

# Manul KumbiaPHP

## SICAP

documentación oficial y actualizada

<https://github.com/KumbiaPHP/Documentation/tree/master/es>

# Contenido

Instalación	3
Configuración	3
Hola Kumbia	4
Controlador	5
parámetros de una acción	6
Filtros	6
Filtros de controladores	6
Filtros de acciones	7
Vistas	7
Templates	8
Partials	8
Helpers	9
HTML	9
Tag	11
Form	12
Js	15
Ajax	16
Active Record	17
Modelos	17
Ejercicios	19
Ejercicio parte 1 creación de un controlador	19
Ejercicio parte 2	19
Ejercicio parte 3 creación de plantillas	20
Ejercicio 4 creación de una vista parcial (partial)	21
Ejercicio 5 creación de modelos	22

## Instalación

Los pre-requisitos para instalar KumbiaPHP son mínimos:

- interprete PHP versión 5.4 o superior.
- Servidor Web con soporte de reescritura de URL
  - Apache, Nginx, Cherokee, Lighttpd, Internet Information Server (IIS).
- Manejador de base de datos soportado por KumbiaPHP.

Una vez comprobados los requisitos de instalación se debe descargar el framework desde su repositorio oficial de github <http://www.kumbiaphp.com/blog/manuales-y-descargas/> se encontrarán dos ficheros: .tgz para sistemas unix y .zip para sistemas windows

los archivos deben ser descomprimidos en el directorio raíz (DocumentRoot) del servidor web que se esté utilizando. Una vez descomprimo el archivo encontrara un directorio con la siguiente estructura de subcarpetas dentro

```
-- KumbiaPHP-master
|-- core
|-- vendors
|-- default
|   |-- app
|   |-- public
|   |-- .htaccess
|   `-- index.php
```

figura estructura del directorio de un proyecto KumbiaPHP.

## Configuración

Kumbia utiliza un módulo para reescritura de URLs haciéndolas simples y fáciles de recordar dentro de nuestras aplicaciones, este módulo debe ser activado y/o instalado

- Habilitando ***“mod\_rewrite”*** de Apache en GNU/Linux (Debian, Ubuntu y derivados)
  - Debe ejecutar los siguientes comandos como super usuario para poder realizar las configuraciones necesarias

habilitar módulos

```
> a2enmod rewrite
Enabling module rewrite.
Run '/etc/init.d/apache2 restart' to activate new configuration!
```

- Permitir el uso de los archivos .htaccess
  - se debe editar el siguiente fichero y adicionar el segmento de código que se muestra a continuación, note que debe reemplazar la cadena ***“/to/document/root”*** por el path que apunte hasta la carpeta del proyecto KumbiaPHP que se ha descargado de github. Nota: puede que deba usar el archivo: ***“/etc/apache2/sites-enabled/000-default.config”***. En lugar de solo ***“/etc/apache2/sites-enabled/000-default”***

```
> vi /etc/apache2/sites-enabled/000-default
```

```
<Directory "/to/document/root">
Options Indexes FollowSymLinks
AllowOverride All
Order allow,deny
Allow from all
</Directory>
```

- Hecho esto se debe reiniciar el servidor apache con el comando.

```
>/etc/init.d/apache2 restart
```

## Hola Kumbia

Kumbia se basa en el modelo MVC completamente, de esta manera una Kumbia aceptara una petición, buscara un controlador dentro de todos lo que tenga disponibles, y dentro de este buscara una acción que atienda a la petición. La acción especificara que tipo de vista está asociada a la petición.

El primer paso es crear un controlador en el directorio “*app/controllers/*”. Los controladores deben seguir el formato “*nombre\_controller.php*”. Después dentro del controlador definiremos una “acción” que no será más que un método que KumbiaPHP utilizara para saber que instrucciones ha de ejecutar

```
<?php
/**
 * Controller Saludo
 */
class SaludoController extends ApplicationController {
    public function hola() {

    }
}
```

Figura controladora “Saludo” con acción “hola”

finalmente se debe crear un archivo con extensión “*.phtml*” dentro del directorio “*app/views/saludo/*” el nombre del archivo debe ser idéntico al nombre de la acción que se intenta responder. Note como el nombre de la carpeta “*saludo*” dentro del directorio “*app/views/*” debe corresponder con el nombre de la clase de controlador “*SaludoController*”. Debe advertir que el nombre de un controlador siempre usara el formato “*CamellCase*” mientras que el nombre de una vista debe usar el formato “*minimal\_case*”.

KumbiaPHP tiene un sistema de reescrita de URLs, esto permite el uso de URLs “*bonitas*” y fáciles de usar y recordar, además de que elimina la exhibición de parámetros y de extensiones de archivos. Los URLs de Kumbia están compuestos de la siguiente manera.



Figura esquemas de las rutas y su significado para el framework KumbiaPHP

También es posible el envío de parámetros a través de la URL, tal como se muestra en la siguiente imagen

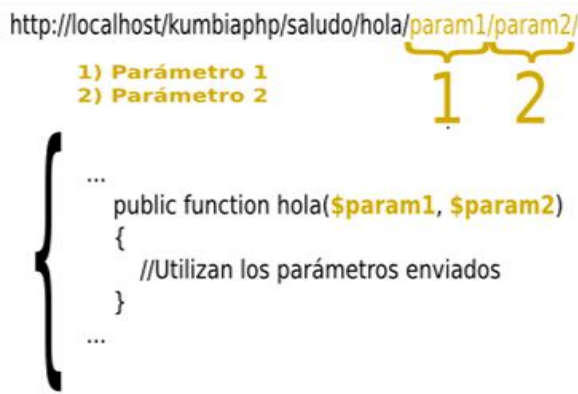


figura Envío de parámetros a través de URL

## Controlador

Es la capa que contiene código encargado de unir la lógica de negocio con la presentación. Esta capa está dividida en múltiples controladores, algunos ya creados y otros que se crean por el desarrollador. Todos estos controladores deben heredar de la súper clase “*AppController*”. El proceso de creación de un componente debe seguir forzosamente los pasos listados a continuación:

- crear un nuevo archivo en el directorio “*app/controllers/*”
- el archivo creado debe ser nombrado siguiendo el formato “*nombreArchivo\_controller.php*”, nótese como el sufijo “*\_controller.php*” es obligatorio
- el contenido del archivo controlador recién creado debe ir entre las etiquetas “*php*” “*<?php ... ?>*”
- se debe crear una clase que represente al controlador
- la clase creada debe heredar(*extends*) obligatoriamente de la superclase “*AppController*”

A continuación, se da un ejemplo creando el controlador Saludo

```
<?php
/**
 * Controller Saludo
 */
class SaludoController extends AppController {
}
```

figura controladora “Saludo”

Un componente tiene una vista (*view*) asociada que se renderiza al llamar una URL definida, un mismo controlador puede llamar tantas vistas como sea necesario, siempre y cuando las vistas estén definidas, cada acción (método) dentro de un componente representará a una vista. De este modo la acción “*saludo()*” llamará a la vista `saludo.phtml`

## parámetros de una acción

las URLs de Kumbia están compuestas de partes importantes, estas son: controlador, acción y parámetros. únicamente controlador y acción son obligatorias mientras que la sección de parámetros puede ser omitida según sea necesario. Se proporciona un ejemplo usando el controlador “*Noticias*”, que contendrá una acción denominada “*ver*” la cual aceptará un único parámetro denominado “*ver*”

<http://www.example.com/noticias/ver/12/>

controlador: noticias

acción: ver

parámetros: 12

```
<?php
/**
 * Controller Noticia
 */
class NoticiasController extends ApplicationController{
    /**
     * método para ver la noticia
     * @param int $id
     */
    public function ver($id){
        echo $this->controller_name;//noticias
        echo $this->action_name;//ver
        //Un array con todos los parámetros enviados a la acción
        var_dump($this-> parameters);
    }
}
```

Figura envío de parámetros para una acción y uso de variable global para almacenar todos los parámetros

si una ruta recibe parámetros adicionales lanzara un error, este es el funcionamiento predefinido de Kumbia, sin embargo, este comportamiento puede ser deshabilitado cuando sea necesario. Bastara con igual la propiedad “*\$limit\_params = false;*” dentro de la definición de la clase controladora que se esté editando. Se proporciona un ejemplo.

```
<?php
/**
 * Controller Saludo
 */
class SaludoController extends ApplicationController {
    /**
     * Limita la cantidad correcta de
     * parámetros de una acción
     */
    public $limit_params = FALSE;
    ... métodos ...
}
```

Figura Deshabilitando el comportamiento predefinido de KumbiaPHP. Error al recibir un numero incorrecto de parámetros

A continuación, se muestra el comportamiento por defecto donde la URL de KumbiaPHP recibe parámetros adicionales y regresa un error. También se muestra el caso en que el comportamiento mencionado es deshabilitado y KumbiaPHP permite la ejecución de la acción con sus respectivos parámetros sin mayor problema

## Filtros

KumbiaPHP posee métodos que realizan comprobaciones antes y después de ejecutar un controlador o una acción. Estos métodos denominados “Filtros” pueden cambiar el procesamiento de una petición y actuar como mecanismos de seguridad, para entre otras cosas verificar si un usuario se encuentra logueado en el sistema. Existen dos categorías de filtros: 1) Filtros de controladores 2) Filtros de acciones.

### Filtros de controladores

Llamados antes o después de la ejecución de un controlador se usan comúnmente para proteger al controlador de información inadecuada, verificar el módulo al que se intenta acceder y comprobar sesiones de usuario. Existen dos métodos dentro de esta categoría

- finalize()
- initialize()

## Filtros de acciones

Llamados antes o después de ejecutar una acción, pueden realizar tareas como verificación de peticiones asíncronas, monitoreo de cambio en los tipos de dato de la petición. Los principales métodos son:

- before\_filter()
- after\_filter()

## Vistas

Las vistas son plantillas de código reutilizable que sirven para mostrar los datos al usuario y se encuentran ubicadas en el directorio ***“app/views/”***. Es buena práctica de desarrollo, que las vistas contengan una cantidad mínima de código en PHP, para que sea suficientemente entendible para un diseñador Web. De ese modo las vistas sólo harán las tareas de visualizar los resultados generados por los controladores y presentarán las capturas de datos para usuarios. primero es utilizar una vista o ***“view”*** asociada a una acción del controlador para convertir los datos que vienen del modelo en lógica de presentación sin especificar ningún formato específico; el segundo paso es establecer el formato de presentación a través de una plantilla o ***“template”***.

Así mismo tanto las vistas de acción como las plantillas pueden utilizar vistas parciales o ***“partials”***. Estas vistas parciales son fragmentos de vistas que son compartidas por distintas vistas, de manera que constituyen lógica de presentación reutilizable en la aplicación. Ejemplos: menús, cabeceras, pies de página, entre otros.

Todos los archivos de vistas deben tener la extensión ***“.phtml”***

Cada controlador tiene un directorio de vistas asociado cuyo nombre coincide con el nombre del controlador en notación ***“smallcase”***. Por ejemplo: si existe un controlador cuya clase se denomina ***“PersonalTecnicoController”*** ésta por convenio tiene un directorio de vistas ***“personal\_tecnico”***.

Cada vez que se ejecuta una acción se intenta cargar una vista cuyo nombre es el mismo que el de la acción ejecutada.

```
<?php
class SaludoController extends ApplicationController
{
    public function saludo()
    {
        View::select('hola');
    }
}
```

Figura controladora llamando una vista específica

```
<?php
class SaludoController extends ApplicationController
{
    public function index()
    {
        View::select(NULL);
    }
}
```

Figura controladora que no llama a ninguna vista, tan solo procesa una petición

Una vista puede recibir parámetros desde un controlador, los parámetros se pasan como variables locales al controlador que invoca a una determinada acción, una vez que una variable ha sido definida dentro del ámbito local de la clase controladora se puede acceder a ella de manera convencional con una simple instrucción ***“echo”***.

```
<?php
class SaludoController extends ApplicationController
{
    public function hola()
    {
        $this->usuario = 'Mundo' ;
    }
}
```

Hola <?php echo \$usuario ?>

Figura declaración de una variable dentro de la acción de un controlador y su posterior impresión en la vista

## Templates

Los templates definen el formato que deben seguir las llamadas vistas de acción (views), se pueden definir tantas templates como sean necesarias estas deben ubicarse en el directorio “*views/\_shared/templates*”. El contenido de una template puede ser como el que se muestra en la siguiente imagen.

```
<!DOCTYPE html>
<html>
<head>
  <title>Template de Ejemplo</title>
</head>
<body>
  <h1>Template de Ejemplo</h1>

  <?php View::content() ?>
</body>
</html>
```

Figura esqueleto de una template

Debe advertir que la instrucción encerrada entre las etiquetas “<?php ?>” indicara a Kumbia la sección que será inyectada desde la acción del controlador

```
<?php
class SaludoController extends ApplicationController
{
    public function hola()
    {
        // Selecciona el template 'mi_template.phtml'
        View::template('mi_template');
    }
}
```

figura indicando la template que será utilizada para resolver la acción que será invocada por el controlador

```
<?php
class SaludoController extends ApplicationController
{
    public function hola()
    {
        // No utilizar template
        View::template(NULL);
    }
}
```

figura indicada al controlador que no se usara ninguna template

para indicarle a Kumbia que template debe usar bastan con utilizar la instrucción “*template*” desde la clase estática View. Tambien es posible indicar a kumbia que no debe renderizar ninguna vista tal como se muestra en la imagen anterior.

## Partials

Las vistas parciales (partials) deben ubicarse en el directorio “*views/\_shared/partials*”

Por defecto se utiliza el template “default” para mostrar las vistas de acción. El envío de parámetros a un partial se realiza de una manera un poco distinta al envío de parámetros en las vistas de acción o él envío de parámetros a una template, en este caso el envío de parámetros a un partial se debe realizar a través de un array asociativo en el que las claves del array serán convertidas en variables de ámbito local

```
<?php View::partial('cabecera', false, array('titulo' =>'Ejemplo')) ?>
```

figura envío de parámetros a un partial. Las claves del arreglo se convierten en variables locales a las que se puede acceder con la notación clásica de php

```
<?php echo $titulo ?>
```



```

<!DOCTYPE html>
<html>
<head>
  <title>Ejemplo</title>
</head>
<body>
  <?php View::partial('cabecera') ?>

  <?php View::content() ?>
</body>
</html>

```

figura uso básico de un partial dentro de una template, aunque los partials pueden ser usados dentro de una vista de acción, o incluso dentro de otros partials

## Helpers

los helpers son bloques de código pre encapsulados que pueden ser utilizados dentro de una vista de acción para reutilizar funcionalidad una y otra vez. KumbiaPHP define 4 grandes categorías de helpers

html: creación de etiquetas html respetando las convenciones de kumbia

Tag: creación de etiquetas para la adicción de CSS o JS así como diversos tipos de metainformación

Form: creación y manejo de formulario

Js: implementaciones de javascript simple

Ajax: Implementaciones de ajax para facilitar su integración con kumbia

### HTML

Html::img()

Permite incluir una imagen

\$src ruta de la imagen

\$alt atributo alt para la imagen

\$attrs atributos adicionales

img (\$src, \$alt=NULL, \$attrs = NULL)

```

/*Ejemplo*/

echo Html::img('spin.gif', 'una imagen'); //se muestra la imagen
spin.gif que se encuentra dentro de "/public/img/"

//con el atributo alt 'una imagen'

```

Html::link()

Permite incluir un link

\$action ruta a la acción

\$text texto a mostrar

\$attrs atributos adicionales

Html::link (\$action, \$text, \$attrs = NULL)

```

/*Ejemplo*/

echo Html::link('pages/show/kumbia/status', 'Configuración'); //se
muestra un link con el texto 'Configuración'

```

Html::lists()

Crea una lista html a partir de un array

\$array contenido de la lista

\$type por defecto ul, y si no ol  
\$attrs atributos adicionales

```
Html::lists($array, $type = 'ul', $attrs = NULL)
```

Html::gravatar()  
Incluye imágenes de gravatar.com  
\$email Correo para conseguir su gravatar  
\$alt Texto alternativo de la imagen. Por defecto: gravatar  
\$size Tamaño del gravatar. Un numero de 1 a 512. Por defecto: 40  
\$default URL gravatar por defecto si no existe, o un default de gravatar. Por defecto: mm  
Html::gravatar(\$email, \$alt='gravatar', \$size=40, \$default='mm')

```
echo Html::gravatar($email); // Simple
echo Html::link(Html::gravatar($email), $url); // Un gravatar que es un link
echo Html::gravatar($email, $name, 20, 'http://www.example.com/default.jpg');
//Completo
```

Html::includeCss()  
Incluye los archivos CSS que previamente fueron cargados a la lista mediante Tag::css()

```
Tag::css('bienvenida'); //Pone en lista un CSS (app/public/css/bienvenida.css)
echo Html::includeCss(); //Adiciona los recursos enlazados de la clase en el proyecto
```

Html::meta()  
Crea un metatag y lo agrega a una lista estática que será añadida más adelante mediante Html::includeMetatags();  
\$content contenido del metatag  
\$attrs atributos adicionales del tag  
Html::meta(\$content, \$attrs = NULL)

```
Html::meta('Kumbiaphp-team', "name = 'Author'");
//Agrega: <meta content="Kumbiaphp-team" name = 'Author' />
Html::meta('text/html; charset=UTF-8', "http-equiv = 'Content-type'");
//Agrega: <meta content="text/html; charset=UTF-8" http-equiv = 'Content-type' />
```

Html::headLink()  
Agrega un elemento de vínculo externo de tipo a la cola de enlaces (para poder ser visualizado se requiere de  
Html::includeHeadLinks() de modo similar que Html::includeCss())  
\$href dirección url del recurso a enlazar  
\$attrs atributos adicionales

```
Html::headLink($href, $attrs = NULL)

Html::headlink('http://www.kumbiaphp.com/public/style.css', "rel='stylesheet', type='text/css' media='screen'")
//Se agrega a la cola de links el enlace a un recurso externo, en este caso la hoja de estilo ubicada en "http

/*Agrega a la cola de links "<link rel="alternate" type="application/rss+xml"
title="KumbiaPHP Framework RSS Feed" href="http://www.kumbiaphp.com/blog/feed/" />" con lo cual
podemos incluir un feed sin usar las convenciones de kumbiaphp */

Html::headlink('http://www.kumbiaphp.com/blog/feed/', "rel='alternate'
type='application/rss+xml' title='KumbiaPHP Framework RSS Feed'");

Html::headlink('http://www.kumbiaphp.com/favicon.ico', "rel='shortcut
icon', type='image/x-icon'"); //Agrega la etiqueta <link> para usar un favicon externo

echo Html::includeHeadLinks(); //Muestra los links que contiene la cola
```

Html::headLinkAction()  
Agrega un elemento de vínculo interno de tipo a la cola de enlaces (para poder ser visualizado se requiere de  
Html::includeHeadLinks() de modo similar que Html::includeCss()) respetando las convenciones de KumbiaPHP.

\$href dirección url del recurso a enlazar  
 \$attrs atributos adicionales  
 Html::headLinkAction(\$action, \$attrs = NULL)

```
/*Agrega a la cola de links "<link rel="alternate" type="application/rss+xml"
title="KumbiaPHP Framework RSS Feed" href="http://www.kumbiaphp.com/blog/feed/" />" con lo cual podemos incluir un feed usando las convenciones de KumbiaPHP.
Siendo 'articulos/feed' el nombre de la vista con el contenido del feed */

Html::headLinkAction('articulos/feed', "rel='alternate'
type='application/rss+xml' title='KumbiaPHP Framework RSS Feed'");
echo Html::includeHeadLinks(); //Muestra los links que contiene la cola
```

Html::headLinkResource()

Agrega un elemento de vínculo a un recurso interno con la etiqueta a la cola de enlaces (para poder ser visualizado se requiere de Html::includeHeadLinks())

\$resource ubicación del recurso en public

\$attrs atributos adicionales

Html::headLinkResource(\$resource, \$attrs = NULL)

```
//Agrega la etiqueta <link> para usar un favicon interno ubicado en el directorio '/public/'
Html::headLinkResource('favicon.ico', "rel='shortcut
icon', type='image/x-icon'");
echo Html::includeHeadLinks(); //Muestra los links que contiene la cola
```

Html::includeHeadLinks()

Incluye los links que previamente se pusieron en cola

```
Html::headLink('http://www.kumbiaphp.com/favicon.ico', "rel='shortcut
icon', type='image/x-icon'"); //Agrega la etiqueta <link> para usar un favicon externo

Html::headLinkAction('articulos/feed', "rel='alternate' type='application/rss+xml' title='KumbiaPHP Framework
echo Html::includeHeadLinks();
```

## Tag

Tag::css()

Incluye un archivo CSS a la lista

```
Tag::css('bienvenida'); //Pone en lista un CSS (app/public/css/bienvenida.css)
echo Html::includeCss(); //Adiciona los recursos enlazados de la clase en el proyecto
```

Tag::js()

Incluye un archivo JavaScript a la vista, partial o template

```
<?= Tag::js('jquery/jquery.kumbiaphp'); //Adiciona un archivo javascript (/app/public/javascript/jquery/jquery
?>
```

Form::open()

Crea una etiqueta de formulario

\$action acción a la que envía los datos, por defecto llama la misma acción de donde proviene

\$method 'POST', 'GET', 'DELETE', 'HEAD', 'PUT'. Por defecto se tiene en 'POST'

\$attrs atributos adicionales

Form::open(\$action = NULL, \$method = 'POST', \$attrs = NULL)

```
/*Ejemplo*/
<?= Form::open() ?> //inicia un formu
<?= Form::open('usuarios/nuevo') ?>
```

## Form

Form::openMultipart()

Crea una etiqueta de formulario multipart, este es ideal para formularios que contienen campos de subida de archivos

\$action acción a la que envía los datos, por defecto llama la misma acción de donde proviene  
\$attrs atributos adicionales

Form::openMultipart (\$action = NULL, \$attrs = NULL)

```
/*Ejemplo*/
//inicia un formulario multipart que enviará los datos a la acción que corresponde a la vista actual
echo Form::openMultipart();
//inicia un formulario multipart que enviará los datos al controller 'usuario' y la acción 'nuevo'
echo Form::openMultipart('usuarios/nuevo');
```

Form::close()

Crea una etiqueta de cierre de formulario

```
/*Ejemplo*/
echo Form::close();
//crea una etiqueta de cierre de formulario </form>
```

Form::input()

Crea un campo de tipo input

\$attrs atributos para el tag

\$content contenido interno

Form::input(\$attrs = NULL, \$content = NULL)

```
/*Ejemplo*/
echo Form::input('nombre');
```

Form::text()

Crea un campo de tipo input

Siempre que se le da el parámetro name de la forma model.campo, es decir un nombre que contenga un punto dentro del string, se crea el campo de texto con el name="model[campo]" y el id="model\_campo".

\$field Nombre de campo

\$attrs atributos de campo

\$value valor inicial para el input

Form::text(\$field, \$attrs = NULL, \$value = NULL)

```
/*Ejemplo*/
//crea un campo de tipo texto con el parámetro name= "nombre", id = "nombre"
echo Form::text('nombre');
//crea un campo de tipo texto con el parámetro name= "usuario[nombre]", id = "usuario.nombre"
echo Form::text('usuario.nombre');
//crea un campo de tipo texto con el parámetro name= "nombre", id = "nombre", class= "caja", value = "55"
echo Form::text('nombre', "class='caja'", '55');
```

Form::pass()  
Crea un

campo de tipo Password

\$field nombre de campo

\$attrs atributos de campo

\$value valor inicial para el campo

Form::pass(\$field, \$attrs = NULL, \$value = NULL)

```
/*Ejemplo*/
echo Form::pass('password'); //crea un campo de tipo password con el
parámetro name= "password"
```

Form::textarea()  
Crea un textarea  
\$field nombre de campo  
\$attrs atributos de campo  
\$value valor inicial para el textarea  
Form::textarea(\$field, \$attrs = NULL, \$value = NULL)

```
echo Form::textarea('detalles'); //Crea un textarea
```

Form::label()  
Crea un label y lo asocia a un campo  
\$text texto a mostrar  
\$field campo al que hace referencia  
\$attrs array de atributos opcionales  
Form::label(\$text, \$field, \$attrs = NULL)

```
//Crea un label  
para el campo nombre con el texto 'nombre de usuario:'  
echo Form::label('nombre de usuario:', 'nombre');  
echo Form::text('nombre');
```

Form::hidden()  
Crea un campo hidden (campo oculto)  
\$field nombre de campo  
\$attrs atributos adicionales de campo  
\$value valor inicial para el campo oculto  
Form::hidden(\$field, \$attrs = NULL, \$value = NULL)

```
echo Form::hidden('id', null, 12); //Crea un campo oculto con el name="id" y el value="12"
```

Form::dbSelect()  
Crea campo Select que toma los valores de objetos de ActiveRecord, para esta versión del framework el uso de este helper ha sido simplificado. Ya no es necesario instanciar el modelo.  
\$field nombre del modelo y campo pk (bajo la convención modelo.campo\_id)  
\$show campo que se mostrará  
\$data array de valores, array('modelo','m e todo','param')  
\$blank campo en blanco  
\$attrs atributos de campo  
\$value valor inicial para el campo

```
//la forma más facil, carga el modelo(campo) y muestra el primer campo después del pk(id)  
echo Form::dbSelect('usuarios.campo_id');  
//muestra el campo y lo ordena ascendentemente  
echo Form::dbSelect('usuarios.campo_id', 'campo');
```

Form::select()  
Crea un campo Select (un combobox)  
\$field nombre de campo  
\$data array de valores para la lista desplegable  
\$attrs atributos de campo  
\$value valor inicial para el campo



Form::select(\$field, \$data, \$attrs = NULL, \$value = NULL)

```
$ar2 = array('Abdomen', 'Brazos', 'Cabeza', 'Cuello', 'Genitales', 'Piernas', 'Torax', 'Otros');  
//Crea un campo Select (un combobox) con el nombre 'region' y teniendo preseleccionado 'Cuello'  
echo Form::Select('region', $ar2, null, 'Cuello');
```

Form::file()

Crea campo File para subir archivos, el formulario se debe abrir con Form::openMultipart()

\$field nombre de campo

\$attrs atributos de campo

Form::file(\$field, \$attrs = NULL)

```
echo Form::openMultipart(); //Abre el formulario multipart  
echo Form::file('subir'); //Crear el campo para subir archivos  
echo Form::close(); //Cierra el formulario
```

Form::button()

Crea un botón

\$text texto del botón

\$attrs atributos del botón

Form::button(\$text, \$attrs = NULL)

```
echo Form::button('calcular'); //Crea un botón con el texto 'calcular'
```

Form::submitImage()

Crea un botón de tipo imagen siguiendo las convenciones de KumbiaPHP, la imagen deberá estar dentro del directorio '/public/img/'

\$img ruta de la imagen que usa el botón

\$attrs atributos del botón

Form::submitImage(\$img, \$attrs = NULL)

```
echo Form::submitImage('botones/edit.gif'); //Crea un botón con la imagen 'botones/edit.gif'
```

Form::submit()

Crea un botón de submit para el formulario actual

\$text texto del botón

\$attrs atributos del botón

Form::submit(\$text, \$attrs = NULL)

```
echo Form::submit('enviar'); //Crea un botón con el texto 'enviar'
```

Form::reset()

Crea un botón reset para el formulario actual

\$text texto del botón

\$attrs atributos del botón

Form::reset(\$text, \$attrs = NULL)

```
echo Form::reset('reiniciar'); //Crea un botón con el texto  
'reiniciar'
```

Form::check()

Crea un checkbox

\$field nombre de campo

\$value valor en el checkbox  
\$attrs atributos de campo  
\$checked indica si se marca el campo  
Form::check(\$field, \$value, \$attrs = NULL, \$checked = NULL)

```
// Crea un check seleccionado con id="recuerdame" , name="recuerdame" y value="1"
echo Form::check ( 'recuerdame' , '1' , '' , true );
// Crea un check NO seleccionado con id="recuerdame" , name="recuerdame" y value="1"
echo Form::check ( 'recuerdame' , '1' , '' , false );
```

Form::radio()  
Crea un radio button  
\$field nombre de campo  
\$value valor en el radio  
\$attrs atributos de campo  
\$checked indica si se marca el campo  
Form::radio(\$field, \$value, \$attrs = NULL, \$checked = NULL)

```
$on = 'masculino' ;
//<input id="rdo1" name="rdo" type="radio" value="masculino" checked="checked">
echo Form::radio("rdo", 'masculino', null, true);
//<input id="rdo2" name="rdo" type="radio" value="femenino">
echo Form::radio("rdo", 'femenino');
```

## Js

Js::link ()  
Crea un enlace que al pulsar muestra un diálogo de confirmación para redireccionamiento a la ruta indicada.  
\$action ruta a la acción  
\$text texto a mostrar  
\$confirm mensaje de confirmación  
\$class clases adicionales para el link  
\$attrs \$attrs atributos adicionales

Js::link (\$action, \$text, \$confirm = '¿Está Seguro?', \$class = NULL, \$attrs = NULL)

```
<?= Js::link('usuario/eliminar/5', 'Eliminar') ?>

<?= Js::link('usuario/eliminar/5', 'Eliminar', '¿Está seguro de esta
operación?', 'b_eliminar') ?>
```

Js::linkAction ()  
Crea un enlace que al pulsar muestra un diálogo de confirmación para redireccionamiento a la acción indicada.  
\$action acción de controlador  
\$text texto a mostrar  
\$confirm mensaje de confirmación  
\$class clases adicionales para el link  
\$attrs \$attrs atributos adicionales

Js::linkAction(\$action, \$text, \$confirm = '¿Está Seguro?', \$class = NULL, \$attrs = NULL)

```
<?php echo Js::linkAction('eliminar/5', 'Eliminar'); ?>
//Si desea aplicar una clase de estilo al enlace debe indicarlo en el argumento $class
<?php echo Js::linkAction('eliminar/5', 'Eliminar', '¿Está seguro de esta operación?', 'b_eliminar') ?>
```

Js::submit ()

Crea un botón submit que al pulsar muestra un diálogo de confirmación.

\$text texto a mostrar

\$confirm mensaje de confirmación

\$class clases adicionales para el link

\$attrs atributos adicionales

Js::submit (\$text, \$confirm = '¿Está Seguro?', \$class = NULL, \$attrs = NULL)

```
<?php echo Js::submit('Guardar') ?>
//Si desea aplicar una clase de estilo al botón debe indicarlo en el argumento $class .
<?= Js::submit('Guardar', '¿Está Seguro?', 'boton_guardar') ?>
```

Js::submitImage ()

Crea un botón tipo image que al pulsar muestra un diálogo de confirmación.

\$img ruta a la imagen

\$confirm mensaje de confirmación

\$class clases adicionales para el link

\$attrs atributos adicionales

Js::submitImage(\$img \$confirm = '¿Está Seguro?', \$class = NULL, \$attrs = NULL)

```
<?php echo Js::submitImage('botones/guardar.png') ?>
//Si desea aplicar una clase de estilo al botón debe indicarlo en el argumento $class .
<?= Js::submitImage('botones/guardar', '¿Está Seguro?', 'boton_guardar') ?>
```

## Ajax

Ajax::link()

Crea un enlace que actualiza la capa indicada con el contenido producto de la petición web.

\$action ruta a la acción

\$text texto a mostrar

\$update capa a actualizar

\$class clases adicionales

\$attrs atributos adicionales

Ajax::link (\$action, \$text, \$update, \$class=NULL, \$attrs=NULL)

```
<div id="capa_saludo"></div>
<?php
    echo Ajax::link('saludo/hola', 'Mostrar Saludo', 'capa_saludo');
    echo Tag::js('jquery/jquery+kumbiaphp.min');
?>
```

Ajax::linkAction()

Crea un enlace a una acción del controlador actual que actualiza la capa indicada con el contenido producto de la petición web.

\$action acción

\$text texto a mostrar

\$update capa a actualizar

\$class clases adicionales

\$attrs atributos adicionales

Ajax::linkAction (\$action, \$text, \$update, \$class=NULL, \$attrs=NULL)

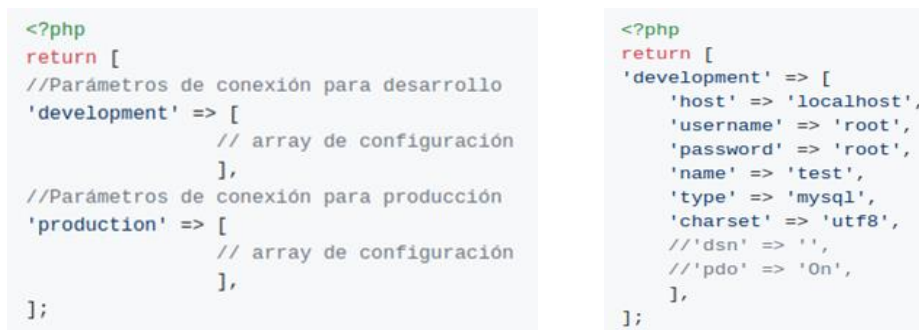
```
<?= Ajax::linkAction('hola', 'Mostrar Saludo', 'capa_saludo') ?>
```



## Active Record

Es la clase base para toda aquella funcionalidad relacionada con el mapeo de tablas SQL en objetos POO. Esta clase base implemente al patrón ORM: Tablas = clases, objetos = atributos. De esta manera los campos de las tablas serán convertidos automáticamente en propiedades adicionalmente esta clase base presenta métodos predefinidos que aportan funcionalidad automáticamente: `ind()`, `find_first()`, `save()`, `update()`.

Antes de comenzar a trabajar con esta clase y sus subclases debe configurar el acceso del framework a la base de datos. Puede realizar esta acción editando el archivo “default/app/config/databases.php”. En versiones anteriores de Kumbia podrá encontrar este archivo con el nombre de `databases.ini`, sin embargo, el uso de este tipo de archivos ya está desaconsejado. El archivo `databases.php` se compone de múltiples arreglos que representan a cada una de las conexiones que pueden existir (tantas como sea necesario) por defecto el archivo contendrá 2 claves dentro del arreglo, estas representarán a la conexión para desarrollo (development) y para producción (production) para conectarse a una base de datos bastará con llenar cada uno de los campos con la información perteneciente a la base de datos objetivo



```
<?php
return [
    //Parámetros de conexión para desarrollo
    'development' => [
        // array de configuración
    ],
    //Parámetros de conexión para producción
    'production' => [
        // array de configuración
    ],
];
```

```
<?php
return [
    'development' => [
        'host' => 'localhost',
        'username' => 'root',
        'password' => 'root',
        'name' => 'test',
        'type' => 'mysql',
        'charset' => 'utf8',
        //'dsn' => '',
        //'pdo' => 'On',
    ],
];
```

figura configuración de ficheros de conexión para base de datos

- Host: Ip o nombre del host de la base de datos
- Username: Nombre de usuario con permisos en la base de datos, no es recomendable usar el usuario root
- Password: Clave del usuario de la base de datos
- Name: Nombre de la base de datos
- Type: Tipo de motor de base de datos (mysql, pgsql, oracle o sqlite)
- Charset: Conjunto de caracteres de conexión, por ejemplo 'utf8'
- Dsn: Cadena de conexión a la base de datos (Opcional)
- Pdo: Para activar conexiones PDO (On/Off)

## Modelos

Los modelos representan la lógica de la aplicación y son parte fundamental de desarrollo de la aplicación, su buen uso asegura un correcto escalado, mantenimiento y reutilización del código. Por lo general un mal uso de los modelos es dejar la clase con solamente la declaración y delegar toda la lógica al controlador. Los controladores deberían presentar la mínima cantidad de lógica de procesamiento, los controladores únicamente se encargan de atender las peticiones del usuario y solicitar dicha información a los modelos.

Para crear un modelo que represente a una tabla de base de datos se debe crear un nuevo archivo con el nombre de la tabla en minúsculas y con extensión “*.php*” dentro del directorio `/models/`. Se muestra un breve ejemplo usando la tabla denominada “*menu*”, cuya definición es la siguiente:

```
CREATE TABLE menus
(
  id          int          unique not null auto_increment,
  nombre      varchar(100),
  titulo      varchar(100) not null,
  primary key(id)
)
```

```
<?php

class Menu extends ActiveRecord
{
  /**
   * Retorna los menús para ser paginados
   *
   * @param int $page [requerido] página a visualizar
   * @param int $page [opcional] por defecto 20 por página
   */
  public function getMenu($page, $page=20)
  {
    return $this->paginate("page: $page", "per_page: $page", 'order: id desc');
  }
}
```

figura mapeo de una tabla (izquierda) de base de datos en una clase de KumbiaPHP (derecha) creación del modelo “Menu” en “/models/menu.php”

## Ejercicios

### Ejercicio parte 1 creación de un controlador

Se creará un nuevo archivo en el directorio “/default/app/controllers/” con el nombre “*clientes\_controller.php*” el nombre de un controlador **SIEMPRE** debe incluir el sufijo “*\_controller.php*”

Dentro del controlador se definen 4 métodos que representaran a cada una de las operaciones CRUD ( Create, Update, Delete) se pueden definir tantos métodos como sea necesario y estos pueden llevar cualquier nombre

```
1 <?php
2 class clientesController extends ApplicationController
3 {
4     public function index()
5     {
6     }
7 }
8
9     public function create()
10    {
11    }
12
13    public function update()
14    {
15    }
16
17    public function delete()
18    {
19    }
20 }
21
```

Figura controlador cliente con métodos CRUD

### Ejercicio parte 2

creación de vistas

se creará una subcarpeta en el directorio “/default/app/views/” note como el nombre de la subcarpeta debe llevar el mismo nombre que controlador que recién se acaba de crear “/default/app/views/clientes/”.

Se creará un nuevo archivo dentro del directorio “/default/app/views/clientes/” este archivo representará a alguno de los métodos definidos en la clase controladora del ejercicio 1. Se creará el archivo “*index.phtml*”. la página creada puede ser verificada en la ruta

<http://localhost/kumbiapp/clientes>

```
index.phtml ×
default > app > views > clientes > index.phtml
1 <h1>Hola desde la vista index</h1>
2 <p>
3 Lorem ipsum dolor, sit amet consectetur adipisicing elit.
4 Nemo ullam velit ea sapiente dolores! Iure a, quae quia magni
5 cupiditate incidunt, eligendi eos et, nihil nostrum necessitatibus
6 sint adipisci quo!
7 </p>
```

Figura contenido de la vista de acción index.phtml y su visualización



App: default (Development)

## Hola desde la vista index

Lorem ipsum dolor, sit amet consectetur adipisicing elit. Nemo ullam velit ea sapiente dolores! Iure a, quae quia magni cupiditate incidunt, eligendi eos et, nihil nostrum necessitatibus sint adipisci quo!

[KumbiaPHP Framework](#) | [Github](#) | [Manual](#) | [Wiki](#) | [Licencia](#)

Ayuda Online: [Stack](#) | [M1](#) | [Grupo](#) | [Foro](#)

© 2007 - 2020 KumbiaPHP Team

Ejecutado en 203.413 ms. usando 0.418 MB de memoria

el contenido del archivo **“phtml”** es renderizado correctamente sin embargo se han agregado elementos adicionales que nunca se le indicaron expresamente a kumbia en la vista, este comportamiento se debe a que kumbia inyecta el contenido de toda vista de acción dentro de una plantilla. Posteriormente se detallará como trabajar con estas plantillas

## Ejercicio parte 3 creación de plantillas

Para crear una nueva plantilla se debe crear un nuevo archivo con extensión **“phtml”** en el directorio **“/default/app/views/templates/”** en este caso se creará el archivo **“bootstraptemplate.phtml”**

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4 <meta charset="UTF-8">
5
6 <meta name="viewport" content="width=device-width, initial-scale=1.0">
7 <title>PlantillaBootstrap</title>
8
9 <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css" integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1tQ884/jq"
10 </head>
11
12 <body>
13 <?php View::content();
14
15 <div class="container-12 m-5">
16 <?php View::content();
17 </div>
18
19 </body>
20
21 </html>

```

Figura plantilla con CDN de Bootstrap y clases para dar formato al HTML inyectado

Debe advertir que esta plantilla contiene meta información para incluir la CDN del framework para desarrollo front-end Bootstrap 4, también debe advertir la inclusión del helper **“View::content()”** que se encargara de inyectar el contenido de las vistas de acción dentro de la plantilla permitiendo a cualquier vista acceder a todo tipo de datos provenientes de la metainformación u otros recursos. A continuación, se agrega un margen y un contenedor para dar un formato a la vista de acción

```

4 public function index()
5 {
6     View::template('bootstraptemplate');
7 }
8
9

```

Figura instrucción para usar una plantilla específica

## Hola desde la vista index

Lorem ipsum dolor, sit amet consectetur adipisicing elit. Nemo ullam velit ea sapiente dolores! lure a, quae quia magni cupiditate incidunt, eligendi eos et, nihil nostrum necessitatibus sint adipisci quo!

Con la instrucción **“view::template(‘bootstraptemplate’)”** se indica a kumbia la plantilla que debe usar y dentro de la cual inyectara el contenido de la vista de acción. Si se reviza la ruta <http://localhost/kumbiaphp/clientes> se podrá obtener un resultado similar al siguiente, en el que la vista ha cambiado por completo

### Ejercicio 4 creación de una vista parcial (partial)

Los partilas inyectan segmentos de HTML dentro de una página principal como una plantilla, una vista de acción o incluso otros partials. En este ejemplo se creará una barra de navegación y un footer para la plantilla del ejemplo 3

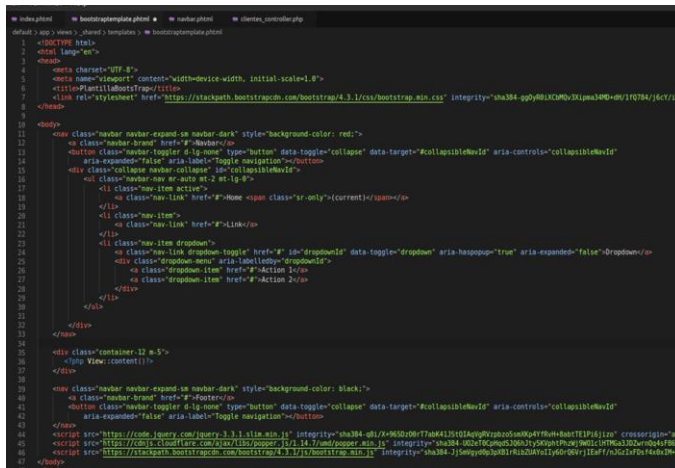


Figura una template sin uso de partials y su resultado. Se aprecia la gran cantidad de código generado



Se aprecia como el objetivo se logra tan solo con agregar el respectivo código html a la plantilla, pero también se vuelve evidente la inmaterialidad del código generado al volcar directamente el html en la plantilla, kumbia permite usar los partials para resolver este problema. A continuación, se crearán partials para encapsular al footer y al navbar

para crear un nuevo partial se debe generar un archivo con la extensión ***“phtml”*** en el directorio ***“/default/app/views/partials/”*** se crearán dos archivos ***“nav.phtml”*** y ***“footer.phtml”***

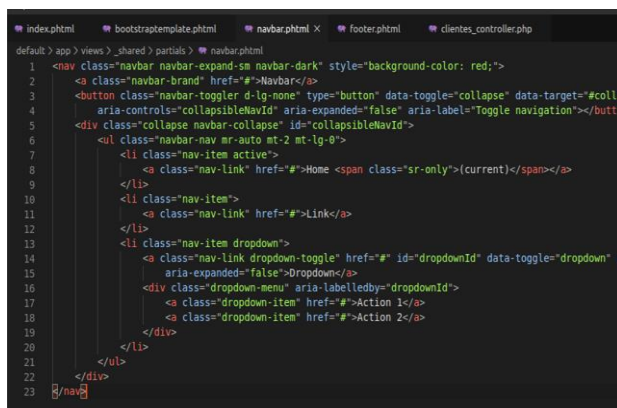
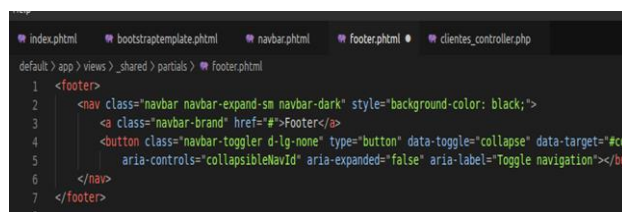


Figura partial navbar.php y footer.php



```

index.phtml bootstraptemplate.phtml navbar.phtml footer.phtml clientes_controller.php
default > app > views > _shared > templates > bootstraptemplate.phtml
6 <title>PlantillaBootstrap</title>
7 <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.m
8 </head>
9
10 <body>
11 <?php View::partial('navbar') ?>
12
13 <div class="container-12 m-5">
14 <?php View::content() ?>
15 </div>
16
17 <?php View::partial('footer') ?>
18
19 <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965Dz00r
20 <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js" integri
21 <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js" integrity=
22 </body>
23 </html>

```

Figura inclusión de un partial dentro de una template

Se debe indicar a la plantilla los partials que debe usar, esto se logra con la instrucción “*View::partial(‘nombreparital’)*”. Se puede observar cómo Código es extremadamente simple y el resultado es el mismo

## Ejercicio 5 creación de modelos

Este ejercicio tomara información de una base de datos de MySQL y la mostrara dentro de las secciones que se han creado. Antes de comenzar se debe configurar el fichero “*databases.php*” en el directorio “*/default/app/config/*” tal como en el ejemplo

```

index.phtml bootstraptemplate.phtml databases.php navbar.phtml footer.phtml
default > app > config > databases.php
4 * Parámetros de conexión a la base de datos
5 */
6 return [
7     'development' => [
8         /**
9          * host: ip o nombre del host de la base de datos
10        */
11        'host' => 'localhost',
12        /**
13         * username: usuario con permisos en la base de datos
14        */
15        'username' => 'frodo', //no es recomendable usar el usuario root
16        /**
17         * password: clave del usuario de la base de datos
18        */
19        'password' => '2010_Wflsy0?!',
20        /**
21         * test: nombre de la base de datos
22        */
23        'name' => 'dbsistema',
24        /**
25         * type: tipo de motor de base de datos (mysql, postgresql, oracle o sqlite)
26        */
27        'type' => 'mysql',
28        /**
29         * charset: Conjunto de caracteres de conexión, por ejemplo 'utf8'
30        */
31        'charset' => 'utf8',
32        /**
33         * dsn: Cadena de conexión a la base de datos
34        */
35        'dsn' => '',
36        /**
37         * pdo: activar conexiones PDO (On/Off); descomentar para usar
38        */
39        'pdo' => 'On',
40    ],
41 ];

```

una vez configurado el acceso a la base de datos se debe crear un modelo para representar a cada tabla en la base de datos. Tenga en cuenta que debe crear un nuevo modelo por cada tabla que desee ser usada. Estos modelos deben crearse en la carpeta “*/default/app/models*”, se debe crear un archivo con el nombre de la tabla en minúsculas y con la extensión “*.php*” en este caso se creará el archivo “*categorias.php*”

Figura configuración del archivo databases.php. Se muestra también la definición de la tabla “categoria”

	idcategoria	nombre	descripcion	condicion
<input type="checkbox"/> Editar <input type="checkbox"/> Copiar <input type="checkbox"/> Borrar	1	categoria 1	ninguna	1
<input type="checkbox"/> Editar <input type="checkbox"/> Copiar <input type="checkbox"/> Borrar	2	categoria 2	ninguna	1
<input type="checkbox"/> Editar <input type="checkbox"/> Copiar <input type="checkbox"/> Borrar	3	categoria 3	ninguna	1
<input type="checkbox"/> Editar <input type="checkbox"/> Copiar <input type="checkbox"/> Borrar	4	categoria n	ninguno	1
<input type="checkbox"/> Editar <input type="checkbox"/> Copiar <input type="checkbox"/> Borrar	5	categoria s	ninguno	1
<input type="checkbox"/> Editar <input type="checkbox"/> Copiar <input type="checkbox"/> Borrar	6	categoria k	ninguna	1

```

index.phtml bootstraptemplate.phtml databases.php
default > app > models > categoria.php > ...
1 <?php
2 class Categoria extends ActiveRecord {
3
4 }
5

```

Figura creación del modelo Categoria



adverta como los campos serán mapeados automáticamente por kumbia, esto libera al usuario de la necesidad de crear código SQL adicional. Se añadirá un método al modelo recién creado para traer todos los registros de la tabla “categorias”, paginarlos de 3 en 3 y ordenarlos de forma descendente

```

1 <?php
2 class Categoria extends ActiveRecord {
3     function getCategories($page, $ppage=3)
4     {
5         return $this->paginate("page: $page", "per_page: $ppage", "order: idCategoria desc");
6     }
7 }

```

Figura creación de método para obtener información de la base de datos, nótese como toda la lógica de extracción esta especificada en el modelo

ahora solo queda usar el modelo recién creado dentro de un controlador para obtener la información de la base de datos

```

1 <?php
2 class clientesController extends ApplicationController
3 {
4     public function index( $page = 1 )
5     {
6         View::template('bootstraptemplate');
7         $this->categoriasMenu = (new Categoria)->getCategories( $page );
8     }
9
10
11     public function create()
12     {
13     }
14
15     public function update()
16     {
17     }
18
19     public function delete()
20     {
21     }
22 }

```

Figura llamada desde un controlador a los métodos de extracción de información de un modelo

Hola desde la vista index

Lorem ipsum dolor, sit amet consectetur adipisicing elit. Nemo abbi veli ea sapiente dolores! Iure a, quae quia magni cupiditate incidunt, eligendi eos et, nihil nostrum necessitatibus ut adipisci quo!

```

1 <?php echo var_dump($categoriasMenu); ?>
2
3
4
5
6
7
8
9

```

Footer

Figura impresión de la variable local en la cual se almacena la información obtenida desde el modelo

Observe como la variable local “*\$this→categoriasMenu*” es usada dentro de una vista de acción index “*\$categoriasMenu*”. El procedimiento es el mismo para usar una variable definida en un controlador desde una plantilla, sin embargo, el procedimiento difiere al intentar usar esta información desde una template.

```
<?php View::partial('navbar', false, array( 'mencat' => $categoriasMenu ) ) ?>
```

para eso se agregaran un par de parámetros al método ***“partial()***, de la plantilla creada previamente. El parámetro **false** indica que nunca expiraran las cookies provistas por el partial mientras el array asociativo contiene todas las variables que serán enviadas al partial, sus claves serán convertidas en variables que podrán ser usadas dentro del partial

```

7 <li class="nav-item active">
8 <a class="nav-link" href="#">Home <span class="sr-only">(current)</span></a>
9 </li>
10 <a class="nav-item">
11 <a class="nav-link" href="#">Link</a>
12 </li>
13 <li class="nav-item dropdown">
14 <a class="nav-link dropdown-toggle" href="#" id="dropdownId" data-toggle="dropdown"
15     aria-expanded="false">Dropdown</a>
16 <div class="dropdown-menu" aria-labelledby="dropdownId">
17 <a class="dropdown-item" href="#">Action 1</a>
18 <a class="dropdown-item" href="#">Action 2</a>
19 </div>
20 </li>
21 </ul>
22 </div>
23 </nav>
24
25 <php echo var_dump($_SERVER); >

```

[illegible]

```

youw@ ~ $ cd /home/youw/Projects/youw/youw && ls
app.component.html  TS youtube.service.ts  index.php  TS app.component.ts
youw@ ~ $ src /app /services && index.php
1  <nav class="navbar navbar-expand-sm navbar-dark bg-primary">
2    <a class="navbar-brand" href="#">Navbar</a>
3    <button class="navbar-toggler d-lg-none" type="button" data-toggle="collapse" data-target="#navbarSupportedContent"
4      aria-expanded="false" aria-label="Toggle navigation"></button>
5    <div class="collapse navbar-collapse" id="navbarSupportedContent">
6      <ul class="navbar-nav mr-auto mt-2 mt-lg-0">
7        <?php foreach ($menu->items as $elemento) : ?>
8          <li class="nav-item active">
9            <a class="nav-link" href="#">
10              $elemento->nombre
11            </a>
12          </li>
13        <?php endforeach; ?>
14      </ul>
15    </div>
16  </nav>

```

Navbar

[categoria k](#) [categoria s](#) [categoria n](#) [categoria 3](#) [categoria 2](#) [categoria 1](#)

Hola desde la vista index

Lorem ipsum dolor, sit amet consectetur adipisicing elit. Nemo ullam velit ea sapiente dolores! Iure a

Footer