Introducción a CakePHP

¿Qué es CakePHP?

CakePHP es un framework (entorno de trabajo) libre y de código abierto para el desarrollo en PHP. Es una estructura de librerías, clases y una infraestructura run-time (en tiempo de ejecucion) para programadores de aplicaciones web originalmente inspirado en el framework Ruby On Rails. Nuestra principal meta es permitir su trabajo de manera estructurada y rápida, sin perdida de flexibilidad

¿Por qué CakePHP?

CakePHP tiene varias características que lo hacen una gran opción como un framework para desarrollo de aplicaciones rápidas y con el menor costo de molestia. Aquí hay unas pocas en ningun orden en particular.

- Comunidad activa y amigable
- Licencia flexible
- Compatibilidad con PHP4 y PHP5
- CRUD integrado para la interacción con la base de datos y las preguntas simplificadas
- Scaffolding
- Arquitectura Modelo Vista Controlador (MVC)
- Despachador de peticiones con buena vista, URL personalizadas
- Validación incorporada
- Plantillas rápidas y flexibles (Sintáxis PHP, con Helpers)
- Helpers en Vistas para AJAX, Javascript, Formularios HTML y más
- Seguridad, Sesiones y Componentes para Manejo de Peticiones
- Lista de Control y Acceso flexible
- Desinfección de datos
- Caché flexible en Vistas
- Trabaja desde cualquier subdirectorio web del sitio, con poca o ninguna configuración de Apache envuelta

Historia de CakePHP

En 2005, Michal Tatarynowicz escribio una minima version de un Framework Rápido para Aplicaciones en PHP. El encontró que ese fue el principio de un muy buen framework. Michal publicó el framework bajo la licencia MIT, mezclando Cake, y abriendose a una comunidad de desarrolladores, quienes ahora mantienen Cake bajo el nombre CakePHP

Conceptos Básicos

Introducción

Este capitulo es una breve introducción al concepto **MVC** el cual ah sido implementado en **Cake**. Si eres nuevo en el **MVC** (Modelo Vista Controlador) este capitulo definitivamente es para ti. Vamos a iniciar con una discusión general de los conceptos **MVC**, trabajando a nuestra manera el **MVC** en **Cakephp** y mostrar algunos ejemplos simples de **Cakephp** usando el patrón de **MVC**.

El Patrón MVC

Modelo-Vista-Controlador es un patrón para diseño de software que ayuda a separar lógicamente el código haciéndolo reusable, mantenible y generalmente mejor. Modelo Vista Controlador fue descrito por primera vez por el autor de "the group Gang of Four", Dean Helman escribío:

```
    "El paradigma MVC es un camino para fragmentar la aplicación, o
tan solo una parte de la interfaz, en tres partes:
    el modelo, la vista y el controlador. Originalmente MVC fue
desarrollado para trazar la relación tradicional de la entrada,
procesamiento y salida en el ámbito de la GUI (Interfaz gráfica de
usuario).

    Entrada -> Procesamiento -> Salida

    Controller -> Model -> View

    La entrada del usuario, el modelado del mundo externo, y la
retroalimentación visual para el usuario son separados y dirigidos por
los objetos modelo, vista y controlador.

    El controlador interpreta las entradas del mouse y teclado desde
el usuario, y convierte sus acciones en comandos que son enviados
hacia el modelo y/o la vista para efectuar el cambio apropiado.

    El modelo maneja uno o más elementos de datos, responde a las
consultas acerca de su estado, y responde a las instrucciones para
cambiar de estado. La vista maneja un área rectangular del área que se
ve y es responsable de presentar los datos al usuario con una
combinación de gráficos y texto."
```

En terminos de **Cake**, El modelo (*"Model"*) representa una base de datos, tabla ó registro, así como sus relaciones con otras tablas ó registros. Los Modelos (*"Models"*) contienen las reglas para las validación de datos. Estas reglas son aplicadas cuando en el modelo (*"model"*) se insertan o actualizan datos. Las vistas en cake estan representadas por los archivos *"view"*, los cuales son archivos de HTML con código PHP incluido. Los controladores (*"controllers"*) de **Cake** manejan las peticiones del servidor. Toma las entradas del usuario (URL y datos de POST), aplica la logica de negocios, utilizar

los modelos (*"models"*) para leer y escribir en nase de datos y otras rutas, y manda la salida apropiada de datos con los archivos de vistas (*"view"*).

Para hacer tan fácil como sea posible la organización de tu aplicación, **Cake** usa este patron no solo para manejar como interactúan los objetos dentro de la aplicacion, si no también como son almacenados los archivo, lo cual se detalla a continuación.

Descripción de la disposición de archivos de Cake

Cuando desempaquete **Cake** en su servidor encontrará 3 carpetas principales.

```
app
cake
vendors
```

La carpeta **cake** es el lugar para las bibliotecas base de **Cake** y generalmente no necesitará tocarlo.

La carpeta **app** es el lugar donde estarán las carpetas y archivos específicos de su aplicación. La separación entre la carpeta **cake** y la carpeta **app** hace posible para usted tener muchas carpetas **app** compartiendo un solo conjunto de bibliotecas de Cake. Esto también facilita actualizar CakePHP: usted solo descargue la última versión de Cake y sobre-escriba sus bibliotecas base actuales. No necesita preocuparse por sobre-escribir algo que haya escrito para su aplicación.

Puede utilizar la carpeta **vendors** para guardar en ella bibliotecas de terceros. Aprenderá más acerca de esto más adelante, pero la idea básica es que se puede acceder a clases que ha colocado en la carpeta vendors utilizando la función **vendor()**.

Observe la disposición de archivos completa:

```
/app
    /config          - Contiene archivos de configuración de la base
de datos, ACL, etc.

    /controllers     - Los controladores van aquí
        /components  - Los componentes van aquí

    /index.php       - Le permite desplegar cake con /app como
DocumentRoot

    /models          - Los modelos van aquí

    /plugins         - Los Plugins van aquí

    /tmp             - Usado para cache y logs

    /vendors         - Contiene las bibliotecas de terceros para esta
aplicación

    /views           - Las vistas van aquí
        /elements    - Los elementos (pequeñas partes de vistas) van
aquí
        /errors      - Sus errores personalizados van aquí
```

```
        /helpers     - Los helpers van aquí
        /layouts     - Los layout de la aplicación van aquí
        /pages       - Las vistas estáticas van aquí.

    /webroot         - El documentroot de su aplicación
        /css
        /files
        /img
        /js

/cake                - Bibliotecas base de Cake. No edite ningún
archivo aquí.

index.php

/vendors             - Utilizado para las bibliotecas de terceros para
todo el servidor.

VERSION.txt          - Aquí se ve la versión que se esta utilizando.
```

Instalando CakePHP

Sección 1

Introducción

Ahora que conoce todo lo que tiene que saber de la estructura y propositode todas las librerias CakePHP, o se a saltado a esta parte por que no le preocupan esas cosas y solo quiere comenzar a jugar. De cualquier forma, está listo para ensuciar sus manos. Este capitulo describirá como debe ser instalado en el servidor, diferentes formas de configurar su servidor, descargando e instalando CakePHP, trayendole la pagina por defecto de CakePHP, y algunos datillos de localizaciones y resoluciones de problemas solo en caso de que todo no vaya como está planeado.

Sección2

Requerimientos

Para usar CakePHP usted debe primero tener un servidor que tenga las librerias y programas para correr CakePHP.

Requerimientos del Servidor

Estos son los requerimientos para la puesta en marcha del servidor para corre CakePHP:

1. Un servidor HTTP (como Apache) con lo siguiente habilitado: sesiones, mod_rewrite (no absolutamente necesario, pero preferido).
2. PHP 4.3.2 o superior. Sí CakePHP trabaja bien en PHP 4 o 5.
3. Un motor de base de datos (ahora, hay soporte para MySQL, PostgreSQL y un envoltorio para ADODB)

Sección 3

Instalando CakePHP

Obteniendo la versión estable más reciente

Hay diferentes opciones al obtener una copia de CakePHP: Obtener una versión estable desde CakeForge, adquirir un paquete que se crea cada noche, u obtener el código en su actualización más reciente desde SVN.

Para descargar el código de la versión estable, revise la sección de archivos de del proyecto CakePHP en CakeForge, entrando a la siguiente dirección:

http://cakeforge.org/projects/cakephp/.

Para adquirir la versión que se crea cada noche, descargela desde http://cakephp.org/downloads/index/nightly. Esta versión que se crea cada noche es estable, y en ocaciones incluye las correcciones a errores del código entre versiones estables.

Para obtener la actualización más reciente desde el repositorio SVN, utilice su cliente SVN favorito y conecte a https://svn.cakephp.org/repo/trunk/cake/ y después seleccione su versión.

Desempaquetando

Ahora que a descargado la versión más reciente, coloque el paquete comprimido dentro del webroot de su servidor web y desempaquetelo ahí. Hay dos opciones para hacer esto: utilizar una configuración de instalación para desarrollo, que le permite ver fácilmente muchas aplicaciones CakePHP bajo un mismo dominio, o utilizar la configuración de instalación para producción, la cual permite una sola aplicación CakePHP en el dominio.

Sección 4

Configuración de la instalación de CakePHP

La primer opción para configurar la instalación de CakePHP es recomendada únicamente para un ambiente de desarrollo, debido a que es menos segura. La segunda opción es considerada más segura y puede ser utilizada en un ambiente de producción.

NOTA: La carpeta /app/tmp debe tener permisos de escritura para el usuario con el que se ejecuta el servidor web.

Configuración de una instalación para desarrollo

Para desarrollar podemos colocar la carpeta de instalación de Cake completa dentro del DocumentRoot especificado de una manera como esta:

```
/wwwroot
    /cake
        /app
        /cake
        /vendors
        .htaccess
        index.php
```

En esta configuración de instalación, la carpeta wwwroot actúa como la raíz del servidor web, por ello sus URLs aparecerán de una manera como esta (si también se está utilizando mod_rewrite):

```
www.example.com/cake/controllerName/actionName/param1/param2
```

Configuración de una instalación para producción

Para utilizar una configuración de instalación para producción, usted necesita tener derechos para cambiar el DocumentRoot en su servidor. Haciendo esto, todo el dominio actúa como una sola aplicación CakePHP

La configuración de una instalación para producción utiliza la siguiente distribución de archivos:

```
..ruta_a_instalación_de_cake
    /app
        /config
        /controllers
        /models
        /plugins
        /tmp
        /vendors
        /views
        /webroot <-- Este debe ser su nuevo DocumentRoot
        .htaccess
        index.php
    /cake
    /vendors
    .htaccess
    index.php
```

Configuración de httpd.conf sugerida para producción

```
DocumentRoot /path_to_cake/app/webroot
```

En esta configuración el directorio webroot esta actuando como la raiz web, por ello sus URLs se pueden ver de una manera similar a esta (si esta utilizando mod_rewrite):

```
http://www.example.com/controllerName/actionName/param1/param2
```

Configuración Avanzada: Opciones alternativas de instalación

Hay algunos casos en los que usted podría querer colocar los directorios de Cake en diferentes lugares del disco. Por ejemplo, podría necesitarlo debido a alguna restricción en su servidor, o tal vez solo quiera que algunas de sus aplicaciones compartan las mismas bibliotecas de Cake.

Estas son tres partes principales en una aplicación Cake:

- Las bibliotecas básicas de CakePHP - Se encuentran en /cake
- El código de su aplicación (por ejemplo controllers, models, layouts y views) - Se encuentran en /app
- Los archivos webroot de su aplicación (por ejemplo imágenes, javascript y css) - Se encuentran en /app/webroot

Cada uno de estos directorios se puede localizar en cualquier lugar de nuestro sistema de archivos, excepto webroot, que necesita ser accesible por su servidor web. Usted también puede mover el directorio webroot fuera del directorio app, siempre que le diga a Cake donde lo ha puesto.

Para configurar su instalación de Cake, usted necesitará realizar algunos cambios en /app/webroot/index.php (al que es distribuido en Cake). Hay tres constantes que usted necesitará editar: ROOT, APP_DIR, and CAKE_CORE_INCLUDE_PATH.

- ROOT debe ser configurado con la ruta de la carpeta que contenga la carpeta app.
- APP_DIR debe ser configurado con la ruta de su directorio app.
- CAKE_CORE_INCLUDE_PATH debe ser configurado con la ruta de la carpeta que contiene sus bibliotecas de Cake.

/app/webroot/index.php (algunos comentarios fueron removidos)

```
if (!defined('ROOT'))
{
    define('ROOT', dirname(dirname(dirname(__FILE__))));
}

if (!defined('APP_DIR'))
{
    define ('APP_DIR', basename(dirname(dirname(__FILE__))));
}

if (!defined('CAKE_CORE_INCLUDE_PATH'))
{
    define('CAKE_CORE_INCLUDE_PATH', ROOT);
}
```

Un ejemplo le puede ayudar a entender esto mejor. Imagine que quiero configurar Cake para trabajar con la siguiente configuración:

- Quiero compartir las bibliotecas de Cake con otras aplicaciones, y colocarlas en /usr/lib/cake.
- Mi directorio webroot necesita estar en /var/www/mysite/.
- Los archivos de mi aplicación serán almacenados en /home/me/mysite.

```
La configuración de archivos seria como esta:

/home
    /me
        /mysite                  <-- Utilizado para ser
/cake_install/app
            /config
            /controllers
            /models
            /plugins
            /tmp
            /vendors
            /views
            index.php
/var
    /www
        /mysite                  <-- Utilizado para ser
/cake_install/app/webroot
            /css
            /files
            /img
            /js
```

```
            .htaccess
            css.php
            favicon.ico
            index.php
/usr
    /lib
        /cake                    <-- Utilizado para ser
/cake_install/cake
            /cake
                /config
                /docs
                /libs
                /scripts
                app_controller.php
                app_model.php
                basics.php
                bootstrap.php
                dispatcher.php
            /vendors
```

Teniendo este tipo de configuración, necesitaría editar mi archivo index.php en el webroot (que en este ejemplo sería /var/www/mysite/index.php) para que quedara como el siguiente:

Es recomendable utilizar la constante 'DS' en lugar de diagonales como separador para las rutas de archivos. Esto previene cualquier error de 'archivos faltantes' que usted pudiera obtener como resultado de utilizar los separadores equivocados, y esto hace su código más *portable*.

```
if (!defined('ROOT'))
{
    define('ROOT', DS.'home'.DS.'me');
}

if (!defined('APP_DIR'))
{
    define ('APP_DIR', 'mysite');
}

if (!defined('CAKE_CORE_INCLUDE_PATH'))
{
    define('CAKE_CORE_INCLUDE_PATH', DS.'usr'.DS.'lib'.DS.'cake');
}
```

Sección 5

Configuración de Apache y mod_rewrite

Mientras CakePHP fue desarrollado para utilizar mod_rewrite desde un inicio, nos hemos dado cuenta que a unos cuantos usuarios se les complica dejar todo funcionando correctamente en sus sistemas. Aquí tenemos unas cuantas cosas que tal vez debería de intentar para dejarlo funcionando correctamente:

- Asegúrese que la sobreescritura de .htaccess está permitida: en su httpd.conf, usted debe de tener una sección que define otra sección por cada carpeta en su

sistema. Asegúrese de que la carpeta correcta tiene asignado **AllowOverride** en **All** .

- Asegúrese de que está editando httpd.conf del sistema en lugar de algún archivo httpd.conf del usuario o sitio en específico
- Por una razón u otra, pudo haber obtenido una copia de CakePHP sin los archivos necesitados .htaccess. Esto ocurre algunas veces debido a que algunos sistemas operativos tratan a los archivos que inician con '.' como ocultos, y no los copian. Asegúrese de que su copa de CakePHP la ha adquirido de la sección de descargas del sitio o de nuestro repositorio SVN.
- ¡Asegúrese de que se está cargando mod_rewrite correctamente!. Debería de ver algo como **LoadModule rewrite_module libexec/httpd/mod_rewrite.so** y **AddModule mod_rewrite.c** en su httpd.conf.
- Si está instalando Cake dentro de una carpeta de usuario (http://ejemplo.com/~miusuario/), necesitará modificar el archivo .htaccess en su carpeta base de su instalación de Cake. Solo agregue la linea " Rewrite /~miusuario/ ".
- Si por alguna razón sus URLS contienen una molesta y muy larga ID de sesión (http://ejemplo.com/posts/?CAKEPHP=4kgj577sgabvnmhjgkdiuy1956if6ska), debería también agregar "php_flag session.trans_id off" al archivo .htaccess en la raíz de su instalación.

Sección 6

Asegurarse de que funciona

Muy bien, veamos este bebé en acción. Dpendiendo que tipo de instalación utilizó, debería de apuntar su navegador a http://www.ejemplo.com ó http://www.ejemplo.com/cake. En este punto, se le presentará el inicio por defecto de CakePHP, y un mensaje que le dice el estado actual de su conexión con la base de datos.

¡Felicidades! Usted está listo para crear su primera aplicación basada en Cake.

Configuración

Sección 1

Configuración de Base de datos

Tu archivo **app/config/database.php** es dónde se configura la base de datos. Una instalación desde cero no tiene el archivo database.php, por lo que tendrás hacer una copia del archivo database.php.default. Una vez hecha la copia y renombrado el archivo tenemos:

```
// app/config/database.php
var $default = array('driver'   => 'mysql',
                     'connect'  => 'mysql_connect',
                     'host'     => 'localhost',
                     'login'    => 'user',
                     'password' => 'password',
                     'database' => 'project_name',
                     'prefix'   => '');
```

Reemplaza la información que viene por defecto con la información de conexión para tu aplicación.

Una nota acerca de la llave prefix: el string que ingreses acá será agregado en todas las llamadas SQL que genera Cake al interactuar con las tablas de tu base de datos. Lo puedes definir acá una única vez para que no tengas que especificarlo en otras partes. También te permite seguir las convenciones de Cake en cuanto al nombramiento de tablas si por ejemplo estas en un webhost que sólo te otorga una única base de datos. Nota: para las tablas join HABTM, agregas el prefix sólo una vez: prefix_apples_bananas, no prefix_apples_prefix_bananas.

CakePHP soporta los siguientes drivers de base de datos:

- mysql
- postgres
- sqlite
- pear-drivername (por ejemplo, pear-mysql)
- adodb-drivername

La llave 'connect' en la conexión $default te permite especificar si la conexión será considerada persistente o no. Revisa los comentarios en el archivo database.php.default para obtener ayudar acerca de cómo especificar tipos de conexión para tu base de datos.

Las tablas de tu base de datos deberían seguir las siguientes convenciones:

- Nombres de tablas usadas por Cake deberían consistir en palabras en Inglés en plural, como “users”, “authors” o “articles”. Notar que los correspondientes molodes tienen nombres en singular.singular names.

- Tus tablas deben tener una llave primaria llamada 'id'.
- Si vas a relacionar tablas, usa llaves foráneas como estas: 'article_id'. El nombre de la tabla en singular, seguido por un underscore, seguido por 'id'.
- Si incluyes una columna llamada 'created' y/o 'modified' en tu tabla, Cake automáticamente poblará el campo.

Podrás notar que también se incluye una conexión $test en el archivo database.php. Al llenar esta configuración (o agregar otras con formato similar) la podrás usar en tu aplicación de la siguiente manera :

```
var $useDbConfig = 'test';
```

Dentro de uno de tus modelos. De esta menera puedes usar cualquier cantidad de conexiónes adicionales.

Modelos

Sección 1

¿Qué es un Modelo?

¿Que es lo que hace? Esto separa el domino lógico de la presentación, aislando la lógica.

Un Modelo es generalmente un punto de acceso a la base de datos, más específicamente a una tabla en concreto de la base de datos. Por defecto, cada modelo usa la tabla cuyo nombre es el plural de dicho modelo. Ej. el modelo 'User' usa la tabla 'users'. Los Modelos también pueden contener reglas de validación de datos, información sobre asociaciones, y métodos específicos para la tabla que usan. De esta manera es como se ve un modelo simple de Usuario (User) en Cake: Ejemplo Modelo Usuario (User Model), guardado en /app/models/user.php

```
<?php

// AppModel te da toda la funcionalidad de un Modelo en Cake

class User extends AppModel
{
    // Siempre es buena práctica incluir esta variable
    var $name = 'User';

    // Esto es usado para validar datos, ver Capitulo "Validación de
Datos" (Data Validation).
    var $validate = array();

    // También puedes definir asociaciones
    // Mira la sección 6.3 para más información.

    var $hasMany = array('Image' =>
                   array('className' => 'Image')
                   );

    // También puedes incluir tus propias funciones:
    function makeInactive($uid)
    {
        // Coloca tu lógica aquí
    }
}

?>
```

Section 2

Funciones del Modelo

Desde un punto de vista de PHP, los modelos son clases que extienden la clase AppModel. La clase AppModel es originalmente definida en el directorio cake/, pero usted puede querer crear la suya propia, ponga esta en app/app_model.php. Ésta debe contener ciertos métodos que son compartidos entre 2 o más modelos. La clase AppModel a su vez extiende la clase Model que es una librería estándar de Cake y es definida en cake/libs/model.php.

Mientras esta sección trata mayoritariamente las funciones mas comunes de un Modelo en Cake, es importante recordar usar el manual de referencia de Cake (http://api.cakephp.org) por una completa información. Funciones definidas por el usuario.

Un ejemplo de un método de tabla específico en un Modelo es un par de métodos para esconder y mostrar (hide, unhide) entradas (posts) en un blog.

Ejemplo de Funciones en un Modelo

```
<?php
class Post extends AppModel
{
   var $name = 'Post';

   function hide ($id=null)
   {
      if ($id)
      {
          $this->id = $id;
          $this->saveField('hidden', '1');
      }
   }

   function unhide ($id=null)
   {
      if ($id)
      {
          $this->id = $id;
          $this->saveField('hidden', '0');
      }
   }
}
?>
```

Recuperando tus Datos

Debajo vemos algunas de las maneras estándar de obtener tus datos usando un Modelo:

- findAll
    - string $conditions
    - array $fields
    - string $order
    - int $limit
    - int $page
    - int $recursive

Esta función regresa los campos específicos en cantidad de registros hasta $limit que cumplan las condiciones especificadas en $conditions (si las hay), empieza listando desde la página $page (por defecto página 1), $conditions debe verse como una sentencia SQL: $conditions = “race = 'wookie' AND thermal_detonators > 3”, como ejemplo.

Cuando la opción $recursive es establecida con un valor mayor a 1, la función findAll() debe hacer un esfuerzo por regresar los modelos asociados a los registros recuperados por findAll(). Si tu 'Propiedad' tiene muchos 'Dueños' quienes a su vez tienen muchos 'Contratos' findAll() en tu Modelo Propiedad debe regresar esos modelos asociados.

- find
    - string $conditions
    - array $fields
    - string $order
    - int $recursive

Devuelve los (o todos si no es especificado) campos especificados desde el primer record que cumpla las condiciones de $conditions.

Cuando a la opción $recursive se le asigna un valor entero entre 1 y 3, la función find() debe hacer un esfuerzo por regresar los modelos asociados a aquellos registros encontrados por find(). La recursión find puede subir hasta 3 niveles de profundidad. Si tu 'Propiedad' tiene muchos 'Dueños' quienes a su vez tienen muchos 'Contratos', una recursión find() en tu modelo 'Propiedad' deberá regresar hasta tres niveles de profundidad de modelos asociados.

- findAllBy<fieldName>
- string $value

Estas funciones mágicas pueden ser usadas como un acceso directo a buscar tus tablas para una fila dada en un cierto campo, y un cierto valor. Solo adjunta el nombre del campo que quieres buscar y escríbelo como 'CamelCase'. Ejemplos (usados en un controlador (Controller)) pueden ser:

$this→Post→findByTitle('My First Blog Post');
$this→Author→findByLastName('Rogers'); $this→Property→findAllByState('AZ');
$this→Specimen→findAllByKingdom('Animalia');

El resultado retogresado es un array formateado como devuelve find() o findAll().

- findNeighbours
- string $conditions
- array $field
- string $value

Devuelve un array con los modelos vecinos (solo con los campos especificados), especificados por $field y $value, filtrados por las condiciones SQL impuestas en $conditions.

Esto es útil en situaciones donde quieres tener links 'Anterior' y 'Siguiente' para que los usuarios naveguen alguna secuencia ordenada a través de tus entradas en el modelo. Esto solo funciona para campos numéricos y del tipo fecha y hora.

```
class ImagesController extends AppController
{
    function view($id)
    {
        // Decimos que queremos mostrar una imagen

        $this->set('image', $this->Image->find("id = $id");

        // Pero también queremos la imagen anterior y siguiente

        $this->set('neighbours', $this->Image->findNeighbours(null,
'id', $id);

    }
}
```

Esto nos da un completo $image['Image'] array, a lo largo con $neighbours['prev']['Image']['id'] y $neighbours['next']['Image']['id'] en nuestra vista.

- field
- string $name
- string $conditions
- string $order

Devuelve como cadena de texto (string) un único campo del primer registro que cumpla $conditions como condiciones ordenado por $order.

- findCount
- string $conditions

Devuelve el número de registros que cumplen las condiciones dadas.

- generateList
- string $conditions
- string $order
- int $limit
- string $keyPath
- string $valuePath

Esta función es un atajo para obtener una lista de pares de valores llave - especialmente para crear un tag select de html desde una lista de tus modelos. Usa $conditions, $order, y $limit como parámetros de igual manera que una solicitud findAll(). En los parametros $keyPath y $valuePath es donde le dices a tu modelo donde debe encontrar las llaves y valores para tu lista generada. Por ejemplo, si tu quieres generar una lista de roles basados en tu modelo Rol, que abierto por ids con valores enteros, la función completa se invoca como algo así: $this→set(

```
  'Roles',
```

```
  $this->Role->generateList(null, 'role_name ASC', null,
'{n}.Role.id', '{n}.Role.role_name')

);

//This would return something like:
array(
    '1' => 'Account Manager',
    '2' => 'Account Viewer',
    '3' => 'System Manager',
    '4' => 'Site Visitor'
);
```

- read
- string $fields
- string $id

Usa esta función para obtener los campos y sus valores desde un registro actualmente recuperado, o de un registro especificado por $id.

Por favor nota que las operaciones read() solo afectarán el primer nivel de asociaciones de los modelos tega el valor que tenga $recursive en el modelo. Para ganar niveles adicionales de acceso, usa find() o findAll().

- query
- string $query

- execute
- string $query

Las llamadas SQL pueden ser hechas usando la función query() y execute() desde un modelo. La diferencia entre estas dos funciones es que query() es usada para hacer consultas personalizadas de SQL (el resultado que regresan), y execute() es usada para ejecutar comandos SQL personalizados que no requieren valor de retorno.

Llamadas SQL Personalizadas con query()

```
<?php
class Post extends AppModel
{
    var $name = 'Post';

    function posterFirstName()
    {
        $ret = $this->query("SELECT first_name FROM posters_table
                                WHERE poster_id = 1");
        $firstName = $ret[0]['first_name'];
        return $firstName;
    }
}
?>
```

Condiciones Complejas Find (usando arrays)

La mayoría de las llamadas de búsqueda en los modelos incluyen pasar condiciones de una u otra manera. El acercamiento más simple a esto es usar la clausula WHERE de SQL, pero si necesitas más control, debes usar arrays. Usar arrays es una forma mas clara y fácil de leer, y también de construir consultas. Esta sintaxis también separa los elementos de tu consulta (campos, valores, operadores, etc.) en discretas y manipulables partes. Esto permite a Cake generar la consulta más eficiente posible, asegurando una sintaxis SQL apropiada, y debidamente escapada (escapar caracteres) en cada parte de la consulta.

En su forma más básica un consulta basada en arrays se ve así:

Ejemplo de Condiciones Básicas de Búsqueda Usando Arrays:

$conditions = array(“Post.title” ⇒ “This is a post”);

//Ejemplo de uso con un Modelo:

```
$this->Post->find($conditions);
```

La estructura es se explica bastante por si misma: eso debe encontrar cualquier post donde el título concuerde con la cadena “This is a post”. Note que podríamos haber usado “title” como nombre de campo, pero siempre es una buena práctica cuando se construyen consultas, especificar el nombre del modelo al que pertenece, clarificando la lectura del código, y previendo colisiones en un futuro. ¿Que hay acerca de otros tipos de coincidencias? Estas son igualmente simples. Dejanos mostrarte como encontramos todos los posts donde el título no es “This is a post”:

```
array("Post.title" => "<> This is a post")
```

Lo único que fue adherido es '<>' antes de la expresión. Cake puede parsear cualquier operador de comparación de SQL válido, incluyendo expresiones usando LIKE, BETWEEN, o REGEX, siempre y cuando tu dejes un espacio entre el operador y la expresión o el valor. Una excepción aquí es IN (…). Digamos que quieres encontrar posts donde el título es una serie de valores:

```
array("Post.title" => array("First post", "Second post", "Third
post"))
```

Agregar filtros adicionales a las condiciones es simple como agregar pares de llaves/valores adicionales al array:

```
array
(
    "Post.title"   => array("First post", "Second post", "Third
post"),
    "Post.created" => "> " . date('Y-m-d', strtotime("-2 weeks"))
)
```

Por defecto, Cake junta multiples condiciones con boleanos AND; lo que significa, el código anterior solo encontrará posts que han sido creados en las 2 semanas pasadas, y tienen algún título de los dados. Como sea, nosotros podemos encontrar fácilmente posts que concuerden con esta otra condición:

```
array
("or" =>
    array
    (
        "Post.title" => array("First post", "Second post", "Third
post"),
        "Post.created" => "> " . date('Y-m-d', strtotime("-2 weeks"))
    )
)
```

Cake acepta todos los operadores boleanos SQL válidos, incluyendo AND, OR, NOT, XOR, etc., y ellos pueden estar en mayúsculas o minúsculas, como prefieras. Estás condiciones son infinitamente anidables. Digamos que tu tienes relaciones hasMany/belongsTo entre Posts y Authors, lo que resultaria en un LEFT JOIN el la búsqueda terminada en Post. Digamos que quieres encontrar todos los posts que contienen una determinada palabra llave o fueron creados en las ultimas dos semanas, pero tu quieres restringir tu búsqueda a los posts escritos por Bob:

```
array
("Author.name" => "Bob", "or" => array
    (
        "Post.title" => "LIKE %magic%",
        "Post.created" => "> " . date('Y-m-d', strtotime("-2 weeks")
    )
)
```

Controladores

Sección 1

¿Qué es un controlador?

Un controlador es usado para manejar la lógica de cierta sección de tu aplicación. Comúnmente, los controladores son usados para manejar la lógica de un sólo modelo. Por ejemplo, si estás construyendo un sitio que maneja una colección de videos, podrías tener un VideoController y un ArriendoController manejando los videos y arriendos, respectivamente. **En Cake, los nombres de los controladores están siempre en plural**.

Los controladores de tu aplicación son sub-clases de la clase AppController de Cake, que a su vez extiende la clase principal Controller. Los controladores pueden tener cualqueir cantidad de acciones: funciones usadas en tu aplicación web para mostrar vistas.

La clase AppController puede ser definida en **/app/app_controller.php** y debe contener métodos que son compartidos entre dos o más controladores. A su vez, AppController es una sub-clase de Controller que es una clase de la biblioteca estándar de Cake.

Una acción es una única funcionalidad de un controlador. Es ejecutada automáticamente por el Dispatcher si una solicitud de página entrante la especifica en la configuración de rutas (routes). Retomando el ejemplo de la colección de videos, nuestro VideoController podría tener las acciones `view()`, `rent()`, y `search()`. El controlador debería estar ubicado en /app/controllers/videos_controller.php y contener:

```
class VideosController extends AppController
{
    function view($id)
    {
        //lógica de la acción...
    }

    function rent($customer_id, $video_id)
    {
        //lógica de la acción...
    }

    function search($query)
    {
```

```
        //lógica de la acción...
    }
}
```

Deberías poder acceder a estas acciones usando las siguientes URLs de ejemplo:

```
http://www.example.com/videos/view/253
http://www.example.com/videos/rent/5124/0-235253
http://www.example.com/videos/search/hudsucker+proxy
```

Pero cómo se verían las páginas? Podrías necesitar definir una vista para cada una de estas acciones - revisa como hacerlo en el próximo capítulo, pero: las siguientes secciones te mostrarán como dominar el poder de un controlador Cake. Específicamente vas a aprender como hacer que tu controlador le pase datos a la vista, redireccionar al usuario, y mucho más.

Sección 2

Funciones del Controlador

Aunque esta sección describirá las funciones más usadas en los controladores de Cake, es importante recordar usar http://api.cakephp.org como una referencia completa.

Interactuando con tus Vistas

- **set**
    - string *$variable*
    - mixed *$valor*

Esta función es la principal manera de llevar datos desde el controlador a la vista. Puedes usarla para enviar lo que sea: variables, arreglos, etc. Una vez usada `set()`, la variable puede ser usada en tu vista; ejecutando set('color', 'blue') en tu controlador hace que $color este disponible en la vista.

- **validateErrors**

Retorna la cantidad de errores generados por un `save` no exitoso.

- **validate**

Valida los datos del modelo de acuerdo a sus reglas de validación. Para más información revisar "Validación de datos".

- **render**
    - string *$accion*
    - string *$layout*
    - string *$archivo*

Esta función puede que no la necesites muy a menudo, ya que `render()` es automáticamente llamada al final de cada acción del controlador, y será renderizada la

vista con el mismo nombre de la acción del controlador. Un uso alternativo podría ser ejecutar esta función en cualquier parte de la lógica del controlador.

Redirección de Usuarios

- **redirect**
  - string *$url*

Indica al usuario hacia dónde dirigirse. La URL usada como parámetro puede ser una URL interna del Cake, o una URL del tipo http://URL

- **flash**
  - string *$mensaje*
  - string *$url*
  - int *$pausa*

Esta función muestra $mensaje por la cantidad $pausa de segundos dentro del layout de flash (ubicado en app/views/layouts/flash.thtml) luego redirecciona al usuario a la $url especificada.

Las funciones `redirect()` y `flash()` de Cake no incluyen una llamada a `exit()`. Si deseas que tu aplicación termine luego de un `redirect()` o `flash()`, deberás incluír tu propia llamada a `exit()` inmediatamente despúes. También probablemente sea preferible llamar a `return` en vez de `exit()`, dependiendo de la situación (por ejemplo, si necesitas que se ejecute un callback).

Callbacks de Controlador

Los controladores de Cake incorporan algunos callbacks que pueden ser usados para ejecutar código antes o despúes de alguna función del controlador. Para utilizar esta funcionalidad se deben declarar estas funciones en tu controlador usando los parámetros y valores de retorno acá detallados.

- **beforeFilter**

Llamada antes de cada acción del controlador. La función perfecta para verificar sesiones activas y privilegios de usuarios.

- **afterFilter**

Llamada después de cada acción del controlador.

- **beforeRender**

Llamada después de la lógica del controlador, y justo antes que una vista es renderizada.

Otras Funciones Útiles

A pesar de que estas funciones son parte de la clase Object de Cake, también estan disponibles dentro de un Controlador:

- **requestAction**
  - string *$url*
  - array *$extra*

Esta función llama una acción del controlador desde cualquier ubicación y devuelve su vista renderizada. La $url es una URL de Cake (/nombrecontrolador/nombreaccion/parametros). Si el arreglo $extra incluye el indice 'return', AutoRender es automáticamente dejado en true para la acción del controlador.

Puedes usar requestAction para obtener datos desde otra acción del controlador, u obtener una vista totalmente renderizada desde un controlador.

Primero, obtener los datos desde un controlador es muy simple. Sólo debes usar requestAction en la vista donde se necesitan los datos.

```
//Acá nuestro controlador simple:

class UsersController extends AppController
{
    function getUserList()
    {
        return $this->User->findAll();
    }
}
```

Imagine que necesitamos crear una tabla simple que muestre los usuarios del sistema. En vez de duplicar el código de otro controlador, podemos obtener los datos de `UsersController::getUserList()` usando `requestAction()`.

```
class ProductsController extends AppController
{
    function showUserProducts()
    {
        $this->set('users', $this-
>requestAction('/users/getUserList'));

        // Ahora la variable $users en la vista tendrá los datos de
        // UsersController::getUserList().
    }
}
```

Si tienes un elemento no estático y muy usado en tu aplicación, podrías usar `requestAction()` para inyectarlo dentro de tus vistas. Por ejemplo en vez de pasar datos desde `UsersController::getUserList()` podríamos quierer renderizar la vista de esa acción (que podría consistir en una tabla) dentro de otro controlador. Esto nos ahorra duplicar código en la vista.

```
class ProgramsController extends AppController
{
```

```
    function viewAll()
    {
        $this->set('userTable', $this-
>requestAction('/users/getUserList', array('return')));

        // Ahora podemos mostrar $userTable en la vista de esta accion
para
        // ver la vistas renderizada que también esta disponible en
/users/getUserList.
    }
}
```

Por favor notar que las acciones llamadas usando `requestAction()` son renderizadas usando un layout vacío - de esta manera no debe preocuparse acerca de layouts siendo renderizadas en layouts.

La función `requestAction()` también es útil en AJAX donde un pequeño elemento de una vista podria ser poblado antes o durante una actualización AJAX.

- **log**
  - string *$message*
  - int *$tipo = LOG_ERROR*

Puedes usar esta función para guardar un log de diferentes eventos que ocurren dentro de tu aplicación web. Los logs quedan guardados en el directorio **/tmp** de Cake.

Si $tipo equivale a la constante LOG_DEBUG de PHP, el mensaje será escrito al log como un mensaje de tipo debug. Cualquier otro tipo es escrito al log como un error.

```
// Dentro de un controlador se puede usar log() para escribir:

$this->log('Mayday! Mayday!');

// Entrada Log:

06-03-28 08:06:22 Error: Mayday! Mayday!

$this->log("Parece que el usuario {$_SESSION['user']} acaba de
ingresar.", LOG_DEBUG);

// Entrada Log:

06-03-28 08:06:22 Debug: Parece que el usuario Bobby acaba de
ingresar.
```

- **postConditions**
  - array *$data*

Un método para pasarle $this→data, y este retornará un arreglo formateado como un arreglo de condiciones para ser usado en un modelo.

Por ejemplo, si tengo un formulario de búsqueda de personas:

```
// app/views/people/search.thtml:
```

```
<?php echo $html->input('Person/last_name'); ?>
```

Enviando el formulario con este elemento resultaría en el siguiente arreglo $this→data:

```
Array
(
    [Person] => Array
        (
            [last_name] => Anderson
        )
)
```

Ahora podemos usar `postConditions()` para darle formato a estos datos y usarlo en un modelo:

```
// app/controllers/people_controller.php:

$conditions = $this->postConditions($this->data);

// Resulta un arreglo como este:

Array
(
    [Person.last_name] => Anderson
)

// Que puede ser usando en las operaciones find de un modelo:

$this->Person->findAll($conditions);
```

Sección 3

Variables de Controlador

Manipular algunas variables especiales dentro de tu controlador permite aprovechar algo de funcionalidad extra de Cake:

$name

A PHP 4 no le agrada usar el nombre de la clase actual en CamelCase. Usa esta variable para setear el nombre CamelCased correcto si tienes problemas.

$uses

Tu controlador usa más de un modelo? Tu FragglesController cargará automáticamente $this→Fraggle, pero si también necesitas $this→Smurf, prueba agregando algo como esto en tu controlador:

```
var $uses = array('Fraggle','Smurf');
```

Por favor observar que es necesario incluir también el modelo Fraggle en el arreglo $uses, a pesar de que antes estuvo automáticamente disponible.

$helpers

Usa esta variable para que tu controlador cargue helpers dentro de sus vistas. El helper HTML es cargado automáticamente, pero puedes usar esta variable para especificar algunos otros:

```
var $helpers = array('Html','Ajax','Javascript');
```

Recuerda que debes incluir el HtmlHelper en el arreglo $helpers si quieres usarlo. Normalmente está disponible por defecto, pero si defines $helpers sin él, tendrás mensajes de error en tus vistas.

$layout

Setear esta variable con el nombre del layout que te gustaría usar para este controlador.

$autoRender

Al setear esta variable a **false** tus acciones dejarán de ser renderizadas automáticamente.

$beforeFilter

Si quieres que algo de código se ejecute cada vez que una acción es llamada (y antes que se ejecute el código de esa acción) puedes usar $beforeFilter. Esta funcionalidad es muy buena para el control de acceso - puedes verificar los permisos de un usuario antes que se ejecute una acción. Esta variable debe ser seteada usando un arreglo que contenga la(s) accion(es) del controlador que deseas se ejecuten:

```
class ProductsController extends AppController
{
    var $beforeFilter = array('checkAccess');

    function checkAccess()
    {
        //Lógica para verificar la identidad de usaurio y su acceso
    }

    function index()
    {
        //Cuando se llame a esta acción, primero será ejecutada la
acción checkAccess()
    }
}
```

$components

Al igual que $helpers y $uses, esta variable es usada para cargar componentes que quieres usar:

```
var $components = array('acl');
```

Sección 4

Parámetros de Controlador

Los parámetros del controlador están disponibles en $this→params en tu controlador Cake. Esta variable es usada para traer datos dentro del controlador y para proveer información de acceso acerca de la actual solicitud. El uso más común de $this→params es para tener acceso a la información que ha sido entregada al controlador mediante las operaciones POST o GET.

$this->data

Usado para manjear los datos POST enviados desde formularios HTML Helper al controlador.

```
// Un HTML Helper es usado para crear un elemento de formulario

$html->input('User/first_name');

// Al ser renderizado como HTML se verá así:

<input name="data[User][first_name]" value="" type="text" />

// Y cuando sea enviado al controlador mediante POST,
// aparecerá en $this->data['User']['first_name']

Array
(
    [data] => Array
        (
            [User] => Array
                (
                    [username] => mrrogers
                    [password] => myn3ighb0r
                    [first_name] => Mister
                    [last_name] => Rogers
                )

        )
)
```

$this->params['form']

Datos POST de cualquier tipo son almacenados acá, incluyendo información encontrada en $_FILES.

$this->params['bare']

Almacena '1' si el actual layout esta vacío, '0' de lo contrario.

$this->params['ajax']

Almacena '1' si el actual layout es ajax, '0' de lo contrario.

$this->params['controller']

Almacena el nombre del controlador que está manejando la solicitud. Por ejemplo, si fue llamada la URL /posts/view/1, $this→params['controller'] será igual a “posts”.

$this->params['action']

Almacena el nombre de la vista que está menejando la solicitud. Por ejemplo, si fue llamada la URL /posts/view/1 , $this→params['action'] será igual a “view”.

$this->params['pass']

Almacena el query string del GET enviado con la actual solicitud. Por ejemplo, si fue llamada la URL /posts/view/?var1=3&var2=4, $this→params['pass'] será igual a ”?var1=3&var2=4”.

$this->params['url']

Almacena la URL solicitada, junto con pares llave-valor de variables get. Por ejemplo, si fue llamada la URL /posts/view/?var1=3&var2=4, $this→params['url'] será algo como esto:

```
[url] => Array
        (
            [url] => posts/view
            [var1] => 3
            [var2] => 4
        )
```

[*Este capítulo ha sido traducido por Ricardo Mun~oz A.*]

Vistas

Vistas

Una vista es una plantilla, usualmente llamada ante una acción. Por ejemplo, la vista *PostController::add()* será encontrada en */app/views/post/add.thtml*. Las vistas en Cake son simples archivos PHP, asi puede usar código PHP dentro de ellas. Aunque la mayoria de sus archivos de vista contienen HTML, una vista puede ser, ciertamente, un grupo de datos, ser un XML, una imagen, etc. En la plantilla del archivo vista, puede usar los datos desde el correspondiente modelo. Estos datos son pasados como un array llamado *$data*. Cualquier dato que usted maneje en la vista usando set() en el controlador está tambien disponible en su vista.

Nota

El helper HTML está disponible en cualquier vista por defecto, y es lejos el helper mas usado en las vistas. Es muy útil en la creación de formularios, incluyendo scripts y multimedia, enlazando y agregando validacion de datos. Por favor, vea la seccion 1 en el capitulo “Helpers” para una discusion del helper HTML.

La mayoria de las funciones disponibles en las vistas son provistas por Helpers. Cake viene con un gran set de helpers (discutido en el capitulo “Helpers”), y usted puede incluir el suyo. Por que las vistas no contienen mucha lógica, no hay muchas funciones publicas en la clase vista. Uno que es útil es renderElement(), quien será explicado en la sección 1.2.

Layouts

Un layout contiene todo el codigo de presentacion que se oculta alrededor de una vista. Cualquier cosa que quiera ver en todas sus vistas debe estar en su layout. Los archivos Layout estan ubicados en */app/views/layouts*. El layout por defecto puede ser sobre escrito, colocando un nuevo layout en */app/views/layouts/default.thtml*. Una vez que el nuevo layout por defecto es creado, el código contralador de la vista es puesto dentro del layout por defecto cuando la página es recargada. Cuando usted crea un layout, necesita decirle a Cake donde poner el codigo del controlador: para hacer esto,

asegurese que su layout incluye un lugar para *$content_for_layout* (y opcionalmente, *$title_for_layout*). Aqui hay un ejemplo de como debe verse un layout por defecto:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title><?php echo $title_for_layout?></title>
<link rel="shortcut icon" href="favicon.ico" type="image/x-icon">
</head>
<body>

<!-- If you'd like some sort of menu to show up on all of your views,
include it here -->
<div id="header">
    <div id="menu">...</div>
</div>

<!-- Here's where I want my views to be displayed -->
<?php echo $content_for_layout ?>

<!-- Add a footer to each displayed page -->
<div id="footer">...</div>

</body>
</html>
```

Para asignar el titulo del layout, es fácil hacerlo en el controlador, usando la variable de controlador $pageTitle.

```
class UsersController extends AppController
{
    function viewActive()
    {
        $this->pageTitle = 'View Active Users';
    }
}
```

Puede crear cuantos layouts quiera para su sitio con Cake, solo coloquelos en el directorio app/views/layouts, y cambie entre ellos dentro del sus acciones en el controlador usando la variable $layout, o la funcion setLayout(). Por ejemplo, si una sección de mi sitio incluye un espacio para un banner pequeño. Puedo crear un nuevo layout con un espacio pequeño para publicidad y especificar que este es el layout por defecto a las acciones del controlador usando algo como esto:

```
var $layout = 'default_small_ad';
```

Elementos

Muchas aplicaciones tienen codigos pequeños en bloques de presentacion que necesitan ser repetidos página a página, algunas veces en lugares diferentes en el layout. Cake puede ayudar a repetir partes de su sitio web que necesitan ser reusadas. Esas partes reusables son llamada Elementos. Anuncios, cajas de ayuda, contrles de navegacion, menús extras, y llamadas son a menudo implementados en Cake como elementos. Un elemneto es basicamente una mini-vista que puede ser usada en otras vistas. Los

elementos viven en la carpeta */app/views/elements*/, y tienen la extension de archivo *.thtml*

Ejemplo 8.1 Llamando un Elemento sin parametros

```
<?php echo $this->renderElement('helpbox'); ?>
```

Ejemplo 8.2 Llamando un Elemento pasando datos en un array

```
<?php echo $this->renderElement('helpbox',array("helptext"=>"Oh, this
text is very helpful")); ?>
```

Dentro del archivo Elemento, todas las variables pasadas estan disponibles con el nombre de las llaves del array (muy parecido como set() trabaja en el controlador con las vistas). En el ejemplo de arriba, el archivo */app/views/elements/helpbox.thtml* puede usar la variable *$helptext*. Desde luego, podría ser más práctico pasar un array al elemento. Los elementos pueden ser usados para hacer una vista mas facil de leer, colocando la presentación del elemento repetido en su propio archivo. Pueden ayudarle tambien a reusar fragmentos de contendido en su sitio web.

Componentes

Presentación

Los componentes se utilizan para ayudar a los controladores en situaciones específicas. En lugar de extender las bibliotecas del motor de Cake, funcionalidades especiales se pueden poner en los componentes. Un tipo llamado *olle* en el canal IRC dijo una vez: `Un componente es un pequeño «controlador» compartido`. Esta nos parece una buena definición. Su objetivo principal es: reusabilidad. Los componentes son a los controladores lo que los helpers son a las vistas. La diferencia principal es que los componentes encapsulan **lógica de acciones** mientras los helpers encapsulan lógica de presentación. Este punto es muy importante, una confusión común para nuevos Bakers cuando tratan de obtener reusabilidad: Estoy tratando de hacer X, ¡¿debería hacer un componente o un helper?! Pues la respuesta es muy simple, ¿qué hace X? Hace lógica de acciones o lógica de presentación, ¿o tal vez ambas? Si es lógica de acciones entonces es un componente. Si es lógica de presentación entonces es un helper. Si es ambas, entonces..es tanto un componente como un helper. Un ejemplo de esto último sería un sistema de identificación. Usted querría ingresar, salir, restringir acceso, y probar permisos de un usuario a un recurso (una acción: editar, añadir, borrar.. o una url), esto es lógica de acciones, así que este sistema sería un componente. Pero también quiere añadir entradas en el menú principal cuando el usuario a ingresado al sistema, y esto es lógica de presentación.

Creando su propio componente

Para crear un componente, añada un archivo al directorio **app/controllers/components/**.

Supongamos que ha creado **foo.php**. Dentro de este archivo, tiene que definir una clase que corresponda al nombre del archivo (adjuntando la palabra 'Component' al nombre del archivo). Así, en este caso, se crearía lo siguiente:

Un componente simple

```
class FooComponent extends Object
{
    var $someVar = null;
    var $controller = true;
```

```
    function startup(&$controller)
    {
        // Este método hace una referencia al controlador que esta
cargando el componente.
        // Aquí se inicializa el controlador.
    }

    function doFoo()
    {
        $this->someVar = 'foo';
    }
}
```

Ahora, para usar su componente, tiene que añadir lo siguiente a la definición de su controlador:

```
var $components = array('Foo');
```

Dentro del controlador ahora puede usar:

```
$this->Foo->doFoo();
```

Un componente obtiene acceso al controlador que lo carga a través del método startup() mostrado arriba. Este método es llamado inmediatamente después de Controller::beforeFilter(). Esto le permite definir las propiedades que el componente necesita en su método startup() en el método beforeFilter. Para usar sus modelos dentro de los componentes, puede crear una nueva instancia así:

```
$foo =&amp; new Foo();
```

También puede usar otros componentes dentro del componente. Simplemente declare es el componente los componentes que quiera usar. En el ejemplo siguiente es el componente de sesiones.

```
var $components = array('Session');
```

Publicando sus componentes

Si cree que sus componentes pueden ser útiles para otras personas, puede agregarlo a CakeForge. Un componente que se vuelva bastante útil para la comunidad podría llegar a ser incluido en la distribución principal.

También vea el archivo de snippets para encontrar componentes enviados por los usuarios.

Sección 1

Helpers

Helpers (ayudantes) han sido pensados para proveer de funciones que son comúnmente necesitadas en las views (vistas) para formatear y presentar los datos de maneras útiles.

HTML

Introducción

El helper HTML es una de las maneras que utiliza Cake para hacer al desarrollo menos monótono y más rápido. El helper HTML tiene dos objetivos principales: ayudar en la inserción de secciones de código HTML que se repiten continuamente, y ayudar en la creación rápida y simple de formularios web. Las secciones siguientes le llevarán a través de las funciones más importantes en este helper, pero recuerde que http://api.cakephp.org siempre será la referencia definitiva sobre éste.

Muchas de las funciones en el helper HTML hacen uso de un fichero de definición de etiquetas HTML llamado **tags.ini.php**. Cake trae un tags.ini.php en su configuración, pero si quisiera hacerle algunos cambios, cree una copia de **/cake/config/tags.ini.php** y colóquela en su directorio **/app/config/**. El helper HTML utiliza las definiciones de etiquetas en este fichero para generar las etiquetas que solicite. Usar el helper HTML para crear parte del código de su view puede ser útil, debido a que cambios al fichero tags.ini.php resultarán en un cambio a nivel de toda la aplicación.

Adicionalmente, si en el fichero de configuración de su aplicación (**/app/config/core.php**) el parámetro AUTO_OUTPUT es puesto a *true*, el helper escribirá la etiqueta automáticamente, en lugar de devolver el valor. Esto es un esfuerzo de apaciguar a quienes no les agrada las etiquetas cortas (<?= ?>) o muchas funciones echo() en el código del view. Las funciones que incluyan el parametro $return permiten sobreescribir la configuración del fichero de configuración. Defina $return como *true* si quiere que el helper HTML retorne el código HTML sin hacer caso a como esté definido AUTO_OUTPUT.

Las funciones del helper HTML también incluyen un parametro $htmlAttributes, que permite añadir cualquier atributo extra a sus etiquetas. Por ejemplo, si quisiera añadir un atributo class, colocaría esto como valor de $htmlAttribute:

```
array('class'=>'someClass')
```

Insertar elementos bien formados

Si quisiera usar Cake para insertar elementos bien formados y repetidos constantemente en su código HTML, el helper HTML es grandioso para hacerlo. Hay funciones en este helper en insertan media, ayuda con las tablas, y hay incluso *guiListTree* el cual crea una lista sin ordenar basada en un array de PHP.

- **charset**
  - *string* $charset
  - *boolean* $return

Este se usa para generar una etiqueta META de codificación de caracteres.

- **css**
  - *string* $path
  - *string* $rel = 'stylesheet'
  - *array* $htmlAttributes
  - *boolean* $return = false

Crea un enlace a una hoja de estilo CSS. El parámetro $rel permite colocar un atributo rel= value en la etiqueta.

- **image**
  - *string* $path
  - *array* $htmlAttributes
  - *boolean* $return = false

Crea una etiqueta de imagen. El código que retorna esta función puede ser usado como parámetro de la función link() para crear imágenes con hipervinculos automáticamente.

- **link**
  - *string* $title
  - *string* $url
  - *array* $htmlAttributes
  - *string* $confirmMessage = false
  - *boolean* $escapeTitle = true
  - *boolean* $return = false

Esta función crea hipervinculos en el view. $confirmMessage es usado cuando se necesita un mensaje de confirmación vía JavaScript una vez el enlace es pulsado. Por ejemplo, un enlace que borra un objeto debería tener probablemente un mensaje de "¿Esta seguro?" para confirmar la acción antes de que el hipervinculo sea activado. Defina $escapeTitle como *true* si quiere que el helper HTML escape los datos que envía en la variable $title.

- **tableHeaders**
  - *array* $names
  - *array* $tr_options
  - *array* $th_options

Usado para crear las cabeceras de una tabla.

- **tableCells**
  - *array* $data
  - *array* $odd_tr_options
  - *array* $even_tr_options

Usado para crear un grupo de celdas formateadas.

- **guiListTree**
  - *array* $data
  - *array* $htmlAttributes
  - *string* $bodyKey = 'body'
  - *string* $childrenKey = 'children'
  - *boolean* $return = false

Genera una lista de árbol sin ordenar y anidada a partir de un array.

Formularios y validación

El helper HTML brilla cuando es momento de agilizar la inserción de formularios en sus views. Genera todas las etiquetas de formulario, rellena valores automáticamente en caso de errores, y muestra mensajes de error. Para ayudar a ilustrar esto, un rápido ejemplo. Imagine por un momento que su aplicación tiene un model *Note*, y quiere crear lógica de control y una vista para añadir y editar objetos Note. En su NotesController, tendría una acción edit que podría lucir así:

Acción Edit dentro de NotesController

```
function edit($id)
   {
      //Primero, revisamos si han sido enviados datos en el formulario
      //a la acción.
      if (!empty($this->data['Note']))
      {
         //Aquí tratamos de validar los datos del formulario (ver Cap.
12)
         //y guardarlos
         if ($this->Note->save($this->data['Note']))
         {
            //Si se guardan exitosamente, llevar al usuario
            //al lugar apropiado
            $this->flash('Your information has been saved.',
'/notes/edit/' . $id);
            exit();
         }
         else
         {
```

```
            //Generar los mensajes de error para los campos apropiados
            //esto no es realmente necesario ya que al guardar se hace
esto, pero es una llamada
            //de ejemplo a $this->Note->validates($this-
>data['Note']); si no está usando save
            //entonces use el método de abajo para llenar la función
de helper tagErrorMsg().
            $this->validateErrors($this->Note);

            //Y muestre el código de la vista edit
            $this->render();
         }
      }

      // Si no se ha recibido ningún dato en el formulario, obtener la
note que queremos editar, y enviar
      // está información a la vista
      $this->set('note', $this->Note->find("id = $id"));
      $this->render();
   }
```

Teniendo ya nuestro controller listo, veamos el código de la vista (que se encontraría en **app/views/notes/edit.thtml**). El modelo Note es bastante simple en este punto ya que solo contiene un id, el id del usuario y el cuerpo del mensaje. Este código sirve para mostrar los datos de Note y permite al usuario ingresar nuevos valores y guardar los datos en el modelo.

El helper HTML está disponible en todas las vistas por omisión, y puede accesarse a el usando $html.

Específicamente, miremos a la tabla donde encontramos las entrañas del formulario:

Código de ejemplo de la vista Edit (edit.thtml)

```
<!-- This tag creates our form tag -->

<?php echo $html->formTag('/notes/edit/' . $html-
>tagValue('Note/id')?>

<table cellpadding="10" cellspacing="0">
<tr>
   <td align="right">Body: </td>
   <td>

      <!-- Here's where we use the HTML helper to render the text
           area tag and its possible error message the $note
           variable was created by the controller, and contains
           the data for the note we're editing. -->
      <?php echo
      $html->textarea('Note/body', array('cols'=>'60', 'rows'=>'10'));
      ?>
      <?php echo $html->tagErrorMsg('Note/body',
      'Please enter in a body for this note.') ?>
   </td>
</tr>
<tr>
   <td></td>
   <td>
```

```
      <!-- We can also use the HTML helper to include
           hidden tags inside our table -->

      <?php echo $html->hidden('Note/id')?>
      <?php echo $html->hidden('note/submitter_id', $this->controller-
>Session->read('User.id'))?>
   </td>
</tr>
</table>

<!-- And finally, the submit button-->
<?php echo $html->submit()?>

</form>
```

La mayor parte de las funciones que generar el formulario (junto con tagErrorMsg) requieren que proporcione un $fieldName. Este $fieldName permite a Cake saber que datos está pasando para que pueda guardar y validar los datos correctamente. La cadena pasada en el parámetro $fieldName se escribe en la manera «modelname/fieldname.» Si fuera a añadir un campo de título a nuestro Note, necesitaría añadir algo a la vista que luzca así:

```
<?php echo $html->input('Note/title') ?>
<?php echo $html->tagErrorMsg('Note/title', 'Please supply a title for
this note.')?>
```

Los mensajes de error mostrados por la función tagErrorMsg() se envuelven en <div class=“error_message”></div> para utilizar CSS con mayor facilidad.

Aquí están las etiquetas que el helper HTML puede generar (la mayoría son sencillas):

- **submit**
  - string $buttonCaption
  - array $htmlAttributes
  - boolean $return = false
- **password**
  - string $fieldName
  - array $htmlAttributes
  - boolean $return = false
- **textarea**
  - string $fieldName
  - array $htmlAttributes
  - boolean $return = false
- **checkbox**
  - string $fieldName
  - array $htmlAttributes
  - boolean $return = false
- **file**
  - string $fieldName

    - array $htmlAttributes
    - boolean $return = false

- **hidden**
    - string $fieldName
    - array $htmlAttributes
    - boolean $return = false

- **input**
    - string $fieldName
    - array $htmlAttributes
    - boolean $return = false

- **radio**
    - string $fieldName
    - array $options
    - array $inbetween
    - array $htmlAttributes
    - boolean $return = false

- **tagErrorMsg**
    - string $fieldName
    - string $message

El helper HTML también incluye un set de funciones que ayudan en la creación de etiquetas de opciones relacionadas con fechas. El parámetro $tagName debería ser manejado de la misma manera que el parámetro $fieldName. Solo provea el nombre del campo para el que esta etiqueta de fecha es relevante. Una vez la data es procesada, la verá en el controller con la parte de la fecha que maneja concatenado al final del nombre del campo. Por ejemplo, si mi Note tiene un campo deadline del tipo fecha, y si el parámetro $tagName de la función dayOptionTag fuera «note/deadline», el dato de día se mostraría en la variable $params una vez el formulario haya sido enviado a la acción de un controller:

```
$this->data['Note']['deadline_day']
```

Puede entonces utilizar esta información para concatenar el dato de tiempo en un formato que es amigable con la configuración de su base de datos. Este código se colocaría justo antes de intentar guardar los datos, y metido al array $data usado para guardar la información en el modelo.

Concatenar datos de tiempo antes de guardar un modelo (extracto de NotesController)

```
function edit($id)
   {
      //First, let's check to see if any form data has been submitted
to the action.
      if (!empty($this->data['Note']))
      {

         //Concatenate time data for storage...
```

```
        $this->data['Note']['deadline'] =
           $this->data['Note']['deadline_year'] . "-" .
           $this->data['Note']['deadline_month'] . "-" .
           $this->data['Note']['deadline_day'];

        //Here's where we try to validate the form data (see Chap.
10) and save it
        if ($this->Note->save($this->data['Note']))
        {

        ...
```

- dayOptionTag ($tagName, $value=null, $selected=null, $optionAttr=null)
- yearOptionTag ($tagName, $value=null, $minYear=null, $maxYear=null, $selected=null, $optionAttr=null)
- monthOptionTag ($tagName, $value=null, $selected=null, $optionAttr=null)
- hourOptionTag ($tagName, $value=null, $format24Hours=false, $selected=null, $optionAttr=null)
- minuteOptionTag ($tagName, $value=null, $selected=null, $optionAttr=null)
- meridianOptionTag ($tagName, $value=null, $selected=null, $optionAttr=null)
- dateTimeOptionTag ($tagName, $dateFormat= 'DMY', $timeFormat= '12', $selected=null, $optionAttr=null)

AJAX

El helper de Ajax de Cake utiliza las populares bibliotecas Prototype y script.aculo.us para operaciones Ajax y efectos en el lado del cliente. Para poder utilizar este helper, debe tener una versión actual de las bibliotecas de Javascripte de http://script.aculo.us en **/app/webroot/js/**. Además, cualquier vista que planea utilizar el Helper Ajax necesitará incluir esas bibliotecas.

La mayoría de las funciones en este helper esperan un array especial $options como parámetro. Este array se utiliza para especificar diferentes cosas acerca de la operación Ajax. Los diferentes valores que se puede especificar son:

AjaxHelper $options Keys

Javascript

Number

Text

Time

Cache

Sección 2

Cómo crear tus propios Helpers

¿Necesita ayuda con el código de su vista? Si se encuentra necesitando una lógica de vista específica una y otra vez, puede hacer su propio helper.

Extending the Cake Helper Class

/app/views/helpers/link.php

/app/views/helpers/link.php (logic added)

Including other Helpers

/app/views/helpers/link.php (using other helpers)

Using your Custom Helper

Contributing

Constantes Globales y Funciones de Cakephp

Aquí hay algunas constantes globales y funciones que puede ser que encuentres útiles para hacer tu aplicación con Cake.

Funciones Globales

Aquí están las funciones disponibles de Cake. Muchas son convenientes atajos de nombres funciones de php, pero algunas (como Vendor() y uses()) pueden ser incluidas en el código o realizar otras funciones útiles. Si buscas una pequeña función que haga algo que tengas que hacer en repetidas ocasiones. Este es el lugar adecuado para encontrarla.

- Config

Carga el archivo de configuración base de cake

- Uses
    - cadena $lib1
    - cadena $lib2…

Utilizado para cargar las bibliotecas base de cake (que están en **cake/libs/**). Da el nombre del archivo de la biblioteca sin la extensión php

```
uses('sanitize','security');
```

- vendor
    - cadena $lib1
    - cadena $lib2..

Se usa para cargar bibliotecas externas que se encuentran en el directorio **/vendor**. Da el nombre del archivo sin la extensión php

```
vendor('myWebService','nusoap');
```

- debug
    - mixta $var
    - booleana $showHTML = false

Si el nivel de la aplicación no es cero $var sera impresa, si $showHTML es true, los datos serán mostrados de una forma agradable para el navegador.

- a

Regresa un arreglo con los parámetros que fueron usados para llamar la función

```
function algunaFuncion() {
  echo print_r(a('Foo','bar'));
}

algunaFuncion();

//imprime:

array(
      [0] => 'foo',
      [1] => 'bar'
      );
```

- aa

Usada para crear arreglos asociativos formados por los parámetros que se le pasan a la función

```
echo aa('a','b');

//imprime:
array('a'=>'b')
```

- e
    - cadena $text

Es un conveniente atajo para echo()

- low

Es un conveniente atajo para strtolower()

- up

Es un conveniente atajo para strtoupper()

- r
    - cadena $search
    - cadena $replace
    - cadena $subject

Es un conveniente atajo para str_replace()

- pr
    - mixto $data

Es un conveniente atajo para:

```
echo "<pre>".print_r($data)."</pre>";
```

Solo se imprime si debug no es cero

- am
    - arreglo $array1
    - arreglo $array2…

Fusiona los arreglos que se le pasan a la función

- env
    - cadena $key

Obtiene una variable de las fuentes disponibles. Es usada como respaldo si $_SERVER o $_ENV están inhabilitadas

Esta función también emula PHP_SELF y DOCUMENT_ROOT en los servidores que no las soportan. De hecho es un buena idea usar siempre env() en lugar de $_SERVER o getenv() (especialmente si tu plan es distribuir código) puesto que es un atajo lleno emulaciones.

- cache
    - cadena $path
    - cadena $expire
    - cadena $target = 'cache'

Escribe la información en una variable $data en la ruta /app/tmp especificada por la variable $path como cache. El tiempo de espiración se especifica en $expire y debe de ser una cadena de texto valida strtotime(). La variable $target puede guardar la información ya sea como 'cache' o 'public'

- clearCache
    - cadena $search
    - cadena $path = 'views'
    - cadena $ext

Es usada para borrar los archivos de las carpetas de cache o limpiar el contenido de las carpetas de cache.

Si $search es una cadena, y coincide con el nombre del directorio de cache o con algún nombre de de archivo será entonces removido del cache. El parámetro $search puede ser pasado como un arreglo con los nombres de los archivos/carpetas que serán limpiados. Si esta vacía la variable entonces /app/tmp/cache/views será limpiada

El parametro $path es usado para especificar cual directorio dentro de /tmp/cache será limpiado. Por defecto es 'views'

El parametro $ext es usado para especificar las extensiones de los archivos que tu quieres limpiar

- stripslashes_deep
    - arreglo $array

Recursivamente quita las diagonales de todos los valores del arreglo

- countdim
    - arreglo $array

Regresa el número de la dimensión en el arreglo provisto.

- fileExistsInPath
    - cadena $file

Busca en la ruta si tiene el archivo, regresa el path archivo si lo encuentra o false si no lo encuentra

- convertSlash
    - cadena $string

Convierte las diagonales en guiones bajos y quita el primero y el último de la cadena

Constantes de CakePHP

ACL_CLASSNAME: El nombre de la clase que actualmente esta usandose y manejandose con las ACL para cakephp. Contiene una constante para permitir integrar clases de terceros.

ACL_FILENAME: El nombre del archivo donde la clase ACL_CLASSNAME puede ser encontrada

AUTO_SESSION: si esta en FALSE, session_start() no es automáticamente llamada durante cualquier petición de la aplicación

CACHE_CHECK: Si esta en false, el cache de la vista esta desactivado para la aplicación entera

CAKE_SECURITY: Determina el nivel de seguridad de la sesión para la aplicación en concordancia con CAKE_SESSION_TIMEOUT. Puede ser puesto como “low”, “medium” o “alto”. Dependiendo de la configuración, CAKE_SESSION_TIMEOUT es multiplicado dependiendo de lo siguiente:

1. low: 300
2. medium: 100

3. alto: 10

CAKE_SESSION_COOKIE: el nombre de la cookie de sesión de la aplicación.

CAKE_SESSION_SAVE: Puede ser 'php','file' o 'database'.

1. php: Cake usa el valor por defecto de PHP para el manejo de sesiones (usualmente definido en el archivo php.ini)
2. file: La información de sesión es guardad y manejada en /tmp
3. database: Guarda la sesión en la base de datos (ver el capitulo 'Componente de sesión de cake" para mas detalles)

CAKE_SESSION_STRING: es una cadena aleatoria que se usa en el manejo de sesión.

CAKE_SESSION_TABLE: Es el nombre de la tabla que se usara para guardar la información de sesión (si CAKE_SESSION_SAVE == 'database'). No incluye el prefijo si alguno ha sido especificado por defecto en la conexión de la base de datos.

CAKE_SESSION_TIMEOUT: Es el número de segundos hasta que la sesión espire. Esta es multiplicada por CAKE_SECURITY.

COMPRESS_CSS: si esta en TRUE, las hojas de estilo de CSS están comprimidas. Esto requiere permitir escribir en la carpeta /var/cache. Para usarla se hace una referencia de tu hoja de estilo usando /ccss (mas bien como /css) o usar cssTag().

DEBUG: define el nivel de avisos de error y de depuración en la aplicación hecha con cakephp cuando esta es mostrada. Puede ser un entero entre 0 y 3

- 0: Modo de producción. No muestra errores en la ejecución. Ni mensajes de depuración son mostrados
- 1: Modo de desarrollo. Muesta advertencias y errores,con mensajes de depuración
- 2: Igual que en 1, pero con la salida SQL
- 3: Igual que en 2 pero con toda la información del objecto actual (usualmente el controlador)

LOG_ERROR: Es la constante de error. Usada para diferenciar el error de loggin y debuggin ("logeo" y depuración). Actualmente PHP soporta LOG_DEBUG

MAX_MD5SIZE: Es el tamaño máximo (en bytes) que md5() permitirá

WEBSERVICES: si esta en true, la funcionalidad de cake para webservices sera puesta en marcha

Constantes para rutas de CAKEPHP

APP: Es la ruta de la carpeta de la aplicación.

APP_DIR: El nombre de la carpeta de la aplicación

APP_PATH: La ruta absoluta de la carpeta de la aplicación.

CACHE: La ruta para la carpeta que contendrá los archivos de cache.

CAKE: La ruta para los archivos de CAKE.

COMPONENTS: La ruta para la carpeta que contrendrá los componentes

CONFIGS: la ruta para la carpeta con los archivos de configuración

CONTROLLER_TESTS: La ruta de la carpeta para hacer pruebas

CONTROLLERS: Ruta para los controladores de la aplicación

CSS: La ruta para los archivos de CSS

ELEMENTS: la ruta para la carpeta que contiene los elements

HELPER_TESTS: La ruta para el directorio de pruebas de los HELPERS

HELPER: La ruta para el directorio que contiene los HELPERS

INFLECTIONS: La ruta para la carpeta de inflexiones (usualmente dentro de la carpeta de configuración)

JS: Ruta para la carpeta que contiene los archivos de javascript

LAYOUTS: Ruta para la carpeta de layouts

LIB_TESTS: Ruta para la carpeta que contiene la biblioteca de cake para pruebas

LIBS: Ruta para la carpeta donde están las bibliotecas de cake

LOGS: Ruta para la carpeta de logs

MODEL_TESTS: Ruta para la carpeta de pruebas de modelos

MODELS: Ruta de la carpeta que contiene los modelos

SCRIPTS: Ruta para la carpeta de scripts de Cake

TESTS: Ruta para la carpeta de pruebas (es padre para las carpetas de pruebas de models,controllers etc)

TMP: Ruta para la carpeta temporal

VENDORS: Ruta para la carpeta vendors

VIEWS: Ruta para la carpeta Views

Configuración De La Carpeta WebRoot

CORE_PATH: ruta donde estan las bibliotecas del núcleo de Cake

WWW_ROOT: Ruta donde esta la carpeta webroot de la aplicación (usualmente en /cake/)

CAKE_CORE_INCLUDE_PATH: Ruta donde están las bibliotecas del núcleo de cake

ROOT: El nombre del directorio padre donde esta index.php de Cakephp

WEBROOT_DIR: the name of the application's webroot directory.

Validación de datos

Sección 1

Validación de datos

Crear reglas de validación personalizadas puede ayudar a asegurar que los datos en un Modelo «Model» estén de acuerdo a las reglas del negocio de la aplicación, tales como contraseñas con longitud de ocho caracteres, nombres de usuario con únicamente letras, etc.

El primer paso para la validación de datos es crear las reglas de validación en el Modelo. Para hacer esto, usa el arreglo `Model::validate` en la definición del Modelo, por ejemplo:

/app/models/user.php

```
<?php
class User extends AppModel
{
   var $name = 'User';

   var $validate = array(
      'login' => '/[a-z0-9\_\-]{3,}$/i',
      'password' => VALID_NOT_EMPTY,
      'email' => VALID_EMAIL,
      'born' => VALID_NUMBER
   );
}
?>
```

Las validaciones son definidas utilizando expresiones regulares compatibles con Perl, algunas de las cuales están predefinidas en libs/validators.php. Estas son:

- VALID_NOT_EMPTY
- VALID_NUMBER
- VALID_EMAIL

- VALID_YEAR

Si existen validaciones presentes en la definición del modelo (p.e. en el arreglo `$validate`), estas serán parseadas y revisadas durante las operaciones de guardado (p.e. en el método `Model::save()`). Para validar los datos directamente utiliza los métodos Model::validates() (retorna falso si los datos son incorrectos) y Model::invalidFields() (el cual retorna un arreglo de mensajes de error).

Pero usualmente la validación de datos está implícita en el código del controlador. El siguiente ejemplo demuestra como crear una acción para manejo de formularios:

Acción para manejo de formularios en /app/controllers/blog_controller.php

```
<?php
class BlogController extends AppController {

   var $uses = array('Post');

   function add ()
   {
      if (empty($this->data))
      {
         $this->render();
      }
      else
      {
         if($this->Post->save($this->data))
         {
             //ok genial, datos válidos
         }
         else
         {
            //Peligro, Will Robinson. Errores de validación.
            $this->set('errorMessage', 'Por favor corrija los errores
a continuación.');
            $this->render();
         }
      }
   }
}
?>
```

La vista usada por esta acción puede verse así:

La vista para el formulario añadir en /app/views/blog/add.thtml

```
<h2>Añadir post al blog</h2>
<form action="<?php echo $html->url('/blog/add')?>" method="post">
    <div class="blog_add">
        <p>Title:
            <?php echo $html->input('Post/title',
array('size'=>'40'))?>
            <?php echo $html->tagErrorMsg('Post/title', Título
requerido.')?>
        </p>
        <p>Body
            <?php echo $html->textarea('Post/body') ?>
```

```
            <?php echo $html->tagErrorMsg('Post/body', 'Cuerpo del
mensaje requerido.')?>
        </p>
        <p><?=$html->submit('Guardar')?></p>
    </div>
</form>
```

El método `Controller::validates($model[, $model…])` es utilizado para revisar cualquier validación a la medida agregada en el modelo. El método `Controller::validationErrors()` retorna cualquier mensaje de error arrojado en el modelo para que puedan ser presentados en la vista por `tagErrorMsg()`.

Si quisieras realizar alguna validación a la medida aparte de la validación de Cake basada en expresiones regulares, puedes usar la función `invalidate()` de tu modelo para marcar campos como erróneos. Imagina que necesitas mostrar un error en un formulario cuando un usuario trate de crear un nombre de usuario que ya exista en el sistema. Debido a que no puedes solo pedir a Cake que encuentre eso usando expresiones regulares, tu necesitarás hacer tu propia validación, y marcar el campo como no válido para invocar el proceso normal de Cake para formularios no válidos.

El controlador podría verse así:

```
<?php

class UsersController extends AppController
{
    function create()
    {
        //Revisa para ver si los datos del formulario han sido
enviados
        if (!empty($this->data['User']))
        {
            //Vee si existe un usuario con es nombre de usuario.
            $user = $this->User->findByUsername($this-
>data['User']['username']);

            //Invalida el campo para disparar el helper HTML para
mensajes de error.
            if (!empty($user['User']['username']))
            {
                $this->User->invalidate('username');//genera
tagErrorMsg('User/username')
            }

            //Trata de guardar normalmente, no debería funcionar si el
campo ha sido invalidado.
            if($this->User->save($this->data))
            {
                $this->redirect('/users/index/saved');
            }
            else
            {
                 $this->render();
            }
        }
    }
}
```

```
?>
```

Plugins

CakePHP te permite configurar una combinacion de controladores, modelos, vistas y liberarlas como un plugin empaquetado que otros pueden utilizar en sus aplicaciones CakePHP. Tienes un modulo para administracion de usuarios, un blog sencillo, o un modulo de servicios web en una de tus aplicaciones? Empaquetala como un plugin CakePHP para que puedas reutilizarla en otras aplicaciones.

13.1 Crear un Plugin

Como ejemplo de trabajo, vamos a crear un plugin que ordene pizza por ti. Que podria ser mas util en cualquier aplicacion CakePHP? Para empezar, necesitamos colocar los archivos del plugin adentro del subdirectorio "*/app/plugins*". El nombre del directorio padre de todos los archivos del plugin es importante, y sera utilizado en muchos lugares, asi que eligelo sabiamente. Para este plugin, usaremos el nombre "pizza". Asi es como la configuracion se vera eventualmente:

Ejemplo 13.1. Disposicion del sistema de archivos del sistema para ordenar pizza.

```
 /app
     /plugins
         /pizza
             /controllers           <- Los controladores del plugin
van aqui
             /models                <- Los modelos del plugin van aqui
             /views                 <- Las vistas del plugin van aqui
             /pizza_app_controller.php      <- AppController del
plugin, nombrado despues que el plugin
             /pizza_app_model.php           <- AppModel del plugin,
nombrado despues que el plugin
```

No se requiere definir un AppController y un AppModel para cualquier aplicacion, pero si se requiere para plugins. Necesitaras crearlos antes que tu plugin trabaje. Estas 2 clases especiales se nombran despues del plugin, y extienden las aplicaciones padre AppController y AppModel. Asi es como deberian verse:

Ejemplo 13.2 plugin "pizza" AppController.

```
<?php

class PizzaAppController extends AppController
{
    //...
}

?>
```

Ejemplo 13.3 plugin "pizza" AppModel.

```
<?php

class PizzaAppModel extends AppModel
{
    //...
}

?>
```

Si olvidas definir estas dos clases especiales, CakePHP te enviara errores de tipo "Missing Controller" hasta que el problema sea rectificado.

13.2 Controladores de Plugin

Los controladores para nuestro plugin "pizza" seran guardados en "*/app/plugins/pizza/controller*s". Ya que lo principal sera rastrear las ordenes de pizza, necesitaremos un OrdersController para este plugin.

Nota

No se requiere, pero es recomendable utilizar nombres unicos para los controladores de plugins para evitar conflictos de nombres con aplicaciones padres. No es extraño pensar que una aplicacion padre pueda tener un UsersController, OrderController, o ProductController, asi que quiza necesites ser creativo con los nombres de tus controladores, o agregar el nombre del plugin al nombre de la clase (PizzaOrdersController, en este caso).

Asi pues, colocaremos nuestro PizzaOrdersControllers en "*/app/plugins/pizza/controllers*" y se vera asi:

```
<?php

class PizzaOrdersController extends PizzaAppController
{
   var $name = 'PizzaOrders';

   function index()
   {
      ...
   }
```

```
    function placeOrder()
    {
       //...
    }
}
?>
```

Notese como este controlador extiende el AppController del plugin (llamado PizzaAppController) en lugar de el AppController de la aplicacion.

13.3 Modelos de plugin

Los modelos para los plugins se guardan en "*/app/plugins/pizza/models*". Ya hemos definido un PizzaOrdersController para este plugin, ahora crearemos el modelo para ese controlador, llamado PizzaOrders (El nombre de la clase PizzaOrders es consistente con nuestro esquema de nombramiento, y es unico, asi que lo dejaremos como esta).

Ejemplo 13.5 /app/plugins/pizza/models/pizza_order.php

```
<?php

class PizzaOrder extends PizzaAppModel
{
   var $name = 'PizzaOrder';
}

?>
```

Nuevamente, notese que esta clase extiende PizzaAppModel en lugar de AppModel.

13.4. Vistas de plugin

Las vistas se comportan exactamente como lo hacen en aplicaciones normales. Solo colocalas en el subdirectorio correcto, dentro de "*/app/plugins/[plugin]/views*". Para nuestro plugin "pizza", necesitaremos cuando menos una vista para la accion PizzaOrdersController::index(), asi que incluyamosla tambien:

Ejemplo 13.6. /app/plugins/pizza/views/pizza_orders/index.thtml

```
<h1>Ordena una Pizza </h1>
<p>Nada combina mejor con Cake que una buena pizza!</p>
<!-- Un formulario de algun tipo podria ir aqui... -->
```

13.5. Trabajando con Plugins

Ahora que haz construido todo, deberia estar listo para distribuirlo (sugeririamos que tambien distribuyeras algunos extras como un readme, un archivo sql, etc.).

Una vez que un plugin ha sido instalado en ”*/app/plugins*”, puedes accesarlo en la direccion ”*/pluginname/controllername/action*”. En nuestro ejemplo de plugin “pizza”, accesaremos nuestro PizzaOrdersController en ”*/pizza/pizzaOrders*”.

Algunos tips finales para trabajar con plugins en tus aplicaciones CakePHP:

- Cuando no tengas un [Plugin]AppController y un [Plugin]AppModel, obtendras errores de tipo “Missing Controller” cuando trates de accesar un controlador de plugin.
- Puedes tener un controlador por default con el nombre de tu plugin. Si haces esto, puedes accesarlo via ”*/[plugin]/action*”. Por ejemplo, un plugin llamado “users” con un controlador llamado “UsersController” puede accesarse en ”*/users/add*” si no hay un plugin llamado “AddController” en tu subdirectorio ”*[plugin]/controllers*”.
- Los plugins usaran los 'layouts' del subdirectorio por defecto ”/app/views/layouts”.
- Puedes intercomunicar plugins usando en tus controladores:

```
$this ->requestAction('/plugin/controller/action');
```

- Si utilizas 'requestAction, asegurate que los nombres del controlador y del modelo sean unicos. De otra manera podrias obtener errores de PHP del tipo “redefined class …”.

Muchas gracias a Felix Geisendorfer (the_undefined) por el material inicial para este capitulo.

Listas de Control de Acceso

Sección 1

Entendiendo como trabaja ACL

Lo mas importante, las cosas potentes requieren algún tipo de control de acceso. Las listas de control de acceso son una forma de gestionar los permisos de las aplicaciones de una manera muy detallada, fácilmente mantenida y manejable. Las listas de control de acceso, o ACL (Access Control List), se encargan principalmente de dos cosas: cosas que quieren mas algo, y cosas que son necesitadas. En argot de ACL, las cosas (mayormente usuarios) que quieren usar algo son llamadas objetos de petición de acceso, o AROs (Access Request Object). Las cosas en el sistema que son necesitadas (mayormente acciones o datos) son llamadas objetos de control de acceso, ó ACOs (Access Control Objects). Las entidades son llamadas 'objetos' porque algunas veces el objeto que hace la petición no es una persona - algunas veces puede que se quiera limitar el acceso que tienen ciertos controladores de Cake y se tenga que iniciar la lógica en otras partes de tu aplicación. Los ACOs pueden ser cualquier cosa que se quiera controlar , desde una acción de controlador, un servicio web, hasta una línea en el díario en linea de la abuela.

Para usar todos los acrónimos de una vez: ACL es usado para decidir cuando un ARO puede tener acceso a un ACO.

Ahora, para entender esto, vamos a realizar un ejemplo práctico. Imagine, por un momento, que un sistema de computadora es usado por un grupo de aventureros. El líder del grupo quiere seguir adelante en su aventura mientras mantiene un cantidad saludable de privacidad y seguridad para los otros miembros del grupo. Los AROs involucrados son los siguientes:

```
Gandalf
Aragorn
```

```
Bilbo
Frodo
Gollum
Legolas
Gimli
Pippin
Merry
```

Estas son las entidades en el sistema que estarán haciendo peticiones (los ACOs) desde el sistema. Se debe de entender que ACL no es un sistema encargado de autentificar a los usuarios. Ya se debe de contar con alguna manera de almacenar la información del usuario y ser capáz de verificar la identidad del usuario cuando estos intentan acceder al sistema. Una vez que se sabe quien es el usuario, es cuando ACL empieza a brillar realmente. De regreso a nuestra aventura.

La siguiente cosa que Gandalf necesita realizar es hacer un lista inicial de cosas, o ACOs, que el sistema manejará. Su lista parecerá a algo como esto:

```
Armas
El anillo
Puerco Salado
Diplomacia
Cerveza
```

Tradicionalmente, los sistemas eran manejados usando una clase de matriz, que mostraba una colección básica de usuarios y permisos relacionados con objetos. Si esta información fuese almacenada en una tabla, se vería de la siguiente manera, con las X indicando acceso denegado y las O indicando permiso permitido.

	Armas	**El anillo**	**Puerco Salado**	**Diplomacia**	**Cerveza**
Gandalf	X	X	O	O	O
Aragorn	O	X	O	O	O
Bilbo	X	X	X	X	O
Frodo	X	O	X	X	O
Gollum	X	X	O	X	X
Legolas	O	X	O	O	O
Gimli	O	X	O	X	X
Pippin	X	X	X	O	O
Merry	X	X	X	X	O

A primera vista, parece que este sistema puede funcionar bastante bien. El trabajo puede hacerse para proteger la seguridad (solamente Frodo puede acceder al anillo) y protegerse contra accidentes (dejando alejados a los hobbits del puerco salado). Parece ser lo suficientemente detallado, y fácil de leer, ¿cierto?.

Para un sistema pequeño como este, tal vez una configuración de matriz puede funcionar. Pero para un sistema en crecimiento, o un sistema con un gran número de recursos (ACOs) y usuarios (AROs), una tabla puede convertirse difícil de manejar de una manera bastantemente rápida. Imagine intentar controlar el acceso a los cientos de campamentos de guerra e intentar manejarlos por unidad, por ejemplo. Otra desventaja de las matrices es que no se puede agrupar secciones de usuarios de una manera lógica,

o hacer cambio de permisos en cascada a grupos de usuarios basados en esas agrupaciones lógicas. Por ejemplo, sería bueno que automáticamente se le permitiera el acceso a los hobbits a la cerveza y al puerco una vez que la batalla ha terminado: Hacerlo de manera individual por cada usuario sería tedioso y un error vergonzoso, mientras que hacer el cambio de permisos en cascada a todos los 'hobbits' sería realmente fácil.

ACL es frecuentemente implementado en una estructura de árbol. Existe normalmente un árbol de AROs y un árbol de ACOs. Organizando los objetos en árboles, los permisos se pueden seguir manejando de modo detallado, mientras se mantiene una buena compostura en la idea principal. Siendo el líder sabio, Gandalf elige usar ACL en su nuevo sistema, y organiza sus objetos a lo largo de las siguientes lineas:

```
Comunidad del Anillo:
Guerreros
        Aragorn
        Legolas
        Gimli
Magos
        Gandalf
Hobbits
        Frodo
        Bilbo
        Merry
        Pippin
Visitantes
        Gollum
```

Estructurando nuestro grupo de esta manera, podemos definir control de accesos en el árbol, y aplicar esos permisos en cualquier hijo. El permiso por defecto es prohibir el acceso a todo. Mientras trabaje hacia abajo en el árbol, elegirá permisos y los aplicará. El último permiso aplicado (que aplica al ACO que se está imaginando) es el que usted guarda. Entonces, usando nuestro árbol de AROs, Gandalf da algunos permisos:

```
Comunidad del Anillo: [Prohibir: TODO]
        Guerreros                       [Permitir: Armas, Cerveza, Once
raciones]
                Aragorn
                Legolas
                Gimli
        Magos                           [Permitir: Puerco Salado,
Diplomacia, Cerveza]
                Gandalf
        Hobbits                         [Permitir: Cerveza]
                Frodo
                Bilbo
                Merry
                Pippin
        Visitantes                      [Permitir: Puerco Salado]
                Gollum
```

Si quisiéramos usar ACL para ver si Pippin tiene permitido acceder a la cerveza, primero obtendríamos su dirección en el árbol, que es Comunidad→Hobbits→Pippin. Entonces veríamos los diferentes permisos que residen en cada uno de esos puntos, y usaríamos el permiso más específico relacionado con Pippin y la Cerveza.

- Comunidad = PROHIBIR Cerveza, entonces prohibir (por que está establecido prohibir todos los ACOs).

- Hobbits = Permitir Cerveza, entonces permitir.

- Pippin = ?; No existe ninguna información específica con la cerveza entonces nos quedamos con PERMITIR.

- Resultado Final: permitir la cerveza.

El árbol también nos permite hacer ajustes más pequeños para tener un control detallado = mientras se cuente con la habilidad de hacer cambios generalizados a los grupos de AROs:

```
Comunidad del Anillo: [Prohibir: TODO]
        Guerreros                       [Permitir: Armas, Cerveza, Once
raciones]
                Aragorn                 [Permitir: Diplomacia]
                Legolas
                Gimli
        Magos                           [Permitir: Puerco Salado,
Diplomacia, Cerveza]
                Gandalf
        Hobbits                         [Permitir: Cerveza]
                Frodo                   [Permitir: Anillo]
                Bilbo
                Merry                   [Prohibir: Cerveza]
                Pippin                  [Permitir: Diplomacia]
        Visitantes                      [Permitir: Puerco Salado]
                Gollum
```

Se puede ver esto porque el ARO de Aragorn mantiene su permiso tal como los demás en el ARO del grupo de Guerreros, pero aún se puede realizar ajustes detallados y casos especiales cuando se ocupen. Otra vez, el permiso por defecto es PROHIBIR, y el único cambio mientras se baja en el árbol nos obliga a PERMITIR. Para ver si Merry puede acceder a la Cerveza, obtendríamos su ruta en el árbol: Comunidad→Hobbits→Merry y a trabajar de manera descendente, manteniendo el rastro de permisos relacionados con la cerveza.

- Comunidad = PROHIBIR (por que está establecido prohibir todo), entonces se prohíbe la cerveza

- Hobbits = PERMITIR: Cerveza, entonces se permite la cerveza.

- Merry = PROHIBIR cerveza, entonces se prohíbe la cerveza.

- Resultado Final: se prohíbe la cerveza.

Sección 2

Definiendo permisos: ACL de Cake basado en INI

La primera implementación de ACL en Cake estuvo basada en archivos INI almacenados en la instalación de Cake. Mientras que sea usable y estable, recomendamos que utilice la solución secundaria de base de datos, mayormente por su habilidad para crear nuevos ACOs y AROs al vuelo. Nos referimos a su utilización e aplicaciones simples - y especialmente para aquellas personas que no estén utilizando una base de datos por alguna razón.

Los permisos de ARO/ACO son especificados en **/app/config/acl.ini.php**. Las instrucciones especificando el acceso pueden ser encontradas al inicio de acl.ini.php:

```
; acl.ini.php - Cake ACL Configuration
;
---------------------------------------------------------------------
; Use este archivo para especificar los permisos de usuario.
; aco = objeto de control de acceso (access control object) (algo en
tu aplicación)
; aro = objeto de petición de acceso (access request object) (algo
pidiendo acceso)
;
; Los registros de Usuario son agregados de la siguiente manera:
;
; [uid]
; groups = grupo1, grupo2, grupo3
; allow = aco1, aco2, aco3
; deny = aco4, aco5, aco6
;
; Los Registros de Grupo son agregados de una manera similar:
;
; [gid]
; allow = aco1, aco2, aco3
; deny = aco4, aco5, aco6
;
; Las secciones de allow, deny y groups (permitir, prohibir y grupos)
son todas opcionales.
; NOTA: Los nombres de grupos *no pueden* ser nunca los mismos que los
nombres de usuarios!
```

Si se utiliza el archivo INI, se puede especificar usuarios (AROs), el (los) grupo(s) a los que pertenecen, y sus propios permisos. También puede especificar grupos junto con sus permisos. Para aprender como utilizar el componente ACL de Cake para verificar los permisos utilizando este archivo INI, vea la sección 11.4.

Sección 3

Defniendo permisos: Base de datos ACL de Cake

Comenzando

La implementación estándar de permisos ACL se almacena en base de datos. La Base de Datos ACL, o dbACL consiste en una conjunto de núcleo de modelos, y un script de línea de comandos que viene con su instalación de Cake. Los modelos que son utilizados por Cake para interactuar con su base de datos con el fin de almacenar y

obtener los nodos del árbol ACL. El script de línea de comandos es utilizado para ayudarle a iniciar y permitirle interactuar con sus árboles.

Para iniciar, primero debe de asegurarse que **/app/config/database.php** está presente y correctamente configurado. Para una nueva instalación de Cake, la manera más fácil de darse cuenta de esto es poniendo el directorio de instalación en un navegador web. Cerca de la parte superior de la página, debe de ver el siguiente mensaje "Your database configuration file is present." ("El archivo de configuración de su base de datos está presente") y "Cake is able to connect to the database" ("Cake ha podido conectarse a su base de datos") si se ha hecho correctamente. Vea la sección 4.1 para mas información de la configuración de su base de datos.

Después, utilice el script de línea de comandos de ACL para inicializar su base de datos y almacenar la información ACL. El script localizado en /cake/scripts/aclphp le ayudará a lograr esto. Inicialice su base de datos para ACL ejecutando el siguiente comando (desde su directorio **/cake/scripts**):

Inicializando su base de datos usando acl.php

```
$ php acl.php initdb
        Initializing Database... (Inicializando base de datos)
        Creating access control objects table (acos)... (Creando tabla
de objetos de control de acceso)
        Creating access request objects table (acos)... (Creando tabla
de objetos de petición de acceso)
        Creating relationships table (aros_acos)... (Creando tabla de
relaciones)
        Done. (Hecho)
```

En este punto, debe de ser capáz de verificar la base de datos de su proyecto y ver las nuevas tablas. Si es lo suficientemente curioso acerca de como Cake almacena el árbol de información en esas tablas, lea y entienda la base de datos modificada visitando cada nodo del árbol. Las tablas de acos y aros almacenan los nodos de sus respectivos árboles, y la tabla aros_acos es usada para vincular sus AROs a los ACOs que pueden acceder.

Ahora, debe de ser capaz de comenzar a crear sus árboles de AROs y ACOs.

Creando objetos de petición de acceso (AROs) y objetos de control de acceso (ACOs)

Existen dos maneras de referirse a los AROs/ACOs. Una es dándoles un id númerico, que es usualmente la llave primaria de la tabla a la que pertenecen. Y la otra manera es dándoles una cadena alias. Las dos no son exclusivas mutuamente.

La manera de crear un nuevo ARO es utilizando los métodos definidos en el modelo de Cake Aro. El método create() de la clase Aro toma tres parámetros: $link_id, $parent_id, y $alias. Éste método crea un nuevo objeto ACL bajo el padre especificado por parent_id - como objeto raíz si $parent_id es nulo. La variable $link_id le permite enlazar un objeto de usuario actual a las estructuras ACL de Cake. El parámetro alias le permite darle una dirección a su objeto sin utilizar un ID entero.

Antes de que podamos crear nuestros ACOs y AROs, necesitaremos cargar esas clases. La manera más fácil de hacer esto es incluir el Componente ACL de Cake en su controlador utilizando el arreglo $components:

```
var $components = array('Acl');
```

Una vez que hayamos realizado esto, veamos algunos ejemplos de como se vería crear estos objetos. El código siguiente puede ser colocado en cualquier parte de una acción de un controlador:

```
$aro = new Aro();

//Primero, configuramos unos cuantos AROs
//Estos objetos no tendrán ningún padre inicialmente.

$aro->create( 1, null, 'Bob Marley' );
$aro->create( 2, null, 'Jimi Hendrix');
$aro->create( 3, null, 'George Washington');
$aro->create( 4, null, 'Abraham Lincoln');

// Ahora, podemos hacer grupos para organizar esos usuarios:
// Note que los IDs para estos objetos es 0, porque
//   nunca estarán atados a usuarios en nuestro sistema.

$aro->create(0, null, 'Presidentes');
$aro->create(0, null, 'Artistas');

// Ahora, "enganchemos" los AROs a sus respectivos grupos
$aro->setParent('Presidentes', 'George Washington');
$aro->setParent('Presidentes', 'Abraham Lincoln');
$aro->setParent('Artistas', 'Jimi Hendrix');
$aro->setParent('Artistas', 'Bob Marley');

// En pocas líneas, esta es la manera en que se utiliza
$aro = new Aro();
$aro->create($user_id, $parent_id, $alias);
```

El comando correspondiente del script en la linea de comandos sería: $acl.php create aco <link_id> <parent_id> <alias>.

Asignando permisos

Después de crear nuestros ACOs y AROs, podemos finalmente asignar permisos entre los dos grupos. Esto se hace utilizando el componente Acl del núcleo de Cake. Continuemos con nuestro ejemplo:

```
// Primero, en un controlador, necesitaremos acceso
// al componente ACL de Cake.

class AlgunControlador extends AppControler
{
        // Tal vez le gustaría colocar esto en el AppController
        // en lugar de aquí, pero también aquí funciona bien.

        var $components = array('Acl');

        // Recuerde: ACL siempre prohibirá algo acerca
```

```
                // de lo que no tiene información. Si alguna
                // verificación fue hecha en algo, será prohibida.
                // Permitamos que un ARO pueda acceder a un ACO.

                function algunaAccion()
                {
                        // PERMITIR

                        // Aquí está  como se concede a un ARO acceso
                        //  completo a un ACO
                        $this->Acl->allow('Jimi Hendrix', 'Guitarra
Eléctrica');
                        $this->Acl->allow('Bob Marley',   'Guitarra
Eléctrica');

                        // También podemos asignar permisos a grupos,
¿recuerda?
                        $this->Acl->Allow('Presidentes', 'Ejército de
los Estados Unidos');

                        // El método allow() tiene un tercer parámetro,
$accion.
                        // Puede especificar acceso parcial usando este
paremtro.
                        // $accion puede establecerce a create, read,
update o delete
                        //      (crear, leer, actualizar o borrar)
                        // Si no se especifica una acción, se asume
acceso total.

                        // Vea, no toque, sea caballero:
                        $this->Acl->allow('George Washington',
'Guitarra Electrica', 'read');
                        $this->Acl->allow('Abraham Lincoln',
'Guitarra Electrica', 'read');

                        // PROHIBIR

                        // El trabajo se prohíbe de la misma manera:

                        // Cuando se termine su mandato...
                        $this->Acl->deny('Abraham Lincoln', 'Ejército
de los Estados Unidos');
                }
        }
```

Este controlador en específico no es útil, pero su objetivo es mayormente mostrarle como funciona el proceso. Utilizar el componente Acl en conexión con su controlador manejador de usuarios sería la mejor utilización. Una vez que un usuario ha sido creado en el sistema, su ARO puede ser creado y colocado en la posición correcta del árbol, y los permisos pueden ser asignados a un ACO específico o grupo de ACOs basados en su identidad.

Los permisos también pueden ser asignados utilizando el script de línea de comandos empacado junto a Cake. La sintaxis es similar al funcionamiento del modelo, y puede ser visto ejecutando $ php acl.php help.

Sección 4

Verificando permisos : El componente ACL

La verificación de permisos es la parte más fácil de utilizar el ACL de Cake: consiste en utilizar un único método del componente Acl: check(). Una buena manera de implementar ACL en su aplicación podría ser el colocar una acción en su AppController que realice una verificación de ACL. Una vez colocado ahí, puede acceder al componente Acl y realizar verificación de permisos a través de toda la aplicación. Aquí está un ejemplo de implementación:

```
        class AppController extends Controller
        {
                //Obtenemos nuestro componente
                var $components = array('Acl');
                //verificaAcceso ($aco)
                function checkAccess($aco)
                {
                        //
                        $acceso = $this->Acl->check($this-Session-
>read('alias_usuario'), $aco, $accion ="*");
                        //Acceso prohibido
                        if ($acceso == false)
                        {
                                echo "Acceso Prohibido"
                                exit;
                        }
                        //Acceso permitido
                        else
                        {
                                echo "Acceso Permitido";
                                exit;
                        }
                }
        }
```

Básicamente, al haber puesto el componente Acl disponible en nuestro AppController, estará visible para utilizarse en cualquier controlador de su aplicación.

```
        //Éste es el formato básico:
        $this->Acl->check($aro, $aco, $action = '*');
```

Saneamiento de datos

15.1 Usando Sanitize en tu aplicacion.

Cake viene con Sanitize, una clase que puedes utilizar para eliminar datos maliciosos enviados por el usuario u otros datos no requeridos. Sanitize es una libreria del nucleo, y puede ser utilizada en cualquier lugar dentro de tu codigo, pero es probablemente mejor usada en controladores y modelos.

```
// Primero, incluir la libreria:
uses ('sanitize');
// Siguiente, crear un nuevo objeto sanitize:
$mrClean = new Sanitize();

// A partir de aqui, puedes usar Sanitize para limpiar los datos
// (Los metodos se explican en la siguiente seccion)
```

15.2 Haciendo los datos seguros para usarlo en SQL y HTML

Esta seccion explica como utilizar algunas de las funciones que Sanitize ofrece.

```
paranoid ($string, $allowedChars);
string $string;
array $allowedChars;
```

Esta funcion extrae cualquier cosa que no sea un simple caracter alfanumerico de la cadena $string. Puedes, sin embargo, dejarlo pasar ciertos caracteres, indicandolos dentro del arreglo $allowed.

```
$badString = ";:<script><html><   // >@@#";
echo $mrClean->paranoid($badString);

//output: scripthtml
```

```
 echo $mrClean->paranoid($badString, array(' ', '@'));

 //output: scripthtml    @@
 html ($string, $remove = false);

 string $string;
 boolean $remove = false;
```

Este metodo te ayuda a obtener datos enviados por el usuario listos para mostrarlos dentro del HTML. Esto es especialmente util si no quieres que los usuarios puedan romper tus 'layouts' o insertar imagenes o scripts dentro de los comentarios de un blog, post de un foro, y similares. Si la opcion $remove se establece en verdadero (true), cualquier codigo HTML sera removido en lugar de ser procesado como codigo HTML.

```
 $badString = '<font size="99"
color="#FF0000">HEY</font><script>...</script>';

 echo $mrClean->html($badString);

 //output: $lt;font size=&amp;quot;99&amp;quot;
color=&amp;quot;#FF000&amp;quot;&amp;gt;HEY&amp;lt;/font&amp;gt;&amp;lt;script&amp;gt;...&amp;lt;/scri
pt&amp;gt;

 echo $mrClean->html($badString, true);

 //output: font size=99 color=#FF0000 HEY fontscript...script
 sql ($string);

 string $string;
```

Se utiliza para librar declaraciones SQL, añadiendo diagonales, dependiendo de la configuracion de 'magic_quotes_gpc' del sistema.

```
 cleanArray ($dirtyArray);

 array @$dirtyArray;
```

Esta funcion es de tipo industrial, limpiador multiproposito, creada para ser usada en arreglos enteros (como $this→params['form'], por ejemplo). La funcion toma un arreglo y lo limpia: no se regresa ningun valor ya que el arreglo se pasa como referencia. Las siguientes operaciones de limpieza son realizadas en cada elemento del arreglo (recursivamente):

- Espacios impares (incluyendo 0xCA) son reemplazados por espacios regulares.
- El codigo HTML es reemplazado por su entidad HTML correspondiente (desde \n hasta
).
- Se hace un doble chequeo de caracteres especiales y se eliminan retornos de carro para incrementar la seguridad SQL.
- Se añaden diagonales para SQL (solo llamadas a la funcion SQL descrita anteriormente).
- Se intercambian las diagonales invertidas ingresadas por el usuario con diagonales invertidas de confianza.

El Componente de Sesión

Sección 1

Opciones de Guardado en Sesión para Cake

Cake viene predefinido para guardar los datos de sesión de 3 maneras: como archivos temporales dentro de la instalación de Cake, usando el mecanismo por default de PHP, o serializando en una base de datos. Cake utiliza la configuración por defecto de PHP. Para cambiar esa configuración y poder usar los archivos temporales o la base de datos, edita tu archivo de configuración core, ubicado en **/app/config/core.php**. Cambia la constante CAKE_SESSION_SAVE a 'cake', 'php' o 'database', dependiendo en las necesidades de tu aplicación.

Configuración de Sesión en core.php

```
/**
 * CakePHP includes 3 types of session saves
 * database or file. Set this to your preferred method.
 * If you want to use your own save handler place it in
 * app/config/name.php DO NOT USE file or database as the name.
 * and use just the name portion below.
 *
 * Setting this to cake will save files to /cakedistro/tmp directory
 * Setting it to php will use the php default save path
 * Setting it to database will use the database
 *
 *
 */
define('CAKE_SESSION_SAVE', 'php');
```

Para guardar los datos de sesión en la base de datos, necesitas crear una tabla en tu base de datos. El diseño para esa tabla se puede encontrar en **/app/config/sql/sessions.sql**.

Sección 2

Usando el componente de Sesión de Cake

El componente de sesión de Cake es usado para interactuar con la información de sesión. Incluye funciones básicas de lectura y escritura, pero también contiene características para el uso de sesiones para mensajes de error y de recibido (por ej. “Tus datos han sido guardados”). El componente de sesión esta disponible en todos los controladores de Cake por defecto.

Aqui hay algunas de las funciones que mas utilizaras:

- check
- string $name

Revisa si la llave actual especificada por $name ha sido enviada en la sesión.

- del
- string $name

- delete
- string $name

Borra la variable de sesión especificada por $name.

- error

Regresa el último error creado por el componente CakeSession. Comúnmente usado para debugging.

- flash
- string $key = 'flash'

Regresa el último mensaje establecido en la sesión usando setFlash(). Si $key ha sido establecida, el mensaje regresado es el mas recientemente guardado en esa llave.

- read
- string $name

Regresa la variable de sesión especificada por $name.

- renew

Renueva la sesión actualmente activa creando un nuevo ID de sesión, borrando la antigua y pasando la vieja información de sesión a la nueva.

- setFlash
- string $flashMessage
- string $layout = 'default'

- array $params
- string $key = 'flash'

Escribe el mensaje especificado por $flashMessage en la sesión (para ser después recuperada por flash()).

Si $layout esta establecido como 'default', el mensaje es guardado como ' '.$flashMessage.' '. Si $default esta establecido como '', el mensaje es guardado solo como ha sido pasado. Si otro valior es pasado, el mensaje es guardado dentro de la vista de Cake especificada por $layout.

Han sido colocados parametros en esta función para uso en un futuro. Revisa esto en un futuro para mas información.

La variable $key te permite guardar mensajes flash en las llaves. Revisa la documentación de flash() para obtener un mensaje basado en una llave.

- valid

Regresa true si la sesión es valida. Es mejor usado antes de operaciones read() para asegurarse que la información de sesión que esta tratando de accesar es valida.

- write
- string $name
- mixed $value

Escribe la variable especificada por $name y $valu en la sesión activa.

El Componente Request Handler

Sección 1

Introducción

El componente Request Handler es usado en Cake para determinar la forma en la cual la información llega en la petición HTTP. Puede usarse para informar mejor al controlador sobre peticiones AJAX, obtiene información sobre la dirección IP del cliente remoto y el tipo de petición, o saca datos indeseados de la salida. Para usar el componente Request Handler, será necesario especificarlo en el array $components del controlador.

```
class ThingsController extends AppController
{
    var $components = array('RequestHandler');

    // ...
}
```

Sección 2

Oteniendo información Client/Request

Vamos a entrar en materia:

- accepts
    - string $type

Devuelve información sobre el tipo de contenido que el client acepta, dependiendo del valor de $type. Si es null o no se pasa un valor, devolverá un array de los tipos de contenido que el cliente acepta. Si se pasa una cadena, devolverá verdadero si el cliente acepta el tipo dado, chequeando $type contra el mapa de tipo-contenido (ver setContent()). Si $type es un array, cada cadena es evaluada individualmente, y accepts() devolverá true si alguno de ellos encaja con un tipo de contenido aceptado. Por ejemplo:

```
class PostsController extends AppController
{
    var $components = array('RequestHandler');

    function beforeFilter ()
    {
        if ($this->RequestHandler->accepts('html'))
        {
            // Ejecuta solo el código si el cliente acepta una
respuesta HTML (text/html)
        }
        elseif ($this->RequestHandler->accepts('rss'))
        {
            // Ejecuta solo código RSS
        }
        elseif ($this->RequestHandler->accepts('atom'))
        {
            // Ejecuta solo código Atom
        }
        elseif ($this->RequestHandler->accepts('xml'))
        {
            // Ejecuta solo código XML
        }

        if ($this->RequestHandler->accepts(array('xml', 'rss',
'atom')))
        {
            // Ejecuta si el cliente acepta alguno de los siguientes:
XML, RSS o Atom
        }
    }
}
```

- getAjaxVersion

Si se esta usando una librería JS Prototype, tu puedes obtener un header especial para asignar a las peticiones AJAX . Esta función devuelve la versión utilizada de Prototype.

- getClientIP

Devuelve la dirección IP del cliente remoto.

- getReferrer

Devuelve el nombre del servidor desde el cual la petición ha sido originada.

- isAjax

Devuelve true si la actual petición fue un XMLHttpRequest.

- isAtom

Devuelve true si el cliente acepta contenido Atom feed (application/atom+xml).

- isDelete

Devuelve true si la actual petición fue vía DELETE.

- isGet

Devuelve true si la actual peticion fue vía GET.

- isMobile

Devuelve true si la cadena user agent concuerda con un navegador web mobile.

- isPost

Devuelve true si la actual petición fue vía POST.

- isPut

Devuelve true si la actual petición fue vía PUT.

- isRss

Devuelve true si los clientes aceptan contenido RSS feed (application/rss+xml).

- isXml

Devuelve true si el cliente acepta contenido XML (application/xml or text/xml).

- setContent
- string $name
- string $type

Añade un alias de tipo de contenido al mapa, para que sea usado con accepts() y prefers(), donde $name el el nombre de el mapping (string), y $type es una cadena cualquiera o un array de cadenas, cada uno de las cuales es un tipo MIME. La incorporación de mapas de tipos es como se muestra a continuación:

```
// Name     => Type
  'js'      => 'text/javascript',
  'css'     => 'text/css',
  'html'    => 'text/html',
  'form'    => 'application/x-www-form-urlencoded',
  'file'    => 'multipart/form-data',
  'xhtml'   => array('application/xhtml+xml', 'application/xhtml',
'text/xhtml'),
```

```
  'xml'     => array('application/xml', 'text/xml'),
  'rss'     => 'application/rss+xml',
  'atom'    => 'application/atom+xml'
```

Sección 3

Stripping Data

Ocasionalmente se podría querer borrar datos desde un petición o salida. Use las siguientes funciones de Request Handler para realizar estos tipos de operaciones.

- stripAll
    - string $str

Saca espacios en blanco, imágenes, y scripts de $str (usando stripWhitespace(), stripImages(), y stripScripts()).

- stripImages
    - string $str

Saca cualquier imágen embebida en el html de $str.

- stripScripts
    - string $str

Saca cualquier etiqueta <script> y <style> de $str.

- stripTags
    - string $str
    - string $tag1
    - string $tag2…

Borra las etiquetas especificadas por $tag1, $tag2, etc. de $str.

```
$someString = '<font color=\\"#FF0000\\"><bold>Foo</bold></font>
<em>Bar</em>';

echo $this->RequestHandler->stripTags($someString, 'font', 'bold');

// Salida: Foo <em>Bar</em>
```

- stripWhiteSpace
    - string $str

Saca espacios en blanco de $str.

Sección 4

Otras útiles funciones

El componente Request Handler es especialmente útil cuando una aplicación incluye peticiones AJAX. La función setAjax se usa para detectar automáticamente peticiones AJAX, y asignar al layout del controlador un AJAX layout para la petición. El beneficio es que se pueden hacer pequeñas vistas modulares que pueden también ser dobles con la vista AJAX.

```
// list.thtml
<ul>
<? foreach ($things as $thing):?>
<li><?php echo $thing;?></li>
<?endforeach;?>
</ul>

//-------------------------------------------------------------

//La accion list en ThingsController:
function list()
{
    $this->RequestHandler->setAjax($this);
    $this->set('things', $this->Thing->findAll());
}
```

Cuando una petición de un navegador normal se hace a /things/list, la lista desordenada es renderizada dentro del layout por defecto de la aplicación. Si la URL es peticionada como parte de una operación AJAX, la lista es automáticamente renderizada en el vacío AJAX layout.

El componente de Seguridad

Sección 1

Introducción

El componente de seguridad es usado para proteger las acciones de su controlador en contra de peticiones malignas o erróneas. Te permite configurar las condiciones bajo las cuales una acción puede ser llamada, y opcionalmente especifica como tratar con peticiones que no cumplen con esos requisitos. Antes de usar el componente de seguridad, debe de asegurarse que 'Security' este listado en el arreglo $components de su controlador.

Sección 2

Protegiendo las acciones del controlador

El componente de seguridad contiene 2 métodos primarios para restringir acceso a acciones del controlador:

- requirePost
- string $accion1
- string $accion2
- string $accion3…

Para que las acciones se puedan ejecutar, deben de ser llamadas por medio de POST.

- requireAuth
- string $accion1
- string $accion2
- string $accion3…

Se asegura que una petición proviene de la misma aplicación, verificando con una llave de autenticación que se encuentra en la información guardada en POST del formulario, contra una llave de autenticación guardada en la sesión del usuario. Si coinciden, la acción es llamada para ejecución. Sin embargo, tenga cuidado que por razones de flexibilidad, esta validación solo se hace si la información de un formulario ya ha sido enviada. Si la acción es llamada con una petición por medio de GET, requireAuth() no haría nada. Para máxima seguridad, debería de usar requirePost() y requireAuth() en acciones que desea completamente protegidas. Aprenda mas acerca de como la llave de autenticación es generada, y como termina donde debería en la sección 4 de mas abajo.

Pero primero, veamos un ejemplo simple:

```
class ThingsController extends AppController
{
    var $components = array('Security');

    function beforeFilter()
    {
        $this->Security->requirePost('delete');
    }

    function delete($id)
    {
        // Esto solo ocurriria si la accion es llamada por medio de
una peticion HTTP POST
        $this->Thing->del($id);
    }
}
```

Aqui, estamos diciendo al componente de seguridad que la acción 'delete' requiere una petición POST. El método beforeFilter() es usualmente donde usted quiere dice al componente de seguridad (y a casi todos los componentes) que hacer. Después ira a hacer lo que se le indica después de que el método beforeFilter() es llamado, pero justo antes de que la acción misma sea ejecutada.

Eso es todo lo que hay que hacer. Lo puede probar escribiendo la URL de la acción en su navegador y viendo que es lo que sucede.

Sección 3

Manejo de peticiones inválidas

Si una petición no cumple con los requisitos de seguridad que hemos definido, ¿que pasa con ella? por defecto, a la petición se le manda una cabecera con código 404 (se le llama 'Black-Hole'), y la aplicación de inmediato termina. Sin embargo, el componente

de seguridad tiene una propiedad $blackHoleCallback, la que puede configurar con el nombre de una función personalizada definida en su controlador.

En lugar de simplemente enviar una cabecera 404 y no hacer nada, esta propiedad le permite realizar algunas validaciones adicionales a la petición, direccionar la petición a otra ubicación o hasta registrar la dirección IP del cliente. Sin embargo, su elije un callback personalizado, es su responsabilidad el terminar la aplicación si la petición es inválida. Si su callback regresa el valor booleano 'true', entonces el componente de seguridad continuara para validar la petición contra otros requerimientos que tenga definidos. De otra manera, detiene la validación y su aplicación continua sin ser molestada.

Sección 4

Autenticaciones avanzadas a las peticiones

El método requireAuth() le permite ser muy detallado al especificar como y de donde la acción puede ser accesada, pero viene con ciertas estipulaciones de uso, las cuales se aclararan cuando entienda como este método de autenticación funciona. Como mencionamos antes, requireAuth() trabaja comparando una llave de autenticación en la información POST contra una llave guardada en la sesión del usuario. Por lo tanto, el componente de seguridad debe ser incluido tanto en el controlador que recibe la petición, como en el controlador que la hace.

Por ejemplo, si yo tengo una acción en PostController con una vista conteniendo un formulario que manda POST a una acción en CommentsController, entonces el componente de seguridad debe de ser incluido tanto en CommentsController (que esta recibiendo la petición, y de hecho protegiendo la acción), como en PostController (desde el cual la petición se originará).

Cada vez que el componente de seguridad es cargado, aún si no es usado para proteger una acción, realiza lo siguiente: Primero, genera una llave de autenticación usando la clase Security. Después, escribe esta llave en la sesión, junto con una fecha de expiración y alguna información adicional (la fecha de expiración es determinada por su configuración de seguridad de sesión en **/app/config/core.php**). En seguida, guarda la llave en su controlador, para ser usada luego.

Después en los archivos de su vista, cualquier etiqueta de formulario que genere usando $html→formTag() también tendra un campo escondido con la llave de autenticación. De esa manera, cuando el formulario es enviado mediante POST, el componente de seguridad puede comparar ese valor con el valor en la sesión donde se recibe la petición. En seguida, la llave de autenticación es regenerada y la sesión es actualizada para la siguiente petición.

View Caching

Sección 1

Que es esto?

Desde la versión 0.10.9.2378_final, Cake ha soportado cacheo para las vistas (También llamado Cacheo a Página Completa). No, no se trara de una broma. Ahora es posible cachear layouts y vistas. También es posible marcar partes de las vistas para que sean ignoradas por el mecanismo de cache. Esta funcionalidad, si es usada sabiamente, puede incrementar la velocidad de la aplicación en una cantidad de tiempo considerable.

Cuando se solicita una URL, Cake primero busca si la URL solicitada no ha sido ya cacheada. Si es así, Cake sortea el dispatcher y devuelve lo ya renderizado, versión cacheada de la pagina. Si la página no está en la cache, Cake se comporta con normalidad.

Si se ha activado la funcionalidad de cacheo, Cake almacenará la salida de su operación normal en la cache para usuarios futuros. La próxima vez que la página sea solicitada, Cake la recogerá de la cache. Genial, eh? Vamos a indagar para ver como funciona.

Sección 2

Como fuciona?

Activando la cache

Por defecto, el view caching está deshabilitado. Para activarlo, primero es necesario cambiar el valor de CACHE_CHECK en /app/config/core.php de false a true:

/app/config/core.php (parcial)

```
define ('CACHE_CHECK', true);
```

Esta linea le dice a Cake que se quiere habilitar el View Caching.

En el controlador para las vistas que se quieran cachear se tiene que añadir el Cache helper a el array helpers:

```
var $helpers = array('Cache');
```

A continuación, Se necesitará especificar lo que se quiere cachear.

Variable del controlador $cacheAction

En esta sección, Se mostrará como decirle a Cake lo que cachear. Esto se realiza inicializando una variable del controlador llamada $cacheAction. La variable $cacheAction debería ser inicializada por un array que contenga las acciones que se quiere que sean cacheadas, y el tiempo (en segundos) que se quiere que se mantengan esos datos. El valor del tiempo también puede ser un string amigable a strtotime() (p.e. '1 day' ó '60 segundos').

Digamos que se tiene un ProductsController, con algunas cosas que nosotros queremos que sean cacheadas. Los siguientes ejemplos muestran como usar $cacheAction para decirle a Cake que cachee ciertas partes de las acciones del controlador.

$cacheAction Ejemplos

```
//Cacheamos un grupo de páginas de los productos que son visitados más
a menudo, por un tiempo de seis horas:
var $cacheAction = array(
    'view/23/'  => 21600,
    'view/48/'  => 21600
);

//Cacheamos una acción entera. En este caso la lista de productos
denominada recalled, por un tiempo de seis horas:
var $cacheAction = array('recalled/' => 86400);

//Si se quisiera también, se podría cachear cada acción mediante la
asignación de una cadena
//que es amigable a strtotime() para indicar el tiempo de cacheo.
var $cacheAction = "1 hour";

//También es posible definir el cacheo en las propias acciones usando
$this->cacheAction = array()...
```

Marcando Contenido en la Vista

Hay instancias donde puede ser que no se quiera que partes de una vista se cacheen. Si se tiene alguna cierta clase de elemento destacando nuevos productos, se querría decirle a Cake que cache la vista… excepto una pequeña parte.

Es posible decirle a Cake que no cachee el contenido en las vistas mediante el uso de las etiquetas <cake:nocache> </cake:nocache> envolviendo al contenido que se quiere que el motor de cache salte.

```
<cake:nocache> example
<h1> New Products! </h1>
<cake:nocache>
<ul>
<?php foreach($newProducts as $product): ?>
<li>$product['name']</li>
<?endforeach;?>
</ul>
</cake:nocache>
```

Limpiando la cache

Primero, es necesario sbaer que Cake limpiará automáticamente la cache si se realiza un cambio en la base de datos. Por ejemplo, si una de las vistas usa información desde un modelo, y se ha producido un INSERT, UPDATE, o DELETE por el modelo, Cake limpiará la cache para esa vista.

Pero podrían existir casos donde se querría limpiar específicos ficheros de cache. Para hacer esto Cake proporciona la función clearCache, la cual está globalmente disponible:

<cake:nocache> ejemplo

```
//Borrar todas las páginas cacheadas que tengan el nombre controlador
clearCache('controlador');

//Borrar todas las páginas cacheadas que tengan el nombre
controlador_accion
clearCache('controlador_accion/');

//Borrar todas las páginas cacheadas que tengan el nombre
controlador_accion_parametros
//Nota: se pueden tener múltiples parámentros.
clearCache('controlador_accion_parametros');

//Tu puedes también usar un array para limpiar múltiples caches en una
vez.
clearCache(array('controlador_accion_parametros','controlador2_accion_
parametros'));
```

Sección 3

Cosas para recordar

A continuación están un grupo de cosas para recordar sobre View Caching:

1. Para habilitar el cache se debe asignar CACHE_CHECK a true en /app/config/core.php
2. En el controlador para las vistas que se desea cachear, se tiene que añadir el Cache helper en el array helpers.
3. Para cachear ciertas URL, use $cacheAction en el controlador.
4. Para saltar ciertas partes de una vista de ser cacheadas, se deben envolver con las etiquetas <cake:nocache> </cake:nocache>
5. Cake limpiar automáticamente copias específicas de cache cuando se realizan cambios en la base de datos.
6. Para realizar una limpieza manual de partes de la cache, use clearCache().

Ejemplo: Autenticación simple de usuarios

El gran escenario

Si eres nuevo en CakePHP, estarás fuertemente tentado a copiar y pegar este código para utilizarlo en tu aplicación en producción de manejo de datos sensibles y de misión crítica. Resiste: este capítulo es una exposición del funcionamiento interno de Cake, no sobre seguridad de aplicaciones. Mientras dudo, proporcionaremos algunas extremadamente obvias trampas de seguridad, **el punto de este ejemplos es mostrarte**

el funcionamiento interno de Cake, y permitirte crear por ti mismo una entrada a prueba de balas para tu aplicación.

Cake tiene control de acceso a través de su incorporado motor de ACLs, pero que respecto a autenticación de usuarios y persistencia? Qué respecto a eso?

Bien, por ahora, hemos encontrado que los sistemas de autenticación de usuarios varían de una aplicación a otra. Algunas como contraseñas ofuscadas, otras como autenticación LDAP y casi cada aplicación tendrá modelos de usuarios que son ligeramente diferentes. Por ahora, te dejamos esta desición. Esto cambiará? No estamos seguros aún. Por ahora, pensamos que la sobrecarga por construir esto en el framework no lo amerita, porque crear tu propio sistema de autenticación de usuarios es fácil con Cake.

Necesitas solo tres cosas:

- Una forma para autenticar usuarios (usualmente hecha verificando la identidad del usuario con una combinación usuario/contraseña)
- Una forma para registrar persistentemente ese usuario mientras navega por tu aplicación (usualmente hecha con sesiones)
- Una forma de revisar si un usuario ha sido autenticado (también a menudo hecha interactuando con sesiones)

En este ejemplo, crearemos un sistema simple para autenticación de usuarios para un sistema de administración de clientes. Esta aplicación ficticia podría probablemente ser usada por una oficina para registrar información de contacto y notas relacionadas a clientes. Toda la funcionalidad del sistema será ubicada detrás de nuestro sistema de autenticación de usuarios excepto por pocos bare-bones, vistas públicas seguras que muestren únicamente los nombres y títulos de los clientes almacenados en el sistema.

Empezaremos por mostrarte como verificar usuarios que tratan de acceder al sistema. La información de los usuarios autenticados será almacenada en sesiones PHP usando el componente de Cake para sessiones. Una vez que tengamos la información del usuario en la sesión, ubicaremos revisiones en la aplicación para asegurar que los usuarios de la aplicación no acceden a lugares a los que no deberían.

Una cosa a notar - autenticación no es lo mismo que control de acceso. Todo en lo que estaremos después en este ejemplo es como ver si la gente es quien dice ser, y permitirles acceso básico a partes de la aplicación. Si deseas afinar este acceso, revisa el capítulo de Listas de Control de Acceso en Cake. Haremos notas sobre donde las ACLs podrían calzar, pero por ahora, concentrémonos en la autenticación simple de usuarios.

Debería además decir que esto no pretende servir como esquema de seguridad para aplicaciones. Solo queremos darte suficiente con que trabajar para que puedas construir aplicaciones a prueba de balas por tí mismo.

Autenticación y Persistencia

Primero, necesitamos una forma para almacenar información acerca de usuarios tratando de acceder nuestro sistema de administración de clientes. El sistema de

administración de clientes que estamos usando almacena información de usuarios en una tabla de base de datos que fue creada usando el siguiente SQL:

Tabla 'users', Base de Datos ficticia del Sistema de Administración de Clientes

```
CREATE TABLE `users` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(255) NOT NULL,
  `password` varchar(32) NOT NULL,
  `first_name` varchar(255) NOT NULL,
  `last_name` varchar(255) NOT NULL,
  PRIMARY KEY  (`id`)
)
```

Bastante simple, correcto? El Modelo de Cake para esta tabla puede ser sencillamente:

```
<?php
class User extends AppModel
{
    var $name = 'User';
}
?>
```

Lo primero que necesitaremos son una vista y una acción para iniciar sesión. Esto proveerá un camino para los usuarios de la aplicación que intenten iniciar sesión y un camino para que el sistema procese esa información para ver si a estos se les debería permitir el acceso al sistema ó no. La vista es únicamente un formulario HTML, creado con la ayuda del Ayudante Html de Cake.

/app/views/users/login/thtml

```
<?if ($error): ?>
<p>Las credenciales para inicio de sesión proporcionadas no pueden ser
reconocidas. Por favor intente nuevamente.</p>
<? endif; ?>

<form action="<?php echo $html->url('/users/login'); ?>"
method="post">
<div>
    <label for="username">Nombre de usuario:</label>
    <?php echo $html->input('User/username', array('size' => 20)); ?>
</div>
<div>
    <label for="password">Contraseña:</label>
    <?php echo $html->password('User/password', array('size' => 20));
?>
</div>
<div>
    <?php echo $html->submit('Login'); ?>
</div>
</form>
```

Esta vista presenta un formulario simple para inicio de sesión para usuarios que traten de acceder al sistema. La acción para el formulario **/users/login** , la cual está en UsersController se ve así:

```
class UsersController extends AppController
{
    function login()
    {
        //No mostrar el mensaje de error si no se han proporcionado
datos.
        $this->set('error', false);

        //Si un usuario ha proporcionado datos a través del
formulario.
        if (!empty($this->data))
        {
            //Primero, veamos si hay usuarios en la base de datos
            //con el nombre de usuario proporcionado por el usuario
            //utilizando el formulario:

            $someone = $this->User->findByUsername($this-
>data['User']['username']);

            //En este punto, $someone está lleno con datos del
usuario, o está vacío.
            //Comparemos la contraseña proporcionada a través del
formulario con la
            //almacenada en la base de datos.

            if(!empty($someone['User']['password']) &&
$someone['User']['password'] == $this->data['User']['password'])
            {

                //Nota: esperanzado en que las contraseñas estén
ofuscadas en la base de datos,
                //tu comparación podría verse más como:
                //md5($this->data['User']['password']) == ...

                //Esto significa que estas son iguales. Ahora podemos
construir alguna información básica de
                //sesión para recordar este usuario como con 'sesión-
iniciada'

                $this->Session->write('User', $someone['User']);

                //Ahora que tenemos a los usuarios registrados en una
sesión, redireccionémoslos a
                //la página de recibimiento para la aplicación.

                $this->redirect('/clients');
            }
            //Caso contrario, han provisto datos incorrectos:
            else
            {
                //Recuerdas la variable $error en la vista? Cambiemos
su valor a verdadero:

                $this->set('error', true);
            }
        }
    }

    function logout()
    {
```

```
        //Redirecciona usuarios a esta acción si hacen click en el
botón Cerrar Sesión.
        //Todo lo que necesitamos hacer aquí es destruir la
información de sesión.

        $this->Session->delete('User');

        //Y probablemente deberíamos redireccionarlos a alguna parte,
también...

        $this->redirect('/');
    }
}
?>
```

No tan mal: el contenido de la acción login() podría ser de menos de 20 líneas si fueras consiso. El resultado de esta acción es tanto 1: la información del usuario es registrada en la sesión y este es redireccionado a la página de bienvenida de la aplicación, o 2: regresar a la pantalla de inicio de sesión y presentar el formulario de inicio de sesión (con un mensaje de error adicional).

Revisando acceso en tu Aplicación

Ahora que podemos autenticar usuarios, hagamos que la aplicación heche usuarios que traten de ingresar al sistema desde puntos diferentes a la pantalla de inicio de sesión y el directorio “básico” de clientes que detallamos anteriormente.

Una forma de hacer esto es crear una función en AppController que hará la revisión de la sesión y echará por tí.

/app/app_controller.php

```
<?php
class AppController extends Controller
{
    function checkSession()
    {
        //Si la información de sesión no ha sido definida...
        if (!$this->Session->check('User'))
        {
            //Forza al usuario a iniciar sesión
            $this->redirect('/users/login');
            exit();
        }
    }
}
?>
```

Ahora tienes una función que puedes utilizar en cualquier controlador para asegurar que los usuarios no traten de acceder a acciones del controlador sin iniciar sesión primero. Una vez que esto está en su sitio puedes revisar el acceso en cualquier nivel - aquí algunos ejemplos:

Forzar autenticación antes de todas la acciones en un controlador

```
<?php
class NotesController extends AppController
{
    //No quieres usuarios no autenticados mirando cualquier acción
    //en este controlador? Utiliza beforeFilter para que Cake ejecute
checkSession
    //antes que cualquier acción.

    function beforeFilter()
    {
        $this->checkSession();
    }
}
?>
```

Forzar la autenticación antes de una acción en un controlador

```
<?php
class NotesController extends AppController
{
    function publicNotes($clientID)
    {
        //Acceso público a esta acción está bien ...
    }

    function edit($noteId)
    {
        //Pero solo quieres usuarios autenticados en esta acción.
        $this->checkSession();
    }
}
?>
```

Ahora que tienes las bases, podrías aventurarte por tu cuenta e implementar algunas características avanzadas ó a la medida sobre las que han sido delineadas aquí. La integración con el componente de ACLs de Cake podría ser un buen primer paso.

Convenciones de Cake

Convenciones, eh?

Si, convenciones. De acuerdo a thefreedictionary:

- Acuerdo general en, o de ciertas prácticas ó actitudes: Por convención, el norte está arriba en la mayoría de mapas.
- Una práctica ó procedimiento ampliamente observada en un grupo, especialmente para facilitar la interacción social; una costumbre: la convención de dar la mano para saludar.
- Un dispositivo ó técnica ampliamente usada y aceptada, tanto en drama, literatura ó pintura: la convención teatral del aparte.

Las convenciones en cake son lo que hace que la magia ocurra, léelo **auto-mágico**. No es necesario decir que favoreciendo las convenciones sobre la configuración, Cake incrementa tu productividad a un temible nivel sin ninguna pérdida de flexibilidad. Las convenciones en cake son realmente simples e intuitivas. Estas son extraidas de las mejores prácticas que buenos desarrolladores web han utilizado a través de años en el campo del desarrollo web.

Nombres de archivos

Los nombres de archivos son con **guiones bajos**. Como regla general, si tienes una clase **MyNiftyClass**, entonces en Cake, su archivo debería ser nombrado my_nifty_class.php

Si encuentras un snippet automáticamente sabrás que:

- Si este es un Controlador nombrado **KissesAndHugsController**, el nombre de su archivo deberá ser **kisses_and_hugs_controller.php** (observa _controller en el nombre del archivo)
- Si este es un Modelo nombrado **OptionValue**, el nombre de su archivo deberá ser **option_value.php**
- Si este es un Componente nombrado **MyHandyComponent**, el nombre de su archivo deberá ser **my_handy.php** (no se necesita _component en el nombre del archivo)
- Si este es un Ayudante nombrado **BestHelperEver**, el nombre de su archivo deberá ser **best_helper_ever.php**

Modelos

- Los nombres de las clases Modelo son **singulares**.
- Los nombres de las clases Modelo empiezan con mayúscula, para modelos con varias palabras se emplea nomenclatura Camel mayúscula.
    - Ejemplos: Person, Monkey, GlassDoor, LineItem, ReallyNiftyThing
- Las tablas de juntura de relaciones muchos a muchos debería ser nombradas: alfabéticamente_primera_tabla_plural_alfabéticamente_segunda_tabla_plural pe: etiquetas_usuarios
- Los nombres de archivos de modelos usan minúsculas y guiones bajos.
    - Ejemplos: person.php, monkey.php, glass_door.php, line_item.php, cosa_realmente_elegante.php
- Las tablas de la base de datos relacionadas a los modelos también usan minúsculas y guiones bajos, pero estas están en **plural**.
    - Ejemplos: people, monkey, glass_doors, line_items, really_nifty_things

Las convenciones de nombres en CakePHP buscan guiar la creación de código y hacer el mismo más fácil de leer. Si estas están en tu camino, puedes sobreescribirlas.

- Nombre de modelo: Set `var $name` en tu definición de modelo
- Tabla de base de datos relacionada a tu modelo: Set `var $useTable` en tu definición de modelo.

Controladores

- Los nombres de las clases Controlador son en **plural**.
- Los nombres de las clases Controlador empienzan con mayúscula, para controladores con varias palabras se emplea nomenclatura Camel mayúscula. También los nombres de clases Controlador terminan con 'Controller'.

- Ejemplos: PeopleController, MonkeysController, GlassDoorsController, LineItemsController, ReallyNiftyThingsController
- Los nombres de archivo de controladores, usan minúsculas y guiones bajos. También los nombres de archivo de controladores terminan con '_controller'. Si tienes una clase controladora llamada PostsController, el nombre del archivo debería ser posts_controller.php
    - Ejemplos: people_controller.php, line_items_controller.php, really_nifty_things_controller.php
- Para visibilidad protegida, a los nombres de las acciones del controlador se deberá anteponer '-'.
- Para visibilidad privada, a los nombres de las acciones del controlador se deberá anteponer '–'.

Vistas

- Las vistas son nombradas como las acciones que presentan.
- Los nombres de archivos de vistas son como los nombres de las acciones, en minúsculas.
    - Ejemplos: `PeopleController::worldPeace()` espera una vista en /app/views/people/world_peace.thtml; `MonkeysController::banana()` espera una vista en /app/views/monkeys/banana.thtml.

Puedes forzar a una acción a presentar (render) una vista específica llamando `$this→render`('nombre_del_archivo_de_la_vista_sin_punto_thtml'); al final de tu acción.

Ayudantes

- El nombre de la clase de un ayudante usa nomenclatura Camel y termina en 'Helper', el nombre del archivo utiliza guiones bajos.
    - Ejemplo: `class MyHelperHelper extends Helper` está en /app/views/helpers/my_helper.php.

Se incluye en el controlador con `var $helpers = array('Html', 'MyHelper');` en la vista puedes acceder con `$myHelper→method()`.

Componentes

- El nombre de una clase componente usa nomenclatura Camel y termina en 'Component', el nombre del archivo utiliza guiones bajos.
    - Ejemplo: `class MyComponentComponent extends Objects` está en /app/controller/components/my_component.php

Se incluye en el controlador con `var $components = array('MyComponent');` en el controlador puedes accecer con `$this→MyComponent→method()`.

Vendedores

Los vendedores no siguen ninguna convención por obvias razones: son piezas de código de terceros. Cake no tiene control sobre estas.