# Introducción

El siguiente documento es un tutorial con los puntos más destacados del framework de PHP escogido para trabajar- Yii. Está desarrollado por miembors del grupo y cuenta con doble finalidad: la primera para introducir los puntos más importantes del modelo de programación que se seguirá en la implementación de Jugador número 12; y también pretende servir como manual de referencia de las características principales de Yii.

Nota: Todos los esquemas y dibujos utilizados son completamente originales hechos expreso para el documento.

# Índice

[1. Funcionamiento del MVC](#h.93wrjpywdm2d)

[1.1. CRUD](#h.onlui8a5c8s2)

[1.2. Rutas](#h.f6skvq2i8cnc)

[2. Estructura de directorios de un proyecto Yii](#h.o4uziuk8af22)

[2.1. Archivos de configuración](#h.oredb41rbd1y)

[2.2. Controladores](#h.29lujo9bn3q2)

[2.3. Modelos](#h.bwt2e7azz5lr)

[2.4. Vistas](#h.vte0f1ycye5a)

[2.4.1. Cómo funcionan los layouts](#h.mmk99e867nsm)

[2.5. Uniéndolo todo](#h.g3kvt3kfvb2f)

[2.5.1 Renderizar una vista](#h.ztemhhsvgww)

[2.5.2 Cargar un modelo desde un controlador](#h.puck4fy7kfa9)

[2.5.3 Guardar en la base de datos](#h.fm9bnscrph43)

[3. Gestión de usuarios](#h.ot21x4uykmdm)

[3.1. El módulo de usuarios de Yii](#h.ncch1kyvooly)

[3.1.1 Clase UserIdentity](#h.ihhsxw2hsat8)

[3.1.2 Clase LoginForm](#h.6pt827uuh2dt)

[3.2 Posibles ampliaciones](#h.br1fs31usy6p)

[4. Formularios](#h.ht1zz493n92l)

[4.1 Widget CActiveForm](#h.lmw3zgvt50ks)

## 

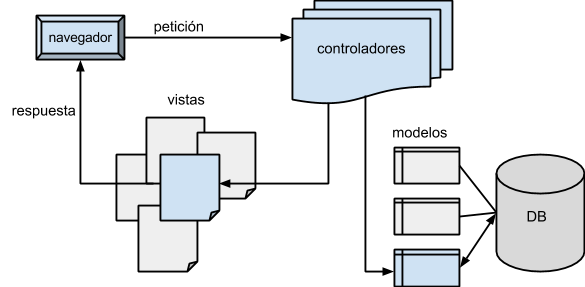
## 1. Funcionamiento del MVC

Modelo-Vista-Controlador (MVC) es un modelo de desarrollo software que separa los datos de una aplicación, la interfaz de usuario, y la lógica que gestiona los recursos en tres componentes.

1. El **modelo** es el sistema de gestión de la base de datos; se encarga de su lectura y escritura así como de validar los datos.

2. La **vista** es la página HTML que se genera como respuesta a la petición del usuario

3. El **controlador** es el responsable de coordinar los elementos; procesa las peticiones buscando y tratando los datos pedidos y renderiza las vistas para construir la respuesta final.



En una aplicación tendremos 1 controlador, 1 modelo y un conjunto de vistas para cada recurso.

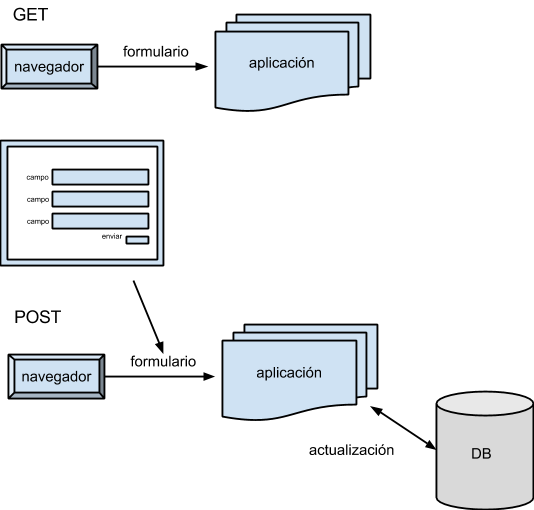
### 1.1. CRUD

Para cualquier recurso existen cuatro funciones o verbos básicos: Create, Read, Update y Delete.

A lo largo de esta guía los ejemplos estarán fijados a <Posts> de un blog ficticio; por lo tanto vamos a hablar de crear nuevos posts; pedir a la aplicación posts para leerlos; modificar un post en concreto o borrar un post existente.

**Create:** Para crear una nueva entrada en el blog, necesitamos rellenar primero los campos necesarios (por ejemplo asunto y contenido) de un formulario. Este formulario lo pediremos por GET y una vez relleno lo enviaremos a la aplicación por POST para que se cree y se guarde el nuevo post en la tabla correspondiente de la base de datos.

La secuencia de llamadas sería:



**Read**: En este caso sólo hacemos una petición GET de un post en concreto. La aplicación accederá a la base de datos para mostrarnos el dato pedido.

**Update**: La secuencia de llamadas es similar que para crear: primero necesitamos por GET el formulario para modificar un post y después mandaremos nuestra petición por POST

**Delete**: Accederemos (GET) al post concreto y mandaremos la petición de borrar (POST) para eliminarlo de la base de datos.

### 1.2. Rutas

Por defecto nuestras rutas serán:

|  |
| --- |
| **raíz/controlador/acción/primer\_parámetro/segundo\_parámetro/…/ultimo\_parametro** |
|  |

Si nuestro controlador para Post define las siguientes acciones:

|  |
| --- |
| **public function actionIndex**  **public function actionCreate**  **public function actionView($id)**  **public function actionUpdate($id)**  **public function actionDelete($id)** |

Tendremos las siguientes rutas disponibles

|  |
| --- |
| **raíz/posts/index**  **raíz/posts/create**  **raíz/posts/view/{id}**  **raíz/posts/update/{id}**  **raíz/posts/delete/{id}** |

## 2. Estructura de directorios de un proyecto Yii

Un proyecto de Yii tiene la siguiente estructura de ficheros y directorios (remarcados los archivos)

Proyecto/

.htaccess: Archivo de configuración del Xampp

index.php: <<Main>> del proyecto. Define el framework que se está usando y lanza la aplicación

assets/

**css/** Aquí se colocarán las hojas de estilo de la aplicación

main.css: archivo que importará la jerarquía de archivos .css

images/ Carpeta para las imágenes estáticas

**protected/** Carpeta principal del proyecto donde escribiremos el código

commands/

components/

Controller.php

UserIdentity.php

**config/** Carpeta donde estarán los archivos de configuración

main.php

**controllers/** Carpeta para los controladores (un archivo por recurso)

**data/** Carpeta para el esquema de la base de datos

schema.mysql.sql

extensions/

messages/

migrations/

**models/** Carpeta para los modelos (un archivo por recurso)

runtime/

**tests**/ Carpeta para los archivos de test

fixtures/

functional/

unit/

**views/** Carpeta para las vistas (una subcarpeta por recurso)

layouts/ Carpeta con los layouts (plantillas) del proyecto

site/ Carpeta para las vistas generales

pages/ Vistas .html estáticas

Además, como trabajaremos con el editor sublime, tendremos:

.sublime-works[pace](#h.o4uziuk8af22): Archivo de configuración de un proyecto en Sublime

.sublime-project: Archivo donde se registra la ruta a un proyecto de Sublime

### 2.1. Archivos de configuración

El archivo /protected/config/main.php define las variables globales de la aplicación, por ejemplo el nombre de la aplicación está definido en este archivo y podríamos acceder con

|  |
| --- |
| **yii::app()->name.** |

También se carga la configuración para conectar con la base de datos con el array <db> y se define cómo recoger los errores (excepciones)

|  |
| --- |
| **'errorHandler'=>array(**  **//use'site/error' action to display errors**  **'errorAction'=>'site/error',**  **),** |

### 2.2. Controladores

Los controladores se situarán todos en la carpeta protected/controllers y usarán todos el mismo modelo de nombrado, que es: ***NombreControlador*Controller.php** (importante que empiecen por mayúsculas, por ejemplo PostsController).   
Internamente, dichos archivos contendrán una clase llamada igual que el archivo (PostsController) y que hereda de “Controller”, clase que se encuentra en protected/components/. Esta clase contendrá diversas funciones (acciones) que recibirán datos, los procesarán (con ayuda de los modelos) y los enviarán a una de las vistas.  
Es obligatorio que estas funciones se llamen con el patrón ***action*Nombreacción([parametros])**. Por ejemplo, la página de index del controlador de los posts, que sería: “localhost/blog/posts/index”, haría referencia al controlador PostsController y a la función actionIndex.

A continuación queda un breve ejemplo de controlador:

|  |
| --- |
| **class PostsController extends Controller**  **{**  **public function actionIndex()**  **{**  **$dataProvider=new CActiveDataProvider('Posts');**  **$this->render('index',array(**  **'dataProvider'=>$dataProvider,**  **));**  **}**  **}** |

En protected/controllers nos encontramos con un controlador especial: SiteController.php  
Esta clase contiene funciones de uso general (sin ser de páginas específicas) como puede ser el login, logout, el formulario de contacto o la gestión de errores.

Debemos mencionar también diversas funciones de gran utilidad respecto a los controladores como por ejemplo:

* + filters()  
    Permite especificar una serie de filtros sobre dicho controlador. Se compone de un array de filtros, de los cuales resaltaremos accessControl que sirve para controlar el acceso a un controlador por parte de los usuarios.
  + accessRules()  
    Esta función será referida por el filtro accessControl y contiene una serie de reglas (allow, deny, etc) en forma de arrays que se evalúan en orden acerca de los permisos de usuarios.
    - Un usuario cualquiera se representará por el símbolo <<\*>>;
    - Un usuario autentificado por <<@>>;
    - Un usuario visitante por <<?>>

A continuación vemos un ejemplo del uso de esta función en el que permitimos a todos los usuarios acceder al índice y vista de un post; y a un usuario registrado a hacer cualquier acción sobre los posts.  
Nota: es importante dejar la última parte del “deny” por si se nos olvidase cualquier permiso.

|  |
| --- |
| **public function accessRules()**  **{**  **return array(**  **array('allow', //allow all users to access 'index' and 'view' actions.**  **'actions'=>array('index','view'),**  **'users'=>array('\*'),**  **),**  **array('allow', //allow authenticated users to access all actions**  **'users'=>array('@'),**  **),**  **array('deny', //deny all users**  **'users'=>array('\*'),**  **),**  **);**  **}** |
|  |

### 2.3. Modelos

Los modelos proporcionan, como hemos visto antes, la parte de interacción con la base de datos. Al igual que los controladores tendrán un directorio y regla de nombrado fijos.  
En este caso, los modelos irán situados en la carpeta protected/models y recibirán el mismo nombre que el de la tabla a la que se refieren empezando por mayúsculas. Por ejemplo, para crear el modelo de la tabla “posts”, iremos a protected/models y crearemos el archivo Posts.php.

Internamente, dichos modelos contienen una clase que hereda de CActiveRecord. Lo primero que haremos en dichas clases es llamar a la constructora del padre con:

|  |
| --- |
| **public static function model($className = \_\_CLASS\_\_)**  **{**  **return parent::model($className);**  **}** |

Tras esto nos encontraremos con diversas funciones, entre las que mencionaremos las siguientes:

* + rules()   
    Nos proporcionará una serie de reglas para comprobar los diversos campos de la tabla antes de, por ejemplo, modificarla.
  + relations()

Permite crear relaciones (has\_many, many\_many, etc) de esta tabla con el resto. Esto agiliza y simplifica tomar datos relacionados a través de este modelo.

* + attributeLabels()  
    Permite dar un nombre de forma general a cada columna de la tabla. Por ejemplo, tenemos una columna llamada id\_post y queremos nombrarla en nuestra página de forma general como “Número de post”.

Tenemos que resaltar un tipo de modelo especial que hereda de la clase CFormModel llamado LoginForm.php y que se usa para el inicio de sesión de usuarios. Dicho modelo se encuentra en la carpeta protected/models y lo explicaremos con más detalle en la sección correspondiente.

### 2.4. Vistas

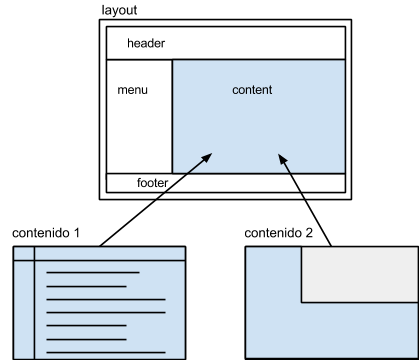
Las vistas se guardarán todas en el directorio protected/views. Para ello, se creará siempre una carpeta con el nombre del controlador asociado en minúsculas. Por ejemplo, para PostsController tendríamos una carpeta en views llamada posts/.  
Dentro de esta carpeta habrá una serie de archivos .php que contendrán las distintas vistas a renderizar por este controlador. Estas vistas se renderizarán en un layout que viene definido en el controlador.

#### 2.4.1. Cómo funcionan los layouts

Los layout son un mecanismo que emplearemos para evitar repetir código de las vistas.

Por norma general tendremos una misma estructura básica para todas las páginas .html de nuestra página; por ejemplo la siguiente:

|  |
| --- |
| **<div id=”container”>**  **<div id=”header”>**  **<!-- contenido para la cabecera -->**  **</div>**  **<div id=”menu”>**  **<!-- barra de menu izquierdo -->**  **</div>**  **<div id=”content”>**  **<!-- contenido principal de la página -->**  **</div>**  **<div id=”footer”>**  **<!-- contenido para el pie de página-->**  **</div>**  **</div>** |



Ahora bien, el contenido principal variará dependiendo de lo que queremos mostrar en cada página. Sin usar un layout, tendríamos que repetir el mismo código .html para cada vista. Pero por medio de este mecanismo podemos “montar” el contenido correspondiente sobre el esqueleto principal, cargando el código .html correspondiente dentro del div content.

¿Y como hacemos esto?

* Primero, decir que los layout se encuentran todos en el directorio protected/views/layouts.
* Definir el layout que emplearemos. Si entramos en la clase Controller.php veremos que se declara una variable $layout que contendrá la ruta de dicho layout.   
  Por defecto todos los controladores tomarán ese valor, pero puede ser redefinido en cualquiera de forma independiente.
* El código del layout del blog column1.php:

|  |
| --- |
| **<?php /\* @var $this Controller \*/ ?>**  **<?php $this->beginContent('//layouts/main'); ?>**  **<div id="content">**  **<?php echo $content; ?>**  **</div><!-- content -->**  **<?php $this->endContent(); ?>** |

Fijándonos solo en la sección del “div”, veremos que tan sólo se hace un echo de la variable $content. Esta variable hace referencia siempre al contenido del nivel inferior. Por ejemplo, desde PostsController llamamos a renderizar la vista index. Lo primero que se hará es tomar dicha vista y enviarla al archivo column1.php. Este archivo tomará la vista index y la añadirá por completo en la variable $content.

### 2.5. Uniéndolo todo

Daremos unos breves ejemplos sobre cómo interconectar todo esto:

#### 2.5.1 Renderizar una vista

Si desde un controlador queremos renderizar una vista, debemos hacer uso de la función

|  |
| --- |
| **$this->render(ruta,array\_de\_parametros);** |

Por ejemplo, para renderizar la vista create pasándole un modelo guardado con anterioridad, haremos algo como:

|  |
| --- |
| **$this->render('create',array( 'model'=>$model ));** |

#### 2.5.2 Cargar un modelo desde un controlador

Para cargar un modelo concreto desde un controlador:

|  |
| --- |
| **$model = Posts::model();** |

Por ejemplo, para tomar los datos de una tabla buscando una PK concreta usaremos:

|  |
| --- |
| **$model = Posts::model()->findByPk($id);** |

#### 2.5.3 Guardar en la base de datos

Una vez procesados los datos de un modelo desde el controlador, para guardarlo en la base de datos emplearemos la función save() suponiendo el modelo guardado en la variable $model

|  |
| --- |
| **$model->save();** |

Es importante comprobar siempre que se ha realizado correctamente viendo el valor devuelto por la función.

Si quisiésemos tomar los datos de una tabla completa, podríamos emplear clases de Yii como CActiveDataProvider. Por ejemplo, para obtener los datos de la tabla asociada a Posts, haremos algo como:

|  |
| --- |
| **$dataProvider = new CActiveDataProvider('Posts');** |

## 3. Gestión de usuarios

### 3.1. El módulo de usuarios de Yii

Por defecto Yii maneja un módulo de autenticación muy básico que puede ser ampliado posteriormente. Para ver la parte básica de su funcionamiento nos centraremos en dos clases:

#### 3.1.1 Clase UserIdentity

Esta clase se encuentra en el directorio protected/components y representa la información necesaria para representar un usuario. En primer lugar, debemos crear un atributo $\_id y sobreescribir la función getId() para que podamos determinar unívocamente de alguna forma un usuario frente a otro. Para ello, añadiríamos a la clase:

|  |
| --- |
| **private $\_id;**  **public function getId()**  **{**  **return $this->\_id;**  **}** |

El siguiente punto importante recae sobre la función authenticate() que se encargará de validar si los datos de un usuario son válidos antes de iniciar sesión. En nuestro caso comprobaremos en la base de datos si el nombre de usuario existe y si la contraseña es correcta, quedando el código:

|  |
| --- |
| **public function authenticate()**  **{**  **$user = Usuarios::model()->findByAttributes(array('nombre'=>$this->username));**  **if ($user === null)**  **{**  **$this->\_id = 'user Null';**  **$this->errorCode = self::ERROR\_USERNAME\_INVALID;**  **}**  **else if ($user->password !== $this->password)**  **{**  **$this->errorCode=self::ERROR\_PASSWORD\_INVALID;**  **}**  **else { // Okay!**  **$this->\_id = $this->username;**  **$this->setState('user',$user->nombre);**  **$this->errorCode = self::ERROR\_NONE;**  **}**  **return !$this->errorCode;**  **}** |
| Como podemos observar, primero creamos una consulta al modelo Usuarios para que nos dé las entradas que tengan como “nombre” el nombre de usuario (que ya veremos de dónde viene). A continuación comprobamos que el usuario es correcto y, en caso contrario, devolvemos los correspondientes errores.  Si las contraseñas coinciden, pasamos a autenticar a nuestro usuario en el sistema. Para ello, primero le asociamos un id a través de la variable $\_id creada anteriormente. En este caso, suponemos que “nombre” es una clave primaria en la tabla Usuarios y no está repetida, con lo que nos aseguramos que el id será único para cada usuario.  Acto seguido, pasamos a registrar una variable de sesión - a través de la función setState - para dicho usuario (array $\_SESSION en php) denominada ‘user’ y que tomará como valor el nombre de usuario obtenido del modelo. |

#### 3.1.2 Clase LoginForm

Esta clase representa la estructura para obtener los datos del formulario de login. Una vez obtenidos los datos, hace uso de dos funciones principales para realizar la autenticación:

* + authenticate()

|  |
| --- |
| **public function authenticate($attribute,$params)**  **{**  **if(!$this->hasErrors())**  **{**  **$this->\_identity=new UserIdentity($this->username,$this->password);**  **if(!$this->\_identity->authenticate())**  **$this->addError('password','Usuario o contraseña incorrecto.');**  **}**  **}** |

Esta función se encarga de comprobar, a través de la clase UserIdentity, si el usuario y contraseña introducidos son correctos. Se basa en la siguiente función.

* + rules()

|  |
| --- |
| **public function rules()**  **{**  **return array(**  **// username and password are required**  **array('username, password', 'required'),**  **// rememberMe needs to be a boolean**  **array('rememberMe', 'boolean'),**  **// password needs to be authenticated**  **array('password', 'authenticate'),**  **);**  **}** |

Este fragmento se encarga de declarar las reglas acerca de los campos de login. En este caso, vemos que usuario y contraseña son requeridos, el botón de “Recordarme” debe ser booleano y la contraseña además debe ser autentificada.

* + login()

|  |
| --- |
| **public function login()**  **{**  **if($this->\_identity===null)**  **{**  **$this->\_identity=new UserIdentity($this->username,$this->password);**  **$this->\_identity->authenticate();**  **}**  **if($this->\_identity->errorCode===UserIdentity::ERROR\_NONE)**  **{**  **$duration = $this->rememberMe ? 3600\*24\*30 : 0; // 30 days**  **Yii::app()->user->login($this->\_identity, $duration);**  **return true;**  **}**  **else**  **return false;**  **}** |

Por último, esta función se encarga de loguear al usuario en el sistema haciendo uso de Yii::app()->user->login($this->\_identity,$duration);. Es importante ver que devuelve un booleano indicando si se ha realizado la operación con éxito, por lo que deberemos capturarlo y actuar en consecuencia.

Una vez vistas estas dos clases gracias a las cuales podemos tener un usuario ya autenticado en el sistema, pasemos a ver un par de detalles útiles:

* Registrar variable de sesión: para ello, como hemos visto antes, hemos hecho uso de la función setState():

|  |
| --- |
| **$this->setState('nombreVariable',valor);** |

* Acceder a variable de sesión: para esto nos serviremos del módulo de usuarios de Yii, al que referenciaremos de la siguiente forma:

|  |
| --- |
| **Yii::app()->user->nombreVariable** |

### 3.2 Posibles ampliaciones

Yii nos ofrece diversas extensiones al módulo de usuarios muy útiles. Entre ellas podemos destacar dos bastante interesantes:

* CDbHttpSession: nos ofrece una forma diferente y más cercana a PHP de trabajo con sesiones, incluyendo la opción (automática) de mantenerlas en una base de datos. Otras de las opciones que aporta son la generación de correos de verificación, recuperación de contraseñas, etc.
* CDbAuthManager: una interesante ampliación al módulo de usuarios básico que añade la posibilidad de creación de un sistema de roles, tareas y operaciones. Para ello hace uso de una base de datos creada automáticamente en la que guarda el árbol de relación entre estas entidades. Ofrece también la posibilidad de crear “reglas de negocio” que servirán para aplicar los permisos bajo parámetros adicionales. Esto muy útil y sencillo de utilizar para detalles tan importantes como permitir que un usuario tan sólo pueda modificar sus propios datos.

## 4. Formularios

A la hora de trabajar con formularios se nos presentan dos opciones posibles:

* Podemos realizar los formularios al estilo tradicional, esto es, crear el formulario Html a mano, definir su archivo de procesado por PHP y su validación por javascript/ajax+PHP.   
  Esto nos permite enviar parámetros por GET o POST al controlador correspondiente y tomarlos allí haciendo uso de $\_GET y $\_POST.
* Otra opción es hacer uso del widget (extensión) CActiveForm, que explicaremos a continuación.

### 4.1 Widget CActiveForm

Esta extensión nos permite trabajar de forma muy sencilla con formularios estándar. De forma predeterminada permite crear un formulario asociado a un modelo concreto. Por ejemplo, pongámonos en el caso de la tabla Posts de la base de datos. Si generamos usando Gii el código para sus vistas, en la parte de “create” y “update” se hará uso de una vista denominada \_form.php que llama a este mismo widget. Este formulario incluye campos para todas las columnas de dicha tabla (Posts), junto con sus reglas de validación.

Para llamar al widget debemos darle una serie de parámetros:

|  |
| --- |
| **$form=$this->beginWidget('CActiveForm', array(**  **'id'=>'posts-form',**  **'enableAjaxValidation'=>false,**  **));** |

En primer lugar le daremos el nombre del widget (CActiveForm). Tras esto, dos parámetros necesarios para dicho widget: el “id” que se asociará al formulario html y la opción de habilitar la validación de los datos mediante Ajax.

Lo importante es que no sólo debemos adaptarnos al formulario predeterminado, sino que podemos lo podremos modificar fácilmente. Por ejemplo, imaginemos que necesitamos añadir la fecha a las entradas del blog. No queremos que el usuario pueda introducirla a través del formulario, por lo que eliminamos el textField y la sección correspondiente de la fecha. A continuación, pasamos al controlador, donde añadiremos la capacidad de añadir esos datos internos:

|  |
| --- |
| **public function actionCreate()**  **{**  **$model = new Posts();**  **// Uncomment the following line if AJAX validation is needed**  **// $this->performAjaxValidation($model);**  **if(isset($\_POST['Posts']))**  **{**  **$model->attributes = $\_POST['Posts'];**  **$model->fecha = time();**  **$model->usuario = Yii::app()->user->user;**  **if($model->save())**  **$this->redirect(array('view','id'=>$model->id));**  **}**  **$this->render('create',array(**  **'model'=>$model,**  **));**  **}** |

Como podemos ver, el widget pasa todos los campos del formulario en un array llamado igual que el modelo (Posts en este caso) empleando el método POST. Por ello, para comprobar que estos datos existan y emplearlos haremos uso de $\_POST[‘Posts’].

Acto seguido pasamos a añadir los datos del array directamente al modelo. Para ello nos servimos de la propiedad attributes del modelo que nos permite hacerlo directamente (toma los valores del array Posts y los añade al modelo).

Ahora, como lo que nos interesaba era modificar la fecha, tan solo debemos crear manualmente dicho campo en nuestro modelo y darle valor. Lo mismo hemos hecho en este caso con el nombre de usuario, que será tomado de la variable de sesión ‘user’.