” enviando como parámetro la función que devuelve los registros paginados y luego, asignarle a la otra, la vista que tendrá la paginación, la cual se encontrará en la carpeta “views”. En este caso agregaremos una vista de “paginación\_ajax” la cual terminaremos de detallar más adelante.

Comenzamos cambiando a la variable post el resultado de la función “paginar” y la vista de la paginación.

public function index(){

$this->\_view->assign(‘titulo’,’Portada de página’);

$this->\_view->assign('posts', $paginador->paginar($this->\_post->getPrueba()));

$this->\_view->assign('paginacion', $paginador->getView('paginacion\_ajax'));

$this->\_view->setTemplate(‘postTemplate’);

$this->\_view->setJs(array(‘prueba’));

$this->\_view->render(‘index’);

}

Creamos la vista de la paginación, la cual se encontrará en la carpeta “\_paginador” debajo de views y contendrá el siguiente código.

<?php if (isset($this->\_paginacion)): ?>

<?php if ($this->\_paginacion['primero']): ?>

<a class="pagina" pagina="<?php echo $this->\_paginacion['primero']; ?>" href="javascript:void(0)">Primero</a>

<?php else: ?>

<span class="disabled">Primero</span>

<?php endif; ?>

&nbsp; - &nbsp;

<?php if ($this->\_paginacion['anterior']): ?>

<a class="pagina" pagina="<?php echo $this->\_paginacion['anterior']; ?>" href="javascript:void(0)">Anterior</a>

<?php else: ?>

<span class="disabled">Anterior</span>

<?php endif; ?>

&nbsp; - &nbsp;

<?php for($i = 0; $i < count($this->\_paginacion['rango']); $i++): ?>

<?php if ($this->\_paginacion['actual'] == $this->\_paginacion['rango'][$i]): ?>

<?php echo $this->\_paginacion['rango'][$i]; ?>

<?php else: ?>

<a class="pagina" pagina="<?php echo $this->\_paginacion['rango'][$i]; ?>" href="javascript:void(0)"><?php echo $this->\_paginacion['rango'][$i]; ?></a>

<?php endif; ?>

<?php endfor; ?>

&nbsp; - &nbsp;

<?php if ($this->\_paginacion['siguiente']): ?>

<a class="pagina" pagina="<?php echo $this->\_paginacion['siguiente']; ?>" href="javascript:void(0)">siguiente</a>

<?php else: ?>

Siguiente

<?php endif; ?>

&nbsp; - &nbsp;

<?php if ($this->\_paginacion['ultimo']): ?>

<a class="pagina" pagina="<?php echo $this->\_paginacion['ultimo']; ?>" href="javascript:void(0)">ultimo</a>

<?php else: ?>

<span class="disabled">Último</span>

<?php endif; ?>

<div style="text-align: center;">

<p><small>Página: <?php echo $this->\_paginacion['actual']; ?> de <?php echo $this->\_paginacion['total']; ?></small><br>

Registros por páginas

<select name="registros" id="registros">

<?php for ($i=10; $i <= 100; $i+=10) { ?>

<option value="<?php echo $i; ?>" <?php if ($i == $this->\_paginacion['limite']) {echo "selected=selected";}?>><?php echo $i; ?></option>

<?php

} ?>

</select>

<?php endif; ?>

El mismo funciona de la siguiente manera, si existe la paginación verifica que no se encuentre en la primera posición, de ser así escribe “Primero” con link a la primera posición, sino, lo escribe sin el link. Luego hace la verificación con “Anterior”, y luego realiza un recuento de la cantidad de páginas en la cual fue dividida la paginación, escribiendo una por una asignándole un link a cada una de ellas, salvo a la que corresponda a la posición en la que nos encontramos actualmente, por ejemplo, si nos encontramos en la página 3 de 10, habrá 10 números y el 3 será el único que no tendrá link. Después la verificación que se realiza con Anterior y Primero, se hace con Siguiente y Último y finalmente hay un Select en el cual se decide la cantidad de registros a mostrar por página.

Por último agregamos a la vista la sentencia que muestra la paginación

<h2>Bienvenido Piyo - Posts</h2>

{if isset($posts) && count($posts)}

<table>

{foreach item=datos from=$posts}

<tr>

<td>{$datos.id}</td>

<td>{$datos.titulo}</td>

<td>{$datos.cuerpo}</td>

<td>

{if isset($datos.imagen)}

<a href="{$\_layoutParams.root}public/img/post/{$datos.imagen}" target="\_blank">

<img src="{$\_layoutParams.root}public/img/post/thumb/thumb\_{$datos.imagen}">

</a>

{/if}

</td>

<td><a href="{$\_layoutParams.root}post/editar/{$datos.idpost}">Editar Post</a></td>

<td><a href="{$\_layoutParams.root}post/eliminar/{$datos.idpost}">Eliminar Post</a></td>

</tr>

{/foreach}

</table>

{else}

<p>No hay posts</p>

{/if}

{$paginacion|default:""}

<a href="{$\_layoutParams.root}post/nuevo">Agregar Post</a> <!-- BASE\_URL para usar el mismo controlador pero con otra función -->

<br>

**Uso de AJAX:**

AJAX nos permite, entre otras cosas, hacer llamados al servidor en tiempo real, refrescando solamente una parte de la pantalla, sin tener la necesidad de actualizar toda, en este caso veremos cómo filtrar los resultados de búsqueda de los post y a la paginación de los registros.

Comenzamos creando en el método dentro del controlador encargado de recibir los datos que se van a enviar por el formulario y de hacer la consulta en tiempo real y devolver los datos necesarios para refrescar solo una parte de la pantalla.

(NOTA: para hacer más claro el ejemplo se le asignan a los post un país y una ciudad a la cual pertenecen).

public function pruebaAjax(){

$pagina = $this->getInt('pagina');

$nombre = $this->getSql('nombre');

$pais = $this->getInt('pais');

$ciudad = $this->getInt('ciudad');

$registros = $this->getInt('registros');

$condicion = '';

if ($nombre) {

$condicion = " AND r.titulo like '%$nombre%'";

}

if ($pais) {

$condicion .= " AND r.idpais = $pais";

}

if ($ciudad) {

$condicion .= " AND r.idciudad = $ciudad";

}

$paginador = new Paginador();

$this->\_view->setJs(array('prueba'));

$this->\_view->assign('posts', $paginador->paginar($this->\_postModel->getPrueba($condicion), $pagina, $registros));

$this->\_view->assign('paginacion', $paginador->getView('paginacion\_ajax'));

$this->\_view->render('ajax/prueba', false, true);

}

En este caso el método recibe todos los datos enviados por el formulario, y si existen genera una consulta a la base de datos para mostrar los resultados acotados.

$pagina, es el valor que se recibe desde la paginación.

$nombre, valor que se recibe desde un input tipo texto, como cuadro de búsqueda.

$pais, valor que se recibe desde un select, con id de países.

$ciudad, valor que se recibe desde un select, con id de ciudades.

$registros, el valor del select en el cual se deciden la cantidad de registros que se muestran por página.

Creamos el método en el Modelo el cual devolverá los registros dependiendo de la búsqueda.

public function getPrueba($condicion = ''){

$post = $this->\_db->query("SELECT r.\*, p.nombre AS pais, c.nombre AS ciudad FROM posts r, paises p, ciudades c WHERE r.idpais = p.idpais AND r.idciudad = c.idciudad $condicion");

return $post->fetchAll();

}

Creamos la vista de la devolución de datos, como verán en el método existe la siguiente línea.

$this->\_view->render('ajax/prueba', false, true);

Esto significa que, la vista se va a encontrar en la carpeta “ajax” dentro de la carpeta de la vista de los posts, el segundo parámetro, se refiere a que no corresponde a ningún ítem extra, y el tercero a que no se dibujará ningún template. Esto se hace de esta manera ya que este contenido se dibujará solo en una parte de la pantalla, la cual ya tiene un template y un ítem definido.

La vista de la devolución del método sería la siguiente.

{if isset($posts) && count($posts)}

<table class="table table-bordered table-condensed table-striped">

<tr>

<th>ID</th>

<th>Nombre</th>

<th>Pais</th>

<th>Ciudad</th>

</tr>

{foreach item=datos from=$posts}

<tr>

<td>{$datos.idpost}</td>

<td>{$datos.titulo}</td>

<td>{$datos.pais}</td>

<td>{$datos.ciudad}</td>

</tr>

{/foreach}

</table>

{else}

<p><strong>No hay posts!</strong></p>

{/if}

{$paginacion|default:""}

Como verán es solo una parte de la vista del index, la cual se actualiza después de hacer la consulta.

Por último creamos el archivo .js que utilizará AJAX por medio de la programación jQuery. Podemos ver que en el método index, está la siguiente línea.

$this->\_view->setJs(array(‘prueba’));

Aquí se llama al archivo prueba.js, ubicado en la carpeta js, dentro de la de la vista. El cual contiene el siguiente código.

$(document).on('ready', function(){

$('#lista\_registros').delegate( '.pagina', 'click', function(){

paginacion($(this).attr("pagina"));

});

var paginacion = function(pagina){

var pagina = 'pagina=' + pagina;

var nombre = '&nombre=' + $("#nombre").val();

var pais = '&pais=' + $("#pais").val();

var ciudad = '&ciudad=' + $("#ciudad").val();

var registros = '&registros=' + $("#registros").val();

$.post(\_root\_ + 'post/pruebaAjax', pagina + nombre + pais + ciudad + registros, function(data){

$("#lista\_registros").html('');

$("#lista\_registros").html(data);

});

}

$("#pais").change(function(){

var pagina = 'pagina=' + pagina;

if($("#pais").val() == "x"){

$("#ciudad").html('');

$("#ciudad").append('<option value=""> - seleccione ciudad - </option>');

};

$.post(\_root\_ + 'ajax/getCiudades', 'pais=' + $("#pais").val(), function(datos){

$("#ciudad").html('');

$("#ciudad").append('<option value=""> - seleccione ciudad - </option>');

for(var i = 0; i < datos.length; i++){

$("#ciudad").append('<option value="' + datos[i].idciudad + '">' + datos[i].nombre + '</option>')

}

}, 'json');

paginacion();

});

$("#btnEnviar").click(function(){

paginacion();

});

$("#ciudad").change(function(){

if($("#pais").val()){

paginacion();

};

});

$('#lista\_registros').delegate( '#registros', 'change', function(){

paginacion();

});

});

Para entender cómo funciona un poco el mismo, cuando el documento está listo, comienza a trabajar una función la cual delega el contenido con id “lista\_registros”, al iniciarse evento click de todos elementos de clase “pagina” (los cuales son los links que se encuentran en la vista de paginación\_ajax), lo que devuelve la función “paginacion” dentro del mismo js, enviando como parámetro el valor de uno de estos links.

La función paginación, toma el valor de todos los elementos del formulario, los envía por POST al método pruebaAjax, el cual hace la consulta y muestra sus resultados.

Por otro lado si el select con id país cambia, toma su valor y el de la página en la que se encuentra, y actualiza el select ciudades con las que corresponden al país, luego llama a paginación.

Más adelante, el botón con id “btn\_enviar”, llama a paginación el cual toma el valor del cuadro de búsqueda, sigue con la actualización a partir de la selección de una ciudad (select con id “ciudad”) y termina con la actualización cuando se elige una cantidad distinta de registros a mostrar por página (select con id “registros”).

**Consideraciones al momento de crear un Widget:**

La implementación de un widget, hay que tener en consideración las siguientes cuestiones.

Primero, el mismo debe alojarse en la carpeta con el nombre del widget, dentro de la carpeta “widgets”, y su notación debe ser de la forma “nombre.php”.

El nombre de la clase debe responder a la forma “nombreWidget” y depender de la clase abstracta “Widget”. Si el widget requiere ejecutar las funciones de un modelo (el cual debe encontrarse en la carpeta “models” dentro de la carpeta del widget, y debe extender de Models), necesita poseer una variable privada que lo guarde, y que debe ser inicializada en el constructor. A continuación se ejemplificará con un widget que maneje los menúes de una aplicación a través de un modelo llamado “menu”.

class menuWidget extends Widget{

private $\_modelo;

public function \_\_construct(){

$this->\_modelo = $this->loadModel('menu', 'menu');

}

public function getMenu($menu, $view, $inverse = null){

$data['menu'] = $this->\_modelo->getMenu($menu);

$data['inverse'] = $inverse;

return $this->render('menu', $view, $data);

}

public function getConfig($menu){

$menus['sidebar'] = array(

'position' => 'sidebar',

'show' => 'all',

'hide' => array('registro')

);

$menus['top'] = array(

'position' => 'top',

'show' => 'all',

);

return $menus[$menu];

}

}

En este código podemos ver que se inicializa la variable cargando el modelo “menú” y luego se implementa una función que devuelve un menú específico, ya que la aplicación puede tener distintos menúes dependiendo la finalidad de cada uno y su posición, esta función recibe el nombre del menú, la vista que va a tener (la cual está guardada en la carpeta “views” dentro de la carpeta del widget) y un atributo opcional que cambia los colores del menú, se crea un array con un primer índice que almacena el menú seleccionado (llamado desde el modelo), un segundo índice que guarda el valor de $inverse y luego se devuelve el “dibujo” del menú con la vista seleccionada.

Como segundo método se registran las configuraciones de los menúes, en las que según a cuál haga referencia, se especifica la posición, en qué ítems o páginas se muestran y en cuáles no.