

# Primeros pasos con Yupp PHP Framework

## Implementación de un blog simple



**Yupp PHP Framework v0.1.6.7**

Este documento explica los pasos necesarios para comenzar a desarrollar aplicaciones web utilizando Yupp PHP Framework a partir de la presentación paso a paso del desarrollo de un blog simple, donde los usuarios serán capaces de crear entradas en el blog y agregar comentarios a estas entradas. Es importante aclarar que no se hicieron consideraciones de seguridad o performance por razones de simpleza.

Puedes encontrar este ejemplo dentro del framework cuando lo descargas desde:

<http://code.google.com/p/yupp/downloads/list>

01/11/2009  
Pablo Pazos Gutiérrez  
[pablo.swp@gmail.com](mailto:pablo.swp@gmail.com)  
[www.SimpleWebPortal.net](http://www.SimpleWebPortal.net)

Pequeña introducción a Yupp PHP Framework .....	3
Yupp en pocas palabras: .....	3
Descargar, Instalar y Correr Yupp PHP Framework .....	4
Descargar: .....	4
Instalar: .....	4
Correr: .....	4
Diseño del sistema de blog .....	5
Diseñar el modelo de datos persistente .....	5
Comenzamos por implementar la clase Entrada: .....	6
Explicación del código de la clase Entrada: .....	6
Implementación de la clase EntradaBlog: .....	9
Explicación del código de la clase EntradaBlog .....	9
Relaciones múltiples con otras clases persistentes: .....	9
Implementación de la clase Comentario: .....	10
Implementación de la clase Usuario: .....	10
Explicación del código de la clase Usuario .....	11
Crear vistas .....	12
Flujo de páginas .....	12
Convenciones: .....	13
Vista: Listado de entradas .....	13
Vista: Detalle de entrada .....	15
Crear los controladores .....	17
Convenciones: .....	17
Explicación del código: .....	19

## Pequeña introducción a Yupp PHP Framework

Yupp es un marco de trabajo (Framework) de código abierto, orientado al desarrollo ágil de aplicaciones web 2.0 basadas en PHP5 (PHP 5.2.3 en adelante). Yupp está orientado a componentes que se pueden implementar, distribuir e instalar de forma sencilla (no más que un simple “copiar y pegar”).

Yupp está 100% Orientado a Objetos e implementa los patrones MVC (Model-View-Controller) y ORM (Object Relational Mapping).

El objetivo principal de Yupp es hacer que la programación de aplicaciones basadas en web sea sencilla, ordenada, productiva, rápida y divertida, creando una herramienta sólida, estable, probada y a la vez flexible para adaptarse a las necesidades de cada proyecto.

Sabemos que existen infinidad de frameworks que tienen similares funcionalidades, la diferencia está en la forma de trabajo que plantea cada uno de ellos. Creemos que Yupp plantea una forma de trabajo que se adapta mejor que otros frameworks a la manera en que los desarrollares trabajamos y creamos aplicaciones, por eso te invitamos a probar Yupp Framework y esperamos recibir comentarios sobre si satisface tus expectativas. Yupp está en pleno desarrollo, mejorando día a día, así que las críticas son muy bienvenidas para poder seguir mejorando y adaptarlo a las necesidades de nosotros: los programadores.

### ***Yupp en pocas palabras:***

- Poca (casi ninguna) configuración, basado en convenciones
- Orientado a componentes, reuso siempre en mente
- Validación automática de datos, el mejor amigo de los formularios
- ORM, nada de SQL para las consultas frecuentes
- Helpers, la creación de vistas nunca fue tan fácil
- I18n, sistemas multi-lenguaje desde el inicio
- Urls amigables, el mejor amigo del SEO
- MVC con POO, para ponerle fin al código espagueti

# Descargar, Instalar y Correr Yupp PHP Framework

## Descargar:

Las liberaciones de Yupp se publican aquí:

- <http://code.google.com/p/yupp/downloads/list>

Si no quieres esperar a las liberaciones, la última versión de implementación siempre estará disponible por SVN:

- <http://code.google.com/p/yupp/source/checkout>

## Instalar:

Si has descargado el archivo ZIP con la liberación, primero debes descomprimirlo. Luego copia el código fuente (todos los archivos y directorios que tiene Yupp) en el directorio www o htdocs (depende del servidor que utilices y su configuración), puedes ponerlo bajo algún subdirectorio, por ejemplo www/yupp. Por ejemplo si usas WAMP la ruta al directorio será similar a esta: "C:\wamp\www". El directorio www o htdocs es el directorio donde está el contenido accesible desde la web mediante tu servidor web (por ejemplo Apache). Obviamente, debes tener algún servidor web que soporte PHP5 instalado y corriendo.

## Correr:

Accede al directorio donde has copiado Yupp desde un navegador, la dirección local para accederlo puede ser por ejemplo: <http://localhost:8080/yupp>, dependiendo de en que puerto tengas configurado el servidor web local y en que subdirectorio hayas copiado los archivos de Yupp (paso anterior).

Si ves una página similar a esta es que la instalación fue exitosa:

<b>Informacion del modelo</b> Muestra que tablas fueron generadas para el modelo y que tablas falta generar, y permite generar las tablas que falten.  Se generaron todas las tablas para el modelo. <ul style="list-style-type: none"><li>• Clase: <b>Comentario</b> se guarda en la tabla: <b>entradas</b> (CREADA)</li><li>• Clase: <b>Entrada</b> se guarda en la tabla: <b>entradas</b> (CREADA)</li><li>• Clase: <b>EntradaBlog</b> se guarda en la tabla: <b>entradas_blog</b> (CREADA)</li><li>• Clase: <b>Usuario</b> se guarda en la tabla: <b>usuarios</b> (CREADA)</li></ul>
<b>Componentes</b> Esta sección le permite ejecutar scripts de inicialización para los componentes del sistema. <ul style="list-style-type: none"><li>• <a href="#">blog Ejecutar Bootstrap</a></li></ul>
<b>Ingreso a los controladores</b> <b>blog:</b> <ul style="list-style-type: none"><li>• <a href="#">[ Comentario ]</a></li><li>• <a href="#">[ EntradaBlog ]</a></li><li>• <a href="#">[ Usuario ]</a></li></ul>
<b>Estadísticas</b> Algunas medidas del sistema. <ul style="list-style-type: none"><li>• <a href="#">Ver estadísticas</a></li></ul>

Como puedes apreciar, en la sección "Ingreso a controladores" están los controladores del sistema de blog que se describe en este documento, por lo que puedes acompañar la lectura con la prueba del sistema, también te invitamos a que recorras el código y veas que hay detrás de todo esto ☺

## Diseño del sistema de blog

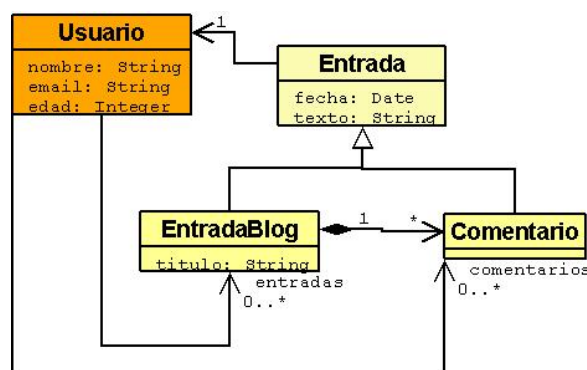
El objetivo es poder crear un sistema de blog que permita a los usuarios poder ingresar al sistema (login), poder crear entradas y poder comentar dichas entradas.

Los pasos a seguir son en cuanto al diseño de la aplicación son:

- Diseño del modelo de datos: que información se guardará en la base de datos.
- Diseño de las pantallas o “vistas”: conjunto de páginas y formularios con los que contará nuestra aplicación.
- Diseño de los controladores con la lógica necesaria para que el sistema funcione: ¿qué controladores necesitamos? ¿qué funcionalidad debe ofrecer cada controlador?

Una vez que tenemos diseñados esos tres elementos podemos pasar a la implementación utilizando Yupp PHP Framework. A continuación se detallan las tareas de diseño previas a la implementación, esto es independiente de Yupp PHP Framework, y aplica al desarrollo de cualquier aplicación que siga el patrón Model-View-Controller (MVC).

### Diseñar el modelo de datos persistente



El modelo de datos necesario para almacenar toda la información del blog consiste de una clase **Entrada** que representa cualquier tipo de entrada en el blog, la entrada puede ser una entrada del blog o un comentario a una entrada del blog. La clase **EntradaBlog** representa una entrada en el blog con su título y texto. La clase **Comentario** representa un comentario a una entrada del blog.

Las relaciones entre estas clases son tales que una entrada del blog puede tener muchos comentarios y un comentario pertenece a una única entrada del blog.

También está el **Usuario**, con sus datos, que es quien puede crear las entradas y los comentarios.

**Convención de Yupp Framework:** Todas las clases que implementan el modelo de datos de un componente se deben crear en el directorio “/model” del componente respectivo. Para el caso del blog, las clases implementadas se encuentran en la ruta “/components/blog/model”. Búscalas en la copia de Yupp que has descargado de nuestro sitio (<http://code.google.com/p/yupp/downloads/list>), y mira la implementación. Puedes agregarle o quitarle los atributos que creas necesario.

## Comenzamos por implementar la clase Entrada:

```
class Entrada extends PersistentObject {

    protected $withTable = "entradas";

    function __construct( $args = array($args = array (), $isSimpleInstance = false) )
    {
        $this->addAttribute("texto", Datatypes::TEXT);
        $this->addAttribute("fecha", Datatypes::DATETIME);

        // Valor por defecto es la fecha actual:
        $this->setFecha(date("Y-m-d H:i:s"));

        // Usuario que hizo la entrada:
        $this->addHasOne("usuario", Usuario);

        // Restricciones sobre campos para validación automática:
        $this->constraints = array(

            "texto" => array(
                Constraint::minLength(10), // Mas de 10 caracteres
                Constraint::maxLength(1000), // Menos de 1000 caracteres
                Constraint::blank(false) // No vacio
            ),
            "usuario" => array(
                Constraint::nullable(false)
            )
        );

        parent::__construct($args, $isSimpleInstance);
    }

    // Funciones que son necesarias implementar porque PHP5.2.X
    // no soporta referencias estáticas a la clase.
    //
    public static function listAll( $params ) {
        self::$thisClass = __CLASS__;
        return PersistentObject::listAll( $params );
    }
    public static function count() {
        self::$thisClass = __CLASS__;
        return PersistentObject::count();
    }
    public static function get( $id ) {
        self::$thisClass = __CLASS__;
        return PersistentObject::get( $id );
    }
    public static function findBy( Condition $condition, $params ) {
        self::$thisClass = __CLASS__;
        return PersistentObject::findBy( $condition, $params );
    }
    public static function countBy( Condition $condition ) {
        self::$thisClass = __CLASS__;
        return PersistentObject::countBy( $condition );
    }
}
```

## Explicación del código de la clase Entrada:

Lo primero que hay que observar es que toda clase persistente debe heredar de PersistentObject. Esto le permite utilizar todas las funcionalidades y ventajas del ORM implementado por Yupp (YORM).

```
class Entrada extends PersistentObject {
```

El atributo withTable indica en que tabla se almacenan las instancias de esta clase.

```
protected $withTable = "entradas";
```

## Campos de la clase:

```
$this->addAttribute("texto", Datatypes::TEXT);  
$this->addAttribute("fecha", Datatypes::DATETIME);
```

La clase Entrada tiene un campo texto de tipo TEXT y un campo fecha de tipo DATETIME. Si bien PHP es un lenguaje dinámico, donde no se declaran los tipos de las variables y campos, en Yupp es necesario especificar el tipo de cada campo para saber con que tipo de la base de datos se corresponde el campo, es parte del ORM.

## Valores por defecto para campos:

```
$this->setFecha(date("Y-m-d H:i:s"));
```

En este caso, cuando se crea una nueva instancia de Entrada, el campo fecha se establece automáticamente con el valor de la fecha actual con un formato determinado.

## Relaciones con otras clases persistentes:

```
$this->addHasOne("usuario", Usuario);
```

Aquí se define una relación con un Usuario, esto es para que la entrada tenga asociada al usuario que la creó.

## Restricciones sobre valores de campos:

```
$this->constraints = array(  
    "texto" => array(  
        Constraint::minLength(10), // Mas de 10 caracteres  
        Constraint::maxLength(1000), // Menos de 1000 caracteres  
        Constraint::blank(false) // No vacio  
    ),  
    "usuario" => array(  
        Constraint::nullable(false)  
    )  
);
```

Las restricciones son validadores automáticos de los datos que se asignan cada campo de la clase, esto permite:

- Verificar validez de los datos ingresados (por ejemplo desde un formulario).
- No almacenar datos inválidos en la base de datos.
- Mostrar mensajes de error para los datos inválidos, así el usuario podrá corregirlos.

Las restricciones se definen en el campo “constraints” (restricciones en inglés), esto es un array asociativo donde las claves son los nombres de campos de la clase, y los valores son arrays de las restricciones para cada campo. Esto permite definir varias restricciones para un mismo campo, pero no es necesario definir restricciones para todos los campos, o sea que puede haber campos sin restricciones.

La clase Constraint provee una serie de restricciones que aplican a distintos tipos de datos de campos, por ejemplo “minLength” y “maxLength” son restricciones para campos de tipo TEXT y no aplican a otro tipo de campo. Otro caso son las restricciones “max”, “min” o “between” que aplican a campos numéricos. Todos los tipos de restricciones están definidos en la clase Constraint, en el futuro agregaremos más restricciones y la posibilidad de declarar restricciones propias y que se verifiquen automáticamente.

### Llamada obligatoria al constructor de la superclase:

```
parent::__construct( $args );
```

Para que todo funcione bien y poder hacer procesamientos necesarios, al final de cada constructor de una clase persistente, se debe llamar al constructor de la superclase como se especifica arriba.

Por último se definen una serie de operaciones necesarias para simplificar el acceso a la interfaz de persistencia del objeto, esto es debido a un problema en PHP que está solucionado su versión 5.3.0. Este problema fue comentado en el blog:

<http://yuppframework.blogspot.com/2008/03/problemas-al-acceder-al-nombre-de-la.html>

Estas operaciones son iguales para todas las clases del modelo, por lo que solo se copian y pegan de una clase a otra, no es necesario escribirlas cada vez. En el futuro ni siquiera será necesario copiarlas.

```
public static function listAll( $params ) {
    self::$thisClass = __CLASS__;
    return PersistentObject::listAll( $params );
}

public static function count() {
    self::$thisClass = __CLASS__;
    return PersistentObject::count();
}

public static function get( $id ) {
    self::$thisClass = __CLASS__;
    return PersistentObject::get( $id );
}

public static function findBy( Condition $condition, $params ) {
    self::$thisClass = __CLASS__;
    return PersistentObject::findBy( $condition, $params );
}

public static function countBy( Condition $condition ) {
    self::$thisClass = __CLASS__;
    return PersistentObject::countBy( $condition );
}
```

## Implementación de la clase *EntradaBlog*:

```
<?php

// Es necesario importar esta clase aquí porque se hereda de ella.
YuppLoader::load( "blog.model", "Entrada" );

class EntradaBlog extends Entrada {

    protected $withTable = "entradas_blog";

    function __construct( $args = array(), $isSimpleInstance = false )
    {
        $this->addAttribute("titulo", Datatypes::TEXT);

        // La entrada del blog tiene varios comentarios.
        $this->addHasMany("comentarios", 'Comentario', PersistentObject::HASMANY_LIST);

        // Otra forma de definir restricciones, esta es por campo individual.
        $this->addConstraints("titulo", array (
            Constraint :: maxLength(24)
        ));

        parent::__construct( $args, $isSimpleInstance );
    }

    // Operaciones necesarias para simplificar el acceso a la interfaz de persistencia.
    // Fueron removidas para simplificar la lectura.
}

?>
```

## Explicación del código de la clase *EntradaBlog*

Esta clase hereda de **Entrada** que a su vez hereda de **PersistentObject**, por lo que no es necesario hacer que herede de **PersistentObject** directamente.

Notar que no se define el nombre de la tabla donde se persistirá la clase, esto es porque se hereda de la clase **Entrada** y esta ya tiene ese atributo definido. El resultado de esto que las instancias de la clase **EntradaBlog** serán almacenadas en la misma tabla que **Entrada**, ya que Yupp Framework PHP implementa el mapeo de herencia de una tabla.

A diferencia de otros frameworks similares, Yupp Framework PHP viene con soporte de persistencia de modelos con herencia sin necesidad de hacer modificaciones, simplemente alcanza con declarar la herencia como siempre en PHP 5. Sobre esto hay dos sabores: mapeo de herencia en una tabla y mapeo de herencia de múltiples tablas. Más info:

<http://yuppframework.blogspot.com/2008/08/orm-mapeo-de-herencia-multiples-tablas.html>

## Relaciones múltiples con otras clases persistentes:

```
$this->addHasMany("comentarios", 'Comentario', PersistentObject::HASMANY_LIST);
```

Cómo una entrada puede tener varios comentarios, se genera una relación “hasMany” (“tiene muchos” en inglés) con la clase persistente **Comentario**. Esto permite que para cada instancia de **EntradaBlog**, se tenga acceso a todos sus comentarios mediante la llamada al método dinámico **\$entrada->getComentarios()**, y esta invocación consulta la base de datos y devuelve todos los comentarios de la entrada, sin la necesidad de escribir SQL.

## Ejemplos de métodos dinámicos:

- para acceder al texto de una entrada se hace mediante `$entrada->getTexto()`
- para acceder a un atributo hasOne, por ejemplo la entrada del comentario se hace mediante `$comentario->getEntrada()`
- para acceder a un atributo hasMany como los comentarios de una entrada se hace mediante `$entrada->getComentarios()`

## Implementación de la clase Comentario:

La clase **Comentario** hereda de **Entrada** y tiene una **EntradaBlog** asociada, esta asociación modela que el **Comentario** es un comentario de una entrada del blog.

```
<?php

// Es necesario importar esta clase aqui.
YuppLoader::load( "blog.model", "Entrada" );

class Comentario extends Entrada {

    function __construct( $args = array(), $isSimpleInstance = false )
    {
        $this->addHasOne("entrada", 'EntradaBlog');

        $this->constraints = array(
            "entrada" => array(
                Constraint::nullable(false)
            )
        );

        parent::__construct( $args, $isSimpleInstance );
    }

    // Operaciones necesarias para simplificar el acceso a la interfaz de persistencia.
    // Fueron removidas para simplificar la lectura.
}

?>
```

## Implementación de la clase Usuario:

La clase **Usuario** modela a la persona que puede ingresar al blog mediante su “login” para crear y modificar entradas y comentarios. Además de poder crear, modificar y eliminar otros usuarios.

El “login” del usuario para ingresar el blog es la dupla email/clave.

```
<?php

class Usuario extends PersistentObject
{
    function __construct($args = array (), $isSimpleInstance = false)
    {
        $this->withTable = "usuarios";

        $this->addAttribute("nombre", Datatypes :: TEXT);
        $this->addAttribute("email", Datatypes :: TEXT);
        $this->addAttribute("clave", Datatypes :: TEXT);
        $this->addAttribute("edad", Datatypes :: INT_NUMBER);
        $this->addAttribute("fechaNacimiento", Datatypes :: DATE);

        $this->addHasMany("comentarios", 'Comentario');
        $this->addHasMany("entradas", 'EntradaBlog');
    }
}
```

```

$this->constraints = array (
    "nombre" => array (
        Constraint :: minLength(1),
        Constraint :: maxLength(30),
        Constraint :: blank(false)
    ),
    "clave" => array (
        Constraint :: minLength(5)
    ),
    "edad" => array (
        Constraint :: between(10, 100)
    ),
    "email" => array (
        Constraint :: email()
    )
);

parent :: __construct($args, $isSimpleInstance);
}

// Operaciones necesarias para simplificar el acceso a la interfaz de persistencia.
// Fueron removidas para simplificar la lectura.
}

?>

```

## Explicación del código de la clase Usuario

Al igual que en las clases vistas previamente, para **Usuario** se crean un conjunto de atributos, algunas relaciones con otras clases (**Comentario** y **EntradaBlog**) y un conjunto de restricciones sobre los valores de los campos del **Usuario**.

Algunas restricciones no vistas anteriormente son:

- **between**: sirve para validar que un número esté entre 2 números dados.
- **email**: sirva para validar que un string tiene la forma de un email válido.

## Crear vistas

Las vistas son las “páginas web” que tendrá nuestro sistema. Las llamamos “vistas” por el componente “view” del patrón de diseño “Model-View-Controller”, en donde las vistas son la presentación al usuario de los datos presentes en una base de datos (este sería el “Modelo” persistente que vimos antes).

Las vistas que tendremos para el sistema de blog son:

- **Listado de entradas:** lista de entradas con paginación (\*).
- **Detalle de entradas:** detalle de entrada con listado de los comentarios de la entrada.
- **Creación de entrada:** donde se ingresan los datos para crear una nueva entrada.
- **Edición de entrada:** similar a la de creación, pero para una entrada que ya existe.
- **Creación de comentario:** donde se ingresan los datos para crear un nuevo comentario sobre una entrada.
- **Login:** donde una persona ingresa su email y una clave e ingresa al sistema para crear entradas y comentarios.

(\*) La paginación es la capacidad de tener muchos registros en la base de datos pero pedir de a pocos (digamos 10) y verlos en forma de páginas, en las cuales si hacemos “siguiente” o “anterior” podemos navegar entre todos los registros de la base, ahorrando tiempo de carga y simplificando la presentación al usuario. Yupp PHP Framework incluye esta capacidad sin la necesidad de escribir una sola línea de código.

## Flujo de páginas

Este diagrama determina el flujo de páginas que se quiere implementar:

- los estados serían las páginas y se corresponden a las páginas mencionadas arriba.
- las transiciones son las acciones que ejecuta el usuario, pueden ser links o envíos de formularios.
- se omitió la página de login.

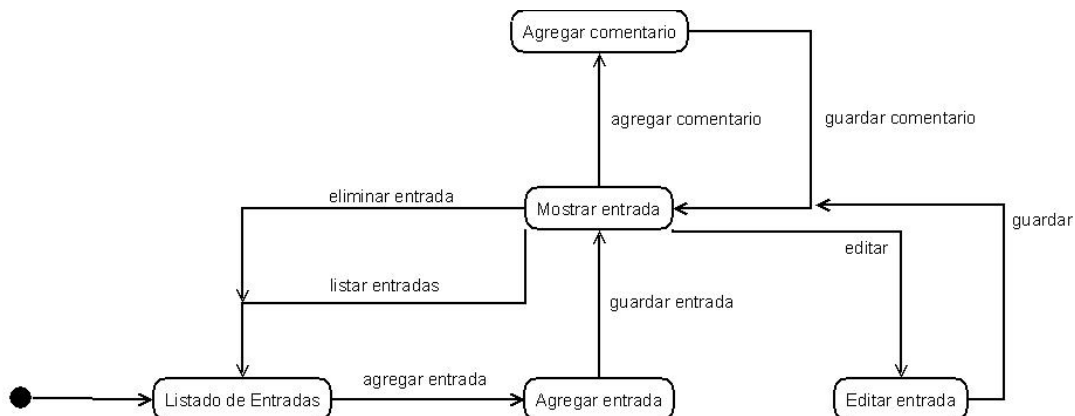


Figura X: Flujo de páginas del blog

## Convenciones:

Todas las vistas serán ubicadas en el directorio “components/\$component\$/views/\$controller\$”, donde “\$component\$” es el nombre del componente (en este caso es “blog”), y \$controller\$ es el nombre del controlador al que queremos acceder, en este caso podrá ser uno de “comentario”, “entradaBlog” o “usuario”, que son los dos controladores que tendrá nuestro sistema de blog.

Los nombres de las vistas deben seguir la convención de nombres, una vista debe llamarse “\$nombre\$.view.php”, donde \$nombre\$ puede ser cualquier nombre, pero se sugiere usar nombres como “list” para los listados, “create” para la vista de creación, “edit” para la de edición y “show” para la de detalles. Esto ayuda a tener consistencia entre los nombres de vistas de distintos controladores. Además puede ahorrar el tener que escribir código, ya que las acciones más comunes (CRUD) ya son implementadas por Yupp PHP Framework, es más, para un sistema sencillo de altas/bajas/modificaciones no se necesitan implementar las vistas.

## Vista: Listado de entradas

Ubicado en “components/blog/views/entradaBlog/list.view.php”.

Este código obtiene el modelo provisto por el controlador, para mostrar esta página. El modelo necesario es el listado de las entradas. La segunda línea incluye el archivo con el texto i18n para las páginas del blog, donde se definen los textos en todos los idiomas disponibles para poder traducir las páginas de forma automática al cambiar de idioma.

```
<?php
$m = Model::getInstance();
YuppLoader::loadScript("components.blog", "Messages");
?>
```

Inclusión del “layout” que usa la vista, se usa para darle el mismo aspecto a todas las páginas.

```
<html>
  <layout name="blog" />
```

Un ejemplo de como se obtienen archivos CSS y Javascript externos mediante helpers.

```
<head>
  <?php echo h("css", array("name" => "niftyCorners" )); ?>
  <?php echo h("js", array("name" => "niftycube" )); ?>

  <script type="text/javascript">
window.onload=function(){
  Nifty("div.flash","transparent");
  Nifty("ul.postnav a","transparent");
}
</script>

  <?php echo h("css", array("name" => "main" )); ?>
</head>
<body>
```

El título se traduce automáticamente a los idiomas disponibles, mucho mejor que tener un texto fijo.

```
<h1><?php echo DisplayHelper::message("blog.entrada.list.title"); ?></h1>
```

El campo “flash” del modelo se usa para enviar mensajes y mostrarlos en la página, este código verifica si hay un mensaje, y si lo hay lo muestra. Por ejemplo, el flash es utilizado para mostrar mensajes como “Entrada creada con éxito” cuando se crea una entrada.

```
<?php if ($m->flash('message')) { ?>
<div class="flash"><?php echo $m->flash('message'); ?></div>
<?php } ?>
```

El menú contiene un link a la acción de crear una entrada. Aquí se utiliza el helper “link” que genera el link a partir de la información que se le pasa, en este caso la acción “create” y el texto internacionalizable “add entry”.

```
<ul class="postnav">
<li>
<?php echo h("link", array("action" => "create",
                           "body" => DisplayHelper :: message(
                               "blog.entrada.list.action.addEntry"
                           )
                           )
);
?>
</li>
</ul><br/><br/>
```

Aquí se muestra el listado de las entradas, se accede a la lista provista por el modelo, que contiene cierto número de entradas que se muestran con cierto formato. En este caso, el formato con el que se muestra la entrada está definido en un template que es un simple script que tiene una determinada estructura html y accede a los campos de la entrada que se le pasa como parámetro para mostrarlos: “components/blog/views/entradaBlog/details.template.php”.

```
<?php
foreach ( $m->get('list') as $obj )
{
    Helpers::template( array("controller" => "entradaBlog",
                           "name"       => "details",
                           "args"      => array("entrada" => $obj)) );
}
?>
```

Este código corresponde al helper “pager” que muestra un pequeño paginador con links a siguiente y anterior, mostrando en que página se está actualmente.

```
<?php echo h('pager', array('offset'=>$m->get('offset'),
                           'max'   =>$m->get('max'),
                           'count' =>$m->get('count')) ); ?>

</body>
</html>
```

Las demás vistas contienen elementos similares a esta, pero veremos una más que es interesante.

## Vista: Detalle de entrada

Ubicado en: "components/blog/views/entradaBlog/show.view.php".

En esta vista se muestra una entrada y todos sus comentarios, repasemos el código:

Se accede al modelo como en la vista de listado, y se incluyen los mensajes i18n.

```
<?php
$m = Model::getInstance();
YuppLoader::loadScript("components.blog", "Messages");
?>
```

Algo de estilo para que se vea mejor.

```
<html>
  <head>
    <style type="text/css">
      ul.postnav, ul.postnav li {
        margin: 0px; padding: 0px; list-style-type:none; }
      ul.postnav li {
        float:left; margin-right: 5px; *width: 150px; }
      ul.postnav a {
        display:block; padding: 5px 10px 5px 10px;
        background: #C7FF5A; color: #666;
        text-decoration:none; text-align:center; }
      ul.postnav a:hover { background: #A8E52F; color:#FFF; }
    </style>

    <?php echo h("css", array("name" => "niftyCorners") ); ?>
    <?php echo h("js", array("name" => "niftycube") ); ?>
    <?php echo h("js", array("name" => "prototype-1.6.0.2") ); ?>

    <script type="text/javascript">
      window.onload=function(){
        Nifty("div.flash","transparent");
        Nifty("ul.postnav a","transparent");
      }
    </script>

    <?php echo h("css", array("name" => "main") ); ?>

  </head>
  <body>
```

En lugar de mostrar un título con un texto fijo, mostramos un título que se traduce automáticamente a distintos idiomas.

```
<h1><?php echo DisplayHelper::message("blog.entrada.show.title"); ?></h1>
```

Mensaje en el flash, se muestra si existe.

```
<?php if ($m->flash('message')) { ?>
  <div class="flash"><?php echo $m->flash('message'); ?></div>
<?php } ?>
```

Declaramos variables que se van a utilizar luego como la entrada que quiero mostrar y su identificador.

```
<?php $obj = $m->get('object'); ?>
```

En el menú se declaran cuatro acciones posibles, volver al listado, crear un comentario para la entrada, editar la entrada y eliminar la entrada.

```
<ul class="postnav">
  <li>
    <?php echo Helpers::link( array("controller" => "entradaBlog",
                                   "action"       => "list",
                                   "body"        => DisplayHelper::message(
                                                "blog.entrada.action.list"
                                              )) ); ?></li>

  <li>
    <?php echo Helpers::link( array("controller" => "comentario",
                                   "action"       => "create",
                                   "id"          => $obj->getId(),
                                   "body"        => DisplayHelper::message(
                                                "blog.entrada.action.addComment"
                                              )) ); ?></li>

  <li>
    <?php echo Helpers::link( array("action"     => "edit",
                                   "id"          => $obj->getId(),
                                   "body"        => DisplayHelper::message(
                                                "blog.entrada.action.edit"
                                              )) ); ?></li>

  <li>
    <?php echo Helpers::link( array("action"     => "delete",
                                   "id"          => $obj->getId(),
                                   "body"        => DisplayHelper::message(
                                                "blog.entrada.action.delete"
                                              )) ); ?></li>

</ul><br/><br/>
```

Aquí se muestra la entrada igual que como se mostraba en el listado, aquí usamos de nuevo el template que usamos en el listado ya que se muestra la entrada con el mismo formato que ahí.

```
<?php echo Helpers::template( array("controller" => "entradaBlog",
                                   "name"         => "details",
                                   "args"         => array("entrada" => $obj)) ); ?>
```

Aquí se listan los comentarios de la entrada. Recordar el atributo que tenía los comentarios de la entrada se llama “comentarios”, con `getComentarios()` se obtienen los comentarios de la entrada:

```
<?php $i = 1; ?>
<?php foreach ( $obj->getComentarios() as $com ) : ?>
  <div class="entrada">
    <div class="top">
      <div class="left">
        <?php echo DisplayHelper::message("blog.entrada.label.comment"); ?>
        # <?php echo $i; ?>
      </div>
      <div class="right">
        <?php echo $com->getFecha(); ?>
      </div>
    </div>
    <br/>
    <div class="content">
      <?php echo $com->getTexto(); ?>
    </div>
  </div>
  <?php $i++; ?>
<?php endforeach; ?>
</body>
</html>
```

Como las demás vistas son similares y más simples no las repasaremos. Puedes verlas directamente en el código, y si tienes alguna duda me envías un mail.

## Crear los controladores

El ejemplo del sistema de blog tiene dos controladores, el principal es el controlador de entradas del blog, llamado **EntradaBlogController**, que contiene operaciones que actúan sobre la clase del modelo **EntradaBlog**, esta es la convención de nombres para los controladores, o sea, el nombre de la clase del modelo sobre la cual trabajan seguido de “Controller”. El segundo controlador es el que se encarga de los comentarios y se llama **ComentarioController**. De **ComentarioController** solo se utiliza la acción de crear un comentario para una entrada.

### Convenciones:

Los controladores de un componente van en el directorio: “/components/\$component\$/controllers”, donde “\$component\$” es el nombre del componente, en este caso “blog”. Como los nombres de los archivos incluyen la ruta donde están ubicados, el archivo que contiene al **EntradaBlogController** se llama: “components.blog.controllers.EntradaBlogController.class.php”, no olvidar el “.class”.

### Código del controlador **EntradaBlog**

```
<?php
class EntradaBlogController extends YuppController {

    public function indexAction()
    {
        $loggedUser = YuppSession::get("user"); // Lo pone en session en el login.
        if ($loggedUser !== NULL)
            return $this->listAction();
        else
            return $this->redirect(array('controller'=>'usuario', 'action'=>'login'));
    }

    /**
     * Mostrar lista de elementos de alguna clase.
     */
    public function listAction()
    {
        if ( !isset($this->params['max']) ) // paginacion
        {
            $this->params['max'] = 5;
            $this->params['offset'] = 0;
        }

        $list = EntradaBlog::listAll( $this->params );
        $this->params['list'] = $list;
        $count = EntradaBlog::count();
        $this->params['count'] = $count; // Maximo valor para el paginador.
        return;
    }

    public function showAction()
    {
        $id = $this->params['id'];
        $obj = EntradaBlog::get( $id );
        $this->params['object'] = $obj;
        return;
    }

    public function getCommentsJSONAction()
    {
        $id = $this->params['id'];
        $entrada = EntradaBlog::get( $id );
        $comentarios = $entrada->getComentarios();

        $json = "";
    }
}
```

```

foreach ($comentarios as $comentario) {
    $json .= $comentario->toJSON() . ", ";
}
$json = substr($json, 0, -2);

sleep(1); // agregamos demora para ver como carga los comentarios por ajax

header('Content-type: application/json');
return $this->renderString( "{ 'comentarios':[ $json ]}" );
}

public function editAction()
{
    $id = $this->params['id'];
    $obj = EntradaBlog::get( $id );
    $this->params['object'] = $obj;
    return;
}

public function saveAction()
{
    $id = $this->params['id'];
    $obj = EntradaBlog::get( $id );
    $obj->setProperties( $this->params );

    if ( !$obj->save() ) // Con validacion de datos!
    {
        $this->params['object'] = $obj;
        return $this->render("edit");
    }

    $this->params['object'] = $obj;
    return $this->render("show");
}

public function deleteAction()
{
    $id = $this->params['id'];
    $ins = EntradaBlog::get( $id );
    $ins->delete(true); // Eliminacion logica.
    $this->flash['message'] = "Elemento [EntradaBlog:$id] eliminado.";
    return $this->redirect( array("action" => "list") );
}

public function createAction()
{
    $obj = new EntradaBlog();

    if (isset($this->params['doit'])) // create
    {
        $obj->setProperties( $this->params );
        if ( !$obj->save() ) // Con validacion de datos!
        {
            $this->params['object'] = $obj;
            return;
        }

        $this->flash['message'] = "Entrada creada con exito.";
        return $this->redirect( array( 'action' => 'show',
                                     'params' => array('id' => $obj->getId() ) ) );
    }

    $this->params['object'] = $obj;
    return;
}
}
?>

```

## Explicación del código:

Todo controlador debe heredar de **YuppController**.

```
class EntradaBlogController extends YuppController {
```

Acción que se llama por defecto cuando no se pasa una acción, por ejemplo si se accede a la url “/blog/entradaBlog”, donde especifica el controlador pero no la acción. En este caso, indexAction verifica si el usuario está logueado y si no lo está, muestra una página de login.

```
public function indexAction()
{
    $loggedUser = YuppSession::get("user"); // Lo pone en session en el login.
    if ($loggedUser !== NULL)
        return $this->listAction();
    else
        return $this->redirect(array('controller'=>'usuario', 'action'=>'login'));
}
```

**Convención:** todos los nombres de las acciones terminan en “Action”.

La acción de listado, pide una lista de entradas del blog a la base de datos. Ese pedido es paginado, en este caso muestra de a 5 entradas por página. También cuenta la cantidad total de entradas, para ajustar el paginador y saber cuantas paginas hay en total.

Luego muestra la vista “list” que está en: “components/blog/views/entradaBlog/list.view.php”

```
public function listAction()
{
    if ( !isset($this->params['max']) ) // paginacion
    {
        $this->params['max'] = 5;
        $this->params['offset'] = 0;
    }

    $list = EntradaBlog::listAll( $this->params );
    $this->params['list'] = $list;
    $count = EntradaBlog::count();
    $this->params['count'] = $count; // Maximo valor para el paginador.

    return;
}
```

La acción de show sirve para mostrar los detalles de una entrada.

- Recibe como parámetro el identificador de la entrada que se quiere ver en detalle.
- Obtiene esa entrada de la base de datos.
- Se la pasa en el modelo a la vista “show”, donde se muestran los detalles de la entrada junto con los comentarios de la misma.

**Nota:** cuando se obtiene la entrada, sus comentarios no son cargados de la base, se cargan automáticamente cuando desde la vista se intenta acceder a dichos comentarios, o sea que la carga es perezosa, solo se carga la información a la que se accede directamente.

```
public function showAction()
{
    $id = $this->params['id'];
    $obj = EntradaBlog::get( $id );
    $this->params['object'] = $obj;
    return;
}
```

La acción para editar una entrada (edit) es similar a la de ver detalles (show):

- Recibe el identificador de la entrada que se quiere editar.
- Se carga la entrada correspondiente de la base de datos.
- Se le pasa como modelo a la vista “edit” dicha entrada.
- En la vista se muestra la información de la entrada para ser editada y salvada posteriormente.

```
public function editAction()
{
    $id = $this->params['id'];
    $obj = EntradaBlog::get( $id );
    $this->params['object'] = $obj;
    return;
}
```

La acción “save” es la que se llama para salvar la información modificada en la edición de la entrada:

- Recibe el identificador de la entrada que fue modificada.
- También recibe los valores de los campos de la entrada, que pudieron haber sido modificados.
- Todos los valores modificados se setean automáticamente usando la función “setProperties()”.
- Luego se salva la entrada con los nuevos valores.
- Por último se muestra la vista “show” para ver los nuevos datos de la entrada.

```
public function saveAction()
{
    $id = $this->params['id'];
    $obj = EntradaBlog::get( $id );
    $obj->setProperties( $this->params );
    $obj->save();

    // show
    $this->params['object'] = $obj;
    return $this->render( "show" );
}
```

La acción “delete” sirve para eliminar una entrada de forma lógica.

En este caso la acción retorna “redirect” en lugar de “render”, esto quiere decir que no va a mostrar directamente una vista, si no que va a ejecutar otra acción del controlador. En este caso se ejecuta la acción “list” para ver el listado de entradas existentes, y podemos ver que la entrada eliminada no aparece.

```
public function deleteAction()
{
    $id = $this->params['id'];
    $ins = EntradaBlog::get( $id );
    $ins->delete();

    $this->flash['message'] = "Elemento [EntradaBlog:$id] eliminado.";
    return $this->redirect( array( "action" => "list" ) );
}
```

**Nota:** la redirección genera un nuevo Request HTTP al servidor a determinada url con determinados parámetros.

La acción “create” se utiliza para crear una nueva entrada. Cuando se ejecuta por primera vez muestra la vista “create” a la cual se le pasa como modelo una nueva instancia de **EntradaBlog**, para poder mostrar valores por defecto para algunos campos de la clase.

Luego si se hace un “submit” de la información ingresada para crear la nueva entrada, viene el parámetro “doit”, por lo que se ejecuta el código dentro del “if”. Como antes, setea los campos de la nueva entrada con los valores submiteados, luego intenta salvar mediante la operación “save()” de **PersistentObject**, la cual si hay algún error retorna “false” generando campos de error dentro de la instancia. Observar que la validación de datos se hace automáticamente al ejecutar “save()”. Si hubo errores se vuelve a la vista de “create” con la instancia que tiene los errores generados y en la vista se muestran los errores y los valores ingresados anteriormente. Si no hubo errores, se pone un mensaje en “flash” que será mostrado en la vista “show”, y se le pasa, como modelo, a dicha vista la nueva entrada.

```
public function createAction()
{
    $obj = new EntradaBlog();

    if ( isset($this->params['doit']) )
    {
        $obj->setProperties( $this->params );
        if ( !$obj->save() ) // Con validacion de datos!
        {
            $this->params['object'] = $obj;
            return;
        }

        $this->flash['message'] = "Entrada creada con exito.";

        return $this->redirect( array( 'action' => 'show',
                                     'params' => array( 'id' => $obj->getId() ) ) );
    }

    $this->params['object'] = $obj;
    return;
}
```

Ahora el código de **ComentarioController**. Solo vamos a ver la acción que se usa, las demás acciones, aunque declaradas, no son ejecutadas en esta demo.

```
<?php
class ComentarioController extends Controller {
    ...
}
```

Esta acción es muy similar a la de “create” del **EntradaBlogController**, lo único distinto es que se le pasa el id de la entrada para la cual se crea el comentario, se pide a la base de datos esa entrada, se le agrega el comentario a sus comentarios, luego al comentario se le setea la entrada y los demás atributos mediante “setProperties()”, el resto es todo igual, se verifican errores, si tiene errores vuelve a la misma vista y los muestra, y si no tiene errores, en este caso, muestra la vista “show”.

Observar que se puede desde un controlador mostrar una vista de otro controlador, esto se puede hacer mientras ambos controladores pertenezcan al mismo componente. Otra forma de hacerlo es redirigiendo a la acción “show” de **EntradaBlogController** pasándole el id de la entrada.

```
public function createAction()
{
    $obj = new Comentario();

    if ( isset($this->params['doit']) )
    {
        // entrada que se esta comentando.
    }
}
```

```

$entrada = EntradaBlog::get( $this->params['id'] );
$obj->setEntrada( $entrada );
$obj->setProperties( $this->params );

if ( !$obj->validate() ) // Validacion de datos!
{
    $this->params['object'] = $obj;
    return;
}

$entrada->addToComentarios( $obj );

$obj->setProperties( $this->params );
if ( !$entrada->save() ) // Salva comentarios en cascada
{
    $this->flash['Hubo un problema al actualizar la entrada'];
    $this->params['object'] = $obj;
    return;
}

$this->flash['message'] = "Comentario creado con exito.";

// show de la entrada con el nuevo comentario
return $this->redirect( array( "controller" => "entradaBlog",
                             "action" => "show",
                             "params" => array("id" => $entrada->getId())
                           ) );
}

$this->params['object'] = $obj;
return;
}
}
?>

```

01/11/2009  
Pablo Pazos Gutiérrez  
[pablo.swp@gmail.com](mailto:pablo.swp@gmail.com)  
[www.SimpleWebPortal.net](http://www.SimpleWebPortal.net)