Hola a Todos.

Personalmente me costo un poco entender el modelo MVC. Ya aprendí orientación a objetos y ahora esto :-P pero buee.. la verdad que facilita mucho el trabajo y nos ayuda bastante a organizarnos.

Cuando entendí el tema de model – view – controller tuve que aprender algo bastante importante (y lo sigo haciendo…). Como unimos estas tres partes?!!

Después de varias caídas y levantadas encontré con ayuda del grupo la forma de hacerlo. Una de las razones por las que me costaba era porque no encontraba ejemplos no tan abstractos especialmente con respecto al view por lo que me gustaría compartir con ustedes el sitio que estoy desarrollando para que juntos podamos seguir aprendiendo ya que me imagino que las críticas constructivas seguirán llegando :-P

Se trata de una empresa de lácteos que tendría varios mundos. El primero de ellos es el sitio institucional propiamente dicho. Se integrarán después las marcas que son Topicaza, Dueto, Disfruta y posiblemente más.

Por el momento tengo terminado el institucional que consta de 6 páginas básicamente. Principal, historia, compromiso social, productos, noticias y contacto.

Estoy usando WAMP5 Version 1.7.0, mysql y kumbia-0.4.5-stable.

Por el momento solo tengo tablas sin relaciones aún, ya que se están definiendo los demás mundos.

Las tablas están explicadas a continuación:

- **datos_estaticos:** yo suelo llamar datos estáticos a los datos que no permiten un ABM completo sobre ellos sino que solo permiten modificación. Siempre deberían existir. Básicamente la tabla tiene un solo registro a menos que sea un sitio multi-idiomas, en este caso tendrá un campo más de idiomas_id y filas como idiomas existan.
- **productos:** básicamente una entidad que contiene los productos de la empresa. Destaco el campo portal_id que después serviría para filtrar a que mundo pertenecen estos idiomas
- **noticias:** noticias del sitio para que los clientes se enteren de que hacemos. Noten el campo portal_id de nuevo. También notemos el campo "destacada" que indica si va a estar como principal en la página principal (valga la redundancia). Las demás irán al costado como links.
- **cgrefcodes:** Esta tabla es mi tabla de parámetros. En dominio ingreso una palabra clave que me permite filtrar y en descripción el valor o texto que necesitaría. Los campos máximo y mínimo tienen otros propósitos que por el momento no los usare. Por ejemplo:
  dominio: EMAIL_INFO
  descripción: info@empresa.com

  dominio: DIRECCION_EMPRESA
  descripción: Gral. Santos 1100

OK. Hasta aquí hicimos un poco de relevamiento, análisis y diseño (mi profesor de la universidad estaría orgulloso de escuchar que hablo así jeje :-P)

El script para la base de datos con los datos de ejemplo acompaña al tutorial.

Bueno vayamos por partes (como dice Jack el Destripador…)

CONFIGURANDO KUMBIA

El sitio está en esta dirección en mi localhost: www.coop.com.py/v1/

Donde v1 es la carpeta contenedora del Kumbia (v1 tomando en cuenta que podría haber otras versiones del sitio con un kumbia más actualizado)

Y, www.coop.com.py es un virtual host creado en el Apache de mi máquina. Estos host virtuales son muy útiles a la hora de programar ya que en este ejemplo vamos a utilizar urls absolutas con relación al HOME de la aplicación (es decir v1/ para este caso ya que el Kumbia está ahí). Esto para tratar de que en diseño se pueda ver de alguna manera la interfaz.
Si no hacemos esto es posible que tengamos problemas al subir la aplicación. Por lo que simulo el lugar exacto donde se utilizaría el sitio emulando la url www.coop.com.py.
Un buen tutorial para hacer esto con WAMP se encuentra en
http://krilbert.justmc.com/posts/leer/virtualhosts_con_wamp/483

Ahora vayamos a la configuración de la base de datos. Buscamos el archivo que se encuentra dentro de coop.com.py/v1/forms/config/config.ini (OJO. www.coop.com.py es mi virtual host que simula que estoy en Internet pero estoy en localhost y coop.com.py es una carpeta que está directamente dentro del HOME de mi Apache. Cuando accedo a www.coop.com.py en realizad accede a esa carpeta)

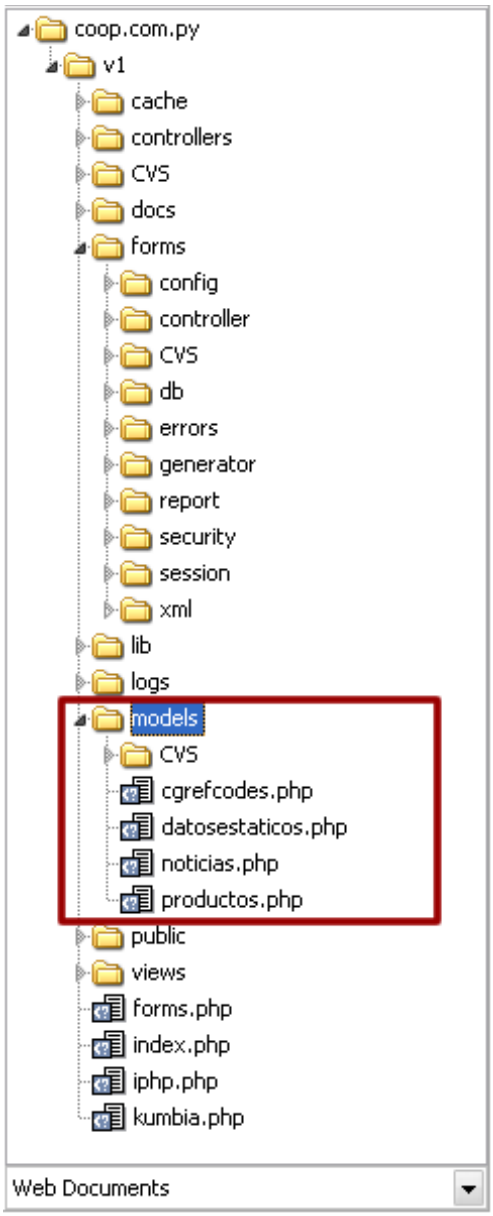
Yo lo tengo configurado así:

```
; Kumbia Web Framework Configuration

; Parámetros Generales del Proyecto
[project]
mode = development
name = "COOP"
interactive = On

; Parámetros de base de datos
; Utiliza el nombre del controlador nativo (mysql, postgresql, oracle)

[production]
database.host = mysql10.hsphere.cc
database.username = usuario
database.password = *******
database.name =  lacteos
database.type = mysql

[development]
database.host = localhost
database.username = root
database.password =
database.name = lacteos
database.type = mysql
```

MODEL

Esta parte del modelo MVC nos permite comunicar con las entidades de la Base de Datos y nos ahorra mucho trabajo que antes solíamos hacer para recuperar o interactuar con los registros.

Por cada tabla creamos un model. Entonces tendremos los models situados en la carpeta coop.com.py/v1/models/



Los models son muy sencillos de crear. Usamos el mismo nombre de la tabla que queremos representar y extendemos de la clase ActiveRecord que nos da la magia para no volvernos locos programando :-P

Uso el public $debug = true; cuando estoy haciendo pruebas usando ese modelo para que me despliegue en pantalla donde está el error ya que si no lo pongo en el momento de haber algún error en un Query por ejemplo no me muestra mucha ayuda. Lo comento cuando ya lo he probado todo.

- Noten que el archivo en sí se llama por ejemplo cgrefcodes.php y la clase Cgrefcodes

- También es muy interesante lo siguiente. Si recuerdan hay una tabla llamada datos_estaticos. A la hora de nombrar el archivo lo escribimos así datosestaticos.php y la clase DatosEstaticos y el por debajo entiende que en el momento de existir una E mayúscula (entiéndase una letra mayúscula) en la base de datos hay un guión bajo (Esto creo que es así ya que al final probando y probando me di cuenta de esto).

Abajo están las imágenes de los models. Son cuatro archivos separados. Para ver los archivos reales pueden ir al archivo adjunto al manual donde está toda la estructura del proyecto.

```php
<?php

class Cgrefcodes extends ActiveRecord
{
    //public $debug  = true;
}

?>
```

```php
<?php

class DatosEstaticos extends ActiveRecord
{
    //public $debug  = true;
}

?>
```

```php
<?php

class Noticias extends ActiveRecord
{
    //public $debug  = true;
}

?>
```

```php
<?php

class Productos extends ActiveRecord
{
    //public $debug  = true;
}

?>
```

OK. Con esto nos olvidamos en gran parte, por no decir completamente de los SELECT * FROM … Etc.

Yyyyyyyyyyyyyy. Ya está. Model terminado.

VIEW

Bueno pasemos al siguiente componente del modelo MVC. Les confieso que odie esta palabra por varios días. Pero por fin le encontré la vuelta.

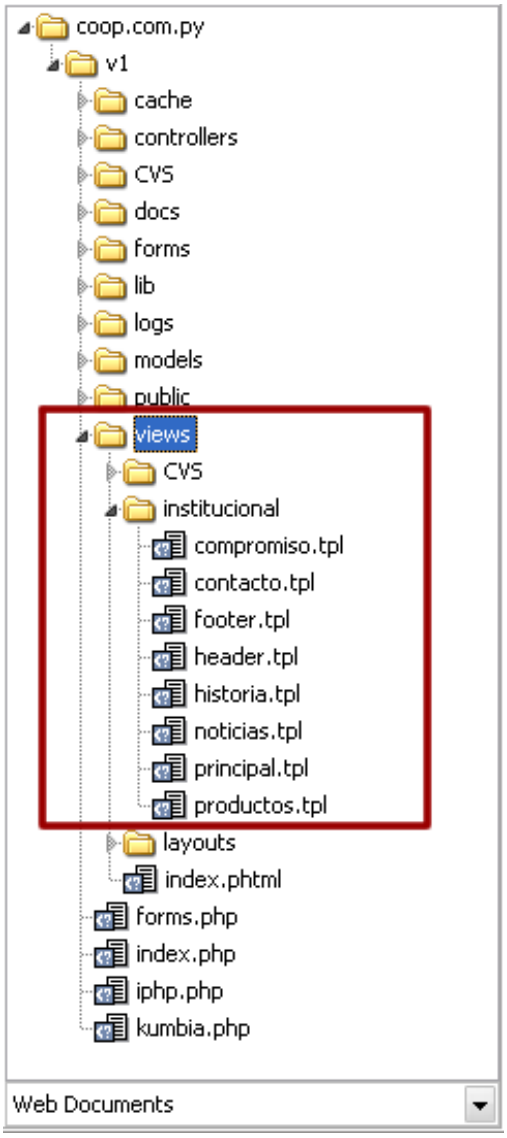
El VIEW nos permite abstraernos de la programación y de la lógica de la aplicación y pensar en como el usuario verá las páginas.

Para esto he usado la funcionalidad de SMARTY que es un gestor de plantillas o templates bastante útil. El siguiente mundo quizá lo haga con PHTMLs. Por el momento lo hice con SMARTY ya que soy un gran seguidor de él y me parece muy interesante su forma de usarlo :-P

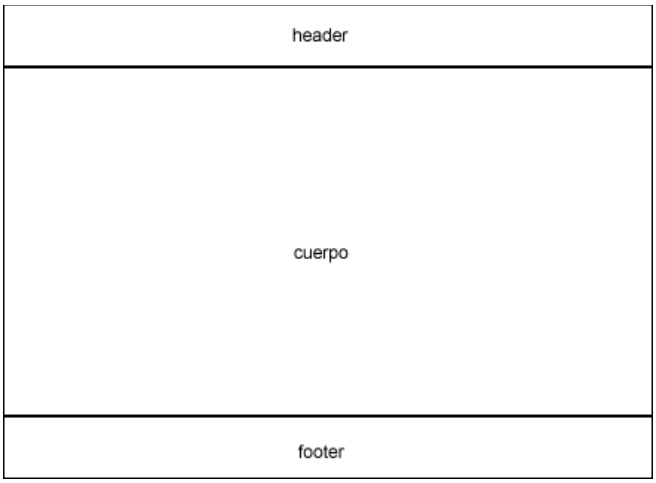
Como muestra el manual, si queremos utilizar smarty debemos crear archivos con extensión tpl en lugar de phtml. **Estos archivos no contienen código php.**

Una vez creado nuestro diseño en photoshop por ejemplo, creamos nuestro archivo HTML básico que tiene la página lista para empezar a programar. En este caso tendremos 6 archivos tpl, uno por cada página, y los situamos dentro de la capeta coop.com.py/v1/views/institucional

De donde salió **institucional?!?!?!?**
Esto es por nuestro controller que veremos en el siguiente componente. Ahora solo entendamos que tenemos un controller llamado institucional porque, como les había explicado, este el primer mundo del sitio completo que tendrá varios otros y corresponde a la parte institucional de la empresa. Entonces en la carpeta de vistas (views) creo una carpeta con el nombre de mi controller y dentro de estos tengo cada template a utilizar.



Bueno. Ahora veremos un poco como interactuar con las vistas. Ellas no son más que el código HTML que muestra la estructura de la página. Notemos que hay dos tpls más de las páginas mencionadas arriba. Estos son header.tpl y footer.tpl. Esto es para ahorrar código y dolores de cabeza a la hora de hacer cambios en secciones que se encuentran repetidas en las páginas que tengamos. Para este caso header son los links de arriba de la página y footer los links que están debajo. La estructura sería así.



Veamos esto en el diseño del sitio.

Esta seria la página principal en HTML directamente en el navegador. Toda la parte azul donde dice "El sabor de la familia y los links historia, compromiso social, productos, noticias y contacto" es el que en todas las páginas, se repite como así también la parte de abajo donde están los links INICIO, TERMINOS Y CONDICIONES, POLITICA DE PRIVACIDAD, FAQ y el Copyright. Esta parte es el footer. Y el cuerpo es lo que irá cambiando de página en página. Aquí tenemos las otras pantallas.


El sabor de la familia
HISTORIA
COMPROMISO SOCIAL
PRODUCTOS
NOTICIAS
CONTACTO
Tropicana
Dueto
DisFruta
Leche Entera
CONTACTO

Bueno como vemos cambia el cuerpo. Pero el header y footer siguen igual. Tomemos en cuenta la página principal para entrar más profundamente en el view.

Pueden acceder a los tpls en el adjunto. Fíjense que en todos los HERF de los links, SRC de los img, SRC de los includes js. y los de CSS están con direcciones absolutas con respecto al HOME de la aplicación, en este caso /V1/ por lo que:

- Javascripts: se graban los js dentro de la carpeta coop.com.py/v1/public/javascript y dentro del tpl la dirección es directamente
  <script src="/v1/javascript/librage.js"></script>

- Css: Misma historia: dentro de la carpeta coop.com.py/v1/public/css por lo que tenemos
  <link href="/v1/css/estilos.css" rel="stylesheet" type="text/css">

- Imágenes: Las imágenes entran dentro de coop.com.py/v1/public/img pero yo cree una carpeta más llamada institucional dentro de img para no mezclar con las imágenes de los demás mundos. Entonces cada mundo tiene su propia carpeta. Lo usamos así
  <img src="/v1/img/institucional/home-str_01.jpg" width="16" height="654" alt="">
- Links: los links los puse también con direcciones absolutas así no me preocupo después para nada. Apuntan al controller "institucional" marcando el action "contacto" en este caso ira a la pagina contacto
  <a href="/v1/institucional/contacto">
- Header y footer: estos archivos como lo vimos arriba, están en la misma carpeta en views así que son llamados directamente ahí.
  {include file="header.tpl"}
  Como verán esa es una instrucción del smarty que dice que en este lugar se debe hacer un <?php include = "header.php" ?> como lo solía yo hacer antes. De esta manera el código está en un solo lugar y si necesito hacer un cambio de link lo hago solo una vez.

Veamos unas instrucciones de smarty antes de pasar al CONTROLLER

- {$noticiasDescripcion|nl2br} $noticiasDescripcion es una variable generada en el controller que el kumbia le pasa al smarty. Entonces, lo que contenga la variable $noticiasDescripcion se vera aquí. El |nl2br hace que con nuevas líneas se creen <br> para mostrar correctamente el texto.

- {section name=row loop=$noticiasSecundarias}
  id}">{$noticiasSecundarias[row]->titulo}
  {/section}

  $noticiasSecundarias es un array dentro del controller (nótese que lo puse en plural) aquí realiza un bucle que va escribiendo una por una las filas del array.

OJO: para que el <title> de las páginas tenga el texto que yo quiera (en este caso .:: www.coop.com.py ::..) lo escribo en el archivo coop.com.py/v1/views/index.phtml en la línea que tiene <title>..:: www.coop.com.py ::..</title>. Este archivo escribe lo básico de las páginas. Así nos ahorra un poco el código. Esto hablando de los <HTML> <HEAD> <TITLE> <BODY> por ejemplo. Es decir que para nuestras páginas tpls borramos el código que ya no hace falta por tener las funcionalidades de este archivo. Noten que borre ciertas etiquetas de este archivo ya que preferí escribir los imports de los css y javascript dentro del HEAD. Por lo que el archivo index.phtml solo escribe hasta antes de cerrar el <HEAD>.

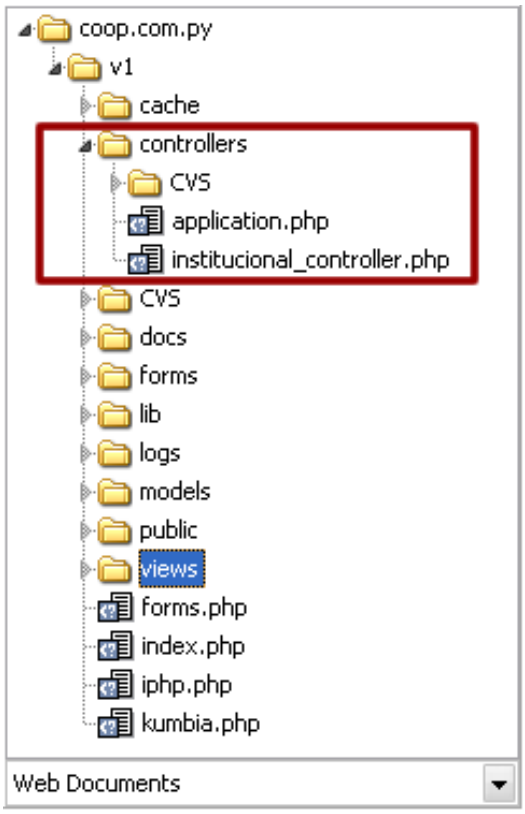
Ok pasemos al siguiente componente MVC

CONTROLLER

El controller permite la interacción entre los dos primeros componentes. Solicita datos al model y los pasa al view para que funcione la aplicación.

Yo cree un controller con varias acciones. Cada acción es un archivo dentro del view a excepción de header y footer que son especiales porque los usamos solo ahí. Entonces tendríamos en la carpeta

coop.com.py/v1/controllers/ el archivo institucional_controller.php. El archivo application.php es un archivo especial del kumbia.



Así tenemos la clase class InstitucionalController extends ApplicationController

Y las acciones son cada function que vemos dentro. Siempre que quiera interactuar con la base de datos creo una instancia del modelo por ejemplo.:
$Noticia = new Noticias(); //-- Aquí $Noticia es el objeto que interactúa con la base de datos.
Para mostrar la notícia principal hago:

```
$noticiaPrincipal = $Noticia->find_first("columns: id, titulo, descripcion",
                                         "conditions: destacada = 'S' AND portal_id = 1");
```

Así busco un solo registro (find_first), con las columnas id, titulo y descripcion cuando el campo desctacada sea Si y el portal igual a 1 que es el mundo institucional para este caso.

Esta variable $noticiaPrincipal vendría a ser mi ResultSet hecho el fetch por lo que accedo a él y grabo cada campo dentro de variables independientes.

```
$this->noticiasId = $noticiaPrincipal->id;
$this->noticiasTitulo = $noticiaPrincipal->titulo;
$this->noticiasDescripcion = substr($noticiaPrincipal->descripcion, 0, 150) . "...";
```

Al hacer $this->variable, esta variable puede ser utilizada dentro del smarty por lo que yendo al view principal.tpl, encontrarán esas variables usadas así {$noticiasId}, {$noticiasTitulo}, {$noticiasDescripcion} esto hace que en modo de ejecución esto sea cambiado por el contenido de esta variable.

OJO: Hay algo interesante aquí que fue un tema que me hizo pensar. Por cada controller hay que crear una carpeta en views y por cada action dentro del controller creamos un archivo tpl (básicamente)
Esto significa que también podría haber hecho un controller por cada página del mundo institucional y dentro de cada controller un action index por ejemplo que tenga el código de cada página. Esto implicaría varias carpetas dentro de views (una por cada controller) y un archivo index.tpl por cada action.
La razón por la que no lo hice de esta manera es que al notar la cantidad de código que me ahorro usando Kumbia me resulto una tontería separarlo por lo que termine haciendo un solo controller con varios actions.

Bueno. En los archivos adjuntos, encontraran en archivo institucional_controller.php con sus 6 actions (uno por cada página), un action llamado index que solo redirige por si alguien escribe solo hasta www.coop.com.py/v1 y un action enviarMail que es invocado al presionar el botón enviar de la página de contactos. Para esto tengo una clase mailSender que la cree dentro de coop.com.py/v1/lib/mail que reutiliza las clases de envió de mail y también una librería de funciones javascript que cree para usarlas en modo de ejecución (esto lo cree en coop.com.py/v1/lib/micayael/ functions_js_v1.php).

Todos los actions tienen comentarios así que pueden endentarlos ahí.

RESUMIENDO

Bueno. Tenemos entonces que a través del **model** interactuamos con la base de datos. Con el **view** creamos la interfaz, y con el **controller** damos vida y unimos ambos componentes.

DESPEDIDA

Usamos smarty para crear el view. Ojala para los siguientes pueda ver con el phtml que creo tiene más funcionalidades.

Esto es todo por el momento amigos. Cualquier sugerencia o cambios para este manual pueden hacerlas a micayael@gmail.com así lo trato de solucionar y lo vuelvo a subir.

```
/**
*
* juanArdissone()
* {
*    Licenciado en Ciencias de la Informática;
*    Especialización en Base de Datos;
*    Web Developer;
*    micayael@gmail.com
*
*    return "PHP, JAVA, javascript, CSS, mysql, FIREBIRD, postgreSQL";
* }
*
**/
```