CakePHP Book

Versión 5.next

Cake Software Foundation

08 de septiembre de 2024

Índice general

CAPÍTULO 1

CakePHP de un vistazo

CakePHP está diseñado para hacer tareas habituales de desarrollo web simples y fáciles. Proporciona una caja de herramientas todo-en-uno y para que puedas empezar rápidamente, las diferentes partes de CakePHP trabajan correctamente de manera conjunta o separada.

El objetivo de este artículo es introducirte en los conceptos generales de CakePHP y darte un rápido vistazo sobre como esos conceptos están implementados en CakePHP. Si estás deseando comenzar un proyecto puedes *empezar con el tutorial*, o profundizar en la documentación.

Convenciones sobre configuración

CakePHP proporciona una estructura organizativa básica que cubre los nombres de las clases, archivos, tablas de base de datos y otras convenciones más. Aunque lleva algo de tiempo aprender las convenciones, siguiéndolas CakePHP evitará que tengas que hacer configuraciones innecesarias y hará que la estructura de la aplicación sea uniforme y que el trabajo con varios proyectos sea sencillo. El capítulo de *convenciones* muestra las que son utilizadas en CakePHP.

La capa Modelo

La capa Modelo representa la parte de tu aplicación que implementa la lógica de negocio. Es la responsable de obtener datos y convertirlos en los conceptos que utiliza tu aplicación. Esto incluye procesar, validar, asociar u otras tareas relacionadas con el manejo de datos.

En el caso de una red social la capa modelo se encargaría de tareas como guardar los datos del usuario, las asociaciones de amigos, almacenar y obtener fotos, buscar sugerencias de amistad, etc. Los objetos modelo serían «Amigo», «Usuario», «Comentario» o «Foto». Si quisiéramos obtener más datos de nuestra tabla `usuarios` podríamos hacer lo siguiente:

```php
use Cake\ORM\Locator\LocatorAwareTrait;

$usuarios = $this->getTableLocator()->get('Usuarios');
$resultset = $usuarios->find()->all();
foreach ($resultset as $row) {
    echo $row->nombreusuario;
}
```

Como te habrás dado cuenta no hemos necesitado escribir ningún código previo para empezar a trabajar con nuestros datos. Al utilizar las convenciones CakePHP usará clases estándar para tablas y clases de entidad que no hayan sido definidas.

Si queremos crear un nuevo usuario y guardarlo (con validaciones) podríamos hacer algo como:

```php
use Cake\ORM\Locator\LocatorAwareTrait;

$usuarios = $this->getTableLocator()->get('Usuarios');
$usuario = $usuarios->newEntity(['email' => 'mark@example.com']);
$usuarios->save($usuario);
```

La capa Vista

La capa Vista renderiza una presentación de datos modelados. Separada de los objetos Modelo, es la responsable de usar la información disponible para producir cualquier interfaz de presentación que pueda necesitar tu aplicación.

Por ejemplo, la vista podría usar datos del modelo para renderizar una plantilla HTML que los contenga o un resultado en formato XML:

```php
// En un archivo de plantilla de vista renderizaremos un 'element' para cada usuario.
<?php foreach ($usuarios as $usuario): ?>
    <li class="usuario">
        <?= $this->element('usuario', ['usuario' => $usuario]) ?>
    </li>
<?php endforeach; ?>
```

La capa Vista proporciona varias extensiones como *Plantillas de Vista*, *Elementos* y *View Cells* que te permiten reutilizar tu lógica de presentación.

Esta capa no se limita a representaciones HTML o texto de los datos. Puede utilizarse para otros formatos habituales como JSON, XML y a través de una arquitectura modular, cualquier otro formato que puedas necesitar como CSV.

La capa Controlador

La capa Controlador maneja peticiones de usuarios. Es la responsable de elaborar una respuesta con la ayuda de las capas Modelo y Vista.

Un controlador puede verse como un gestor que asegura que todos los recursos necesarios para completar una tarea son delegados a los trabajadores oportunos. Espera por las peticiones de los clientes, comprueba la validez de acuerdo con las reglas de autenticación y autorización, delega la búsqueda o procesado de datos al modelo, selecciona el tipo de presentación que el cliente acepta y finalmente delega el proceso de renderizado a la capa Vista. Un ejemplo de controlador para el registro de un usuario sería:

```php
public function add()
{
    $usuario = $this->Usuarios->newEmptyEntity();
    if ($this->request->is('post')) {
        $usuario = $this->Usuarios->patchEntity($usuario, $this->request->getData());
        if ($this->Usuarios->save($usuario, ['validate' => 'registration'])) {
            $this->Flash->success(__('Ahora estás registrado.'));
        } else {
            $this->Flash->error(__('Hubo algunos problemas.'));
        }
    }
    $this->set('usuario', $usuario);
}
```

Puedes fijarte en que nunca renderizamos una vista explícitamente. Las convenciones de CakePHP se harán cargo de seleccionar la vista correcta y de renderizarla con los datos que preparemos con `set()`.

Ciclo de una petición CakePHP

Ahora que te has familiarizado con las diferentes capas en CakePHP, revisemos como funciona el ciclo de una petición:

El ciclo de petición típico de CakePHP comienza con un usuario solicitando una página o recurso en tu aplicación. A un alto nivel cada petición sigue los siguientes pasos:

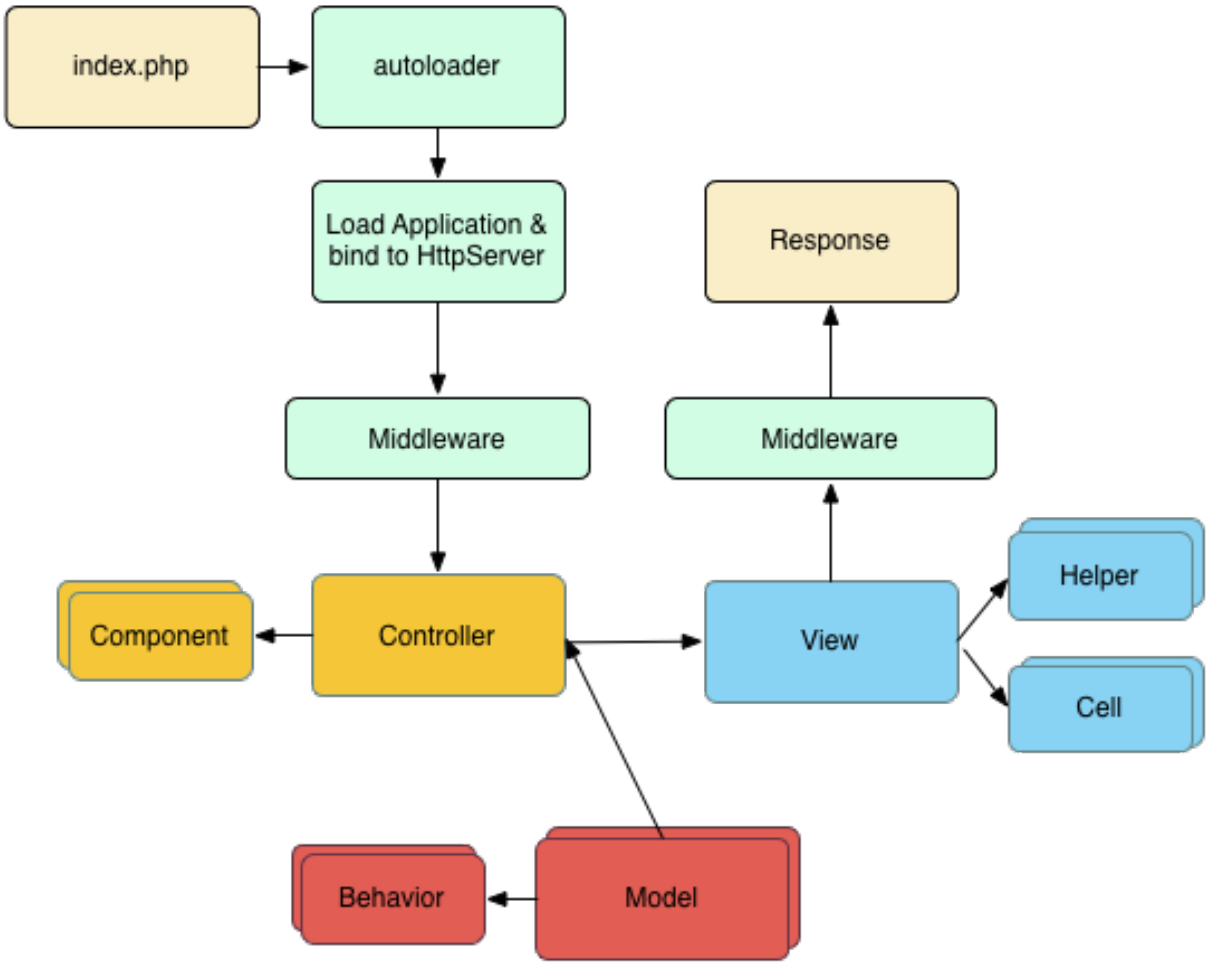
1. Las reglas de rescritura del servidor web envían la petición a **webroot/index.php**.
2. Tu aplicación es cargada y ligada a un `HttpServer`.
3. Se inicializa el `midleware` de tu aplicación.
4. Una petición y respuesta son precesadas a través del `Middleware PSR-7` que tu aplicación utiliza. Normalmente, esto incluye la captura de errores y enrutamiento.
5. Si no recibe ninguna respuesta del `middleware` y la petición contiene información de enrutamiento, se selecciona un controlador y una acción.
6. La acción del controlador es ejecutada y el controlador interactúa con los Modelos y Componentes necesarios.
7. El controlador delega la creación de la respuesta a la Vista para generar la salida a partir de los datos del modelo.
8. La vista utiliza `Helpers` y `Cells` para generar el cuerpo y las cabeceras de la respuesta.
9. La respuesta es devuelta a través del /controllers/middleware.
10. El `HttpServer` envía la respuesta al servidor web.

Esto es solo el comienzo

Ojalá este repaso rápido haya despertado tu curiosidad. Otras funcionalidades geniales de CakePHP son:

- Un *framework para caché* que se integra con Memcached, Redis y otros métodos de caché.
- Poderosas herramientas de generación de código para que puedas comenzar inmediatamente.
- *Framework para la ejecución de pruebas integrado* para que puedas asegurarte de que tu código funciona perfectamente.

Los siguientes pasos obvios son *descargar CakePHP* y leer el *tutorial y crear algo asombroso*.


index.php
autoloader
Load Application & bind to HttpServer
Middleware
Controller
Component
Behavior
Model
View
Helper
Cell
Middleware
Response

Lecturas complementarias

Donde obtener ayuda

La página oficial de CakePHP

https://cakephp.org

La página oficial de CakePHP es siempre un gran lugar para visitar. Proporciona enlaces a las herramientas más utilizadas por desarrolladores, `screencasts`, oportunidades para hacer una donación y descargas.

El Cookbook

https://book.cakephp.org

Este manual probablemente debería ser el primer lugar al que debas acudir para obtener respuestas. Como muchos otros proyectos de código libre, nuevos colaboradores se unen regularmente. Intenta encontrar por ti mismo las respuestas a tus preguntas primero, puede que así tardes más en encontrar las respuestas, pero permanecerán durante más tiempo - y además aliviarás nuestra carga de soporte. Tanto el manual como la API tienen una versión online.

La Bakery

https://bakery.cakephp.org

La «panadería» (`bakery`) de CakePHP es un lugar de intercambio para todo lo relacionado con CakePHP. Consúltala para tutoriales, casos de estudio y ejemplos de código. Cuando estés familiarizado con CakePHP, accede y comparte tus conocimientos con la comunidad y gana fortuna y fama de forma instantánea.

La API

https://api.cakephp.org/

Directo al punto y directo para desarrolladores del núcleo de CakePHP, la API (`Application Programming Interface`) es la documentación más completa para todos los detalles esenciales del funcionamiento interno del `framework`. Es referencia directa al código, asi que trae tu sombrero de hélice.

Los casos de prueba

Si crees que la información proporcionada en la API no es suficiente, comprueba el código de los casos de prueba proporcionados con CakePHP. Pueden servirte como ejemplos prácticos de funciones y datos de una clase.

```
tests/TestCase/
```

El canal IRC

Canales IRC en irc.freenode.net:

- #cakephp – Discusión general
- #cakephp-docs – Documentación
- #cakephp-bakery – Bakery
- #cakephp-fr – Canal francés.

Si estás atascado, péganos un grito en el canal IRC de CakePHP. Alguien del equipo de desarrollo[4] está normalmente, especialmente durante las horas de día para usuarios de América del Norte y del Sur. Estaremos encantados de escucharte, tanto si necesitas ayuda como si quieres encontrar usuarios en tu zona o si quieres donar tu nuevo coche deportivo de marca.

Foro oficial de CakePHP

Foro oficial de CakePHP[5]

Nuestro foro oficial donde puedes pedir ayuda, sugerir ideas y conversar sobre CakePHP. Es un lugar perfecto para encontrar rápidamente respuestas y ayudar a otros. Únete a la familia CakePHP registrándote.

Stackoverflow

https://stackoverflow.com/[6]

Etiqueta tus preguntas con `cakephp` y la versión específica que utilizas para permitir encontrar a los usuarios de stackoverflow tus preguntas.

Donde encontrar ayuda en tu idioma

Portugúes de Brasil

- Comunidad brasileña de CakePHP[7]

Danés

- Canal danés de CakePHP en Slack[8]

[4] https://cakephp.org/team
[5] https://discourse.cakephp.org
[6] https://stackoverflow.com/questions/tagged/cakephp/
[7] https://cakephp-br.org
[8] https://cakesf.slack.com/messages/denmark/

Francés

- Comunidad francesa de CakePHP[9]

Alemán

- Canal alemán de CakePHP en Slack[10]
- Grupo alemán de CakePHP en Facebook[11]

Iraní

- Comunidad iraní de CakePHP[12]

Holandés

- Canal holandés de CakePHP en Slack[13]

Japonés

- Canal japonés de CakePHP en Slack[14]
- Grupo japonés de CakePHP en Facebook[15]

Portugués

- Grupo portugés de CakePHP en Google[16]

Español

- Canal español de CakePHP en Slack[17]
- Canal IRC de CakePHP en español
- Grupo español de CakePHP en Google[18]

[9] https://cakephp-fr.org
[10] https://cakesf.slack.com/messages/german/
[11] https://www.facebook.com/groups/146324018754907/
[12] https://cakephp.ir
[13] https://cakesf.slack.com/messages/netherlands/
[14] https://cakesf.slack.com/messages/japanese/
[15] https://www.facebook.com/groups/304490963004377/
[16] https://groups.google.com/group/cakephp-pt
[17] https://cakesf.slack.com/messages/spanish/
[18] https://groups.google.com/group/cakephp-esp

Convenciones CakePHP

Somos muy fans de la convención por encima de la configuración. A pesar de que toma algo de tiempo aprender las convenciones de CakePHP, ahorrarás tiempo a la larga. Siguiendo las convenciones obtendrás funcionalidades gratuitas y te liberarás de la pesadilla de mantener archivos de configuración. Las convenciones también hacen que el desarrollo sea uniforme, permitiendo a otros desarrolladores intervenir y ayudar fácilmente.

Convenciones de Controlador

Los nombres de las clases Controlador son en plurar, en formato `CamelCase`, y finalizan con `Controller`. Ejemplos de nombres son: `UsuariosController` y `CategoriasArticulosController`.

Los métodos públicos de los Controladores a menudo se exponen como "acciones" accesibles a través de un navegador web. Tienen formato `camelBacked`. Por ejemplo, `/users/view-me` mapea al método `viewMe()` de `UsersController` sin tener que hacer nada en el enrutamiento de la aplicación. Los métodos protegidos o privados no son accesibles con el enrutamiento.

Consideraciones URL para los nombres de Controladores

Como acabas de ver, los controladores de una sola palabra mapean a una dirección URL en minúscula. Por ejemplo: a `UsuariosController` (que debería estar definido en **UsuariosController.php**) se puede acceder desde http://example.com/usuarios.

Aunque puedes enrutar controladores de múltiples palabaras de la forma que desees, la convención es que tus URLs separen las palabras con guiones utilizando la clase `DashedRoute`, de este modo `/categorias-articulos/ver-todas` es la forma correcta para acceder a la acción `CategoriasArticulosController::verTodas()`.

Cuando creas enlaces utilizando `this->Html->link()` puedes utilizar las siguientes convenciones para el array url:

```
$this->Html->link('titulo-enlace', [
    'prefix' => 'MiPrefijo' // CamelCase
    'plugin' => 'MiPlugin', // CamelCase
    'controller' => 'NombreControlador', // CamelCase
    'action' => 'nombreAccion' // camelBack
]
```

Para más información sobre URLs de CakePHP y el manejo de sus parámetros puedes consultar *Conectando Rutas*.

Convenciones de nombre de clase y archivo

En general, los nombres de los archivos coinciden con los nombres de las clases y sigue el estándar PSR-4 para cargarse automáticamente. Los siguientes son ejemplos de nombres de clases y de sus archivos:

- La clase Controlador `LatestArticlesController` debería estar en un archivo llamado **LatestArticlesController.php**
- La clase Componente `MyHandyComponent` debería estar en un archivo llamado **MyHandyComponent.php**
- La clase Tabla `OptionValuesTable` debería estar en un archivo llamado **OptionValuesTable.php**.
- La clase Entidad `OptionValue` debería estar en un archivo llamado **OptionValue.php**.
- La clase Behavior `EspeciallyFunkableBehavior` debería estar en un archivo llamado **EspeciallyFunkableBehavior.php**
- La clase Vista `SuperSimpleView` debería estar en un archivo llamado **SuperSimpleView.php**

- La clase Helper `BestEverHelper` debería estar en un archivo llamado **BestEverHelper.php**

Cada archivo deberá estar ubicado en la carpeta/namespace correcta dentro de tu carpeta de tu aplicación.

Convenciones de base de datos

Los nombres de las tablas correspondientes a los modelos de CakePHP son en plural y con con "_". Por ejemplo `users`, `menu_links` y `user_favorite_pages` respectivamente. Los nombres de tablas formadas por múltiples palabras sólo deben usar el plural en la última palabra, por ejemplo, `menu_links`.

Los nombres de campos con dos o más palabras se escriben con "_", por ejemplo: `first_name`.

Las claves foráneas en relaciones `1-n` (`hasMany`) y `1-1` (`belongsTo/hasOne`) son reconocidas por defecto mediante el nombre (en singular) de la tabla relacionada seguido de `_id`. De este modo si `Users` tiene varios `Articles` (relación `hasMany`), la tabla `articles` se relacionará con la tabla `users` a través de la clave foránea `user_id`. Para una tabla como `menu_links` cuyo nombre está formado por varias palabras, la clave foránea sería `menu_link_id`.

Las tablas de unión, usadas en las relaciones `n-n` (`BelongsToMany`) entre modelos, deberían ser nombradas después de las tablas que unirán. Los nombres deberán estar en plural y en orden alfabético: `articles_tags` en lugar de `tags_articles` o `article_tags`. *El comando ``bake``no funcionará correctamente si ésta convención no se sigue.* Si la tabla de unión guarda alguna información que no sean las claves foráneas, debes crear la clase de la entidad y modelo para esa tabla.

Además de utilizar claves auto-incrementales como claves primarias, también puedes utilizar columnas UUID. CakePHP creará un único UUID de 36 caracteres (*Cake\Utility\Text::uuid()*) cada vez que guardes un nuevo registro usando el método `Table::save()` .

Convenciones de modelo

Los nombres de las clases para las tablas son en plural, formato `CamelCase` y terminan en `Table`. `UsersTable`, `MenuLinksTable` y `UserFavoritePagesTable` son ejemplos de nombres de clases que corresponden a las tablas `users`, `menu_links` y `user_favorite_pages` respectivamente.

Los nombres de las clases para las entidades son en singular, formato `CamelCase` y no tienen sufijo. `User`, `MenuLink` y `UserFavoritePage` son ejemplos de nombres de clases que corresponden a las entidades `users`, `menu_links` y `user_favorite_pages` respectivamente.

Convenciones de vistas

Los archivos de las plantillas de vistas son nombrados según las funciones de controlador que las muestran empleando "_". La función `viewAll()` de la clase `ArticlesController` mostrará la vista **templates/Articles/view_all.php**.

El patrón base es **templates/Controller/nombre_funcion.php**.

Nota: Por defecto CakePHP usa palabras en inglés para las convenciones de nombres. Si utilizas otro idioma CakePHP puede que no sea capaz de procesar correctamente las conversiones (de singular a plural y viceversa). Si necesitas añadir reglas para tu idioma para algunas palabras, puedes utilizar la clase *Cake\Utility\Inflector*. Además de definir tus reglas de conversión personalizadas, esta clase te permite comprobar que CakePHP comprenda tu sintaxis personalizada para palabras en plural y singular. Mira la documentación sobre *Inflector* para más información.

Convenciones de plugins

Es útil añadir el prefijo «cakephp-» en el nombre del paquete para los plugins de CakePHP. Esto hace que el nombre esté relacionado semánticamente al «framework» del que depende.

No uses el espacio de nombre de CakePHP (cakephp) como nombre de «vendor», ya que es un espacio reservado para los plugins que son propiedad de CakePHP. La convención es usar letras en minúscula y guiones como separadores:

```
// Bad
cakephp/foo-bar

// Good
your-name/cakephp-foo-bar
```

Ver lista asombrosa de recomendaciones[19] par mas detalles.

Resumen

Nombrando los elementos de tu aplicación empleando las convenciones de CakePHP ganarás funcionalidad sin los fastidios y ataduras de mantenimiento de la configuración.

Un último ejemplo que enlaza todas las convenciones:

- Tabla de base de datos: «articles», «menu_links»
- Clase Tabla: `ArticlesTable`, ubicada en **src/Model/Table/ArticlesTable.php**
- Clase Entidad: `Article`, ubicada en **src/Model/Entity/Article.php**
- Clase Controlador: `ArticlesController`, ubicada en **src/Controller/ArticlesController.php**
- Plantilla vista, ubicada en **templates/Articles/index.php**

Usando estas convenciones, CakePHP sabe que una petición a http://example.com/articles/ hace una llamada a la función `index()` de la clase `ArticlesController`, donde el modelo `Articles` está disponible automáticamente y enlazada automáticamente a la tabla `articles` en la base de datos . Ninguna de estas relaciones han sido configuradas de ningún modo salvo creando clases y archivos que has tenido que crear de todas formas.

[19] https://github.com/FriendsOfCake/awesome-cakephp/blob/master/CONTRIBUTING.md#tips-for-creating-cakephp-plugins

Ejemplo	articles	menu_links	
Tabla base de datos	articles	menu_links	Nombres de tablas que se corresponden a modelos son en plural y con guión bajo "_".
Archivo	ArticlesController.php	MenuLinksController.php	
Tabla	ArticlesTable.php	MenuLinksTable.php	Los nombres de clase de las tablas son en plural, formato "CamelCased" y acaban con el sufijo "Table"
Entidad	Article.php	MenuLink.php	Los nombres de clase de las entidades son en singular y "CamelCased": Article and MenuLink
Clase	ArticlesController	MenuLinksController	
Controlador	ArticlesController	MenuLinksController	Plural, CamelCased, acaba en "Controller"
Plantillas de vistas	Articles/index.php Articles/add.php Articles/edit.php	MenuLinks/index.php MenuLinks/add.php MenuLinks/add.php	Los archivos de plantillas de vistas son nombrados según las funciones que el controlador muestra, en minúscula y guión bajo
Comportamiento	ArticlesBehavior.php	MenuLinksBehavior.php	
Vista	ArticlesView.php	MenuLinksView.php	
Ayudante	ArticlesHelper.php	MenuLinksHelper.php	
Componente	ArticlesComponent.php	MenuLinksComponent.php	
Plugin	Mal: cakephp/articles Bien: you/cakephp-articles	cakephp/menu-links you/cakephp-menu-links	Útil añadir el prefijo «cakephp-» a los plugins en el nombre del paquete. No uses el espacio de nombre (cakephp) como nombre de vendor ya que está para los plugins propiedad de CakePHP. La convención es usar letras minúsculas y guiones como separadores.
Cada fichero estará localizado en la "carpeta/espacio de nombre" apropiado dentro de la carpeta de tu aplicación.			

Database Convention Summary

Claves foráneas hasMany belongsTo/ hasOne BelongsToMany	Las relaciones son reconocidas por defecto como el nombre (singular) de la tabla relacionada, seguida de `_id`. Para Users "hasMany" Articles, la tabla `articles` hará referencia a `users` a través de la clave foránea `user_id`.
Múltiples palabras	`menu_links` cuyo nombre contiene múltiples palabras, su clave foránea será `menu_link_id`.
Auto Increment	Además de utilizar claves auto-incrementales como claves primarias, también puedes utilizar columnas UUID. CakePHP creará un único UUID de 36 caracteres usando (*`Cake\Utility\Text::uuid()`*) cada vez que guardes un nuevo registro usando el método `Table::save()` .
Join tables	Deberán ser nombradas según las tablas que unirán o el comando de "bake" no funcionará y ordenarse alfabéticamente (`articles_tags` en vez de `tags_articles`). Si tiene campos adicionales que guardan información, debes crear un archivo de entidad y modelo para esa tabla.

Ahora que te has introducido en los fundamentos de CakePHP. puedes tratar de realizar el tutorial *Tutorial Gestor de Contenidos* para ver como las cosas encajan juntas.

Estructura de carpetas de CakePHP

Después de haber descargado el esqueleto de aplicación de CakePHP, estos son los directorios de primer nivel que deberías ver:

- La carpeta *bin* contiene los ejecutables por consola de Cake.
- La carpeta *config* contiene los documentos de *Configuración* que utiliza CakePHP. Detalles de la conexión a la Base de Datos, bootstrapping, archivos de configuración del core y otros, serán almacenados aquí.
- La carpeta *plugins* es donde se almacenan los *Plugins* que utiliza tu aplicación.
- La carpeta de *logs* contiene normalmente tus archivos de log, dependiendo de tu configuración de log.
- La carpeta *src* será donde tu crearás tu magia: es donde se almacenarán los archivos de tu aplicación.
- La carpeta *templates* contiene los archivos de presentación: elementos, páginas de error, plantillas generales y plantillas de vistas.
- La carpeta *resources* contiene sub carpetas para varios tipos de archivos.
- La carpeta *locales* contiene sub carpetas para los archivos de traducción a otros idiomas.
- La carpeta *tests* será donde pondrás los test para tu aplicación.
- La carpeta *tmp* es donde CakePHP almacenará temporalmente la información. La información actual que almacenará dependerá de cómo se configure CakePHP, pero esta carpeta es normalmente utilizada para almacenar descripciones de modelos y a veces información de sesión.
- La carpeta *vendor* es donde CakePHP y otras dependencias de la aplicación serán instaladas por Composer[20]. Editar estos archivos no es recomendado, ya que Composer sobreescribirá tus cambios en la próxima actualización.
- El directorio *webroot* es la raíz de los documentos públicos de tu aplicación. Contiene todos los archivos que quieres que sean accesibles públicamente.

[20] https://getcomposer.org

Asegúrate de que las carpetas *tmp* y *logs* existen y permiten escritura, en caso contrario el rendimiento de tu aplicación se verá gravemente perjudicado. En modo debug, CakePHP te avisará si este no es el caso.

La carpeta src

La carpeta *src* de CakePHP es donde tú harás la mayor parte del desarrollo de tu aplicación. Observemos más detenidamente dentro de la carpeta *src*.

Command
Contiene los comandos de consola de tu aplicación. Para más información mirar /console-commands/commands.

Console
Contiene los "scripts" de instalación ejecutados por Composer.

Controller
Contiene los *Controladores* de tu aplicación y sus componentes.

Middleware
Contiene cualquier /controllers/middleware para tu aplicación.

Model
Contiene las tablas, entidades y comportamientos de tu aplicación.

View
Las clases de presentación se ubican aquí: plantillas de vistas, células y ayudantes.

Nota: La carpeta `Command` no está creada por defecto. Puedes añadirla cuando la necesites.

CAPÍTULO 2

Guía de inicio rápido

La mejor forma de experimentar y aprender CakePHP es sentarse y construir algo.

Para empezar crearemos una sencilla aplicación para guardar favoritos.

Tutorial Gestor de Contenidos

Este tutorial lo guiará a través de la creación de un CMS (Sistema de Gestión de Contenidos) simple. Para empezar, instalaremos CakePHP, creando nuestra base de datos y construyendo una gestión simple de artículos.

Esto es lo que se necesitará:

1. Un servidor de base de datos. Vamos a utilizar el servidor MySQL en este tutorial. Necesitará saber lo suficiente sobre SQL para crear una base de datos y ejecutar fragmentos SQL del tutorial. CakePHP se encargará de construir todas las consultas que su aplicación necesita. Como estamos usando MySQL, también asegúrese de tener `pdo_mysql` habilitado en PHP.
2. Conocimientos básicos de PHP.

Antes de comenzar, debe asegurarse de tener una versión de PHP actualizada:

```
php -v
```

Al menos debería haber instalado PHP 8.1 (CLI) o superior. La versión PHP de su servidor web también debe ser de 8.1 o superior, y debería ser la misma versión que su interfaz de línea de comando (CLI) de PHP.

Obteniendo CakePHP

La forma más fácil de instalar CakePHP es usar Composer. Composer es una manera simple de instalar CakePHP desde su terminal o línea de comandos. Primero, necesita descargar e instalar Composer si aún no lo ha hecho. Si tiene cURL instalado, es tan fácil como ejecutar lo siguiente:

```
curl -s https://getcomposer.org/installer | php
```

O, puede descargar `composer.phar` desde el sitio web de Composer[21].

Luego simplemente escriba la siguiente línea en su terminal desde el directorio de instalación para instalar el esqueleto de la aplicación CakePHP en la carpeta **cms** del directorio de trabajo actual:

```
php composer.phar create-project --prefer-dist cakephp/app:5 cms
```

Si ha descargado y ejecutado el Instalador de Composer de Windows[22], entonces, escriba la siguiente línea en el terminal desde el directorio de instalación (ej. C:\wamp\www\dev):

```
composer self-update && composer create-project --prefer-dist cakephp/app:5.* cms
```

La ventaja de usar Composer es que completará automáticamente algunas tareas de configuración importantes, como establecer los permisos de archivo correctos y crear el archivo **config/app.php** por usted.

Hay otras formas de instalar CakePHP. Si no puede o no quiere usar Composer, consulte la sección *Instalación*.

Independientemente de cómo haya descargado e instalado CakePHP, una vez que la configuración es completada, la disposición de su directorio debería ser similar a la siguiente:

```
cms/
  bin/
  config/
  logs/
  plugins/
  resources/
  src/
  templates/
  tests/
  tmp/
  vendor/
  webroot/
  .editorconfig
  .gitignore
  .htaccess
  composer.json
  index.php
  phpunit.xml.dist
  README.md
```

Ahora podría ser un buen momento para aprender un poco sobre cómo funciona la estructura de directorios de CakePHP: consulte la sección *Estructura de carpetas de CakePHP*.

Si se pierde durante este tutorial, puede ver el resultado final en GitHub[23].

[21] https://getcomposer.org/download/
[22] https://getcomposer.org/Composer-Setup.exe
[23] https://github.com/cakephp/cms-tutorial

Comprobando nuestra instalación

PPodemos verificar rápidamente que nuestra instalación es correcta, verificando la página de inicio predeterminada. Antes de que pueda hacer eso, deberá iniciar el servidor de desarrollo:

```
cd /path/to/our/app
bin/cake server
```

Nota: Para Windows, el comando debe ser `bin\cake server` (tenga en cuenta la barra invertida).

Esto iniciará el servidor web incorporado de PHP en el puerto 8765. Abra **http://localhost:8765** en su navegador web para ver la página de bienvenida. Todos las viñetas deben ser sombreros de chef verdes indicando que CakePHP puede conectarse a De lo contrario, es posible que deba instalar extensiones adicionales de PHP o establecer permisos de directorio.

A continuación, crearemos nuestra *Base de datos y crearemos nuestro primer modelo*.

Tutorial CMS - Creando la Base de Datos

Ahora que tenemos CakePHP instalado, configuremos la base de datos para nuestro CMS. Si aún no lo ha hecho, cree una base de datos vacía para usar en este tutorial, con un nombre de su elección, p. ej. `cake_cms`. Si está utilizando MySQL/MariaDB, puede ejecutar el siguiente SQL para crear las tablas necesarias:

```
USE cake_cms;

CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    created DATETIME,
    modified DATETIME
);

CREATE TABLE articles (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(255) NOT NULL,
    slug VARCHAR(191) NOT NULL,
    body TEXT,
    published BOOLEAN DEFAULT FALSE,
    created DATETIME,
    modified DATETIME,
    UNIQUE KEY (slug),
    FOREIGN KEY user_key (user_id) REFERENCES users(id)
) CHARSET=utf8mb4;

CREATE TABLE tags (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(191),
    created DATETIME,
    modified DATETIME,
```

(continué en la próxima página)

(proviene de la página anterior)

```
    UNIQUE KEY (title)
) CHARSET=utf8mb4;

CREATE TABLE articles_tags (
    article_id INT NOT NULL,
    tag_id INT NOT NULL,
    PRIMARY KEY (article_id, tag_id),
    FOREIGN KEY tag_key(tag_id) REFERENCES tags(id),
    FOREIGN KEY article_key(article_id) REFERENCES articles(id)
);

INSERT INTO users (email, password, created, modified)
VALUES
('cakephp@example.com', 'secret', NOW(), NOW());

INSERT INTO articles (user_id, title, slug, body, published, created, modified)
VALUES
(1, 'First Post', 'first-post', 'This is the first post.', 1, NOW(), NOW());
```

Si está utilizando PostgreSQL, conéctese a la base de datos `cake_cms` y ejecute el siguiente SQL en su lugar:

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    created TIMESTAMP,
    modified TIMESTAMP
);

CREATE TABLE articles (
    id SERIAL PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(255) NOT NULL,
    slug VARCHAR(191) NOT NULL,
    body TEXT,
    published BOOLEAN DEFAULT FALSE,
    created TIMESTAMP,
    modified TIMESTAMP,
    UNIQUE (slug),
    FOREIGN KEY (user_id) REFERENCES users(id)
);

CREATE TABLE tags (
    id SERIAL PRIMARY KEY,
    title VARCHAR(191),
    created TIMESTAMP,
    modified TIMESTAMP,
    UNIQUE (title)
);

CREATE TABLE articles_tags (
    article_id INT NOT NULL,
```

(continué en la próxima página)

(proviene de la página anterior)

```
    tag_id INT NOT NULL,
    PRIMARY KEY (article_id, tag_id),
    FOREIGN KEY (tag_id) REFERENCES tags(id),
    FOREIGN KEY (article_id) REFERENCES articles(id)
);

INSERT INTO users (email, password, created, modified)
VALUES
('cakephp@example.com', 'secret', NOW(), NOW());

INSERT INTO articles (user_id, title, slug, body, published, created, modified)
VALUES
(1, 'First Post', 'first-post', 'This is the first post.', TRUE, NOW(), NOW());
```

Es posible que haya notado que la tabla `articles_tags` utiliza una clave primaria compuesta. CakePHP admite claves primarias compuestas en casi todas partes, lo que le permite tener esquemas más simples que no requieren columnas `id` adicionales.

Los nombres de tabla y columna que usamos no fueron arbitrarios. Al usar las *convenciones de nomenclatura* de CakePHP, podemos aprovechar CakePHP más eficazmente y evitar la necesidad de configurar el framework. Si bien CakePHP es lo suficientemente flexible para adaptarse a casi cualquier esquema de base de datos, adherirse a las convenciones le ahorrará tiempo, ya que puede aprovechar los valores predeterminados basados en convenciones que ofrece CakePHP.

Configuración de la base de datos

A continuación, digamos a CakePHP dónde está nuestra base de datos y cómo conectarse a ella. Reemplace los valores en el arreglo `Datasources.default` en su archivo **config/app_local.php** con los que aplican a su configuración. Una arreglo de configuración completo de muestra podría tener el siguiente aspecto:

```
<?php
// config/app_local.php
return [
    // Más configuración arriba.
    'Datasources' => [
        'default' => [
            'host' => 'localhost',
            'username' => 'cakephp',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_cms',
            'url' => env('DATABASE_URL', null),
        ],
    ],
    // Más configuración abajo.
];
```

Una vez que haya guardado su archivo **config/app_local.php**, debería ver que la sección "CakePHP is able to connect to the database" tiene un gorro de cocinero verde.

Nota: El fichero **config/app_local.php** se utiliza para sobreescribir los valores por defecto de la configuración en **config/app.php**. Esto facilita la configuración en los entornos de desarrollo.

Creando nuestro primer modelo

Los modelos son el corazón de las aplicaciones CakePHP. Nos permiten leer y modificar nuestros datos. Nos permiten construir relaciones entre nuestros datos, validarlos y aplicar reglas de aplicación. Los modelos construyen las bases necesarias para construir nuestras acciones y plantillas del controlador.

Los modelos de CakePHP se componen de objetos `Table` y `Entity`. Los objetos `Table` brindan acceso a la colección de entidades almacenadas en una tabla específica. Se almacenan en **src/Model/Table**. El archivo que crearemos se guardará en **src/Model/Table/ArticlesTable.php**. El archivo completo debería verse así:

```
<?php
// src/Model/Table/ArticlesTable.php
declare(strict_types=1);

namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        parent::initialize($config);
        $this->addBehavior('Timestamp');
    }
}
```

Hemos agregado el comportamiento *Timestamp Behavior* que automáticamente llenará las columnas `created` y `modified` de nuestra tabla. Al nombrar nuestro objeto `Table` `ArticlesTable`, CakePHP puede usar convenciones de nomenclatura para saber que nuestro modelo usa la tabla *articles* ` de la base de datos. CakePHP también usa convenciones para saber que la columna `id` es la clave primaria de nuestra tabla.

Nota: CakePHP creará dinámicamente un objeto modelo para usted si no puede encontrar un archivo correspondiente en **src/Model/Table**. Esto también significa que si accidentalmente asigna un nombre incorrecto a su archivo (es decir, articlestable.php o ArticleTable.php), CakePHP no reconocerá ninguna de sus configuraciones y utilizará el modelo generado en su lugar.

También crearemos una clase `Entity` para nuestros artículos. Las `Entity` representan un solo registro en la base de datos y proporcionan un comportamiento a nivel de fila para nuestros datos. Nuestra `Entity` se guardará en **src/Model/Entity/Article.php**. El archivo completo debería verse así:

```
<?php
// src/Model/Entity/Article.php
declare(strict_types=1);

namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
    protected array $_accessible = [
        'title' => true,
```

(continué en la próxima página)

(proviene de la página anterior)

```
        'body' => true,
        'published' => true,
        'created' => true,
        'modified' => true,
        'users' => true,
    ];
}
```

Nuestra entidad es bastante delgada en este momento, y solo hemos configurado la propiedad `_accessible` que controla cómo las propiedades pueden ser modificadas por *entities-mass-assignment*.

No podemos hacer mucho con nuestros modelos en este momento, así que a continuación crearemos nuestro primer *Controller y Template </tutorials-and-examples/cms/articles-controller>* para permitirnos interactuar con nuestro modelo.

CMS Tutorial - Creating the Articles Controller

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[24] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

---

[24] https://github.com/cakephp/docs

CAPÍTULO 3

Tutoriales y Ejemplos

En esta sección puedes encontrar varias aplicaciones completas construidas en CakePHP que te ayudarán a comprender el framework y ver cómo se relacionan todas las piezas.

También puedes ver otros ejemplos en: CakePackages[25] y en Bakery[26] encontrarás también componentes listos para usar.

Tutorial Gestor de Contenidos

Este tutorial lo guiará a través de la creación de un CMS simple. Para empezar, instalaremos CakePHP, creando nuestra base de datos y construyendo una gestión simple de artículos.

Esto es lo que se necesitará:

1. Un servidor de base de datos. Vamos a utilizar el servidor MySQL en este tutorial. Necesitará saber lo suficiente sobre SQL para crear una base de datos y ejecutar fragmentos SQL del tutorial. CakePHP se encargará de construir todas las consultas que su aplicación necesita. Como estamos usando MySQL, también asegúrese de tener `pdo_mysql` habilitado en PHP.
2. Conocimientos básicos de PHP.

Antes de comenzar, debe asegurarse de tener una versión de PHP actualizada:

```
php -v
```

Al menos debería haber instalado PHP 8.1 (CLI) o superior. La versión PHP de su servidor web también debe ser de 8.1 o superior, y debería ser la misma versión que su interfaz de línea de comando (CLI) de PHP.

[25] https://plugins.cakephp.org/
[26] https://bakery.cakephp.org/

Obteniendo CakePHP

La forma más fácil de instalar CakePHP es usar Composer. Composer es una manera simple de instalar CakePHP desde su terminal o línea de comandos. Primero, necesita descargar e instalar Composer si aún no lo ha hecho. Si tiene cURL instalado, es tan fácil como ejecutar lo siguiente:

```
curl -s https://getcomposer.org/installer | php
```

O, puede descargar `composer.phar` desde el sitio web de Composer[27].

Luego simplemente escriba la siguiente línea en su terminal desde el directorio de instalación para instalar el esqueleto de la aplicación CakePHP en la carpeta **cms** del directorio de trabajo actual:

```
php composer.phar create-project --prefer-dist cakephp/app:5 cms
```

Si ha descargado y ejecutado el Instalador de Composer de Windows[28], entonces, escriba la siguiente línea en el terminal desde el directorio de instalación (ej. C:\wamp\www\dev):

```
composer self-update && composer create-project --prefer-dist cakephp/app:5.* cms
```

La ventaja de usar Composer es que completará automáticamente algunas tareas de configuración importantes, como establecer los permisos de archivo correctos y crear el archivo **config/app.php** por usted.

Hay otras formas de instalar CakePHP. Si no puede o no quiere usar Composer, consulte la sección *Instalación*.

Independientemente de cómo haya descargado e instalado CakePHP, una vez que la configuración es completada, la disposición de su directorio debería ser similar a la siguiente:

```
cms/
  bin/
  config/
  logs/
  plugins/
  resources/
  src/
  templates/
  tests/
  tmp/
  vendor/
  webroot/
  .editorconfig
  .gitignore
  .htaccess
  composer.json
  index.php
  phpunit.xml.dist
  README.md
```

Ahora podría ser un buen momento para aprender un poco sobre cómo funciona la estructura de directorios de CakePHP: consulte la sección *Estructura de carpetas de CakePHP*.

Si se pierde durante este tutorial, puede ver el resultado final en GitHub[29].

[27] https://getcomposer.org/download/
[28] https://getcomposer.org/Composer-Setup.exe
[29] https://github.com/cakephp/cms-tutorial

Comprobando nuestra instalación

PPodemos verificar rápidamente que nuestra instalación es correcta, verificando la página de inicio predeterminada. Antes de que pueda hacer eso, deberá iniciar el servidor de desarrollo:

```
cd /path/to/our/app
bin/cake server
```

Nota: Para Windows, el comando debe ser `bin\cake server` (tenga en cuenta la barra invertida).

Esto iniciará el servidor web incorporado de PHP en el puerto 8765. Abra **http://localhost:8765** en su navegador web para ver la página de bienvenida. Todos las viñetas deben ser sombreros de chef verdes indicando que CakePHP puede conectarse a De lo contrario, es posible que deba instalar extensiones adicionales de PHP o establecer permisos de directorio.

A continuación, crearemos nuestra *Base de datos y crearemos nuestro primer modelo*.

Tutorial CMS - Creando la Base de Datos

Ahora que tenemos CakePHP instalado, configuremos la base de datos para nuestro CMS. Si aún no lo ha hecho, cree una base de datos vacía para usar en este tutorial, con un nombre de su elección, p. ej. `cake_cms`. Si está utilizando MySQL/MariaDB, puede ejecutar el siguiente SQL para crear las tablas necesarias:

```
USE cake_cms;

CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    created DATETIME,
    modified DATETIME
);

CREATE TABLE articles (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(255) NOT NULL,
    slug VARCHAR(191) NOT NULL,
    body TEXT,
    published BOOLEAN DEFAULT FALSE,
    created DATETIME,
    modified DATETIME,
    UNIQUE KEY (slug),
    FOREIGN KEY user_key (user_id) REFERENCES users(id)
) CHARSET=utf8mb4;

CREATE TABLE tags (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(191),
    created DATETIME,
    modified DATETIME,
```

(continué en la próxima página)

(proviene de la página anterior)

```
    UNIQUE KEY (title)
) CHARSET=utf8mb4;

CREATE TABLE articles_tags (
    article_id INT NOT NULL,
    tag_id INT NOT NULL,
    PRIMARY KEY (article_id, tag_id),
    FOREIGN KEY tag_key(tag_id) REFERENCES tags(id),
    FOREIGN KEY article_key(article_id) REFERENCES articles(id)
);

INSERT INTO users (email, password, created, modified)
VALUES
('cakephp@example.com', 'secret', NOW(), NOW());

INSERT INTO articles (user_id, title, slug, body, published, created, modified)
VALUES
(1, 'First Post', 'first-post', 'This is the first post.', 1, NOW(), NOW());
```

Si está utilizando PostgreSQL, conéctese a la base de datos `cake_cms` y ejecute el siguiente SQL en su lugar:

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL,
    created TIMESTAMP,
    modified TIMESTAMP
);

CREATE TABLE articles (
    id SERIAL PRIMARY KEY,
    user_id INT NOT NULL,
    title VARCHAR(255) NOT NULL,
    slug VARCHAR(191) NOT NULL,
    body TEXT,
    published BOOLEAN DEFAULT FALSE,
    created TIMESTAMP,
    modified TIMESTAMP,
    UNIQUE (slug),
    FOREIGN KEY (user_id) REFERENCES users(id)
);

CREATE TABLE tags (
    id SERIAL PRIMARY KEY,
    title VARCHAR(191),
    created TIMESTAMP,
    modified TIMESTAMP,
    UNIQUE (title)
);

CREATE TABLE articles_tags (
    article_id INT NOT NULL,
```

(continué en la próxima página)

(proviene de la página anterior)

```
    tag_id INT NOT NULL,
    PRIMARY KEY (article_id, tag_id),
    FOREIGN KEY (tag_id) REFERENCES tags(id),
    FOREIGN KEY (article_id) REFERENCES articles(id)
);

INSERT INTO users (email, password, created, modified)
VALUES
('cakephp@example.com', 'secret', NOW(), NOW());

INSERT INTO articles (user_id, title, slug, body, published, created, modified)
VALUES
(1, 'First Post', 'first-post', 'This is the first post.', TRUE, NOW(), NOW());
```

Es posible que haya notado que la tabla `articles_tags` utiliza una clave primaria compuesta. CakePHP admite claves primarias compuestas en casi todas partes, lo que le permite tener esquemas más simples que no requieren columnas `id` adicionales.

Los nombres de tabla y columna que usamos no fueron arbitrarios. Al usar las *convenciones de nomenclatura* de CakePHP, podemos aprovechar CakePHP más eficazmente y evitar la necesidad de configurar el framework. Si bien CakePHP es lo suficientemente flexible para adaptarse a casi cualquier esquema de base de datos, adherirse a las convenciones le ahorrará tiempo, ya que puede aprovechar los valores predeterminados basados en convenciones que ofrece CakePHP.

Configuración de la base de datos

A continuación, digamos a CakePHP dónde está nuestra base de datos y cómo conectarse a ella. Reemplace los valores en el arreglo `Datasources.default` en su archivo **config/app_local.php** con los que aplican a su configuración. Una arreglo de configuración completo de muestra podría tener el siguiente aspecto:

```
<?php
// config/app_local.php
return [
    // Más configuración arriba.
    'Datasources' => [
        'default' => [
            'host' => 'localhost',
            'username' => 'cakephp',
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_cms',
            'url' => env('DATABASE_URL', null),
        ],
    ],
    // Más configuración abajo.
];
```

Una vez que haya guardado su archivo **config/app_local.php**, debería ver que la sección "CakePHP is able to connect to the database" tiene un gorro de cocinero verde.

Nota: El fichero **config/app_local.php** se utiliza para sobreescribir los valores por defecto de la configuración en **config/app.php**. Esto facilita la configuración en los entornos de desarrollo.

Creando nuestro primer modelo

Los modelos son el corazón de las aplicaciones CakePHP. Nos permiten leer y modificar nuestros datos. Nos permiten construir relaciones entre nuestros datos, validarlos y aplicar reglas de aplicación. Los modelos construyen las bases necesarias para construir nuestras acciones y plantillas del controlador.

Los modelos de CakePHP se componen de objetos `Table` y `Entity`. Los objetos `Table` brindan acceso a la colección de entidades almacenadas en una tabla específica. Se almacenan en **src/Model/Table**. El archivo que crearemos se guardará en **src/Model/Table/ArticlesTable.php**. El archivo completo debería verse así:

```
<?php
// src/Model/Table/ArticlesTable.php
declare(strict_types=1);

namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config): void
    {
        parent::initialize($config);
        $this->addBehavior('Timestamp');
    }
}
```

Hemos agregado el comportamiento *Timestamp Behavior* que automáticamente llenará las columnas `created` y `modified` de nuestra tabla. Al nombrar nuestro objeto `Table` `ArticlesTable`, CakePHP puede usar convenciones de nomenclatura para saber que nuestro modelo usa la tabla *articles* ` de la base de datos. CakePHP también usa convenciones para saber que la columna `id` es la clave primaria de nuestra tabla.

Nota: CakePHP creará dinámicamente un objeto modelo para usted si no puede encontrar un archivo correspondiente en **src/Model/Table**. Esto también significa que si accidentalmente asigna un nombre incorrecto a su archivo (es decir, articlestable.php o ArticleTable.php), CakePHP no reconocerá ninguna de sus configuraciones y utilizará el modelo generado en su lugar.

También crearemos una clase `Entity` para nuestros artículos. Las `Entity` representan un solo registro en la base de datos y proporcionan un comportamiento a nivel de fila para nuestros datos. Nuestra `Entity` se guardará en **src/Model/Entity/Article.php**. El archivo completo debería verse así:

```
<?php
// src/Model/Entity/Article.php
declare(strict_types=1);

namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{
    protected array $_accessible = [
        'title' => true,
```

(continué en la próxima página)

(proviene de la página anterior)

```
        'body' => true,
        'published' => true,
        'created' => true,
        'modified' => true,
        'users' => true,
    ];
}
```

Nuestra entidad es bastante delgada en este momento, y solo hemos configurado la propiedad `_accessible` que controla cómo las propiedades pueden ser modificadas por *entities-mass-assignment*.

No podemos hacer mucho con nuestros modelos en este momento, así que a continuación crearemos nuestro primer *Controller y Template </tutorials-and-examples/cms/articles-controller>* para permitirnos interactuar con nuestro modelo.

CMS Tutorial - Creating the Articles Controller

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[30] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

CMS Tutorial - Tags and Users

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[31] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

CMS Tutorial - Authentication

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[32] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

[30] https://github.com/cakephp/docs
[31] https://github.com/cakephp/docs
[32] https://github.com/cakephp/docs

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

CMS Tutorial - Authorization

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[33] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Tutorial Blog

Bienvenido a CakePHP. Probablemente estás consultando este tutorial porque quieres aprender más sobre cómo funciona CakePHP. Nuestro objetivo es potenciar tu productividad y hacer más divertido el desarrollo de aplicaciones. Esperamos que puedas comprobarlo a medida que vas profundizando en el código.

Este tutorial te guiará en la creación de una aplicación sencilla de blog. Obtendremos e instalaremos CakePHP, crearemos y configuraremos la base de datos y añadiremos suficiente lógica como para listar, añadir, editar y eliminar artículos del blog.

Esto es lo que necesitarás:

1. Servidor web funcionando. Asumiremos que estás usando Apache, aunque las instrucciones para otros servidores son similares. Igual tendremos que ajustar un poco la configuración inicial, pero la mayoría pueden poner en marcha CakePHP sin configuración alguna. Asegúrate de tener PHP 8.1 o superior así como tener las extensiones `mbstring`, `intl` y `mcrypt` activadas en PHP.
2. Servidor de base de datos. Usaremos MySQL en este tutorial. Necesitarás saber cómo crear una base de datos nueva. CakePHP se encargará del resto. Dado que utilizamos MySQL, asegúrate también de tener `pdo_mysql` habilitado en PHP.
3. Conocimientos básicos de PHP.

¡Vamos allá!

Obtener CakePHP

La manera más sencilla de ponerse en marcha es utilizando Composer. Composer te permite instalar fácilmente CakePHP desde tu terminal o consola. Primero, debes descargar e instalar Composer si todavía no lo has hecho. Si tienes cURL instalado, es tan fácil como ejecutar lo siguiente:

```
curl -s https://getcomposer.org/installer | php
```

O puedes descargar `composer.phar` desde la página web de Composer[34].

Instalando Composer de manera global evitarás tener que repetir este paso para cada proyecto.

[33] https://github.com/cakephp/docs
[34] https://getcomposer.org/download/

Luego, simplemente escribe la siguiente línea en tu terminal desde tu directorio de instalación para instalar el esqueleto de la aplicación de CakePHP en el directorio [nombre_app].

```
php composer.phar create-project --prefer-dist cakephp/app:4.* [nombre_app]
```

O si tienes Composer instalado globalmente:

```
composer create-project --prefer-dist cakephp/app:4.* [nombre_app]
```

La ventaja de utilizar Composer es que automáticamente completará algunas tareas de inicialización, como aplicar permisos a ficheros y crear tu fichero config/app.php por ti.

Existen otros modos de instalar CakePHP si no te sientes cómodo con Composer. Para más información revisa la sección *Instalación*.

Dejando de lado cómo has descargado e instalado CakePHP, una vez ha terminado la configuración, tu directorio de instalación debería tener la siguiente estructura:

```
/directorio_raiz
    /config
    /logs
    /src
    /plugins
    /tests
    /tmp
    /vendor
    /webroot
    .gitignore
    .htaccess
    .travis.yml
    README.md
    composer.json
    phpunit.xml.dist
```

Quizás sea buen momento para aprender algo sobre cómo funciona esta estructura de directorios: echa un vistazo a la sección *Estructura de carpetas de CakePHP*.

Permisos de directorio en tmp

También necesitarás aplicar los permisos adecuados en el directorio `/tmp` para que el servidor web pueda escribir en él. El mejor modo de hacer esto es encontrar con qué usuario corre tu servidor web (`<?= `whoami`; ?>`) y cambiar la propiedad del directorio `tmp` hacia dicho usuario. El comando final que ejecutarás (en *nix) se parecerá al siguiente:

```
$ chown -R www-data tmp
```

Si por alguna razón CakePHP no puede escribir en ese directorio, serás informado mediante una alerta mientras no estés en modo producción.

A pesar de que no se recomienda, si no eres capaz de aplicar la propiedad del directorio al mismo usuario que el servidor web, puedes simplemente aplicar permisos de escritura al directorio ejecutando un comando tipo:

```
$ chmod -R 777 tmp
```

Creando la base de datos del Blog

Vamos a crear una nueva base de datos para el blog. Puedes crear una base de datos en blanco con el nombre que quieras. De momento vamos a definir sólo una tabla para nuestros artículos («posts»). Además crearemos algunos artículos de test para usarlos luego. Una vez creada la tabla, ejecuta el siguiente código SQL en ella:

```
# Primero, creamos la tabla artículos
CREATE TABLE articles (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(50),
    body TEXT,
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);

# Luego insertamos algunos artículos para probar
INSERT INTO articles (title,body,created)
    VALUES ('El título', 'Esto es el cuerpo del artículo.', NOW());
INSERT INTO articles (title,body,created)
    VALUES ('Un título de nuevo', 'Y el cuerpo sigue.', NOW());
INSERT INTO articles (title,body,created)
    VALUES ('El título ataca de nuevo', '¡Esto es realmente emocionante! No.', NOW());
```

La elección de los nombres para el nombre de la tabla y de algunas columnas no se ha hecho al azar. Si sigues las convenciones para nombres en la Base de Datos, y las demás convenciones en tus clases (ver más sobre convenciones aquí: *Convenciones CakePHP*), aprovecharás la potencia del framework y ahorrarás mucho trabajo de configuración. CakePHP es suficientemente flexible como para acomodarse hasta en el peor esquema de base de datos, pero utilizando las convenciones ahorrarás tiempo.

Echa un vistazo a *las convencionnes* para más información, pero basta decir que nombrando nuestra tabla "articles" automáticamente lo vincula a nuestro modelo Articles y que campos llamados *modified* y *created* serán gestionados automáticamente por CakePHP.

Configurando la Base de Datos

Rápido y sencillo, vamos a decirle a CakePHP dónde está la Base de Datos y cómo conectarnos a ella. Seguramente esta sea la primera y última vez que configuras nada.

Una copia del fichero de configuración de CakePHP puede ser hallado en **config/app.default.php**. Copia este fichero en su mismo directorio, pero nómbralo **app.php**.

El fichero de configuración debería de ser bastante sencillo: simplemente reemplaza los valores en la matriz `` Datasources.default`` con los que encajen con tu configuración. Una configuración completa de ejemplo podría parecerse a esto:

```
return [
    // Más configuración arriba
    'Datasources' => [
        'default' => [
            'className' => 'Cake\Database\Connection',
            'driver' => 'Cake\Database\Driver\Mysql',
            'persistent' => false,
            'host' => 'localhost',
            'username' => 'cake_blog',
```

(continué en la próxima página)

(proviene de la página anterior)

```
            'password' => 'AngelF00dC4k3~',
            'database' => 'cake_blog',
            'encoding' => 'utf8',
            'timezone' => 'UTC'
        ],
    ],
    // Más configuración abajo
];
```

En cuanto guardes tu nuevo fichero **app.php** deberías de ser capaz de acceder mediante tu navegador web y ver la página de bienvenida de CakePHP. También debería decirte que se ha encontrado el fichero de configuración así como que ha podido conectarse a la base de datos.

Nota: Recuerda que debes tener PDO y pdo_mysql habilitados en tu php.ini.

Configuración Opcional

Aún hay unas pocas cosas que puedes configurar. La mayoría de desarrolladores acaban estos ítems de la lista de la compra, pero no se necesitan para este tutorial. Uno de ellos es definir un string de seguridad (security salt) para realizar los "hash" de seguridad.

El string de seguridad se utiliza para generar "hashes". Cambia el valor por defecto editando el fichero **config/app.php**. No importa mucho el valor que contenga, cuanto más largo más difícil de averiguar:

```
'Security' => [
    'salt' => 'Algo largo y conteniendo un montón de distintos valores.',
],
```

Sobre mod_rewrite

Si eres nuevo usuario de apache, puedes encontrar alguna dificultad con mod_rewrite, así que lo trataremos aquí.

Si al cargar la página de bienvenida de CakePHP ves cosas raras (no se cargan las imágenes ni los estilos y se ve todo en blanco y negro), esto significa que probablemente mod_rewrite no está funcionando en tu sistema. Por favor, consulta la sección para tu servidor entre las siguientes acerca de re-escritura de URLs para poder poner en marcha la aplicación:

1. Comprueba que existen los ficheros .htaccess en el directorio en el que está instalada tu aplicación web. A veces al descomprimir el archivo o al copiarlo desde otra ubicación, estos ficheros no se copian correctamente. Si no están ahí, obtén otra copia de CakePHP desde el servidor oficial de descargas.
2. Asegúrate de tener activado el módulo mod_rewrite en la configuración de apache. Deberías tener algo así:

   ```
       LoadModule rewrite_module       libexec/httpd/mod_rewrite.so

   (para apache 1.3)::

       AddModule       mod_rewrite.c

   en tu fichero httpd.conf
   ```

Si no puedes (o no quieres) configurar mod_rewrite o algún otro módulo compatible, necesitarás activar las url amigables en CakePHP. En el fichero **config/app.php**, quita el comentario a la línea:

```
'App' => [
    // ...
    // 'baseUrl' => env('SCRIPT_NAME'),
]
```

Borra también los ficheros .htaccess que ya no serán necesarios:

```
/.htaccess
/webroot/.htaccess
```

Esto hará que tus url sean así: www.example.com/index.php/nombredelcontrolador/nombredelaaccion/parametro en vez de www.example.com/nombredelcontrolador/nombredelaaccion/parametro.

Si estás instalando CakePHP en otro servidor diferente a Apache, encontrarás instrucciones para que funcione la reescritura de URLs en la sección url-rewriting

Ahora continúa hacia *Tutorial Blog - Parte 2* para empezar a construir tu primera aplicación en CakePHP.

Tutorial Blog - Parte 2

Nota: The documentation is currently partially supported in es language for this page.

Por favor, siéntase libre de enviarnos un pull request en Github[35] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Crear un modelo Artículo (`Article`)

Los modelos son una parte fundamental en CakePHP. Cuando creamos un modelo, podemos interactuar con la base de datos para crear, editar, ver y borrar con facilidad cada ítem de ese modelo.

Los modelos están separados entre los objetos `Tabla` (`Table`) y `Entidad` (`Entity`). Los objetos `Tabla` proporcionan acceso a la coleción de entidades almacenada en una tabla específica y va en **src/Model/Table**. El fichero que crearemos se guardará en **src/Model/Table/ArticlesTable.php**. El fichero completo debería tener este aspecto:

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Timestamp');
    }
}
```

[35] https://github.com/cakephp/docs

Los convenios usados para los nombres son importantes. Llamando a nuestro objeto Tabla `ArticlesTable`, CakePHP deducirá automáticamente que esta Tabla será utilizada en el controlador ArticlesController, y que se vinculará a una tabla en nuestra base de datos llamada `articles`.

Nota: CakePHP creará dinámicamente un objeto para el modelo si no encuentra el fichero correspondiente en **src/Model/Table**. Esto significa que si te equivocas al nombrar el fichero (por ejemplo lo llamas articlestable.php —en minúscula— o ArticleTable.php —en singular) CakePHP no va a reconocer la configuración que escribas en ese fichero y utilizará valores por defecto.

Para más información sobre modelos, como callbacks y validaciones echa un vistazo al capítulo del Manual *Acceso a la base de datos & ORM*.

Crear el Controlador de Artículos (`Articles Controller`)

Vamos a crear ahora un controlador para nuestros artículos. En el controlador es donde escribiremos el código para interactuar con nuestros artículos. Es donde se utilizan los modelos para llevar a cabo el trabajo que queramos hacer con nuestros artículos. Vamos a crear un nuevo fichero llamado **ArticlesController.php** dentro del directorio **src/Controller**. A continuación puedes ver el aspecto básico que debería tener este controlador:

```
namespace App\Controller;

class ArticlesController extends AppController
{
}
```

Vamos a añadir una acción a nuestro nuevo controlador. Las acciones representan una función concreta o interfaz en nuestra aplicación. Por ejemplo, cuando los usuarios recuperan la url www.example.com/articles/index (que es lo mismo que www.example.com/articles/) esperan ver un listado de artículos. El código para tal acción sería este:

```
namespace App\Controller;

class ArticlesController extends AppController
{

    public function index()
    {
        $articles = $this->Articles->find('all');
        $this->set(compact('articles'));
    }
}
```

Por el hecho de haber definido el método `index()` en nuestro ArticlesController, los usuarios ahora pueden acceder a su lógica solicitando www.example.com/articles/index. Del mismo modo, si definimos un método llamado `foobar()` los usuarios tendrán acceso a él desde www.example.com/articles/foobar.

Advertencia: Puede que tengas la tentación de llamar tus controladores y acciones de cierto modo para obtener una URL en concreto. Resiste la tentación. Sigue las convenciones de CakePHP (mayúsculas, nombre en plural, etc.) y crea acciones comprensibles, que se dejen leer. Luego podrás asignar URLs a tu código utilizando «rutas», que veremos más adelante.

La única instrucción en la acción utiliza `set()` para pasar datos desde el controlador hacia la vista (que crearemos a

continuación). La línea en cuestión asigna una variable en la vista llamada "articles" igual al valor retornado por el método `find('all')` del objeto de tabla Artículos (`ArticlesTable`).

Para aprender más sobre los controladores, puedes visitar el capítulo *Controladores*.

Crear Vistas de Artículos (`Article Views`)

Ahora que tenemos nuestros datos fluyendo por el modelo, y que la lógica de nuestra aplicación está definida en nuestro controlador, vamos a crear una vista para la acción índex creada en el paso anterior.

Las vistas en CakePHP únicamente son fragmentos de presentación que encajan dentro de la plantilla (`layout`) de nuestra aplicación. Para la mayoría de aplicaciones son HTML mezclados con PHP, pero bien podrían acabar siendo XML, CSV o incluso datos binarios.

Una plantilla es una presentación de código que envuelve una vista. Se pueden definir múltiples plantillas y puedes cambiar entre ellas pero, por ahora, utilizaremos la plantilla por defecto (`default`).

¿Recuerdas cómo en la sección anterior hemos asignado la variable "articles" a la vista utilizando el método `set()`? Esto asignaría el objeto de consulta (`query object`) a la vista para ser invocado por una iteración `foreach`.

Las vistas en CakePHP se almacenan en la ruta `/src/Template` y en un directorio con el mismo nombre que el controlador al que pertenecen (tendremos que crear una carpeta llamada "Articles" en este caso). Para dar formato a los datos de este artículo en una bonita tabla, el código de nuestra vista debería ser algo así:

```
<!-- File: /templates/Articles/index.php -->

<h1>Artículos</h1>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Created</th>
    </tr>

    <!-- Aquí es donde iteramos nuestro objeto de consulta $articles, mostrando en␣
↪pantalla la información del artículo -->

    <?php foreach ($articles as $article): ?>
    <tr>
        <td><?= $article->id ?></td>
        <td>
            <?= $this->Html->link($article->title,
            ['controller' => 'Articles', 'action' => 'view', $article->id]) ?>
        </td>
        <td><?= $article->created->format(DATE_RFC850) ?></td>
    </tr>
    <?php endforeach; ?>
</table>
```

Esto debería ser sencillo de comprender.

Como habrás notado, hay una llamada a un objeto `$this->Html`. Este objeto es una instancia de la clase *Cake\View\Helper\HtmlHelper* de CakePHP. CakePHP proporciona un conjunto de ayudantes de vistas (`helpers`) para ayudarte a completar acciones habituales, como por ejemplo crear un enlace o un formulario. Puedes aprender más sobre esto en *Helpers*, pero lo que es importante destacar aquí es que el método `link()` generará un enlace HTML con el título como primer parámetro y la URL como segundo parámetro.

Cuando crees URLs en CakePHP te recomendamos emplear el formato de array. Se explica con detenimiento en la sección de Rutas (`Routes`). Si utilizas las rutas en formato array podrás aprovecharte de las potentes funcionalidades de generación de rutas inversa de CakePHP en el futuro. Además puedes especificar rutas relativas a la base de tu aplicación de la forma `/controlador/accion/param1/param2` o incluso utilizar *Usar Rutas con Nombre*.

Llegados a este punto, deberías ser capaz de acceder con tu navegador a http://www.example.com/articles/index. Deberías ver tu vista, correctamente formatada con el título y la tabla listando los artículos.

Si te ha dado por hacer clic en uno de los enlaces que hemos creado en esta vista (que enlazan el título de un artículo hacia la URL `/articles/view/un\_id`), seguramente habrás sido informado por CakePHP de que la acción no ha sido definida todavía. Si no has sido infromado, o bien algo ha ido mal o bien ya la habías definido, en cuyo caso eres muy astuto. En caso contrario, la crearemos ahora en nuestro controlador de artículos:

```
namespace App\Controller;

class ArticlesController extends AppController
{

    public function index()
    {
         $this->set('articles', $this->Articles->find('all'));
    }

    public function view($id = null)
    {
        $article = $this->Articles->get($id);
        $this->set(compact('article'));
    }
}
```

Si observas la función view(), ahora el método set() debería serte familiar. Verás que estamos usando `get()` en vez de `find('all')` ya que sólo queremos un artículo concreto.

Verás que nuestra función view toma un parámetro: el ID del artículo que queremos ver. Este parámetro se gestiona automáticamente al llamar a la URL `/articles/view/3`, el valor "3" se pasa a la función view como primer parámetro `$id`.

También hacemos un poco de verificación de errores para asegurarnos de que el usuario realmente accede a dicho registro. Si el usuario solicita `/articles/view` lanzaremos una excepción `NotFoundException` y dejaremos al ErrorHandler tomar el control. Utilizando el método `get()` en la tabla Articles también hacemos una verificación similar para asegurarnos de que el usuario ha accedido a un registro que existe. En caso de que el artículo solicitado no esté presente en la base de datos, el método `get()` lanzará una excepción `NotFoundException`.

Ahora vamos a definir la vista para esta nueva función "view" ubicándola en **templates/Articles/view.php**.

```
<!-- File: /templates/Articles/view.php -->
<h1><?= h($article->title) ?></h1>
<p><?= h($article->body) ?></p>
<p><small>Created: <?= $article->created->format(DATE_RFC850) ?></small></p>
```

Verifica que esto funciona probando los enlaces en `/articles/index` o puedes solicitándolo manualmente accediendo a `/articles/view/1`.

Añadiendo Artículos

Leer de la base de datos y mostrar nuestros artículos es un gran comienzo, pero permitamos también añadir nuevos artículos.

Lo primero, añadir una nueva acción `add()` en nuestro controlador ArticlesController:

```
namespace App\Controller;

class ArticlesController extends AppController
{
    public $components = ['Flash'];

    public function index()
    {
        $this->set('articles', $this->Articles->find('all'));
    }

    public function view($id)
    {
        $article = $this->Articles->get($id);
        $this->set(compact('article'));
    }

    public function add()
    {
        $article = $this->Articles->newEmptyEntity();
        if ($this->request->is('post')) {
            $article = $this->Articles->patchEntity($article, $this->request->getData());
            if ($this->Articles->save($article)) {
                $this->Flash->success(__('Your article has been saved.'));

                return $this->redirect(['action' => 'index']);
            }
            $this->Flash->error(__('Unable to add your article.'));
        }
        $this->set('article', $article);
    }
}
```

Nota: Necesitas incluir el FlashComponent en cualquier controlador donde vayas a usarlo. Si lo ves necesario, inclúyelo en tu AppController.

Lo que la función add() hace es: si el formulario enviado no está vacío, intenta salvar un nuevo artículo utilizando el modelo Articles. Si no se guarda bien, muestra la vista correspondiente, así podremos mostrar los errores de validación u otras alertas.

Cada petición de CakePHP incluye un objeto `ServerRequest` que es accesible utilizando `$this->request`. El objeto de petición contiene información útil acerca de la petición que se recibe y puede ser utilizado para controlar el flujo de nuestra aplicación. En este caso, utilizamos el método `Cake\Network\ServerRequest::is()` para verificar que la petición es una petición HTTP POST.

Cuando un usuario utiliza un formulario y efectúa un POST a la aplicación, esta información está disponible en `$this->request->getData()`. Puedes usar la función `pr()` o `debug()` para mostrar el contenido de esa variable y

ver la pinta que tiene.

Utilizamos el método mágico `__call` del `FlashComponent` para guardar un mensaje en una variable de sesión que será mostrado en la página después de la redirección. En la plantilla tenemos `<?= $this->Flash->render() ?>` que muestra el mensaje y elimina la correspondiente variable de sesión. El método *Cake\Controller\Controller::redirect* del controlador redirige hacia otra URL. El parámetro `['action' => 'index']` se traduce a la URL /articles (p.e. la acción index del controlador de artículos). Puedes echar un ojo al método `Cake\Routing\Router::url()` en la API[36] para ver los formatos en que puedes especificar una URL para varias funciones de CakePHP.

Al llamar al método `save()`, comprobará si hay errores de validación primero y si encuentra alguno, no continuará con el proceso de guardado. Veremos a continuación cómo trabajar con estos errores de validación.

Validando los Datos

CakePHP te ayuda a evitar la monotonía al construir tus formularios y su validación. Todos odiamos teclear largos formularios y gastar más tiempo en reglas de validación de cada campo. CakePHP lo hace más rápido y sencillo.

Para aprovechar estas funciones es conveniente que utilices el FormHelper en tus vistas. La clase *Cake\View\Helper\FormHelper* está disponible en tus vistas por defecto a través de `$this->Form`.

He aquí nuestra vista `add`:

```
<!-- File: templates/Articles/add.php -->

<h1>Añadir Artículo</h1>
<?php
    echo $this->Form->create($article);
    echo $this->Form->input('title');
    echo $this->Form->input('body', ['rows' => '3']);
    echo $this->Form->button(__('Guardar artículo'));
    echo $this->Form->end();
?>
```

Hemos usado FormHelper para generar la etiqueta "form". La ejecución de `$this->Form->create()` genera el siguiente código:

```
<form method="post" action="/articles/add">
```

Si `create()` no tiene parámetros al ser llamado, asume que estás creando un formulario que envía vía POST a la acción `add()` (o `edit()` cuando `id` es incluido en los datos de formulario) del controlador actual.

El método `$this->Form->input()` se utiliza para crear elementos de formulario del mismo nombre. El primer parámetro le indica a CakePHP a qué campo corresponde y el segundo parámetro te permite especificar un abanico muy ámplio de opciones - en este caso, el número de filas del textarea que se generará. Hay un poco de introspección y «automagia» aquí: `input()` generará distintos elementos de formulario en función del campo del modelo especificado.

La llamada a `$this->Form->end()` cierra el formulario. También generará campos ocultos si la CSRF/prevención de manipulación de formularios ha sido habilitada.

Volvamos atrás un minuto y actualicemos nuestra vista **templates/Articles/index.php** para añadir un enlace de «Añadir Artículo». Justo antes del tag <table> añade la siguiente línea:

```
<?= $this->Html->link(
    'Añadir artículo',
```

(continué en la próxima página)

[36] https://api.cakephp.org

(proviene de la página anterior)

```
    ['controller' => 'Articles', 'action' => 'add']
) ?>
```

Te estarás preguntando: ¿Cómo le digo a CakePHP la forma en la que debe validar estos datos? Muy sencillo, las reglas de validación se escriben en el modelo. Volvamos al modelo `Articles` y hagamos algunos ajustes:

```
namespace App\Model\Table;

use Cake\ORM\Table;
use Cake\Validation\Validator;

class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Timestamp');
    }

    public function validationDefault(Validator $validator)
    {
        $validator
            ->notEmpty('title')
            ->notEmpty('body');

        return $validator;
    }
}
```

El método `validationDefault()` le dice a CakePHP cómo validar tus datos cuando se invoca el método `save()`. Aquí hemos especificado que ambos campos, el cuerpo y el título, no pueden quedar vacíos. El motor de validaciones de CakePHP es potente y con numerosas reglas ya predefinidas (tarjetas de crédito, direcciones de e-mail, etc.) así como flexibilidad para añadir tus propias reglas de validación. Para más información en tal configuración, echa un vistazo a la documentación *Validation*.

Ahora que ya tienes las reglas de validación definidas, usa tu aplicación para crear un nuevo artículo con un título vacío y verás cómo funcionan. Como hemos usado el método `Cake\View\Helper\FormHelper::input()`, los mensajes de error se construyen automáticamente en la vista sin código adicional.

Editando Artículos

Editando artículos: allá vamos. Ya eres un profesional de CakePHP, así que habrás cogido la pauta. Crear una acción, luego la vista. He aquí cómo debería ser la acción `edit()` del controlador `ArticlesController`:

```
public function edit($id = null)
{
    $article = $this->Articles->get($id);
    if ($this->request->is(['post', 'put'])) {
        $this->Articles->patchEntity($article, $this->request->getData());
        if ($this->Articles->save($article)) {
            $this->Flash->success(__('Tu artículo ha sido actualizado.'));

            return $this->redirect(['action' => 'index']);
```

(continué en la próxima página)

(proviene de la página anterior)

```
        }
        $this->Flash->error(__('Tu artículo no se ha podido actualizar.'));
    }

    $this->set('article', $article);
}
```

Lo primero que hace este método es asegurarse de que el usuario ha intentado acceder a un registro existente. Si no han pasado el parámetro `$id` o el artículo no existe lanzaremos una excepción `NotFoundException` para que el `ErrorHandler` se ocupe de ello.

Luego verifica si la petición es POST o PUT. Si lo es, entonces utilizamos los datos recibidos para actualizar nuestra entidad artículo (`article`) utilizando el método "patchEntity". Finalmente utilizamos el objeto tabla para guardar la entidad de nuevo o mostrar errores de validación al usuario en caso de haberlos.

La vista sería algo así:

```
<!-- File: templates/Articles/edit.php -->

<h1>Edit Article</h1>
<?php
    echo $this->Form->create($article);
    echo $this->Form->input('title');
    echo $this->Form->input('body', ['rows' => '3']);
    echo $this->Form->button(__('Guardar artículo'));
    echo $this->Form->end();
?>
```

Mostramos el formulario de edición (con los valores actuales de ese artículo), junto a los errores de validación que hubiese.

CakePHP utilizará el resultado de `$article->isNew()` para determinar si un `save()` debería insertar un nuevo registro o actualizar uno existente.

Puedes actualizar tu vista índice (`index`) con enlaces para editar artículos específicos:

```
<!-- File: templates/Articles/index.php  (edit links added) -->

<h1>Artículos</h1>
<p><?= $this->Html->link("Añadir artículo", ['action' => 'add']) ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Created</th>
        <th>Action</th>
    </tr>

<!-- Aquí es donde iteramos nuestro objeto de consulta $articles, mostrando en pantalla␣
↪la información del artículo -->

<?php foreach ($articles as $article): ?>
    <tr>
        <td><?= $article->id ?></td>
```

(continué en la próxima página)

(proviene de la página anterior)

```
        <td>
            <?= $this->Html->link($article->title, ['action' => 'view', $article->id]) ?>
        </td>
        <td>
            <?= $article->created->format(DATE_RFC850) ?>
        </td>
        <td>
            <?= $this->Html->link('Editar', ['action' => 'edit', $article->id]) ?>
        </td>
    </tr>
<?php endforeach; ?>

</table>
```

Borrando Artículos

Vamos a permitir a los usuarios que borren artículos. Empieza con una acción `delete()` en el controlador `ArticlesController`:

```
public function delete($id)
{
    $this->request->allowMethod(['post', 'delete']);

    $article = $this->Articles->get($id);
    if ($this->Articles->delete($article)) {
        $this->Flash->success(__('El artículo con id: {0} ha sido eliminado.', h($id)));

        return $this->redirect(['action' => 'index']);
    }
}
```

La lógica elimina el artículo especificado por $id y utiliza `$this->Flash->success()` para mostrar al usuario un mensaje de confirmación tras haber sido redirigidos a `/articles`. Si el usuario intenta eliminar utilizando una petición GET, el "allowMethod" devolvería una Excepción. Las excepciones que no se traten serán capturadas por el manejador de excepciones de CakePHP (`exception handler`) y una bonita página de error es mostrada. Hay muchas *Excepciones* que pueden ser utilizadas para indicar los varios errores HTTP que tu aplicación pueda generar.

Como estamos ejecutando algunos métodos y luego redirigiendo a otra acción de nuestro controlador, no es necesaria ninguna vista (nunca se usa). Lo que si querrás es actualizar la vista index.php para incluír el ya habitual enlace:

```
<!-- File: templates/Articles/index.php -->

<h1>Artículos</h1>
<p><?= $this->Html->link("Añadir artículo", ['action' => 'add']) ?></p>
<table>
    <tr>
        <th>Id</th>
        <th>Title</th>
        <th>Created</th>
        <th>Action</th>
    </tr>
```

(continué en la próxima página)

(proviene de la página anterior)

```
<!-- Aquí es donde iteramos nuestro objeto de consulta $articles, mostrando en pantalla␣
↪la información del artículo -->

<?php foreach ($articles as $article): ?>
    <tr>
        <td><?= $article->id ?></td>
        <td>
            <?= $this->Html->link($article->title, ['action' => 'view', $article->id]) ?>
        </td>
        <td>
            <?= $article->created->format(DATE_RFC850) ?>
        </td>
        <td>
            <?= $this->Form->postLink(
                'Eliminar',
                ['action' => 'delete', $article->id],
                ['confirm' => '¿Estás seguro?'])
            ?>
            <?= $this->Html->link('Editar', ['action' => 'edit', $article->id]) ?>
        </td>
    </tr>
<?php endforeach; ?>

</table>
```

Utilizando `postLink()` crearemos un enlace que utilizará JavaScript para hacer una petición POST que eliminará nuestro artículo. Permitiendo que contenido sea eliminado vía peticiones GET es peligroso, ya que arañas web (`crawlers`) podrían eliminar accidentalmente tu contenido.

Nota: Esta vista utiliza el FormHelper para pedir confirmación vía diálogo de confirmación de JavaScript al usuario antes de borrar un artículo.

Rutas (`Routes`)

En muchas ocasiones, las rutas por defecto de CakePHP funcionan bien tal y como están. Los desarroladores que quieren rutas diferentes para mejorar la usabilidad apreciarán la forma en la que CakePHP relaciona las URLs con las acciones de los controladores. Vamos a hacer cambios ligeros para este tutorial.

Para más información sobre las rutas así como técnicas avanzadas revisa *Conectando Rutas*.

Por defecto CakePHP responde a las llamadas a la raíz de tu sitio (por ejemplo http://www.example.com) usando el controlador PagesController, mostrando una vista llamada «home». En lugar de eso, lo reemplazaremos con nuestro controlador `ArticlesController` creando una nueva ruta.

Las reglas de enrutamiento están en **config/routes.php**. Querrás eliminar o comentar la línea que define la raíz por defecto. Dicha ruta se parece a esto:

```
Router::connect('/', ['controller' => 'Pages', 'action' => 'display', 'home']);
```

Esta línea conecta la url "/" con la página por defecto de inicio de CakePHP. Queremos conectarla a nuestro propio controlador, así que reemplaza dicha línea por esta otra:

```
Router::connect('/', ['controller' => 'Articles', 'action' => 'index']);
```

Esto debería, cuando un usuario solicita "/", devolver la acción index() del controlador ArticlesController.

Nota: CakePHP también calcula las rutas a la inversa. Si en tu código pasas el array `['controller' => 'Articles', 'action' => 'index']` a una función que espera una url, el resultado será "/". Es buena idea usar siempre arrays para configurar las URL, lo que asegura que los links irán siempre al mismo lugar.

Conclusión

Creando aplicaciones de este modo te traerá paz, honor, amor, dinero a carretas e incluso tus fantasías más salvajes. Simple, no te parece? Ten en cuenta que este tutorial es muy básico, CakePHP tiene *muchas* otras cosas que ofrecer y es flexible aunque no hemos cubierto aquí estos puntos para que te sea más simple al principio. Usa el resto de este manual como una guía para construir mejores aplicaciones.

Ahora que ya has creado una aplicación CakePHP básica, estás listo para la vida real. Empieza tu nuevo proyecto y lee el resto del Cookbook así como la API[37].

Si necesitas ayuda, hay muchos modos de encontrar la ayuda que buscas - por favor, míralo en la página *Donde obtener ayuda*. ¡Bienvenido a CakePHP!

Lectura sugerida para continuar desde aquí

Hay varias tareas comunes que la gente que está aprendiendo CakePHP quiere aprender después:

1. *Layouts*: Personaliza la plantilla *layout* de tu aplicación
2. *Elementos* Incluír vistas y reutilizar trozos de código
3. /bake/usage: Generación básica de CRUDs
4. *Tutorial Blog - Autenticación y Autorización*: Tutorial de autenticación y permisos

Tutorial Blog - Parte 3

Crear categorias en Arbol

Vamos a continuar con nuestro blog e imaginar que queremos categorizar nuestros articulos. Queremos que las categorias estén ordenadas, y para esto, vamos a usar *Tree behavior* para ayudarnos a organizar las categorías.

Pero primero necesitamos modificar nuestras tablas.

[37] https://api.cakephp.org

Plugin de migración

Vamos a usar el migrations plugin[38] para crear una tabla en nuestra base de datos. Si tienes una tabla de articulos en tu base de datos, borrala.

Abre tu archivo `composer.json`. Generalmente el plugin de migración ya esta incluido en `require`. Si no es el caso, agrégalo:

```
"require": {
    "cakephp/migrations": "~1.0"
}
```

Luego corre el comando `composer update`. El plugin de migración se alojara en tu carpeta de `plugins`. Agrega también `Plugin::load('Migrations');` en el archivo bootstrap.php de tú aplicación.

Una vez que el plugin sea cargado, corre el siguiente comando para crear el archivo de migración:

```
bin/cake migrations create Initial
```

Un archivo de migración será creado en la carpeta `/config/Migrations`. Puedes abrir tu archivo y agregar las siguientes lineas:

```
<?php

use Phinx\Migration\AbstractMigration;

class Initial extends AbstractMigration
{
    public function change()
    {
        $articles = $this->table('articles');
        $articles->addColumn('title', 'string', ['limit' => 50])
            ->addColumn('body', 'text', ['null' => true, 'default' => null])
            ->addColumn('category_id', 'integer', ['null' => true, 'default' => null])
            ->addColumn('created', 'datetime')
            ->addColumn('modified', 'datetime', ['null' => true, 'default' => null])
            ->save();

        $categories = $this->table('categories');
        $categories->addColumn('parent_id', 'integer', ['null' => true, 'default' =>␣
↪null])
            ->addColumn('lft', 'integer', ['null' => true, 'default' => null])
            ->addColumn('rght', 'integer', ['null' => true, 'default' => null])
            ->addColumn('name', 'string', ['limit' => 255])
            ->addColumn('description', 'string', ['limit' => 255, 'null' => true,
↪'default' => null])
            ->addColumn('created', 'datetime')
            ->addColumn('modified', 'datetime', ['null' => true, 'default' => null])
            ->save();
    }
}
```

Ahora corre el siguiente comando para crear túS tablas:

[38] https://github.com/cakephp/migrations

```
bin/cake migrations migrate
```

Modificando las tablas

Con nuestras tablas creadas, ahora podemos enfocarnos en categorizar los artículos.

Suponemos que ya tienes los archivos (Tables, Controllers y Templates de Articles) de la parte 2 de esta serie de tutoriales, por lo que solamente vamos a agregar referencia a las categorías.

Necesitamos asociar las tablas de Articles y Categories. Abre el archivo **src/Model/Table/ArticlesTable.php** y agrega las siguientes lineas:

```
// src/Model/Table/ArticlesTable.php

namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{
    public function initialize(array $config)
    {
        $this->addBehavior('Timestamp');
        // Just add the belongsTo relation with CategoriesTable
        $this->belongsTo('Categories', [
            'foreignKey' => 'category_id',
        ]);
    }
}
```

Generando el código base para las Categorías

Crea todos los archivos corriendo los siguientes comandos:

```
bin/cake bake model Categories
bin/cake bake controller Categories
bin/cake bake template Categories
```

La herramienta bake ha creado todos los archivos en un instante. Puedes darles una rápida leida si necesitas re-familiarizarte con la forma en la que CakePHP funciona.

Nota: Si estás en Windows recordá usar en lugar de / .

Agregar el TreeBehavior a CategoriesTable

TreeBehavior ayuda a manejar estructuras de árbol jerarquica en una tabla. Utiliza MPTT logic[39] para manejar los datos. Las estructuras en árbol MPTT están optimizadas para lecturas, lo cual las hace ideal para aplicaciones con gran carga de lectura como los blogs.

Si abres el archivo **src/Model/Table/CategoriesTable.php** veras que el TreeBehavior fue agregado a CategoriesTable en el método `initialize()`. Bake agrega este behavior a cualquier tabla que contenga las columnas `lft` y `rght`:

```
$this->addBehavior('Tree');
```

Con el TreeBehavior agregado ahora podras acceder a algunas funcionalidades como reordenar las categorias. Veremos eso en un momento.

Pero por ahora tendrás que removar los siguientes inputs en tus archivos add y edit de Categories:

```
echo $this->Form->input('lft');
echo $this->Form->input('rght');
```

Esos campos son manejados automáticamento por el TreeBehavior cuando una categoría es guardada.

Con tú navegador, agrega alguna nueva categoría usando la acción `/yoursite/categories/add`.

Reordenando categorías con TreeBehavior

En el index de categorias, puedes listar y re-ordenar categorias.

Vamos a modificar el método index en tu `CategoriesController.php`, agregando `move_up()` y `move_down()` para poder reordenar las categorías en ese árbol:

```
class CategoriesController extends AppController
{
    public function index()
    {
        $categories = $this->Categories->find('threaded')
            ->order(['lft' => 'ASC']);
        $this->set(compact('categories'));
    }

    public function move_up($id = null)
    {
        $this->request->allowMethod(['post', 'put']);
        $category = $this->Categories->get($id);
        if ($this->Categories->moveUp($category)) {
            $this->Flash->success('The category has been moved Up.');
        } else {
            $this->Flash->error('The category could not be moved up. Please, try again.
↪');
        }

        return $this->redirect($this->referer(['action' => 'index']));
    }
```

(continué en la próxima página)

[39] https://www.sitepoint.com/hierarchical-data-database-2/

(proviene de la página anterior)

```
    public function move_down($id = null)
    {
        $this->request->allowMethod(['post', 'put']);
        $category = $this->Categories->get($id);
        if ($this->Categories->moveDown($category)) {
            $this->Flash->success('The category has been moved down.');
        } else {
            $this->Flash->error('The category could not be moved down. Please, try again.
↪');
        }

        return $this->redirect($this->referer(['action' => 'index']));
    }
}
```

En **templates/Categories/index.php** reemplazá el contenido existente por el siguiente:

```
<div class="actions columns large-2 medium-3">
    <h3><?= __('Actions') ?></h3>
    <ul class="side-nav">
        <li><?= $this->Html->link(__('New Category'), ['action' => 'add']) ?></li>
    </ul>
</div>
<div class="categories index large-10 medium-9 columns">
    <table cellpadding="0" cellspacing="0">
    <thead>
        <tr>
            <th>id</th>
            <th>Parent Id</th>
            <th>Title</th>
            <th>Lft</th>
            <th>Rght</th>
            <th>Name</th>
            <th>Description</th>
            <th>Created</th>
            <th class="actions"><?= __('Actions') ?></th>
        </tr>
    </thead>
    <tbody>
    <?php foreach ($categories as $category): ?>
        <tr>
            <td><?= $this->Number->format($category->id) ?></td>
            <td><?= $this->Number->format($category->parent_id) ?></td>
            <td><?= $this->Number->format($category->lft) ?></td>
            <td><?= $this->Number->format($category->rght) ?></td>
            <td><?= h($category->name) ?></td>
            <td><?= h($category->description) ?></td>
            <td><?= h($category->created) ?></td>
            <td class="actions">
                <?= $this->Html->link(__('View'), ['action' => 'view', $category->id]) ?>
                <?= $this->Html->link(__('Edit'), ['action' => 'edit', $category->id]) ?>
                <?= $this->Form->postLink(__('Delete'), ['action' => 'delete', $category-
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪>id], ['confirm' => __('Are you sure you want to delete # {0}?', $category->id)]) ?>
                <?= $this->Form->postLink(__('Move down'), ['action' => 'move_down',
↪$category->id], ['confirm' => __('Are you sure you want to move down # {0}?',
↪$category->id)]) ?>
                <?= $this->Form->postLink(__('Move up'), ['action' => 'move_up',
↪$category->id], ['confirm' => __('Are you sure you want to move up # {0}?', $category->
↪id)]) ?>
            </td>
        </tr>
    <?php endforeach; ?>
    </tbody>
    </table>
</div>
```

Modificando el ArticlesController

En tú `ArticlesController`, vamos a obtener el listado de categorías. Esto nos permitirá elegir una categoría para un Article al momento de crearlo o editarlo:

```
// src/Controller/ArticlesController.php

namespace App\Controller;

// Prior to 3.6 use Cake\Network\Exception\NotFoundException
use Cake\Http\Exception\NotFoundException;

class ArticlesController extends AppController
{

    // ...

    public function add()
    {
        $article = $this->Articles->newEmptyEntity();
        if ($this->request->is('post')) {
            $article = $this->Articles->patchEntity($article, $this->request->getData());
            if ($this->Articles->save($article)) {
                $this->Flash->success(__('Your article has been saved.'));

                return $this->redirect(['action' => 'index']);
            }
            $this->Flash->error(__('Unable to add your article.'));
        }
        $this->set('article', $article);

        // Just added the categories list to be able to choose
        // one category for an article
        $categories = $this->Articles->Categories->find('treeList');
        $this->set(compact('categories'));
    }
}
```

Modificando el template de Articles

El template add de Article debería verse similar a esto:

```
.. code-block:: php
```

<!– File: templates/Articles/add.php –>

<h1>Add Article</h1> <?php echo $this->Form->create($article); // just added the categories input echo $this->Form->input("categories"); echo $this->Form->input("title"); echo $this->Form->input("body", ["rows" => "3"]); echo $this->Form->button(__("Save Article")); echo $this->Form->end();

Ingresando a `/yoursite/categories/add` deberías ver una lista de categorías para elegir.

Tutorial Blog - Autenticación y Autorización

Siguiendo con nuestro ejemplo de aplicación *Tutorial Blog*, imaginá que necesitamos no permitir que usuarios no autenticados puedan crear artículos.

Creando la tabla users y el Controlador

Primero, vamos a crear una tabla en nuestra base de datos para guardar los datos de usuarios:

```
CREATE TABLE users (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255),
    password VARCHAR(255),
    role VARCHAR(20),
    created DATETIME DEFAULT NULL,
    modified DATETIME DEFAULT NULL
);
```

Si estás usando PostgreSQL, conecta a la base de datos cake_blog y ejecuta el siguiente SQL en su lugar:

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(255),
    password VARCHAR(255),
    role VARCHAR(20),
    created TIMESTAMP DEFAULT NULL,
    modified TIMESTAMP DEFAULT NULL
);
```

Seguimos las convenciones de CakePHP para nombrar tablas, pero también estamos aprovechando otra convención: al usar los campos email y password en nuestra tabla users, CakePHP configurará automáticamente la mayoría de las cosas al momento de implementar el login.

El siguiente paso es crear nuestra clase `UsersTable`, responsable de buscar, guardar y validar los datos de usuario:

```
// src/Model/Table/UsersTable.php
namespace App\Model\Table;

use Cake\ORM\Table;
```

(continué en la próxima página)

(proviene de la página anterior)

```
use Cake\Validation\Validator;

class UsersTable extends Table
{
    public function validationDefault(Validator $validator): Validator
    {
        return $validator
            ->notEmpty('email', 'An email is required')
            ->email('email')
            ->notEmpty('password', 'A password is required')
            ->notEmpty('role', 'A role is required')
            ->add('role', 'inList', [
                'rule' => ['inList', ['admin', 'author']],
                'message' => 'Please enter a valid role'
            ]);
    }

}
```

También vamos a crear nuestro `UsersController`. El siguiente contenido se corresponde con una clase `UsersController` básica «baked» usando las utilidades de generación de código que están incluidas en CakePHP:

```
// src/Controller/UsersController.php

namespace App\Controller;

use App\Controller\AppController;
use Cake\Event\EventInterface;

class UsersController extends AppController
{
    public function index()
    {
        $this->set('users', $this->Users->find()->all());
    }

    public function view($id)
    {
        $user = $this->Users->get($id);
        $this->set(compact('user'));
    }

    public function add()
    {
        $user = $this->Users->newEmptyEntity();
        if ($this->request->is('post')) {
            $user = $this->Users->patchEntity($user, $this->request->getData());
            if ($this->Users->save($user)) {
                $this->Flash->success(__('The user has been saved.'));

                return $this->redirect(['action' => 'add']);
            }
```

(continué en la próxima página)

(proviene de la página anterior)

```
            $this->Flash->error(__('Unable to add the user.'));
        }
        $this->set('user', $user);
    }
}
```

Creamos las vistas para nuestros artículos de la misma forma que el controlador, usando las herramientas de generación de código "bake", lo que nos permite implementar las vistas de los usuarios. Para el propósito de este tutorial, mostraremos solamente **add.php**:

```
<!-- templates/Users/add.php -->

<div class="users form">
<?= $this->Form->create($user) ?>
    <fieldset>
        <legend><?= __('Add User') ?></legend>
        <?= $this->Form->control('email') ?>
        <?= $this->Form->control('password') ?>
        <?= $this->Form->control('role', [
            'options' => ['admin' => 'Admin', 'author' => 'Author']
        ]) ?>
   </fieldset>
<?= $this->Form->button(__('Submit')); ?>
<?= $this->Form->end() ?>
</div>
```

Añadiendo Autenticación

Ya estamos listos para agregar nuestra autenticación. En CakePHP esto es manejado por el plugin `authentication`. Empezaremos instalándolo. Usa composer para instalar el plugin:

```
composer require "cakephp/authentication:^2.0"
```

Luego añade la siguiente línea en la función `bootstrap()` del archivo `Application.php`:

```
// in src/Application.php in the bootstrap() method.
$this->addPlugin('Authentication');
```

Añadiendo hashing de contraseña

Lo siguiente, creamos la entidad `User` y añadimos el hashing del password. Crear el archivo de la entidad **src/Model/Entity/User.php** y añade lo siguiente:

```
// src/Model/Entity/User.php
namespace App\Model\Entity;

use Cake\Auth\DefaultPasswordHasher;
use Cake\ORM\Entity;

class User extends Entity
```

(continué en la próxima página)

(proviene de la página anterior)

```
{
    // Make all fields mass assignable except for primary key field "id".
    protected array $_accessible = [
        '*' => true,
        'id' => false
    ];

    // ...

    protected function _setPassword($password)
    {
        if (strlen($password) > 0) {
            return (new DefaultPasswordHasher)->hash($password);
        }
    }

    // ...
}
```

Ahora, siempre que la propiedad `password` es asignada a la entidad User, se le aplicara el Hash usando la clase `DefaultPasswordHasher`.

Configurando Autenticación

Ahora es el momento de configurar el plugin Authentication. El plugin gestionará el proceso de autenticación usando 3 clases diferentes:

- `Application` usara el Authentication Middleware y proporciona un AuthenticationService, teniendo todas las configuraciones que queramos para definir como se van a comprobar las credenciales y donde encontrarlas.
- `AuthenticationService` es una clase de utilidad que te permite configurar el proceso de autenticación.
- `AuthenticationMiddleware` será ejecutado como parte de la cola de middleware, esto será antes de que tu controlador sea procesado por el framework, recogerá las credenciales y las procesará para comprobar si el usuario está autenticado.

La lógica de autenticación es dividida en clases específicas y el proceso se realiza antes de la capa del controlador. Primero, se comprueba si el usuario está autenticado (basado en la configuración proporcionada) e inyecta el usuario y el resultado de la autenticación en la petición para futura referencia.

En **src/Application.php**, añade los siguientes imports:

```
// In src/Application.php add the following imports
use Authentication\AuthenticationService;
use Authentication\AuthenticationServiceInterface;
use Authentication\AuthenticationServiceProviderInterface;
use Authentication\Middleware\AuthenticationMiddleware;
use Psr\Http\Message\ServerRequestInterface;
```

Luego implementa el interfaz de autenticación en tu clase Application:

```
// in src/Application.php
class Application extends BaseApplication
```

(continué en la próxima página)

(proviene de la página anterior)

```
    implements AuthenticationServiceProviderInterface
{
```

Después añade lo siguiente:

```
// src/Application.php
public function middleware(MiddlewareQueue $middlewareQueue): MiddlewareQueue
{
    $middlewareQueue
        // ... other middleware added before
        ->add(new RoutingMiddleware($this))
        // add Authentication after RoutingMiddleware
        ->add(new AuthenticationMiddleware($this));

    return $middlewareQueue;
}

public function getAuthenticationService(ServerRequestInterface $request):␣
↪AuthenticationServiceInterface
{
    $authenticationService = new AuthenticationService([
        'unauthenticatedRedirect' => '/users/login',
        'queryParam' => 'redirect',
    ]);

    // Carga los identificadores, asegurando que se comprueban los campos email y␣
↪password
    $authenticationService->loadIdentifier('Authentication.Password', [
        'fields' => [
            'username' => 'email',
            'password' => 'password',
        ],
    ]);

    // Carga los authenticators, quieres que la sesión comprueba primero
    $authenticationService->loadAuthenticator('Authentication.Session');
    // Configure form data check to pick email and password
    $authenticationService->loadAuthenticator('Authentication.Form', [
        'fields' => [
            'username' => 'email',
            'password' => 'password',
        ],
        'loginUrl' => '/users/login',
    ]);

    return $authenticationService;
}
```

En tu clase `AppController` añade el siguiente código:

```
// src/Controller/AppController.php
public function initialize(): void
```

(continué en la próxima página)

(proviene de la página anterior)

```
{
    parent::initialize();
    $this->loadComponent('Flash');

    // Añade ésta línea para comprobar la autenticación y asegurar tu aplicación
    $this->loadComponent('Authentication.Authentication');
```

Ahora, en cada petición, el `AuthenticationMiddleware` inspeccionará la sesión para comprobar si existe un usuario autenticado. Si estamos cargando la página `/users/login`, también inspeccionará los datos del formulario enviado en «post» (si hay alguno) para extraer las credenciales. Por defecto las credenciales se extraerán de los campos `email` y `password` de la información del request. El resultado de la autenticación será inyectado in un atributo de la petición llamado `authentication`. Puedes inspeccionar el resultado en cualquier momento usando `$this->request->getAttribute('authentication')` desde la acción de un controlador. Todas tus páginas serán restringidas ya que `AuthenticationComponent` está comprobando el resultado en cada petición. Cuando falla al buscar un usuario autenticado, redirigirá al usuario a la página `/users/login`. Te en cuenta que en éste punto del tutorial, la aplicación no funcionará ya que aún no tenemos la página de login. Si visitas tu aplicación, obtendrás un bucle infinito de redirección, asi que, vamos a arreglarlo!

En tu `UsersController`, añade el siguiente código:

```
public function beforeFilter(\Cake\Event\EventInterface $event)
{
    parent::beforeFilter($event);
    // Configura la acción de login para no requerir autenticación, para
    // prevenir el bucle infinito de redirección
    $this->Authentication->addUnauthenticatedActions(['login']);
}

public function login()
{
    $this->request->allowMethod(['get', 'post']);
    $result = $this->Authentication->getResult();
    // sin importar si es POST o GET, redirige si el usuario esta autenticado
    if ($result->isValid()) {
        // redirige a /articles después de iniciar sesión correctamente
        $redirect = $this->request->getQuery('redirect', [
            'controller' => 'Articles',
            'action' => 'index',
        ]);

        return $this->redirect($redirect);
    }
    // muestra los errors si el usuario envió el formulario y fallo la autenticación
    if ($this->request->is('post') && !$result->isValid()) {
        $this->Flash->error(__('Invalid email or password'));
    }
}
```

Añade la lógica de la vista para la acción de login:

```
<!-- in /templates/Users/login.php -->
<div class="users form">
    <?= $this->Flash->render() ?>
```

(continué en la próxima página)

(proviene de la página anterior)

```
    <h3>Login</h3>
    <?= $this->Form->create() ?>
    <fieldset>
        <legend><?= __('Please enter your email and password') ?></legend>
        <?= $this->Form->control('email', ['required' => true]) ?>
        <?= $this->Form->control('password', ['required' => true]) ?>
    </fieldset>
    <?= $this->Form->submit(__('Login')); ?>
    <?= $this->Form->end() ?>

    <?= $this->Html->link("Add User", ['action' => 'add']) ?>
</div>
```

Ahora la página de login nos permitirá iniciar sesión en la aplicación. Compruébalo haciendo una petición a cualquier página de tu aplicación. Después de haber sido redirigido a la página `/users/login`, introduce el email y password que usaste previamente para crear el usuario. Deberías ser redirigido correctamente después de iniciar sesión.

Necesitamos añadir un par de detalles más para configurar nuestra aplicación. Queremos que todas las páginas `view` e `index` sean accesible sin necesitar iniciar sesión, así que añadiremos ésta configuración específica en `AppController`:

```
// en src/Controller/AppController.php
public function beforeFilter(\Cake\Event\EventInterface $event)
{
    parent::beforeFilter($event);
    // para todos los controladores de nuestra aplicación, hacer el index y view
    // acciones públicas, saltándonos la autenticación
    $this->Authentication->addUnauthenticatedActions(['index', 'view']);
}
```

Terminar sesión

Añade la acción `logout` a la clase `UsersController`:

```
// en src/Controller/UsersController.php
public function logout()
{
    $result = $this->Authentication->getResult();
    // sin importar si es POST o GET, redirige si el usuario esta autenticado
    if ($result->isValid()) {
        $this->Authentication->logout();

        return $this->redirect(['controller' => 'Users', 'action' => 'login']);
    }
}
```

Ahora puedes visitar `/users/logout` para terminar la sesión. Luego serás redirigido a la página de login. Si has llegado tan lejos, felicidades, ahora tienes un blog simple que:

- Permite usuarios autenticados crear y editar artículos.
- Permite usuarios no autenticados ver artículos y etiquetas.

Lectura sugerida

1. /bake/usage Generar código CRUD básico
2. Authentication Plugin documentación.

CAPÍTULO 4

Contribuir

Existen diversas maneras con las que puedes contribuir a CakePHP:

Documentación

Contribuir con la documentación es fácil. Los archivos están hospedados en https://github.com/cakephp/docs. Siéntete libre de hacer un *fork* del repositorio, añadir tus cambios, mejoras, traducciones y comenzar a ayudar a través de un nuevo *pull request*. También puedes editar los archivos de manera online con GitHub sin la necesidad de descargarlos – el botón *Improve this Doc* que aparece en todas las páginas te llevará al editor online de GitHub de esa página.

La documentación de CakePHP dispone de integración continua[40] y se despliega automáticamente tras realizar el *merge* del *pull request*.

Traducciones

Envía un email al equipo de documentación (docs *arroba* cakephp *punto* org) o utiliza IRC (#cakephp en *freenode*) para hablar de cualquier trabajo de traducción en el que quieras participar.

[40] https://es.wikipedia.org/wiki/Integraci%C3%B3n_continua

Nueva traducción

Nos gustaría poder disponer de traducciones que estén todo lo completas posible. Sin embargo, hay ocasiones donde un archivo de traducción no está al día, por lo que debes considerar siempre la versión en inglés como la versión acreditada.

Si tu idioma no está entre los disponibles, por favor, contacta con nosotros a través de Github y estudiaremos la posibilidad de crear la estructura de archivos para ello.

Las siguientes secciones son las primeras que deberías considerar traducir, ya que estos archivos no cambian a menudo:

- index.rst
- intro.rst
- quickstart.rst
- installation.rst
- /intro (carpeta)
- /tutorials-and-examples (carpeta)

Recordatorio para administradores de documentación

La estructura de archivos de todos los idiomas deben seguir la estructura de la versión en inglés. Si la estructura cambia en esta versión debemos realizar dichos cambios en los demás idiomas.

Por ejemplo, si se crea un nuevo archivo en inglés en **en/file.rst** tendremos que:

- Añadir el archivo en todos los idiomas: **fr/file.rst**, **zh/file.rst**, ...
- Borrar el contenido pero manteniendo el `title`, `meta` información y `toc-tree` que pueda haber. Se añadirá la siguiente nota mientras nadie traduzca el archivo:

```
File Title
##########

.. note::
    The documentation is not currently supported in XX language for this
    page.

    Please feel free to send us a pull request on
    `Github <https://github.com/cakephp/docs>`_ or use the **Improve This Doc**
    button to directly propose your changes.

    You can refer to the English version in the select top menu to have
    information about this page's topic.

// If toc-tree elements are in the English version
.. toctree::
    :maxdepth: 1

    one-toc-file
    other-toc-file

.. meta::
    :title lang=xx: File Title
    :keywords lang=xx: title, description,...
```

Consejos para traductores

- Navega y edita en el idioma al que quieras traducir el contenido - de otra manera no verás lo que ya está traducido.
- Siéntete libre de bucear en la traducción si ya existe en tu idioma.
- Usa la Forma informal[41].
- Traduce el título y el contenido a la vez.
- Compara con la versión en inglés antes de subir una corrección (si corriges algo, pero no indicas una referencia tu subida no será aceptada).
- Si necesitas escribir un término en inglés envuélvelo en etiquetas `<em>`. E.g. «asdf asdf *Controller* asdf» o «asdf asdf Kontroller (*Controller*) asfd» como proceda.
- No subas traducciones parciales.
- No edites una sección con cambios pendientes.
- No uses entidades HTML[42] para caracteres acentudados, la documentación utiliza UTF-8.
- No cambies significatibamente el etiquetado (HTML) o añadas nuevo contenido.
- Si falta información en el contenido original sube primero una corrección de ello.

Guía de formato para la documentación

La nueva documentación de CakePHP está escrito con texto en formato ReST[43].

ReST (*Re Structured Text*) es una sintaxis de marcado de texto plano similar a *Markdown* o *Textile*.

Para mantener la consistencia cuando añadas algo a la documentación de CakePHP recomendamos que sigas las siguientes líneas guía sobre como dar formato y estructurar tu texto.

Tamaño de línea

Las líneas de texto deberían medir como máximo 40 caracteres. Las únicas excepciones son URL largas y fragmentos de código.

Cabeceras y secciones

Las cabeceras de las secciones se crean subrayando el título con caracteres de puntuación. El subrayado deberá ser por lo menos tan largo como el texto.

- # Se utiliza para indicar los títulos de páginas.
- = Se utiliza para los títulos de las secciones de una página.
- - Se utiliza para los títulos de subsecciones.
- ~ Se utiliza para los títulos de sub-subsecciones.
- ^ Se utiliza para los títulos de sub-sub-subsecciones.

Los encabezados no deben anidarse con más de 5 niveles de profundidad y deben estar precedidos y seguidos por una línea en blanco.

[41] https://es.wikipedia.org/wiki/Registro_ling%C3%BC%C3%ADstico
[42] https://es.wikipedia.org/wiki/Anexo:Entidades_de_caracteres_XML_y_HTML
[43] https://es.wikipedia.org/wiki/ReStructuredText

Párrafos

Párrafos son simplemente bloques de texto con todas las líneas al mismo nivel de indexación. Los párrafos deben separarse por al menos una línea vacía.

Marcado en línea

- Un asterisco: *texto* en cursiva. Lo usaremos para enfatizar/destacar de forma general.
  - `*texto*`.
- Dos astericos: **texto** en negrita. Lo usaremos para indicar directorios de trabajo, títulos de listas y nombres de tablas (excluyendo la palabra *table*).
  - `**/config/Migrations**`, `**articulos**`, etc.
- Dos acentos graves (``): `texto` para ejemplos de código. Lo usaramos para nombres de opciones de métodos, columnas de tablas, objetos (excluyendo la palabra «objeto») y para nombres de métodos y funciones (incluídos los paréntesis)
  - ``` ``cascadeCallbacks`` ```, ``` ``true`` ```, ``` ``id`` ```, ``` ``PagesController`` ```, ``` ``config()`` ```, etc.

Si aparecen asteriscos o acentos graves en el texto y pueden ser confundidos con los delimitadores de marcado habrá que escaparlos con \.

Los marcadores en línea tienen algunas restricciones:

- **No pueden** estar anidados.
- El contenido no puede empezar o acabar con espacios en blanco: `* texto*` está mal.
- El contenido debe separarse del resto del texto por caracteres que no sean palabras. Utiliza \ para escapar un espacio y solucionarlo: `onelong\ *bolded*\ word`.

Listas

El etiquetado de listas es muy parecido a *Markdown*. Las listas no ordenadas se indican empezando una línea con un asterisco y un espacio.

Las listas enumeradas pueden crearse con enumeraciones o # para auto enumeración:

- **Esto es una viñeta**
  - Esto también, pero esta línea tiene dos líneas.

1. **Primera línea**
   2. Segunda línea
2. La enumeración automática
3. Te ahorrará algo de tiempo.

También se pueden crear listas anidadas tabulando secciones y separándolas con una línea en blanco:

```
* Primera línea
* Segunda línea

    * Bajando un nivel
    * Yeah!
```

(continué en la próxima página)

(proviene de la página anterior)

```
* Volviendo al primer nivel
```

Pueden crearse listas de definiciones haciendo lo siguiente:

```
Término
    Definición
CakePHP
    Un framework MVC para PHP
```

Los términos no pueden ocupar más de una línea, pero las definiciones pueden ocupar más líneas mientras se aniden consistentemente.

Enlaces

Hay diferentes tipos de enlaces, cada uno con sus características.

Enlaces externos

Los enlaces a documentos externos pueden hacerse de la siguiente manera:

```
`Enlace externo a php.net <https://php.net>`_
```

El resultado debería verse así: Enlace externo a php.net[44]

Enlaces a otras páginas

`:doc:`

Puedes crear enlaces a otras páginas de la documentación usando la función `::doc:`. Puedes enlazar a un archivo específico empleando rutas relativas o absolutas omitiendo la extensión `.rst`. Por ejemplo: si apareciese `:doc:`form`` en el documento `core-helpers/html`, el enlace haría referencia a `core-helpers/form`. Si la referencia fuese `:doc:`/core-helpers`` el enlace sería siempre a `/core-helpers` sin importar donde se utilice.

Enlaces a referencias cruzadas

`:ref:`

Puedes hacer referncia cruzada a cualquier título de cualquier documento usando la función `:ref:`. Los enlaces a etiquetas de destino deben ser únicos a lo largo de toda la documentación. Cuando se crean etiquetas para métodos de clase lo mejor es usar `clase-método` como formato para tu etiqueta de destino.

El uso más habitual de etiquetas es encima de un título. Ejemplo:

```
.. _nombre-etiqueta:

Título sección
--------------
```

(continué en la próxima página)

[44] https://php.net

(proviene de la página anterior)

```
Resto del contenido.
```

En otro sitio podrías enlazar a la sección de arriba usando `:ref:`nombre-etiqueta``. El texto del enlace será el título al que precede el enlace pero puedes personalizarlo usando `:ref:`Texto del enlace <nombre-etiqueta>``.

Evitar alertas de Sphinx

Sphinx mostrará avisos si un archivo no es referenciado en un *toc-tree*. Es una buena manera de asegurarse de que todos los archivos tienen un enlace dirigido a ellos. Pero a veces no necesitas introducir un enlace a un archivo, p.ej. para nuestros archivos *epub-contents* y *pdf-contents*. En esos casos puedes añadir `:orphan:` al inicio del archivo para eliminar las alertas de que el archivo no está en el *toc-tree*

Describir clases y sus contenidos

La documentación de CakePHP usa el phpdomain[45] para proveer directivas personalizadas para describir objetos PHP y constructores. El uso de estas directivas y funciones es necesario para una correcta indexación y uso de las herramientas de referenciación cruzada.

Describir clases y constructores

Cada directiva introduce el contenido del índice y/o índice del *namespace*.

`.. php:global:: nombre`
: Esta directiva declara una nueva variable PHP global.

`.. php:function:: nombre(firma)`
: Define una nueva función global fuera de una clase.

`.. php:const:: nombre`
: Esta directiva declara una nueva constante PHP, puedes usarla también anidada dentro de una directiva de clase para crear constantes de clase.

`.. php:exception:: nombre`
: Esta directiva declara una nueva excepción en el *namespace* actual. La firma puede incluir argumentos de constructor.

`.. php:class:: nombre`
: Describe una clase. Métodos, atributos y atributos que pertenezcan a la clase deberán ir dentro del cuerpo de la directiva:

```
.. php:class:: MyClass

    Descripción de la clase

   .. php:method:: method($argument)

   Descripción del método
```

[45] https://pypi.org/project/sphinxcontrib-phpdomain/

Atributos, métodos y constantes no necesitan estar anidados, pueden seguir la siguiente declaración de clase:

```
.. php:class:: MyClass

    Texto sobre la clase

.. php:method:: methodName()

    Texto sobre el método
```

.. php:method:: nombre(firma)

Describe un método de clase, sus argumentos, salida y excepciones:

```
.. php:method:: instanceMethod($one, $two)

    :param string $one: El primer parámetro.
    :param string $two: El segundo parámetro.
    :returns: Un array de cosas
    :throws: InvalidArgumentException

   Esto es una instancia de método.
```

.. php:staticmethod:: ClassName::nombreMetodo(firma)

Describe un método estático, sus argumentos, salida y excepciones, ver *php:method* para opciones.

.. php:attr:: nombre

Describe una propiedad/atributo en una clase.

Evitar avisos de Sphinx

Sphinx mostrará avisos si una función es referenciada en múltiples archivos. Es una buena manera de asegurarse de que no añades una función dos veces, pero algunas veces puedes querer escribir una función en dos o más archivos, p.ej. *"debug object"* es referenciado en *`/development/debugging`* y *`/core-libraries/global-constants-and-functions`*. En este caso tu puedes añadir `:noindex:` debajo de la función *debug* para eliminar los avisos. Mantén únicamente una referencia **sin** `:no-index:` para seguir teniendo la función referenciada:

```
.. php:function:: debug(mixed $var, boolean $showHtml = null, $showFrom = true)
    :noindex:
```

Referencias cruzadas

Los siguientes *roles* hacen referencia a objetos PHP y los enlaces son generados si se encuentra una directiva que coincida:

:php:func:

Referencia a una función PHP.

:php:global:

Referencia a una variable global cuyo nombre tiene prefijo $.

:php:const:

Referencia tanto a una constante global como a una de clase. Las constantes de clase deberán ir precedidas por la clase que las contenga:

```
DateTime tiene una constante :php:const:`DateTime::ATOM`.
```

:php:class:

Referencia una clase por el nombre:

```
:php:class:`ClassName`
```

:php:meth:

Referencia un método de una clase. Este *role* soporta ambos tipos de métodos:

```
:php:meth:`DateTime::setDate`
:php:meth:`Classname::staticMethod`
```

:php:attr:

Referencia una propiedad de un objeto:

```
:php:attr:`ClassName::$propertyName`
```

:php:exc:

Referencia una excepción.

Código fuente

Los bloques de citas de código fuente se crean finalizando un párrafo con `::`. El bloque debe ir anidado y, como todos los párrafos, separados por líneas en blanco:

```
Esto es un párrafo::

    while ($i--) {
        doStuff()
    }

Esto es otra vez texto normal.
```

Los textos citados no son modificados ni formateados salvo el primer nivel de anidamiento, que es eliminado.

Notas y avisos

Hay muchas ocasiones en las que quieres avisar al lector de un consejo importante, una nota especial o un peligro potencial. Las admonestaciones en *Sphinx* se utilizan justo para eso. Hay cinco tipos de admonestaciones:

- `.. tip::` Los consejos (*tips*) se utilizan para documentar o reiterar información interesante o importante. El contenido de la directiva debe escribirse en sentencias completas e incluir todas las puntuaciones apropiadas.
- `.. note::` Las notas (*notes*) se utilizan para documentar una pieza de información importante. El contenido de la directiva debe escribirse en sentencias completas e incluir todas las puntuaciones apropiadas.
- `.. warning::` Avisos (*warnings*) se utilizan para documentar posibles obstáculos o información relativa a seguridad. El contenido de la directiva debe escribirse en sentencias completas e incluir todas las puntuaciones apropiadas.
- `.. versionadded:: X.Y.Z` las admonestaciones *«Version added»* se utilizan para mostrar notas específicas a nuevas funcionalidades añadidas en una versión específica, siendo `X.Y.Z` la versión en la que se añadieron.

- `.. deprecated:: X.Y.Z` es lo opuesto a *versionadded*, se utiliza para avisar de una funcionalidad obsoleta, siendo `X.Y.Z` la versión en la que pasó a ser obsoleta.

Todas las admonestaciones se escriben igual:

```
.. note::

    Anidado y precedido por una línea en blanco.
    Igual que un párafo.

Este texto no es parte de la nota.
```

Ejemplos

Truco: Esto es un consejo útil que probablemente hayas olvidado.

Nota: Deberías prestar atención aquí.

Advertencia: Podría ser peligroso.

Nuevo en la versión 4.0.0: Esta funcionalidad tan genial fue añadida en la versión 4.0.0

Obsoleto desde la versión 4.0.1: Esta antigua funcionalidad pasó a ser obsoleta en la versión 4.0.1

Tickets

Aportar *feedback* y ayudar a la comunidad en la forma de tickets es una parte extremadamente importante en el proceso de desarrollo de CakePHP. Todos los tickets de CakePHP están alojados en GitHub[46].

Reportar errores

Los reportes de errores bien escritos son de mucha ayuda. Para ello hay una serie de pasos que ayudan a crear el mejor reporte de error posible:

- **Correcto**: Por favor, busca tickets[47] similares que ya existan y asegúrate de que nadie haya reportado ya tu problema o que no haya sido arreglado en el repositorio.
- **Correcto**: Por favor, incluye instrucciones detalladas de **cómo reproducir el error**. Esto podría estar escrito en el formato de caso de prueba o con un **snippet** de código que demuestre el problema. No tener una forma de reproducir el error significa menos probabilidades de poder arreglarlo.
- **Correcto**: Por favor, danos todos los detalles posibles de tu entorno: sistema operativo, versión de PHP, versión de CakePHP…

[46] https://github.com/cakephp/cakephp/issues
[47] https://github.com/cakephp/cakephp/search?q=it+is+broken&ref=cmdform&type=Issues

- **Incorrecto**: Por favor, no utilices el sistema de tickets para hacer preguntas de soporte. El canal #cakephp IRC en Freenode[48] tiene muchos desarrolladores disponibles para ayudar a responder tus preguntas. También échale un vistazo a Stack Overflow[49].

Reportar problemas de seguridad

Si has encontrado problemas de seguridad en CakePHP, por favor, utiliza el siguiente procedimiento en vez del sistema de reporte de errores. En vez de utilizar el *tracker* de errores, lista de correos o IRC, por favor, envía un email a **security [at] cakephp.org**. Los emails enviados a esta dirección van al equipo principal de CakePHP en una lista de correo privada.

Por cada reporte primero tratamos de confirmar la vulnerabilidad, una vez confirmada el equipo de CakePHP tomará las siguientes acciones:

- Dar a conocer al reportador que hemos recibido el problema y que estamos trabajando en una solución. Pediremos al reportador que mantenga en secreto el problema hasta que nosotros lo anunciemos.
- Preparar una solución/parche.
- Preparar un *post* describiendo la vulnerabilidad y las posibles consecuencias.
- Publicar nuevas versiones para todas las que estén afectadas.
- Mostrar de manera acentuada el problema en el anuncio de la publicación.

Código

Parches y *pull requests* son una manera genial de contribuir con código a CakePHP. Los *Pull requests* pueden ser creados en Github, preferiblemente a los archivos de parches en los comentarios de tickets.

Configuración inicial

Antes de trabajar en parches para CakePHP es una buena idea configurar tu entorno de trabajo.

Necesitarás los siguientes programas:

- Git
- PHP 8.1 o mayor
- PHPUnit 5.7.0 o mayor

Configura tu información de usuario con tu nombre/alias y correo electrónico de trabajo:

```
git config --global user.name 'Bob Barker'
git config --global user.email 'bob.barker@example.com'
```

Nota: Si eres nuevo en Git, te recomendamos encarecidamente que leas el maravilloso y gratuito libro ProGit[50]

Clona el código fuente de CakePHP desde GitHub:

[48] https://webchat.freenode.net
[49] https://stackoverflow.com/questions/tagged/cakephp
[50] https://git-scm.com/book/

- Si no tienes una cuenta de GitHub[51] créate una.
- Haz un *fork* del repositorio CakePHP[52] haciendo clic en el botón **Fork**.

Después de haber hecho el fork, clónalo en tu equipo local:

```
git clone git@github.com:TUNOMBRE/cakephp.git
```

Añade el repositorio original de CakePHP como respositorio remoto, lo usarás más adelante para buscar cambios en el repositorio de CakePHP. Esto te mantendrá actualizado con CakePHP:

```
cd cakephp
git remote add upstream git://github.com/cakephp/cakephp.git
```

Ahora que tienes configurado CakePHP deberías poder definir un `$test` de *conexión de base de datos* y *ejecutar todos los tests*.

Trabajar en un parche

Cada vez que quieras trabajar en un bug, una funcionalidad o en una mejora, crea una rama específica.

Tu rama debería ser creada a partir de la versión que quieras arreglar/mejorar. Por ejemplo, si estás arreglando un error en la versión `3.x` deberías utilizar la rama `master` como rama origen. Si tu cambio es para un error de la serie 2.x deberías usar la rama `2.x`. Esto hará más adelante tus *merges* más sencillos al no permitirte Github editar la rama destino:

```
# arreglando un error en 3.x
git fetch upstream
git checkout -b ticket-1234 upstream/master

# arreglando un error en 2.x
git fetch upstream
git checkout -b ticket-1234 upstream/2.x
```

Truco: Usa un nombre descriptivo para tu rama, referenciar el ticket o nombre de la característica es una buena convención. P.ej. ticket-1234, nueva-funcionalidad

---

Lo anterior creará una rama local basada en la rama *upstream* 2.x (CakePHP)

Trabaja en tu correción y haz tantos *commits* como necesites, pero ten siempre en mente lo siguiente:

- Sigue las *Estándares de codificación*.
- Añade un caso de prueba para mostrar el error arreglado o que la nueva funcionalidad funciona.
- Mantén lógicos tus commits y escribe comentarios de *commit* bien claros y concisos.

[51] https://github.com
[52] https://github.com/cakephp/cakephp

Enviar un *Pull Request*

Una vez estén hechos tus cambios y estés preparado para hacer el *merge* con CakePHP tendrás que actualizar tu rama:

```
# Hacer rebase de la corrección en el top de master
git checkout master
git fetch upstream
git merge upstream/master
git checkout <nombre_rama>
git rebase master
```

Esto buscará y hará *merge* de cualquier cambio que haya sucedido en CakePHP desde que empezaste. Entonces ejecutará *rebase* o replicará tus cambios en el *top* del actual código.

Puede que encuentres algún conflicto durante el *rebase*. Si este finaliza precipitadamente puedes ver qué archivos son conflictivos/*un-merged* con `git status`. Resuelve cada conflicto y continúa con el *rebase*:

```
git add <nombre_archivo> # haz esto con cada archivo conflictivo.
git rebase --continue
```

Comprueba que todas tus pruebas continúan pasando. Entonces sube tu rama a tu *fork*:

```
git push origin <nombre-rama>
```

Si has vuelto a hacer *rebase* después de hacer el *push* de tu rama necesitarás forzar el *push*:

```
git push --force origin <nombre-rama>
```

Una vez tu rama esté en GitHub puedes enviar un *pull request* en GitHub.

Seleccionar donde harán el *merge* tus cambios

Cuando hagas *pull requests* deberás asegurarte de seleccionar la rama correcta como base ya que no podrás editarla una vez creada.

- Si tus cambios son un *bugfix* (corrección de error) y no introduce ninguna funcionalidad nueva entonces selecciona **master** como destino del merge.
- Si tu cambio es una *new feature* (nueva funcionalidad) o un añadido al framework entonces deberías seleccionar la rama con el número de la siguiente versión. Por ejemplo si la versión estable actualmente es la `3.2.10`, la rama que estará aceptando nuevas funcionalidades será la `3.next`.
- Si tu cambio cesa una funcionalidad existente o de la *API* entonces tendrás que escojer la versión mayor siguiente. Por ejemplo, si la actual versión estable es la `3.2.2` entonces la siguiente versión en la que se puede cesar es la `4.x` por lo que deberás seleccionar esa rama.

Nota: Recuerda que todo código que contribuyas a CakePHP será licenciado bajo la Licencia MIT, y la Cake Software Foundation[53] será la propietaria de cualquier código contribuido. Los contribuidores deberán seguir las Guías de la comunidad CakePHP[54].

Todos los *merge* de corrección de errores que se hagan a una rama de mantenimiento se harán también periódicamente sobre futuros lanzamientos por el equipo central.

[53] https://cakefoundation.org/old
[54] https://cakephp.org/get-involved

Estándares de codificación

Los desarrolladores de CakePHP deberán utilizar la Guia de estilo de codificación PSR-12[55] además de las siguientes normas como estándares de codificación.

Es recomendable que otos *CakeIngredients* que se desarrollen sigan los mismos estándares.

Puedes utilizar el CakePHP Code Sniffer[56] para comprobar que tu código siga los estándares requeridos.

Añadir nuevas funcionalidades

Las nuevas funcionalidades no se deberán añadir sin sus propias pruebas, las cuales deberán ser superadas antes de hacer el *commit* en el repositorio.

Configuración del *IDE*

Asegúrate de que tu *IDE* haga *trim* por la derecha para que no haya espacios al final de las líneas.

La mayoría de los *IDE* modernos soportan archivos `.editorconfig`. El esqueleto de aplicación de CakePHP viene con él por defecto y contiene las mejores prácticas de forma predeterminada.

Tabulación

Se utilizará cuatro espacios para la tabulación.

Por lo que debería everse asi:

```
// nivel base
    // nivel 1
        // nivel 2
    // nivel 1
// nivel base
```

O también:

```
$booleanVariable = true;
$stringVariable = 'moose';
if ($booleanVariable) {
    echo 'Boolean value is true';
    if ($stringVariable === 'moose') {
        echo 'We have encountered a moose';
    }
}
```

En los casos donde utilices llamadas de funciones que ocupen más de una línea usa las siguientes guías:

- El paréntesis de abertura de la llamada de la función deberá ser lo último que contenga la línea.
- Sólo se permite un argumento por línea.
- Los paréntesis de cierre deben estar solos y en una línea por separado.

Por ejemplo, en vez de utilizar el siguiente formato:

[55] https://www.php-fig.org/psr/psr-12/
[56] https://github.com/cakephp/cakephp-codesniffer

```
$matches = array_intersect_key($this->_listeners,
               array_flip(preg_grep($matchPattern,
                   array_keys($this->_listeners), 0)));
```

Utiliza éste en su lugar:

```
$matches = array_intersect_key(
    $this->_listeners,
    array_flip(
        preg_grep($matchPattern, array_keys($this->_listeners), 0)
    )
);
```

Tamaño de línea

Es recomendable mantener un tamaño de 100 caracteres por línea para una mejor lectura del código y tratar de no pasarse de los 120.

En resumen:

- 100 caracteres es el límite recomendado.
- 120 caracteres es el límite máximo.

Estructuras de control

Las estructuras de control son por ejemplo «`if`», «`for`», «`foreach`», «`while`», «`switch`» etc. A continuación un ejemplo con «`if`»:

```
if ((expr_1) || (expr_2)) {
    // accion_1;
} elseif (!(expr_3) && (expr_4)) {
    // accion_2;
} else {
    // accion_por_defecto;
}
```

- En las estructuras de control deberá haber un espacio antes del primer paréntesis y otro entre el último y la llave de apertura.
- Utiliza siempre las llaves en las estructuras de control incluso si no son necesarias. Aumentan la legibilidad del código y te proporcionan menos errores lógicos.
- Las llaves de apertura deberán estar en la misma línea que la estructura de control, las de cierre en líneas nuevas y el código dentro de las dos llaves en un nuevo nivel de tabulación.
- No deberán usarse las asignaciones *inline* en las estructras de control.

```
// Incorrecto: sin llaves y declaración mal posicionada
if (expr) statement;

// Incorrecto: sin llaves
if (expr)
    statement;
```

(continué en la próxima página)

(proviene de la página anterior)

```
// Correcto
if (expr) {
    statement;
}

// Incorrecto = asignación inline
if ($variable = Class::function()) {
    statement;
}

// Correcto
$variable = Class::function();
if ($variable) {
    statement;
}
```

Operador ternario

Los operadores ternarios están permitidos cuando toda su declaración cabe en una sola línea. Operadores más largos deberán ir dentro de una declaración `if else`. Los operadores ternarios no deberían ir nunca anidados y opcionalmente pueden utilizarse paréntesis alrededor de las condiciones para dar claridad:

```
// Correcto, sencillo y legible
$variable = isset($options['variable']) ? $options['variable'] : true;

// Incorrecto, operadores anidados
$variable = isset($options['variable']) ? isset($options['othervar']) ? true : false :␣
↪false;
```

Archivos de plantilla

En los archivos de plantilla (archivos .php) los desarrolladores deben utilizar estructuras de control `keyword` al ser más fáciles de leer en archivos complejos. Las estructuras de control pueden estar dentro de bloques de PHP o en etiquetas PHP separadas:

```
<?php
if ($esAdmin):
    echo '<p>Eres el usuario admin.</p>';
endif;
?>
<p>Lo siguiente es aceptado también:</p>
<?php if ($esAdmin): ?>
    <p>Eres el usuario admin.</p>
<?php endif; ?>
```

Comparación

Intenta ser siempre lo más estricto posible. Si una comparación no es estricta de forma deliberada, puede ser inteligente añadir un comentario para evitar confundirla con un error.

Para comprobar si una variable es `null` se recomienda utilizar la comprobación estricta:

```
if ($value === null) {
    // ...
}
```

El valor contra el que se va a realizar la comparación deberá ir en el lado derecho de esta:

```
// no recomendado
if (null === $this->foo()) {
    // ...
}

// recomendado
if ($this->foo() === null) {
    // ...
}
```

Llamadas de funciones

Las llamadas a funciones deben realizarse sin espacios entre el nombre de la función y el parentesis de apertura y entre cada parámetro de la llamada deberá haber un espacio:

```
$var = foo($bar, $bar2, $bar3);
```

Como puedes ver arriba también deberá haber un espacio a ambos lados de los signos de igual.

Definición de métodos

Ejemplo de definición de un método:

```
public function someFunction($arg1, $arg2 = '')
{
    if (expr) {
        statement;
    }

    return $var;
}
```

Parámetros con un valor por defecto deberán ir al final de las definiciones. Trata que tus funciones devuelvan siempre un resultado, al menos `true` o `false`, para que se pueda determinar cuando la llamada a la función ha sido correcta:

```
public function connection($dns, $persistent = false)
{
    if (is_array($dns)) {
        $dnsInfo = $dns;
```

(continué en la próxima página)

(proviene de la página anterior)

```
    } else {
        $dnsInfo = BD::parseDNS($dns);
    }

    if (!($dnsInfo) || !($dnsInfo['phpType'])) {
        return $this->addError();
    }

    return true;
}
```

Como puedes ver hay un espacio a ambos lados del signo de igual.

Declaración de tipo

Los argumentos que esperan objetos, arrays o `callbacks` pueden ser tipificados. Solo tipificamos métodos públicos, aunque la tipificación no está libre de costes:

```
/**
 * Alguna descripción del método
 *
 * @param \Cake\ORM\Table $table La clase table a utilizar.
 * @param array $array Algún valor array.
 * @param callable $callback Algún callback.
 * @param bool $boolean Algún valor boolean.
 */
public function foo(Table $table, array $array, callable $callback, $boolean)
{
}
```

Aquí `$table` debe ser una instancia de `\Cake\ORM\Table`, `$array` debe ser un `array` y `$callback` debe ser de tipo `callable` (un `callback` válido).

Fíjate en que si quieres permitir que `$array` sea también una instancia de `\ArrayObject` no deberías tipificarlo ya que `array` acepta únicamente el tipo primitivo:

```
/**
 * Alguna descripción del método.
 *
 * @param array|\ArrayObject $array Algún valor array.
 */
public function foo($array)
{
}
```

Funciones anónimas (`Closures`)

Para definir funciones anónimas sigue la guía de estilo de código PSR-12[57] , donde se declaran con un espacio después de la palabra `function` y antes y después de la palabra `use`:

```
$closure = function ($arg1, $arg2) use ($var1, $var2) {
    // código
};
```

Encadenación de métodos

Las encadenaciones de métodos deberán distribuir estos en líneas separadas y tabulados con cuatro espacios:

```
$email->from('foo@example.com')
    ->to('bar@example.com')
    ->subject('A great message')
    ->send();
```

Comentar el código

Todos los comentarios deberán ir escritos en inglés y describir de un modo claro el bloque de código comentado.

Los comentarios pueden incluir las siguientes etiquetas de phpDocumentor[58]:

- @deprecated[59] Usando el formato `@version <vector> <description>`, donde `version` y `description` son obligatorios.
- @example[60]
- @ignore[61]
- @internal[62]
- @link[63]
- @see[64]
- @since[65]
- @version[66]

Las etiquetas PhpDoc son muy similares a las etiquetas JavaDoc en Java. Las etiquetas solo son procesadas si son el primer elemento en una línea DocBlock, por ejemplo:

```
/**
 * Ejemplo de etiqueta.
 *
```

(continué en la próxima página)

[57] https://www.php-fig.org/psr/psr-12/
[58] https://phpdoc.org
[59] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/deprecated.html
[60] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/example.html
[61] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/ignore.html
[62] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/internal.html
[63] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/link.html
[64] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/see.html
[65] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/since.html
[66] https://docs.phpdoc.org/latest/guide/references/phpdoc/tags/version.html

(proviene de la página anterior)

```
 * @author esta etiqueta es parseada, pero esta @version es ignorada
 * @version 1.0 esta etiqueta es parseada también
 */
```

```
/**
 * Ejemplo de etiquetas phpDoc inline.
 *
 * Esta función trabaja duramente con foo() para manejar el mundo.
 *
 * @return void
 */
function bar()
{
}

/**
 * Función foo.
 *
 * @return void
 */
function foo()
{
}
```

Los bloques de comentarios, con la excepción del primer bloque en un archivo, deberán ir siempre precedidos por un salto de línea.

Tipos de variables

Tipos de variables para utilizar en DocBlocks:

Tipo
: Descripción

mixed
: Una variable de tipo indefinido o múltiples tipos.

int
: Variable de tipo integer (números enteros).

float
: Tipo float (número de coma flotante).

bool
: Tipo booleano (true o false).

string
: Tipo string (cualquier valor entre « « o “ “).

null
: Tipo null. Normalmente usado conjuntamente con otro tipo.

array
: Tipo array.

object
: Tipo object. Debe usarse un nombre de clase específico si es posible.

resource
: Tipo resource (devuelto por ejemplo por mysql_connect()). Recuerda que cuando especificas el tipo como mixed deberás indicar si es desconocido o cuáles son los tipos posibles.

callable
: Función Callable.

Puedes combinar tipos usando el caracter `|`:

```
int|bool
```

Para más de dos tipos normalmente lo mejor es utilizar `mixed`.

Cuando se devuelva el propio objeto, por ejemplo para encadenar, deberás utilizar `$this` en su lugar:

```
/**
 * Función foo.
 *
 * @return $this
 */
public function foo()
{
    return $this;
}
```

Incluir archivos

`include`, `require`, `include_once` y `require_once` no tienen paréntesis:

```
// mal = paréntesis
require_once('ClassFileName.php');
require_once ($class);

// bien = sin paréntesis
require_once 'ClassFileName.php';
require_once $class;
```

Cuando se incluyan archivos con clases o librerías usa siempre y únicamente la función require_once[67].

Etiquetas PHP

Utiliza siempre las etiquetas <?php y ?> en lugar de <? y ?>.

La sintaxis abreviada de `echo` deberá usarse en los archivos de plantilla (**.php**) donde proceda.

[67] https://php.net/require_once

Sintaxis abreviada de echo

La sintaxis abreviada de `echo` (<?=) deberá usarse en los archivos de plantillas en lugar de `<?php echo`. Deberá ir seguido inmediatamente por un espacio, la variable o valor de la función a imprimir, un espacio y la etiqueta php de cierre:

```
// mal = con punto y coma y sin espacios
<td><?=$name;?></td>

// bien = con espacios y sin punto y coma
<td><?= $name ?></td>
```

A partir de la versión 5.4 de PHP la etiqueta (<?=) no es considerada un `short tag` y está siempre disponible sin importar la directiva `ini` de `short_open_tag`.

Convenciones de nomenclatura

Funciones

Escribe todas las funciones en `camelBack`:

```
function nombreFuncionLargo()
{
}
```

Clases

Los nombres de las clases deberán escribirse en `CamelCase`, por ejemplo:

```
class ClaseEjemplo
{
}
```

Variables

Los nombres de variables deberán ser todo lo descriptibles que puedan, pero también lo más corto posible. Se escribirán en minúscula salvo que estén compuestos por múltiples palabras, en cuyo caso irán en `camelBack`. Los nombres de las variables que referencien objetos deberán ir asociados de algún modo a la clase de la cual es objeto. Ejemplo:

```
$usuario = 'John';
$usuarios = ['John', 'Hans', 'Arne'];

$dispatcher = new Dispatcher();
```

Visibilidad de miembros

Usa las palabras clave `public`, `protected` y `private` de PHP para métodos y variables.

Direcciones de ejemplos

Para los ejemplos de URL y correos electrónicos usa «example.com», «example.org» y «example.net», por ejemplo:

- Email: someone@example.com
- WWW: http://www.example.com
- FTP: ftp://ftp.example.com

El nombre de dominio «example.com» está reservado para ello (ver **RFC 2606**[68]) y está recomendado para usar en documentaciones o como ejemplos.

Archivos

Los nombres de archivos que no contengan clases deberán ir en minúsculas y con guiones bajos, por ejemplo:

```
nombre_de_archivo_largo.php
```

Hacer `casts`

Para hacer `casts` usamos:

Tipo
: Descripción

(bool)
: Cast a boolean.

(int)
: Cast an integer.

(float)
: Cast a float.

(string)
: Cast a string.

(array)
: Cast an array.

(object)
: Cast an object.

Por favor utiliza `(int)$var` en lugar de `intval($var)` y `(float)$var` en lugar de `floatval($var)` cuando aplique.

[68] https://datatracker.ietf.org/doc/html/rfc2606.html

Constantes

Los nombres de constantes deberán ir en mayúsculas:

```
define('CONSTANTE', 1);
```

Si el nombre de una constante se compone de varias palabras deberán ir separadas por guiones bajos, por ejemplo:

```
define('NOMBRE_DE_CONSTANTE_LARGO', 2);
```

Cuidado al usar empty()/isset()

Aunque `empty()` es una función sencilla de utilizar, puede enmascarar errores y causar efectos accidentales cuando se usa con `'0'` y `0`. Cuando las variables o propiedades están ya definidas el uso de `empty()` no es recomendable. Al trabajar con variables es mejor utilizar la conversión a tipo booleano en lugar de `empty()`:

```
function manipulate($var)
{
    // No recomendado, $var está definido en el ámbito
    if (empty($var)) {
        // ...
    }

    // Utiliza la conversión a booleano
    if (!$var) {
        // ...
    }
    if ($var) {
        // ...
    }
}
```

Cuando trates con propiedades definidas deberías favorecer las comprobaciones sobre `null` en lugar de `empty()/isset()`:

```
class Thing
{
    private $property; // Definido

    public function readProperty()
    {
        // No recomendado al estar definida la propiedad en la clase
        if (!isset($this->property)) {
            // ...
        }
        // Recomendado
        if ($this->property === null) {

        }
    }
}
```

Cuando se trabaja con arrays, es mejor hacer `merge` de valores por defecto en vez de hacer comprobaciones con `empty()`. Haciendo `merge` de valores por defecto puedes asegurarte de que las claves necesarias están definidas:

```php
function doWork(array $array)
{
    // Hacer merge de valor por defecto para eliminar la necesidad
    // de comprobaciones empty
    $array += [
        'key' => null,
    ];

    // No recomendado, la clave ya está seteada
    if (isset($array['key'])) {
        // ...
    }

    // Recomendado
    if ($array['key'] !== null) {
        // ...
    }
}
```

Guía de compatibilidad hacia atrás

Asegurar que puedas actualizar tus aplicaciones fácilmente es importante para nosotros. Por ello sólo rompemos la compatibilidad en las liberaciones de versiones `major`. Puedes familiarizarte con el versionado semántico[69], el cual utilizamos en todos los proyectos de CakePHP. Pero resumiendo, el versionado semántico significa que sólo las liberaciones de versiones `major` (tales como 2.0, 3.0, 4.0) pueden romper la compatibilidad hacia atrás. Las liberaciones `minor` (tales como 2.1, 3.1, 3.2) pueden introducir nuevas funcionalidades, pero no pueden romper la compatibilidad. Los lanzamientos de correcciones de errores (tales como 3.0.1) no añaden nuevas funcionaliades, sólo correcciones de errores o mejoras de rendimiento.

Nota: CakePHP empezó a seguir el versionado semántico a partir de la 2.0.0. Estas reglas no se aplican en las versiones 1.x.

Para aclarar que cambios puedes esperar de cada nivel de lanzamiento tenemos más información detallada para desarrolladores que utilizan CakePHP y que trabajan en él que ayudan a aclarar que puede hacerse en liberaciones `minor`. Las liberaciones `major` pueden tener tantas rupturas de compatibilidad como sean necesarias.

Guías de migración

Para cada liberación `major` y `minor` el equipo de CakePHP facilitará una guía de migración. Estas guías explican las nuevas funcionaliades y cualquier ruptura de compatibilidad que haya en cada lanzamiento. Pueden encontrarse en la sección *Apéndices* del `cookbook`.

[69] https://semver.org/lang/es/

Usar CakePHP

Si estás desarrollando tu aplicación con CakePHP las siguientes pautas explican la estabilidad que puedes esperar.

Interfaces

Con excepción de las liberaciones `major`, las interfaces que provee CakePHP **no** tendrán ningún cambio en los métodos existentes. Podrán añadirse nuevos métodos, pero no habrá cambios en los ya existentes.

Clases

Las clases que proporciona CakePHP pueden estar construidas y tener sus métodos y propiedades públicos usados por el código de la aplicación y, a excepción de las liberaciones `major`, la compatibilidad hacia atrás está garantizada.

Nota: Algunas clases en CakePHP están marcadas con la etiqueta API doc `@internal`. Estas clases **no** son estables y no garantizan la compatibilidad hacia atrás.

En liberaciones `minor` pueden añadirse nuevos métodos a las clases y a los ya existentes nuevos argumentos. Cualquier argumento nuevo tendrá un valor por defecto, pero si sobreescribes métodos con una firma diferente puedes encontrar `fatal errors`. Los métodos con nuevos argumentos estarán documentados en las guías de migración..

La siguiente tabla esboza varios casos de uso y qué compatibilidad puedes esperar de CakePHP:

Si tu…	¿Compatible hacia atrás?
Tipificas contra la clase	Si
Creas una nueva instancia	Si
Extiendes la clase	Si
Accedes a una propiedad pública	Si
Llamas un método público	Si
Extiendes una clase y…	
Sobrescribes una propiedad pública	Si
Accedes a una propiedad protegida	No[1]
Sobreescribes una propiedad protegida	No[1]
Sobreescribes un método protegido	No[1]
Llamas a un método protegido	No[1]
Añades una propiedad pública	No
Añades un método público	No
Añades un argumento a un método sobreescrito	No[1]
Añades un valor por defecto a un argumento de método existente	Si

[1] Tu código *puede* romperse en lanzamientos `minor`. Comprueba la guía de migración para más detalles.

Trabajando en CakePHP

Si estás ayudando a que CakePHP sea aún mejor, por favor, ten en mente las siguientes pautas cuando añadas/cambies funcionalidades:

En una liberación `minor` puedes:

En una liberación `minor` puedes...	
Clases	
Eliminar una clase	No
Eliminar una interfaz	No
Eliminar un `trait`	No
Hacer final	No
Hacer abstract	No
Cambiar el nombre	Si[2]
Propiedades	
Añadir una propiedad pública	Si
Eliminar una propiedad pública	No
Añadir una propiedad protegida	Si
Eliminar una propiedad protegida	Si[3]
Métodos	
Añadir un método público	Si
Eliminar un método público	No
Añadir un método protegido	Si
Mover a la clase padre	Si
Eliminar un método protegido	Si[3]
Reducir visibilidad	No
Cambiar nombre del método	Si[2]
Añadir un argumento nuevo con valor por defecto	Si
Añadir un nuevo argumento obligatorio a un método existente	No
Eliminar un valor por defecto de un argumento existente	No

[2] Puedes cambiar el nombre de una clase/método siempre y cuando el antiguo nombre se mantenga disponible. Esto es evitado generalmente a menos que el cambio de nombre sea significativamente beneficioso.

[3] Evitarlo cuando sea posible. Cualquier borrado tendrá que ser documentado en la guía de migración.

CAPÍTULO 5

Instalación

CakePHP se instala rápida y fácilmente. ¡Los requisitos mínimos son un servidor web y una copia de CakePHP, y ya! Aunque este manual se enfoca principalmente en configurar Apache (ya que es el más utilizado), puedes configurar CakePHP para que corra con una variedad de servidores web como nginx, LightHTHPD o Microsoft IIS.

Requisitos

- Servidor HTTP. Por ejemplo: Apache. mod_rewrite es recomendado, pero no requerido.
- PHP 8.1 o mayor.
- extensión mbstring.
- extensión intl.

Técnicamente, una base de datos no es necesaria, pero imaginamos que la mayoría de aplicaciones utiliza alguna. CakePHP soporta una gran variedad de sistemas de bases de datos:

- MySQL (5.1.10 o mayor).
- PostgreSQL.
- Microsoft SQL Server (2008 o mayor).
- SQLite 3.

Nota: Todos los drivers nativos necesitan PDO. Debes asegurarte de tener las extensiones de PDO correctas.

Licencia

CakePHP está licenciado bajo la Licencia MIT[70]. Esto significa que eres libre para modificar, distribuir y republicar el código fuente con la condición de que las notas de copyright queden intactas. También eres libre para incorporar CakePHP en cualquier aplicación comercial o de código cerrado.

Instalando CakePHP

CakePHP utiliza Composer[71], una herramienta de manejo de dependencias para PHP 5.3+, como el método de instalación oficialmente soportado.

Primero, necesitas descargar e instalar Composer, si no lo has hecho ya. Si tienes instalado cURL, es tan fácil como correr esto en un terminal:

```
curl -s https://getcomposer.org/installer | php
```

O, puedes descargar `composer.phar` desde el sitio web de Composer[72].

Para sistemas Windows, puedes descargar el Instalador de Composer para Windows aquí[73]. Para más instrucciones acerca de esto, puedes leer el README del instalador de Windows aquí[74].

Ya que has descargado e instalado Composer puedes generar una aplicación CakePHP ejecutando:

```
php composer.phar create-project --prefer-dist cakephp/app:4.* [app_name]
```

O si tienes Composer definido globalmente:

```
composer create-project --prefer-dist cakephp/app:4.* [app_name]
```

Una vez que Composer termine de descargar el esqueleto y la librería core de CakePHP, deberías tener una aplicación funcional de CakePHP instalada vía Composer. Asegúrate de que los ficheros composer.json y composer.lock se mantengan junto con el resto de tu código fuente.

Ahora puedes visitar el destino donde instalaste la aplicación y ver los diferentes avisos tipo semáforo de los ajustes.

Mantente al día con los últimos cambios de CakePHP

Si quieres mantenerte al corriente de los últimos cambios en CakePHP puedes añadir las siguientes líneas al `composer.json` de tu aplicación:

```
"require": {
    "cakephp/cakephp": "dev-master"
}
```

Donde <branch> es el nombre del branch que quieres seguir. Cada vez que ejecutes `php composer.phar update` recibirás las últimas actualizaciones del branch seleccionado.

[70] https://www.opensource.org/licenses/mit-license.php
[71] https://getcomposer.org
[72] https://getcomposer.org/download/
[73] https://github.com/composer/windows-setup/releases/
[74] https://github.com/composer/windows-setup

Instalación usando DDEV

Otra manera rápida de instalar CakePHP es via DDEV[75]. DDEV es una herramienta de código abierto para lanzar ambientes de desarrollo web en local.

Si quieres configurar un nuevo proyecto, sólo necesitas ejecutar:

```
mkdir my-cakephp-app
cd my-cakephp-app
ddev config --project-type=cakephp --docroot=webroot
ddev composer create --prefer-dist cakephp/app:~5.0
ddev launch
```

Si tienes un proyecto existente:

```
git clone <your-cakephp-repo>
cd <your-cakephp-project>
ddev config --project-type=cakephp --docroot=webroot
ddev composer install
ddev launch
```

Por favor revisa la Documentación de DDEV[76] para más detalles de cómo instalar / actualizar DDEV.

Nota: IMPORTANTE: Ésto no es un script de despliegue. Su objetivo es ayudar desarrolladores a configurar ambientes de desarrollo rápidamente. En ningún caso su intención es que sea utilizado en ambientes de producción.

Permisos

CakePHP utiliza el directorio **tmp** para varias operaciones. Descripciones de Modelos, el caché de las vistas y la información de la sesión son algunos ejemplos de lo anterior. El directorio **logs** es utilizado para escribir ficheros de log por el motor de `FileLog` por defecto.

Asegúrate de que los directorios **logs**, **tmp** y todos sus subdirectorios tengan permisos de escritura por el usuario del Servidor Web. La instalación de CakePHP a través de Composer se encarga de este proceso haciendo que dichos directorios tengan los permisos abiertos globalmente con el fin de que puedas tener ajustado todo de manera más rápida. Obviamente, es recomendable que revises, y modifiques si es necesario, los permisos tras la instalación vía Composer para mayor seguridad.

Un problema común es que **logs**, **tmp** y sus subdirectorios deben poder ser modificados tanto por el usuario del Servidor Web como por el usuario de la línea de comandos. En un sistema UNIX, si los usuarios mencionados difieren, puedes ejecutar los siguientes comandos desde el directorio de tu aplicación para asegurarte de que todo esté configurado correctamente:

```
HTTPDUSER=`ps aux | grep -E '[a]pache|[h]ttpd|[_]www|[w]ww-data|[n]ginx' | grep -v root␣
↪| head -1 | cut -d\  -f1`
setfacl -R -m u:${HTTPDUSER}:rwx tmp
setfacl -R -d -m u:${HTTPDUSER}:rwx tmp
setfacl -R -m u:${HTTPDUSER}:rwx logs
setfacl -R -d -m u:${HTTPDUSER}:rwx logs
```

[75] https://ddev.com/
[76] https://ddev.readthedocs.io/

Configuración

Configurar una aplicación de CakePHP puede ser tan simple como colocarla en el directorio raíz de tu Servidor Web, o tan complejo y flexible como lo desees. Esta sección cubrirá los dos tipos principales de instalación de CakePHP: Desarrollo y Producción.

- Desarrollo: fácil de arrancar, las URL de la aplicación incluyen el nombre del directorio de la aplicación de CakePHP y es menos segura.
- Producción: Requiere tener la habilidad de configurar el directorio raíz del Servidor Web, cuenta con URL limpias y es bastante segura.

Desarrollo

Este es el método más rápido para configurar CakePHP. En este ejemplo utilizaremos la consola de CakePHP para ejecutar el servidor web nativo de PHP para hacer que tu aplicación esté disponible en **http://host:port**. Para ello ejecuta desde el directorio de la aplicación:

```
bin/cake server
```

Por defecto, sin ningún argumento, esto colocará tu aplicación en **http://localhost:8765/**.

Si tienes algún conflicto con **localhost** o el puerto **8765**, puedes indicarle a la consola de CakePHP que corra el servidor de manera más específica utilizando los siguientes argumentos:

```
bin/cake server -H 192.168.13.37 -p 5673
```

Esto colocará tu aplicación en **http://192.168.13.37:5673/**.

¡Eso es todo! Tu aplicación de CakePHP está corriendo perfectamente sin tener que haber configurado el servidor web manualmente.

Nota: Prueba `bin/cake server -H 0.0.0.0` si el servidor no es accesible desde otra máquina.

Advertencia: El servidor de desarrollo *nunca* debe ser utilizado en un ambiente de producción. Se supone que esto es un servidor básico de desarrollo y nada más.

Si prefieres usar un servidor web «real», Debes poder mover todos tus archivos de la instalación de CakePHP (incluyendo los archivos ocultos) dentro la carpeta raíz de tu servidor web. Debes entonces ser capaz de apuntar tu navegador al directorio donde moviste los archivos y ver tu aplicación en acción.

Producción

Una instalación de producción es una manera más flexible de montar una aplicación de CakePHP. Utilizando este método, podrás tener un dominio entero actuando como una sola aplicación de CakePHP. Este ejemplo te ayudará a instalar CakePHP donde quieras en tu sistema de ficheros y tenerlo disponible en `http://www.example.com`. Toma en cuenta que esta instalación requiere que tengas los derechos de cambiar el directorio raíz (`DocumentRoot`) del servidor web Apache.

Después de instalar tu aplicación utilizando cualquiera de los métodos mencionados en el directorio elegido - asumiremos que has escogido /cake_install - tu estructura de ficheros debe ser la siguiente:

```
/cake_install/
    bin/
    config/
    logs/
    plugins/
    src/
    tests/
    tmp/
    vendor/
    webroot/ (este directorio es ajutado como el DocumentRoot)
    .gitignore
    .htaccess
    .travis.yml
    composer.json
    index.php
    phpunit.xml.dist
    README.md
```

Si utilizas Apache debes configurar la directiva `DocumentRoot` del dominio a:

```
DocumentRoot /cake_install/webroot
```

Si tu configuración del Servidor Web es correcta debes tener tu aplicación disponible ahora en http://www.example.com.

A rodar!

Muy bien, ahora veamos a CakePHP en acción. Dependiendo de los ajustes que hayas utilizado, deberías dirigirte en tu navegador a http://example.com/ o http://localhost:8765/. En este punto, encontrarás la página principal de CakePHP y un mensaje que te dice el estado actual de tu conexión a la base de datos.

¡Felicidades! Estás listo para *Crear tu primera aplicación de CakePHP*.

URL Rewriting

Apache

Mientras que CakePHP está diseñado para trabajar con mod_rewrite recién sacado del horno, usualmente hemos notado que algunos usuarios tienen dificultades para lograr que todo funcione bien en sus sistemas.

Aquí hay algunas cosas que puedes tratar de conseguir para que funcione correctamente. La primera mirada debe ir a httpd.conf. (Asegura de que estás editando el httpd.conf del sistema en lugar del httpd.conf de un usuario o sitio específico)

Hay archivos que pueden variar entre diferentes distribuciones y versiones de Apache. Debes también mirar en https://cwiki.apache.org/confluence/display/httpd/DistrosDefaultLayout para obtener información.

1. Asegura de que un archivo .htaccess de sobreescritura esté permitido y que *AllowOverride* esté ajustado en *All* para el correcto *DocumentRoot*. Debes ver algo similar a:

   ```
   # Cada directorio al que Apache puede acceder puede ser configurado
   # con sus respectivos permitidos/denegados servicios y características
   # en ese directorios (y subdirectorios).
   #
   # Primero, configuramos el por defecto para ser muy restrictivo con sus
   # ajustes de características.
   <Directory />
       Options FollowSymLinks
       AllowOverride All
   #    Order deny,allow
   #    Deny from all
   </Directory>
   ```

2. Asegura que tú estás cargando mod_rewrite correctamente. Debes ver algo similar a esto:

   ```
   LoadModule rewrite_module libexec/apache2/mod_rewrite.so
   ```

   En muchos sistemas esto estará comentado por defecto, así que solo debes remover el símbolo # al comienzo de la línea.

   Después de hacer los cambios, reinicia Apache para asegurarte que los ajustes estén activados.

   Verifica que tus archivos .htaccess está actualmente en directorio correcto. Algunos sistemas operativos tratan los archivos que empiezan con "." como ocultos y, por lo tanto, no podrás copiarlos.

3. Asegúrate que tu copia de CakePHP provenga desde la sección descargas del sitio o de nuestro repositorio de Git, y han sido desempacados correctamente, revisando los archivos .htaccess.

   El directorio app de CakePHP (Será copiado en la raíz de tu aplicación por bake):

   ```
   <IfModule mod_rewrite.c>
      RewriteEngine on
      RewriteRule    ^$    webroot/    [L]
      RewriteRule    (.*) webroot/$1    [L]
   </IfModule>
   ```

   El directorio webroot de CakePHP (Será copiado a la raíz de tu aplicación web por bake):

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [L]
</IfModule>
```

Si tu sitio aún tiene problemas con mod_rewrite, querrás probar modificar los ajustes para el Servidor Virtual. En Ubuntu, edita el archivo **/etc/apache2/sites-available/default** (la ubicación depende de la distribución). En este archivo, debe estar `AllowOverride None` cambiado a``AllowOverride All``, así tendrás:

```
<Directory />
    Options FollowSymLinks
    AllowOverride All
</Directory>
<Directory /var/www>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride All
    Order Allow,Deny
    Allow from all
</Directory>
```

En macOS, otra solución es usar la herramienta virtualhostx[77] para crear servidores virtuales y apuntarlos a tu carpeta.

Para muchos servicios de alojamiento (GoDaddy, 1and1), tu servidor web estará actualmente sirviendo desde un directorio de usuario que actualmente usa mod_rewrite. Si tú estás instalando CakePHP en la carpeta de usuario (http://example.com/~username/cakephp/), o alguna otra estructura de URL que ya utilice mod_rewrite, necesitarás agregar una declaración a los archivos .htaccess que CakePHP usa (.htaccess, webroot/.htaccess).

Esto puede ser agregado a la misma sección con la directiva RewriteEngine, entonces por ejemplo, tu .htaccess en el webroot debería verse algo así:

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/app
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [L]
</IfModule>
```

Los detalles de estos cambios dependerán de tu configuración, y puede incluir algunas líneas adicionales que no están relacionadas con CakePHP. Por favor dirígete a la documentación en línea de Apache para más información.

4. (Opcional) Para mejorar la configuración de producción, debes prevenir archivos adicionales inválidos que sean tomados por CakePHP. Modificando tu .htaccess del webroot a algo cómo esto:

```
<IfModule mod_rewrite.c>
    RewriteEngine On
    RewriteBase /path/to/app/
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_URI} !^/(webroot/)?(img|css|js)/(.*)$
    RewriteRule ^ index.php [L]
</IfModule>
```

[77] https://clickontyler.com/virtualhostx/

Lo anterior simplemente previene que archivos adicionales incorrectos sean enviados a index.php en su lugar muestre la página 404 de tu servidor web.

Adicionalmente, puedes crear una página 404 que concuerde, o usar la página 404 incluida en CakePHP agregando una directiva `ErrorDocument`:

```
ErrorDocument 404 /404-not-found
```

Nginx

Nginx no hace uso de un archivo .htaccess como Apache, por esto es necesario crear la reescritura de URL en la configuracion de *sites-available*. Esto usualmente se encuentra en `/etc/nginx/sites-available/your_virtual_host_conf_file`. Dependiendo de la configuración, necesitarás modificar esto, pero por lo menos, necesitas PHP corriendo como una instancia FastCGI:

```
server {
    listen   80;
    server_name www.example.com;
    rewrite ^(.*) http://example.com$1 permanent;
}

server {
    listen   80;
    server_name example.com;

    # root directive should be global
    root   /var/www/example.com/public/webroot/;
    index  index.php;

    access_log /var/www/example.com/log/access.log;
    error_log /var/www/example.com/log/error.log;

    location / {
        try_files $uri $uri/ /index.php?$args;
    }

    location ~ \.php$ {
        try_files $uri =404;
        include /etc/nginx/fastcgi_params;
        fastcgi_pass    127.0.0.1:9000;
        fastcgi_index   index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }
}
```

En algunos servidores (Como Ubuntu 14.04) la configuración anterior no funcionará recién instalado, y de todas formas la documentación de nginx recomienda una forma diferente de abordar esto (https://nginx.org/en/docs/http/converting_rewrite_rules.html). Puedes intentar lo siguiente (Notarás que esto es un bloque de servidor {}, en vez de dos, pese a que si quieres que example.com resuelva a tu aplicación CakePHP en adición a www.example.com consulta el enlace de nginx anterior):

```
server {
    listen   80;
```

(continué en la próxima página)

(proviene de la página anterior)

```
    server_name www.example.com;
    rewrite 301 http://www.example.com$request_uri permanent;

    # root directive should be global
    root   /var/www/example.com/public/webroot/;
    index  index.php;

    access_log /var/www/example.com/log/access.log;
    error_log /var/www/example.com/log/error.log;

    location / {
        try_files $uri /index.php?$args;
    }

    location ~ \.php$ {
        try_files $uri =404;
        include /etc/nginx/fastcgi_params;
        fastcgi_pass    127.0.0.1:9000;
        fastcgi_index   index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    }
}
```

IIS7 (Windows)

IIS7 no soporta de manera nativa los archivos .htaccess. Mientras haya *add-ons* que puedan agregar soporte a estos archivos, puedes también importar las reglas htaccess en IIS para usar las redirecciones nativas de CakePHP. Para hacer esto, sigue los siguientes pasos:

1. Usa el Intalador de plataforma Web de Microsoft[78] para instalar el Modulo de Redirreción 2.0[79] de URLs o descarga directamente (32-bit[80] / 64-bit[81]).
2. Crear un nuevo archivo llamado web.config en tu directorio de raíz de CakePHP.
3. Usando Notepad o cualquier editor de XML, copia el siguiente código en tu nuevo archivo web.config:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <system.webServer>
        <rewrite>
            <rules>
                <rule name="Exclude direct access to webroot/*"
                  stopProcessing="true">
                    <match url="^webroot/(.*)$" ignoreCase="false" />
                    <action type="None" />
                </rule>
                <rule name="Rewrite routed access to assets(img, css, files, js, favicon)
↪"
```

(continué en la próxima página)

[78] https://www.microsoft.com/web/downloads/platform.aspx
[79] https://www.iis.net/downloads/microsoft/url-rewrite
[80] https://download.microsoft.com/download/D/8/1/D81E5DD6-1ABB-46B0-9B4B-21894E18B77F/rewrite_x86_en-US.msi
[81] https://download.microsoft.com/download/1/2/8/128E2E22-C1B9-44A4-BE2A-5859ED1D4592/rewrite_amd64_en-US.msi

(proviene de la página anterior)

```
                    stopProcessing="true">
                    <match url="^(img|css|files|js|favicon.ico)(.*)$" />
                    <action type="Rewrite" url="webroot/{R:1}{R:2}"
                      appendQueryString="false" />
                </rule>
                <rule name="Rewrite requested file/folder to index.php"
                    stopProcessing="true">
                    <match url="^(.*)$" ignoreCase="false" />
                    <action type="Rewrite" url="index.php"
                      appendQueryString="true" />
                </rule>
            </rules>
        </rewrite>
    </system.webServer>
</configuration>
```

Una vez el archivo web.config es creado con las reglas de redirección amigables de IIS, los enlaces, CSS, JavaScript y redirecciones de CakePHP deberían funcionar correctamente.

No puedo usar Redireccionamientos de URL

Si no quieres o no puedes obtener mod_rewirte (o algún otro módulo compatible) en el servidor a correr, necesitarás usar el decorador de URL incorporado en CakePHP. En **config/app.php**, descomentar la línea para que se vea así:

```
'App' => [
    // ...
    // 'baseUrl' => env('SCRIPT_NAME'),
]
```

También remover estos archivos .htaccess:

```
/.htaccess
webroot/.htaccess
```

Esto hará tus URL verse así www.example.com/index.php/controllername/actionname/param antes que www.example.com/controllername/actionname/param.

CAPÍTULO 6

Configuración

Aunque las convenciones eliminan la necesidad de configurar todas las partes de CakePHP, todavía necesitarás configurar algunas cosas, como las credenciales de tu base de datos.

Además, existen opciones de configuración opcionales que te permiten cambiar los valores y las implementaciones predeterminadas por otros personalizados para tu aplicación.

Configurando tu Aplicación

La configuración generalmente se almacena en archivos PHP o INI, y se carga durante el inicio de la aplicación. CakePHP viene con un archivo de configuración por defecto, pero si es necesario, puedes agregar archivos de configuración adicionales y cargarlos en el código de inicio de tu aplicación. La clase `Cake\Core\Configure` se utiliza para la configuración global, y clases como `Cache` proporcionan métodos como `setConfig()` para hacer que la configuración sea simple y transparente.

El esqueleto de la aplicación incluye un archivo **config/app.php** que debería contener configuraciones que no varían en los diversos entornos en los que se despliega tu aplicación. El archivo **config/app_local.php** debería contener datos de configuración que varían entre los entornos y deben ser gestionados por herramientas de gestión de configuración o tus herramientas de implementación. Ambos archivos hacen referencia a variables de entorno a través de la función `env()` que permite establecer valores de configuración a través del entorno del servidor.

Cargar Archivos de Configuración Adicionales

Si tu aplicación tiene muchas opciones de configuración, puede ser útil dividir la configuración en varios archivos. Después de crear cada uno de los archivos en tu directorio **config/**, puedes cargarlos en **bootstrap.php**:

```
use Cake\Core\Configure;
use Cake\Core\Configure\Engine\PhpConfig;

Configure::setConfig('default', new PhpConfig());
Configure::load('app', 'default', false);
Configure::load('other_config', 'default');
```

Variables de Entorno

Muchos proveedores de servicios en la nube modernos, como Heroku, te permiten definir variables de entorno para datos de configuración. Puedes configurar tu aplicación CakePHP a través de variables de entorno en el estilo de aplicación *12factor <https://12factor.net/>*. Las variables de entorno permiten que tu aplicación sea fácil de gestionar cuando se implementa en varios entornos.

Como puedes ver en tu archivo **app.php**, la función `env()` se utiliza para leer la configuración del entorno y construir la configuración de la aplicación. CakePHP utiliza cadenas de conexión *DSN* para bases de datos, registros, transportes de correo electrónico y configuraciones de caché, lo que te permite variar fácilmente estas bibliotecas en cada entorno.

Para el desarrollo local, CakePHP utiliza dotenv[82] para recargar automáticamente las variables de entorno locales. Utiliza Composer para requerir esta biblioteca y luego hay un bloque de código en `bootstrap.php` que debe descomentarse para aprovecharla.

Encontrarás un archivo `config/.env.example` en tu aplicación. Al copiar este archivo en `config/.env` y personalizar los valores, puedes configurar tu aplicación.

Debes evitar incluir el archivo `config/.env` en tu repositorio y, en su lugar, utilizar `config/.env.example` como una plantilla con valores predeterminados para que todos en tu equipo sepan qué variables de entorno se están utilizando y qué debe ir en cada una.

Una vez que se hayan establecido tus variables de entorno, puedes usar `env()` para leer datos del entorno:

```
$debug = env('APP_DEBUG', false);
```

El segundo valor pasado a la función env es el valor predeterminado. Este valor se utilizará si no existe una variable de entorno para la clave dada.

Configuración General

A continuación se muestra una descripción de las variables y cómo afectan a tu aplicación CakePHP.

- **debug** Cambia la salida de depuración de CakePHP. `false` = Modo de producción. No se muestran mensajes de error o advertencias. `true` = Se muestran errores y advertencias.
- **App.namespace** El espacio de nombres para encontrar las clases de la aplicación.

> **Nota:** Al cambiar el espacio de nombres en tu configuración, también deberás actualizar tu archivo **composer.json** para usar este espacio de nombres. Además, crea un nuevo autoloader ejecutando `php composer.phar`

[82] https://github.com/josegonzalez/php-dotenv

`dumpautoload`.

- **App.baseUrl** Descomenta esta definición si **no** planeas usar mod_rewrite de Apache con CakePHP. No olvides eliminar tus archivos .htaccess también.
- **App.base** El directorio base en el que reside la aplicación. Si es `false`, se detectará automáticamente. Si no es `false`, asegúrate de que tu cadena comience con un / y **NO** termine con un /. Por ejemplo, */basedir* es un valor válido para App.base.
- **App.encoding** Define qué codificación utiliza tu aplicación. Esta codificación se utiliza para definir la codificación en las vistas y codificar entidades. Debería coincidir con los valores de codificación especificados para tu base de datos.
- **App.webroot** El directorio webroot.
- **App.wwwRoot** La ruta de archivo al directorio webroot.
- **App.fullBaseUrl** El nombre de dominio completamente cualificado (incluyendo el protocolo) hasta la raíz de tu aplicación. Se utiliza al generar URLs absolutas. Por defecto, este valor se genera utilizando la variable `$_SERVER` del entorno. Sin embargo, debes definirlo manualmente para optimizar el rendimiento o si te preocupa que las personas manipulen el encabezado «Host». En un contexto CLI (desde la línea de comandos), el *fullBaseUrl* no se puede leer de $_SERVER, ya que no hay un servidor web involucrado. Debes especificarlo tú mismo si necesitas generar URLs desde una terminal (por ejemplo, al enviar correos electrónicos).
- **App.imageBaseUrl** Ruta web al directorio público de imágenes dentro del webroot. Si estás utilizando un *CDN*, debes configurar este valor con la ubicación del CDN.
- **App.cssBaseUrl** Ruta web al directorio público de CSS dentro del webroot. Si estás utilizando un *CDN*, debes configurar este valor con la ubicación del CDN.
- **App.jsBaseUrl** Ruta web al directorio público de JavaScript dentro del webroot. Si estás utilizando un *CDN*, debes configurar este valor con la ubicación del CDN.
- **App.paths** Configura rutas para recursos que no son de clase. Admite las subclaves `plugins`, `templates`, `locales`, que permiten la definición de rutas para los archivos de plugins, plantillas de vista y archivos de traducción, respectivamente.
- **App.uploadedFilesAsObjects** Define si los archivos cargados se representan como objetos (`true`) o como arrays (`false`). Esta opción está habilitada de forma predeterminada. Consulta la sección *File Uploads* en el capítulo de Objetos de Request & Response para obtener más información.
- **Security.salt** Una cadena aleatoria utilizada en el cifrado. Esta cadena también se utiliza como la sal de HMAC al hacer cifrado simétrico.
- **Asset.timestamp** Añade una marca de tiempo, que es la última vez que se modificó el archivo en particular, al final de las URLs de los archivos de activos (CSS, JavaScript, Imagen) cuando se utilizan los ayudantes adecuados. Valores válidos:
  - (bool) `false` - No hace nada (predeterminado)
  - (bool) `true` - Añade la marca de tiempo cuando el modo de depuración es `true`
  - (string) "force" - Siempre añade la marca de tiempo.
- **Asset.cacheTime** Establece el tiempo de caché del archivo de activo. Esto determina el encabezado `Cache-Control`, `max-age` y el tiempo de `Expire` del encabezado de HTTP para los activos. Esto puede tomar cualquier valor que la función strtotime[83] tu versión de PHP pueda tomar. El valor predeterminado es +1 `día`.

[83] https://php.net/manual/es/function.strtotime.php

Usar un CDN

Para utilizar un CDN para cargar tus activos estáticos, cambia las variables `App.imageBaseUrl`, `App.cssBaseUrl`, `App.jsBaseUrl` para que apunten a la URI del CDN, por ejemplo: `https://micdn.ejemplo.com/` (nota la barra diagonal al final `/`).

Todas las imágenes, scripts y estilos cargados a través de HtmlHelper agregarán la ruta absoluta del CDN, coincidiendo con la misma ruta relativa utilizada en la aplicación. Ten en cuenta que hay un caso de uso específico cuando se utilizan activos basados en plugins: los plugins no utilizarán el prefijo del plugin cuando se utiliza una URI absoluta `...BaseUrl`, por ejemplo, por defecto:

- `$this->Helper->assetUrl('TestPlugin.logo.png')` resuelve a `test_plugin/logo.png`

Si configuras `App.imageBaseUrl` como `https://micdn.ejemplo.com/`:

- `$this->Helper->assetUrl('TestPlugin.logo.png')` se resuelve a `https://micdn.ejemplo.com/logo.png`.

Configuración de la Base de Datos

Consulta la *Configuración de la Base de Datos* para obtener información sobre cómo configurar las conexiones a tu base de datos.

Configuración de Caché

Consulta la *Configuración de Caché* para obtener información sobre cómo configurar la caché en CakePHP.

Configuración de Manejo de Errores y Excepciones

Consulta la *Configuración de Errores y Excepciones* para obtener información sobre cómo configurar los manejadores de errores y excepciones.

Configuración de Registro (Logs)

Consulta la *Configuración de Registro* para obtener información sobre cómo configurar el registro (logs) en CakePHP.

Configuración de Correo Electrónico

Consulta la *Configuración de Correo Electrónico* para obtener información sobre cómo configurar preajustes de correo electrónico en CakePHP.

Configuración de Sesión

Consulta la *Configuración de Sesión* para obtener información sobre cómo configurar el manejo de sesiones en CakePHP.

Configuración de Enrutamiento

Consulta la *Configuración de Rutas* para obtener más información sobre cómo configurar el enrutamiento y crear rutas para tu aplicación.

Rutas de Clases Adicionales

Las rutas de clases adicionales se configuran a través de los cargadores automáticos que utiliza tu aplicación. Cuando utilizas *composer* para generar tu cargador automático, puedes hacer lo siguiente para proporcionar rutas alternativas para los controladores en tu aplicación:

```
"autoload": {
    "psr-4": {
        "App\\Controller\\": "/ruta/a/directorio/con/carpetas/de/controladores/",
        "App\\": "src/"
    }
}
```

El ejemplo anterior establecería rutas para los espacios de nombres *App* y *AppController*. Se buscará la primera clave y, si esa ruta no contiene la clase/archivo, se buscará la segunda clave. También puedes asignar un solo espacio de nombres a múltiples directorios de la siguiente manera:

```
"autoload": {
    "psr-4": {
        "App\\": ["src/", "/ruta/a/directorio/"]
    }
}
```

Rutas de Plugins, Plantillas de Vista y Localizaciones

Dado que los plugins, las plantillas de vista y las localizaciones no son clases, no pueden tener un cargador automático configurado. CakePHP proporciona tres variables de configuración para establecer rutas adicionales para estos recursos. En tu **config/app.php**, puedes configurar estas variables:

```
return [
    // Otras configuraciones
    'App' => [
        'paths' => [
            'plugins' => [
                ROOT . DS . 'plugins' . DS,
                '/ruta/a/otros/plugins/',
            ],
            'templates' => [
                ROOT . DS . 'templates' . DS,
                ROOT . DS . 'templates2' . DS,
            ],
            'locales' => [
                ROOT . DS . 'resources' . DS . 'locales' . DS,
            ],
        ],
```

(continué en la próxima página)

(proviene de la página anterior)

```
    ],
]
```

Las rutas deben terminar con un separador de directorio, o no funcionarán correctamente.

Configuración de Inflexión

Consulta la documentación de *Configurando las Inflecciones* para obtener más información.

Clase Configure

class Cake\Core**Configure**

La clase Configure de CakePHP se puede utilizar para almacenar y recuperar valores específicos de la aplicación o en tiempo de ejecución. Sin embargo, debes tener cuidado, ya que esta clase te permite almacenar cualquier cosa y usarla en cualquier parte de tu código, lo que puede ser una tentación para romper el patrón MVC para el que CakePHP fue diseñado. El principal objetivo de la clase Configure es mantener variables centralizadas que puedan compartirse entre varios objetos. Recuerda intentar seguir el principio «convención sobre configuración» para no terminar rompiendo la estructura MVC que CakePHP proporciona.

Escritura de Datos de Configuración

static Cake\Core\Configure::**write**(*$clave*, *$valor*)

Utiliza `write()` para almacenar datos en la configuración de la aplicación:

```
Configure::write('Company.name', 'Pizza, Inc.');
Configure::write('Company.slogan', 'Pizza for your body and soul');
```

Nota: La *notación de punto* utilizada en el parámetro `$clave` se puede utilizar para organizar tus configuraciones en grupos lógicos.

El ejemplo anterior también se podría escribir en una sola llamada:

```
Configure::write('Company', [
    'name' => 'Pizza, Inc.',
    'slogan' => 'Pizza for your body and soul'
]);
```

Puedes utilizar `Configure::write('debug', $boolean)` para alternar entre los modos de depuración y producción sobre la marcha.

Nota: Cualquier cambio en la configuración realizado mediante `Configure::write()` se mantiene en memoria y no persistirá entre solicitudes.

Lectura de Datos de Configuración

static Cake\Core\Configure::**read**(*$clave = null*, *$predeterminado = null*)

Se utiliza para leer datos de configuración de la aplicación. Si se proporciona una clave, se devolverán los datos. Usando nuestros ejemplos anteriores de `write()`, podemos leer esos datos de la siguiente manera:

```
# Devuelve 'Pizza, Inc.'
Configure::read('Company.name');

# Devuelve 'Pizza for your body and soul'
Configure::read('Company.slogan');

Configure::read('Company');
# Devuelve:
['name' => 'Pizza, Inc.', 'slogan' => 'Pizza for your body and soul'];

# Devuelve 'fallback' ya que Company.nope no está definido.
Configure::read('Company.nope', 'fallback');
```

Si se deja el parámetro `$clave` como nulo, se devolverán todos los valores en Configure.

static Cake\Core\Configure::**readOrFail**(*$clave*)

Lee datos de configuración igual que `Cake\Core\Configure::read()`, pero espera encontrar un par clave/valor. Si el par solicitado no existe, se lanzará una `RuntimeException`.

> Configure::readOrFail("Company.name"); # Devuelve: "Pizza, Inc." Configure::readOrFail("Company.geolocation"); # Lanzará una excepción
>
> Configure::readOrFail("Company");
>
> # Devuelve: ["name" => "Pizza, Inc.", "slogan" => "Pizza for your body and soul"];

Comprobación para ver si los Datos de Configuración están Definidos

static Cake\Core\Configure::**check**(*$clave*)

Se utiliza para comprobar si una clave/ruta existe y tiene un valor distinto de nulo:

```
$existe = Configure::check('Company.name');
```

Eliminación de Datos de Configuración

static Cake\Core\Configure::**delete**(*$clave*)

Se utiliza para eliminar información de la configuración de la aplicación:

```
Configure::delete('Company.name');
```

Lectura y Eliminación de Datos de Configuración

static Cake\Core\Configure::**consume**(*$clave*)

Lee y elimina una clave de Configure. Esto es útil cuando deseas combinar la lectura y eliminación de valores en una sola operación.

static Cake\Core\Configure::**consumeOrFail**(*$clave*)

Consume datos de configuración de la misma manera que `Cake\Core\Configure::consume()`, pero espera encontrar un par clave/valor. Si el par solicitado no existe, se lanzará una `RuntimeException`.

> Configure::consumeOrFail("Company.name"); # Devuelve: "Pizza, Inc." Configure::consumeOrFail("Company.geolocation"); # Lanzará una excepción
>
> Configure::consumeOrFail("Company");
>
> # Devuelve: ["name" => "Pizza, Inc.", "slogan" => "Pizza for your body and soul"]

Lectura y Escritura de Archivos de Configuración

static Cake\Core\Configure::**setConfig**(*$nombre*, *$motor*)

CakePHP viene con dos motores de archivos de configuración integrados. `Cake\Core\Configure\Engine\PhpConfig` es capaz de leer archivos de configuración PHP, en el mismo formato que Configure ha leído históricamente. `Cake\Core\Configure\Engine\IniConfig` es capaz de leer archivos de configuración ini. Consulta la documentación de PHP[84] para obtener más información sobre los detalles de los archivos ini. Para utilizar un motor de configuración central, debes adjuntarlo a Configure utilizando :php:`Configure::config()`:

```
use Cake\Core\Configure\Engine\PhpConfig;

# Leer archivos de configuración desde config
Configure::config('default', new PhpConfig());

# Leer archivos de configuración desde otra ruta.
Configure::config('default', new PhpConfig('/ruta/a/tus/archivos/de/configuración/'));
```

Puedes tener varios motores adjuntos a Configure, cada uno leyendo diferentes tipos o fuentes de archivos de configuración. Puedes interactuar con los motores adjuntos usando los métodos definidos en Configure. Para verificar qué alias de motor están adjuntos, puedes usar `Configure::configured()`:

```
# Obtén el array de alias para los motores adjuntos.
Configure::configured();

# Comprueba si un motor específico está adjunto
Configure::configured('default');
```

static Cake\Core\Configure::**drop**(*$nombre*)

También puedes eliminar motores adjuntos. `Configure::drop('default')` eliminaría el alias del motor predeterminado. Cualquier intento futuro de cargar archivos de configuración con ese motor fallaría:

```
Configure::drop('default');
```

[84] https://php.net/parse_ini_file

Carga de Archivos de Configuración

static Cake\Core\Configure::**load**(*$clave*, *$config = 'default'*, *$merge = true*)

Una vez que hayas adjuntado un motor de configuración a Configure, puedes cargar archivos de configuración:

```
# Cargar my_file.php usando el objeto de motor 'default'.
Configure::load('my_file', 'default');
```

Los archivos de configuración cargados fusionan sus datos con la configuración en tiempo de ejecución existente en Configure. Esto te permite sobrescribir y agregar nuevos valores a la configuración en tiempo de ejecución existente. Al establecer `$merge` en `true`, los valores nunca sobrescribirán la configuración existente.

Advertencia: Al fusionar archivos de configuración con *$merge = true*, la notación de puntos en las claves no se expande:

```
# config1.php
'Clave1' => [
    'Clave2' => [
        'Clave3' => ['ClaveAnidada1' => 'Valor'],
    ],
],

# config2.php
'Clave1.Clave2' => [
    'Clave3' => ['ClaveAnidada2' => 'Valor2'],
]

Configure::load('config1', 'default');
Configure::load('config2', 'default', true);

# Ahora Clave1.Clave2.Clave3 tiene el valor ['ClaveAnidada2' => 'Valor2']
# en lugar de ['ClaveAnidada1' => 'Valor', 'ClaveAnidada2' => 'Valor2']
```

Creación o Modificación de Archivos de Configuración

static Cake\Core\Configure::**dump**(*$clave*, *$config = 'default'*, *$claves = []*)

Vuelca todos o algunos de los datos en Configure en un archivo o sistema de almacenamiento compatible con un motor de configuración. El formato de serialización lo decide el motor de configuración adjunto como $config. Por ejemplo, si el motor "default" es una `Cake\Core\Configure\Engine\PhpConfig`, el archivo generado será un archivo de configuración PHP que se puede cargar mediante el `Cake\Core\Configure\Engine\PhpConfig`

Dado que el motor "default" es una instancia de PhpConfig. Guarda todos los datos en Configure en el archivo *mi_configuracion.php*:

```
Configure::dump('mi_configuracion', 'default');
```

Guarda solo la configuración de manejo de errores:

```
Configure::dump('error', 'default', ['Error', 'Exception']);
```

`Configure::dump()` se puede utilizar para modificar o sobrescribir archivos de configuración que se pueden leer con `Configure::load()`

Almacenamiento de Configuración en Tiempo de Ejecución

static Cake\Core\Configure::**store**(*$nombre*, *$configuracionCache = 'default'*, *$datos = null*)

También puedes almacenar valores de configuración en tiempo de ejecución para usarlos en solicitudes futuras. Dado que configure solo recuerda valores para la solicitud actual, deberás almacenar cualquier información de configuración modificada si deseas usarla en solicitudes posteriores:

```
# Almacena la configuración actual en la clave 'usuario_1234' en la caché 'default'.
Configure::store('usuario_1234', 'default');
```

Los datos de configuración almacenados persisten en la configuración de caché con el nombre especificado. Consulta la documentación sobre *Caching* para obtener más información sobre el almacenamiento en caché.

Restauración de Configuración en Tiempo de Ejecución

static Cake\Core\Configure::**restore**(*$nombre*, *$configuracionCache = 'default'*)

Una vez que hayas almacenado la configuración en tiempo de ejecución, probablemente necesitarás restaurarla para poder acceder a ella nuevamente. `Configure::restore()` hace precisamente eso:

```
# Restaura la configuración en tiempo de ejecución desde la caché.
Configure::restore('usuario_1234', 'default');
```

Al restaurar información de configuración, es importante restaurarla con la misma clave y configuración de caché que se usó para almacenarla. La información restaurada se fusiona con la configuración en tiempo de ejecución existente.

Motores de Configuración

CakePHP proporciona la capacidad de cargar archivos de configuración desde varias fuentes diferentes y cuenta con un sistema plugable para crear tus propios motores de configuración[85]. Los motores de configuración integrados son:

- JsonConfig[86]
- IniConfig[87]
- PhpConfig[88]

Por defecto, tu aplicación utilizará `PhpConfig`.

[85] https://api.cakephp.org/5.x/interface-Cake.Core.Configure.ConfigEngineInterface.html
[86] https://api.cakephp.org/5.x/class-Cake.Core.Configure.Engine.JsonConfig.html
[87] https://api.cakephp.org/5.x/class-Cake.Core.Configure.Engine.IniConfig.html
[88] https://api.cakephp.org/5.x/class-Cake.Core.Configure.Engine.PhpConfig.html

Desactivación de Tablas Genéricas

Aunque utilizar clases de tabla genéricas, también llamadas auto-tablas, al crear rápidamente nuevas aplicaciones y hornear modelos es útil, las clases de tabla genéricas pueden dificultar la depuración en algunos escenarios.

Puedes verificar si se emitió alguna consulta desde una clase de tabla genérica a través del panel SQL de DebugKit. Si aún tienes problemas para diagnosticar un problema que podría ser causado por las auto-tablas, puedes lanzar una excepción cuando CakePHP utiliza implícitamente una `Cake\ORM\Table` genérica en lugar de tu clase concreta de la siguiente manera:

```
# En tu bootstrap.php
use Cake\Event\EventManager;
use Cake\Http\Exception\InternalErrorException;

$seEjecutaCakeBakeShell = (PHP_SAPI === 'cli' && isset($argv[1]) && $argv[1] === 'bake');
if (!$seEjecutaCakeBakeShell) {
    EventManager::instance()->on('Model.initialize', function($event) {
        $subject = $event->getSubject();
        if (get_class($subject) === 'Cake\ORM\Table') {
            $mensaje = sprintf(
                'Clase de tabla faltante o alias incorrecto al registrar la clase de␣
↪tabla para la tabla de base de datos %s.',
                $subject->getTable());
            throw new InternalErrorException($mensaje);
        }
    });
}
```

CAPÍTULO 7

Routing

class Cake\Routing**RouterBuilder**

El enrutamiento te provee de herramientas que permiten mapear URLs a acciones de un controlador. Al definir rutas, puedes separar cómo está implementada tu aplicación y cómo están estructuradas sus URLs.

El enrutamiento en CakePHP también abarca la idea de enrutamiento inverso, donde una matriz de parámetros se puede transformar en una cadena URL. Al utilizar el enrutamiento inverso, puedes refactorizar la estructura de tus URLs sin necesidad de actualizar todo tu código.

Vistazo rápido

Esta sección te enseñará los usos más comunes del enrutamiento en CakePHP con ejemplos. Normalmente, deseas mostrar algo como una página de destino, por lo que tendrás que añadir esto a tu archivo **config/routes.php**:

```
/** @var \Cake\Routing\RouteBuilder $routes */
$routes->connect('/', ['controller' => 'Articles', 'action' => 'index']);
```

Esto ejecutará el método index que se encuentra en `ArticlesController` cuando se visite la página principal de tu sitio. A veces necesitas rutas dinámicas que aceptarán múltiples parámetos, por ejemplo cuando necesites una ruta para ver el contenido de un artículo:

```
$routes->connect('/articles/*', ['controller' => 'Articles', 'action' => 'view']);
```

La ruta anterior aceptará cualquier URL que se parezca a `/article/15` e invocará el método `view(15)` de `ArticlesController`. Esto no prevendrá que las personas intenten acceder a URLs como `/articles/foobar`. Si quieres, puedes restringir algunos parámetros que se ajusten a una expresión regular:

```
// Utilizando una interfaz fluida
$routes->connect(
```

(continué en la próxima página)

(proviene de la página anterior)

```
    '/articles/{id}',
    ['controller' => 'Articles', 'action' => 'view'],
)
->setPatterns(['id' => '\d+'])
->setPass(['id']);

// Utilizando una matriz de opciones
$routes->connect(
    '/articles/{id}',
    ['controller' => 'Articles', 'action' => 'view'],
    ['id' => '\d+', 'pass' => ['id']]
);
```

En el ejemplo anterior se cambió el comparador asterisco por un nuevo marcador de posición `{id}`. Utilizar marcadores de posición nos permite valiadr partes de la URL, en este caso utilizamos la expresión regular `\d+` por lo que sólo los dígitos son comparados. Finalmente, le indicamos al enrutador que trate el marcador de posición `id` como un argumento de función para el método `view()` especificando la opción `pass`. Hablaremos más sobre el uso de esta opción más adelante.

El enrutador de CakePHP también puede revertir rutas de coincidencia. Esto quiere decir que desde una matriz que contiene parámetros de coincidencia es capaz de generar una cadena de URL:

```
use Cake\Routing\Router;

echo Router::url(['controller' => 'Articles', 'action' => 'view', 'id' => 15]);
// Obtendrás como salida
/articles/15
```

Las rutas también pueden etiquetarse con un nombre único, esto te permite referenciarlas rápidamente cuando creas enlaces en lugar de especificar cada uno de los parámetros de la ruta:

```
// En routes.php
$routes->connect(
    '/upgrade',
    ['controller' => 'Subscriptions', 'action' => 'create'],
    ['_name' => 'upgrade']
);

use Cake\Routing\Router;

echo Router::url(['_name' => 'upgrade']);
// Obtendrás como salida
/upgrade
```

Para ayudar a mantener tu código de enrutamiento *DRY*, el Enrutador tiene el concepto de "ámbitos". Un ámbito define un segmento de ruta común y, opcionalmente, rutas predeterminadas. Cualquier ruta conectada dentro de un ámbito heredará la ruta y valores por defecto de su ámbito:

```
$routes->scope('/blog', ['plugin' => 'Blog'], function (RouteBuilder $routes) {
    $routes->connect('/', ['controller' => 'Articles']);
});
```

La rua anterior coincidiría con `/blog/` y la enviaría a `Blog\Controller\ArticlesController::index()`.

El esqueleto de la aplicación viene con algunas rutas de inicio. Una vez has añadido tus tuyas propias, puedes eliminar las rutas por defecto si no las necesitas.

Conectando Rutas

Para mantener tu código *DRY* debes utilizar "ámbitos de ruta". Los ámbitos de ruta no sólo te facilitan mantener tu código DRY, sino que ayudan al Enrutador a optimizar sus operaciones. Este método se aplica por defecto al ámbito `/`. Para crear un ámbito y conectar algunas rutas utilizarás el método `scope()`:

```
// En config/routes.php
use Cake\Routing\RouteBuilder;
use Cake\Routing\Route\DashedRoute;

$routes->scope('/', function (RouteBuilder $routes) {
    // Conecta las rutas alternativas genéricas.
    $routes->fallbacks(DashedRoute::class);
});
```

El método `connect()` acepta hasta tres parámetros: la plantilla de URL para la que deseas conincidencias, los valores predeterminados para los elementos de tu ruta. Las opciones frecuentemente incluyen reglas de expresión regular que para ayudar al enrutador a coincidir con elementos de la URL.

El formato básico para la definición de una ruta es:

```
$routes->connect(
    '/url/template',
    ['targetKey' => 'targetValue'],
    ['option' => 'matchingRegex']
);
```

El primer parámetro se utiliza para indicarle al enrutador qué tipo de URL se está intentando controlar. La URL es una cadena normal delimitada por barras diagonales, pero también puede contener un comodín (*) o *Elementos de ruta*. El uso de un comodín le indica al enrutador que puede aceptar cualquier argumento adicional que se le proporcione. Las rutas sin * sólo coincidirán con el patrón de plantilla exacto proporcionado.

Una vez que hayas especificado una URL, utiliza los dos últimos parámetros de `connect()` para indicar a CakePHP qué debe hacer con la solicitud cuando haya coincidencia. El segundo parámetro define la ruta "objetivo". Esto se puede definir como una matriz o como una cadena de destino. Algunos ejemplos de ruta objetivo son:

```
// Matriz de destino a un controlador de aplicación
$routes->connect(
    '/users/view/*',
    ['controller' => 'Users', 'action' => 'view']
);
$routes->connect('/users/view/*', 'Users::view');

// Matriz de destino a un controlador de plugin con prefijo
$routes->connect(
    '/admin/cms/articles',
    ['prefix' => 'Admin', 'plugin' => 'Cms', 'controller' => 'Articles', 'action' =>
↪'index']
);
$routes->connect('/admin/cms/articles', 'Cms.Admin/Articles::index');
```

La primera ruta que conectamos coincide con las URL que comienzan con `/users/view` y asigna esas solucitudes al `UsersController->view()`. El `/*` indica al enrutador para pasar cualquier segmento adicional como argumentos del método. Por ejemplo, `/users/view/123` se asignaría a `UsersController->view(123)`.

El ejemplo anterior también ilustra los destinos de cadena. Los destinos de cadena proporcionan una forma compacta de definir el destino de una ruta. Los destinos de cadena tienen la siguiente sintaxis:

```
[Plugin].[Prefix]/[Controller]::[action]
```

Algunos ejemplos de destino de cadena son:

```
// Controlador de aplicación
'Articles::view'

// Controlador de aplicación con prefijo
Admin/Articles::view

// Controlador de plugin
Cms.Articles::edit

// Controlador de plugin con prefijo
Vendor/Cms.Management/Admin/Articles::view
```

Anteriormente, usamos el asterisco final (`/*`) para capturar segmentos de ruta adicionales, también está el doble asterisco final (`/**`). Utilizando el doble asterisco final, capturará el resto de una URL como un único argumento. Esto es útil cuando se desea utilizar un argumento que incluye `/`:

```
$routes->connect(
    '/pages/**',
    ['controller' => 'Pages', 'action' => 'show']
);
```

La URL entrante `/pages/the-example-/-and-proof` daría como resultado el paso de un único argumento `the-example-/-and-proof`.

El segundo parámetro de `connect()` puede definir cualquier parámetro para componer los parámetros de ruta po predeterminado:

```
$routes->connect(
    '/government',
    ['controller' => 'Pages', 'action' => 'display', 5]
);
```

Este ejemplo utiliza el segundo parámetro de `connect()` para definir los parámetros predeterminados. Si creas una aplicación que presenta productos para diferentes categorías de clientes, podrías considerar crear una ruta. Esto permite enlazar `/government` en lugar de `/pages/display/5`.

Un uso común del enrutamiento es renombrar los controladores y sus acciones. En lugar de acceder a nuestro controlador de usuarios como `/users/some-action/5`, nos gustaría acceder a él a través de `/cooks/some-action/5`. La siguiente ruta se encarga de eso:

```
$routes->connect(
    '/cooks/{action}/*', ['controller' => 'Users']
);
```

Esto indica al enrutador que cualquier URL que empieze por `/cooks/` deberá ser enviada al `UsersController`. La acción invocada dependerá del valor del parámetro `{action}`. Utilizando *Elementos de ruta*, puedes crear rutas

variables que aceptan entradas del usuario o variables. La ruta anterior también utiliza el asteristo final. El asterisco final indica que esta ruta debe aceptar cualquier argumento posicional adicional dado. Estos argumentos estarán disponibles en la matriz *Argumentos Pasados*.

Al generar URL también se utilizan rutas. Utilizando `['controller' => 'Users', 'action' => 'some-action', 5]` como una URL, generará `/cooks/some-action/5` si la ruta anterior es la primera coincidencia encontrada.

Las ruts que hemos conectado hasta ahora coincidirán con cualquier tipo de petición HTTP. Si estás contruyendo un API REST, a menudo querrás asignar acciones HTTP a diferentes métodos de controlador. El `RouteBuilder` proporciona métodos auxiliares que facilitan la definición de rutas para tipos de peticiones HTTP específicas más simples:

```
// Crea una ruta que sólo responde a peticiones GET.
$routes->get(
    '/cooks/{id}',
    ['controller' => 'Users', 'action' => 'view'],
    'users:view'
);

// Crea una ruta que sólo responde a peticiones PUT
$routes->put(
    '/cooks/{id}',
    ['controller' => 'Users', 'action' => 'update'],
    'users:update'
);
```

Las rutas anteriores asignan la misma URL a diferentes acciones del controlador según el tipo de petición HTTP utilizada. Las solicitudes GET irán a la acción "view", mientras que las solicitudes PUT irán a la acción UPDATE. Existen métodos auxiliares HTTP para:

- GET
- POST
- PUT
- PATCH
- DELETE
- OPTIONS
- HEAD

Todos estos métodos devuelven una instancia de ruta, lo que permite aprovechar la *fluent setterts* para configurar aún más la ruta.

Elementos de ruta

Puedes especificar tus propios elementos de ruta y al hacerlo podrás definir los lugares de la URL donde los parámetros para las acciones del controlador deben estar. Cuando se realiza una solicitud, los valores para estos elementos de ruta se encuentran en `$this->request->getParam()` en el controlador. Cuando defines un elemento de ruta personalizado, opcionalmente puedes especificar una expresión regular; esto le dice a CakePHP como saber si la URL está formada correctamente o no. Si eliges no proporcionar una expresión regular, cualquier caracter que no sea `/` será tratado como parte del parámetro:

```
$routes->connect(
    '/{controller}/{id}',
    ['action' => 'view']
)->setPatterns(['id' => '[0-9]+']);

$routes->connect(
    '/{controller}/{id}',
    ['action' => 'view'],
    ['id' => '[0-9]+']
);
```

El ejemplo anterior ilustra cómo crear una forma rápida de ver modelos desde cualquier controlador creando una URL que se parezca a `/controllername/{id}`. La URL proporcionada a `connect()` especifica dos elementos de ruta: `{controller}` y `{id}`. El elemento `{controller}` es un elemento de ruta predeterminado de CakePHP, por lo que el enrutador conoce cómo identificar y emparejar los nombres de controladores en la URL. El elemento `{id}` es un elemento de ruta personalizado y debe aclararse especificando una expresión regular en el tercer parámetro de `connect()`.

CakePHP no produce automáticamente URL en minúsculas y con guiones cuando utiliza el parámetro `{controller}`. Si necesitas hacer esto, el ejemplo anterior podría ser reescrito así:

```
use Cake\Routing\Route\DashedRoute;

// Crea un constructor con una clase de ruta diferente
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->setRouteClass(DashedRoute::class);
    $routes->connect('/{controller}/{id}', ['action' => 'view'])
        ->setPatterns(['id' => '[0-9]+']);

    $routes->connect(
        '/{controller}/{id}',
        ['action' => 'view'],
        ['id' => '[0-9]+']
    );
});
```

La clase `DashedRoute` se asegurará de que los parámetros `{controller}` y `{plugin}` están correctamente en minúsculas y con guiones.

Nota: Los patrones utilizados por los elementos de ruta no deben contener ningún grupo de captura. Si lo hacen, el enrutador no funcionará correctamente.

Una vez que se ha definido esta ruta, al solicitar `/apples/5` se llamará al método `view()` de ApplesController. Dento del método `view()`, necesitarás acceder al ID pasado en `$this->request->getParam('id')`.

Si tienes un único controlador en tu aplicación y no quieres que el nombre del controlador aparezca en la URL, puedes asignar todas las URL a acciones en tu controlador. Por ejemplo, para asignar todas las URL a acciones del contolador `home`, para tener una URL como `/demo` en lugar de `/home/demo`, puedes hacer lo siguiente:

```
$routes->connect('/{action}', ['controller' => 'Home']);
```

Si quieres proporcionar una URL que no distinga entre mayúsculas y minúsculas, puedes utilizar modificadores en línea de expresiones regulares:

```
$routes->connect(
    '/{userShortcut}',
    ['controller' => 'Teachers', 'action' => 'profile', 1],
)->setPatterns(['userShortcut' => '(?i:principal)']);
```

Un ejemplo más y serás un profesional del enrutamiento:

```
$routes->connect(
    '/{controller}/{year}/{month}/{day}',
    ['action' => 'index']
)->setPatterns([
    'year' => '[12][0-9]{3}',
    'month' => '0[1-9]|1[012]',
    'day' => '0[1-9]|[12][0-9]|3[01]'
]);
```

Esto es bastante complicado, pero muestra cuán poderosas pueden ser las rutas. La URL proporcionada tiene cuatro elementos de ruta. El primero nos resulta familiar: es un elemento de ruta por defecto que incica a CakePHP que espere un nombre de controlador.

A continuación, especificamos algunos valores predeterminados. Independientemente del controlador, queremos que se llame a la acción `index()`.

Finalmente, especificamos algunas expresiones regulares que coincidirán con año, mes y día en forma numérica. Ten en cuenta que los paréntesis (captura de grupos) no se admiten en expresiones regulares. Aún podemos especificar alternativas, como se indicó anteriormente, pero no agrupadas entre paréntesis.

Una vez definida, esta ruta coincidirá con `/articles/2007/02/01`, `/articles/2004/11/16`, entregando las solicitudes a la acción `index()` de sus respectivos controladores, con los parámetros de fecha en `$this->request->getParams()`.

Elementos de Ruta Reservados

Hay varios elementos de ruta que tienen un significado especial en CakePHP, y no deben usarse a menos que desee un significado especial

- `controller` Se utiliza para nombrar el controlador de una ruta.
- `action` Se utiliza para nombrar la acción del controlador para una ruta.
- `plugin` Se utiliza para nombrar el complemento en el que se encuentra un controlador.
- `prefix` Usado para *Enrutamiento de Prefijo*
- `_ext` Usado para *File extentions routing*.
- `_base` Se establece a `false` para eliminar la ruta base de la URL generada. Si su aplicación no está en el directorio raíz, esto puede utilizarse para generar URL que son “cake relative”.
- `_scheme` Configurado para crear enlaces en diferentes esquemas como *webcal* o *ftp*. El valor predeterminado es el esquema actual.
- `_host` Establece el host que se utilizará para el enlace. El valor predeterminado es el host actual.
- `_port` Establece el puerto si necesitamos crear enlaces en puertos no estándar.
- `_full` Si es `true` el valor de `App.fullBaseUrl` mencionado en *Configuración General* se atepondrá a la URL generada.
- # Permite configurar fragmentos de hash de URL.

- `_https` Establecerlo en `true` para convertir la URL generada a https o``false`` para forzar http. Antes de 4.5.0 utilizar `_ssl`.
- `_method` Define el tipo de petición/método a utilizar. Útil cuando trabajamos con *Enrutamiento RESTful*.
- `_name` Nombre de la ruta. Si has configurado rutas con nombre, puedes utilizar esta clave para especificarlo.

Configurando Opciones de Ruta

Hay varias opciones de ruta que se pueden configurar en cada ruta. Después de conectar una ruta, puedes utilizar sus métodos de creación fluidos para configurar aún más la ruta. Estos métodos reemplazan muchas de las claves en el parámetro `$options` de `connect()`:

```
$routes->connect(
    '/{lang}/articles/{slug}',
    ['controller' => 'Articles', 'action' => 'view'],
)
// Permite peticiones GET y POST.
->setMethods(['GET', 'POST'])

// Sólo coincide con el subdominio del blog.
->setHost('blog.example.com')

// Establecer los elementos de ruta que deben convertirse en argumentos pasados
->setPass(['slug'])

// Establecer los patrones de coincidencia para los elementos de ruta
->setPatterns([
    'slug' => '[a-z0-9-_]+',
    'lang' => 'en|fr|es',
])

// También permite archivos con extensión JSON
->setExtensions(['json'])

// Establecer lang para que sea un parámetro persistente
->setPersist(['lang']);
```

Pasar Parámetros a una Acción

Cuando conectamos rutas utilizando :*Elementos de ruta* es posible que desees que los elementos enrutados se pasen como argumentos. La opción `pass` indica qué elementos de ruta también deben estaar disponibles como argumentos pasados a las funciones del controlador:

```
// src/Controller/BlogsController.php
public function view($articleId = null, $slug = null)
{
    // Algún código aquí...
}

// routes.php
$routes->scope('/', function (RouteBuilder $routes) {
```

(continué en la próxima página)

(proviene de la página anterior)

```
    $routes->connect(
        '/blog/{id}-{slug}', // For example, /blog/3-CakePHP_Rocks
        ['controller' => 'Blogs', 'action' => 'view']
    )
    // Definir los elementos de ruta en la plantilla de ruta
    // para anteponerlos como argumentos de la función. El orden
    // es importante ya que esto pasará los elementos `$id` y `$slug`
    // como primer y segundo parámetro. Cualquier otro parámetro
    // adicional pasado en tu ruta se agregará después de los
    // argumentos de setPass().
    ->setPass(['id', 'slug'])
    // Definir un patrón con el que `id` debe coincidir.
    ->setPatterns([
        'id' => '[0-9]+',
    ]);
});
```

Ahora, gracias a las capacidades de enturamiento inverso, puedes pasar la matriz de URL como se muestra a continuación y CakePHP sabrá cómo formar la URL como se define en las rutas:

```
// view.php
// Esto devolverá un enlace a /blog/3-CakePHP_Rocks
echo $this->Html->link('CakePHP Rocks', [
    'controller' => 'Blog',
    'action' => 'view',
    'id' => 3,
    'slug' => 'CakePHP_Rocks'
]);

// También podemos utilizar índices numéricos como parámetros.
echo $this->Html->link('CakePHP Rocks', [
    'controller' => 'Blog',
    'action' => 'view',
    3,
    'CakePHP_Rocks'
]);
```

Uso del Enrutamiento de Ruta

Hablamos de objetivos de cadena anteriormente. Lo mismo también funciona para la generación de URL usando `Router::pathUrl()`:

```
echo Router::pathUrl('Articles::index');
// salida: /articles

echo Router::pathUrl('MyBackend.Admin/Articles::view', [3]);
// salida: /admin/my-backend/articles/view/3
```

Truco: La compatibilidad del IDE para el autocompletado del enrutamiento de ruta se puede habilitar con CakePHP

IdeHelper Plugin[89].

Usar Rutas con Nombre

A veces encontrarás que escribir todos los parámetros de la URL para una ruta es demasiado detallado, o le gustaría aprovechar las mejoras de rendimiento que tienen las rutas con nombre. Al conectar rutas, puedes especificar una opción `_name`, esta opción se puede utilizar en rutas inversas para identificar la ruta que deseas utilizar:

```
// Conectar una ruta con nombre.
$routes->connect(
    '/login',
    ['controller' => 'Users', 'action' => 'login'],
    ['_name' => 'login']
);

// Nombrar una ruta específica según el tipo de petición
$routes->post(
    '/logout',
    ['controller' => 'Users', 'action' => 'logout'],
    'logout'
);

// Generar una URL utilizando una ruta con nombre.
$url = Router::url(['_name' => 'logout']);

// Generar una URL utilizando una ruta con nombre,
// con algunos argumentos de cadena en la consulta.
$url = Router::url(['_name' => 'login', 'username' => 'jimmy']);
```

Si tu plantilla de ruta contienen elementos de ruta como `{controller}` deberás proporcionarlos como parte de las opciones de `Router::url()`.

Nota: Los nombres de las rutas deben ser únicos en toda la aplicación. El mismo `_name` no se puede utilizar dos veces, incluso si los nombres aparecen dentro de un alcance de enrutamiento diferente.

Al crear rutas con nombre, probablemente querrás ceñirte a algunas convenciones para los nombres de las rutas. CakePHP facilita la creación de nombres de rutas al permitir definir prefijos de nombres en cada ámbito:

```
$routes->scope('/api', ['_namePrefix' => 'api:'], function (RouteBuilder $routes) {
    // El nombre de esta ruta será `api:ping`
    $routes->get('/ping', ['controller' => 'Pings'], 'ping');
});
// Generar una URL para la ruta de ping
Router::url(['_name' => 'api:ping']);

// Utilizar namePrefix con plugin()
$routes->plugin('Contacts', ['_namePrefix' => 'contacts:'], function (RouteBuilder
↪$routes) {
    // Conectar rutas.
```

(continué en la próxima página)

[89] https://github.com/dereuromark/cakephp-ide-helper

(proviene de la página anterior)

```
});

// O con prefix()
$routes->prefix('Admin', ['_namePrefix' => 'admin:'], function (RouteBuilder $routes) {
    // Conectar rutas.
});
```

También puedes utilizar la opción `_namePrefix` dentro de ámbitos anidados y funciona como se esperaba:

```
$routes->plugin('Contacts', ['_namePrefix' => 'contacts:'], function (RouteBuilder
↪$routes) {
    $routes->scope('/api', ['_namePrefix' => 'api:'], function (RouteBuilder $routes) {
        // Este nombre de ruta será `contacts:api:ping`
        $routes->get('/ping', ['controller' => 'Pings'], 'ping');
    });
});

// Generar una URL para la ruta de ping
Router::url(['_name' => 'contacts:api:ping']);
```

Las rutas conectadas en ámbitos con nombre sólo se les agregarán nombres si la ruta también tiene nombre. A las rutas sin nombre no se les aplicará el `_namePrefix`. Routes connected in named scopes will only have names added if the route is also named. Nameless routes will not have the `_namePrefix` applied to them.

Enrutamiento de Prefijo

static Cake\Routing\RouterBuilder::**prefix**(*$name*, *$callback*)

Muchas aplicaciones requieren una sección de adminitración donde los usuarios con privilegios puedan realizar cambios. Esto se hace a menudo a través de una URL especial como `/admin/users/edit/5`. En CakePHP, el enrutamiento de prefijo puede ser habilitado utilizando el método de ámbito `prefix`:

```
use Cake\Routing\Route\DashedRoute;

$routes->prefix('Admin', function (RouteBuilder $routes) {
    // Todas las rutas aquí tendrán el prefijo `/admin`, y
    // tendrán el elemento de ruta `'prefix' => 'Admin'` agregado que
    // será necesario para generar URL para estas rutas
    $routes->fallbacks(DashedRoute::class);
});
```

Los prefijos se asignan a subespacios de nombres en el espacio de nombres `Controller` en tu aplicación. Al tener prefijos como controladores separados, puedes crear controladores más pequeños y simples. El comportamiento que es común a los controladores con y sin prefijo se puede encapsular mediante herencia, *Componentes*, o traits. Utilizando nuestro ejemplo de usuarios, acceder a la URL `/admin/users/edit/5` llamaría al médito `edit()` de nuestro **src/Controller/Admin/UsersController.php** pasando 5 como primer parámetro. El archivo de vista utilizado sería **templates/Admin/Users/edit.php**

Puedes asignar la URL /admin a tu acción `index()` del controlador pages utilizando la siguiente ruta:

```
$routes->prefix('Admin', function (RouteBuilder $routes) {
    // Dado que te encuentras en el ámbito de admin,
```

(continué en la próxima página)

(proviene de la página anterior)

```
    // no necesitas incluir el prefijo /admin ni el
    // elemento de ruta Admin.
    $routes->connect('/', ['controller' => 'Pages', 'action' => 'index']);
});
```

Al crear rutas de prefijo, puedes establecer parámetros de ruta adicionales utilizando el argumento `$options`:

```
$routes->prefix('Admin', ['param' => 'value'], function (RouteBuilder $routes) {
    // Las rutas conectadas aquí tienen el prefijo '/admin' y
    // tienen configurada la clave de enrutamiento 'param'.
    $routes->connect('/{controller}');
});
```

Los prefijos con varias palabras se convierten de forma predeterminada utilizando la inflexión dasherize, es decir, `MyPrefix` se asignará a `my-prefix` en la URL. Asegúrate de establecer una ruta para dichos prefijos si deseas utilizar un formato diferente como, por ejemplo, subrayado:

```
$routes->prefix('MyPrefix', ['path' => '/my_prefix'], function (RouteBuilder $routes) {
    // Las rutas conectadas aquí tiene el prefijo '/my_prefix'
    $routes->connect('/{controller}');
});
```

También puedes definir prefijos dentro del alcance de un plugin:

```
$routes->plugin('DebugKit', function (RouteBuilder $routes) {
    $routes->prefix('Admin', function (RouteBuilder $routes) {
        $routes->connect('/{controller}');
    });
});
```

Lo anterior crearía una plantilla de ruta como `/debug-kit/admin/{controller}`. La ruta conectada tendría establecidos los elementos de ruta `plugin` y `prefix`.

Al definir prefijos, puedes anidar varios prefijos si es necesario:

```
$routes->prefix('Manager', function (RouteBuilder $routes) {
    $routes->prefix('Admin', function (RouteBuilder $routes) {
        $routes->connect('/{controller}/{action}');
    });
});
```

Lo anterior crearía una plantilla de ruta como `/manager/admin/{controller}/{action}`. La ruta conectada tendría establecido el elemento de ruta `prefix` a `Manager/Admin`.

El prefijo actual estará disponible desde los métodos del controlador a través de `$this->request->getParam('prefix')`

Cuando usamos rutas de prefijo es importante configurar la opción `prefix` y utilizar el mismo formato CamelCased que se utiliza in el método `prefix()`. A continuación se explica cómo crear este enlace utilizando el helper HTML:

```
// Ve a una ruta de prefijo
echo $this->Html->link(
    'Manage articles',
    ['prefix' => 'Manager/Admin', 'controller' => 'Articles', 'action' => 'add']
```

(continué en la próxima página)

(proviene de la página anterior)

```
);

// Deja un prefijo
echo $this->Html->link(
    'View Post',
    ['prefix' => false, 'controller' => 'Articles', 'action' => 'view', 5]
);
```

Crear Enlaces a Rutas de Prefijo

Puedes crear enlaces que apunten a un prefijo añadiendo la clave del prefijo a la matriz de URL:

```
echo $this->Html->link(
    'New admin todo',
    ['prefix' => 'Admin', 'controller' => 'TodoItems', 'action' => 'create']
);
```

Al utilizar anidamiento, es necesario encadenarlos entre sí:

```
echo $this->Html->link(
    'New todo',
    ['prefix' => 'Admin/MyPrefix', 'controller' => 'TodoItems', 'action' => 'create']
);
```

Esto se vincularía a un controlador con el espacio de nombre `App\\Controller\\Admin\\MyPrefix` y la ruta de archivo `src/Controller/Admin/MyPrefix/TodoItemsController.php`.

Nota: Aquí el prefijo siempre es CamelCased, incluso si el resultado del enrutamiento es discontinuo. La propia ruta hará la inflexión si es necesario.

Enrutamiento de Plugin

static Cake\Routing\RouterBuilder::**plugin**(*$name*, *$options = []*, *$callback*)

Las rutas para *Plugins* deben crearse utilizando el método `plugin()`. Este método crea un nuevo ámbito de enrutamiento para las rutas del plugin:

```
$routes->plugin('DebugKit', function (RouteBuilder $routes) {
    // Las rutas conectadas aquí tienen el prefijo '/debug-kit' y
    // el elemento de ruta plugin configurado a 'DebugKit'.
    $routes->connect('/{controller}');
});
```

Cuando creamos ámbitos de plugin, puedes personalizar el elemento de ruta utilizado con la opción `path`:

```
$routes->plugin('DebugKit', ['path' => '/debugger'], function (RouteBuilder $routes) {
    // Las rutas conectadas aquí tiene el prefijo '/debugger' y
    // el elemento de ruta plugin configurado a 'DebugKit'.
    $routes->connect('/{controller}');
});
```

Al utilizar ámbitos, puedes anidar ámbitos de plugin dentro de ámbitos de prefijos:

```
$routes->prefix('Admin', function (RouteBuilder $routes) {
    $routes->plugin('DebugKit', function (RouteBuilder $routes) {
        $routes->connect('/{controller}');
    });
});
```

Lo anteior crearía una ruta similar a `/admin/debug-kit/{controller}`. Tendría configurados los elementos de ruta `prefix` y `plugin`. En la sección *Rutas del Plugin* hay más información sobre la creación de rutas de plugin.

Crear Enlaces a Rutas de Plugin

Puedes crear enlaces que apunten a un plugin añadiendo la clave plugin a tu matrix de URL:

```
echo $this->Html->link(
    'New todo',
    ['plugin' => 'Todo', 'controller' => 'TodoItems', 'action' => 'create']
);
```

Por el contrario, si la solicitud activa es una solicitud de plugin y deseas crear un enlace que no tenga plugin puedes hacer lo siguiente:

```
echo $this->Html->link(
    'New todo',
    ['plugin' => null, 'controller' => 'Users', 'action' => 'profile']
);
```

Estableciendo `'plugin' => null` le indicas al Enrutador que quieres crear un enlace que no forme parte de un plugin.

Enrutamiento SEO-Friendly

Algunos desarrolladores prefieren utilizar guiones en las URL, ya que se percibe que dan un mejor posicionamiento en los motores de búsqueda. La clase `DashedRoute` se puede utilizar en tu aplicación con la capacidad de enrutar plugin, controlador y acciones camelizadas a una URL con guiones.

Por ejemplo, si tenemos un plugin `ToDo`, con un controlador `TodoItems`, y una acción `showItems()`, se podría acceder en `/to-do/todo-items/show-items` con la siguiente conexión de enrutador:

```
use Cake\Routing\Route\DashedRoute;

$routes->plugin('ToDo', ['path' => 'to-do'], function (RouteBuilder $routes) {
    $routes->fallbacks(DashedRoute::class);
});
```

Coincidencia de Métodos HTTP Específicos

Las rutas pueden coincidir con métodos HTTP específicos utilizando los métodos del helper HTTP:

```
$routes->scope('/', function (RouteBuilder $routes) {
    // Esta ruta sólo coincide con peticiones POST.
    $routes->post(
        '/reviews/start',
        ['controller' => 'Reviews', 'action' => 'start']
    );

    // Coincide con múltiples tipos de peticiones
    $routes->connect(
        '/reviews/start',
        [
            'controller' => 'Reviews',
            'action' => 'start',
        ]
    )->setMethods(['POST', 'PUT']);
});
```

Puedes hacer coincidir múltiples métodos HTTP utilizando una matriz. Dada que el parámetro `_method` es una clave de enturamiento, participa tanto en el análisis como en la generación de URL. Para generar URLs para rutas específicas de un método necesitarás incluir la clave `_method` al generar la URL:

```
$url = Router::url([
    'controller' => 'Reviews',
    'action' => 'start',
    '_method' => 'POST',
]);
```

Coincidencia con Nombres de Dominio Específicos

Las rutas pueden utilizar la opción `_host` para coincidir sólo con dominios específicos. Puedes utilizar el comodín `*.` para coincidir con cualquier subdominio:

```
$routes->scope('/', function (RouteBuilder $routes) {
    // Esta ruta sólo coincide en http://images.example.com
    $routes->connect(
        '/images/default-logo.png',
        ['controller' => 'Images', 'action' => 'default']
    )->setHost('images.example.com');

    // Esta ruta sólo coincide en http://*.example.com
    $routes->connect(
        '/images/old-logo.png',
        ['controller' => 'Images', 'action' => 'oldLogo']
    )->setHost('*.example.com');
});
```

La opción `_host` también se utiliza en la generación de URL. Si tu opción `_host` especifica un dominio exacto, ese dominio se incluirá en la URL generada. Sin embargo, si utilizas un comodín, tendrás que indicar el parámetro `_host` al generar la URL:

```
// Si tienes esta ruta
$routes->connect(
    '/images/old-logo.png',
    ['controller' => 'Images', 'action' => 'oldLogo']
)->setHost('images.example.com');

// Necesitas esto para generar la URL
echo Router::url([
    'controller' => 'Images',
    'action' => 'oldLogo',
    '_host' => 'images.example.com',
]);
```

Enrutamiento de Extensiones de Archivo

static Cake\Routing\RouterBuilder::**extensions**(*string|array|null $extensions*, *$merge = true*)

Para manejar diferentes extensiones de archivo en tus URL, puedes definir las extensiones utilizando el método `Cake\Routing\RouteBuilder::setExtensions()`:

```
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->setExtensions(['json', 'xml']);
});
```

Esto habilitará ls extensiones nombradas para todas las rutas que se estén conectando en ese ámbito **después** de la llamada a `setExtensions()`, incluidas aquellas que se estén conectando en ámbitos anidados.

Nota: Configurar las extensiones debe ser lo primero que hagas en un ámbito, ya que las extensiones sólo se aplicarán a rutas conectadas **después** de configurar las extensiones.

También ten en cuenta que los ámbitos reabiertos **no** heredarán las extensiones definidas en ámbitos abiertos anteriormente.

Al utilizar extensiones, le indicas al enrutador que elimine cualquier extensión de archivo coincidente en la URL y luego analice lo que queda. Si deseas crear una URL como /page/title-of-page.html, crearías su ruta usando:

```
$routes->scope('/page', function (RouteBuilder $routes) {
    $routes->setExtensions(['json', 'xml', 'html']);
    $routes->connect(
        '/{title}',
        ['controller' => 'Pages', 'action' => 'view']
    )->setPass(['title']);
});
```

Luego, para crear enlaces que correspondan con las rutas, simplemente usa:

```
$this->Html->link(
    'Link title',
    ['controller' => 'Pages', 'action' => 'view', 'title' => 'super-article', '_ext' =>
↪'html']
);
```

Middleware de Ámbito de Ruta

Si bien el Middleware se puede aplicar a toda tu aplicación, aplicar middleware a ámbitos de enrutamiento específicos ofrece más flexibilidad, ya que puedes aplicar middleware sólo donde sea necesario, lo que permite que tu middleware no se preocupe por cómo y dónde se aplica.

Nota: El middleware con ámbito aplicado se ejecutará mediante RoutingMiddleware, normalmente al final de la cola de middleware de tu aplicación.

Antes de que se pueda aplicar middleware a tu acplicación, es necesario registrarlo en la colección de rutas:

```
// en config/routes.php
use Cake\Http\Middleware\CsrfProtectionMiddleware;
use Cake\Http\Middleware\EncryptedCookieMiddleware;

$routes->registerMiddleware('csrf', new CsrfProtectionMiddleware());
$routes->registerMiddleware('cookies', new EncryptedCookieMiddleware());
```

Una vez registrado, el middleware con ámbito se podrá aplicar a ámbitos específicos:

```
$routes->scope('/cms', function (RouteBuilder $routes) {
    // Activa CSRF y cookies middleware
    $routes->applyMiddleware('csrf', 'cookies');
    $routes->get('/articles/{action}/*', ['controller' => 'Articles']);
});
```

En situaciones en las que tienes ámbitos anidados, los ámbitos internos heredarán el middleware aplicado en el ámbito contenedor:

```
$routes->scope('/api', function (RouteBuilder $routes) {
    $routes->applyMiddleware('ratelimit', 'auth.api');
    $routes->scope('/v1', function (RouteBuilder $routes) {
        $routes->applyMiddleware('v1compat');
        // Definir las rutas aquí.
    });
});
```

En el ejemplo anterior, las rutas definidas en `/v1` tendrán aplicado el middleware “ratelimit”, “auth.api” y “v1compat”. Si vuelves a abrir un ámbito, el middleware aplicado a las rutas en cada ámbito quedará aislado:

```
$routes->scope('/blog', function (RouteBuilder $routes) {
    $routes->applyMiddleware('auth');
    // Conecta las acciones authenticadas para el blog aquí.
});
$routes->scope('/blog', function (RouteBuilder $routes) {
    // Conecta las acciones públicas para el blog aquí.
});
```

En el ejemplo anterior, los dos usos del alcance `/blog` no comparten middleware. Sin embargo, ambos ámbitos heredarán el middleware definido en los ámbitos que los engloban.

Agrupación de Middleware

Para ayudar a mantener tu código de ruta DRY (Do not Repeat Yourself) el middleware se puede combinar en grupos. Una vez combinados, los grupos pueden aplicarse como middleware:

```
$routes->registerMiddleware('cookie', new EncryptedCookieMiddleware());
$routes->registerMiddleware('auth', new AuthenticationMiddleware());
$routes->registerMiddleware('csrf', new CsrfProtectionMiddleware());
$routes->middlewareGroup('web', ['cookie', 'auth', 'csrf']);

// Aplicar el grupo
$routes->applyMiddleware('web');
```

Enrutamiento RESTful

El enrutador ayuda a generar rutas RESTful para tus controladores. Las rutas RESTful son útiles cuando estás creando API endpoints para tus aplicaciones. Si quisiéramos permitir el acceso REST a un controlador de recetas, haríamos algo como esto:

```
// En config/routes.php...

$routes->scope('/', function (RouteBuilder $routes) {
    $routes->setExtensions(['json']);
    $routes->resources('Recipes');
});
```

La primera línea configura una serie de rutas predeterminadas para el acceso REST donde el método especifica el formato de resultado deseado, por ejemplo, xml, json y rss. Estas rutas son sensible al método de solicitud HTTP.

HTTP format	URL.format	Acción del controlador invocada
GET	/recipes.format	RecipesController::index()
GET	/recipes/123.format	RecipesController::view(123)
POST	/recipes.format	RecipesController::add()
PUT	/recipes/123.format	RecipesController::edit(123)
PATCH	/recipes/123.format	RecipesController::edit(123)
DELETE	/recipes/123.format	RecipesController::delete(123)

Nota: El patrón predeterminado para los ID de recursos sólo coincide con números enterors o UUID. Si tus ID son diferentes, tendrás que proporcionar un patrón de expresión regular a través de la opción `id`, por ejemplo `$builder->resources('Recipes', ['id' => '.*'])`.

El método HTTP utilizado se detecta desde algunas fuentes diferentes. Las fuentes en orden de preferencia son:

1. La variable POST `_method`
2. El encabezado The `X_HTTP_METHOD_OVERRIDE`.
3. El encabezado `REQUEST_METHOD`

La variable POST `_method` es útil para utilizar un navegador como cliente REST (o cualquier otra cosa que pueda realizar POST). Simplemente, establece el valor de `_method()` con el nombre del método de la solicitud HTTP que deseas emular.

Crear Rutas de Recursos Anidadas

Una vez hayas conectado recursos en un alcance, también puedes conectar rutas para subrecursos. Las rutas de subrecursos estarán precedidas por el nombre del recurso original y un parámetro de identificación. Por ejemplo:

```
$routes->scope('/api', function (RouteBuilder $routes) {
    $routes->resources('Articles', function (RouteBuilder $routes) {
        $routes->resources('Comments');
    });
});
```

Generará rutas de recursos tanto para `articles` como para `comments`. Las rutas de comments se verán así:

```
/api/articles/{article_id}/comments
/api/articles/{article_id}/comments/{id}
```

Puedes obtener el `article_id` en `CommentsController` mediante:

```
$this->request->getParam('article_id');
```

De forma predeterminada, las rutas de recursos se asignan al mismo prefijo que el ámbito contenedor. Si tienes controladores de recursos anidados y no anidados, puedes utilizar un controlador diferente en cada contexto mediante el uso de prefijos:

```
$routes->scope('/api', function (RouteBuilder $routes) {
    $routes->resources('Articles', function (RouteBuilder $routes) {
        $routes->resources('Comments', ['prefix' => 'Articles']);
    });
});
```

Lo anterior asignará el recurso "Comments" a `App\Controller\Articles\CommentsController`. Tener controladores separados te permite mantener la lógica del controlador más simple. Los prefijos creados de esta manera son compatibles con *Enrutamiento de Prefijo*.

Nota: Si bien puedes anidar recursos con la profundidas que necesites, no se recomienda anidar más de dos recursos juntos.

Limitar las Rutas Creadas

Por defecto, CakePHP conectará 6 rutas para cada recurso. Si deseas conectar sólo rutas de recursos específicas podrás utilizar la opción `only`:

```
$routes->resources('Articles', [
    'only' => ['index', 'view']
]);
```

Crearía rutas de recurso de sólo lectura. Los nombres de las rutas son `create`, `update`, `view`, `index`, and `delete`.

El **nombre de ruta y acción del controlador utilizados** predeterminados son los siguientes:

Nombre de ruta	Acción del controlador utilizada
create	add
update	edit
view	view
index	index
delete	delete

Cambiar las Acciones del Controlador Utilizadas

Es posible que debas cambiar los nombres de las acciones del controlador que se utilizan al conectar rutas. Por ejemplo, si tu acción `edit()` se llama `put()` puedes utilizar la clave `actions` para renombrar las acciones utilizadas:

```
$routes->resources('Articles', [
    'actions' => ['update' => 'put', 'create' => 'add']
]);
```

Lo anterior utilizaría `put()` para la acción `edit()` y `add()` en lugar de `create()`.

Mapeo de Rutas de Recursos Adicionales

Puedes asignar métodos de recursos adicionales utilizando la opción `map`:

```
$routes->resources('Articles', [
   'map' => [
       'deleteAll' => [
           'action' => 'deleteAll',
           'method' => 'DELETE'
       ]
   ]
]);
// Esto conectaría /articles/deleteAll
```

Además de las rutas predeterminadas, esto también conectaría una ruta para */articles/delete-all*. De forma predeterminada, el segmento de ruta coincidirá con el nombre de la clave. Puedes utilizar la clave "path" dentro de la defición del recurso para personalizar el nombre de la ruta:

```
$routes->resources('Articles', [
    'map' => [
        'updateAll' => [
            'action' => 'updateAll',
            'method' => 'PUT',
            'path' => '/update-many',
        ],
    ],
]);
// Esto conectaría /articles/update-many
```

Si defines "only" and "map", asegúrate de que tus métodos asignados también están en la lista "only".

Enrutamiento de Recursos Prefijados

[[Continuar]] Las rutas de recursos pueden conectarse a los controladores en prefijos de enrutamiento conectando rutas en un ámbito prefijado or utilizando la opción `prefix`:

```
$routes->resources('Articles', [
    'prefix' => 'Api',
]);
```

Clases de Ruta Personalizada para Rutas de Recursos

Puedes proporcionar la clave `coneectOptions` en la matriz `$options` para `resources()` para proporcionar la configuración personalizada utilizada por `connect()`:

```
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->resources('Books', [
        'connectOptions' => [
            'routeClass' => 'ApiRoute',
        ]
    ];
});
```

Inflexión de URL para Rutas de Recursos

De forma predeterminada, los fragmentos de URL de los controladores con varias palabras están en la forma con guiones del nombre del controlador. Por ejemplo, el fragmento de URL de `BlogPostsController` sería **/blog-posts**.

Puedes especificar un tipo de inflexión alternativo utilizando la opción `inflect`:

```
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->resources('BlogPosts', [
        'inflect' => 'underscore' // Will use ``Inflector::underscore()``
    ]);
});
```

Lo anterior generará una URL del tipo: **/blog_posts**.

Cambiar el Elemento de Ruta

De forma predeterminada, las rutas de recursos utilizan una forma inflexionada del nombre del recurso para el segmento de URL. Puedes configurar un segmento de ruta personalizado con la opción `path`:

```
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->resources('BlogPosts', ['path' => 'posts']);
});
```

Argumentos Pasados

Los argumentos pasados son argumentos adicionales or segmentos de ruta que se utilizan al realizar una solicitud. A menudo se utilizan para pasar parámetros a los métodos de tu controlador:

```
http://localhost/calendars/view/recent/mark
```

En el ejemplo anterior, tanto `recent` como `mark` se pasan como argumentos a `CalendarsController::view()`. Los argumentos pasados se entregan a tus controladores de tres maneras. En primer lugar, como argumentos para el método de acción llamado y, en segundo lugar, están disponibles en `$this->request->getParams('pass')` como una matriz indexada numéricamente. Al utilizar rutas personalizadas, también puedes forzar que parámetros particulares entren en los argumentos pasados.

Si visitara la URL mencionada anteriormente, y tuviera una acción de controlador similar a:

```
class CalendarsController extends AppController
{
    public function view($arg1, $arg2)
    {
        debug(func_get_args());
    }
}
```

Otendrías el siguiente resultado:

```
Array
(
    [0] => recent
    [1] => mark
)
```

Estos mismos datos también están disponibles en `$this->request->getParam('pass')` en tus controladores, vistas y helpers. Los valores en la matriz de paso están indexados numéricamente según el orden en el que aparecen en la URL llamada:

```
debug($this->request->getParam('pass'));
```

Cualquiera de los anteriores generaría:

```
Array
(
    [0] => recent
    [1] => mark
)
```

Al generar URL, utilizando un *arreglo de enrutamiento* agregas argumentos pasados como valores sin claves de cadena en la matriz:

```
['controller' => 'Articles', 'action' => 'view', 5]
```

Dado que 5 es una clave numérica, se trata como un argumento pasado.

Generando URL

static Cake\Routing\RouterBuilder::**url**(*$url = null*, *$full = false*)

static Cake\Routing\RouterBuilder::**reverse**(*$params*, *$full = false*)

Generar URL o enrutamiento inverso es una característica de CakePHP que se utiliza para permitirte cambiar la estructura de tu URL sin tener que modificar todo tu código.

Si creas URL utilizando cadenas como:

```
$this->Html->link('View', '/articles/view/' . $id);
```

Y luego decides que `/articles` realmente debería llamarse `posts`, tendría que ir por toda tu aplicación renombrando las URL. Sin embargo, si definiste tu enlace como:

```
//`link()` utiliza internamente Router::url() y acepta una matriz de enrutamiento

$this->Html->link(
    'View',
    ['controller' => 'Articles', 'action' => 'view', $id]
);
```

o:

```
//'Router::reverse()' opera en la matriz de parámetos de la petición
//y producirá una cadena de url válida para `link()`

$requestParams = Router::getRequest()->getAttribute('params');
$this->Html->link('View', Router::reverse($requestParams));
```

Luego, cuando decidieras cambiar tus URL, podrías hacerlo definiendo una ruta. Esto cambiaría tanto la asignación de URL entrante como las URL generadas.

La elección de la técnica está determinada por qué tan bien se pueden predecir los elementos de la matriz de enrutamiento.

Utilizando `Router::url()`

`Router::url()` te permite utilizar *routing arrays* en situaciones donde los elementos requeridos de la matriz son fijos o se deducen fácilmente.

Proporcionará enrutamiento inverso cuando la url de destino esté bien definida:

```
$this->Html->link(
    'View',
    ['controller' => 'Articles', 'action' => 'view', $id]
);
```

También es útil cuando el destino es desconocido, pero sigue un patrón bien definido:

```
$this->Html->link(
    'View',
    ['controller' => $controller, 'action' => 'view', $id]
);
```

Los elementos con claves numéricas se tratan como *Argumentos Pasados*.

Al utilizar matrices de enrutamiento, puedes definir tanto los parámetros de la cadena de consulta como los fragmentos de documentos utilizando claves especiales:

```
$routes->url([
    'controller' => 'Articles',
    'action' => 'index',
    '?' => ['page' => 1],
    '#' => 'top'
]);

// Generará una URL del tipo.
/articles/index?page=1#top
```

También puedes utilizar cualquiera de los elementos de ruta especiales al generar URL:

- `_ext` Se utiliza para enrutamiento de *Enrutamiento de Extensiones de Archivo*.
- `_base` Establecer en `false` para eliminar la ruta base de la URL generada. Si tu aplicación no está en el directorio raíz, esto se puede utilizar para generar URL relativas a cake.
- `_scheme` Configurado para crear enlaces en diferentes esquemas como `webcal` o `ftp`. El valor predeterminado es el esquema actual.
- `_host` Establece el host que se utilizará en el enlace. El valor por defecto es el del host actual.
- `_port` Establece el puerto si necesitas crear enlaces a puestos no estándar.
- `_method` Define el verbo HTTP para el que es la URL.
- `_full` Si es `true` el valor de `App.fullBaseUrl` mencionado en *Configuración General* se antepondrá a las URL generadas.
- `_https` Establecer en `true` para convertir la URL generada a https o `false` para forzar http.
- `_name` Nombre de la ruta. Si has configurado rutas con nombre, puedes utilizar esta clave para especificarlas.

Utilizando `Router::reverse()`

`Router::reverse()` te permite utilizar los *Parámetros de la solicitud* en casos donde la URL actual con algunas modificaciones es la base para el destino y los elementos de la URL actual son impredecibles.

Como ejemplo, imagina un blog que permite a los usuarios crear **Articles** y **Comments**, y marcar ambos como *published* o *draft*. Ambas URL de la página index pueden incluir el ID del usuario. La URL de **Comments** también puede incluir el ID de un **Article** para identificar a qué artículo se refieren los comentarios.

Aquí están las url para este escenario:

```
/articles/index/42
/comments/index/42/18
```

Cuando el autor utilice estas páginas, sería conveniente incluir enlaces que permitan mostrar la página con todos los resultados, sólo publicados o sólo borrador.

Para mantener el código DRY, sería mejor incluir los enlaces a través de un elemento:

```
// element/filter_published.php
```

(continué en la próxima página)

(proviene de la página anterior)

```
$params = $this->getRequest()->getAttribute('params');

/* preparar la url para Borrador */
$params = Hash::insert($params, '?.published', 0);
echo $this->Html->link(__('Draft'), Router::reverse($params));

/* Preparar la url para Publicados */
$params = Hash::insert($params, '?.published', 1);
echo $this->Html->link(__('Published'), Router::reverse($params));

/* Preparar la url para Todos */
$params = Hash::remove($params, '?.published');
echo $this->Html->link(__('All'), Router::reverse($params));
```

Los enlaces generados por estas llamadas incluirían uno o dos parámetros de paso dependiendo de la estructura de la URL actual. Y el código funcionaría para cualquier URL futura, por ejemplo, si comenzara a usar prefijos de ruta o si agregara más parámetros del paso.

Matrices de Enrutamiento vs Parámetros de Solicitud

La diferencia significativa entre las dos matrices y su uso en estos métodos de enrutamiento inverso está la forma en la que incluyen los parámetros de paso.

Las matrices de enrutamiento incluyen los parámetros de paso como valores sin clave en la matriz:

```
$url = [
    'controller' => 'Articles',
    'action' => 'View',
    $id, //un parámetro de paso
    'page' => 3, //un argumento de consulta
];
```

Los parámetros de consulta incluyen parámtros de paso en la clave “pass” de la matriz:

```
$url = [
    'controller' => 'Articles',
    'action' => 'View',
    'pass' => [$id], //los parámetros de paso
    '?' => ['page' => 3], //los parámtros de consulta
];
```

Por lo tanto, si los deseas, es posible convertir los parámetros de solicitud en una matriz de enrutamiento o viceversa.

Generando URL de Activos

La clase `Asset` proporciona métodos para generar URL para los archivos css, javascript, imágenes y otros archivos de activos estáticos de tu aplicación:

```
use Cake\Routing\Asset;

// Generar una URL para APP/webroot/js/app.js
$js = Asset::scriptUrl('app.js');

// Generar una URL para APP/webroot/css/app.css
$css = Asset::cssUrl('app.css');

// Generar una URL para APP/webroot/image/logo.png
$img = Asset::imageUrl('logo.png');

// Generar una URL para APP/webroot/files/upload/photo.png
$file = Asset::url('files/upload/photo.png');
```

Los métodos anteriores también aceptan una matriz de opciones como segundo parámetro:

- `fullBase` Agrega la URL completa con el nombre de dominio.
- `pathPrefix` Prefijo de ruta para URL relativas.
- `plugin` Puedes añdirlo como `false` para evitar que las rutas se traten como un recurso de un plugin.
- `timestamp` Sobrescribe el valor de `Asset.timestamp` en Configure. Establecer a `false` para omitir la generación de la marca de tiempo. Establecer a `true` para aplicar marcas de tiempo cuando el debug is true. Set to `'force'` para habilitar siempre la marca de tiempo independientemente del valor de debug.

```
// Genera http://example.org/img/logo.png
$img = Asset::url('logo.png', ['fullBase' => true]);

// Genera /img/logo.png?1568563625
// Donde la marca de tiempo es la ultima hora de modificación del archivo
$img = Asset::url('logo.png', ['timestamp' => true]);
```

Para generar URL de activo para archivos en un plugin utiliza *Sintaxis de plugin*:

```
// Genera `/debug_kit/img/cake.png`
$img = Asset::imageUrl('DebugKit.cake.png');
```

Redirección de Enrutamiento

La redirección de enrutamiento te permite emitir redirección de estado HTTP 30x para rutas entrantes y apuntarlas a URL diferentes. Esto es útil cuando deseas informar a las aplicaciones cliente que un recurso se ha movido y no deseas exponer dos URL para el mismo contenido.

Las rutas de redireccionamiento son diferentes de las rutas normales en que realizan una redirección de encabezado real si se encuentra una coincidencia. La redirección puede ocurrir a un destino dentro de tu aplicación o a una ubicación externa:

```php
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->redirect(
        '/home/*',
        ['controller' => 'Articles', 'action' => 'view'],
        ['persist' => true]
        // O ['persist'=>['id']] para el enrutamiento predeterminado
        // donde la acción view espera $id como argumento.
    );
})
```

Redirige `/home/*` a `/articles/view` y pasa los parámetros a `/articles/view`. El uso de una matriz como destino de redireccionamiento te permite utilizar otras rutas para definir hacia dónde se debe dirigir una cadena de URL. Puedes redirigir a ubicaciones externas utilizando una cadena de URL como destino:

```php
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->redirect('/articles/*', 'http://google.com', ['status' => 302]);
});
```

Esto redirigiría `/articles/*` a `http://google.com` con un estado HTTP de 302.

Enrutamiento de Entidades

El enrutamiento de entidades te permite utilizar una entidad, una matriz o un objeto que implemente `ArrayAccess` como una fuente de parámetros de enrutamiento. Esto te permite refactorizar rutas fácilmente y generar URL con menos código. Por ejemplo, si comienzas con una ruta similar a:

```php
$routes->get(
    '/view/{id}',
    ['controller' => 'Articles', 'action' => 'view'],
    'articles:view'
);
```

Puedes generar URL para esta ruta utilizando:

```php
// $article es una entidad en el ámbito local.
Router::url(['_name' => 'articles:view', 'id' => $article->id]);
```

Más adelante, es posible que desees exponer el slug del artículo en la URL con fines de SEO. Para hacer esto necesitarás actualizar todos los lugares donde generes una URL a la ruta `articles:view`, lo que podría llevar algún tiempo. Si utilizamos rutas de entidad, pasamos toda la entidad del artículo a la generación de URL, lo que nos permite omitir cualquier reelaboración cuando las URL requiren más parámetros:

```php
use Cake\Routing\Route\EntityRoute;

// Crea más rutas de entidad para el resto de este ámbito.
$routes->setRouteClass(EntityRoute::class);

// Crea la ruta como antes.
$routes->get(
    '/view/{id}/{slug}',
    ['controller' => 'Articles', 'action' => 'view'],
```

(continué en la próxima página)

(proviene de la página anterior)

```
    'articles:view'
);
```

Ahora podemos generar URL utilizando la clave `_entity`:

```
Router::url(['_name' => 'articles:view', '_entity' => $article]);
```

Esto extraerá las propiedades `id` y `slug` de la entidad proporcionada.

Clases de Ruta Personalizadas

Las clases de ruta personalizadas te permiten ampliar y cambiar la forma en que las rutas individuales analizan las solicitudes y manejan el enrutamiento inverso. Las clases de ruta tienen algunas convenciones:

- Se espera que las clases de ruta se encuentren en el espacio de nombres `Routing\\Route` de tu aplicación o plugin.
- Las clases de ruta deben extender `Cake\Routing\Route\Route`.
- Las clases de ruta deben implementar uno o ambos `match()` y/o `parse()`.

El método `parse()` se utiliza para analizar una URL entrante. Debería generar una matriz de parámetros de solicitud que pueda resolverse en un controlador y acción. Este método devuelve `null` para indicar un error de coincidencia.

El método `match()` se utiliza para hacer coincidir una matriz de parámetros de URL y crear una cadena de URL. Si los parámertos de la URL no coinciden, se debe devolver la ruta `false`.

Puedes utilizar una clase de ruta personalizada al realizar una ruta utilizando la opción `routeClass`:

```
$routes->connect(
    '/{slug}',
    ['controller' => 'Articles', 'action' => 'view'],
    ['routeClass' => 'SlugRoute']
);

// O configurando routeClass en su ámbito.
$routes->scope('/', function (RouteBuilder $routes) {
    $routes->setRouteClass('SlugRoute');
    $routes->connect(
        '/{slug}',
        ['controller' => 'Articles', 'action' => 'view']
    );
});
```

Esta ruta crearía una instancia de `SlugRoute` y te permitiría implementar un manejo de parámetros personalizado. Puedes utilizar clases de ruta de plugin utilizando el estándar *Sintaxis de plugin*.

Clase de Ruta Predeterminada

static Cake\Routing\RouterBuilder::**setRouteClass**(*$routeClass = null*)

Si desea utilizar una ruta de clase alternativa para tus rutas además de la `Ruta` predeterminada, puedes hacerlo llamando a `RouterBuilder::setRouteClass()` antes de configurar cualquier ruta y evitar tener que especificar la opción `routeClass` para cada ruta. Por ejemplo utilizando:

```
use Cake\Routing\Route\DashedRoute;

$routes->setRouteClass(DashedRoute::class);
```

Hará que todas las rutas conectadas después de esto utilicen la clase de ruta `DashedRoute`. Llamando al método sin un argumento devolverá la clase de ruta predeterminada actual.

Método de Respaldo/Alternativas

Cake\Routing\RouterBuilder::**fallbacks**(*$routeClass = null*)

El método de respaldo es un atajo simple para definir rutas predeterminadas. El método utiliza la clase de enrutamiento pasada para las reglas definidas o, si no se proporciona ninguna clase, se utiliza la clase devuelta por `RouterBuilder::setRouteClass()`.

Llamar a alternativas así:

```
use Cake\Routing\Route\DashedRoute;

$routes->fallbacks(DashedRoute::class);
```

Es equivalente a las siguientes llamadas explícitas:

```
use Cake\Routing\Route\DashedRoute;

$routes->connect('/{controller}', ['action' => 'index'], ['routeClass' =>␣
↪DashedRoute::class]);
$routes->connect('/{controller}/{action}/*', [], ['routeClass' => DashedRoute::class]);
```

Nota: El uso de la clase de ruta predeterminada (`Route`) con alternativas, or cualquier ruta con elemento de ruta `{plugin}` o `{controller}` dará como resultado una URL inconsistente.

Advertencia: Las plantillas de ruta alternativas son muy genéricas y permites generar y analizar URL para controladore sy acciones que no existen. Las URL alternativas también pueden introducir ambigüedad y duplicidad en tus URL.

A medida que tu aplicaicón crece, se recomienda alejarse de las URL alternativas y definir explícitamente las rutas en tu aplicación.

Crear Parámetros de URL Persistentes

Puedes conectarte al proceso de generación de URL utilizando funciones de filtro de URL. Las funciones de filtro se llaman *antes* de que las URL coincidan con las rutas, esto te permite preparar las URL antes de enrutarlas.

La devolución de llamada de las funciones de filtro deben contar con los siguientes parámetos:

- `$params` La matriz de parámetros de URL que se está procensando.
- `$request` La petición actual (instancia de `Cake\Http\ServerRequest`).

La función de filtro de URL *siempre* debería devolver los parámetros incluso si no están modificados.

Los filtros de URL te permiten implementar funciones como parámetros persistentes:

```
Router::addUrlFilter(function (array $params, ServerRequest $request) {
    if ($request->getParam('lang') && !isset($params['lang'])) {
        $params['lang'] = $request->getParam('lang');
    }

    return $params;
});
```

Las funciones de filtro se aplican en el orden en que están conectadas.

Otro caso de uso es cambiar una determinada ruta en tiempo de ejecución (por ejemplo, rutas de plugin):

```
Router::addUrlFilter(function (array $params, ServerRequest $request) {
    if (empty($params['plugin']) || $params['plugin'] !== 'MyPlugin' || empty($params[
↪'controller'])) {
        return $params;
    }
    if ($params['controller'] === 'Languages' && $params['action'] === 'view') {
        $params['controller'] = 'Locations';
        $params['action'] = 'index';
        $params['language'] = $params[0];
        unset($params[0]);
    }

    return $params;
});
```

Esto alterará la siguiente ruta:

```
Router::url(['plugin' => 'MyPlugin', 'controller' => 'Languages', 'action' => 'view', 'es
↪']);
```

En esto:

```
Router::url(['plugin' => 'MyPlugin', 'controller' => 'Locations', 'action' => 'index',
↪'language' => 'es']);
```

Advertencia: Si estás utilizando las funcionees de almacenamiento de caché de routing-middleware debes definir los filtros de URL en tu aplicación `bootstrap()` ya que los filtros no son parte de los datos almacenados en caché.

CAPÍTULO 8

Objetos de Solicitud y Respuesta

Los objetos de solicitud y respuesta proporcionan una abstracción en torno a las solicitudes y respuestas HTTP. El objeto de solicitud en CakePHP le permite realizar una introspección de una solicitud entrante, mientras que el objeto de respuesta le permite crear respuestas HTTP sin esfuerzo desde sus controladores.

Solicitud (Request)

class Cake\Http**ServerRequest**

`ServerRequest` es el objeto de solicitud predeterminado utilizado en CakePHP. Centraliza una serie de funciones para interrogar e interactuar con los datos de la solicitud. En cada solicitud, se crea un Request y luego se pasa por referencia a las distintas capas de una aplicación que utiliza datos de solicitud. De forma predeterminada, la solicitud se asigna a `$this->request` y está disponible en Controladores, Celdas, Vistas y Ayudantes. También puede acceder a él en Componentes usando la referencia del controlador.

Algunas de las tareas que realiza `ServerRequest` incluyen:

- Procesar los arreglos GET, POST y FILES en las estructuras de datos con las que está familiarizado.
- Proporcionar una introspección del entorno correspondiente a la solicitud. Información como los encabezados enviados, la dirección IP del cliente y los nombres de subdominio/dominio en el servidor en el que se ejecuta su aplicación.
- Proporcionar acceso a los parámetros de solicitud tanto como índices de matriz como propiedades de objetos.

El objeto de la solicitud de CakePHP implementa PSR-7 ServerRequestInterface[90], lo que facilita el uso de bibliotecas desde fuera de CakePHP.

[90] https://www.php-fig.org/psr/psr-7/

Parámetros de la solicitud

La solicitud expone los parámetros de enrutamiento a través del método `getParam()`:

```
$controllerName = $this->request->getParam('controller');
```

Para obtener todos los parámetros de enrutamiento como una matriz, use `getAttribute()`:

```
$parameters = $this->request->getAttribute('params');
```

Se accede a todos los *Elementos de ruta* a través de esta interfaz.

Además de *Elementos de ruta*, a menudo también necesita acceso a *Argumentos Pasados*. Ambos también están disponibles en el objeto de solicitud:

> // Argumentos pasados $passedArgs = $this->request->getParam("pass");

Todos le proporcionarán acceso a los argumentos pasados. Hay varios parámetros importantes/útiles que CakePHP usa internamente, y todos ellos también se encuentran en los parámetros de enrutamiento:

- `plugin` El complemento que maneja la solicitud. Será nulo cuando no haya ningún complemento.
- `controller` El controlador que maneja la solicitud actual.
- `action` La acción que maneja la solicitud actual.
- `prefix` El prefijo de la acción actual. Consulte *Enrutamiento de Prefijo* para obtener más información.

Parámetros de cadena de consulta

`Cake\Http\ServerRequest::getQuery`(*$name*, *$default = null*)

Los parámetros de la cadena de consulta se pueden leer usando el método `getQuery()`:

```
// URL es /posts/index?page=1&sort=title
$page = $this->request->getQuery('page');
```

Puede acceder directamente a la propiedad de consulta o puede utilizar el método `getQuery()` para leer la matriz de consulta de URL sin errores. Cualquier clave que no exista devolverá `null`:

```
$foo = $this->request->getQuery('value_that_does_not_exist');
// $foo === null

// También puede proporcionar valores predeterminados
$foo = $this->request->getQuery('does_not_exist', 'default val');
```

Si desea acceder a todos los parámetros de consulta, puede utilizar `getQueryParams()`:

> $query = $this->request->getQueryParams();

Datos del cuerpo de la solicitud

Cake\Http\ServerRequest::**getData**(*$name*, *$default = null*)

Se puede acceder a todos los datos POST normalmente disponibles a través de la variable global `$_POST` de PHP usando *Cake\Http\ServerRequest::getData()*. Por ejemplo:

```
// Se puede acceder a una entrada con un atributo de nombre 'título'
$title = $this->request->getData('title');
```

Puede utilizar nombres separados por puntos para acceder a datos anidados. Por ejemplo:

```
$value = $this->request->getData('address.street_name');
```

Para nombres inexistentes se devolverá el valor `$default`:

```
$foo = $this->request->getData('value.that.does.not.exist');
// $foo == null
```

También puede utilizar body-parser-middleware para analizar el cuerpo de la solicitud de diferentes tipos de contenido en una matriz, de modo que sea accesible a través de `ServerRequest::getData()`.

Si desea acceder a todos los parámetros de datos, puede utilizar `getParsedBody()`:

```
$data = $this->request->getParsedBody();
```

Cargas de archivos

Se puede acceder a los archivos cargados a través de los datos del cuerpo de la solicitud, utilizando el método *Cake\Http\ServerRequest::getData()* descrito anteriormente. Por ejemplo, se puede acceder a un archivo desde un elemento de entrada con un atributo de nombre `attachment` de esta manera:

```
$attachment = $this->request->getData('attachment');
```

De forma predeterminada, las cargas de archivos se representan en los datos de la solicitud como objetos que implementan \Psr\Http\Message\UploadedFileInterface[91]. En la implementación actual, la variable `$attachment` en el ejemplo anterior contendría de forma predeterminada una instancia de `\Laminas\Diactoros\UploadedFile`.

Acceder a los detalles del archivo cargado es bastante simple, así es como puede obtener los mismos datos que proporciona la matriz de carga de archivos de estilo antiguo:

> $name = $attachment->getClientFilename(); $type = $attachment->getClientMediaType(); $size = $attachment->getSize(); $tmpName = $attachment->getStream()->getMetadata("uri"); $error = $attachment->getError();

Mover el archivo cargado desde su ubicación temporal a la ubicación de destino deseada no requiere acceder manualmente al archivo temporal, sino que se puede hacer fácilmente usando el método `moveTo()` del objeto:

```
$attachment->moveTo($targetPath);
```

En un entorno HTTP, el método `moveTo()` validará automáticamente si el archivo es un archivo cargado real y generará una excepción en caso de que sea necesario. En un entorno CLI, donde no existe el concepto de cargar archivos, permitirá mover el archivo al que ha hecho referencia independientemente de sus orígenes, lo que hace posible probar la carga de archivos.

[91] https://www.php-fig.org/psr/psr-7/#16-uploaded-archivos

Cake\Http\ServerRequest::**getUploadedFile**(*$path*)

Devuelve el archivo cargado en una ruta específica. La ruta utiliza la misma sintaxis de puntos que el método *Cake\Http\ServerRequest::getData()*:

```
$attachment = $this->request->getUploadedFile('attachment');
```

A diferencia de *Cake\Http\ServerRequest::getData()*, *Cake\Http\ServerRequest::getUploadedFile()* solo devolvería datos cuando exista una carga de archivo real para la ruta dada, si hay datos regulares del cuerpo de la solicitud que no son archivos presentes en la ruta dada, entonces este método devolverá «nulo», tal como lo haría para cualquier ruta inexistente.

Cake\Http\ServerRequest::**getUploadedFiles**()

Devuelve todos los archivos cargados en una estructura de matriz normalizada. Para el ejemplo anterior con el nombre de entrada del archivo `attachment`, la estructura se vería así:

```
[
      'attachment' => object(Laminas\Diactoros\UploadedFile) {
          // ...
      }
]
```

Cake\Http\ServerRequest::**withUploadedFiles**(*array $files*)

Este método establece los archivos cargados del objeto de solicitud, acepta una matriz de objetos que implementan \Psr\Http\Message\UploadedFileInterface[92]. Reemplazará todos los archivos cargados posiblemente existentes:

```
$files = [
    'MyModel' => [
        'attachment' => new \Laminas\Diactoros\UploadedFile(
            $streamOrFile,
            $size,
            $errorStatus,
            $clientFilename,
            $clientMediaType
        ),
        'anotherAttachment' => new \Laminas\Diactoros\UploadedFile(
            '/tmp/hfz6dbn.tmp',
            123,
            \UPLOAD_ERR_OK,
            'attachment.txt',
            'text/plain'
        ),
    ],
];

$this->request = $this->request->withUploadedFiles($files);
```

Nota: Los archivos cargados que se agregaron a la solicitud a través de este método *no* estarán disponibles en los datos del cuerpo de la solicitud, es decir, no puede recuperarlos a través de *Cake\Http\ServerRequest::getData()* ! Si los necesita en los datos de la solicitud (también), entonces debe configurarlos mediante `Cake\Http\ServerRequest::withData()` o `Cake\HttpServerRequest::withParsedBody()`.

[92] https://www.php-fig.org/psr/psr-7/#16-uploaded-files

PUT, PATCH o DELETE Datos

Cake\Http\ServerRequest::**input**(*$callback*[, *$options*])

Al crear servicios REST, a menudo se aceptan datos de solicitud en solicitudes PUT y DELETE. Cualquier dato del cuerpo de solicitud `application/x-www-form-urlencoded` se analizará automáticamente y se establecerá en `$this->data` para las solicitudes PUT y DELETE. Si acepta datos JSON o XML, consulte a continuación cómo puede acceder a esos cuerpos de solicitud.

Al acceder a los datos de entrada, puede decodificarlos con una función opcional. Esto resulta útil al interactuar con el contenido del cuerpo de la solicitud XML o JSON. Se pueden pasar parámetros adicionales para la función de decodificación como argumentos a `input()`:

```
$jsonData = $this->request->input('json_decode');
```

Variables de entorno (de $_SERVER y $_ENV)

Cake\Http\ServerRequest::**putenv**(*$key*, *$value = null*)

`ServerRequest::getEnv()` es un contenedor para la función global `getenv()` y actúa como un captador/establecedor de variables de entorno sin tener que modificar los globales `$_SERVER` y `$_ENV`:

```
// Obtener el host
$host = $this->request->getEnv('HTTP_HOST');

// Establecer un valor, generalmente útil en las pruebas.
$this->request->withEnv('REQUEST_METHOD', 'POST');
```

Para acceder a todas las variables de entorno en una solicitud, utilice `getServerParams()`:

```
$env = $this->request->getServerParams();
```

Datos XML o JSON

Las aplicaciones que emplean *REST* a menudo intercambian datos en cuerpos de publicaciones sin codificación URL. Puede leer datos de entrada en cualquier formato usando *`input()`*. Al proporcionar una función de decodificación, puede recibir el contenido en un formato deserializado:

```
// Obtenga datos codificados en JSON enviados a una acción PUT/POST
$jsonData = $this->request->input('json_decode');
```

Algunos métodos de deserialización requieren parámetros adicionales cuando se llaman, como el parámetro "as array" en `json_decode`. Si desea convertir XML en un objeto DOMDocument, *`input()`* también admite el paso de parámetros adicionales:

```
// Obtener datos codificados en XML enviados a una acción PUT/POST
$data = $this->request->input('Cake\Utility\Xml::build', ['return' => 'domdocument']);
```

Información de ruta

El objeto de solicitud también proporciona información útil sobre las rutas de su aplicación. Los atributos `base` y `webroot` son útiles para generar URL y determinar si su aplicación está o no en un subdirectorio. Los atributos que puedes utilizar son:

> // Supongamos que la URL de solicitud actual es /subdir/articles/edit/1?page=1
>
> // Contiene /subdir/articles/edit/1?page=1 $here = $request->getRequestTarget();
>
> // Contiene /subdir $base = $request->getAttribute("base");
>
> // Contiene /subdir/ $base = $request->getAttribute("webroot");

Comprobación de las condiciones de la solicitud

`Cake\Http\ServerRequest::is`(*$type*, *$args...*)

El objeto de solicitud proporciona una forma de inspeccionar ciertas condiciones en una solicitud determinada. Al utilizar el método `is()`, puede comprobar una serie de condiciones comunes, así como inspeccionar otros criterios de solicitud específicos de la aplicación:

> $isPost = $this->request->is("post");

También puede ampliar los detectores de solicitudes que están disponibles, utilizando *Cake\Http\ServerRequest::addDetector()* para crear nuevos tipos de detectores. Hay diferentes tipos de detectores que puedes crear:

- Comparación de valores del entorno: compara un valor obtenido de `env()` para determinar su igualdad con el valor proporcionado.
- Comparación del valor del encabezado: si el encabezado especificado existe con el valor especificado o si el invocable devuelve verdadero.
- Comparación de valores de patrón: la comparación de valores de patrón le permite comparar un valor obtenido de `env()` con una expresión regular.
- Comparación basada en opciones: las comparaciones basadas en opciones utilizan una lista de opciones para crear una expresión regular. Las llamadas posteriores para agregar un detector de opciones ya definido fusionarán las opciones.
- Detectores de devolución de llamada: los detectores de devolución de llamada le permiten proporcionar un tipo de «callback» para manejar la verificación. La devolución de llamada recibirá el objeto de solicitud como único parámetro.

`Cake\Http\ServerRequest::addDetector`(*$name*, *$options*)

Algunos ejemplos serían:

```
// Agregue un detector de entorno.
$this->request->addDetector(
    'post',
    ['env' => 'REQUEST_METHOD', 'value' => 'POST']
);

// Agregue un detector de valor de patrón.
$this->request->addDetector(
    'iphone',
    ['env' => 'HTTP_USER_AGENT', 'pattern' => '/iPhone/i']
```

(continué en la próxima página)

(proviene de la página anterior)

```
);

// Agregar un detector de opciones
$this->request->addDetector('internalIp', [
    'env' => 'CLIENT_IP',
    'options' => ['192.168.0.101', '192.168.0.100']
]);


// Agregue un detector de encabezado con comparación de valores
$this->request->addDetector('fancy', [
    'env' => 'CLIENT_IP',
    'header' => ['X-Fancy' => 1]
]);

// Agregue un detector de encabezado con comparación invocable
$this->request->addDetector('fancy', [
    'env' => 'CLIENT_IP',
    'header' => ['X-Fancy' => function ($value, $header) {
        return in_array($value, ['1', '0', 'yes', 'no'], true);
    }]
]);

// Agregue un detector de devolución de llamada. Debe ser un invocable válido.
$this->request->addDetector(
    'awesome',
    function ($request) {
        return $request->getParam('awesome');
    }
);

// Agregue un detector que use argumentos adicionales.
$this->request->addDetector(
    'csv',
    [
        'accept' => ['text/csv'],
        'param' => '_ext',
        'value' => 'csv',
    ]
);
```

Hay varios detectores integrados que puedes utilizar:

- `is('get')` Verifique si la solicitud actual es un GET.
- `is('put')` Verifique si la solicitud actual es un PUT.
- `is('patch')` Verifique si la solicitud actual es un PATCH.
- `is('post')` Verifique si la solicitud actual es una POST.
- `is('delete')` Verifique si la solicitud actual es DELETE.
- `is('head')` Verifique si la solicitud actual es HEAD.
- `is('options')` Verifique si la solicitud actual es OPTIONS.

- `is('ajax')` Verifique si la solicitud actual vino con X-Requested-With = XMLHttpRequest.
- `is('ssl')` Compruebe si la solicitud se realiza a través de SSL.
- `is('flash')` Verifique si la solicitud tiene un User-Agent de Flash.
- `is('json')` Verifique si la solicitud tiene la extensión "json" y acepte el tipo mime "application/json".
- `is('xml')` Verifique si la solicitud tiene la extensión "xml" y acepte el tipo mime "application/xml" o "text/xml".

`ServerRequest` También incluye métodos como *Cake\Http\ServerRequest::domain()*, *Cake\Http\ServerRequest::subdomains()* y *Cake\Http\ServerRequest::host()* para simplificar las aplicaciones que utilizan subdominios.

Datos de sesión

Para acceder a la sesión para una solicitud determinada utilice el método `getSession()` o utilice el atributo `session`:

```
$session = $this->request->getSession();
$session = $this->request->getAttribute('session');

$data = $session->read('sessionKey');
```

Para obtener más información, consulte la documentación *Sesiones* sobre cómo utilizar el objeto de sesión.

Host y nombre de dominio

`Cake\Http\ServerRequest::`**`domain`**(*$tldLength = 1*)

Devuelve el nombre de dominio en el que se ejecuta su aplicación:

```
// Muestra 'example.org'
echo $request->domain();
```

`Cake\Http\ServerRequest::`**`subdomains`**(*$tldLength = 1*)

Devuelve los subdominios en los que se ejecuta su aplicación como una matriz:

```
// Regresa ['my', 'dev'] de 'my.dev.example.org'
$subdomains = $request->subdomains();
```

`Cake\Http\ServerRequest::`**`host`**()

Devuelve el host en el que se encuentra su aplicación:

```
// Muestra 'my.dev.example.org'
echo $request->host();
```

Leyendo el método HTTP

`Cake\Http\ServerRequest::getMethod()`

Devuelve el método HTTP con el que se realizó la solicitud:

```
// Salida POST
echo $request->getMethod();
```

Restringir qué método HTTP acepta una acción

`Cake\Http\ServerRequest::allowMethod($methods)`

Establecer métodos HTTP permitidos. Si no coincide, arrojará `MethodNotAllowedException`. La respuesta 405 incluirá el encabezado `Allow` requerido con los métodos pasados:

```
public function delete()
{
    // Solo acepte solicitudes POST y DELETE
    $this->request->allowMethod(['post', 'delete']);
    ...
}
```

Lectura de encabezados HTTP

Le permite acceder a cualquiera de los encabezados `HTTP_*` que se utilizaron para la solicitud. Por ejemplo:

```
// Obtener el encabezado como una cadena
$userAgent = $this->request->getHeaderLine('User-Agent');

// Obtenga una matriz de todos los valores.
$acceptHeader = $this->request->getHeader('Accept');

// Comprobar si existe un encabezado
$hasAcceptHeader = $this->request->hasHeader('Accept');
```

Si bien algunas instalaciones de Apache no hacen que el encabezado `Authorization` sea accesible, CakePHP lo hará disponible a través de métodos específicos de Apache según sea necesario.

`Cake\Http\ServerRequest::referer($local = true)`

Devuelve la dirección de referencia de la solicitud.

`Cake\Http\ServerRequest::clientIp()`

Devuelve la dirección IP del visitante actual.

Confiar en los encabezados de proxy

Si su aplicación está detrás de un balanceador de carga o se ejecuta en un servicio en la nube, a menudo obtendrá el host, el puerto y el esquema del balanceador de carga en sus solicitudes. A menudo, los balanceadores de carga también enviarán encabezados `HTTP-X-Forwarded-*` con los valores originales. CakePHP no utilizará los encabezados reenviados de fábrica. Para que el objeto de solicitud utilice estos encabezados, establezca la propiedad `trustProxy` en `true`:

```
$this->request->trustProxy = true;

// Estos métodos ahora utilizarán los encabezados proxy.
$port = $this->request->port();
$host = $this->request->host();
$scheme = $this->request->scheme();
$clientIp = $this->request->clientIp();
```

Una vez que se confía en los servidores proxy, el método `clientIp()` utilizará la *última* dirección IP en el encabezado `X-Forwarded-For`. Si su aplicación está detrás de varios servidores proxy, puede usar `setTrustedProxies()` para definir las direcciones IP de los servidores proxy bajo su control:

```
$request->setTrustedProxies(['127.1.1.1', '127.8.1.3']);
```

Después de que los servidores proxy sean confiables, `clientIp()` usará la primera dirección IP en el encabezado `X-Forwarded-For` siempre que sea el único valor que no provenga de un proxy confiable.

Comprobando encabezados aceptados

`Cake\Http\ServerRequest::accepts`(*$type = null*)

Descubra qué tipos de contenido acepta el cliente o compruebe si acepta un tipo de contenido en particular.

Consigue todos los tipos:

```
$accepts = $this->request->accepts();
```

Consulta por un solo tipo:

```
$acceptsJson = $this->request->accepts('application/json');
```

`Cake\Http\ServerRequest::acceptLanguage`(*$language = null*)

Obtenga todos los idiomas aceptados por el cliente o verifique si se acepta un idioma específico.

Obtenga la lista de idiomas aceptados:

```
$acceptsLanguages = $this->request->acceptLanguage();
```

Compruebe si se acepta un idioma específico:

```
$acceptsSpanish = $this->request->acceptLanguage('es-es');
```

Leyendo Cookies

Las cookies de solicitud se pueden leer a través de varios métodos:

> // Obtenga el valor de la cookie, o nulo si falta la cookie. $rememberMe = $this->request->getCookie("remember_me");
>
> // Lea el valor u obtenga el valor predeterminado de 0 $rememberMe = $this->request->getCookie("remember_me", 0);
>
> // Obtener todas las cookies como hash $cookies = $this->request->getCookieParams();
>
> // Obtener una instancia de CookieCollection $cookies = $this->request->getCookieCollection()

Consulte la documentación *Cake\Http\Cookie\CookieCollection* para saber cómo trabajar con la recopilación de cookies.

Archivos cargados

Las solicitudes exponen los datos del archivo cargado en `getData()` o `getUploadedFiles()` como objetos `UploadedFileInterface`:

```
// Obtener una lista de objetos UploadedFile
$files = $request->getUploadedFiles();

// Lea los datos del archivo.
$files[0]->getStream();
$files[0]->getSize();
$files[0]->getClientFileName();

// Mover el archivo
$files[0]->moveTo($targetPath);
```

Manipulación de URI

Las solicitudes contienen un objeto URI, que contiene métodos para interactuar con el URI solicitado:

```
// Obtener la URI
$uri = $request->getUri();

// Leer datos de la URI.
$path = $uri->getPath();
$query = $uri->getQuery();
$host = $uri->getHost();
```

Respueta (Response)

class Cake\Http**Response**

Cake\Http\Response es la clase de respuesta predeterminada en CakePHP. Encapsula una serie de características y funcionalidades para generar respuestas HTTP en su aplicación. También ayuda en las pruebas, ya que se puede simular o eliminar, lo que le permite inspeccionar los encabezados que se enviarán.

`Response` proporciona una interfaz para envolver las tareas comunes relacionadas con la respuesta, como por ejemplo:

- Envío de encabezados para redireccionamientos.
- Envío de encabezados de tipo de contenido.
- Envío de cualquier encabezado.
- Envío del cuerpo de la respuesta.

Tratar con tipos de contenido

Cake\Http\Response::**withType**(*$contentType = null*)

Puede controlar el tipo de contenido de las respuestas de su aplicación con *Cake\Http\Response::withType()*. Si su aplicación necesita manejar tipos de contenido que no están integrados en Response, también puede asignarlos con `setTypeMap()`:

```
// Agregar un tipo de vCard
$this->response->setTypeMap('vcf', ['text/v-card']);

// Establezca el tipo de contenido de respuesta en vcard
$this->response = $this->response->withType('vcf');
```

Por lo general, querrás asignar tipos de contenido adicionales en la devolución de llamada de tu controlador `beforeFilter()`, para poder aprovechar las funciones de cambio automático de vista de `RequestHandlerComponent` si lo están usando.

Enviando arhivos

Cake\Http\Response::**withFile**(*string $path*, *array $options = []*)

Hay ocasiones en las que desea enviar archivos como respuesta a sus solicitudes. Puedes lograrlo usando *Cake\Http\Response::withFile()*:

```
public function sendFile($id)
{
    $file = $this->Attachments->getFile($id);
    $response = $this->response->withFile($file['path']);
    // Devuelve la respuesta para evitar que el controlador intente representar una␣
↪vista.
    return $response;
}
```

Como se muestra en el ejemplo anterior, debe pasar la ruta del archivo al método. CakePHP enviará un encabezado de tipo de contenido adecuado si es un tipo de archivo conocido que figura en *Cake\Http\Response::$_mimeTypes*.

Puede agregar nuevos tipos antes de llamar a *Cake\Http\Response::withFile()* usando el método *Cake\Http\Response::withType()* .

Si lo desea, también puede forzar la descarga de un archivo en lugar de mostrarlo en el navegador especificando las opciones:

```
$response = $this->response->withFile(
    $file['path'],
    ['download' => true, 'name' => 'foo']
);
```

Las opciones admitidas son:

name
: El nombre le permite especificar un nombre de archivo alternativo para enviarlo al usuario.

download
: Un valor booleano que indica si los encabezados deben configurarse para forzar la descarga.

Enviar una cadena como archivo

Puedes responder con un archivo que no existe en el disco, como un pdf o un ics generado sobre la marcha a partir de una cadena:

```
public function sendIcs()
{
    $icsString = $this->Calendars->generateIcs();
    $response = $this->response;

    // Inyectar contenido de cadena en el cuerpo de la respuesta
    $response = $response->withStringBody($icsString);

    $response = $response->withType('ics');

    // Opcionalmente forzar la descarga de archivos
    $response = $response->withDownload('filename_for_download.ics');

    // Devuelve un objeto de respuesta para evitar que el controlador intente␣
↪representar una vista.
    return $response;
}
```

Configuración de encabezados

Cake\Http\Response::**withHeader**(*$header*, *$value*)

La configuración de los encabezados se realiza con el método *Cake\Http\Response::withHeader()*. Como todos los métodos de la interfaz PSR-7, este método devuelve una instancia *nueva* con el nuevo encabezado:

```
// Agregar/reemplazar un encabezado
$response = $response->withHeader('X-Extra', 'My header');

// Establecer múltiples encabezados
```

(continué en la próxima página)

(proviene de la página anterior)

```
$response = $response->withHeader('X-Extra', 'My header')
    ->withHeader('Location', 'http://example.com');

// Agregar un valor a un encabezado existente
$response = $response->withAddedHeader('Set-Cookie', 'remember_me=1');
```

Los encabezados no se envían cuando se configuran. En cambio, se retienen hasta que `Cake\Http\Server` emite la respuesta.

Ahora puede utilizar el método conveniente `Cake\Http\Response::withLocation()` para configurar u obtener directamente el encabezado de ubicación de redireccionamiento.

Configurando el cuerpo

`Cake\Http\Response::withStringBody`(*$string*)

Para establecer una cadena como cuerpo de respuesta, haga lo siguiente:

> // Coloca una cadena en el cuerpo. $response = $response->withStringBody("My Body");
>
> // Si quieres una respuesta json $response = $response->withType("application/json")->withStringBody(json_encode(["Foo" => "bar"]));

`Cake\Http\Response::withBody`(*$body*)

Para configurar el cuerpo de la respuesta, use el método `withBody()`, que es proporcionado por `Laminas\Diactoros\MessageTrait`:

```
$response = $response->withBody($stream);
```

Asegúrese de que `$stream` sea un objeto `Psr\Http\Message\StreamInterface`. Vea a continuación cómo crear un nuevo stream.

También puedes transmitir respuestas desde archivos usando `Laminas\Diactoros\Stream` streams:

```
// Para transmitir desde un archivo
use Laminas\Diactoros\Stream;

$stream = new Stream('/path/to/file', 'rb');
$response = $response->withBody($stream);
```

También puedes transmitir respuestas desde una devolución de llamada usando `CallbackStream`. Esto es útil cuando tiene recursos como imágenes, archivos CSV o PDF que necesita transmitir al cliente:

```
// Transmisión desde una devolución de llamada
use Cake\Http\CallbackStream;

// Crea una imagen.
$img = imagecreate(100, 100);
// ...

$stream = new CallbackStream(function () use ($img) {
    imagepng($img);
});
$response = $response->withBody($stream);
```

Configuración del juego de caracteres

Cake\Http\Response::**withCharset**(*$charset*)

Establece el juego de caracteres que se utilizará en la respuesta:

```
$this->response = $this->response->withCharset('UTF-8');
```

Interactuar con el almacenamiento en caché del navegador

Cake\Http\Response::**withDisabledCache**()

A veces es necesario obligar a los navegadores a no almacenar en caché los resultados de una acción del controlador. *Cake\Http\Response::withDisabledCache()* está destinado precisamente a eso:

```
public function index()
{
    // Deshabilitar el almacenamiento en caché
    $this->response = $this->response->withDisabledCache();
}
```

Advertencia: Deshabilitar el almacenamiento en caché de dominios SSL al intentar enviar archivos a Internet Explorer puede generar errores.

Cake\Http\Response::**withCache**(*$since*, *$time = '+1 day'*)

También puede decirles a los clientes que desea que almacenen en caché las respuestas. Usando *Cake\Http\Response::withCache()*:

```
public function index()
{
    // Habilitar el almacenamiento en caché
    $this->response = $this->response->withCache('-1 minute', '+5 days');
}
```

Lo anterior les indicaría a los clientes que guarden en caché la respuesta resultante durante 5 días, con la esperanza de acelerar la experiencia de sus visitantes. El método `withCache()` establece el valor `Última modificación` en el primer argumento. El encabezado `Expires` y la directiva `max-age` se establecen en función del segundo parámetro. La directiva «pública» de Cache-Control también está configurada.

Ajuste fino de la caché HTTP

Una de las mejores y más sencillas formas de acelerar su aplicación es utilizar la caché HTTP. Según este modelo de almacenamiento en caché, solo debe ayudar a los clientes a decidir si deben usar una copia en caché de la respuesta configurando algunos encabezados, como la hora de modificación y la etiqueta de entidad de respuesta.

En lugar de obligarlo a codificar la lógica para el almacenamiento en caché y para invalidarla (actualizarla) una vez que los datos han cambiado, HTTP utiliza dos modelos, caducidad y validación, que generalmente son mucho más simples de usar.

Además de usar *Cake\Http\Response::withCache()*, también puedes usar muchos otros métodos para ajustar los encabezados de caché HTTP para aprovechar el almacenamiento en caché del navegador o del proxy inverso.

El encabezado de control de caché

Cake\Http\Response::**withSharable**(*$public*, *$time = null*)

Utilizado bajo el modelo de vencimiento, este encabezado contiene múltiples indicadores que pueden cambiar la forma en que los navegadores o servidores proxy usan el contenido almacenado en caché. Un encabezado `Cache-Control` puede verse así:

```
Cache-Control: private, max-age=3600, must-revalidate
```

La clase `Response` le ayuda a configurar este encabezado con algunos métodos de utilidad que producirán un encabezado `Cache-Control` final válido. El primero es el método `withSharable()`, que indica si una respuesta debe considerarse compartible entre diferentes usuarios o clientes. Este método en realidad controla la parte «pública» o «privada» de este encabezado. Establecer una respuesta como privada indica que toda o parte de ella está destinada a un solo usuario. Para aprovechar las cachés compartidas, la directiva de control debe configurarse como pública.

El segundo parámetro de este método se utiliza para especificar una `max-age` para el caché, que es el número de segundos después de los cuales la respuesta ya no se considera nueva:

```
public function view()
{
    // ...
    // Configure Cache-Control como público durante 3600 segundos
    $this->response = $this->response->withSharable(true, 3600);
}

public function my_data()
{
    // ...
    // Configure Cache-Control como privado durante 3600 segundos
    $this->response = $this->response->withSharable(false, 3600);
}
```

`Response` expone métodos separados para configurar cada una de las directivas en el encabezado `Cache-Control`.

El encabezado de vencimiento

Cake\Http\Response::**withExpires**(*$time*)

Puede configurar el encabezado `Expires` en una fecha y hora después de la cual la respuesta ya no se considera nueva. Este encabezado se puede configurar usando el método `withExpires()`:

```
public function view()
{
    $this->response = $this->response->withExpires('+5 days');
}
```

Este método también acepta una instancia `DateTime` o cualquier cadena que pueda ser analizada por la clase `DateTime`.

El encabezado de la etiqueta electrónica

`Cake\Http\Response::withEtag(`*$tag*, *$weak = false*`)`

La validación de caché en HTTP se usa a menudo cuando el contenido cambia constantemente y le pide a la aplicación que solo genere el contenido de la respuesta si el caché ya no está actualizado. Bajo este modelo, el cliente continúa almacenando páginas en el caché, pero pregunta a la aplicación cada vez si el recurso ha cambiado, en lugar de usarlo directamente. Esto se usa comúnmente con recursos estáticos como imágenes y otros activos.

El método `withEtag()` (llamado etiqueta de entidad) es una cadena que identifica de forma única el recurso solicitado, como lo hace una suma de comprobación para un archivo, para determinar si coincide con un recurso almacenado en caché.

Para aprovechar este encabezado, debe llamar al método `checkNotModified()` manualmente o incluir *Checking HTTP Cache* en su controlador:

```
public function index()
{
    $articles = $this->Articles->find('all')->all();

    // Suma de comprobación simple del contenido del artículo.
    // Debería utilizar una implementación más eficiente en una aplicación del mundo␣
↪real.
    $checksum = md5(json_encode($articles));

    $response = $this->response->withEtag($checksum);
    if ($response->checkNotModified($this->request)) {
        return $response;
    }

    $this->response = $response;
    // ...
}
```

Nota: La mayoría de los usuarios de proxy probablemente deberían considerar usar el encabezado de última modificación en lugar de Etags por razones de rendimiento y compatibilidad.

El último encabezado modificado

`Cake\Http\Response::withModified(`*$time*`)`

Además, bajo el modelo de validación de caché HTTP, puede configurar el encabezado `Last-Modified` para indicar la fecha y hora en la que se modificó el recurso por última vez. Configurar este encabezado ayuda a CakePHP a decirle a los clientes de almacenamiento en caché si la respuesta se modificó o no según su caché.

Para aprovechar este encabezado, debe llamar al método `checkNotModified()` manualmente o incluir *Checking HTTP Cache* en su controlador:

```
public function view()
{
    $article = $this->Articles->find()->first();
    $response = $this->response->withModified($article->modified);
    if ($response->checkNotModified($this->request)) {
```

(continué en la próxima página)

(proviene de la página anterior)

```
        return $response;
    }
    $this->response;
    // ...
}
```

El encabezado variable

Cake\Http\Response::**withVary**(*$header*)

En algunos casos, es posible que desee publicar contenido diferente utilizando la misma URL. Este suele ser el caso si tiene una página multilingüe o responde con HTML diferente según el navegador. En tales circunstancias, puede utilizar el encabezado `Vary`:

```
$response = $this->response->withVary('User-Agent');
$response = $this->response->withVary('Accept-Encoding', 'User-Agent');
$response = $this->response->withVary('Accept-Language');
```

Envío de respuestas no modificadas

Cake\Http\Response::**checkNotModified**(*Request $request*)

Compara los encabezados de la caché del objeto de solicitud con el encabezado de la caché de la respuesta y determina sitodavía se puede considerar nuevo. Si es así, elimina el contenido de la respuesta y envía el encabezado *304 Not Modified*:

```
// En una acción del controlador.
if ($this->response->checkNotModified($this->request)) {
    return $this->response;
}
```

Configuración de cookies

Las cookies se pueden agregar a la respuesta usando una matriz o un objeto *Cake\Http\Cookie\Cookie*:

```
use Cake\Http\Cookie\Cookie;
use DateTime;

// Agregar una cookie
$this->response = $this->response->withCookie(Cookie::create(
    'remember_me',
    'yes',
    // Todas las claves son opcionales.
    [
        'expires' => new DateTime('+1 year'),
        'path' => '',
        'domain' => '',
        'secure' => false,
        'httponly' => false,
```

(continué en la próxima página)

(proviene de la página anterior)

```
        'samesite' => null // O una de las constantes CookieInterface::SAMESITE_*
    ]
));
```

Consulte la sección *Creando cookies* para saber cómo utilizar el objeto cookie. Puede utilizar `withExpiredCookie()` para enviar una cookie caducada en la respuesta. Esto hará que el navegador elimine su cookie local:

```
$this->response = $this->response->withExpiredCookie(new Cookie('remember_me'));
```

Configuración de encabezados de solicitud de origen cruzado (CORS)

El método `cors()` se utiliza para definir Control de acceso HTTP[93] encabezados relacionados con una interfaz fluida:

```
$this->response = $this->response->cors($this->request)
    ->allowOrigin(['*.cakephp.org'])
    ->allowMethods(['GET', 'POST'])
    ->allowHeaders(['X-CSRF-Token'])
    ->allowCredentials()
    ->exposeHeaders(['Link'])
    ->maxAge(300)
    ->build();
```

Los encabezados relacionados con CORS solo se aplicarán a la respuesta si se cumplen los siguientes criterios:

1. The request has an `Origin` header.
2. The request's `Origin` value matches one of the allowed Origin values.

Truco: CakePHP no tiene middleware CORS incorporado porque manejar solicitudes CORS es muy específico de la aplicación. Le recomendamos que cree su propio `CORSMiddleware` si lo necesita y ajuste el objeto de respuesta como desee.

Errores comunes con respuestas inmutables

Los objetos de respuesta ofrecen varios métodos que tratan las respuestas como objetos inmutables. Los objetos inmutables ayudan a prevenir efectos secundarios accidentales difíciles de rastrear y reducen los errores causados por llamadas a métodos causadas por la refactorización que cambia el orden. Si bien ofrecen una serie de beneficios, es posible que sea necesario algo de tiempo para acostumbrarse a los objetos inmutables. Cualquier método que comience con `with` opera en la respuesta de forma inmutable y **siempre** devolverá una **nueva** instancia. Olvidar conservar la instancia modificada es el error más frecuente que cometen las personas cuando trabajan con objetos inmutables:

$this->response->withHeader("X-CakePHP", "yes!");

En el código anterior, a la respuesta le faltará el encabezado `X-CakePHP`, ya que el valor de retorno del método `withHeader()` no se retuvo. Para corregir el código anterior escribirías:

```
$this->response = $this->response->withHeader('X-CakePHP', 'yes!');
```

[93] https://developer.mozilla.org/es/docs/Web/HTTP/CORS

Colección de Cookies

class Cake\Http\Cookie**CookieCollection**

Se puede acceder a los objetos `CookieCollection` desde los objetos de solicitud y respuesta. Le permiten interactuar con grupos de cookies utilizando patrones inmutables, que permiten preservar la inmutabilidad de la solicitud y la respuesta.

Creando cookies

class Cake\Http\Cookie**Cookie**

Los objetos `Cookie` se pueden definir a través de objetos constructores o utilizando la interfaz fluida que sigue patrones inmutables:

```
use Cake\Http\Cookie\Cookie;

// Todos los argumentos en el constructor.
$cookie = new Cookie(
    'remember_me', // nombre
    1, // valor
    new DateTime('+1 year'), // tiempo de vencimiento, si corresponde
    '/', // ruta, si corresponde
    'example.com', // dominio, si corresponde
    false, // ¿Solo seguro?
    true // ¿Solo http?
);

// Usando los métodos constructores
$cookie = (new Cookie('remember_me'))
    ->withValue('1')
    ->withExpiry(new DateTime('+1 year'))
    ->withPath('/')
    ->withDomain('example.com')
    ->withSecure(false)
    ->withHttpOnly(true);
```

Una vez que haya creado una cookie, puede agregarla a una `CookieCollection` nueva o existente:

```
use Cake\Http\Cookie\CookieCollection;

// Crear una nueva colección
$cookies = new CookieCollection([$cookie]);

// Agregar a una colección existente
$cookies = $cookies->add($cookie);

// Eliminar una cookie por nombre
$cookies = $cookies->remove('remember_me');
```

Nota: Recuerde que las colecciones son inmutables y agregar o eliminar cookies de una colección crea un *nuevo* objeto

de colección.

Se pueden agregar objetos cookie a las respuestas:

```
// Agregar una cookie
$response = $this->response->withCookie($cookie);

// Reemplazar toda la colección de cookies
$response = $this->response->withCookieCollection($cookies);
```

Las cookies configuradas para las respuestas se pueden cifrar utilizando encrypted-cookie-middleware.

Leyendo Cookies

Una vez que tenga una instancia `CookieCollection`, podrá acceder a las cookies que contiene:

```
// Comprobar si existe una cookie
$cookies->has('remember_me');

// Obtener el número de cookies de la colección.
count($cookies);

// Obtener una instancia de cookie. Lanzará un error si no se encuentra la cookie.
$cookie = $cookies->get('remember_me');

// Obtener una cookie o nulo
$cookie = $cookies->remember_me;

// Comprobar si existe una cookie
$exists = isset($cookies->remember_me)
```

Una vez que tenga un objeto `Cookie`, puede interactuar con su estado y modificarlo. Tenga en cuenta que las cookies son inmutables, por lo que deberá actualizar la colección si modifica una cookie:

```
// Obtener el valor
$value = $cookie->getValue()

// Acceder a datos dentro de un valor JSON
$id = $cookie->read('User.id');

// Comprobar estado
$cookie->isHttpOnly();
$cookie->isSecure();
```

CAPÍTULO 9

Controladores

class Cake\Controller**Controller**

Los controladores son la "C" en MVC. Después de aplicar el enrutamiento y que el controlador ha sido encontrado, la acción de tu controlador es llamado. Tu controlador debe manejar la interpretación de los datos de la solicitud, asegurándose de que se llamen a los modelos correctos y se muestre la respuesta o vista correcta. Los controladores se pueden considerar como una capa intermedia entre el Modelo y la Vista. Quieres mantener tus controladores delgados, y tus modelos gruesos. Esto te ayudará a reutilizar tu código y lo hará más fácil de probar.

Comúnmente, un controlador se usa para administrar la lógica en torno a un solo modelo. Por ejemplo, si estuvieras construyendo un sitio online para una panadería, podrías tener un RecipesController que gestiona tus recetas y un IngredientsController que gestiona tus ingredientes. Sin embargo, es posible hacer que los controladores trabajen con más de un modelo. En CakePHP, un controlador es nombrado a raíz del modelo que maneja.

Los controladores de tu aplicación extienden de la clase `AppController`, que a su vez extiende de la clase principal *`Controller`*. La clase `AppController` puede ser definida en **src/Controller/AppController.php** y debería contener los métodos que se comparten entre todos los controladores de tu aplicación.

Los controladores proveen una serie de métodos que manejan las peticiones. Estos son llamadas *acciones*. Por defecto, cada método público en un controlador es una acción, y es accesible mediante una URL. Una acción es responsable de interpretar la petición y crear la respuesta. Por lo general, las respuestas son de la forma de una vista renderizada, pero también, hay otras maneras de crear respuestas.

El App Controller

Como se indicó en la introducción, la clase `AppController` es clase padre de todos los controladores de tu aplicación. `AppController` extiende de la clase *`Cake\Controller\Controller`* que está incluida en CakePHP. `AppController` se define en **src/Controller/AppController.php** como se muestra a continuación:

```
namespace App\Controller;

use Cake\Controller\Controller;

class AppController extends Controller
{
}
```

Los atributos y métodos del controlador creados en tu `AppController` van a estar disponibles en todos los controladores que extiendan de este. Los componentes (que aprenderás más adelante) son mejor usados para código que se encuentra en muchos (pero no necesariamente en todos) los componentes.

Puedes usar tu `AppController` para cargar componentes que van a ser utilizados en cada controlador de tu aplicación. CakePHP proporciona un método `initialize()` que es llamado al final del constructor de un controlador para este tipo de uso:

```
namespace App\Controller;

use Cake\Controller\Controller;

class AppController extends Controller
{
    public function initialize(): void
    {
        // Always enable the FormProtection component.
        $this->loadComponent('FormProtection');
    }
}
```

Flujo de solicitud

Cuando se realiza una solicitud a una aplicación CakePHP, las clases CakePHP `Cake\Routing\Router` y `Cake\Routing\Dispatcher` usan *Conectando Rutas* para encontrar y crear la instancia correcta del controlador. Los datos de la solicitud son encapsulados en un objeto de solicitud. CakePHP pone toda la información importante de la solicitud en la propiedad `$this->request`. Consulta la sección sobre *Solicitud (Request)* para obtener más información sobre el objeto de solicitud de CakePHP.

Acciones del controlador

Las acciones del controlador son las responsables de convertir los parámetros de la solicitud en una respuesta para el navegador/usuario que realiza la petición. CakePHP usa convenciones para automatizar este proceso y eliminar algunos códigos repetitivos que de otro modo se necesitaría escribir.

Por convención, CakePHP renderiza una vista con una versión en infinitivo del nombre de la acción. Volviendo a nuestro ejemplo de la panadería online, nuestro RecipesController podría contener las acciones `view()`, `share()`, y `search()`. El controlador sería encontrado en **src/Controller/RecipesController.php** y contiene:

```
// src/Controller/RecipesController.php

class RecipesController extends AppController
{
    public function view($id)
    {
        // La lógica de la acción va aquí.
    }

    public function share($customerId, $recipeId)
    {
        // La lógica de la acción va aquí.
    }

    public function search($query)
    {
        // La lógica de la acción va aquí.
    }
}
```

Las plantillas para estas acciones serían **templates/Recipes/view.php**, **templates/Recipes/share.php**, y **templates/Recipes/search.php**. El nombre convencional para un archivo de vista es con minúsculas y con el nombre de la acción entre guiones bajos.

Las acciones de los controladores por lo general usan `Controller::set()` para crear un contexto que `View` usa para renderizar la capa de vista. Debido a las convenciones que CakePHP usa, no necesitas crear y renderizar la vista manualmente. En su lugar, una vez que se ha completado la acción del controlador, CakePHP se encargará de renderizar y entregar la vista.

Si por algún motivo deseas omitir el comportamiento predeterminado, puedes retornar un objeto *Cake\Http\Response* de la acción con la respuesta creada.

Para que puedas usar un controlador de manera efectiva en tu aplicación, cubriremos algunos de los atributos y métodos principales proporcionados por los controladores de CakePHP.

Interactuando con vistas

Los controladores interactúan con las vistas de muchas maneras. Primero, los controladores son capaces de pasar información a las vistas, usando `Controller::set()`. También puedes decidir qué clase de vista usar, y qué archivo de vista debería ser renderizado desde el controlador.

Configuración de variables de vista

`Cake\Controller\Controller::set`(*string $var*, *mixed $value*)

El método `Controller::set()` es la manera principal de mandar información desde el controlador a la vista. Una vez que hayas utilizado `Controller::set()`, la variable puede ser accedida en tu vista:

```
// Primero pasas las información desde el controlador:

$this->set('color', 'rosa');

// Después, en la vista, puede utilizar la información:
?>

Has seleccionado cubierta <?= h($color) ?> para la tarta.
```

El método `Controller::set()` también toma un array asociativo como su primer parámetro. A menudo, esto puede ser una forma rápida de asignar un conjunto de información a la vista:

```
$data = [
    'color' => 'pink',
    'type' => 'sugar',
    'base_price' => 23.95,
];

// Hace $color, $type, y $base_price
// disponible para la vista:

$this->set($data);
```

Ten en cuenta que las variables de la vista se comparten entre todas las partes renderizadas por tu vista. Estarán disponibles en todas las partes de la vista: la plantilla y todos los elementos dentro de estas dos.

Configuración de las opciones de la vista

Si deseas personalizar la clase vista, las rutas de diseño/plantillas, ayudantes o el tema que se usarán para renderizar la vista, puede usar el método `viewBuilder()` para obtener un constructor. Este constructor se puede utilizar para definir propiedades de la vista antes de crearlas:

```
$this->viewBuilder()
    ->addHelper('MyCustom')
    ->setTheme('Modern')
    ->setClassName('Modern.Admin');
```

Lo anterior muestra cómo puedes cargar ayudantes personalizados, configurar el tema y usar una clase vista personalizada.

Renderizando una vista

Cake\Controller\Controller::**render**(*string $view*, *string $layout*)

El método `Controller::render()` es llamado automáticamente al final de cada solicitud de la acción del controlador. Este método realiza toda la lógica de la vista (usando la información que has enviado usando el método `Controller::set()`), coloca la vista dentro de su `View::$layout`, y lo devuelve al usuario final.

El archivo de vista por defecto utilizado para el renderizado es definido por convención. Si la acción `search()` de RecipesController es solicitada, el archivo vista en **templates/Recipes/search.php** será renderizado:

```
namespace App\Controller;

class RecipesController extends AppController
{
// ...
    public function search()
    {
        // Renderiza la vista en templates/Recipes/search.php
        return $this->render();
    }
// ...
}
```

Aunque CakePHP va a llamarlo automáticamente después de cada acción de lógica (a menos que llames a `$this->disableAutoRender()`), puedes usarlo para especificar un archivo de vista alternativo especificando el nombre de este como primer argumento del método `Controller::render()`.

Si `$view` empieza con "/", se asume que es una vista o un archivo relacionado con la carpeta **templates**. Esto permite el renderizado directo de elementos, muy útil en llamadas AJAX:

```
// Renderiza el elemento en templates/element/ajaxreturn.php
$this->render('/element/ajaxreturn');
```

El segundo parámetro `$layout` de `Controller::render()` te permita especificar la estructura con la que la vista es renderizada.

Renderizando una plantilla específica

En tu controlador, puede que quieras renderizar una vista diferente a la que es convencional. Puedes hacer esto llamando a `Controller::render()` directamente. Una vez que hayas llamado a `Controller::render()`, CakePHP no tratará de re-renderizar la vista:

```
namespace App\Controller;

class PostsController extends AppController
{
    public function my_action()
    {
        $this->render('custom_file');
    }
}
```

Esto renderizará **templates/Posts/custom_file.php** en vez de **templates/Posts/my_action.php**.

También puedes renderizar vistas dentro de plugins usando la siguiente sintaxis: `$this->render('PluginName.PluginController/custom_file')`. Por ejemplo:

```
namespace App\Controller;

class PostsController extends AppController
{
    public function myAction()
    {
        $this->render('Users.UserDetails/custom_file');
    }
}
```

Esto renderizará **plugins/Users/templates/UserDetails/custom_file.php**

Negociación del tipo de contenido

Cake\Controller\Controller::**viewClasses**()

Los controladores pueden definir una lista de clases de vistas que soportan. Después de que la acción del controlador este completa, CakePHP usará la lista de vista para realizar negociación del tipo de contenido. Esto permite a tu aplicación rehusar la misma acción del controlador para renderizar una vista HTML o renderizar una respuesta JSON o XML. Para definir la lista de clases de vista que soporta un controlador se utiliza el método `viewClasses()`:

```
namespace App\Controller;

use Cake\View\JsonView;
use Cake\View\XmlView;

class PostsController extends AppController
{
    public function viewClasses(): array
    {
        return [JsonView::class, XmlView::class];
    }
}
```

La clase `View` de la aplicación se usa automáticamente como respaldo cuando no se puede seleccionar otra vista en función del encabezado de la petición `Accept` o de la extensión del enrutamiento. Si tu aplicación sólo soporta tipos de contenido para una acción específica, puedes definir esa lógica dentro de `viewClasses()`:

```
public function viewClasses(): array
{
    if ($this->request->getParam('action') === 'export') {
        // Usa una vista CSV personalizada para exportación de datos
        return [CsvView::class];
    }

    return [JsonView::class];
}
```

Si dentro de las acciones de tu controlador necesitas procesar la petición o cargar datos de forma diferente dependiendo del tipo de contenido puedes usar *Comprobación de las condiciones de la solicitud*:

```
// En la acción de un controlador

// Carga información adicional cuando se preparan respuestas JSON
if ($this->request->is('json')) {
    $query->contain('Authors');
}
```

También puedes definir las clases View soportadas por tu controlador usando el método `addViewClasses()` que unirá la vista proporcionada con aquellas que están actualmente en la propiedad `viewClasses`.

Nota: Las clases de vista deben implementar el método estático `contentType()` para participar en las negociaciones del tipo de contenido.

Negociación de tipo de contenido alternativos

Si ninguna vista puede coincidir con las preferencias del tipo de contenido de la petición, CakePHP usará la clase base `View`. Si deseas solicitar una negociación del tipo de contenido, puedes usar `NegotiationRequiredView` que setea un código de estatus 406:

```
public function viewClasses(): array
{
    // Requiere aceptar la negociación del encabezado o devuelve una respuesta 406.
    return [JsonView::class, NegotiationRequiredView::class];
}
```

Puede usar el valor del tipo de contenido `TYPE_MATCH_ALL` para crear tu lógica de vista alternativa:

```
namespace App\View;

use Cake\View\View;

class CustomFallbackView extends View
{
    public static function contentType(): string
    {
        return static::TYPE_MATCH_ALL;
    }

}
```

Es importante recordar que las vistas coincidentes se aplican sólo *después* de intentar la negociación del tipo de contenido.

Redirigiendo a otras páginas

Cake\Controller\Controller::**redirect**(*string|array $url*, *integer $status*)

El método `redirect()` agrega un encabezado `Location` y establece un código de estado de una respuesta y la devuelve. Deberías devolver la respuesta creada por `redirect()` para que CakePHP envíe la redirección en vez de completar la acción del controlador y renderizar la vista.

Puedes redigir usando los valores de un array ordenado:

```
return $this->redirect([
    'controller' => 'Orders',
    'action' => 'confirm',
    $order->id,
    '?' => [
        'product' => 'pizza',
        'quantity' => 5
    ],
    '#' => 'top'
]);
```

O usando una URL relativa o absoluta:

```
return $this->redirect('/orders/confirm');

return $this->redirect('http://www.example.com');
```

O la referencia de la página:

```
return $this->redirect($this->referer());
```

Usando el segundo parámetro puede definir un código de estatus para tu redirección:

```
// Haz un 301 (movido permanentemente)
return $this->redirect('/order/confirm', 301);

// Haz un 303 (Ver otro)
return $this->redirect('/order/confirm', 303);
```

Reenviando a un acción en el mismo controlador

Cake\Controller\Controller::**setAction**(*$action*, *$args...*)

Si necesitas reenviar la acción actual a una acción diferente en el *mismo* controlador, puedes usar `Controller::setAction()` para actualizar el objeto de la solicitud, modifica la plantilla de vista que será renderizada y reenvía la ejecución a la nombrada acción:

```
// Desde una acción de eliminación, puedes renderizar a lista de página
// actualizada.
$this->setAction('index');
```

Cargando modelos adicionales

Cake\Controller\Controller::**fetchModel**(*string $alias*, *array $config = []*)

La función `fetchModel()` es útil cuando se necesita cargar un modelo o tabla del ORM que no es la predeterminada por el controlador. Modelos obtenidos de ésta manera no serán seteados como propiedades en el controlador:

```
// Obtiene un modelo de ElasticSearch
$articles = $this->fetchModel('Articles', 'Elastic');

// Obtiene un modelo de webservice
$github = $this->fetchModel('GitHub', 'Webservice');
```

Nuevo en la versión 4.5.0.

Cake\Controller\Controller::**fetchTable**(*string $alias*, *array $config = []*)

La función `fetchTable()` es útil cuando se necesita usar una tabla del ORM que no es la predeterminada por el controlador:

```
// En un método del controlador.
$recentArticles = $this->fetchTable('Articles')->find('all',
        limit: 5,
        order: 'Articles.created DESC'
    )
    ->all();
```

Nuevo en la versión 4.3.0: `Controller::fetchTable()` fue añadido. Antes de 4.3 necesitas usar `Controller::loadModel()`.

Paginación de un modelo

Cake\Controller\Controller::**paginate**()

Este método se utiliza para paginar los resultados obtenidos por tus modelos. Puedes especificar tamaño de páginas, condiciones de búsqueda del modelo y más. Ve a la sección pagination para más detalles sobre como usar `paginate()`.

El atributo `$paginate` te da una manera de personalizar cómo `paginate()` se comporta:

```
class ArticlesController extends AppController
{
    protected array $paginate = [
        'Articles' => [
            'conditions' => ['published' => 1],
        ],
    ];
}
```

Configuración de componentes para cargar

Cake\Controller\Controller::**loadComponent**(*$name*, *$config = []*)

En el método `initialize()` de tu controlador, puedes definir cualquier componente que deseas cargar, y cualquier dato de configuración para ellos:

```
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Flash');
    $this->loadComponent('Comments', Configure::read('Comments'));
}
```

Callbacks del ciclo de vida de la petición

Los controladores de CakePHP activan varios eventos/callbacks que puedes usar para insertar lógica alrededor del ciclo de vida de la solicitud.

Lista de eventos

- `Controller.initialize`
- `Controller.startup`
- `Controller.beforeRedirect`
- `Controller.beforeRender`
- `Controller.shutdown`

Métodos de callback del controlador

Por defecto, los siguientes métodos de callback están conectados a eventos relacionados si los métodos son implementados por tus controladores.

Cake\Controller\Controller::**beforeFilter**(*EventInterface $event*)

Llamado durante el evento `Controller.initialize` que ocurre antes de cada acción en el controlador. Es un lugar útil para comprobar si hay una sesión activa o inspeccionar los permisos del usuario.

Nota: El método beforeFilter() será llamado por acciones faltantes.

Devolver una respuesta del método `beforeFilter` no evitará que otros oyentes del mismo evento sean llamados. Debes explícitamente parar el evento.

Cake\Controller\Controller::**beforeRender**(*EventInterface $event*)

Llamado durante el evento `Controller.beforeRender` que ocurre después de la lógica de acción del controlador, pero antes de que la vista sea renderizada. Este callback no se usa con frecuencia, pero puede ser necesaria si estas llamando *`render()`* de forma manual antes del final de una acción dada.

Cake\Controller\Controller::**afterFilter**(*EventInterface $event*)

Llamado durante el evento `Controller.shutdown` que se desencadena después de cada acción del controlador, y después de que se complete el renderizado. Este es el último método del controlador para ejecutar.

Además de las devoluciones de llamada del ciclo de vida del controlador, *Componentes* también proporciona un conjunto similar de devoluciones de llamada.

Recuerda llamar a los callbacks de `AppController` dentro de los callbacks del controlador hijo para mejores resultados:

```
//use Cake\Event\EventInterface;
public function beforeFilter(EventInterface $event)
{
    parent::beforeFilter($event);
}
```

Middleware del controlador

Cake\Controller\Controller::**middleware**(*$middleware*, *array $options = []*)

Middleware puede ser definido globalmente, en un ámbito de enrutamiento o dentro de un controlador. Para definir el middleware para un controlador en específico usa el método `middleware()` de tu método `initialize()` del controlador:

```
public function initialize(): void
{
    parent::initialize();

    $this->middleware(function ($request, $handler) {
        // Haz la lógica del middleware.

        // Verifica que devuelves una respuesta o llamas a handle()
        return $handler->handle($request);
    });
}
```

El middleware definido por un controlador será llamado **antes** `beforeFilter()` y se llamarán a los métodos de acción.

Más sobre controladores

El controlador de Páginas

El esqueleto oficial de CakePHP incluye un controlador por defecto **PagesController.php**. Este es un controlador simple y opcional que se usa para servir contenido estático. La página home que ves después de la instalación es generada usando este controlador y el archivo de vista **templates/Pages/home.php**. Si se crea el archivo de vista **templates/Pages/about_us.php** se podrá acceder a este usando la URL **http://example.com/pages/about_us**. Sientete libre de modificar el controlador para que cumpla con tus necesidades.

Cuando se cocina una app usando Composer el controlador es creado en la carpeta **src/Controller/**.

Componentes

Los componentes son paquetes de lógica que se comparten entre los controladores. CakePHP viene un con fantástico conjunto de componentes básicos que puedes usar para ayudar en varias tareas comunes. También puedes crear tus propios componentes. Si te encuentras queriendo copiar y pegar cosas entre componentes, deberías considerar crear tu propio componente que contenga la funcionalidad. Crear componentes mantiene el código del controlador limpio y te permite rehusar código entre los diferentes controladores.

Para más información sobre componentes incluidos en CakePHP, consulte el capítulo de cada componente:

FlashComponent

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[94] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

FormProtection

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[95] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Checking HTTP Cache

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[96] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

---

[94] https://github.com/cakephp/docs
[95] https://github.com/cakephp/docs
[96] https://github.com/cakephp/docs

Configurando componentes

Muchos de los componentes principales requieren configuración. Un ejemplo sería *FormProtection*. La configuración para estos componentes, y para los componentes en general, es usualmente hecho a través `loadComponent()` en el método `initialize()` del controlador o a través del array `$components`:

```
class PostsController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('FormProtection', [
            'unlockedActions' => ['index'],
        ]);
        $this->loadComponent('Flash');
    }
}
```

También puedes configurar los componentes en tiempo de ejecución usando el método `setConfig()`. A veces, esto es hecho en el método `beforeFilter()` del controlador. Lo anterior podría ser también expresado como:

```
public function beforeFilter(EventInterface $event)
{
    $this->FormProtection->setConfig('unlockedActions', ['index']);
}
```

Al igual que los helpers, los componentes implementan los métodos `getConfig()` y `setConfig()` para leer y escribir los datos de configuración:

```
// Lee los datos de configuración.
$this->FormProtection->getConfig('unlockedActions');

// Escribe los datos de configuración
$this->Flash->setConfig('key', 'myFlash');
```

Al igual que con los helpers, los componentes fusionarán automáticamente su propiedad `$_defaultConfig` con la configuración del controlador para crear la propiedad `$_config` que es accesible con `getConfig()` y `setConfig()`.

Componentes de alias

Una configuración común para usar es la opción `className`, que te permite utilizar componentes de alias. Esta característica es útil cuando quieres reemplazar `$this->Flash` u otra referencia común de componente con una implementación personalizada:

```
// src/Controller/PostsController.php
class PostsController extends AppController
{
    public function initialize(): void
    {
        $this->loadComponent('Flash', [
            'className' => 'MyFlash'
        ]);
    }
```

(continué en la próxima página)

(proviene de la página anterior)

```
}

// src/Controller/Component/MyFlashComponent.php
use Cake\Controller\Component\FlashComponent;

class MyFlashComponent extends FlashComponent
{
    // Agrega tu código para sobreescribir el FlashComponent principal
}
```

Lo de arriba haría *alias* `MyFlashComponent` a `$this->Flash` en tus controladores.

Nota: El alias de un componente reemplaza esa instancia en cualquier lugar donde se use ese componente, incluso dentro de otros componentes.

Carga de componentes sobre la marcha

Es posible que no necesites todos tus componentes disponibles en cada acción del controlador. En situaciones como estas, puedes cargar un componente en tiempo de ejecución usando el método `loadComponent()` en tu controlador:

```
// En una acción del controlador
$this->loadComponent('OneTimer');
$time = $this->OneTimer->getTime();
```

Nota: Ten en cuenta que los componentes cargados sobre la marcha no perderán devoluciones de llamadas. Si te basas en que las devoluciones de llamada `beforeFilter` o `startup` serán llamadas, necesitarás llamarlas manualmente dependiendo de cuándo cargas tu componente.

Uso de componentes

Una vez que hayas incluido algunos componentes a tu controlador, usarlos es bastante simple. Cada componente que uses se exponen como una propiedad en tu controlador. Si cargaste el `Cake\Controller\Component\FlashComponent` en tu controlador, puedes acceder a él de esta forma:

```
class PostsController extends AppController
{
    public function initialize(): void
    {
        parent::initialize();
        $this->loadComponent('Flash');
    }

    public function delete()
    {
        if ($this->Post->delete($this->request->getData('Post.id')) {
            $this->Flash->success('Post deleted.');
```

(continué en la próxima página)

(proviene de la página anterior)

```
            return $this->redirect(['action' => 'index']);
        }
    }
}
```

Nota: Dado que tanto los modelos como los componentes se agregan a los controladores como propiedades, comparten el mismo "espacio de nombres". Asegúrate de no dar a un componente y un modelo el mismo nombre.

Advertencia: Los métodos de un componente **no** tienen acceso a /development/dependency-injection como lo tienen los controladores. Usa una clase de servicio dentro de las acciones de tu controlador en lugar de en el componente si necesitas ésta funcionalidad.

Creando un componente

Supongamos que nuestra aplicación necesita realizar una operación matemática compleja en muchas partes diferentes de la aplicación. Podríamos crear un componente para albergar esta lógica compartida para su uso en muchos controladores diferentes.

El primer paso es crear un nuevo archivo de componente y clase. Crea el archivo en **src/Controller/Component/MathComponent.php**. La estructura básica para el componente debería verse algo como esto:

```
namespace App\Controller\Component;

use Cake\Controller\Component;

class MathComponent extends Component
{
    public function doComplexOperation($amount1, $amount2)
    {
        return $amount1 + $amount2;
    }
}
```

Nota: Todos los componentes deben extender de `Cake\Controller\Component`. De lo contrario, se disparará una excepción.

Incluyendo tu componente en tus controladores

Una vez que nuestro componente está terminado, podemos usarlo en los controladores de la aplicación cargándolo durante el método `initialize()` del controlador. Una vez cargado, el controlador recibirá un nuevo atributo con el nombre del componente, a través del cual podemos acceder a una instancia del mismo:

```
// En un controlador
// Haz que el nuevo componente esté disponible en $this->Math,
// así como el estándar $this->Flash
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Math');
    $this->loadComponent('Flash');
}
```

Al incluir componentes en un controlador, también puedes declarar un conjunto de parámetros que se pasarán al constructor del componente. Estos parámetros pueden ser manejados por el componente:

```
// En tu controlador.
public function initialize(): void
{
    parent::initialize();
    $this->loadComponent('Math', [
        'precision' => 2,
        'randomGenerator' => 'srand'
    ]);
    $this->loadComponent('Flash');
}
```

Lo anterior pasaría el array que contiene precision y randomGenerator a `MathComponent::initialize()` en el parámetro `$config`.

Usando otros componentes en tu componente

A veces, uno de tus componentes necesita usar otro componente. Puedes cargar otros componentes agregándolos a la propiedad *$components*:

```
// src/Controller/Component/CustomComponent.php
namespace App\Controller\Component;

use Cake\Controller\Component;

class CustomComponent extends Component
{
    // El otro componente que tu componente usa
    protected $components = ['Existing'];

    // Ejecuta cualquier otra configuración adicional para tu componente.
    public function initialize(array $config): void
    {
        $this->Existing->foo();
```

(continué en la próxima página)

(proviene de la página anterior)

```
    }

    public function bar()
    {
        // ...
    }
}

// src/Controller/Component/ExistingComponent.php
namespace App\Controller\Component;

use Cake\Controller\Component;

class ExistingComponent extends Component
{
    public function foo()
    {
        // ...
    }
}
```

Nota: A diferencia de un componente incluido en un controlador, no se activarán devoluciones de llamada en el componente de un componente.

Accediendo al controlador de un componente

Desde dentro de un componente, puedes acceder al controlador actual a través del registro:

```
$controller = $this->getController();
```

Devoluciones de llamadas de componentes

Los componentes también ofrecen algunas devoluciones de llamadas de ciclo de vida de las solicitudes que les permiten aumentar el ciclo de solicitud.

beforeFilter(*EventInterface $event*)

Es llamado antes que el método beforeFilter() del controlador, pero *después* del método initialize() del controlador.

startup(*EventInterface $event*)

Es llamado después del método beforeFilter() del controlador, pero antes de que el controlador ejecute el «handler» de la acción actual

beforeRender(*EventInterface $event*)

Es llamado después de que el controlador ejecute la lógica de la acción solicitada, pero antes de que el controlador renderize las vistas y el diseño.

afterFilter(*EventInterface $event*)

Es llamado durante el evento `Controller.shutdown`, antes de enviar la salida al navegador.

beforeRedirect(*EventInterface $event*, *$url*, *Response $response*)

Es llamado cuando el método de redirección del controlador es llamado pero antes de cualquier otra acción. Si este método devuelve `false` el controlador no continuará en redirigir la petición. Los parámetros $url y $response permiten modificar e inspeccionar la ubicación o cualquier otro encabezado en la respuesta.

Usando redireccionamiento en eventos de componentes

Para redirigir desde dentro de un método de devolución de llamada de un componente, puedes usar lo siguiente:

```
public function beforeFilter(EventInterface $event)
{
    $event->stopPropagation();

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

Al detener el evento, le haces saber a CakePHP que no quieres ninguna otra devolución de llamada de componente para ejecutar, y que el controlador no debe manejar la acción más lejos. A partir de 4.1.0 puedes generar una `RedirectException` para señalar una redirección:

```
use Cake\Http\Exception\RedirectException;
use Cake\Routing\Router;

public function beforeFilter(EventInterface $event)
{
    throw new RedirectException(Router::url('/'))
}
```

Generar una excepción detendrá todos los demás detectores de eventos y creará una nueva respuesta que no conserva ni hereda ninguno de los encabezados de la respuesta actual. Al generar una `RedirectException` puedes incluir encabezados adicionales:

```
throw new RedirectException(Router::url('/'), 302, [
    'Header-Key' => 'value',
]);
```

Nuevo en la versión 4.1.0.

CAPÍTULO 10

Vistas

class Cake\View**View**

Las vistas son la **V** en MVC. Son responsables de generar la salida específica requerida para la solicitud. A menudo, esto se hace en forma de HTML, XML o JSON, pero también es responsabilidad de la Capa de Vistas transmitir archivos y crear PDFs que los usuarios puedan descargar.

CakePHP viene con algunas clases de Vista incorporadas para manejar los escenarios de renderizado más comunes:

- Para crear servicios web XML o JSON, puedes usar los *Vistas JSON y XML*.
- Para servir archivos protegidos o archivos generados dinámicamente, puedes usar *Enviando arhivos*.
- Para crear vistas con varios temas, puedes usar *Themes*.

La Vista de la Aplicación

`AppView` es la clase de Vista predeterminada de tu aplicación. `AppView` en sí misma extiende la clase `Cake\View\View` incluida en CakePHP y está definida en **src/View/AppView.php** de la siguiente manera:

```
<?php
namespace App\View;

use Cake\View\View;

class AppView extends View
{
}
```

Puedes usar tu `AppView` para cargar ayudantes que se utilizarán en cada vista renderizada en tu aplicación. CakePHP proporciona un método `initialize()` que se invoca al final del constructor de una Vista para este tipo de uso:

```php
<?php
namespace App\View;

use Cake\View\View;

class AppView extends View
{
    public function initialize(): void
    {
        // Always enable the MyUtils Helper
        $this->addHelper('MyUtils');
    }
}
```

Plantillas de Vista

La capa de vista de CakePHP es la forma en que te comunicas con tus usuarios. La mayor parte del tiempo, tus vistas estarán renderizando documentos HTML/XHTML para los navegadores, pero también podrías necesitar responder a una aplicación remota a través de JSON o generar un archivo CSV para un usuario.

Los archivos de plantilla de CakePHP son archivos PHP regulares y utilizan la sintaxis PHP alternativa[97] para las estructuras de control y la salida. Estos archivos contienen la lógica necesaria para preparar los datos recibidos del controlador en un formato de presentación que está listo para tu audiencia.

Alternativas de impresion

Puedes imprimir o mostrar una variable en tu plantilla de la siguiente manera:

```php
<?php echo $variable; ?>
```

Utilizando soporte para etiquetas cortas:

```php
<?= $variable ?>
```

Estructuras de Control Alternativas

Las estructuras de control, como `if`, `for`, `foreach`, `switch` y `while`, pueden escribirse en un formato simplificado. Observa que no hay llaves. En su lugar, la llave de cierre para el `foreach` se reemplaza con `endforeach`. Cada una de las estructuras de control mencionadas anteriormente tiene una sintaxis de cierre similar: `endif`, `endfor`, `endforeach` y `endwhile`. También observa que en lugar de usar un `punto y coma` después de cada estructura (excepto la última), hay dos puntos `:`.

El siguiente es un ejemplo utilizando `foreach`:

```php
<ul>
<?php foreach ($todo as $item): ?>
  <li><?= $item ?></li>
<?php endforeach; ?>
</ul>
```

[97] https://php.net/manual/en/control-structures.alternative-syntax.php

Otro ejemplo, usando if/elseif/else. Observa los dos puntos:

```
<?php if ($username === 'sally'): ?>
   <h3>Hi Sally</h3>
<?php elseif ($username === 'joe'): ?>
   <h3>Hi Joe</h3>
<?php else: ?>
   <h3>Hi unknown user</h3>
<?php endif; ?>
```

Si prefieres utilizar un lenguaje de plantillas como Twig[98], una subclase de View facilitará la conexión entre tu lenguaje de plantillas y CakePHP.

Los archivos de plantilla se almacenan en **templates/**, en una carpeta nombrada según el controlador que utiliza los archivos y el nombre de la acción a la que corresponde. Por ejemplo, el archivo de vista para la acción `view()` del controlador `Products`, normalmente se encontraría en **templates/Products/view.php**.

La capa de vista en CakePHP puede estar compuesta por varias partes diferentes. Cada parte tiene usos distintos y se cubrirán en este capítulo:

- **templates**: Las plantillas son la parte de la página que es única para la acción que se está ejecutando. Constituyen el contenido principal de la respuesta de tu aplicación.
- **elements**: pequeños fragmentos de código de vista reutilizables. Por lo general, los elementos se renderizan dentro de las vistas.
- **layouts**: archivos de plantilla que contienen código de presentación que envuelve muchas interfaces en tu aplicación. La mayoría de las vistas se renderizan dentro de un diseño.
- **helpers**: estas clases encapsulan la lógica de la vista que se necesita en muchos lugares en la capa de vista. Entre otras cosas, los helpers en CakePHP pueden ayudarte a construir formularios, funcionalidades AJAX, paginar datos de modelos o servir feeds RSS.
- **cells**: estas clases proporcionan características similares a un controlador para crear componentes de interfaz de usuario autosuficientes. Consulta la *View Cells* documentación para obtener más información.

Variables de Vista

Cualquier variable que establezcas en tu controlador con `set()` estará disponible tanto en la vista como en el diseño que tu acción renderiza. Además, cualquier variable establecida también estará disponible en cualquier elemento. Si necesitas pasar variables adicionales desde la vista al diseño, puedes llamar a `set()` en la plantilla de vista o utilizar *Bloques de Vista*.

Debes recordar **siempre** escapar cualquier dato del usuario antes de mostrarlo, ya que CakePHP no escapa automáticamente la salida. Puedes escapar el contenido del usuario con la función `h()`::»

<?= h($user->bio); ?>

[98] https://twig.symfony.com

Estableciendo Variables de Vista

`Cake\View\View::set`(*string $var*, *mixed $value*)

Las vistas tienen un método `set()` que es análogo al `set()` que se encuentra en los objetos del Controlador. Utilizar `set()` desde tu archivo de vista agregará las variables al diseño y a los elementos que se renderizarán más adelante. Consulta *Configuración de variables de vista* para obtener más información sobre el uso de `set()`.

En tu archivo de vista puedes hacer:

```
$this->set('activeMenuButton', 'posts');
```

Entonces, en tu diseño, la variable `$activeMenuButton` estará disponible y contendrá el valor "posts".»

Extendiendo Vistas

La extensión de vistas te permite envolver una vista dentro de otra. Combinar esto con los *bloques de vista* te brinda una forma poderosa de mantener tus vistas *DRY* (Don't Repeat Yourself o No te repitas). Por ejemplo, tu aplicación tiene una barra lateral que necesita cambiar según la vista específica que se está renderizando. Al extender un archivo de vista común, puedes evitar repetir el marcado común para tu barra lateral y solo definir las partes que cambian:

```
<!-- templates/Common/view.php -->
<h1><?= h($this->fetch('title')) ?></h1>
<?= $this->fetch('content') ?>

<div class="actions">
    <h3>Acciones relacionadas</h3>
    <ul>
    <?= $this->fetch('sidebar') ?>
    </ul>
</div>
```

El archivo de vista anterior podría ser utilizado como una vista principal. Espera que la vista que lo extiende definirá los bloques `sidebar` y `title`. El bloque `content` es un bloque especial que CakePHP crea. Contendrá todo el contenido no capturado de la vista que lo extiende. Suponiendo que nuestro archivo de vista tiene una variable `$post` con los datos de nuestra publicación, la vista podría verse así:

```
<!-- templates/Posts/view.php -->
<?php
$this->extend('/Common/view');

$this->assign('title', $post->title);

$this->start('sidebar');
?>
<li>
<?php
echo $this->Html->link('edit', [
    'action' => 'edit',
    $post->id,
]);
?>
</li>
<?php $this->end(); ?>
```

(continué en la próxima página)

(proviene de la página anterior)

```
// The remaining content will be available as the 'content' block
// In the parent view.
<?= h($post->body) ?>
```

La vista de la publicación anterior muestra cómo puedes extender una vista y llenar un conjunto de bloques. Cualquier contenido que no esté ya definido en un bloque será capturado y colocado en un bloque especial llamado `content`. Cuando una vista contiene una llamada a `extend()`, la ejecución continúa hasta el final del archivo de vista actual. Una vez que se completa, se renderizará la vista extendida. Llamar a `extend()` más de una vez en un archivo de vista anulará la vista principal que se procesará a continuación:

```
$this->extend('/Common/view');
$this->extend('/Common/index');
```

Lo anterior hará que **/Common/index.php** se renderice como la vista principal para la vista actual.

Puedes anidar vistas extendidas tantas veces como sea necesario. Cada vista puede extender otra vista si así lo deseas. Cada vista principal obtendrá el contenido de la vista anterior como el bloque `content`.

Nota: Debes evitar usar `content` como nombre de bloque en tu aplicación. CakePHP lo utiliza para el contenido no capturado en vistas extendidas.

Extendiendo Layouts

Al igual que las vistas, los layouts también pueden ser extendidos. Al igual que con las vistas, se utiliza `extend()` para extender los layouts. Las extensiones de layouts pueden actualizar o reemplazar bloques y actualizar o reemplazar el contenido renderizado por el layout secundario. Por ejemplo, si quisiéramos envolver un bloque con un marcado adicional, podríamos hacer lo siguiente:

```
// Nuestro layout extiende el layout de la aplicación.
$this->extend('application');
$this->prepend('content', '<main class="nosidebar">');
$this->append('content', '</main>');

// Generar más marcado.

// Recuerda imprimir el contenido del layout anterior.
echo $this->fetch('content');
```

Uso de Bloques de Vista

Los bloques de vista proporcionan una API flexible que te permite definir ranuras o bloques en tus vistas/diseños que se definirán en otro lugar. Por ejemplo, los bloques son ideales para implementar cosas como barras laterales o regiones para cargar activos en la parte inferior o superior del diseño. Los bloques se pueden definir de dos maneras: como un bloque capturador o mediante asignación directa. Los métodos `start()`, `append()`, `prepend()`, `assign()`, `fetch()`, y `end()` te permiten trabajar con bloques capturadores:

```
// Crear el bloque de la barra lateral.
$this->start('sidebar');
echo $this->element('sidebar/recent_topics');
echo $this->element('sidebar/recent_comments');
$this->end();

// Anexar contenido al bloque de la barra lateral más adelante.
$this->start('sidebar');
echo $this->fetch('sidebar');
echo $this->element('sidebar/popular_topics');
$this->end();
```

También puedes añadir contenido a un bloque usando `append()`:

```
$this->append('sidebar');
echo $this->element('sidebar/popular_topics');
$this->end();

// Lo mismo que el ejemplo anterior.
$this->append('sidebar', $this->element('sidebar/popular_topics'));
```

Si necesitas borrar u sobrescribir un bloque, hay un par de alternativas. El método `reset()` eliminará o sobrescribirá un bloque en cualquier momento. El método `assign()` con una cadena de contenido vacía también se puede utilizar para borrar el bloque especificado.:

```
// Limpiar el contenido anterior del bloque de la barra lateral.
$this->reset('sidebar');

// Asignar una cadena vacía también borrará el bloque de la barra lateral.
$this->assign('sidebar', '');
```

Asignar el contenido de un bloque a menudo es útil cuando deseas convertir una variable de vista en un bloque. Por ejemplo, es posible que desees usar un bloque para el título de la página y, a veces, asignar el título como una variable de vista en el controlador:

```
// En el archivo de vista o diseño, arriba de $this->fetch('title')
$this->assign('title', $title);
```

El método `prepend()` te permite agregar contenido al principio de un bloque existente:

```
// Agregar al principio de la barra lateral
$this->prepend('sidebar', 'este contenido va en la parte superior de la barra lateral');
```

Mostrar Bloques

Puedes mostrar bloques usando el método `fetch()`. `fetch()` mostrará un bloque, devolviendo "" si un bloque no existe:

```
<?= $this->fetch('sidebar') ?>
```

También puedes usar `fetch()` para mostrar condicionalmente contenido que debería rodear a un bloque si existe. Esto es útil en diseños o vistas extendidas donde quieres mostrar condicionalmente encabezados u otro marcado:

```
// En templates/layout/default.php
<?php if ($this->fetch('menu')): ?>
<div class="menu">
    <h3>Opciones de Menú</h3>
    <?= $this->fetch('menu') ?>
</div>
<?php endif; ?>
```

También puedes proporcionar un valor predeterminado para un bloque si no existe. Esto te permite agregar contenido de marcador de posición cuando un bloque no existe. Puedes proporcionar un valor predeterminado usando el segundo argumento:

```
<div class="carrito-de-compras">
    <h3>Tu Carrito</h3>
    <?= $this->fetch('carrito', 'Tu carrito está vacío') ?>
</div>
```

Usando Bloques para Archivos de Scripts y CSS

El *HtmlHelper* se integra con bloques de vista, y sus métodos *script()*, *css()*, y *meta()* actualizan un bloque con el mismo nombre cuando se usan con la opción *block = true*:

```
<?php
// En tu archivo de vista
$this->Html->script('carousel', ['block' => true]);
$this->Html->css('carousel', ['block' => true]);
?>

// En tu archivo de diseño.
<!DOCTYPE html>
<html lang="en">
    <head>
    <title><?= h($this->fetch('title')) ?></title>
    <?= $this->fetch('script') ?>
    <?= $this->fetch('css') ?>
    </head>
    // El resto del diseño sigue
```

El *Cake\View\Helper\HtmlHelper* también te permite controlar a qué bloque van los scripts y el CSS:

```
// En tu vista
$this->Html->script('carousel', ['block' => 'scriptBottom']);

// En tu diseño
<?= $this->fetch('scriptBottom') ?>
```

Layouts

Un layout contiene código de presentación que envuelve una vista. Todo lo que desees ver en todas tus vistas debe colocarse en un layout.

El layout predeterminado de CakePHP se encuentra en **templates/layout/default.php**. Si deseas cambiar el aspecto general de tu aplicación, este es el lugar correcto para comenzar, porque el código de vista renderizado por el controlador se coloca dentro del layout predeterminado cuando se renderiza la página.

Otros archivos de layout deben colocarse en **templates/layout**. Cuando creas un layout, necesitas decirle a CakePHP dónde colocar la salida de tus vistas. Para hacerlo, asegúrate de que tu layout incluye un lugar para `$this->fetch('content')`. Aquí tienes un ejemplo de cómo podría verse un layout predeterminado:

```
<!DOCTYPE html>
<html lang="es">
<head>
<title><?= h($this->fetch('title')) ?></title>
<link rel="shortcut icon" href="favicon.ico" type="image/x-icon">
<!-- Incluye archivos externos y scripts aquí (Consulta el ayudante HTML para obtener␣
↪más información.) -->
<?php
echo $this->fetch('meta');
echo $this->fetch('css');
echo $this->fetch('script');
?>
</head>
<body>

<!-- Si deseas que aparezca algún tipo de menú en
todas tus vistas, inclúyelo aquí -->
<div id="header">
    <div id="menu">...</div>
</div>

<!-- Aquí es donde quiero que se muestren mis vistas -->
<?= $this->fetch('content') ?>

<!-- Agrega un pie de página a cada página mostrada -->
<div id="footer">...</div>

</body>
</html>
```

Los bloques `script`, `css` y `meta` contienen cualquier contenido definido en las vistas usando el ayudante HTML incorporado. Útil para incluir archivos JavaScript y CSS desde vistas.

Nota: Al usar `HtmlHelper::css()` o `HtmlHelper::script()` en archivos de plantilla, especifica `'block' => true` para colocar la fuente HTML en un bloque con el mismo nombre. (Consulta la API para obtener más detalles sobre el uso).

El bloque `content` contiene el contenido de la vista renderizada.

Puedes establecer el contenido del bloque `title` desde dentro de tu archivo de vista:

```
$this->assign('title', 'Ver Usuarios Activos');
```

Los valores vacíos para el bloque `title` se reemplazarán automáticamente con una representación de la ruta de la plantilla actual, como `'Admin/Artículos'`.

Puedes crear tantos layouts como desees: solo colócalos en el directorio **templates/layout**, y alterna entre ellos dentro de tus acciones del controlador usando la propiedad `$layout` del controlador o la vista:

```
// Desde un controlador
public function vista()
{
    // Establecer el diseño.
    $this->viewBuilder()->setLayout('admin');
}

// Desde un archivo de vista
$this->layout = 'registrado';
```

Por ejemplo, si una sección de mi sitio incluyera un espacio para un banner publicitario más pequeño, podría crear un nuevo layout con el espacio publicitario más pequeño y especificarlo como el layout para todas las acciones de los controladores usando algo como:

```
namespace App\Controller;

class UsuariosController extends AppController
{
    public function verActivos()
    {
        $this->set('title', 'Ver Usuarios Activos');
        $this->viewBuilder()->setLayout('default_small_ad');
    }

    public function verImagen()
    {
        $this->viewBuilder()->setLayout('imagen');

        // Mostrar imagen del usuario
    }
}
```

Además de un layout predeterminado, la aplicación de esqueleto oficial de CakePHP también tiene un layout “ajax”. El layout Ajax es útil para crear respuestas AJAX; es un diseño vacío. (La mayoría de las llamadas AJAX solo requieren un poco de marcado como respuesta, en lugar de una interfaz completamente renderizada).

La aplicación de esqueleto también tiene un diseño predeterminado para ayudar a generar RSS.

Usando Layouts desde Plugins

Si deseas usar un layout que existe en un plugin, puedes utilizar la *sintaxis de plugin*. Por ejemplo, para usar el diseño de contacto del plugin Contacts:

```
namespace App\Controller;

class UsersController extends AppController
{
    public function verActivos()
    {
        $this->viewBuilder()->setLayout('Contacts.contact');
    }
}
```

Elementos

`Cake\View\View::element`(*string $elementPath*, *array $data*, *array $options = []*)

Muchas aplicaciones tienen pequeños bloques de código de presentación que deben repetirse de una página a otra, a veces en diferentes lugares en el diseño. CakePHP puede ayudarte a repetir partes de tu sitio web que necesitan ser reutilizadas. Estas partes reutilizables se llaman Elementos. Los anuncios, los cuadros de ayuda, los controles de navegación, los menús adicionales, los formularios de inicio de sesión y las llamadas a la acción a menudo se implementan en CakePHP como elementos. Un elemento es básicamente una mini-vista que se puede incluir en otras vistas, en diseños e incluso dentro de otros elementos. Los elementos se pueden usar para hacer que una vista sea más legible, colocando la representación de elementos repetidos en su propio archivo. También te pueden ayudar a reutilizar fragmentos de contenido en tu aplicación.

Los elementos se encuentran en la carpeta **templates/element/** y tienen la extensión de archivo .php. Se generan utilizando el método element de la vista:

```
echo $this->element('helpbox');
```

Pasar Variables a un Elemento

Puedes pasar datos a un elemento a través del segundo argumento del elemento:

```
echo $this->element('helpbox', [
    'helptext' => 'Oh, este texto es muy útil.',
]);
```

Dentro del archivo del elemento, todas las variables pasadas están disponibles como miembros del array de parámetros (de la misma manera que `Controller::set()` en el controlador funciona con los archivos de plantilla). En el ejemplo anterior, el archivo **templates/element/helpbox.php** puede usar la variable `$helptext`:

```
// Dentro de templates/element/helpbox.php
echo $helptext; // Muestra `Oh, este texto es muy útil.`
```

Ten en cuenta que en esas variables de vista se fusionan con las variables de vista desde la vista en sí misma. Entonces, todas las variables de vista establecidas usando `Controller::set()` en el controlador y `View::set()` en la vista en sí también están disponibles dentro del elemento.

El método `View::element()` también admite opciones para el elemento. Las opciones admitidas son "cache" y "callbacks". Un ejemplo:

```
echo $this->element('helpbox', [
        'helptext' => "Esto se pasa al elemento como $helptext",
        'foobar' => "Esto se pasa al elemento como $foobar",
    ],
    [
        // utiliza la configuración de caché `long_view`
        'cache' => 'long_view',
        // establece en true para que se llame a before/afterRender para el elemento
        'callbacks' => true,
    ]
);
```

El almacenamiento en caché del elemento se facilita a través de la clase `Cache`. Puedes configurar elementos para que se almacenen en cualquier configuración de caché que hayas establecido. Esto te brinda una gran cantidad de flexibilidad para decidir dónde y por cuánto tiempo se almacenan los elementos. Para almacenar en caché diferentes versiones del mismo elemento en una aplicación, proporciona un valor de clave de caché único usando el siguiente formato:

```
$this->element('helpbox', [], [
        'cache' => ['config' => 'short', 'key' => 'valor único'],
    ]
);
```

Si necesitas más lógica en tu elemento, como datos dinámicos de una fuente de datos, considera usar un View Cell en lugar de un elemento. Descubre más *sobre View Cells*.

Almacenamiento en Caché de Elementos

Puedes aprovechar el almacenamiento en caché de CakePHP si proporcionas un parámetro de caché. Si se establece en `true`, almacenará en caché el elemento en la configuración de Caché "default". De lo contrario, puedes establecer qué configuración de caché se debe utilizar. Consulta *Caching* para obtener más información sobre cómo configurar `Cache`. Un ejemplo simple de cómo almacenar en caché un elemento sería:

```
echo $this->element('helpbox', [], ['cache' => true]);
```

Si renderizas el mismo elemento más de una vez en una vista y tienes el almacenamiento en caché habilitado, asegúrate de establecer el parámetro "key" con un nombre diferente cada vez. Esto evitará que cada llamada sucesiva sobrescriba el resultado en caché de la llamada anterior a `element()`. Por ejemplo:

```
echo $this->element(
    'helpbox',
    ['var' => $var],
    ['cache' => ['key' => 'primer_uso', 'config' => 'view_long']]
);

echo $this->element(
    'helpbox',
    ['var' => $otraVar],
    ['cache' => ['key' => 'segundo_uso', 'config' => 'view_long']]
);
```

Lo anterior asegurará que ambos resultados del elemento se almacenen en caché por separado. Si deseas que todos los elementos en caché utilicen la misma configuración de caché, puedes evitar algo de repetición configurando `View::$elementCache` con la configuración de caché que deseas utilizar. CakePHP utilizará esta configuración cuando no se proporcione ninguna.

Solicitando Elementos desde un Plugin

Si estás usando un plugin y deseas utilizar elementos desde dentro del plugin, simplemente usa la conocida *sintaxis de plugin*. Si la vista se está renderizando para un controlador/acción del plugin, el nombre del plugin se agregará automáticamente a todos los elementos utilizados, a menos que haya otro nombre de plugin presente. Si el elemento no existe en el plugin, buscará en la carpeta principal de la APLICACIÓN (APP):

```
echo $this->element('Contacts.helpbox');
```

Si tu vista es parte de un plugin, puedes omitir el nombre del plugin. Por ejemplo, si estás en el `ContactsController` del plugin Contacts, lo siguiente:

```
echo $this->element('helpbox');
// y
echo $this->element('Contacts.helpbox');
```

son equivalentes y darán como resultado que se renderice el mismo elemento.

Para elementos dentro de una subcarpeta de un plugin (por ejemplo, **plugins/Contacts/Template/element/sidebar/helpbox.php**), usa el siguiente formato:

```
echo $this->element('Contacts.sidebar/helpbox');
```

Prefijo de Enrutamiento y Elementos

Si tienes un prefijo de enrutamiento configurado, la resolución de la ruta del Elemento puede cambiar a una ubicación con prefijo, como sucede con los Diseños (Layouts) y la Vista de acción. Supongamos que tienes configurado un prefijo «Admin» y llamas a:

```
echo $this->element('mi_elemento');
```

El elemento se buscará primero en **templates/Admin/element/**. Si dicho archivo no existe, se buscará en la ubicación predeterminada.

Almacenamiento en Caché de Secciones de tu Vista

`Cake\View\View::cache`(*callable $block*, *array $options = []*)

A veces, generar una sección de la salida de tu vista puede ser costoso debido a *View Cells* renderizados u operaciones de ayuda costosas. Para ayudar a que tu aplicación se ejecute más rápido, CakePHP proporciona una forma de almacenar en caché secciones de vista:

```
// Suponiendo algunas variables locales
echo $this->cache(function () use ($usuario, $articulo) {
    echo $this->cell('PerfilUsuario', [$usuario]);
    echo $this->cell('ArticuloCompleto', [$articulo]);
}, ['key' => 'mi_clave_de_vista']);
```

Por defecto, el contenido de la vista almacenado en caché se guardará en la configuración de caché `View::$elementCache`, pero puedes usar la opción `config` para cambiar esto.

Eventos de Vista

Al igual que los Controladores, la vista activa varios eventos o llamadas de retorno (callbacks) que puedes utilizar para insertar lógica alrededor del ciclo de vida de renderización:

Lista de Eventos

- `View.beforeRender`
- `View.beforeRenderFile`
- `View.afterRenderFile`
- `View.afterRender`
- `View.beforeLayout`
- `View.afterLayout`

Puedes adjuntar *escuchadores de eventos de la aplicación* a estos eventos o utilizar *Llamadas de Retorno de Ayudantes (Helper Callbacks)*.

Creando tus Propias Clases de Vista

Puede que necesites crear clases de vista personalizadas para habilitar nuevos tipos de vistas de datos o agregar lógica de renderización de vista personalizada adicional a tu aplicación. Como la mayoría de componentes de CakePHP, las clases de vista tienen algunas convenciones:

- Los archivos de clases de vista deben colocarse en **src/View**. Por ejemplo: **src/View/PdfView.php**
- Las clases de vista deben tener el sufijo `View`. Por ejemplo: `PdfView`.
- Al referenciar nombres de clases de vista, deberías omitir el sufijo `View`. Por ejemplo: `$this->viewBuilder()->setClassName('Pdf');`.

También querrás extender `View` para asegurar que las cosas funcionen correctamente:

```
// En src/View/PdfView.php
namespace App\View;

use Cake\View\View;

class PdfView extends View
{
    public function render($view = null, $layout = null)
    {
        // Lógica personalizada aquí.
    }
}
```

Reemplazar el método render te permite tener control total sobre cómo se renderiza tu contenido.

Más acerca de Vistas

View Cells

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[99] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Themes

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[100] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Vistas JSON y XML

La integración de `JsonView` y `XmlView` con las funcionalidades de *Negociación del tipo de contenido* de CakePHP y te permite crear respuestas JSON y XML.

Éstas clases View son usadas de forma mas común junto con `CakeControllerController::viewClasses()`.

Hay dos formas de generar vistas de datos. La primera es mediante el uso de la opción `serialize` y la segunda es mediante la creación de archivos de plantilla normales.

Definiendo clases View pata Negociar

En tu `AppController` o en un controlador individual puedes implementar la función `viewClasses()` y proporcionarle todas las clases View que quieras soportar:

```
use Cake\View\JsonView;
use Cake\View\XmlView;

public function viewClasses(): array
{
    return [JsonView::class, XmlView::class];
}
```

[99] https://github.com/cakephp/docs
[100] https://github.com/cakephp/docs

Opcionalmente, puede habilitar las extensiones json y/o xml con *file-extensions*. Esto le permitirá acceder a `JSON`, `XML` o cualquier otra vista de formato especial utilizando una URL personalizada que termine con el nombre del tipo de respuesta como una extensión de archivo como `http://example.com/articles.json`.

De forma predeterminada, cuando no se habilitan las *file-extensions*, se utiliza la solicitud, seleccionando el encabezado `Accept`, seleccionando qué tipo de formato se debe presentar al usuario. Un ejemplo de formato `Accept` que se utiliza para representar respuestas `JSON` es `application/json`.

Uso de vistas de datos con la clave Serialize

La opción `serialize` indica qué variable(s) de vista se deben serializar cuando se utiliza una vista de datos. Esto le permite omitir la definición de archivos de plantilla para las acciones del controlador si no necesita realizar ningún formateo personalizado antes de que los datos se conviertan en json/xml.

Si necesita realizar algún formateo o manipulación de las variables de vista antes de generar la respuesta, debe usar archivos de plantilla. El valor de `serialize` puede ser un string o un array de variables de vista para serializar:

```
namespace App\Controller;

use Cake\View\JsonView;

class ArticlesController extends AppController
{
    public function viewClasses(): array
    {
        return [JsonView::class];
    }

    public function index()
    {
        // Asigna las variables a la vista
        $this->set('articles', $this->paginate());
        // Especifica las variables de vista que JsonView deberá serializar
        $this->viewBuilder()->setOption('serialize', 'articles');
    }
}
```

También puede definir `serialize` como un array de variables de vista para combinar:

```
namespace App\Controller;

use Cake\View\JsonView;

class ArticlesController extends AppController
{
    public function viewClasses(): array
    {
        return [JsonView::class];
    }

    public function index()
    {
        // Código que crear las variables $articles y $comments
```

(continué en la próxima página)

(proviene de la página anterior)

```
        // Asigna las variables a la vista
        $this->set(compact('articles', 'comments'));

        // Specify which view vars JsonView should serialize.
        $this->viewBuilder()->setOption('serialize', ['articles', 'comments']);
    }
}
```

La definición de `serialize` como un array ha añadido la ventaja de anexar automáticamente un elemento `<response>` de nivel superior cuando se utiliza *XmlView*. Si utiliza un valor de string para `serialize` y XmlView, asegúrese de que la variable de vista tiene un único elemento de nivel superior. Sin un solo elemento de nivel superior, el Xml no podrá generarse.

Uso de una vista de datos con archivos de plantilla

Debe usar archivos de plantilla si necesita manipular el contenido de la vista antes de crear el resultado final. Por ejemplo, si tuviéramos artículos con un campo que contuviera HTML generado, probablemente querríamos omitirlo de una respuesta JSON. Esta es una situación en la que un archivo de vista sería útil:

```
// Controller code
class ArticlesController extends AppController
{
    public function index()
    {
        $articles = $this->paginate('Articles');
        $this->set(compact('articles'));
    }
}

// View code - templates/Articles/json/index.php
foreach ($articles as $article) {
    unset($article->generated_html);
}
echo json_encode(compact('articles'));
```

Puede hacer manipulaciones más complejas o usar ayudantes para formatear también. Las clases de vista de datos no admiten diseños. Asumen que el archivo de vista generará el contenido serializado.

Creación de vistas XML

class XmlView

De forma predeterminada, cuando se utiliza `serialize`, XmlView ajustará las variables de vista serializadas con un nodo `<response>`. Puede establecer un nombre personalizado para este nodo mediante la opción `rootNode`.

La clase XmlView admite la opción `xmlOptions` que le permite personalizar las opciones utilizadas para generar XML, por ejemplo, `tags` frente `attributes`.

Un ejemplo de uso de `XmlView` sería generar un sitemap.xml[101]. Este tipo de documento requiere que cambie `rootNode` y establezca atributos. Los atributos se definen mediante el prefijo `@`:

[101] https://www.sitemaps.org/protocol.html

```
public function sitemap()
{
    $pages = $this->Pages->find()->all();
    $urls = [];
    foreach ($pages as $page) {
        $urls[] = [
            'loc' => Router::url(['controller' => 'Pages', 'action' => 'view', $page->
↪slug, '_full' => true]),
            'lastmod' => $page->modified->format('Y-m-d'),
            'changefreq' => 'daily',
            'priority' => '0.5',
        ];
    }

    // Define a custom root node in the generated document.
    $this->viewBuilder()
        ->setOption('rootNode', 'urlset')
        ->setOption('serialize', ['@xmlns', 'url']);
    $this->set([
        // Define an attribute on the root node.
        '@xmlns' => 'http://www.sitemaps.org/schemas/sitemap/0.9',
        'url' => $urls,
    ]);
}
```

Creación de vistas JSON

class JsonView

La clase JsonView admite la opción `jsonOptions` que permite personalizar la máscara de bits utilizada para generar JSON. Consulte la documentación de json_encode[102] para conocer los valores válidos de esta opción.

Por ejemplo, para serializar la salida de errores de validación de las entidades CakePHP en una forma coherente de JSON:

```
// In your controller's action when saving failed
$this->set('errors', $articles->errors());
$this->viewBuilder()
    ->setOption('serialize', ['errors'])
    ->setOption('jsonOptions', JSON_FORCE_OBJECT);
```

[102] https://php.net/json_encode

Respuestas JSONP

Al utilizar `JsonView`, puede utilizar la variable de vista especial `jsonp` para habilitar la devolución de una respuesta JSONP. Si se establece en `true` la clase de vista comprueba si se establece el parámetro de string de consulta denominado «callback» y, de ser así, envuelve la respuesta json en el nombre de función proporcionado. Si desea utilizar un nombre de parámetro de string de consulta personalizado en lugar de «callback», establezca `jsonp` al nombre requerido en lugar de `true`..

Eligiendo una clase View

Aunque puedes usar la función `viewClasses` la mayoría de las veces, si quieres un control total sobre la selección de la clase de vista, puedes elegir directamente la clase:

```
// src/Controller/VideosController.php
namespace App\Controller;

use App\Controller\AppController;
use Cake\Http\Exception\NotFoundException;

class VideosController extends AppController
{
    public function export($format = '')
    {
        $format = strtolower($format);

        // Format to view mapping
        $formats = [
          'xml' => 'Xml',
          'json' => 'Json',
        ];

        // Error on unknown type
        if (!isset($formats[$format])) {
            throw new NotFoundException(__('Unknown format.'));
        }

        // Set Out Format View
        $this->viewBuilder()->setClassName($formats[$format]);

        // Get data
        $videos = $this->Videos->find('latest')->all();

        // Set Data View
        $this->set(compact('videos'));
        $this->viewBuilder()->setOption('serialize', ['videos']);

        // Set Force Download
        return $this->response->withDownload('report-' . date('YmdHis') . '.' . $format);
    }
}
```

Helpers

Los Helpers son clases similares a componentes para la capa de presentación de tu aplicación. Contienen lógica de presentación que se comparte entre muchas vistas, elementos o diseños. Este capítulo te mostrará cómo configurar los helpers, cómo cargarlos y usarlos, y te guiará en los simples pasos para crear tus propios helpers personalizados.

CakePHP incluye varios helpers que ayudan en la creación de vistas. Ayudan en la creación de estructuras bien formadas (incluyendo formularios), ayudan en el formato de texto, horas y números, e incluso pueden acelerar la funcionalidad AJAX. Para obtener más información sobre los helpers incluidos en CakePHP, consulta el capítulo correspondiente para cada helper:

Breadcrumbs

class Cake\View\Helper**BreadcrumbsHelper**(*View $view*, *array $config = []*)

El Helper de Navegación por rastro proporciona una manera sencilla de gestionar la creación y representación de un rastro de migas de pan para tu aplicación.

Creando un Rastro de Migas de Pan

Puedes agregar una miga a la lista utilizando el método «add()». Este método toma tres argumentos:

- **title** Un texto que se mostrará como el título de la miga.
- **url** Un textto o un arreglo de parámetros que se proporcionarán a la *UrlHelper*
- **options** Un arreglo de atributos para el `item` y `itemWithoutLink` template. Ver la section acerca *definir atributos para el registro* para más información.

Además de agregar al final del rastro, puedes realizar una variedad de operaciones:

```
// Añade una miga al final.
$this->Breadcrumbs->add(
    'Products',
    ['controller' => 'products', 'action' => 'index']
);

// Añade varias migas al final del rastro.
$this->Breadcrumbs->add([
    ['title' => 'Products', 'url' => ['controller' => 'products', 'action' => 'index']],
    ['title' => 'Product name', 'url' => ['controller' => 'products', 'action' => 'view',
↪ 1234]],
]);

// Añade una miga al principio.
$this->Breadcrumbs->prepend(
    'Products',
    ['controller' => 'products', 'action' => 'index']
);

// Añade múltiple migas al final, en el orden dado
$this->Breadcrumbs->prepend([
    ['title' => 'Products', 'url' => ['controller' => 'products', 'action' => 'index']],
    ['title' => 'Product name', 'url' => ['controller' => 'products', 'action' => 'view',
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪ 1234]],
]);

// Inserta la miga en una posición específica. Si la posición está fuera de los
// límites, se generará una excepción.
$this->Breadcrumbs->insertAt(
    2,
    'Products',
    ['controller' => 'products', 'action' => 'index']
);

// Inserta la miga antes de una miga específica, basado en el título.
// Si no se puede encontrar el título de la miga nombrada,
// se generará una excepción.
$this->Breadcrumbs->insertBefore(
    'A product name', // el título de la miga para insertar antes
    'Products',
    ['controller' => 'products', 'action' => 'index']
);

// Inserta la miga despues de una miga específica, basado en el título.
// Si no se puede encontrar el título de la miga nombrada,
// se generará una excepción.
$this->Breadcrumbs->insertAfter(
    'A product name', // el título de la miga para insertar despues
    'Products',
    ['controller' => 'products', 'action' => 'index']
);
```

El uso de estos métodos te proporciona la capacidad de trabajar con el proceso de representación de dos pasos de CakePHP. Dado que los `templates` y `layouts` se representan de adentro hacia afuera (es decir, los elementos incluidos se representan primero), esto te permite definir con precisión dónde deseas agregar una miga de pan.

Renderización del Rastro de Migas de Pan

Después de agregar migas al rastro, puedes representarlo fácilmente utilizando el método `render()` . Este método acepta dos argumentos en forma de arreglos:

- **`$attributes`**
  [Un arreglo de atributos que se aplicarán a la plantilla «wrapper».] Esto te permite agregar atributos a la etiqueta HTML. Acepta la clave especial «templateVars» para permitir la inserción de variables de plantilla personalizadas en el `template`.
- **`$separator`**
  [Un arreglo de atributos para el `separator` template.] Las propiedades posibles son:
  - `separator` El texto que se mostrará como separador.
  - `innerAttrs` Para proporcionar atributos en caso de que tu separador esté dividido en dos elementos.
  - `templateVars` Permite la inserción de una variable de plantilla personalizada en el `template`.

  Todas las demás propiedades se convertirán en atributos HTML y reemplazarán la clave «attrs» en la plantilla. Si utilizas el valor predeterminado para esta opción `(empty)`, no se representará un separador.

Aquí tienes un ejemplo de cómo representar un rastro:

```
echo $this->Breadcrumbs->render(
    ['class' => 'breadcrumbs-trail'],
    ['separator' => '<i class="fa fa-angle-right"></i>']
);
```

Personalizando el resultado

El `BreadcrumbsHelper` internamente usa el `StringTemplateTrait`, lo que proporciona la capacidad de personalizar fácilmente la salida de varias cadenas HTML. Incluye cuatro plantillas, con la siguiente declaración predeterminada:

```
[
    'wrapper' => '<ul{{attrs}}>{{content}}</ul>',
    'item' => '<li{{attrs}}><a href="{{url}}"{{innerAttrs}}>{{title}}</a></li>{
↪{separator}}',
    'itemWithoutLink' => '<li{{attrs}}><span{{innerAttrs}}>{{title}}</span></li>{
↪{separator}}',
    'separator' => '<li{{attrs}}><span{{innerAttrs}}>{{separator}}</span></li>'
]
```

Puedes personalizarlos fácilmente utilizando el método `setTemplates()` de `StringTemplateTrait`:

```
$this->Breadcrumbs->setTemplates([
    'wrapper' => '<nav class="breadcrumbs"><ul{{attrs}}>{{content}}</ul></nav>',
]);
```

Dado que tus `templates` serán renderizados, la opción `templateVars` te permite agregar tu propio template de variables a los diferentes templates

```
$this->Breadcrumbs->setTemplates([
    'item' => '<li{{attrs}}>{{icon}}<a href="{{url}}"{{innerAttrs}}>{{title}}</a></li>{
↪{separator}}'
]);
```

Para definir el parámetro `{{icon}}`, simplemente especifícala al agregar la miga:

```
$this->Breadcrumbs->add(
    'Products',
    ['controller' => 'products', 'action' => 'index'],
    [
        'templateVars' => [
            'icon' => '<i class="fa fa-money"></i>',
        ],
    ]
);
```

Definiendo Atributos

Si deseas aplicar atributos HTML específicos tanto al elemento como a su subelemento, puedes aprovechar la clave `innerAttrs`, que proporciona el argumento `$options`. Todo excepto `innerAttrs` y `templateVars` se representará como atributos HTML.:

```
$this->Breadcrumbs->add(
    'Products',
    ['controller' => 'products', 'action' => 'index'],
    [
        'class' => 'products-crumb',
        'data-foo' => 'bar',
        'innerAttrs' => [
            'class' => 'inner-products-crumb',
            'id' => 'the-products-crumb',
        ],
    ]
);

// Según la plantilla predeterminada, esto representará el siguiente HTML:
<li class="products-crumb" data-foo="bar">
    <a href="/products/index" class="inner-products-crumb" id="the-products-crumb">
↪Products</a>
</li>
```

Borrando las Migas de Pan

Puedes borrar las migas de pan utilizando el método `reset()`. Esto puede ser útil cuando deseas transformar las migas y sobrescribir la lista:

```
$crumbs = $this->Breadcrumbs->getCrumbs();
$crumbs = collection($crumbs)->map(function ($crumb) {
    $crumb['options']['class'] = 'breadcrumb-item';

    return $crumb;
})->toArray();

$this->Breadcrumbs->reset()->add($crumbs);
```

FlashHelper

class Cake\View\Helper**FlashHelper**(*View $view*, *array $config = []*)

FlashHelper proporciona una forma de representar mensajes flash que se establecieron en `$_SESSION` a través de *FlashComponent*. Tanto *FlashComponent* como FlashHelper utilizan principalmente `elements` para renderizar mensajes flash. Flash elements se pueden encontrar en el directorio **templates/element/flash**. Puedes notar CakePHP's App template viene con tres elementos flash: **success.php**, **default.php**, and **error.php**.

Renderizando Mensajes Flash

Para renderizar un mensaje flash, puedes simplemente utilizar el método `render()` del FlashHelper en el template default `templates/layout/default.php`:

```
<?= $this->Flash->render() ?>
```

Por defecto, CakePHP utiliza un «flash» key para los mensajes flash messages en la sesión. Sin embargo, si has especificado un «key» al establecer el mensaje flash en *FlashComponent*, puedes especificar cuál «key» de flash renderizar:

```
<?= $this->Flash->render('other') ?>
```

También puedes sobreescribir cualquiera de las opciones establecidas en FlashComponent:

```
// En el Controller
$this->Flash->set('The user has been saved.', [
    'element' => 'success'
]);

// En el template: Utilizará great_success.php en vez de success.php
<?= $this->Flash->render('flash', [
    'element' => 'great_success'
]);

// En el template: el elemento flashy del plugin "Company"
<?= $this->Flash->render('flash', [
    'element' => 'Company.flashy'
]);
```

Nota: Cuando construyas algun template personalizado para mensajes flash, asegúrate de codificar (encode) correctamente en HTML cualquier dato del usuario. CakePHP no escapará (escape) los parámetros de los mensajes flash por ti.

Para obtener más información sobre las opciones disponibles en el arreglo, consulta la sección *FlashComponent*

Routing Prefix y Mensajes Flash

If you have a Routing prefix configured, ahora puedes tener tus elementos Flash almacenados en **templates/{Prefix}/element/flash**. De esta manera, puedes tener mensajes específicos para cada parte de tu aplicación. Por ejemplo, puedes utilizar diferentes «layoouts» para la sección de frontend y la sección de administración.

Mensajes Flash y Tema

El The FlashHelper utiliza `elements` normales para renderizar los mensajes y, por lo tanto, respetará cualquier Tema que hayas especificado. Entonces, cuando tu Tema tiene un archivo templates/element/flash/error.php, se utilizará, al igual que con cualquier otro `Elements` y `Views`.

FormHelper

`class` Cake\View\Helper**FormHelper**(*View $view*, *array $config = []*)

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[103] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

HtmlHelper

`class` Cake\View\Helper**HtmlHelper**(*View $view*, *array $config = []*)

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[104] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Helper Number

`class` Cake\View\Helper**NumberHelper**(*View $view*, *array $config = []*)

El NumberHelper contiene métodos convenientes que permiten mostrar números en formatos comunes en tus vistas. Estos métodos incluyen formas de formatear moneda, porcentajes, tamaños de datos, precisiones específicas y también ofrecen mayor flexibilidad en el formato de números.

Todas estas funciones devuelven el número formateado; no imprimen automáticamente la salida en la vista.

Formato de Valores Monetarios

Cake\View\Helper\NumberHelper::**currency**(*mixed $value*, *string $currency = null*, *array $options = []*)

Este método se utiliza para mostrar un número en formatos de moneda comunes (EUR, GBP, USD), basándose en el código de moneda de tres letras ISO 4217. Su uso en una vista se ve así:

```
// Llamado como NumberHelper
echo $this->Number->currency($value, $currency);

// Llamado como Number
echo Number::currency($value, $currency);
```

[103] https://github.com/cakephp/docs
[104] https://github.com/cakephp/docs

El primer parámetro, `$value`, debería ser un número de punto flotante que representa la cantidad de dinero que estás expresando. El segundo parámetro es una cadena utilizada para elegir un esquema de formato de moneda predefinido:

$currency	1234,56, formateado por tipo de moneda
EUR	€1.234,56
GBP	£1.234,56
USD	$1.234,56

El tercer parámetro es un arreglo de opciones para definir aún más la salida. Las siguientes opciones están disponibles:

Opción	Descripción
before	Texto para mostrar antes del número formateado.
after	Texto para mostrar después del número formateado.
zero	El texto a usar para los valores cero; puede ser una cadena o un número, por ejemplo, 0, "¡Gratis!".
places	Número de lugares decimales a usar, por ejemplo, 2
precision	Número máximo de lugares decimales a usar, por ejemplo, 2.
locale	El nombre de la localidad a usar para formatear el número, por ejemplo, «es_ES».
fractionSymbol	Cadena a usar para números fraccionarios, por ejemplo, "centavos".
fractionPosition	Ya sea "antes" o "después" para colocar el símbolo fraccionario.
pattern	Un patrón de número ICU para usar para formatear el número, por ejemplo, #,###.00.
useIntlCode	Establecer en `true` para reemplazar el símbolo de moneda con el código de moneda internacional.

Si el valor de `$currency` es `null`, la moneda predeterminada se recuperará de `Cake\I18n\Number::defaultCurrency()`. Para formatear monedas en un formato de contabilidad, debes establecer el formato de la moneda:

```
Number::setDefaultCurrencyFormat(Number::FORMAT_CURRENCY_ACCOUNTING);
```

Configurar la Moneda Predeterminada

`Cake\View\Helper\NumberHelper::setDefaultCurrency(`*$currency*`)`

Configura la moneda predeterminada. Esto evita la necesidad de pasar siempre la moneda a *`Cake\I18n\Number::currency()`* y cambiar todas las salidas de moneda configurando otro valor predeterminado. Si `$currency` se establece en `null`, se eliminará el valor almacenado actualmente.

Obtener la Moneda Predeterminada

`Cake\View\Helper\NumberHelper::getDefaultCurrency()`

Obtén la moneda predeterminada. Si la moneda predeterminada se configuró anteriormente utilizando `setDefaultCurrency()`, se devolverá ese valor. De forma predeterminada, recuperará el valor de la ini de `intl.default_locale` si está configurado y `'en_US'` si no lo está.

Formato de Números de Punto Flotante

Cake\View\Helper\NumberHelper::**precision**(*float $value*, *int $precision = 3*, *array $options = []*)

Este método muestra un número con la cantidad especificada de precisión (lugares decimales). Se redondeará para mantener el nivel de precisión definido.

```
// Llamado como NumberHelper
echo $this->Number->precision(456.91873645, 2);

// Salida
456.92

// Llamado como Number
echo Number::precision(456.91873645, 2);
```

Formato de Porcentajes

Cake\View\Helper\NumberHelper::**toPercentage**(*mixed $value*, *int $precision = 2*, *array $options = []*)

Opción	Descripción
multiply	Booleano para indicar si el valor debe ser multiplicado por 100. Útil para porcentajes decimales.

Al igual que *Cake\I18n\Number::precision()*, este método formatea un número según la precisión proporcionada (donde los números se redondean para cumplir con la precisión dada). Adicionalmente, también expresa el número como un porcentaje y agrega un signo de porcentaje a la salida.

```
// Llamado como NumberHelper. Salida: 45.69%
echo $this->Number->toPercentage(45.691873645);

// Llamado como Number. Salida: 45.69%
echo Number::toPercentage(45.691873645);

// Llamado con multiplicar. Salida: 45.7%
echo Number::toPercentage(0.45691, 1, [
    'multiply' => true
]);
```

Interactuar con Valores Legibles para Humanos

Cake\View\Helper\NumberHelper::**toReadableSize**(*string $size*)

Este método formatea tamaños de datos en formas legibles para humanos. Proporciona una forma abreviada de convertir bytes a KB, MB, GB y TB. El tamaño se muestra con un nivel de precisión de dos dígitos, de acuerdo con el tamaño de los datos suministrados (es decir, los tamaños más altos se expresan en términos más grandes):

```
// Llamado como NumberHelper
echo $this->Number->toReadableSize(0); // 0 Byte
echo $this->Number->toReadableSize(1024); // 1 KB
```

(continué en la próxima página)

(proviene de la página anterior)

```
echo $this->Number->toReadableSize(1321205.76); // 1.26 MB
echo $this->Number->toReadableSize(5368709120); // 5 GB

// Llamado como Number
echo Number::toReadableSize(0); // 0 Byte
echo Number::toReadableSize(1024); // 1 KB
echo Number::toReadableSize(1321205.76); // 1.26 MB
echo Number::toReadableSize(5368709120); // 5 GB
```

Formato de Números

Cake\View\Helper\NumberHelper::**format**(*mixed $value*, *array $options = []*)

Este método te brinda mucho más control sobre el formato de números para usar en tus vistas (y se utiliza como el método principal por la mayoría de los otros métodos de NumberHelper). Usar este método puede verse así:

```
// Llamado como NumberHelper
$this->Number->format($value, $options);

// Llamado como Number
Number::format($value, $options);
```

El parámetro `$value` es el número que estás planeando formatear para la salida. Sin opciones proporcionadas, el número 1236.334 se mostraría como 1,236. Ten en cuenta que la precisión predeterminada es cero decimales.

El parámetro `$options` es donde reside la verdadera magia para este método.

- Si pasas un entero, este se convierte en la cantidad de precisión o lugares para la función.
- Si pasas un arreglo asociado, puedes usar las siguientes claves:

Opción	Descripción
places	Número de lugares decimales a usar, por ejemplo, 2
precision	Número máximo de lugares decimales a usar, por ejemplo, 2.
pattern	Un patrón de número ICU para usar para formatear el número, por ejemplo, #,###.00.
locale	El nombre de la localidad a usar para formatear el número, por ejemplo, «es_ES».
before	Texto para mostrar antes del número formateado.
after	Texto para mostrar después del número formateado.

Ejemplo:

```
// Llamado como NumberHelper
echo $this->Number->format('123456.7890', [
    'places' => 2,
    'before' => '¥ ',
    'after' => ' !'
]);
// Salida '¥ 123,456.79 !'

echo $this->Number->format('123456.7890', [
    'locale' => 'fr_FR'
```

(continué en la próxima página)

(proviene de la página anterior)

```
]);
// Salida '123 456,79 !'

// Llamado como Number
echo Number::format('123456.7890', [
    'places' => 2,
    'before' => '¥ ',
    'after' => ' !'
]);
// Salida '¥ 123,456.79 !'

echo Number::format('123456.7890', [
    'locale' => 'fr_FR'
]);
// Salida '123 456,79 !'
```

Cake\View\Helper\NumberHelper::**ordinal**(*mixed $value*, *array $options = []*)

Este método mostrará un número ordinal.

Ejemplos:

```
echo Number::ordinal(1);
// Salida '1st'

echo Number::ordinal(2);
// Salida '2nd'

echo Number::ordinal(2, [
    'locale' => 'fr_FR'
]);
// Salida '2e'

echo Number::ordinal(410);
// Salida '410th'
```

Diferencias en el Formato

Cake\View\Helper\NumberHelper::**formatDelta**(*mixed $value*, *array $options = []*)

Este método muestra diferencias en el valor como un número con signo:

```
// Llamado como NumberHelper
$this->Number->formatDelta($value, $options);

// Llamado como Number
Number::formatDelta($value, $options);
```

El parámetro `$value` es el número que estás planeando formatear para la salida. Sin opciones proporcionadas, el número 1236.334 se mostraría como 1,236. Ten en cuenta que la precisión predeterminada es cero decimales.

El parámetro `$options` toma las mismas claves que `Number::format()` en sí:

Opción	Descripción
places	Número de lugares decimales a usar, por ejemplo, 2
precision	Número máximo de lugares decimales a usar, por ejemplo, 2.
locale	El nombre de la localidad a usar para formatear el número, por ejemplo, «es_ES».
before	Texto para mostrar antes del número formateado.
after	Texto para mostrar después del número formateado.

Ejemplo:

```
// Llamado como NumberHelper
echo $this->Number->formatDelta('123456.7890', [
    'places' => 2,
    'before' => '[',
    'after' => ']'
]);
// Salida '[+123,456.79]'

// Llamado como Number
echo Number::formatDelta('123456.7890', [
    'places' => 2,
    'before' => '[',
    'after' => ']'
]);
// Salida '[+123,456.79]'
```

Advertencia: Todos los símbolos son UTF-8.

PaginatorHelper

class Cake\View\Helper**PaginatorHelper**(*View $view*, *array $config = []*)

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[105] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

---

[105] https://github.com/cakephp/docs

TextHelper

`class` Cake\View\Helper**TextHelper**(*View $view*, *array $config = []*)

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[106] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

TimeHelper

`class` Cake\View\Helper**TimeHelper**(*View $view*, *array $config = []*)

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[107] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

UrlHelper

`class` Cake\View\Helper**UrlHelper**(*View $view*, *array $config = []*)

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[108] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Configurando Helpers

Configuras los helpers en CakePHP declarándolos en una clase de vista. Cada aplicación CakePHP incluye una clase `AppView` que es el lugar ideal para añadir helpers para su uso global:

```
class AppView extends View
{
    public function initialize(): void
    {
        parent::initialize();
```

(continué en la próxima página)

[106] https://github.com/cakephp/docs
[107] https://github.com/cakephp/docs
[108] https://github.com/cakephp/docs

(proviene de la página anterior)

```
        $this->addHelper('Html');
        $this->addHelper('Form');
        $this->addHelper('Flash');
    }
}
```

Para añadir helpers desde plugins, utiliza la *Sintaxis de plugin* utilizada en otros lugares en CakePHP:

```
$this->addHelper('Blog.Comment');
```

No es necesario añadir explícitamente los Helpers que provienen de CakePHP o de tu aplicación. Estos helpers se pueden cargar de forma tardía (lazy loaded) cuando se utilizan por primera vez. Por ejemplo:

```
// Carga el FormHelper si aún no se ha añadido/cargado explícitamente.
$this->Form->create($article);
```

Desde las vistas de un plugin, los helpers del plugin también se pueden cargar de forma tardía. Por ejemplo, las plantillas de vista en el plugin "Blog" pueden cargar los helpers del mismo plugin.

Carga Condicional de Helpers

Puedes utilizar el nombre de la acción actual para añadir helpers de forma condicional:

```
class AppView extends View
{
    public function initialize(): void
    {
        parent::initialize();
        if ($this->request->getParam('action') === 'index') {
            $this->addHelper('ListPage');
        }
    }
}
```

También puedes utilizar el método `beforeRender` de tu controlador para añadir helpers:

```
class ArticlesController extends AppController
{
    public function beforeRender(EventInterface $event)
    {
        parent::beforeRender($event);
        $this->viewBuilder()->addHelper('MyHelper');
    }
}
```

Opciones de Configuración

Puedes pasar opciones de configuración a los helpers. Estas opciones se pueden utilizar para establecer valores de atributos o modificar el comportamiento de un helper:

```
namespace App\View\Helper;

use Cake\View\Helper;
use Cake\View\View;

class AwesomeHelper extends Helper
{
    public function initialize(array $config): void
    {
        debug($config);
    }
}
```

Por defecto, todas las opciones de configuración se fusionarán con la propiedad `$_defaultConfig`. Esta propiedad debe definir los valores por defecto de cualquier configuración que tu helper requiera. Por ejemplo:

```
namespace App\View\Helper;

use Cake\View\Helper;
use Cake\View\StringTemplateTrait;

class AwesomeHelper extends Helper
{
    use StringTemplateTrait;

    /**
     * @var array<string, mixed>
     */
    protected $_defaultConfig = [
        'errorClass' => 'error',
        'templates' => [
            'label' => '<label for="{{for}}">{{content}}</label>',
        ],
    ];
}
```

Cualquier configuración proporcionada al constructor de tu helper se fusionará con los valores por defecto durante la construcción y los datos fusionados se establecerán en `_config`. Puedes utilizar el método `getConfig()` para leer la configuración en tiempo de ejecución:

```
// Lee la opción de configuración errorClass.
$class = $this->Awesome->getConfig('errorClass');
```

Usar la configuración del helper te permite configurar declarativamente tus helpers y mantener la lógica de configuración fuera de las acciones de tu controlador. Si tienes opciones de configuración que no se pueden incluir como parte de una declaración de clase, puedes configurarlas en el callback `beforeRender` de tu controlador:

```
class PostsController extends AppController
{
```

(continué en la próxima página)

(proviene de la página anterior)

```
    public function beforeRender(EventInterface $event)
    {
        parent::beforeRender($event);
        $builder = $this->viewBuilder();
        $builder->helpers([
            'CustomStuff' => $this->_getCustomStuffConfig(),
        ]);
    }
}
```

Alias de Helpers

Un ajuste común para usar es la opción `className`, que te permite crear alias para los helpers en tus vistas. Esta característica es útil cuando quieres reemplazar `$this->Html` u otra referencia común de Helper con una implementación personalizada:

```
// src/View/AppView.php
class AppView extends View
{
    public function initialize(): void
    {
        $this->addHelper('Html', [
            'className' => 'MyHtml',
        ]);
    }
}

// src/View/Helper/MyHtmlHelper.php
namespace App\View\Helper;

use Cake\View\Helper\HtmlHelper;

class MyHtmlHelper extends HtmlHelper
{
    // Agrega tu código para sobrescribir el HtmlHelper principal
}
```

Lo anterior haría que `MyHtmlHelper` se pudiera utilizar usando el *alias* `$this->Html` en tus vistas.

Nota: Hacer un alias de un helper reemplaza esa instancia en cualquier lugar donde se utilice ese helper, incluso dentro de otros Helpers.

Usar Helpers

Una vez que hayas configurado qué helpers quieres usar en tu controlador, cada helper se expone como una propiedad pública en la vista. Por ejemplo, si estás utilizando el `HtmlHelper`, podrías acceder a él haciendo lo siguiente:

```
echo $this->Html->css('styles');
```

Lo anterior llamaría al método `css()` en el HtmlHelper. Puedes acceder a cualquier helper cargado usando `$this->{$nombreDelHelper}`.

Cargar Helpers Dinámicamente

Puede haber situaciones en las que necesites cargar dinámicamente un helper desde dentro de una vista. Puedes usar el `Cake\View\HelperRegistry` de la vista para hacer esto:

```
// Cualquiera de los dos funciona.
$mediaHelper = $this->loadHelper('Media', $mediaConfig);
$mediaHelper = $this->helpers()->load('Media', $mediaConfig);
```

El HelperRegistry es un *registro* y es compatible con la API de registro utilizada en otros lugares en CakePHP.

Métodos de Eventos

Los helpers tienen varios métodos que se ejecutan con determinados eventos yque te permiten cambiar el proceso de renderización de la vista. Consulta el *Clase Helper* y la documentación de *Events System* para obtener más información.

Creación de Helpers

Puedes crear clases de helper personalizadas para su uso en tu aplicación o plugins. Al igual que la mayoría de los componentes de CakePHP, las clases de helper tienen algunas convenciones:

- Los archivos de clase del helper deben colocarse en **src/View/Helper**. Por ejemplo: **src/View/Helper/LinkHelper.php**
- Los nombres de las clases de helper deben llevar el sufijo `Helper`. Por ejemplo: `LinkHelper`.
- Al hacer referencia a los nombres de los helper, debes omitir el sufijo `Helper`. Por ejemplo: `$this->addHelper('Link');` o `$this->loadHelper('Link');`.

También querrás extender `Helper` para asegurarte de que todo funcione correctamente:

```
/* src/View/Helper/LinkHelper.php */
namespace App\View\Helper;

use Cake\View\Helper;

class LinkHelper extends Helper
{
    public function makeEdit($title, $url)
    {
        // Lógica para crear un enlace con formato especial va aquí...
    }
}
```

Inclusión de otros Helpers

Es posible que desees utilizar alguna funcionalidad ya existente en otro helper. Para hacerlo, puedes especificar los helpers que deseas utilizar con un array `$helpers`, formateado de la misma manera que en un controlador:

```
/* src/View/Helper/LinkHelper.php (usando otros helpers) */

namespace App\View\Helper;

use Cake\View\Helper;

class LinkHelper extends Helper
{
    protected $helpers = ['Html'];

    public function makeEdit($title, $url)
    {
        // Usa el helper HTML para generar
        // Datos formateados:

        $link = $this->Html->link($title, $url, ['class' => 'edit']);

        return '<div class="editOuter">' . $link . '</div>';
    }
}
```

Usando tu nuevo Helper

Una vez que hayas creado tu helper y lo hayas colocado en **src/View/Helper/**, puedes cargarlo en tus vistas:

```
class AppView extends View
{
    public function initialize(): void
    {
        parent::initialize();
        $this->addHelper('Link');
    }
}
```

Una vez que se ha cargado tu helper, puedes usarlo en tus vistas accediendo a la propiedad de vista correspondiente:

```
<!-- crea un enlace usando el nuevo helper -->
<?= $this->Link->makeEdit('Cambiar esta Receta', '/recipes/edit/5') ?>
```

Nota: El `HelperRegistry` intentará cargar de forma tardía cualquier helper que no esté específicamente identificado en tu `Controller`.

Acceso a Variables de Vista dentro de tu Helper

Si deseas acceder a una variable de vista dentro de un helper, puedes utilizar `$this->getView()->get()` como sigue:

```
class AwesomeHelper extends Helper
{
    public $helpers = ['Html'];

    public function someMethod()
    {
        // establece la meta descripción
        return $this->Html->meta(
            'description', $this->getView()->get('metaDescription'), ['block' => 'meta']
        );
    }
}
```

Renderizar un Elemento de Vista dentro de tu Helper

Si deseas renderizar un elemento dentro de tu Helper, puedes usar `$this->getView()->element()` como sigue:

```
class AwesomeHelper extends Helper
{
    public function someFunction()
    {
        return $this->getView()->element(
            '/ruta/a/elemento',
            ['foo'=>'bar','bar'=>'foo']
        );
    }
}
```

Clase Helper

class Helper

Métodos de Eventos

Implementando un método de evento en un helper, CakePHP suscribirá automáticamente tu helper al evento relevante. A diferencia de las versiones anteriores de CakePHP, *no* debes llamar a `parent` en tus métodos, ya que la clase base Helper no implementa ninguno de los métodos de evento.

`Helper::`**beforeRenderFile**(*EventInterface $event*, *$viewFile*)

Se llama antes de renderizar cada archivo de vista. Esto incluye elementos, vistas, vistas principales y diseños (layouts).

`Helper::`**afterRenderFile**(*EventInterface $event*, *$viewFile*, *$content*)

Se llama después de renderizar cada archivo de vista. Esto incluye elementos, vistas, vistas principales y diseños (layouts). Este evento puede modificar y devolver `$content` para cambiar cómo se mostrará el contenido renderizado en el navegador.

`Helper::beforeRender(`*EventInterface $event*, *$viewFile*`)`

El método beforeRender se llama después del método beforeRender del controlador pero antes de que el controlador renderice la vista y el diseño (layout). Recibe el archivo que se está renderizando como argumento.

`Helper::afterRender(`*EventInterface $event*, *$viewFile*`)`

Se llama después de que la vista ha sido renderizada pero antes de que comience el renderizado del diseño (layout).

`Helper::beforeLayout(`*EventInterface $event*, *$layoutFile*`)`

Se llama antes de que comience el renderizado del diseño (layout). Recibe el nombre del archivo del diseño como argumento.

`Helper::afterLayout(`*EventInterface $event*, *$layoutFile*`)`

Se llama después de que se haya completado el renderizado del diseño (layout). Recibe el nombre del archivo del diseño (layout) como argumento.

CAPÍTULO 11

Acceso a la base de datos & ORM

En CakePHP el acceso a la base de datos se hace por medio de dos objetos primarios. El primer tipo de objeto son **repositories** -repositorios- o **table objects** -objetos de tabla-. Estos objetos proveen acceso a colecciones de datos. Nos permiten guardar nuevos registros, modificar y borrar existentes, definir relaciones y realizar operaciones en masa. El segundo tipo de objeto son **entities** -entidades-. Las Entidades representan registros individuales y permiten definir funcionalidad y comportamiento a nivel de registro/fila.

Estas dos clases son responsables de manejar todo lo que sucede con datos, validez, interacción y evolución en tu área de trabajo.

El ORM incluído en CakePHP se especializa en base de datos relacionales, pero puede ser extendido para soportar alternativas.

El ORM de CakePHP toma ideas y conceptos de los modelos ActiveRecord y Datamapper. Aspira a crear una implementación híbrida que combine aspectos de los dos modelos para crear un ORM rápido y fácil de usar.

Antes de comentar explorando el ORM, asegurate de configurar tu conexion *configure your database connections*.

Ejemplo rápido

Para comenzar no es necesario escribir código. Si has seguido las convenciones de nombres para las tablas puedes comenzar a utilizar el ORM. Por ejemplo si quisieramos leer datos de nuestra tabla `articles`:

```
use Cake\ORM\TableRegistry;

// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$query = $articles->find();
foreach ($query as $row) {
    echo $row->title;
}
```

Como se ve, no es necesario agregar código extra ni ninguna otra configuración, gracias al uso de las convenciones de CakePHP. Si quisieramos modificar nuestra clase ArticlesTable para agregar asociaciones o definir métodos adicionales deberiamos agregar las siguientes líneas en **src/Model/Table/ArticlesTable.php**

```
namespace App\Model\Table;

use Cake\ORM\Table;

class ArticlesTable extends Table
{

}
```

Las clases Table usan una version en CamelCase del nombre de la tabla, con el sufijo `Table`. Una vez que tu clase fue creada, puedes obtener una referencia a esta usando `TableRegistry` como antes:

```
use Cake\ORM\TableRegistry;

// Now $articles is an instance of our ArticlesTable class.
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
```

Ahora que tenemos una clase Table concreta, probablemente querramos usar una clase Entity concreta. Las clases Entity permiten definir métodos de acceso y mutación, lógica para registros individuales y mucho más. Comenzaremos agregando las siguientes líneas en **src/Model/Entity/Article.php**:

```
namespace App\Model\Entity;

use Cake\ORM\Entity;

class Article extends Entity
{

}
```

Las Entity usan la version CamelCase en singular del nombre de la tabla como su nombre. Ahora que hemos creado una clase Entity, cuando carguemos entidades de nuestra base de datos, vamos a obtener instancias de nuestra clase Article:

```
use Cake\ORM\TableRegistry;

// Now an instance of ArticlesTable.
// Prior to 3.6 use TableRegistry::get('Articles')
$articles = TableRegistry::getTableLocator()->get('Articles');
$query = $articles->find();

foreach ($query as $row) {
    // Each row is now an instance of our Article class.
    echo $row->title;
}
```

CakePHP usa convenciones de nombres para asociar las clases Table y Entity. Si necesitas modificar qué entidad utilizada una tabla, puedes usar el método `entityClass()` para especificar el nombre de una clase.

Vea *Table Objects* y *Entities* para más información sobre como utilizar objetos Table y Entity en su aplicación.

Más información

Database Basics

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[109] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Query Builder

`class Cake\ORM\Query`

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[110] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Table Objects

`class Cake\ORM\Table`

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[111] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[109] https://github.com/cakephp/docs
[110] https://github.com/cakephp/docs
[111] https://github.com/cakephp/docs

Entities

class Cake\ORM**Entity**

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[112] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Retrieving Data & Results Sets

class Cake\ORM**Table**

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[113] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Custom Finder Methods

Validating Data

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

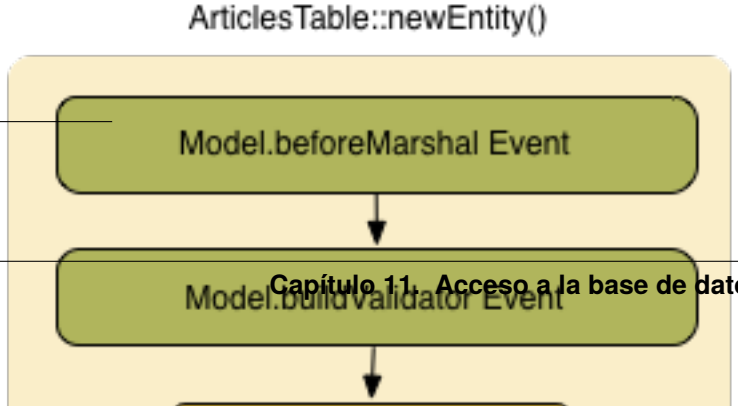
Por favor, siéntase libre de enviarnos un pull request en Github[114] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Saving Data

class Cake\ORM**Table**

Nota: La documentación no es compatible actualmente con el idioma español en esta página.



Por favor,

[112] https://github.com/cakephp/docs
[113] https://github.com/cakephp/docs
[114] https://github.com/cakephp/docs

siéntase libre de enviarnos un pull request en Github[115] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Deleting Data

class Cake\ORM**Table**

[115] https://github.com/cakephp/docs

Cake\ORM\Table::del

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[116] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Associations - Linking Tables Together

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[117] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Behaviors

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[118] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Core Behaviors

CounterCache Behavior

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[119] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

[116] https://github.com/cakephp/docs
[117] https://github.com/cakephp/docs
[118] https://github.com/cakephp/docs
[119] https://github.com/cakephp/docs

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Timestamp Behavior

class Cake\Model\Behavior**TimestampBehavior**

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[120] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Translate

class Cake\Model\Behavior**TranslateBehavior**

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[121] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Tree

class Cake\ORM\Behavior**TreeBehavior**

Muchas veces nos encontramos frente a la necesidad de tener que almacenar datos jerarquizados en una base de datos. Podría tomar la forma de categorías sin límite de subcategorías, datos relacionados con un sistema de menú multinivel o una representación literal de la jerarquía como un departamento en una empresa.

Las bases de datos relacionales no son verdaderamente apropiadas para almacenar y recobrar este tipo de datos, pero existen algunas técnicas para hacerlas eficientes y trabajar con una información multinivel.

[120] https://github.com/cakephp/docs
[121] https://github.com/cakephp/docs

El TreeBehavior le ayuda a mantener una estructura de datos jerárquica en la base de datos que puede ser solicitada fácilmente y ayuda a reconstruir los datos bajo una forma de árbol que permite encontrar y visualizar los procesos.

Prerrequisitos

Ese behavior requiere que las siguientes columnas estén presentes en la tabla:

- `parent_id` (nullable) La columna que contiene el ID del registro padre
- `lft` (integer, signed) Utilizado para mantener la estructura en forma de árbol
- `rght` (integer, signed) Utilizado para mantener la estructura en forma de árbol

Usted puede configurar el nombre de esos campos. Encontrará más información sobre la significación de los campos y sobre la manera de utilizarlos en este artículo que describe la MPTT logic[122]

Advertencia

Por el momento, TreeBehavior no soporta las llaves primarias composites.

[122] https://www.sitepoint.com/hierarchical-data-database-2/

Rápido vistazo

Active el Tree behavior agregándolo a la Tabla donde usted desea almacenar los datos jerarquizados en:

```
class␣
↪CategoriesTable␣
↪extends␣
↪Table
{
    public␣
↪function␣
↪initialize(array␣
↪$config)
    {

↪$this->
↪addBehavior(
↪'Tree');
    }
}
```

Tras agregarlas, puede dejar que CakePHP construya la estructura interna si la tabla ya contiene algunos registros:

```
// Prior to␣
↪3.6 use␣
↪TableRegistry::get(
↪'Categories
↪')
$categories␣
↪=␣
↪TableRegistry::getTableLo
↪>get(
↪'Categories
↪');
$categories-
↪>
↪recover();
```

Usted puede comprobar que funciona recuperando cualquier registro de la tabla y preguntando cuantos descendientes posee:

```
$node =
↪$categories-
↪>get(1);
echo
↪$categories-
↪>
↪childCount(
↪$node);
```

Obtener una lista plana de los descendientes de un nodo es igual de fácil:

```

↪$descendants␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪=
↪$categories-
↪>find(
↪'children
↪', ['for'␣
↪=> 1]);

foreach (
↪$descendants␣
↪as
↪$category)␣
↪{
    echo
↪$category-
↪>name . "\
↪n";
}
```

En cambio, si necesita una lista enlazada donde los hijos de cada nodo están anidados en una jerarquía, usted puede utilizar el finder 'threaded':

```
$children =
↪$categories
    ->find(
↪'children
↪', ['for'␣
↪=> 1])
    ->find(
↪'threaded
↪')
    ->
↪toArray();

foreach (
↪$children␣
↪as
↪$child) {
    echo "{
↪$child->
↪name} has
↪" . count(
↪$child->
↪children)␣
↪. "␣
↪direct␣
↪children";
}
```

Recorrer los resultados encadenados requiere generalmente funciónes recursivas, pero si usted necesita solamente un conjunto de resultados que contenga un campo único a partir de cada nivel para obtener una lista, en un <select> HTML por ejemplo, le será preferible recurrir al finder 'treeList':

```
$list =
↪$categories-
↪>find(
↪'treeList
↪');

// En un␣
↪fichero␣
↪plantilla␣
↪de Cake␣
↪PHP:
echo $this->
↪Form->
↪input(
↪'categories
↪', [
↪'options'␣
↪=>
↪$list]);

//  O puede␣
↪aficharlo␣
↪bajo␣
↪forma de␣
↪texto,␣
↪por␣
↪ejemplo␣
↪en un␣
↪script de␣
↪CLI
foreach (
↪$list as
↪$categoryName)␣
↪{
    echo
↪$categoryName␣
↪. "\n";
}
```

La salida se parecerá a esto:

```
My␣
↪Categories
_Fun
__Sport
___Surfing
___Skating
_Trips
__National
__
↪International
```

El finder `treeList` acepta una serie de opciones:

- `keyPath`: el camino separado por puntos para recuperar el campo que se utilizará en llave de array, o una clausura que devuelve la llave del registro suministrado.
- `valuePath`: el camino separado por puntos para recuperar el campo que se utilizará en llave de array, o una clausura que devuelve la llave del registro suministrado.
- `spacer`: una cadena de caracteres utilizada como prefijo para designar la profundidad del árbol para cada elemento.

Un ejemplo de uso de todas las opciones sería:

```
$query =
↪$categories-
↪>find(
↪'treeList
↪', [
    'keyPath
↪' => 'url
↪',

↪'valuePath
↪' => 'id',
    'spacer
↪' => ' '
]);
```

Une tarea común consiste en encontrar el camino en el árbol a partir de un nodo específico hacia la raíz. Es útil, por ejemplo, para añadir la lista de los hilos de Ariadna para una estructura de menú:

```
$nodeId = 5;
$crumbs =
↪$categories-
↪>find(
↪'path', [
↪'for' =>
↪$nodeId]);

foreach (
↪$crumbs␣
↪as
↪$crumb) {
    echo
↪$crumb->
↪name . ' >
↪ ';
}
```

Los árboles construidos con TreeBehavior no pueden ser clasificados con otras columnas que *lft* `, porque la representación interna del árbol depende de esa clasificación. Afortunadamente se pueden reestructurar los nodos dentro del mismo nivel sin tener que cambiar el elemento padre:

```
$node =
↪$categories-
↪>get(5);

// Desplaza␣
↪el nudo␣
↪para que␣
↪incremente␣
↪de una␣
↪posición␣
↪cuando␣
↪listamos␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪los hijos
$categories-
↪>moveUp(
↪$node);

// ␣
↪Desplaza␣
↪el nudo␣
↪hacia lo␣
↪alto de␣
↪la lista␣
↪en el␣
↪mismo␣
↪nivel
$categories-
↪>moveUp(
↪$node,␣
↪true);

// ␣
↪Desplaza␣
↪el nudo␣
↪hacia␣
↪abajo.
$categories-
↪>moveDown(
↪$node,␣
↪true);
```

Configuración

Si los números de columna predeterminados empleados por ese behavior no corresponden a su esquema, usted puede ponerles alias:

```
public␣
↪function␣
↪initialize(array␣
↪$config)
{
    $this->
↪addBehavior(
↪'Tree', [

↪'parent'␣
↪=>
↪'ancestor_
↪id', // ␣
↪Utilice␣
↪esto␣
↪preferencialmente␣
↪en vez de␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪parent_id

↪'left' =>
↪'tree_left
↪', // ␣
↪Utilice␣
↪esto en␣
↪vez de Ift

↪'right' =>
↪ 'tree_
↪right' //␣
↪ Utilice␣
↪esto en␣
↪vez de␣
↪rght
    ]);
}
```

Nivel de Nodos (profundidad)

Conocer la profundidad de una estructura en árbol puede ser útil cuando quiere recuperar los nodos solo hasta cierto nivel, por ejemplo para generar un menú. Puede utilizar la opción `level` para especificar los campos que guardarán el nivel de cada nodo:

```
$this->
↪addBehavior(
↪'Tree', [
    'level'␣
↪=> 'level
↪', //␣
↪null por␣
↪defecto,␣
↪i.e. no␣
↪guarda el␣
↪nivel
]);
```

Si usted no quiere copiar en caché el nivel utilizando un campo de la base de datos, puede utilizar el método `TreeBehavior::getLevel()` para conocer el nivel de un nodo.

Alcance y árboles múltiples

Si usted desea tener más de una estructura de árbol en la misma tabla, puede hacerlo utilizando la configuración 'scope' (alcance). Por ejemplo, si en una tabla locations desea crear un árbol por país:

```
class␣
↪LocationsTable␣
↪extends␣
↪Table
{
    public␣
↪function␣
↪initialize(array␣
↪$config)
    {

↪$this->
↪addBehavior(
↪'Tree', [

↪'scope' =>
↪ [
↪'country_
↪name' =>
↪'Brazil']
        ]);
    }
}
```

En el precedente ejemplo precedentela totalidad de las operaciones realizadas sobre el árbol solo se enfocarán en los registros que tienen la columna `country_name` que vale 'Brazil'. Usted puede cambiar el scope al vuelo utilizando la función 'config':

```
$this->
↪behaviors()-
↪>Tree->
↪config(
↪'scope', [
↪'country_
↪name' =>
↪'France
↪']);
```

Opcionalmente, puede ejercer un control más riguroso pasando una clausura como scope

```
$this->
↪behaviors()-
↪>Tree->
↪config(
↪'scope',␣
↪function (
↪$query) {

↪$country␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪= $this->
↪getConfigureContry();
↪ // A␣
↪made-up␣
↪function
    return
↪$query->
↪where([
↪'country_
↪name' =>
↪$country]);
↪
});
```

Recobro con campo de clasificación personalizada

Por defecto, recover() clasifica los elementos por llave primaria. Eso funciona muy bien si se trata de una columna numérica (con incremento automático), pero puede ocasionar resultados raros si usted utiliza los UUIDs. Si necesita una clasificación personalizada para la recuperación de datos, puede agregar una cláusula de orden en la configuración:

```
$this->
↪addBehavior(
↪'Tree', [

↪'recoverOrder
↪' => [
↪'country_
↪name' =>
↪'DESC'],
]);
```

Guardar los datos jerarquizados

Generalmente cuando utiliza el Tree behavior, no tiene que preocuparse por la representación interna de la estructura jerarquizada. Las posiciones donde los nodos están colocados en el árbol se deducen de la columna 'parent_id' en cada una de sus entities:

```
$aCategory␣
↪=
↪$categoriesTable-
↪>get(10);
$aCategory->
↪parent_id␣
↪= 5;

↪$categoriesTable-
↪>save(
↪$aCategory);
↪
```

Proveer ids de padres inexistentes al grabar o intentar crear un bucle en el árbol (hacer un nodo hijo del mismo) provocará una excepción. Puede hacer un nodo a la raíz del árbol asignándolenull a la columna 'parent_id':

```
$aCategory␣
↪=
↪$categoriesTable-
↪>get(10);
$aCategory->
↪parent_id␣
↪= null;

↪$categoriesTable-
↪>save(
↪$aCategory);
↪
```

Los hijos para el nuevo nodo serán preservados.

Suprimir Nodos

Es fácil Suprimir un nodo, así como todo su sub-árbol (todos los hijos que puede tener a todo nivel del árbol):

```
$aCategory␣
↪=
↪$categoriesTable-
↪>get(10);

↪$categoriesTable-
↪>delete(
↪$aCategory);
↪
```

TreeBehavior se ocupará de todas las operaciones internas de supresión. También es posible suprimir solamente un nodo y reasignar todos los hijos al nodo padre inmediatamente superior en el árbol:

```
$aCategory␣
↪=
↪$categoriesTable-
↪>get(10);

↪$categoriesTable-
↪>
↪removeFromTree(
↪$aCategory);
↪

↪$categoriesTable-
↪>delete(
↪$aCategory);
↪
```

Todos los nodos hijos serán conservados y un nuevo padre les será asignado. La supresión de un nodo se basa sobre los valores lft y rght de la entity. Es importante observarlo cuando se ejecuta un bucle sobre los hijos de un nodo para supresiones condicionales:

```
↪$descendants␣
↪= $teams->
↪find(
↪'children
↪', ['for'␣
↪=> 1]);
foreach (
↪$descendants␣
↪as
↪$descendant)␣
↪{
    $team =
↪$teams->
↪get(
↪$descendant-
↪>id); //␣
↪busca el␣
↪objeto␣
↪entity al␣
↪día
    if (
↪$team->
↪expired) {

↪$teams->
↪delete(
↪$team); //
↪ la␣
↪supresión␣
↪reclasifica␣
↪las␣
↪entradas␣
↪lft y␣
↪rght de␣
↪la base␣
↪de datos
    }
}
```

TreeBehavior reclasifica los valores lft y rght de los registros de la tabla cuando se suprime un nodo. Tal como están, los valores lft y rght de las entities dentro de `$descendants` (guardadas antes de la operación de supresión) serán erróneas. Las entities tendrán que estar cargadas, y modificadas al vuelo para evitar incoherencias en la tabla.

Schema System

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[123] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Herramienta de esquemas de caché (Schema Cache)

SchemaCacheShell proporciona una herramienta CLI sencilla para administrar las cachés de metadatos de su aplicación. En situaciones de implementación, resulta útil reconstruir la caché de metadatos in situ sin borrar los datos de la caché existente. Puedes hacer esto ejecutando:

```
bin/cake␣
↪schema_
↪cache␣
↪build --
↪connection␣
↪default
```

Esto reconstruirá el caché de metadatos para todas las tablas en la conexión `default`. Si solo necesita reconstruir una única tabla, puede hacerlo proporcionando su nombre:

```
bin/cake␣
↪schema_
↪cache␣
↪build --
↪connection␣
↪default␣
↪articles
```

Además de crear datos almacenados en caché, también puede utilizar SchemaCacheShell para eliminar metadatos almacenados en caché:

```
# Borrar␣
↪todos los␣
↪metadatos
bin/cake␣
↪schema_
↪cache␣
↪clear

# Limpiar␣
↪una sola␣
↪tabla
bin/cake␣
↪schema_
```

(continué en la próxima página)

[123] https://github.com/cakephp/docs

(proviene de la página anterior)

```
↪cache␣
↪clear␣
↪articles
```

CAPÍTULO 12

Consola bake

Esta página se ha movido[124].

[124] https://book.cakephp.org/bake/1.x/es/

CAPÍTULO 13

Caching

class Cake\Cache**Cache**

El almacenamiento en caché se puede utilizar para acelerar la lectura de recursos caros o lentos, manteniendo una segunda copia de los datos requeridos en un sistema de almacenamiento más rápido o más cercano. Por ejemplo, puedes almacenar los resultados de consultas costosas o el acceso a servicios web remotos que no cambian con frecuencia en una caché. Una vez que los datos están en la caché, leerlos desde la caché es mucho más económico que acceder al recurso remoto.

En CakePHP, el almacenamiento en caché se facilita mediante la clase `Cache`. Esta clase proporciona una interfaz estática y uniforme para interactuar con diversas implementaciones de almacenamiento en caché. CakePHP proporciona varios motores de caché y ofrece una interfaz sencilla si necesitas construir tu propio backend. Los motores de almacenamiento en caché integrados son:

- `File`: el almacenamiento en caché de archivos es una caché simple que utiliza archivos locales. Es el motor de caché más lento y no proporciona

muchas características para operaciones atómicas. Sin embargo, dado que el almacenamiento en disco a menudo es bastante económico, almacenar objetos grandes o elementos que rara vez se escriben funciona bien en archivos.

- `Memcached`: utiliza la extensión Memcached[125].
- `Redis`: utiliza la extensión phpredis[126]. Redis proporciona un sistema de caché rápido y persistente similar a Memcached y también ofrece opera-

[125] https://php.net/memcached
[126] https://github.com/phpredis/phpredis

ciones atómicas.

- `Apcu`: la caché de APCu utiliza la extensión PHP APCu[127]. Esta extensión utiliza memoria compartida en el servidor web para almacenar objetos. Esto lo hace muy rápido y capaz de proporcionar funciones de lectura/escritura atómicas.
- `Array`: almacena todos los datos en una matriz. Este motor no proporciona almacenamiento persistente y está destinado a su uso en suites de pruebas de aplicaciones.
- `Null`: el motor

[127] https://php.net/apcu

nulo en realidad no almacena nada y falla en todas las operaciones de lectura.

Independientemente del motor de caché que elijas usar, tu aplicación interactúa con *Cake\Cache\Cache*.

Configuración de los Motores de Caché

static Cake\Cache\Cache::se

Tu aplicación puede configurar cualquier número de "motores" durante su proceso de inicio. Las configuraciones del motor de caché se definen en **config/app.php**.

Para un rendimiento óptimo, CakePHP requiere que se definan dos motores de caché.

- `_cake_core_` se utiliza para almacenar mapas de archivos y resultados analizados de archivos de *Internationalization & Localization*.
- `_cake_model_` se utiliza para almacenar descripciones de esquemas para los

modelos de tu aplicación.

Usar múltiples configuraciones de motores también te permite cambiar incrementalmente el almacenamiento según sea necesario. Por ejemplo, en tu **config/app.php** podrías poner lo siguiente:

```
// ...
'Cache' => [
    'short'␣
↪=> [

↪'className
↪' => 'File
↪',

↪'duration
↪' => '+1␣
↪hours',

↪'path' =>␣
↪CACHE,

↪'prefix'␣
↪=> 'cake_
↪short_',
    ],
    //␣
↪Usando un␣
↪nombre␣
↪completamente␣
↪calificado.
↪
    'long'␣
↪=> [

↪'className
↪' =>
↪'Cake\
↪Cache\
↪Engine\
↪FileEngine
↪',

↪'duration
↪' => '+1␣
↪week',

↪'probability
↪' => 100,

↪'path' =>␣
↪CACHE .
↪'long' .␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪DS,
    ],
]
// ...
```

Las opciones de configuración también se pueden proporcionar como una cadena *DSN*. Esto es útil cuando se trabaja con variables de entorno o proveedores de *PaaS*:

```
Cache::setConfig(
↪'short', [
    'url' =>
↪
↪'memcached:/
↪/
↪user:password@cache-
↪host/?
↪timeout=3600&
↪prefix=myapp_
↪',
]);
```

Cuando usas una cadena DSN, puedes definir cualquier parámetro/opción adicional como argumentos de cadena de consulta.

También puedes configurar los motores de caché en tiempo de ejecución:

```
// Usando␣
↪un nombre␣
↪corto
Cache::setConfig(
↪'short', [

↪'className
↪' => 'File
↪',

↪'duration
↪' => '+1␣
↪hours',
    'path'␣
↪=> CACHE,
    'prefix
↪' =>
↪'cake_
↪short_'
]);

// Usando␣
↪un nombre␣
↪completamente␣
↪calificado.
↪
Cache::setConfig(
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪'long', [

↪'className
↪' =>
↪'Cake\
↪Cache\
↪Engine\
↪FileEngine
↪',

↪'duration
↪' => '+1␣
↪week',

↪'probability
↪' => 100,
    'path'␣
↪=> CACHE .
↪ 'long' .␣
↪DS,
]);

// Usando␣
↪un objeto␣
↪construido.
↪
$objeto =␣
↪new␣
↪FileEngine(
↪$configuracion);
↪
Cache::setConfig(
↪'otro',
↪$objeto);
```

Los nombres de estas configuraciones de motor ("short" y "long") se utilizan como el parámetro `$config` para *Cake\Cache\Cache::write()* y *Cake\Cache\Cache::read()*. Al configurar los motores de caché, puedes referenciar el nombre de la clase utilizando las siguientes sintaxis:

```
// Nombre␣
↪corto (en␣
↪App\ o en␣
↪los␣
↪espacios␣
↪de␣
↪nombres␣
↪de Cake)
Cache::setConfig(
↪'long', [
↪'className
↪' => 'File
↪']);
```

(continué en la próxima página)

(proviene de la página anterior)

```
// Nombre␣
↪corto del␣
↪plugin
Cache::setConfig(
↪'long', [
↪'className
↪' =>
↪'MyPlugin.
↪SuperCache
↪']);

// Espacio␣
↪de␣
↪nombres␣
↪completo
Cache::setConfig(
↪'long', [
↪'className
↪' =>
↪'Cake\
↪Cache\
↪Engine\
↪FileEngine
↪']);

// Un␣
↪objeto␣
↪que␣
↪implementa␣
↪CacheEngineInterface
Cache::setConfig(
↪'long', [
↪'className
↪' =>
↪$miCache]);
↪
```

Nota: Al utilizar FileEngine, es posible que necesites usar la opción `mask` para asegurarte de que los archivos de caché se creen con los permisos correctos.

Opciones del Motor

Cada motor acepta las siguientes opciones:

- `duration`: especifica cuánto tiempo duran los elementos en esta configuración de caché. Se especifica como una expresión compatible con `strtotime()`.
- `groups`: lista de grupos o "etiquetas" asociados a cada clave almacenada en esta configuración. Útil cuando necesitas eliminar un subconjunto de datos de una caché.
- `prefix`: se antepone a todas las entradas. Bueno cuando necesitas compartir un espacio de claves con

otra configuración de caché o con otra aplicación.

- `probability`: probabilidad de activar una limpieza de la caché. Establecerlo en 0 deshabilitará automáticamente la llamada a `Cache::gc()`

Opciones del Motor de FileEngine

FileEngine utiliza las siguientes opciones específicas del motor:

- `isWindows`: se rellena automáticamente con si el host es Windows o no.
- `lock`: ¿deberían bloquearse los archivos antes de escribir en ellos?
- `mask`: la máscara utilizada para los archivos creados.

- `path`: ruta donde deben guardarse los archivos de caché. Por defecto, es el directorio temporal del sistema.

Opciones del Motor RedisEngine

RedisEngine utiliza las siguientes opciones específicas del motor:

- `port`: el puerto en el que se está ejecutando tu servidor Redis.
- `host`: el host en el que se está ejecutando tu servidor Redis.
- `database`: el número de base de datos a usar para la conexión.
- `password`: contraseña del servidor Redis.
- `persistent`: ¿se debe realizar

una conexión persistente a Redis?

- `timeout`: tiempo de espera de conexión para Redis.
- `unix_socket`: ruta a un socket Unix para Redis.

Opciones del Motor MemcacheEngine

- `compress`: si comprimir datos o no.
- `username`: usuario para acceder al servidor Memcache.
- `password`: contraseña para acceder al servidor Memcache.
- `persistent`: el nombre de la conexión persistente. Todas las configuraciones

que usan el mismo valor persistente compartirán una única conexión subyacente.

- `serialize`: el motor de serialización utilizado para serializar datos. Los motores disponibles son php, igbinary y json. Además de php, la extensión memcached debe estar compilada con el soporte adecuado para el serializador correspondiente.
- `servers`: cadena o array de servidores memcached. Si es un array, MemcacheEngine los usará como un

grupo.

- `duration`: ten en cuenta que cualquier duración mayor de 30 días se tratará como un valor de tiempo Unix real en lugar de un desfase desde el tiempo actual.
- `options`: opciones adicionales para el cliente memcached. Debe ser un array de opción => valor. Usa las constantes \ `Memcached::OPT_*` como claves.

Configuración de la Caída de Caché

En caso de que un motor no esté disponible, como el `FileEngine` que intenta escribir en una carpeta no escribible o el `RedisEngine` que no puede conectarse a Redis, el motor volverá al `NullEngine` y generará un error que se puede registrar. Esto evita que la aplicación genere una excepción no capturada debido a un error de caché.

Puedes configurar las configuraciones de la caché para que vuelvan a una configuración especificada usando la clave de configuración `fallback`:

```
Cache::setConfig(
↪'redis', [

↪'className
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪' =>
↪'Redis',

↪'duration
↪' => '+1␣
↪hours',
    'prefix
↪' =>
↪'cake_
↪redis_',
    'host'␣
↪=> '127.0.
↪0.1',
    'port'␣
↪=> 6379,

↪'fallback
↪' =>
↪'default',
]);
```

Si falla la inicialización de la instancia `RedisEngine`, la configuración de caché `redis` volverá a usar la configuración de caché `default`. Si también falla la inicialización del motor para la configuración de caché `default` en este escenario, el motor volvería nuevamente al `NullEngine` y evitaría que la aplicación genere una excepción no capturada.

Puedes desactivar las caídas de caché con `false`:

```
Cache::setConfig(
↪'redis', [

↪'className
↪' =>
↪'Redis',

↪'duration
↪' => '+1␣
↪hours',
    'prefix
↪' =>
↪'cake_
↪redis_',
    'host'␣
↪=> '127.0.
↪0.1',
    'port'␣
↪=> 6379,

↪'fallback
↪' => false
]);
```

Cuando no hay una caída, los errores de caché se generarán como excepciones.

Eliminación de Motores de Caché Configurados

static Cake\Cache\Cache::**dr**

Una vez que se crea una configuración, no puedes cambiarla. En su lugar, debes eliminar la configuración y volver a crearla usando *Cake\Cache\Cache::drop()* y *Cake\Cache\Cache::setConfig()*. Eliminar un motor de caché eliminará la configuración y destruirá el adaptador si se construyó.

Escritura en Caché

static Cake\Cache\Cache::**wr**

`Cache::write()` escribirá un $valor en la caché. Puedes leer o eliminar este valor más tarde refiriéndote a él por `$clave`. Puedes especificar una configuración opcional para almacenar la caché también. Si no se especifica ninguna `$configuración`, se usará la predeterminada. `Cache::write()` puede almacenar cualquier tipo de objeto y es ideal para almacenar resultados de búsquedas de modelos:

```
$entradas =␣
↪Cache::read(
↪'entradas
↪');
if (
↪$entradas␣
↪=== null)
↪{

↪$entradas␣
↪=
↪$servicio-
↪>
↪obtenerTodasLasEntradas()
↪

  ␣
↪Cache::write(
↪'entradas
↪',
↪$entradas);
↪
}
```

Usar `Cache::write()` y `Cache::read()` para reducir el número de consultas realizadas a la base de datos para obtener las entradas.

Nota: Si planeas almacenar en caché el resultado de las consultas realizadas con el ORM de CakePHP, es mejor utilizar las capacidades de almacenamiento en caché integradas del objeto de consulta como se describe en la sección de *Utilizando la Caché para Almacenar Resultados Comunes de Consultas*

Escritura de Múltiples Claves a la Vez

static Cake\Cache\Cache::**wr**

Puede que necesites escribir múltiples claves de caché a la vez. Aunque podrías usar múltiples llamadas a `write()`, `writeMany()` permite a CakePHP utilizar API de almacenamiento más eficientes cuando están disponibles. Por ejemplo, usando `writeMany()` ahorras múltiples conexiones de red cuando usas Memcached:

```
$resultado␣
↪=␣
↪Cache::writeMany([

↪'articulo-
↪' . $slug␣
↪=>
↪$articulo,

↪'articulo-
↪' . $slug␣
↪. '-
↪comentarios
↪' =>
↪$comentarios
]);

//
↪$resultado␣
↪contendrá
['articulo-
↪primer-
↪post' =>␣
↪true,
↪'articulo-
↪primer-
↪post-
↪comentarios
↪' => true]
```

Escrituras Atómicas

static Cake\Cache\Cache::ad

Usar `Cache::add()` te permitirá establecer atómicamente una clave en un valor si la clave aún no existe en la caché. Si la clave ya existe en el backend de la caché o la escritura falla, `add()` devolverá `false`:

```
// Establecer una clave para actuar como bloqueo
$resultado = Cache::add($claveBloqueo, true);
if (!$resultado) {
    return;
}
// Realizar una acción donde solo puede haber un proceso activo a la vez.

// Eliminar la clave de bloqueo.
Cache::delete($claveBloqueo);
```

Advertencia:
La caché basada en archivos no admite escrituras atómicas.

Caché de Lectura Directa

static Cake\Cache\Cache::**re**

La caché ayuda con la caché de lectura directa. Si la clave de caché nombrada existe, se devolverá. Si la clave no existe, se invocará la función de llamada y los resultados se almacenarán en la caché en la clave proporcionada.

Por ejemplo, a menudo quieres cachear los resultados de las llamadas a servicios remotos. Puedes usar `remember()` para hacerlo simple:

```
class␣
↪ServicioDeAsunto
{
    public␣
↪function␣
↪todasLasTemas(
↪$repositorio)
    {
        ␣
↪return␣
↪Cache::remember(
↪$repositorio␣
↪. '-temas
↪',␣
↪function␣
↪() use (
↪$repositorio)␣
↪{
            ␣
↪return
↪$this->
↪obtenerTodos(
↪$repositorio);
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪
        });
    }
}
```

Lectura Desde la Caché

static Cake\Cache\Cache::**re**

`Cache::read()` se usa para leer el valor en caché almacenado bajo `$clave` desde la `$configuración`. Si `$configuración` es nulo, se usará la configuración predeterminada configuración. `Cache::read()` devolverá el valor en caché si es una caché válida o `null` si la caché ha caducado o no existe. Utiliza los operadores de comparación estricta === o !== para comprobar el éxito de la operación `Cache::read()`.

Por ejemplo:

```
$nube =␣
↪Cache::read(
↪'nube');
if ($nube !
↪== null) {
    return
↪$nube;
}

// Generar␣
↪datos de␣
↪la nube
// ...

//␣
↪Almacenar␣
↪datos en␣
↪la caché
Cache::write(
↪'nube',
↪$nube);

return
↪$nube;
```

O si estás usando otra configuración de caché llamada `corta`, puedes especificarlo en las llamadas a `Cache::read()` y `Cache::write()` de la siguiente manera:

```
// Leer la␣
↪clave
↪"nube",␣
↪pero de␣
↪la␣
↪configuración␣
↪corta en␣
↪lugar de␣
↪la␣
↪predeterminada
$nube =␣
↪Cache::read(
↪'nube',
↪'corta');
if ($nube␣
↪=== null)
↪{
    //␣
↪Generar␣
↪datos de␣
↪la nube
    // ...

    //␣
↪Almacenar␣
↪datos en␣
↪la caché,␣
↪usando la␣
↪configuración␣
↪de caché␣
↪corta en␣
↪lugar de␣
↪la␣
↪predeterminada
  ␣
↪Cache::write(
↪'nube',
↪$nube,
↪'corta');
}

return
↪$nube;
```

Lectura de Múltiples Claves a la Vez

static Cake\Cache\Cache::**re**

Después de haber escrito múltiples claves a la vez, probablemente querrás leerlas también. Aunque podrías usar múltiples llamadas a `read()`, `readMany()` permite a CakePHP utilizar API de almacenamiento más eficientes donde estén disponibles. Por ejemplo, usando `readMany()` ahorras múltiples conexiones de red cuando usas Memcached:

```
$resultado␣
↪=␣
↪Cache::readMany([

↪'articulo-
↪' . $slug,

↪'articulo-
↪' . $slug␣
↪. '-
↪comentarios
↪'
]);
//
↪$resultado␣
↪contendrá
['articulo-
↪primer-
↪post' =>
↪'...',
↪'articulo-
↪primer-
↪post-
↪comentarios
↪' => '...
↪']
```

Eliminación de la Caché

static Cake\Cache\Cache::de

`Cache::delete()` te permitirá eliminar completamente un objeto en caché del almacén:

```
// Eliminar␣
↪una clave
Cache::delete(
↪'mi_clave
↪');
```

A partir de la versión 4.4.0, el `RedisEngine` también proporciona un método `deleteAsync()` que utiliza la operación `UNLINK` para eliminar las claves de caché:

```
Cache::pool(
↪'redis')->
↪deleteAsync(
↪'mi_clave
↪');
```

Eliminación de Múltiples Claves a la Vez

static Cake\Cache\Cache::de

Después de haber escrito múltiples claves a la vez, es posible que desees eliminarlas. Aunque podrías usar múltiples llamadas a `delete()`, `deleteMany()` permite a CakePHP utilizar API de almacenamiento más eficientes donde estén disponibles. Por ejemplo, usando `deleteMany()` ahorras múltiples conexiones de red cuando usas Memcached:

```
$resultado␣
↪=␣
↪Cache::deleteMany([

↪'articulo-
↪' . $slug,
```

(continué en la próxima página)

(proviene de la página anterior)

```
    ↪'articulo-
    ↪' . $slug␣
    ↪. '-
    ↪comentarios
    ↪'
]);
//
↪$resultado␣
↪contendrá
['articulo-
↪primer-
↪post' =>␣
↪true,
↪'articulo-
↪primer-
↪post-
↪comentarios
↪' => true]
```

Limpieza de Datos en Caché

static Cake\Cache\Cache::**cl**

Elimina todos los valores en caché para una configuración de caché. En motores como: Apcu, Memcached, se utiliza el prefijo de la configuración de caché para eliminar entradas de caché. Asegúrate de que las diferentes configuraciones de caché tengan diferentes prefijos:

```
//␣
↪Eliminará␣
↪todas las␣
↪claves.
Cache::clear();
↪
```

A partir de la versión 4.4.0, el `RedisEngine` también proporciona un método `clearBlocking()` que utiliza la operación `UNLINK` para eliminar las claves de caché:

```
Cache::pool(
↪'redis')->
↪clearBlocking();
↪
```

Nota: Debido a que APCu utiliza cachés aisladas para el servidor web y la interfaz de línea de comandos, deben ser limpiadas por separado (la CLI no puede limpiar el servidor web y viceversa).

Uso de Caché para Almacenar Contadores

static Cake\Cache\Cache::in

static Cake\Cache\Cache::de

Los contadores en tu aplicación son buenos candidatos para ser almacenados en caché. Por ejemplo, una contador de días para un evento puede ser guardado en la caché. La clase Cache expone formas de incrementar y decrementar los valores del contador. El hecho de que estas operaciones sean atómicas es importante para que se reduzca el riesgo de contención y la abilidad de que dos usuarios simultaneamente incrementen o decrementen el mismo valor.

Después de guardar un valor entero en la caché, puedes manipularlo usando `increment()` y `decrement()`:

```
Cache::write(
↪'initial_
↪count',␣
↪10);

// Later on
Cache::decrement(
↪'initial_
↪count');

// Or
Cache::increment(
↪'initial_
↪count');
```

Nota: Recuerda que las operaciones de incremento y decremento no están disponibles en FileEngine. Debes usar APCu, Redis o Memcached.

Utilizando la Caché para Almacenar Resultados Comunes de Consultas

Puedes mejorar significativamente el rendimiento de tu aplicación almacenando en caché los resultados que rara vez cambian o que están sujetos a lecturas frecuentes. Un ejemplo perfecto de esto son los resultados de `Cake\ORM\Table::find()`. El objeto de consulta te permite almacenar en caché los resultados utilizando el método `cache()`. Consulta la sección *Utilizando la Caché para Almacenar Resultados Comunes de Consultas* para obtener más información.

Uso de Grupos

A veces querrás marcar varias entradas en caché para que pertenezcan a cierto grupo o espacio de nombres. Esta es una necesidad común para invalidar masivamente claves cada vez que cambia alguna información que se comparte entre todas las entradas en el mismo grupo. Esto es posible declarando los grupos en la configuración de la caché:

```
Cache::setConfig(
↪'site_home
↪', [

↪'className
↪' =>
↪'Redis',

↪'duration
↪' =>
↪'+999 days
↪',
    'groups
↪' => [
↪'comment',
↪ 'article
↪'],
]);
```

Cake\Cache\Cache::**clearGrou**

Digamos que quieres almacenar en caché el HTML generado para tu página de inicio, pero también quieres invalidar automáticamente esta caché cada vez que se agrega un comentario o una publicación a tu base de datos. Al agregar los grupos `comment` y `article`, hemos etiquetado efectivamente cualquier clave almacenada en esta configuración de caché con ambos nombres de grupo.

Por ejemplo, cada vez que se añade una nueva publicación, podríamos decirle al motor de caché que elimine todas las entradas asociadas al grupo `article`:

```
// src/
↪Model/
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪Table/
↪ArticlesTable.
↪php
public␣
↪function␣
↪afterSave(
↪$event,
↪$entity,
↪$options␣
↪= [])
{
    if (
↪$entity->
↪isNew()) {
        ␣
↪Cache::clearGroup(
↪'article',
↪ 'site_
↪home');
    }
}
```

static Cake\Cache\Cache::**gr**

`groupConfigs()` se puede utilizar para recuperar la asignación entre el grupo y las configuraciones, es decir, tener el mismo grupo:

```
// src/
↪Model/
↪Table/
↪ArticlesTable.
↪php

/**
 * Una␣
↪variación␣
↪del␣
↪ejemplo␣
↪anterior␣
↪que␣
↪limpia␣
↪todas las␣
↪configuraciones␣
↪de caché
 * que␣
↪tienen el␣
↪mismo␣
↪grupo
 */
public␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪function␣
↪afterSave(
↪$event,
↪$entity,
↪$options␣
↪= [])
{
    if (
↪$entity->
↪isNew()) {

↪$configs␣
↪=␣
↪Cache::groupConfigs(
↪'article
↪');
      ␣
↪foreach (
↪$configs[
↪'article
↪'] as
↪$config) {
          ␣
↪Cache::clearGroup(
↪'article',
↪ $config);
        }
    }
}
```

Los grupos se comparten en todas las configuraciones de caché que utilizan el mismo motor y el mismo prefijo. Si estás usando grupos y quieres aprovechar la eliminación de grupos, elige un prefijo común para todas tus configuraciones.

Habilitar o Deshabilitar Globalmente la Caché

static
Cake\Cache\
Cache::**disable**

Puede que necesites deshabilitar todas las lecturas y escrituras en la caché cuando intentas resolver problemas relacionados con la expiración de la caché. Puedes hacerlo usando `enable()` y `disable()`:

```
//␣
↪Deshabilitar␣
↪todas las␣
↪lecturas␣
↪y␣
↪escrituras␣
↪en la␣
↪caché.
```

(continué en la próxima página)

(proviene de la página anterior)

```
Cache::disable();
```

Una vez deshabilitada, todas las lecturas y escrituras devolverán `null`.

static
Cake\Cache\
Cache::**enable**

Una vez deshabilitada, puedes usar `enable()` para habilitar nuevamente la caché:

```
// Habilitar nuevamente todas las lecturas y escrituras en la caché.
Cache::enable();
```

static
Cake\Cache\
Cache::**enabled**

Si necesitas verificar el estado de la caché, puedes usar `enabled()`.

Creación de un Motor de Caché

Puedes proporcionar motores de `Cache` personalizados en `App\Cache\Engine`, así como en plugins usando `$plugin\Cache\Engine`. Los motores de caché deben estar en un directorio de caché. Si tuvieras un motor de caché llamado `MyCustomCacheEngine`, se colocaría en **src/Cache/Engine/MyCustomCacheEngine.php**. O en **plugins/MyPlugin/src/Cache/Engine/MyCustomCacheEngine.php** como parte de un plugin. Las configuraciones de caché de los plugins deben utilizar la sintaxis de puntos del plugin:

```
Cache::setConfig('custom', [

'className' => 'MyPlugin.MyCustomCache',
    // ...
]);
```

Los motores de caché personalizados deben extender *Cake\Cache\CacheEngine*, que define varios métodos abstractos y también proporciona algunos métodos de inicialización.

La API requerida para un CacheEngine es

class Cake\Cache\CacheEngine

La clase base para todos los motores de caché utilizados con Cache.

Cake\Cache\CacheEngine::wri

Devuelve booleano para indicar el éxito.

Escribe el valor de una clave en la caché, devuelve `true` si los datos se almacenaron correctamente, `false` en caso de fallo.

Cake\Cache\CacheEngine::rea

Devuelve El valor en caché o `null` en

caso de fallo.

Lee una clave de la caché. Devuelve `null` para indicar que la entrada ha caducado o no existe.

`Cake\Cache\CacheEngine::del`

Devuelve Booleano `true` en caso de éxito.

Elimina una clave de la caché. Devuelve `false` para indicar que la entrada no existía o no se pudo eliminar.

`Cake\Cache\CacheEngine::cle`

Devuelve Booleano `true` en caso de éxito.

Elimina todas las claves de la caché. Si $check es `true`, debes validar que cada valor realmente ha caducado.

Cake\Cache\CacheEngine::cle

Devuelve
Booleano `true` en caso de éxito.

Elimina todas las claves de la caché pertenecientes al mismo grupo.

Cake\Cache\CacheEngine::dec

Devuelve
Booleano `true` en caso de éxito.

Decrementa un número bajo la

clave y devuelve el valor decrecido.

Cake\Cache\CacheEngine::inc

Devuelve Booleano `true` en caso de éxito.

Incrementa un número bajo la clave y devuelve el valor incrementado.

CAPÍTULO 14

Depuración

La depuración es una parte inevitable y necesaria de cualquier ciclo de desarrollo. Aunque CakePHP no ofrece ninguna herramienta que se conecte directamente con algún IDE o editor, CakePHP proporciona varias herramientas para asistirte en la depuración y exponer lo que se está ejecutando bajo el capó de tu aplicación.

Depuración Básica

debug(*mixed $var*, *boolean $showHtml = null*, *$showFrom = true*)

La función `debug()` es una función que está disponible globalmente y funciona de manera similar a la función `print_r()` de PHP. La función `debug()` te permite mostrar el contenido de una variable de varias maneras. Primero, si deseas que los datos se muestren de una forma amigable con HTML, debes establecer el segundo parámetro en `true`. La función también imprime la línea y el archivo de origen por defecto.

El resultado de esta función solo se mostrará si la variable `$debug` en el archivo core es `true`.

Ver también `dd()`, `pr()` y `pj()`.

stackTrace()

La función `stackTrace()` está disponible globalmente, esta permite mostrar el seguimiento de pila donde sea que se llame.

breakpoint()

Si tienes *Psysh <https://psysh.org/>* _ instalado, puedes usar esta función en entornos CLI para abrir una consola interactiva con el ámbito local actual:

```
// Algún
↪código
eval(breakpoint());
↪
```

Abrirá una consola interactiva que puede ser usada para revisar variables locales y ejecutar otro código. Puedes salir del depurador interactivo y reanudar la ejecución original corriendo `quit` o `q` en la sesion interactiva.

Usando La Clase Debugger

class Cake\Error**Debugger**

Para usar el depurador, primero asegúrate de que `Configure::read('debug')` sea `true`.

Imprimiendo Valores

static Cake\Error\Debugger

Dump imprime el contenido de una variable. Imprimirá todas las propiedades y métodos (si existen) de la variable que se le pase:

```
$foo = [1,2,
↪3];

Debugger::dump(
↪$foo);

// Salida
array(
    1,
    2,
    3
)

// Objeto
↪simple
$car = new
↪Car();

Debugger::dump(
↪$car);
```

(continué en la próxima página)

(proviene de la página anterior)

```
// Salida
object(Car)
↪{
    color =>
↪ 'red'
    make =>
↪'Toyota'
    model =>
↪ 'Camry'
    mileage␣
↪=>␣
↪(int)15000
}
```

Enmascarando Datos

Al volcar datos con `Debugger` o mostrar páginas de error, es posible que desees ocultar claves sensibles como contraseñas o claves API. En tu `config/bootstrap.php` puedes enmascarar claves específicas:

```
Debugger::setOutputMask([

↪'password
↪' =>
↪'xxxxx',
    'awsKey
↪' =>
↪'yyyyy',
]);
```

Registros Con Trazas De Pila

```
static Cake\Error\Debugger
```

Crea un registro de seguimiento de pila detallado al momento de la invocación. El método `log()` imprime datos similar a como lo hace `Debugger::dump()`, pero al debug.log en vez de al buffer de salida. Ten en cuenta que tu directorio **tmp** (y su contenido) debe ser reescribible por el servidor web para que `log()` funcione correctamente.

Generando seguimientos de pila

static Cake\Error\Debugger

Devuelve el seguimiento de pila actual. Cada línea de la pila incluye cuál método llama, incluyendo el archivo y la línea en la que se originó la llamada:

```
// En␣
↪PostsController::index()
pr(Debugger::trace());
↪

// Salida
PostsController::index()␣
↪- APP/
↪Controller/
↪DownloadsController.
↪php, line␣
↪48
Dispatcher::_
↪invoke() -
↪ CORE/src/
↪Routing/
↪Dispatcher.
↪php, line␣
↪265
Dispatcher::dispatch()␣
↪- CORE/
↪src/
↪Routing/
↪Dispatcher.
↪php, line␣
↪237
[main] -␣
↪APP/
↪webroot/
↪index.php,
↪ line 84
```

Arriba está el seguimiento de pila generado al llamar `Debugger::trace()` en una acción de un controlador. Leer el seguimiento de pila desde abajo hacia arriba muestra el órden de las funciones (cuadros de pila).

Obtener Un Extracto De Un Archivo

static Cake\Error\Debugger

Saca un extracto de un archivo en $path (el cual es una dirección absoluta), resalta el número de la línea $line con el número $context de líneas alrededor de este.

```
pr(Debugger::excerpt(ROOT␣
↪. DS .␣
↪LIBS .
↪'debugger.
↪php', 321,
↪ 2));

// Mostrará␣
↪lo␣
↪siguiente.
Array
(
    [0] =>
↪<code>
↪<span␣
↪style=
↪"color:
↪#000000">␣
↪* @access␣
↪public</
↪span></
↪code>
    [1] =>
↪<code>
↪<span␣
↪style=
↪"color:
↪#000000">␣
↪*/</span>
↪</code>
    [2] =>
↪<code>
↪<span␣
↪style=
↪"color:
↪#000000">␣
↪  ␣
↪function␣
↪excerpt(
↪$file,
↪$line,
↪$context␣
↪= 2) {</
↪span></
↪code>

    [3] =>
↪<span␣
↪class=
↪"code-
↪highlight
↪"><code>
↪<span␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪style=
↪"color:
↪#000000">␣
↪
↪$data =
↪$lines =␣
↪array();</
↪span></
↪code></
↪span>
    [4] =>
↪<code>
↪<span␣
↪style=
↪"color:
↪#000000">␣
↪
↪$data =␣
↪@explode(
↪"\n",␣
↪file_get_
↪contents(
↪$file));</
↪span></
↪code>
)
```

Aunque este método es usado internamente, puede ser útil si estás creando tus propios mensajes de error o entradas de registros para situaciones customizadas.

static Cake\Error\Debugger

Consigue el tipo de una variable. Los objetos devolverán el nombre de su clase.

Usando El Registro Para Depurar

Registrar mensajes es otra buena manera de depurar aplicaciones, puedes usar *Cake\Log\Log* para hacer registros en tu aplicación. Todos los objetos que usen `LogTrait` tienen una instancia del método `log()` que puede ser usado para registrar mensajes:

```
$this->log(
↪'Llegó␣
↪aquí',
↪'debug');
```

Lo anterior escribiría `Llegó aquí` en el registro de depuración. Puedes usar entradas de registro para ayudar a los métodos de depuración que involucran redireccionamientos o búcles complejos. También puedes usar *Cake\Log\Log::write()* para escribir mensajes de registro. Este método puede ser llamado estáticamente en cualquier lugar de tu aplicación que un Log haya sido cargado:

```
// En el␣
↪tope del␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪archivo␣
↪que␣
↪quieras␣
↪hacer␣
↪registros.
use Cake\
↪Log\Log;

// En␣
↪cualquier␣
↪parte que␣
↪Log haya␣
↪sido␣
↪importado.
Log::debug(
↪'Llegó␣
↪aquí');
```

Kit De Depuración

DebugKit es un complemento que proporciona una serie de buenas herramientas de depuración. Principalmente, provee una barra de herramientas en el HTML renderizado, que porporciona una gran cantidad de información sobre tu aplicación y la solicitud actual. Ver el capítulo *Debug Kit* para saber cómo instalar y usar DebugKit.

CAPÍTULO 15

Despliegue

Una vez que tu aplicación esté lista para ser desplegada, hay algunas cosas que debes hacer.

Mover archivos

Puedes clonar tu repositorio en tu servidor de producción y luego seleccionar la revisión/etiqueta que deseas ejecutar. Luego, ejecuta `composer install`. Aunque esto requiere un cierto conocimiento sobre git y una instalación existente de `git` y `composer`, este proceso se encargará de las dependencias de las bibliotecas y los permisos de archivos y carpetas.

Ten en cuenta que al desplegar a través de FTP deberás corregir los permisos de archivo y carpeta.

También puedes utilizar esta técnica de despliegue para configurar un servidor de pruebas o demostración (preproducción) y mantenerlo sincronizado con tu entorno local.

Ajustar la configuración

Querrás hacer algunos ajustes en la configuración de tu aplicación para un entorno de producción. El valor de `debug` es extremadamente importante. Al desactivar debug = `false` se deshabilitan una serie de características de desarrollo que no deberían ser expuestas a Internet en general. Deshabilitar debug cambia las siguientes características:

- Los mensajes de depuración, creados con `pr()`, `debug()` y `dd()`,

están deshabilitados.

- La duración de las cachés básicas de CakePHP se establece en 365 días, en lugar de 10 segundos, como en desarrollo.
- Las vistas de errores son menos informativas y se muestran páginas de error genéricas en lugar de mensajes de error detallados con trazas de pila.
- Los avisos y errores de PHP no se muestran.

Además de lo anterior, muchos complementos y extensiones de la aplicación usan `debug` para modificar su comportamiento.

Puedes utilizar una variable de entorno para establecer dinámicamente el nivel de depuración entre entornos. Esto evitará desplegar una aplicación con debug `true` y también te ahorrará tener que cambiar el nivel de depuración cada vez antes de desplegar en un entorno de producción.

Por ejemplo, puedes establecer una variable de entorno en tu configuración de Apache:

```
SetEnv␣
↪CAKEPHP_
↪DEBUG 1
```

Y luego puedes establecer dinámicamente el nivel de depuración en **app_local.php**:

```
$debug =␣
↪(bool)getenv(
↪'CAKEPHP_
↪DEBUG');

return [
    'debug'␣
↪=> $debug,
    .....
];
```

Se recomienda que coloques la configuración que se comparte en todas los entornos de tu aplicación en **config/app.php**. Para la configuración que varía entre entornos, utiliza **config/app_local.php** o variables de entorno.

Verificar tu Seguridad

Si estás lanzando tu aplicación al mundo, es una buena idea asegurarte de que no tenga ningun problema de seguridad obvio:

- Asegúrate de estar usando el componente o middleware *Middleware CSRF*.
- Puedes habilitar el componente *FormProtection*. Puede ayudar a prevenir varios tipos de manipulación de formularios y reducir la posibilidad de problemas de asignación masiva.

- Asegúrate de que tus modelos tengan las reglas de *Validation* correctas habilitadas.
- Verifica que solo tu directorio `webroot` sea públicamente visible y que tus secretos (como tu sal de aplicación y cualquier clave de seguridad) sean privados y únicos también.

Establecer la Raíz (Document Root)

Establecer correctamente la raíz en tu aplicación es un paso importante para mantener tanto tu código como tu aplicación seguros. Las aplicaciones de CakePHP deben tener la raíz establecida en el `webroot` de la aplicación. Esto hace que los archivos de aplicación y configuración sean inaccesibles a través de una URL. Establecer la raíz es diferente para diferentes servidores web. Consulta la documentación de *URL Rewriting* para obtener información específica del servidor web.

En todos los casos, querrás establecer la raiz del host virtual/dominio en `webroot/`. Esto elimina la posibilidad de que se ejecuten archivos fuera del directorio raíz.

Mejora el Rendimiento de tu Aplicación

La carga de clases puede llevarse una gran parte del tiempo de procesamiento de tu aplicación. Para evitar este problema, se recomienda que ejecutes este comando en tu servidor de producción una vez que la aplicación esté implementada:

```
php␣
↪composer.
↪phar␣
↪dumpautoload␣
↪-o
```

Dado que manejar los archivos estáticos, como imágenes, archivos JavaScript y CSS de los complementos, a través del `Dispatcher` es increíblemente ineficiente, se recomienda encarecidamente crear enlaces simbólicos para producción. Esto se puede hacer usando el comando `plugin`:

```
bin/cake␣
↪plugin␣
↪assets␣
↪symlink
```

El comando anterior creará enlaces simbólicos del directorio `webroot` de todos los complementos cargados a la ruta adecuada en el directorio `webroot` de la aplicación.

Si tu sistema de archivos no permite crear enlaces simbólicos, los directorios se copiarán en lugar de enlazarse. También puedes copiar explícitamente los directorios usando:

```
bin/cake␣
↪plugin␣
↪assets␣
↪copy
```

CakePHP utiliza internamente `assert()` para proporcionar comprobación de tipos en tiempo de ejecución y proporcionar mejores mensajes de error durante el desarrollo. Puedes hacer que PHP omita estas comprobaciones `assert()` actualizando tu `php.ini` para incluir:

```
;␣
↪Desactivar␣
↪la␣
↪generación␣
↪de código␣
↪assert().
zend.
↪assertions␣
↪= -1
```

Omitir la generación de código para `assert()` proporcionará un rendimiento de ejecución más rápido, y se recomienda para aplicaciones que tienen una buena cobertura de pruebas o que están usando un analizador estático.

Desplegar una actualización

En cada implementación es probable que tengas algunas tareas para coordinar en tu servidor web. Algunas tareas típicas son:

1. Instalar dependencias con `composer install`. Evita usar `composer update` al hacer implementaciones, ya que podrías obtener versiones inesperadas de paquetes.
2. Ejecutar migraciones de base de datos con el complemento Migrations u otra herramienta.
3. Limpiar la caché del esquema del modelo con `bin/cake schema_cache clear`. La página *Herramienta de esquemas de caché (Schema Cache)* tiene más infor-

mación sobre este comando.

CAPÍTULO 16

Mailer

class Cake\Mailer**Mailer**(*st*
$p
fi-
le
=
nu

`Mailer` es una clase de conveniencia para enviar correos electrónicos. Con esta clase, puedes enviar correos electrónicos desde cualquier lugar dentro de tu aplicación.

Uso Básico

Primero, asegúrate de que la clase esté cargada:

```
use Cake\
↪Mailer\
↪Mailer;
```

Después de cargar `Mailer`, puedes enviar un correo electrónico de la siguiente manera:

```
$mailer =␣
↪new␣
↪Mailer(
↪'default
↪');
$mailer->
↪setFrom([
↪'me@example.
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪com' =>
↪'Mi Sitio
↪'])
    ->setTo(
↪'you@example.
↪com')
    ->
↪setSubject(
↪'Acerca de
↪')
    ->
↪deliver(
↪'Mi␣
↪mensaje');
```

Dado que los métodos setter de `Mailer` devuelven una instancia de la clase, puedes configurar sus propiedades encadenando los métodos.

`Mailer` tiene varios métodos para definir destinatarios: `setTo()`, `setCc()`, `setBcc()`, `addTo()`, `addCc()` y `addBcc()`. La principal diferencia es que los primeros tres sobrescribirán lo que ya se haya establecido, mientras que los últimos simplemente agregarán más destinatarios a su campo respectivo:

```
$mailer =␣
↪new␣
↪Mailer();
$mailer->
↪setTo(
↪'to@example.
↪com',
↪'Destinatario␣
↪Ejemplo');
$mailer->
↪addTo(
↪'to2@example.
↪com',
↪'Destinatario2␣
↪Ejemplo');
// Los␣
↪destinatarios␣
↪del␣
↪correo␣
↪electrónico␣
↪son:␣
↪to@example.
↪com y␣
↪to2@example.
↪com
$mailer->
↪setTo(
↪'test@example.
↪com',
↪'DestinatarioPrueba␣
↪Ejemplo');
```

(continué en la próxima página)

(proviene de la página anterior)

```
// El␣
↪destinatario␣
↪del␣
↪correo␣
↪electrónico␣
↪es:␣
↪test@example.
↪com
```

Elección del Remitente

Cuando envíes correos electrónicos en nombre de otras personas, suele ser una buena idea definir el remitente original usando el encabezado del remitente (Sender header). Puedes hacerlo usando `setSender()`:

```
$mailer =␣
↪new␣
↪Mailer();
$mailer->
↪setSender(
↪'app@example.
↪com', 'Mi␣
↪aplicación␣
↪de correo
↪');
```

Nota: También es una buena idea establecer el remitente del sobre (envelope sender) al enviar correos electrónicos en nombre de otra persona. Esto evita que reciban mensajes sobre la entregabilidad.

Configuración

Los perfiles de Mailer y las configuraciones de transporte de correo electrónico se definen en los archivos de configuración de tu aplicación. Las claves `'Email'` y `'EmailTransport'` definen perfiles de Mailer y configuraciones de transporte de correo electrónico respectivamente. Durante el inicio de la aplicación, los valores de configuración se pasan desde `Configure` a las clases `Mailer` y `TransportFactory` utilizando `setConfig()`. Al definir perfiles y transportes, puedes mantener el código de tu aplicación libre de datos de configuración y evitar la duplicación que complica el mantenimiento y el despliegue.

Para cargar una configuración predefinida, puedes usar el método `setProfile()` o pasarlo al constructor de `Mailer`:

```
$mailer =␣
↪new␣
↪Mailer();
$mailer->
↪setProfile(
↪'default
↪');

// O en el␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪constructor
$mailer =␣
↪new␣
↪Mailer(
↪'default
↪');
```

En lugar de pasar una cadena que coincida con un nombre de configuración preestablecido, también puedes cargar simplemente un array de opciones:

```
$mailer =␣
↪new␣
↪Mailer();
$mailer->
↪setProfile([
↪'from' =>
↪'me@example.
↪org',
↪'transport
↪' => 'my_
↪custom']);

// O en el␣
↪constructor
$mailer =␣
↪new␣
↪Mailer([
↪'from' =>
↪'me@example.
↪org',
↪'transport
↪' => 'my_
↪custom']);
```

Perfiles de Configuración

Definir perfiles de entrega te permite consolidar la configuración común del correo electrónico en perfiles reutilizables. Tu aplicación puede tener tantos perfiles como sea necesario. Se utilizan las siguientes claves de configuración:

- `'from'`: Mailer o array del remitente. Ver `Mailer::setFrom()`.
- `'sender'`: Mailer o array del remitente real. Ver `Mailer::setSender()`.

- `'to'`: Mailer o array del destino. Ver `Mailer::setTo()`.
- `'cc'`: Mailer o array de copia carbono. Ver `Mailer::setCc()`.
- `'bcc'`: Mailer o array de copia carbono oculta. Ver `Mailer::setBcc()`.
- `'replyTo'`: Mailer o array para responder al correo electrónico. Ver `Mailer::setReplyTo()`.
- `'readReceipt'`: Dirección del Mailer o un array de direcciones para recibir el recibo de lectura. Ver `Mailer::setReadReceipt`
- `'returnPath'`: Dirección del Mailer o un array de direcciones para devolver si

hay algún error. Ver `Mailer::setReturnPath(`

- `'messageId'`: ID del mensaje del correo electrónico. Ver `Mailer::setMessageId()`
- `'subject'`: Asunto del mensaje. Ver `Mailer::setSubject()`.
- `'message'`: Contenido del mensaje. No establezcas este campo si estás usando contenido renderizado.
- `'priority'`: Prioridad del correo electrónico como valor numérico (generalmente de 1 a 5, siendo 1 el más alto).
- `'headers'`: Cabeceras a incluir. Ver `Mailer::setHeaders()`.
-

'viewRenderer': Si estás usando contenido renderizado, establece el nombre de la clase de vista. Ver ViewBuilder::setClassN

- 'template': Si estás usando contenido renderizado, establece el nombre de la plantilla. Ver ViewBuilder::setTempla
- 'theme': Tema utilizado al renderizar la plantilla. Ver ViewBuilder::setTheme(
- 'layout': Si estás usando contenido renderizado, establece el diseño a renderizar. Ver ViewBuilder::setTempla
- 'autoLayout': Si quieres renderizar una plantilla sin diseño, establece este

campo en `false`. Ver `ViewBuilder::disableAu`

- `'viewVars'`: Si estás usando contenido renderizado, establece el array con variables que se utilizarán en la vista. Ver `Mailer::setViewVars()`.
- `'attachments'`: Lista de archivos para adjuntar. Ver `Mailer::setAttachments`
- `'emailFormat'`: Formato del correo electrónico (html, texto o ambos). Ver `Mailer::setEmailFormat`
- `'transport'`: Nombre de la configuración del transporte. Ver *Configuración de Transportes*.
- `'log'`: Nivel de registro para re-

gistrar las cabeceras y el mensaje del correo electrónico. `true` utilizará LOG_DEBUG. Ver *Usando Niveles*. Ten en cuenta que los registros se emitirán bajo el ámbito denominado `email`. Ver también *Ámbitos de Registro (scope)*.

- `'helpers'`: Array de helpers utilizados en la plantilla del correo electrónico. `ViewBuilder::setHelper`

Nota: Los valores de las claves mencionadas anteriormente que usan Mailer o array, como from, to, cc, etc., se pasarán como el primer parámetro de los métodos correspondientes. El equivalente a: `$mailer->setFrom('mi@example.com', 'Mi Sitio')` se definiría como `'from' => ['mi@example.com' => 'Mi Sitio']` en tu configuración.

Configurando Cabeceras

En `Mailer`, eres libre de establecer las cabeceras que desees. No olvides agregar el prefijo `X-` a tus cabeceras personalizadas.

Consulta `Mailer::setHeaders()` y `Mailer::addHeaders()`

Envío de Correos Electrónicos con Plantillas

Los correos electrónicos a menudo son mucho más que un simple mensaje de texto. Para facilitar eso, CakePHP proporciona una forma de enviar correos electrónicos utilizando la *capa de vista* de CakePHP.

Las plantillas para correos electrónicos residen en una carpeta especial `templates/email` de tu aplicación. Las vistas del Mailer también pueden utilizar diseños y elementos al igual que las vistas normales:

```
$mailer = new Mailer();
$mailer->setEmailFormat('html')->setTo('bob@example.com')->setFrom('app@domain.com')->viewBuilder()->setTemplate('bienvenida')->setLayout('elegante');

$mailer->deliver();
```

Lo anterior utilizará **templates/email/html/bienvenida.php** para la vista y **templates/layout/email/html/elegante.php** para el diseño. También puedes enviar mensajes de correo electrónico con varias partes de plantilla:

```
$mailer =␣
↪new␣
↪Mailer();
$mailer
                ␣
↪->
↪setEmailFormat(
↪'both')
                ␣
↪->setTo(
↪'bob@example.
↪com')
                ␣
↪->setFrom(
↪'app@domain.
↪com')
                ␣
↪->
↪viewBuilder()
                ␣
↪    ->
↪setTemplate(
↪'bienvenida
↪')
                ␣
↪    ->
↪setLayout(
↪'elegante
↪');

$mailer->
↪deliver();
```

Esto utilizará los siguientes archivos de plantilla:

- **templates/email/text/bienvenida.p**
- **templates/layout/email/text/elegan**
- **templates/email/html/bienvenida.**
- **templates/layout/email/html/elega**

Cuando envíes correos electrónicos con plantillas, tienes la opción de enviar `texto`, `html` o `ambos`.

Puedes configurar toda la configuración relacionada con la vista usando la instancia de creador de vistas `Mailer::viewBuilder()` de manera similar a como lo haces en el controlador.

Puedes establecer variables de vista con `Mailer::setViewVars()`:

```
$mailer =␣
↪new␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪Mailer(
↪'plantilla
↪');
$mailer->
↪setViewVars([
↪'valor' =>
↪ 12345]);
```

O puedes usar los métodos del creador de vistas `ViewBuilder::setVar()` y `ViewBuilder::setVars()`.

En tus plantillas de correo electrónico, puedes usarlos de la siguiente manera:

```
<p>Aquí␣
↪está tu␣
↪valor: <b>
↪<?=
↪$valor ?>
↪</b></p>
```

También puedes usar ayudantes en los correos electrónicos, al igual que en los archivos de plantilla normales. De forma predeterminada, solo se carga el `HtmlHelper`. Puedes cargar ayudantes adicionales utilizando el método `ViewBuilder::addHelpers()`:

```
$mailer->
↪viewBuilder()-
↪>
↪addHelpers([
↪'Html',
↪'Custom',
↪'Text']);
```

Cuando agregues ayudantes, asegúrate de incluir "Html" o se eliminará de los ayudantes cargados en tu plantilla de correo electrónico.

Nota: En versiones anteriores a 4.3.0, deberás usar `setHelpers()` en su lugar.

Si deseas enviar correos electrónicos utilizando plantillas en un plugin, puedes usar la familiar *Sintaxis de plugin* para hacerlo:

```
$mailer =␣
↪new␣
↪Mailer();
$mailer->
↪viewBuilder()-
↪>
↪setTemplate(
↪'Blog.new_
↪comment');
```

Lo anterior utilizará la plantilla y el diseño del plugin Blog como ejemplo.

En algunos casos, es posible que necesites anular la plantilla predeterminada proporcionada por los complementos. Puedes hacer esto usando temas:

```
$mailer->viewBuilder()->setTemplate('Blog.new_comment')->setLayout('Blog.auto_message')->setTheme('MiTema');
```

Esto te permite anular la plantilla «new_comment» en tu tema sin modificar el complemento Blog. El archivo de plantilla debe crearse en la siguiente ruta: **templates/plugin/MiTema/plugin/Blog/email/text/new_comment.php**.

Envío de Archivos Adjuntos

Cake\Mailer\Mailer::**setAtta**

También puedes adjuntar archivos a los mensajes de correo electrónico. Hay algunos formatos diferentes dependiendo del tipo de archivos que tengas y de cómo quieras que aparezcan los nombres de archivo en el cliente de correo del destinatario:

1. Array: `$mailer->setAttachment('ruta/completa/archivo.png'])` adjuntará este archivo con el nombre archivo.png..
2. Array con clave: `$mailer->setAttachment('png' => '/ruta/completa/algun_hash.png'])` adjuntará some_hash.png con el nombre foto.png.

El destinatario verá foto.png, no some_hash.png.

3. Arrays anidados:

```
↪$mailer-
↪>
↪setAttachments([
↪'foto.
↪png'␣
↪=> [
    ␣
↪
↪'archivo
↪' =>
↪'/
↪ruta/
↪completa/
↪algun_
↪hash.
↪png',
    ␣
↪
↪'mimetype
↪' =>
↪'image/
↪png',
    ␣
↪
↪'contentId
↪' =>
↪'mi-
↪id-
↪unico
↪',
  ],
]);
```

Lo anterior adjuntará el archivo con un tipo MIME diferente y con un ID de contenido

personalizado (cuando se establece el ID de contenido, el archivo adjunto se convierte en incrustado). El tipo MIME y contentId son opcionales en esta forma.

3.1. Cuando estás usando el `contentId`, puedes usar el archivo en el cuerpo HTML como `<img src="cid:mi-id-conteni`

3.2. Puedes usar la opción `contentDisposition` para desactivar el encabezado `Content-Disposition` para un archivo adjunto. Esto es útil cuando envías invitaciones ical a clientes que usan Outlook.

3.3 En

lugar de la opción `archivo`, puedes proporcionar el contenido del archivo como una cadena utilizando la opción `datos`. Esto te permite adjuntar archivos sin necesidad de tener rutas de archivo para ellos.

Relajando las Reglas de Validación de Direcciones

`Cake\Mailer\Mailer::setEmai`

Si tienes problemas de validación al enviar a direcciones no conformes, puedes relajar el patrón utilizado para validar direcciones de correo electrónico. Esto es a veces necesario al tratar con algunos proveedores de servicios de Internet:

```
$mailer =␣
↪new␣
↪Mailer(
↪'predeterminado
↪');

// Relaja␣
↪el patrón␣
↪de correo␣
↪electrónico,
↪ para que␣
↪puedas␣
↪enviar
// a␣
↪direcciones␣
↪no␣
↪conformes.
$mailer->
↪setEmailPattern(
↪$nuevoPatrón);
↪
```

Envío de Correos Electrónicos desde la CLI

Cuando envíes correos electrónicos dentro de un script de CLI (Shells, Tasks, ...), debes establecer manualmente el nombre de dominio que Mailer utilizará. Servirá como el nombre de host para el ID del mensaje (ya que no hay un nombre de host en un entorno CLI):

```
$mailer->
↪setDomain(
↪'www.
↪ejemplo.
↪org');
// Da como␣
↪resultado␣
↪IDs de␣
↪mensajes␣
↪como ``
↪<UUID@www.
↪ejemplo.
↪org>``␣
↪(válidos)
// En lugar␣
↪de ``
↪<UUID@>``␣
↪(inválidos)
```

Un ID de mensaje válido puede ayudar a evitar que los correos electrónicos terminen en carpetas de spam.

Creación de Correos Electrónicos Reutilizables

Hasta ahora hemos visto cómo usar directamente la clase `Mailer` para crear y enviar un correo electrónico. Pero la característica principal del mailer es permitir la creación de correos electrónicos reutilizables en toda tu aplicación. También se pueden usar para contener múltiples configuraciones de correo electrónico en un solo lugar. Esto ayuda a mantener tu código DRY y a evitar la configuración de correo electrónico en otras áreas de tu aplicación.

En este ejemplo, crearemos un `Mailer` que contiene correos electrónicos relacionados con el usuario. Para crear nuestro `UserMailer`, crea el archivo **src/Mailer/UserMailer.php**. El contenido del archivo debería verse así:

```
namespace␣
↪App\
↪Mailer;

use Cake\
↪Mailer\
↪Mailer;

class␣
↪UserMailer␣
↪extends␣
↪Mailer
{
    public␣
↪function␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪welcome(
↪$user)
    {

↪$this
                ␣
↪->setTo(
↪$user->
↪email)
                ␣
↪->
↪setSubject(sprintf(
↪'Welcome
↪%s',
↪$user->
↪name))
                ␣
↪->
↪viewBuilder()
                ␣
↪    ->
↪setTemplate(
↪'welcome_
↪mail'); //
↪ Por␣
↪defecto,␣
↪se␣
↪utiliza␣
↪la␣
↪plantilla␣
↪con el␣
↪mismo␣
↪nombre␣
↪que el␣
↪nombre␣
↪del␣
↪método.
    }

    public␣
↪function␣
↪resetPassword(
↪$user)
    {

↪$this
                ␣
↪->setTo(
↪$user->
↪email)
                ␣
↪->
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪setSubject(
↪'Reset␣
↪password')
            ␣
↪->
↪setViewVars([
↪'token' =>
↪ $user->
↪token]);
    }
}
```

En nuestro ejemplo, hemos creado dos métodos, uno para enviar un correo electrónico de bienvenida y otro para enviar un correo electrónico de restablecimiento de contraseña. Cada uno de estos métodos espera una entidad de usuario y utiliza sus propiedades para configurar cada correo electrónico.

Ahora podemos usar nuestro `UserMailer` para enviar nuestros correos electrónicos relacionados con el usuario desde cualquier parte de nuestra aplicación. Por ejemplo, si queremos enviar nuestro correo de bienvenida podríamos hacer lo siguiente:

```
namespace␣
↪App\
↪Controller;
↪

use Cake\
↪Mailer\
↪MailerAwareTrait;
↪

class␣
↪UsersController␣
↪extends␣
↪AppController
{
    use␣
↪MailerAwareTrait;
↪

    public␣
↪function␣
↪register()
    {

↪$user =

 $this->
↪Users->
↪newEmptyEntity();
↪
        if (
↪$this->
↪request->
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪is('post
↪')) {

↪$user =
↪$this->
↪Users->
↪patchEntity(
↪$user,
↪$this->
↪request->
↪getData());
↪
                ␣
↪if ($this-
↪>Users->
↪save(
↪$user)) {
                ␣
↪    //␣
↪Enviar␣
↪correo␣
↪electrónico␣
↪de␣
↪bienvenida.
↪
                ␣
↪    $this-
↪>
↪getMailer(
↪'User')->
↪send(
↪'welcome',
↪ [$user]);
                ␣
↪    //␣
↪Redirigir␣
↪a la␣
↪página de␣
↪inicio de␣
↪sesión u␣
↪otra␣
↪página de␣
↪destino.
                ␣
↪   ␣
↪return
↪$this->
↪redirect([
↪'controller
↪' =>
↪'Users',
↪'action'␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪=> 'login
↪']);
                ␣
↪}

↪$this->
↪Flash->
↪error(__(
↪'Unable␣
↪to␣
↪register␣
↪user.␣
↪Please␣
↪try again.
↪'));
        }

↪$this->
↪set(compact(
↪'user'));
    }
}
```

Si quisiéramos separar por completo el envío del correo de bienvenida del usuario de nuestro código de aplicación, podemos hacer que nuestro *UserMailer* se suscriba al evento *Model.afterSave*. Al suscribirse a un evento, podemos mantener nuestras clases relacionadas con el usuario completamente libres de lógica e instrucciones relacionadas con el correo electrónico de nuestra aplicación. Por ejemplo, podríamos agregar lo siguiente a nuestro *UserMailer*:

```
public␣
↪function␣
↪implementedEvents()
{
    return [

↪'Model.
↪afterSave
↪' =>
↪'onRegistration
↪',
    ];
}

public␣
↪function␣
↪onRegistration(EventInte
↪$event,␣
↪EntityInterface␣
↪$entity,␣
↪ArrayObject␣
↪$options)
{
    if (
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪$entity->
↪isNew()) {

↪$this->
↪send(
↪'welcome',
↪ [
↪$entity]);
    }
}
```

Ahora puedes registrar el mailer como un oyente de eventos y el método *onRegistration()* se invocará cada vez que se dispare el evento *Model.afterSave*:

```
// Adjuntar␣
↪al gestor␣
↪de␣
↪eventos␣
↪de␣
↪Usuarios
$this->
↪Users->
↪getEventManager()-
↪>on($this-
↪>
↪getMailer(
↪'User'));
```

Configuración de Transportes

Los mensajes de correo electrónico se entregan mediante transportes. Diferentes transportes te permiten enviar mensajes a través de la función *mail()* de PHP, servidores SMTP o no enviarlos en absoluto, lo cual es útil para depurar. Configurar transportes te permite mantener los datos de configuración fuera del código de tu aplicación y simplifica la implementación, ya que simplemente puedes cambiar los datos de configuración. Una configuración de transporte de ejemplo se ve así:

```
// En␣
↪config/
↪app.php

↪'EmailTransport
↪' => [
    //␣
↪Configuración␣
↪de␣
↪ejemplo␣
↪para␣
↪correo
    'default
↪' => [
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪'className
↪' => 'Mail
↪',
    ],
    //␣
↪Configuración␣
↪de␣
↪ejemplo␣
↪para SMTP
    'gmail'␣
↪=> [

↪'host' =>
↪'smtp.
↪gmail.com
↪',

↪'port' =>␣
↪587,

↪'username
↪' =>
↪'mi@gmail.
↪com',

↪'password
↪' =>
↪'secreto',

↪'className
↪' => 'Smtp
↪',
        'tls
↪' => true,
    ],
],
```

Los transportes también se pueden configurar en tiempo de ejecución utilizando *TransportFactory::setConfig()*:

```
use Cake\
↪Mailer\
↪TransportFactory;
↪

// Definir␣
↪un␣
↪transporte␣
↪SMTP
TransportFactory::setConfig
↪'gmail', [
    'host'␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪=> 'ssl://
↪smtp.
↪gmail.com
↪',
    'port'␣
↪=> 465,

↪'username
↪' =>
↪'mi@gmail.
↪com',

↪'password
↪' =>
↪'secreto',

↪'className
↪' => 'Smtp
↪'
]);
```

Puedes configurar servidores SMTP SSL, como Gmail. Para hacerlo, coloca el prefijo *ssl://* en el host y configura el valor del puerto en consecuencia. También puedes habilitar SMTP TLS usando la opción *tls*:

```
use Cake\
↪Mailer\
↪TransportFactory;
↪

TransportFactory::setConfig
↪'gmail', [
    'host'␣
↪=> 'smtp.
↪gmail.com
↪',
    'port'␣
↪=> 587,

↪'username
↪' =>
↪'mi@gmail.
↪com',

↪'password
↪' =>
↪'secreto',

↪'className
↪' => 'Smtp
↪',
    'tls' =>
↪ true
```

(continué en la próxima página)

(proviene de la página anterior)

```
]);
```

La configuración anterior habilitaría la comunicación TLS para los mensajes de correo electrónico.

Para configurar tu mailer para usar un transporte específico, puedes usar el método `Cake\Mailer\Mailer::setTransport()` o tener el transporte en tu configuración:

```
// Usa un transporte con nombre ya configurado usando TransportFactory::setCon
$mailer->setTransport('gmail');

// Usa un objeto construido.
$mailer->setTransport(new \Cake\Mailer\Transport\DebugTransport());
```

Advertencia: Deberás tener habilitado el acceso para aplicaciones menos seguras en tu cuenta de Google para que funcione: Permitir que aplicaciones menos seguras accedan a tu cuenta[128].

[128] https://support.google.com/accounts/answer/6010255

Nota: Configuración SMTP de Gmail[129].

Nota: Para usar SSL + SMTP, necesitarás tener SSL configurado en tu instalación de PHP.

También se pueden proporcionar opciones de configuración como una cadena *DSN*. Esto es útil cuando trabajas con variables de entorno o proveedores de *PaaS*:

```
TransportFactory::setConfig
↪'default',
↪ [
    'url' =>
↪ 'smtp://
↪mi@gmail.
↪com:secreto@smtp.
↪gmail.
↪com:587?
↪tls=true',
]);
```

Cuando usas una cadena DSN, puedes definir cualquier parámetro / opción adicional como argumentos de cadena de consulta.

```
static Cake\Mailer\Mailer::
```

Una vez configurados, los transportes no se pueden modificar. Para modificar un transporte, primero debes eliminarlo y luego reconfigurarlo.

Creación de Transportes Personalizados

Puedes crear tus propios transportes para situaciones como enviar correos electrónicos utilizando servicios como SendGrid, MailGun o Postmark. Para crear tu transporte, primero crea el archivo **src/Mailer/Transport/ExampleTransport.php** (donde Example es el nombre de tu transporte). Para empezar, tu archivo debería verse así:

```
namespace␣
↪App\
↪Mailer\
↪Transport;

use Cake\
↪Mailer\
↪AbstractTransport;
↪
use Cake\
↪Mailer\
↪Message;

class␣
↪ExampleTransport␣
```

(continué en la próxima página)

[129] https://support.google.com/a/answer/176600?hl=es

(proviene de la página anterior)

```
↪extends␣
↪AbstractTransport
{
    public␣
↪function␣
↪send(Message␣
↪
↪$message):␣
↪array
    {
        //␣
↪Haz algo.
    }
}
```

Debes implementar el método `send(Message $message)` con tu lógica personalizada.

Envío de correos electrónicos sin usar Mailer

El `Mailer` es una clase de abstracción de nivel superior que actúa como un puente entre las clases `Cake\Mailer\Message`, `Cake\Mailer\Renderer` y `Cake\Mailer\AbstractTransport` para configurar correos electrónicos con una interfaz fluida.

Si lo deseas, también puedes usar estas clases directamente con el `Mailer`.

Por ejemplo:

```
$render =␣
↪new \Cake\
↪Mailer\
↪Renderer();
↪
$render->
↪viewBuilder()
    ->
↪setTemplate(
↪'custom')
    ->
↪setLayout(
↪'sparkly
↪');

$message =␣
↪new \Cake\
↪Mailer\
↪Message();
$message
    ->
↪setFrom(
↪'admin@cakephp.
↪org')
```

(continué en la próxima página)

(proviene de la página anterior)

```
    ->setTo(
↪'user@foo.
↪com')
    ->
↪setBody(
↪$render->
↪render());

$transport␣
↪= new \
↪Cake\
↪Mailer\
↪Transport\
↪MailTransport();
↪
$result =
↪$transport-
↪>send(
↪$message);
```

Incluso puedes omitir el uso del `Renderer` y establecer el cuerpo del mensaje directamente usando los métodos `Message::setBodyText()` y `Message::setBodyHtml()`.

Pruebas de Mailers

Para probar mailers, agrega `Cake\TestSuite\EmailTrait` a tu caso de prueba.El `MailerTrait` utiliza ganchos de PHPUnit para reemplazar los transportes de correo electrónico de tu aplicación con un proxy que intercepta los mensajes de correo electrónico y te permite hacer afirmaciones sobre el correo que se enviaría.

Agrega el trait a tu caso de prueba para comenzar a probar correos electrónicos, y carga rutas si tus correos electrónicos necesitan generar URL:

```
namespace␣
↪App\Test\
↪TestCase\
↪Mailer;

use App\
↪Mailer\
↪WelcomeMailer;
↪
use App\
↪Model\
↪Entity\
↪User;

use Cake\
↪TestSuite\
↪EmailTrait;
↪
use Cake\
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪TestSuite\
↪TestCase;

class␣
↪WelcomeMailerTestCase␣
↪extends␣
↪TestCase
{
    use␣
↪EmailTrait;
↪

    public␣
↪function␣
↪setUp():␣
↪void
    {
        ␣
↪parent::setUp();
↪

↪$this->
↪loadRoutes();
↪
    }
}
```

Supongamos que tenemos un mailer que envía correos electrónicos de bienvenida cuando un nuevo usuario se registra. Queremos comprobar que el asunto y el cuerpo contienen el nombre del usuario:

```
// en␣
↪nuestra␣
↪clase␣
↪WelcomeMailerTestCase.
↪
public␣
↪function␣
↪testName()
{
    $user =␣
↪new User([

↪'name' =>
↪'Alice␣
↪Alittea',

↪'email' =>
↪
↪'alice@example.
↪org',
    ]);
    $mailer␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪= new␣
↪WelcomeMailer();
↪
    $mailer-
↪>send(
↪'welcome',
↪ [$user]);

    $this->
↪assertMailSentTo(
↪$user->
↪email);
    $this->
↪assertMailContainsText(
↪'Hola ' .
↪$user->
↪name);
    $this->
↪assertMailContainsText(
↪'¡Bienvenido␣
↪a CakePHP!
↪');
}
```

Métodos de afirmación

El trait `Cake\TestSuite\EmailTrait` proporciona las siguientes afirmaciones:

```
// Asegura␣
↪que se␣
↪enviaron␣
↪un número␣
↪esperado␣
↪de␣
↪correos␣
↪electrónicos
$this->
↪assertMailCount(
↪$count);

// Asegura␣
↪que no se␣
↪enviaron␣
↪correos␣
↪electrónicos
$this->
↪assertNoMailSent();
↪

// Asegura␣
↪que se␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
    ↪envió un␣
    ↪correo␣
    ↪electrónico␣
    ↪a una␣
    ↪dirección
    $this->
    ↪assertMailSentTo(
    ↪$address);

    // Asegura␣
    ↪que se␣
    ↪envió un␣
    ↪correo␣
    ↪electrónico␣
    ↪desde una␣
    ↪dirección
    $this->
    ↪assertMailSentFrom(
    ↪$emailAddress);
    ↪
    $this->
    ↪assertMailSentFrom([
    ↪$emailAddress␣
    ↪=>
    ↪$displayName]);
    ↪

    // Asegura␣
    ↪que un␣
    ↪correo␣
    ↪electrónico␣
    ↪contiene␣
    ↪los␣
    ↪contenidos␣
    ↪esperados
    $this->
    ↪assertMailContains(
    ↪$contents);
    ↪

    // Asegura␣
    ↪que un␣
    ↪correo␣
    ↪electrónico␣
    ↪contiene␣
    ↪los␣
    ↪contenidos␣
    ↪HTML␣
    ↪esperados
    $this->
    ↪assertMailContainsHtml(
    ↪$contents);
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪

// Asegura␣
↪que un␣
↪correo␣
↪electrónico␣
↪contiene␣
↪los␣
↪contenidos␣
↪de texto␣
↪esperados
$this->
↪assertMailContainsText(
↪$contents);
↪

// Asegura␣
↪que un␣
↪correo␣
↪electrónico␣
↪contiene␣
↪el valor␣
↪esperado␣
↪dentro de␣
↪un getter␣
↪de␣
↪Message␣
↪(por␣
↪ejemplo,
↪"subject")
$this->
↪assertMailSentWith(
↪$expected,
↪
↪$parameter);
↪

// Asegura␣
↪que un␣
↪correo␣
↪electrónico␣
↪en un␣
↪índice␣
↪específico␣
↪se envió␣
↪a una␣
↪dirección
$this->
↪assertMailSentToAt(
↪$at,
↪$address);
```

(continué en la próxima página)

(proviene de la página anterior)

```
// Asegura
↪que un
↪correo
↪electrónico
↪en un
↪índice
↪específico
↪se envió
↪desde una
↪dirección
$this->
↪assertMailSentFromAt(
↪$at,
↪$address);

// Asegura
↪que un
↪correo
↪electrónico
↪en un
↪índice
↪específico
↪contiene
↪los
↪contenidos
↪esperados
$this->
↪assertMailContainsAt(
↪$at,
↪$contents);
↪

// Asegura
↪que un
↪correo
↪electrónico
↪en un
↪índice
↪específico
↪contiene
↪los
↪contenidos
↪HTML
↪esperados
$this->
↪assertMailContainsHtmlAt
↪$at,
↪$contents);
↪

// Asegura
↪que un
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪correo␣
↪electrónico␣
↪en un␣
↪índice␣
↪específico␣
↪contiene␣
↪los␣
↪contenidos␣
↪de texto␣
↪esperados
$this->
↪assertMailContainsTextAt
↪$at,
↪$contents);
↪

// Asegura␣
↪que un␣
↪correo␣
↪electrónico␣
↪contiene␣
↪un␣
↪archivo␣
↪adjunto
$this->
↪assertMailContainsAttachm
↪'test.png
↪');

// Asegura␣
↪que un␣
↪correo␣
↪electrónico␣
↪en un␣
↪índice␣
↪específico␣
↪contiene␣
↪el valor␣
↪esperado␣
↪dentro de␣
↪un getter␣
↪de␣
↪Message␣
↪(por␣
↪ejemplo,
↪"cc")
$this->
↪assertMailSentWithAt(
↪$at,
↪$expected,
↪
↪$parameter);
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪

// Asegura␣
↪que un␣
↪correo␣
↪electrónico␣
↪contiene␣
↪una␣
↪subcadena␣
↪en el␣
↪asunto.
$this->
↪assertMailSubjectContains
↪'Oferta␣
↪Gratuita
↪');

// Asegura␣
↪que un␣
↪correo␣
↪electrónico␣
↪en un␣
↪índice␣
↪específico␣
↪contiene␣
↪una␣
↪subcadena␣
↪en el␣
↪asunto.
$this->
↪assertMailSubjectContains
↪ 'Oferta␣
↪Gratuita
↪');
```

CAPÍTULO 17

Manejo de Errores y Excepciones

Las aplicaciones de CakePHP vienen con la configuración predeterminada de manejo de errores y excepciones. Los errores de PHP son capturados y mostrados o registrados. Las excepciones no capturadas se representan automáticamente en páginas de error.

Configuración

La configuración de errores se realiza en el archivo **config/app.php** de tu aplicación. Por defecto, CakePHP utiliza `Cake\Error\ErrorTrap` y `Cake\Error\ExceptionTrap` para manejar tanto errores de PHP como excepciones, respectivamente. La configuración de errores te permite personalizar el manejo de errores para tu aplicación. Las siguientes opciones son compatibles:

- `errorLevel` - int - El nivel de errores que te interesa capturar. Usa las constantes de error de PHP integradas y las máscaras de bits

para seleccionar el nivel de error que te interesa. Consulta *Advertencias de Obsolescencia* para deshabilitar advertencias de obsolescencia.

- `trace` - bool - Incluir trazas para errores en los archivos de registro. Las trazas se incluirán en el registro después de cada error. Esto es útil para encontrar dónde/cuándo se están generando los errores.
- `exceptionRenderer` - string - La clase responsable de representar excepciones no capturadas.

Si eliges una clase personalizada, debes colocar el archivo para esa clase en **src/Error**. Esta clase debe implementar un método `render()`.

- `log` - bool - Cuando es `true`, las excepciones y sus trazas se registrarán en *Cake\Log\Log*.
- `skipLog` - array - Un array de nombres de clases de excepción que no deben ser registrados. Esto es útil para eliminar mensajes de registro comunes pero poco interesantes, como NotFoundExceptions.
- `extraFatalErrorMemory`

- int - Establece el número de megabytes para aumentar el límite de memoria cuando se encuentra un error fatal. Esto permite espacio para completar el registro o el manejo de errores.

- `logger` (antes de la versión 4.4.0, usa `errorLogger`) - `Cake\Error\ErrorLoggerInterface` - La clase responsable de registrar errores y excepciones no controladas. Por defecto, es `Cake\Error\ErrorLogger`.

- `errorRenderer` - `Cake\Error\ErrorRendererInterface` - La clase responsable de representar errores.

Se elige automáticamente en función del SAPI de PHP.

- `ignoredDeprecationPath` - array - Una lista de rutas compatibles con la sintaxis Glob que deben ignorar errores de obsolescencia. Añadido en la versión 4.2.0

Por defecto, los errores de PHP se muestran cuando `debug` es `true`, y se registran cuando debug es `false`. El manejador de errores fatal se llamará independientemente del nivel de `debug` o la configuración de `errorLevel`, pero el resultado será diferente según el nivel de `debug`. El comportamiento predeterminado para errores fatales es mostrar una página de error interno del servidor (`debug` deshabilitado) o una página con el mensaje, archivo y línea (`debug` habilitado).

Nota: Si utilizas un manejador de errores personalizado, las opciones compatibles dependerán de tu manejador.

Advertencias de Obsolescencia

CakePHP utiliza advertencias de obsolescencia para indicar cuándo se ha marcado como obsoleta alguna característica. También recomendamos este sistema para su uso en tus plugins y código de aplicación cuando sea útil. Puedes activar advertencias de obsolescencia con `deprecationWarning()`:

```
deprecationWarning(
↪'5.0',
↪'El␣
↪método␣
↪example()␣
↪está␣
↪obsoleto.␣
↪Usa␣
↪getExample()␣
↪en su␣
↪lugar.');
```

Al actualizar CakePHP o plugins, es posible que te encuentres con nuevas advertencias de obsolescencia. Puedes desactivar temporalmente las advertencias de obsolescencia de varias formas:

1. Usar la configuración `Error.errorLevel` con `E_ALL ^ E_USER_DEPRECATED` para ignorar *todas* las advertencias de obsolescencia.
2. Usar la opción de configuración `Error.ignoredDeprecationPath` para ignorar advertencias con expresiones compatibles con la sintaxis Glob. Por ejemplo:

```
'Error
↪' =>␣
↪[

↪'ignoredDeprecationP
↪' =>␣
↪[
    ␣
↪
↪'vendors/
↪company/
↪contacts/
↪*',
    ␣
↪
↪'src/
↪Models/
↪*',
```

(continué en la próxima página)

(proviene de la página anterior)

```
    ],
],
```

Ignoraría todas las advertencias de obsolescencia de tu directorio `Models` y el plugin `Contacts` en tu aplicación.

Cambiar el Manejo de Excepciones

El manejo de excepciones en CakePHP ofrece varias formas de personalizar cómo se manejan las excepciones. Cada enfoque te brinda diferentes niveles de control sobre el proceso de manejo de excepciones.

1. *Escucha eventos* Esto te permite recibir notificaciones a través de eventos de CakePHP cuando se han manejado errores y excepciones.
2. *Plantillas personalizadas* Esto te permite cambiar las plantillas de vista renderizadas como lo harías con cualquier

otra plantilla en tu aplicación.

3. *Controlador personalizado* Esto te permite controlar cómo se renderizan las páginas de excepción.
4. *ExceptionRenderer personalizado* Esto te permite controlar cómo se realizan las páginas de excepción y el registro.
5. *Crea y registra tus propios manejadores* Esto te brinda control total sobre cómo se manejan, registran y representan los errores y excepciones. Utiliza `Cake\Error\ExceptionTrap` y `Cake\`

Error\ErrorTrap como referencia cuando implementes tus manejadores.

Escuchar Eventos

Los manejadores `ErrorTrap` y `ExceptionTrap` activarán eventos de CakePHP cuando manejan errores. Puedes escuchar el evento `Error.beforeRender` para ser notificado de los errores de PHP. El evento `Exception.beforeRender` se desencadena cuando se maneja una excepción:

```
$errorTrap␣
↪= new␣
↪ErrorTrap(Configure::reac
↪'Error'));
$errorTrap->
↪getEventManager()-
↪>on(
    'Error.
↪beforeRender
↪',
   ␣
↪function␣
↪(EventInterface␣
↪$event,␣
↪PhpError
↪$error) {
        //␣
↪haz lo␣
↪que␣
↪necesites
    }
);
```

Dentro de un manejador `Error.beforeRender`, tienes algunas opciones:

- Detener el evento para evitar la representación.
- Devolver una cadena para omitir la representación y

usar la cadena proporcionada en su lugar.

Dentro de un manejador `Exception.beforeRender`, también tienes algunas opciones:

- Detener el evento para evitar la representación.
- Establecer el atributo de datos `exception` con `setData('exception', $err)` para reemplazar la excepción que se está representando.
- Devolver una respuesta desde el evento para omitir la representación y usar la respuesta proporcionada en su lugar.

Plantillas Personalizadas

El atrapador de excepciones predeterminado representa todas las excepciones no capturadas que tu aplicación genera con la ayuda de `Cake\Error\WebExceptionRenderer` y tu `ErrorController` de la aplicación.

Las vistas de página de error están ubicadas en **templates/Error/**. Todos los errores 4xx usan la plantilla **error400.php**, y los errores 5xx usan la plantilla **error500.php**. Tus plantillas de error tendrán las siguientes variables disponibles:

- `message` El mensaje de la excepción.
- `code` El código de la excepción.
- `url` La URL de la solicitud.
- `error` El objeto de la excepción.

En modo de depuración, si tu error se extiende de `Cake\Core\Exception\CakeException`, los datos devueltos por `getAttributes()` se expondrán también como variables de vista.

Nota: Necesitarás establecer `debug` en falso para ver tus plantillas **error404** y **error500**. En modo de depuración, verás la página de error de desarrollo de CakePHP.

Diseño Personalizado para la Página de Error

Por defecto, las plantillas de error usan **templates/layout/error.php** para un diseño. Puedes usar la propiedad `layout` para elegir un diseño diferente:

```
// dentro de templates/Error/error400.php
$this->layout = 'mi_error';
```

Lo anterior usaría **templates/layout/mi_error.php** como el diseño para tus páginas de error.

Muchas excepciones generadas por CakePHP representarán plantillas de vista específicas en modo de depuración. Con la depuración desactivada, todas las excepciones generadas por CakePHP usarán **error400.php** o **error500.php** según su código de estado.

Controlador Personalizado

La clase `App\Controller\ErrorController` se utiliza para la representación de excepciones de CakePHP para renderizar la vista de la página de error y recibe todos los eventos estándar del ciclo de vida de la solicitud. Al modificar esta clase, puedes controlar qué componentes se utilizan y qué plantillas se representan.

Si tu aplicación utiliza *rutas con prefijo*, puedes crear controladores de error personalizados para cada prefijo de enrutamiento. Por ejemplo, si tienes un prefijo `Admin`, podrías crear la siguiente clase:

```
namespace␣
↪App\
↪Controller\
↪Admin;

use App\
↪Controller\
↪AppController;
↪
use Cake\
↪Event\
↪EventInterface;
↪

class␣
↪ErrorController␣
↪extends␣
↪AppController
{
    /**
     *␣
↪Callback␣
↪beforeRender.
↪
     *
     *␣
↪@param \
↪Cake\
↪Event\
↪EventInterface␣
↪$event␣
↪Evento.
     *␣
↪@return␣
↪void
     */
    public␣
↪function␣
↪beforeRender(EventInterfa
↪$event)
    {

↪$this->
↪viewBuilder()-
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪>
↪setTemplatePath(
↪'Error');
    }
}
```

Este controlador solo se utilizaría cuando se encuentra un error en un controlador con prefijo y te permite definir lógica/plantillas específicas del prefijo según sea necesario.

ExceptionRenderer Personalizado

Si deseas controlar todo el proceso de representación y registro de excepciones, puedes utilizar la opción `Error.exceptionRenderer` en **config/app.php** para elegir una clase que representará las páginas de excepciones. Cambiar el ExceptionRenderer es útil cuando quieres cambiar la lógica utilizada para crear un controlador de error, elegir la plantilla o controlar el proceso general de representación.

Tu clase personalizada de ExceptionRenderer debe colocarse en **src/Error**. Supongamos que nuestra aplicación usa `App\Exception\MissingWidgetException` para indicar un widget faltante. Podríamos crear un ExceptionRenderer que represente páginas de error específicas cuando se maneja este error:

```
// En src/
↪Error/
↪AppExceptionRenderer.
↪php
namespace␣
↪App\Error;

use Cake\
↪Error\
↪WebExceptionRenderer;
↪

class␣
↪AppExceptionRenderer␣
↪extends␣
↪WebExceptionRenderer
{
    public␣
↪function␣
↪missingWidget(
↪$error)
    {

↪$response␣
↪= $this->
↪controller-
↪>
↪getResponse();
↪


        ␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
    return $response->withStringBody('Oops, ese widget está perdido.');
    }
}

// En config/app.php
'Error' => [

    'exceptionRenderer' => 'App\Error\AppExceptionRenderer',
    // ...
],
// ...
```

Lo anterior manejaría nuestro `MissingWidgetException`, y nos permitiría proporcionar lógica de visualización/manejo personalizado para esas excepciones de aplicación.

Los métodos de representación de excepciones reciben la excepción manejada como argumento y deben devolver un objeto `Response`. También puedes implementar métodos para agregar lógica adicional al manejar errores de CakePHP:

```
// En src/Error/AppExceptionRenderer.php
namespace App\Error;

use Cake\Error\WebExceptionRenderer;

class AppExceptionRenderer extends WebExceptionRenderer
{
    public function notFound(
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪$error)
    {
        //␣
↪Haz algo␣
↪con␣
↪objetos␣
↪NotFoundException.
↪
    }
}
```

Cambiar la Clase ErrorController

El ExceptionRenderer dicta qué controlador se utiliza para la representación de excepciones. Si quieres cambiar qué controlador se utiliza para representar excepciones, puedes anular el método `_getController()` en tu ExceptionRenderer:

```
   // en␣
↪src/Error/
↪AppExceptionRenderer
 ␣
↪namespace␣
↪App\Error;

   use App\
↪Controller\
↪SuperCustomErrorControlle
↪
   use Cake\
↪Controller\
↪Controller;
↪
   use Cake\
↪Error\
↪WebExceptionRenderer;
↪

   class␣
↪AppExceptionRenderer␣
↪extends␣
↪WebExceptionRenderer
   {
      ␣
↪protected␣
↪function _
↪getController():␣
↪Controller
       {
          ␣
↪return␣
↪new␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪SuperCustomErrorControlle
↪
        }
    }

    // en␣
↪config/
↪app.php
    'Error'␣
↪=> [

↪'exceptionRenderer
↪' => 'App\
↪Error\
↪AppExceptionRenderer
↪',
        // ..
↪.
    ],


// ...
```

Crear tus Propias Excepciones de Aplicación

Puedes crear tus propias excepciones de aplicación utilizando cualquiera de las excepciones SPL incorporadas[130], `Exception` en sí, o *`Cake\Core\Exception\Exception`*. Si tu aplicación contiene la siguiente excepción:

```
use Cake\
↪Core\
↪Exception\
↪CakeException;
↪

class␣
↪MissingWidgetException␣
↪extends␣
↪CakeException
{
}
```

Podrías proporcionar errores de desarrollo detallados, creando **templates/Error/missing_widget.php**. Cuando estás en modo de producción, el error anterior se trataría como un error 500 y usaría la plantilla **error500**.

Las excepciones que son subclases de `Cake\Http\Exception\HttpException`, usarán su código de error como código de estado HTTP si el código de error está entre `400` y `506`.

El constructor para `Cake\Core\Exception\CakeException` te permite pasar datos adicionales. Estos datos adicionales se interpolan en el `_messageTemplate`. Esto te permite crear excepciones ricas en datos que proporcionen más contexto sobre tus errores:

[130] https://php.net/manual/en/spl.exceptions.php

```
use Cake\
↪Core\
↪Exception\
↪CakeException;
↪

class␣
↪MissingWidgetException␣
↪extends␣
↪Exception
{
    // Los␣
↪datos del␣
↪contexto␣
↪se␣
↪interpolan␣
↪en esta␣
↪cadena de␣
↪formato.
   ␣
↪protected
↪$_
↪messageTemplate␣
↪= 'Parece␣
↪que falta
↪%s.';

    //␣
↪También␣
↪puedes␣
↪establecer␣
↪un código␣
↪de␣
↪excepción␣
↪predeterminado.
↪
   ␣
↪protected
↪$_
↪defaultCode␣
↪= 404;
}

throw new␣
↪MissingWidgetException([
↪'widget'␣
↪=>
↪'Puntiagudo
↪']);
```

Cuando se representa, tu plantilla de vista tendría una variable `$widget` establecida. Si lanzas la excepción como una cadena o usas su método `getMessage()`, obtendrás `Parece que falta Puntiagudo.`.

Nota: Antes de CakePHP 4.2.0, usa la clase `Cake\Core\Exception\Exception` en lugar de `Cake\Core\Exception\CakeException`

Registro de Excepciones

Usando el manejo de excepciones incorporado, puedes registrar todas las excepciones que son tratadas por ErrorTrap configurando la opción `log` en `true` en tu **config/app.php**. Al habilitar esto, se registrarán todas las excepciones en *Cake\Log\Log* y en los registradores configurados.

Nota: Si estás utilizando un manejador de excepciones personalizado, esta configuración no tendrá ningún efecto, a menos que la referencies dentro de tu implementación.

Excepciones Incorporadas para CakePHP

Excepciones HTTP

Hay varias excepciones incorporadas en CakePHP, además de las excepciones internas del framework, hay varias excepciones para métodos HTTP.

exception Cake\Http\Exception**BadRequestException**

Usado para el error 400 Bad Request.

exception Cake\Http\Exception**UnauthorizedException**

Usado para el error 401

Unauthorized.

exception Cake\Http\Exception**ForbiddenException**

Usado para el error 403 Forbidden.

exception Cake\Http\Exception**InvalidCsrfTokenException**

Usado para el error 403 causado por un token CSRF inválido.

exception Cake\Http\Exception**NotFoundException**

Usado pa-

ra el error 404 Not found.

exception Cake\Http\Exception**MethodNotAllowedException**

Usado para el error 405 Method Not Allowed.

exception Cake\Http\Exception**NotAcceptableException**

Usado para el error 406 Not Acceptable.

exception Cake\Http\Exception**ConflictException**

Usado para el error

	409 Conflict.
exception Cake\Http\Exception**GoneException**	Usado para el error 410 Gone.

Para más detalles sobre los códigos de estado 4xx del protocolo HTTP, consulta **RFC 2616#section-10.4**[131].

exception Cake\Http\Exception**InternalErrorException**	Usado para el error 500 Internal Server Error.
exception Cake\Http\Exception**NotImplementedException**	Usado para el error 501

[131] https://datatracker.ietf.org/doc/html/rfc2616.html#section-10.4

	Not Implemented Errors.
exception Cake\Http\Exception**ServiceUnavailableException**	Usado para el error 503 Service Unavailable.

Para más detalles sobre los códigos de estado 5xx del protocolo HTTP, consulta **RFC 2616#section-10.5**[132].

Puedes lanzar estas excepciones desde tus controladores para indicar estados de error o errores HTTP. Un ejemplo de uso de las excepciones HTTP podría ser renderizar páginas 404 para los elementos que no se han encontrado:

```
use Cake\
↪Http\
↪Exception\
↪NotFoundException;
↪

public␣
↪function␣
↪ver($id =␣
↪null)
{

↪$articulo␣
↪= $this->
↪Articulos-
↪>findById(
↪$id)->
↪first();
    if␣
↪(empty(
↪$articulo))␣
```

(continué en la próxima página)

[132] https://datatracker.ietf.org/doc/html/rfc2616.html#section-10.5

(proviene de la página anterior)

```
↪{
        ␣
↪throw new␣
↪NotFoundException(_
↪_(
↪'Artículo␣
↪no␣
↪encontrado
↪'));
    }
    $this->
↪set(
↪'articulo
↪',
↪$articulo);
↪
    $this->
↪viewBuilder()-
↪>
↪setOption(
↪'serialize
↪', ['
```

articulo"]);
}

Usar excepciones para errores HTTP te permite mantener tu código limpio y dar respuestas RESTful a aplicaciones de clientes y usuarios.

Uso de Excepciones HTTP en tus Controladores

Puedes lanzar cualquiera de las excepciones relacionadas con HTTP desde las acciones de tu controlador para indicar estados de error. Por ejemplo:

```
use Cake\
↪Network\
↪Exception\
↪NotFoundException;
↪

public␣
↪function␣
↪ver($id =␣
↪null)
{

↪$articulo␣
↪= $this->
↪Articulos-
↪>findById(
↪$id)->
↪first();
```

(continué en la próxima página)

(proviene de la página anterior)

```
    if␣
↪(empty(
↪$articulo))␣
↪{
        ␣
↪throw new␣
↪NotFoundException(_
↪_(
↪'Artículo␣
↪no␣
↪encontrado
↪'));
    }
    $this->
↪set(
↪'articulo
↪',
↪'articulo
↪');
    $this->
↪viewBuilder()-
↪>
↪setOption(
↪'serialize
↪', [
↪'articulo
↪']);
}
```

Lo anterior causaría que el manejador de excepciones configurado capture y procese la *NotFoundException*. Por defecto, esto creará una página de error y registrará la excepción.

Otras Excepciones Incorporadas

Además, CakePHP utiliza las siguientes excepciones:

exception
Cake\View\
Exception\
MissingViewException

No se pudo encontrar la clase de

vista elegida.

exception Cake\View\Exception**MissingTemplateException**

No se pudo encontrar el archivo de plantilla elegido.

exception Cake\View\Exception**MissingLayoutException**

No se pudo encontrar el diseño elegido.

exception Cake\View\Exception**MissingHelperException**

No se pudo encontrar el ayudante elegido.

exception Cake\View\Exception**MissingElementException**

No se pudo encontrar el archivo de elemento elegido.

exception Cake\View\Exception**MissingCellException**

No se pudo encontrar la

clase de celda elegida.

exception Cake\View\Exception**MissingCellViewException**

No se pudo encontrar el archivo de vista de celda elegido.

exception Cake\Controller\Exception**MissingComponentException**

No se pudo encontrar el componente

configurado.

exception
Cake\
Controller\
Exception\
MissingActionException

No se pudo encontrar la acción del controlador solicitada.

exception
Cake\
Controller\
Exception\
PrivateActionException

Acceder a acciones con prefijos privados/protegidos/_.

exception
Cake\
Console\
Exception\
ConsoleException

Una clase de biblioteca de consola encontró un error.

exception
Cake\
Database\
Exception\
MissingConnectionException

Falta una conexión de modelo.

exception
Cake\
Database\
Exception\
MissingDriverException

No se pudo encon-

trar un controlador de base de datos.

exception Cake\Database\Exception**MissingExtensionException**

Falta una extensión de PHP para el controlador de base de datos.

exception Cake\ORM\Exception**MissingTableException**

No se pudo encontrar

la tabla de un modelo.

exception Cake\ORM\Exception**MissingEntityException**

No se pudo encontrar la entidad de un modelo.

exception Cake\ORM\Exception**MissingBehaviorException**

No se pudo encontrar el comportamiento de un mode-

lo.

exception Cake\ORM\Exception**PersistenceFailedException**

No se pudo guardar/eliminar una entidad al usar Cake\ORM\Table::saveOrFail(o Cake\ORM\Table::deleteOrFai

exception Cake\Datasource\Exception**RecordNotFoundException**

No se pudo encontrar el registro solicitado. Esto también establecerá las cabeceras de respuesta HTTP en 404.

exception Cake\Routing\Exception**MissingControllerException**

No

se pudo encontrar el controlador solicitado.

exception
Cake\Routing\Exception**MissingRouteException**

No se pudo hacer coincidir la URL solicitada o no se pudo analizar.

exception
Cake\Core\Exception**Exception**

Clase base de excepción en CakePHP. Todas las excepciones de capa de framework lanzadas por CakePHP extenderán esta clase.

Estas clases de excepción se extienden de *Exception*. Al extender Exception, puedes crear tus propios errores de "framework".

Cake\Core\Exception\Excepti

Consulta

```
Cake\
Network\
Request::header()
```

Todas las excepciones Http y Cake extienden la clase Exception, que tiene un método para agregar encabezados a la respuesta. Por ejemplo, al lanzar un 405 MethodNotAllowedException, el rfc2616 dice:

```
"La␣
↪respuesta␣
↪DEBE␣
↪incluir␣
↪un␣
↪encabezado␣
↪Allow que␣
↪contenga␣
↪una lista␣
↪de␣
↪métodos␣
↪válidos
para el␣
↪recurso␣
↪solicitado.
↪"
```

Manejo Personalizado de Errores de PHP

Por defecto, los errores de PHP se representan en la consola o en la salida HTML, y también se registran. Si es necesario, puedes cambiar la lógica de manejo de errores de CakePHP con la tuya propia.

Registro de Errores Personalizado

Los manejadores de errores utilizan instancias de `Cake\Error\ErrorLoggingInterface` para crear mensajes de registro y registrarlos en el lugar apropiado. Puedes reemplazar el registrador de errores utilizando el valor de configuración `Error.errorLogger`. Un ejemplo de registrador de errores:

```
namespace␣
↪App\Error;

use Cake\
↪Error\
↪ErrorLoggerInterface;
↪
use Cake\
↪Error\
↪PhpError;
use Psr\
↪Http\
↪Message\
↪ServerRequestInterface;
↪
use␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪Throwable;

/**
 * Registra␣
↪errores y␣
↪excepciones␣
↪no␣
↪manejadas␣
↪en `Cake\
↪Log\Log`
 */
class␣
↪ErrorLogger␣
↪implements␣
↪ErrorLoggerInterface
{
    /**
     *␣
↪@inheritDoc
     */
    public␣
↪function␣
↪logError(
        ␣
↪PhpError
↪$error,
        ?
↪ServerRequestInterface␣
↪$request,
        ␣
↪bool
↪$includeTrace␣
↪= false
    ): void
↪{
        //␣
↪Registra␣
↪errores␣
↪de PHP
    }

    /**
     *␣
↪@inheritDoc
     */
    public␣
↪function␣
↪logException(
        ?
↪ServerRequestInterface␣
↪$request,
        ␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪bool
↪$includeTrace␣
↪= false
    ): void
↪{
        //␣
↪Registra␣
↪excepciones.
↪
    }
}
```

Renderizado Personalizado de Errores

CakePHP incluye renderizadores de errores tanto para entornos web como de consola. Sin embargo, si deseas reemplazar la lógica que renderiza los errores, puedes crear una clase personalizada:

```
// src/
↪Error/
↪CustomErrorRenderer.
↪php
namespace␣
↪App\Error;

use Cake\
↪Error\
↪ErrorRendererInterface;
↪
use Cake\
↪Error\
↪PhpError;

class␣
↪CustomErrorRenderer␣
↪implements␣
↪ErrorRendererInterface
{
    public␣
↪function␣
↪write(string␣
↪$out):␣
↪void
    {
        //␣
↪enviar el␣
↪error␣
↪renderizado␣
↪al flujo␣
↪de salida␣
↪apropiado
    }

    public␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
    ↪function␣
    ↪render(PhpError␣
    ↪$error,␣
    ↪bool
    ↪$debug):␣
    ↪string
        {
            //␣
    ↪Convertir␣
    ↪el error␣
    ↪en una␣
    ↪cadena de␣
    ↪salida.
        }
}
```

El constructor de tu renderizador recibirá un array con la configuración almacenada en *Error*. Conecta tu renderizador de errores personalizado a CakePHP a través del valor de configuración *Error.errorRenderer*. Al reemplazar el manejo de errores, deberás tener en cuenta tanto los entornos web como los de línea de comandos.

CAPÍTULO 18

Events System

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[133] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[133] https://github.com/cakephp/docs

CAPÍTULO 19

Internationalization & Localization

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[134] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[134] https://github.com/cakephp/docs

CAPÍTULO 20

Logging

Si bien la configuración de la clase Configure de CakePHP puede ayudarte a ver lo que está sucediendo en el sistema, hay momentos en los que necesitarás registrar datos en el disco para averiguar lo que está ocurriendo. Con tecnologías como SOAP, AJAX y API REST, la depuración puede ser bastante difícil.

Logging también puede ser una forma de averiguar lo que ha estado ocurriendo en tu aplicación con el tiempo. ¿Qué términos de búsqueda se están utilizando? ¿Qué tipos de errores están viendo mis usuarios? ¿Con qué frecuencia se ejecuta una consulta en particular?

Logging data in CakePHP is done with the `log()` function. It is provided by the `LogTrait`, which is the common ancestor for many CakePHP classes. If the context is a CakePHP class (Controller, Component, View,...), you can log your data. You can also use `Log::write()` directly. See *Escribiendo en los archivos de Log*.

El registro de datos en CakePHP se realiza con la función «log()». Esta función es proporcionada por el «LogTrait», que es el ancestro común de muchas clases de CakePHP. Si el contexto es una clase de CakePHP (Controlador, Componente, Vista, etc.), puedes registrar tus datos. También puedes usar «Log::write()» directamente. Consulta la sección *Escribiendo en los archivos de Log* para obtener más información.

Logging Configuration

La configuración de `Log` debe realizarse durante la fase de arranque de tu aplicación. El archivo **config/app.php** está diseñado precisamente para esto. Puedes definir tantos `loggers` como necesite tu aplicación. Los `loggers` deben configurarse utilizando la clase *`Cake\Log\Log`*. Un ejemplo sería:

```
use Cake\
↪Log\
↪Engine\
↪FileLog;
use Cake\
↪Log\Log;
```

(continué en la próxima página)

(proviene de la página anterior)

```
// Nombre␣
↪de la␣
↪clase␣
↪utilizando␣
↪la␣
↪constante
↪'class'␣
↪del␣
↪logger.
Log::setConfig(
↪'info', [

↪'className
↪' =>␣
↪FileLog::class,
↪
    'path'␣
↪=> LOGS,
    'levels
↪' => [
↪'info'],
    'file'␣
↪=> 'info',
]);

// Nombre␣
↪de clase␣
↪corto
Log::setConfig(
↪'debug', [

↪'className
↪' => 'File
↪',
    'path'␣
↪=> LOGS,
    'levels
↪' => [
↪'notice',
↪'debug'],
    'file'␣
↪=> 'debug
↪',
]);

// Fully␣
↪namespaced␣
↪name.
Log::setConfig(
↪'error', [
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪'className
↪' =>
↪'Cake\Log\
↪Engine\
↪FileLog',
    'path'␣
↪=> LOGS,
    'levels
↪' => [
↪'warning',
↪ 'error',
↪'critical
↪', 'alert
↪',
↪'emergency
↪'],
    'file'␣
↪=> 'error
↪',
]);
```

Lo anterior crea tres loggers, llamados `info`, `debug` and `error`. Cada uno está configurado para manejar diferentes niveles de mensajes. También almacenan sus mensajes de registro en archivos separados, de esta manera, podemos separar los registros de depuración/aviso/información de los errores más graves. Consulta la sección sobr *Usando Niveles* para obtener más información sobre los diferentes niveles y lo que significan.

Una vez que se crea una configuración, no se puede cambiar. En su lugar, debes eliminar la configuración y volver a crearla utilizando *Cake\Log\Log::drop()* y *Cake\Log\Log::setConfig()*.

También es posible crear loggers proporcionando un cierre (closure). Esto es útil cuando necesitas un control completo sobre cómo se construye el objeto del logger. El cierre debe devolver la instancia del logger. Por ejemplo:

```
Log::setConfig(
↪'special',
↪ function␣
↪() {
    return␣
↪new \Cake\
↪Log\
↪Engine\
↪FileLog([
↪'path' =>␣
↪LOGS,
↪'file' =>
↪'log']);
});
```

Las opciones de configuración también se pueden proporcionar como una cadena *DSN*. Esto es útil cuando se trabaja con variables de entorno o proveedores *PaaS*:

```
Log::setConfig(
↪'error', [
    'url' =>
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪ 'file:///
↪full/path/
↪to/logs/?
↪levels[]=warning&
↪levels[]=error&
↪file=error
↪',
]);
```

Advertencia: Si no configuras motores de registro (logging), los mensajes de log no se almacenarán.

Registro de Errores y Excepciones

Los errores y excepciones también pueden registrarse configurando los valores correspondientes en tu archivo **config/app.php**. Los errores se mostrarán cuando el modo de depuración esté en `true` y se registrarán en los archivos de log cuando el modo de depuración esté en `false`. Para registrar excepciones no capturadas, configura la opción `log` como `true`. Consulta :*Configuración* para obtener más información.

Escribiendo en los archivos de Log

Escribir en los archivos de registro se puede hacer de dos maneras diferentes. La primera es utilizando el método estático :*Cake\Log\Log::write()*:

```
Log::write(
↪'debug',
↪'Something␣
↪did not␣
↪work');
```

La segunda opción es utilizar la función de acceso directo `log()` disponible en cualquier clase que utilice el `LogTrait`. Llamar a``log()`` llamará internamente a``Log::write()``:

```
//␣
↪Ejecutando␣
↪esto␣
↪dentro de␣
↪una clase␣
↪que␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪utiliza␣
↪LogTrait
$this->log(
↪'Something␣
↪did not␣
↪work!',
↪'debug');
```

Todos los `log` configurados se escriben secuencialmente cada vez que se llama a *Cake\Log\Log::write()*. Si no has configurado ningún motor de registro, `log()` devolverá «false» y no se escribirán mensajes de registro.

Usando marcadores de posición (placeholders) en mensajes

Si necesitas registrar datos definidos dinámicamente, puedes utilizar marcadores de posición en tus mensajes de registro y proporcionar un array de pares clave/valor en el parámetro `$context` como sigue:

```
// Se␣
↪registrará␣
↪`No se␣
↪pudo␣
↪procesar␣
↪para el␣
↪usuario␣
↪id = 1`
Log::write(
↪'error',
↪'No se␣
↪pudo␣
↪procesar␣
↪para el␣
↪usuario␣
↪id ={user}
↪', ['user
↪' =>
↪$user->
↪id]);
```

Los marcadores (placeholders) que no tienen claves definidas no serán reemplazados. Si necesitas utilizar una palabra entre llaves de forma literal, debes escapar el marcador:

```
// Se␣
↪registrará␣
↪`No
↪{replace}`
Log::write(
↪'error',
↪'No \\
↪{replace}
↪', [
↪'replace'␣
↪=> 'no']);
```

Si incluyes objetos en los marcadores, esos objetos deben implementar uno de los siguientes métodos:

- `__toString()`
- `toArray()`
- `__debugInfo()`

Usando Niveles

CakePHP admite el conjunto estándar de niveles de registro POSIX. Cada nivel representa un aumento en el nivel de gravedad:

- Emergency: el sistema no es utilizable
- Alert: se debe tomar una acción inmediata
- Critical: condiciones críticas
- Error: condiciones de error
- Warning: condiciones de advertencia
- Notice: condiciones normales pero significativas
- Info: mensajes informativos
- Debug: mensajes

de depuración

Puedes hacer referencia a estos niveles por nombre al configurar lo `loggers` y al escribir mensajes de registro. Alternativamente, puedes utilizar métodos de conveniencia como : *Cake\Log\Log::error()* para indicar claramente el nivel de registro. Utilizar un nivel que no esté en la lista de niveles anteriores resultará en una excepción.

Nota: Cuando `levels` se establece en un valor vacío en la configuración de un `logger`, aceptará mensajes de cualquier nivel.

Ámbitos de Registro (scope)

En muchas ocasiones, querrás configurar diferentes comportamientos de registro para diferentes subsistemas o partes de tu aplicación. Tomemos como ejemplo una tienda en línea. Probablemente, quieras manejar el registro de pedidos y pagos de manera diferente a como lo haces con otros registros menos críticos.

CakePHP expone este concepto como ámbitos de registro. Cuando se escriben mensajes de registro, puedes incluir un nombre de ámbito `scope`. Si hay un registrador configurado para ese ámbito, los mensajes de registro se dirigirán a esos `loggers`. Por ejemplo:

```
use Cake\
↪Log\
↪Engine\
↪FileLog;

//␣
↪Configura␣
↪logs/
↪shops.log␣
↪para␣
↪recibir␣
↪todos los␣
↪niveles,␣
↪pero solo␣
↪aquellos␣
↪con␣
↪``scope``
// `orders`␣
↪y␣
↪`payments`.
Log::setConfig(
↪'shops', [

↪'className
↪' =>␣
↪FileLog::class,
↪
    'path'␣
↪=> LOGS,
    'levels
↪' => [],
    'scopes
```

(continué en la próxima página)

(proviene de la página anterior)

```
    ↪' => [
    ↪'orders',
    ↪'payments
    ↪'],
        'file'␣
    ↪=> 'shops.
    ↪log',
]);

//␣
    ↪Configura␣
    ↪logs/
    ↪payments.
    ↪log para␣
    ↪recibir␣
    ↪todos los␣
    ↪niveles,␣
    ↪pero solo␣
    ↪aquellos␣
    ↪con␣
    ↪``scope``
//␣
    ↪`payments`.
Log::setConfig(
    ↪'payments
    ↪', [

    ↪'className
    ↪' =>␣
    ↪FileLog::class,
    ↪
        'path'␣
    ↪=> LOGS,
        'levels
    ↪' => [],
        'scopes
    ↪' => [
    ↪'payments
    ↪'],
        'file'␣
    ↪=>
    ↪'payments.
    ↪log',
]);

Log::warning(
    ↪'this␣
    ↪gets␣
    ↪written␣
    ↪only to␣
    ↪shops.log
    ↪', ['scope
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪' => [
↪'orders
↪']]);
Log::warning(
↪'this␣
↪gets␣
↪written␣
↪to both␣
↪shops.log␣
↪and␣
↪payments.
↪log', [
↪'scope' =>
↪ [
↪'payments
↪']]);
```

Los `scopes` también se pueden pasar como una cadena única o como una matriz indexada numéricamente. Ten en cuenta que al usar esta forma, se limitará la capacidad de pasar más datos como contexto:

```
Log::warning(
↪'This is␣
↪a warning
↪', [
↪'orders
↪']);
Log::warning(
↪'This is␣
↪a warning
↪',
↪'payments
↪');
```

Nota: Cuando `scopes` se establece como un arreglo vacío o null en la configuración de un `logger`, aceptará mensajes de cualquier `scope`. Establecerlo como false solo coincidirá con mensajes sin `scope`.

Guardando logs en Archivos

Como su nombre indica, `FileLog` escribe mensajes de registro en archivos. El nivel del mensaje de registro que se está escribiendo determina el nombre del archivo en el que se almacena el mensaje. Si no se proporciona un nivel, se utiliza `LOG_ERR`, que escribe en el registro de errores. La ubicación de registro predeterminada es **logs/$level.log**:

```
// Es␣
↪ejecutado␣
↪asi␣
↪dentro de␣
↪una clase␣
↪CakePHP
$this->log(
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪"Something␣
↪didn't␣
↪work!");

// Se␣
↪añadirá␣
↪lo␣
↪siguiente␣
↪al␣
↪archivo␣
↪logs/
↪error.log.
// 2007-11-
↪02␣
↪10:22:02␣
↪Error:␣
↪Something␣
↪didn't␣
↪work!
```

El directorio configurado debe tener permisos de escritura por el usuario del servidor web para que el registro funcione correctamente.

Puedes configurar ubicaciones adicionales o alternativas para FileLog al configurar un registrador. FileLog acepta un «path» que permite utilizar rutas personalizadas:

```
Log::setConfig(
↪'custom_
↪path', [

↪'className
↪' => 'File
↪',
    'path'␣
↪=> '/path/
↪to/custom/
↪place/'
]);
```

El motor de `FileLog` toma las siguientes opciones:

- **`size`** **Se utiliza para implementar una rotación básica de archivos de registro. Si el tamaño**

del archivo de registro alcanza el tamaño especificado, el archivo existente se renombra agregando una marca de tiempo al nombre de archivo y se crea un nuevo archivo de re-

gistro. Puede ser un valor entero en bytes o valores como "10MB", "100KB", etc. El valor predeterminado es 10MB.

- `rotate` Los archivos de registro se rotan un número especificado de veces antes de ser eliminados. Si el valor es 0, se eliminan las versiones antiguas en lugar de rotarlas. El valor

predeterminado es 10.

- **mask Establece los permisos de archivo para los archivos creados. Si se deja vacío, se utilizan** los permisos predeterminados.

Nota: Los directorios faltantes se crearán automáticamente para evitar errores innecesarios cuando se utiliza FileEngine.

Guardando logs en Syslog

En entornos de producción, se recomienda encarecidamente configurar tu sistema para utilizar el syslog en lugar del guardar los logs en archivos. Esto mejorará el rendimiento, ya que cualquier escritura se realizará de manera (casi) no bloqueante y el `logger` del sistema operativo se puede configurar de forma independiente para rotar archivos, preprocesar escrituras o utilizar un almacenamiento completamente diferente para tus registros.

Usar syslog es prácticamente como usar el motor de registro de archivos predeterminado, simplemente necesitas especificar `Syslog` como el motor a utilizar para el registro de logs. El siguiente fragmento de configuración reemplazará el `logger` predeterminado con syslog, esto se debe hacer en el archivo **config/bootstrap.php**:

```
Log::setConfig(
↪'default',
↪ [
    'engine
↪' =>
↪'Syslog'
]);
```

El arreglo de configuración aceptado para el motor de registro Syslog comprende las siguientes claves:

- `format`: Una ca-

dena de plantilla sprintf con dos marcadores de posición (placeholdes), el primero para el nivel de error y el segundo para el mensaje en sí. Esta clave es útil para agregar información adicional sobre el servidor o el proceso en el mensaje registrado. Por ejemplo: `%s - Servidor web 1 - %s` se verá como `error - Servidor web 1 - Ocurrió un error en esta solicitud` después de reemplazar los placeholders. Esta opción está obsoleta. Deberías usar

Formateadores de Logs en su lugar.

- `prefix`: Una cadena que se utilizará como prefijo para cada mensaje registrado.
- **`flag`: Una bandera tipo `int` que se usará para abrir la conexión al registro,** por defecto se usará `LOG_ODELAY`. Consulta la documentación de `openlog` para ver más opciones.

- **facility: El espacio de registro a utilizar en syslog. Por defecto se utiliza LOG_USER.** Consulta la documentación de `syslog` para ver más opciones.

Creación de Motores de Logs

Los motores de registro pueden formar parte de tu aplicación o de plugins. Por ejemplo, si tuvieras un registro en base de datos llamado `DatabaseLog`, como parte de tu aplicación se colocaría en **src/Log/Engine/DatabaseLog.php**. Como parte de un plugin se colocaría en **plugins/LoggingPack/src/Log/Engine/DatabaseLog.php**. Para configurar el motor de registro, debes usar *Cake\Log\Log::setConfig()*. Por ejemplo, la configuración de nuestro DatabaseLog se vería así:

```
// Para src/
↪Log
Log::setConfig(
↪'otherFile
↪', [

↪'className
↪' =>
↪'Database
↪',
    'model'␣
↪=>
↪'LogEntry
↪',
    // ...
]);
```

(continué en la próxima página)

(proviene de la página anterior)

```
// Para el␣
↪plugin␣
↪llamado␣
↪LoggingPack
Log::setConfig(
↪'otherFile
↪', [

↪'className
↪' =>
↪'LoggingPack.
↪Database',
    'model'␣
↪=>
↪'LogEntry
↪',
    // ...
]);
```

Al configurar un motor de registro, el parámetro `className` se utiliza para localizar y cargar el controlador de registro. Todas las demás propiedades de configuración se pasan al constructor del motor de registro como un array.:

```
namespace␣
↪App\Log\
↪Engine;
use Cake\
↪Log\
↪Engine\
↪BaseLog;

class␣
↪DatabaseLog␣
↪extends␣
↪BaseLog
{
    public␣
↪function _
↪_
↪construct(array␣
↪$config =␣
↪[])
    {
        ␣
↪parent::__
↪construct(
↪$config);
        // .
↪..
    }

    public␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪function␣
↪log(
↪$level,␣
↪string
↪$message,␣
↪array
↪$context␣
↪= [])
    {
        //␣
↪Write to␣
↪the␣
↪database.
    }
}
```

CakePHP requiere que todos los motores de registro implementen PsrLogLoggerInterface. La clase `CakeLogEngineBaseLog` es una forma sencilla de cumplir con la interfaz, ya que solo requiere que implementes el método log().

Formateadores de Logs

Los formateadores de registro te permiten controlar cómo se formatean los mensajes de registro de forma independiente al motor de almacenamiento. Cada motor de registro proporcionado por defecto viene con un formateador configurado para mantener una salida compatible con versiones anteriores. Sin embargo, puedes ajustar los formateadores para satisfacer tus requisitos. Los formateadores se configuran junto al motor de registro:

```
use Cake\
↪Log\
↪Engine\
↪SyslogLog;
use App\Log\
↪Formatter\
↪CustomFormatter;
↪

//␣
↪Configuración␣
↪de␣
↪formato␣
↪simple␣
↪sin␣
↪opciones.
Log::setConfig(
↪'error', [

↪'className
↪' =>␣
↪SyslogLog::class,
↪
```

(continué en la próxima página)

(proviene de la página anterior)

```
    ↪'formatter
    ↪' =>␣
    ↪CustomFormatter::class,
    ↪
]);

//␣
↪Configurar␣
↪un␣
↪formateador␣
↪con␣
↪algunas␣
↪opciones.
Log::setConfig(
↪'error', [

    ↪'className
    ↪' =>␣
    ↪SyslogLog::class,
    ↪

    ↪'formatter
    ↪' => [

        ↪'className
        ↪' =>␣
        ↪CustomFormatter::class,
        ↪
            'key
        ↪' =>
        ↪'value',
    ],
]);
```

Para implementar tu propio formateador de registro, necesitas extender `Cake\Log\Format\AbstractFormatter` o una de sus subclases. El método principal que debes implementar es `format($level, $message, $context)` que es responsable de formatear los mensajes de log.

Log API

class
Cake\Log**Log**

Una clase sencilla para escribir logs.

static Cake\Log\Log::**setCon**

Parámetros

- **$name** (string) – Nombre para el registro al que se está conectando, utilizado para eliminar un registro más adelante.
- **$config** (array) – Arreglo de configuración y argu-

mentos del constructor para el `logger`.

Devuelve o establece la configuración de un `logger`. Para mas información ver *Logging Configuration*.

static Cake\Log\Log::**configured**

Devuelve Arreglo de los `loggers` configurados

Devuelve los nombres de los `loggers` configurados.

static Cake\Log\Log::**drop**(

Parámetros

- **$name** (`string`) –

Nombre del `logger` del que ya no deseas recibir mensajes.

static Cake\Log\Log::**write**(

Escribe un mensaje en todos los `loggers` configurados `$level` indica el nivel del mensaje de registro que se está creando. `$message` es el mensaje de la entrada del registro que se está escribiendo. `$scope` es el(los) ámbito(s) en el que se está creando

un mensaje de registro.

static Cake\Log\Log::levels

Llama a este método sin argumentos, por ejemplo: *Log::levels()* para obtener la configuración actual del nivel.

Métodos de conveniencia

Se agregaron los siguientes métodos útiles para registrar *$message* con el nivel de registro apropiado.

static Cake\Log\Log::emerge

static Cake\Log\Log::alert

static Cake\Log\Log::critic

static Cake\Log\Log::error

static Cake\Log\Log::warnin

static Cake\Log\Log::notice

static Cake\Log\Log::info($

static Cake\Log\Log::debug(

Logging Trait

trait Cake\Log\LogTrait

Un trait que proporciona métodos abreviados para el registro de mensajes.

Cake\Log\LogTrait::log($msg, $level = LOG

Agregar un mensaje al log. De forma predeterminada, los mensajes se registran como mensajes de ERROR.

Usando Monolog

Monolog es una librería de logging popular en PHP. Dado que implementa las mismas interfaces que los `loggers` de CakePHP, puedes usarlos en tu aplicación como el `logger` predeterminado.

Una vez instalado Monolog utilizando composer, configura el `logger` usando el método `Log::setConfig()`:

```
// config/
↪bootstrap.
↪php
```

(continué en la próxima página)

(proviene de la página anterior)

```
use Monolog\
↪Logger;
use Monolog\
↪Handler\
↪StreamHandler;
↪

Log::setConfig(
↪'default',
↪ function␣
↪() {
    $log =␣
↪new␣
↪Logger(
↪'app');
    $log->
↪pushHandler(new␣
↪StreamHandler(
↪'ruta/a/
↪tu/
↪combined.
↪log'));

    return
↪$log;
});

//␣
↪Opcionalmente␣
↪deja de␣
↪usar los␣
↪``loggers``␣
↪predeterminados␣
↪que ahora␣
↪son␣
↪redundantes.
↪
Log::drop(
↪'debug');
Log::drop(
↪'error');
```

Utiliza métodos similares si deseas configurar un `logger` diferente para tu consola:

```
// config/
↪bootstrap_
↪cli.php

use Monolog\
↪Logger;
use Monolog\
↪Handler\
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪StreamHandler;
↪

Log::setConfig(
↪'default',
↪ function␣
↪() {
    $log =␣
↪new␣
↪Logger(
↪'cli');
    $log->
↪pushHandler(new␣
↪StreamHandler(
↪'ruta/a/
↪tu/
↪combined-
↪cli.log
↪'));

    return
↪$log;
});

//␣
↪Opcionalmente␣
↪deja de␣
↪usar los␣
↪``logger``␣
↪predeterminados␣
↪redundantes␣
↪para la␣
↪línea de␣
↪comando.
Configure::delete(
↪'Log.debug
↪');
Configure::delete(
↪'Log.error
↪');
```

Nota: Cuando uses un `logger` específico para la consola, asegúrate de configurar condicionalmente tu `logger` de aplicación. Esto evitará entradas de registro duplicadas.

CAPÍTULO 21

Modelless Forms

class Cake\
Form**Form**

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[135] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[135] https://github.com/cakephp/docs

CAPÍTULO 22

Plugins

CakePHP te permite configurar una combinación de controladores, modelos y vistas y liberarlos como un plugin de aplicación pre-empaquetado que otros pueden usar en sus aplicaciones CakePHP. Si has creado una gran gestión de usuarios, un blog sencillo, o adaptadores de servicios web en una de tus aplicaciones ¿por qué no empaquetarlo como un plugin CakePHP? De esta manera puedes utilizarlo en tus otras aplicaciones y compartirlo con la comunidad.

Un plugin de CakePHP es independiente de la aplicación principal y generalmente proporciona alguna funcionalidad bien definida que se puede empaquetar de forma ordenada y reutilizar con poco esfuerzo en otras aplicaciones. La aplicación y el plugin operan en sus respectivos espacios, pero comparten los datos de configuración de la aplicación (por ejemplo, conexiones de base de datos, transportes de correo electrónico).

Los plugins deben definir su propio espacio de nombres (namespace) de nivel superior. Por ejemplo: `DebugKit`. Por convención, los plugins utilizan el nombre de su paquete como su espacio de nombres. Si deseas utilizar un espacio de nombres diferente, puedes configurarlo cuando se cargan los plugins.

Instalación de un plugin con Composer

Muchos plugins están disponibles en Packagist[136] y se pueden instalar con `Composer`. Para instalar DebugKit debes hacer lo siguiente:

```
php␣
↪composer.
↪phar␣
↪require␣
↪cakephp/
↪debug_kit
```

Esto instalará la última versión de DebugKit y actualizará tus archivos **composer.json** y **composer.lock**, también actualizará el archivo **vendor/cakephp-plugins.php** y tu cargador automático (autoloader).

[136] https://packagist.org

Instalación manual de un Plugin

Si el complemento que deseas instalar no está disponible en packagist.org, puedes clonar o copiar el código del complemento en tu directorio de **plugins**. Supongamos que deseas instalar un complemento llamado "ContactManager", debes tener una carpeta en **plugins** llamada "ContactManager". En este directorio se encuentran las carpetas src, tests y cualquier otra carpeta del complemento.

Carga automática manual de clases de Plugins

Si instalas tus complementos mediante `composer` o `bake`, no deberías necesitar configurar la carga automática de clases para tus plugins.

Si creas un plugin manualmente en el directorio `plugins`, entonces necesitarás indicarle a `composer` que actualice su caché de carga automática:

```
php␣
↪composer.
↪phar␣
↪dumpautoload
```

Si estás utilizando espacios de nombres de proveedores para tus plugins, deberás agregar el mapeo de espacio de nombres a la ruta en el archivo `composer.json` de la siguiente manera, antes de ejecutar el comando de composer mencionado anteriormente:

```
{

↪"autoload
↪": {

↪"psr-4": {

↪"AcmeCorp\
↪\Users\\
↪":
↪"plugins/
↪AcmeCorp/
↪Users/src/
↪",
        }
    },

↪"autoload-
↪dev": {

↪"psr-4": {

↪"AcmeCorp\
↪\Users\\
↪Test\\":
↪"plugins/
↪AcmeCorp/
↪Users/
↪tests/"
```

(continué en la próxima página)

(proviene de la página anterior)

```
        }
    }
}
```

Cargar un Plugin

Si deseas utilizar las rutas, comandos de consola, middlewares, listeners de eventos, plantillas o assets de la carpeta web (webroot) de un plugin, deberás cargar el plugin.

Si solo deseas utilizar helpers, comportamientos o componentes de un plugin, no es necesario cargar explícitamente el plugin, aunque siempre es recomendable hacerlo.

También hay un práctico comando de consola para cargar el plugin. Ejecuta la siguiente línea:

```
bin/cake␣
↪plugin␣
↪load␣
↪ContactManager
```

Esto actualizaría el array en el archivo `config/plugins.php` de tu aplicación con una entrada similar a `'ContactManager' => []`.

Configuración de Hooks del Plugin

Los plugins ofrecen varios hooks (ganchos) que permiten que un plugin se inyecte en las partes apropiadas de tu aplicación. Los hooks son:

- `bootstrap` Se utiliza para cargar archivos de configuración predeterminados del plugin, definir constantes y otras funciones globales.
- `routes` Se utiliza para cargar las rutas de un plugin.

Se activa después de que se cargan las rutas de la aplicación.

- `middleware` middleware Se utiliza para agregar el middleware del plugin a la cola de middlewares de una aplicación.
- `console` Se utiliza para agregar comandos de consola a la colección de comandos de una aplicación.
- `services` Se utiliza para registrar servicios del contenedor de aplicaciones.

Por defecto, todos los hooks de los plugins están habilitados. Puedes deshabilitar los hooks utilizando las opciones relacionadas del comando plugin load:

```
bin/cake␣
↪plugin␣
↪load␣
↪ContactManager␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪--no-
↪routes
```

Esto actualizaría el array en el archivo `config/plugins.php` de tu aplicación con una entrada similar a `'ContactManager' => ['routes' => false]`.

Opciones de Carga de Plugins

Además de las opciones para los hooks de complementos, el comando `plugin load` tiene las siguientes opciones para controlar la carga del plugin:

- `--only-debug` Carga el plugin solo cuando el modo de depuración (debug) está habilitado.
- `--only-cli` Carga el plugin solo para CLI.
- `--optional` No arroja un error si el plugin no está disponible.

Cargar Plugins a través de `Application::bootstrap()`

Además del array de configuración en `config/plugins.php`, los plugins también se pueden cargar en el método `bootstrap()` de tu aplicación:

```
// In src/
↪Application.
↪php
use Cake\
↪Http\
↪BaseApplication;
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪
use␣
↪ContactManager\
↪ContactManagerPlugin;
↪

class␣
↪Application␣
↪extends␣
↪BaseApplication
{
    public␣
↪function␣
↪bootstrap()
    {
        ␣
↪parent::bootstrap();
↪

        //␣
↪Load the␣
↪contact␣
↪manager␣
↪plugin by␣
↪class name

↪$this->
↪addPlugin(ContactManagerP
↪

        //␣
↪Load a␣
↪plugin␣
↪with a␣
↪vendor␣
↪namespace␣
↪by 'short␣
↪name' with␣
↪options

↪$this->
↪addPlugin(
↪'AcmeCorp/
↪ContactManager
↪', [
↪'console'␣
↪=>␣
↪false]);

        //␣
↪Load a␣
↪dev␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪dependency␣
↪that will␣
↪not exist␣
↪in␣
↪production␣
↪builds.

↪$this->
↪addOptionalPlugin(
↪'AcmeCorp/
↪ContactManager
↪');
    }
}
```

Puedes configurar hooks con opciones de array o utilizando los métodos proporcionados por las clases del plugin:

```
// In␣
↪Application::bootstrap()
use␣
↪ContactManager\
↪ContactManagerPlugin;
↪

// Use the␣
↪disable/
↪enable to␣
↪configure␣
↪hooks.
$plugin =␣
↪new␣
↪ContactManagerPlugin();
↪

$plugin->
↪disable(
↪'bootstrap
↪');
$plugin->
↪enable(
↪'routes');
$this->
↪addPlugin(
↪$plugin);
```

Los objetos del plugin también conocen sus nombres e información de la ruta:

```
$plugin =␣
↪new␣
↪ContactManagerPlugin();
↪
```

(continué en la próxima página)

(proviene de la página anterior)

```
// Get the
↪plugin
↪name.
$name =
↪$plugin->
↪getName();

// Path to
↪the
↪plugin
↪root, and
↪other
↪paths.
$path =
↪$plugin->
↪getPath();
$path =
↪$plugin->
↪getConfigPath();
↪
$path =
↪$plugin->
↪getClassPath();
↪
```

Uso de las Clases del Plugins

Puedes hacer referencia a los controladores, modelos, componentes, comportamientos y helpers de un plugin prefijando el nombre del plugin.

Por ejemplo, supongamos que quieres usar el helper `ContactInfoHelper` del plugin ContactManager para mostrar información de contacto formateada en una de tus vistas. En tu controlador, usar `addHelper()` podría verse así:

```
$this->
↪viewBuilder()-
↪>
↪addHelper(
↪'ContactManager.
↪ContactInfo
↪');
```

Nota: Este nombre de clase separado por puntos se denomina *Sintaxis de plugin*.

Luego podrías acceder al `ContactInfoHelper` de la misma manera que cualquier otro helper en tu vista, como por ejemplo:

```
echo $this->
↪ContactInfo-
↪>address(
↪$contact);
```

Los plugins pueden utilizar los modelos, componentes, comportamientos y helpers proporcionados por la aplicación, o por otros plugins si es necesario:

```
// Use an␣
↪application␣
↪component
$this->
↪loadComponent(
↪'AppFlash
↪');

// Use␣
↪another␣
↪plugin's␣
↪behavior
$this->
↪addBehavior(
↪'OtherPlugin.
↪AuditLog
↪');
```

Creación de tus propios Plugins

Como ejemplo práctico, comencemos a crear el plugin ContactManager mencionado anteriormente. Para empezar, configuraremos la estructura básica del directorio de nuestro complemento. Debería verse así:

```
/src
/plugins
    /
↪ContactManager
        /
↪config
        /src
            ␣
↪/
↪ContactManagerPlugin.
↪php
            ␣
↪/
↪Controller
            ␣
↪    /
↪Component
            ␣
↪/Model
            ␣
↪    /Table
            ␣
↪    /
↪Entity
            ␣
↪    /
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪Behavior
                ␣
↪/View
                ␣
↪    /
↪Helper
          /
↪templates
                ␣
↪/layout
          /
↪tests
                ␣
↪/TestCase
                ␣
↪/Fixture
          /
↪webroot
```

Ten en cuenta el nombre de la carpeta del plugin, "**ContactManager**". Es importante que esta carpeta tenga el mismo nombre que el plugin.

Dentro de la carpeta del plugin, notarás que se parece mucho a una aplicación de CakePHP, y básicamente eso es lo que es. En lugar de un archivo `Application.php`, tienes un archivo `ContactManagerPlugin.php`. No es necesario incluir ninguna de las carpetas que no estés usando. Algunos plugins solo pueden definir un Component y un Behavior, y en ese caso pueden omitir completamente el directorio "templates".

Un plugin también puede tener prácticamente cualquiera de los otros directorios que puede tener tu aplicación, como Config, Console, webroot, etc.

Crear un Plugin Utilizando Bake

El proceso de creación de plugins puede simplificarse enormemente utilizando Bake.

Para hornear (bakear) un plugin, utiliza el siguiente comando:

```
bin/cake␣
↪bake␣
↪plugin␣
↪ContactManager
```

Bake puede utilizarse para crear clases en tu plugin. Por ejemplo, para generar un controlador de plugin podrías ejecutar:

```
bin/cake␣
↪bake␣
↪controller␣
↪--plugin␣
↪ContactManager␣
↪Contacts
```

Por favor, consulta el capítulo /bake/usage si tienes problemas al utilizar la línea de comandos. Asegúrate de regenerar tu cargador automático (autoloader) una vez que hayas creado tu plugin:

```
php␣
↪composer.
↪phar␣
↪dumpautoload
```

Objetos del Plugin

Los Objetos de Plugin permiten a un autor de plugin definir lógica de configuración, ganchos (hooks) predeterminados, cargar rutas, middleware y comandos de consola. Los objetos de plugin se encuentran en **src/<PluginName>Plugin.php**. Para nuestro plugin ContactManager, nuestra clase de plugin podría verse así:

```
namespace␣
↪ContactManager;
↪

use Cake\
↪Core\
↪BasePlugin;
↪
use Cake\
↪Core\
↪ContainerInterface;
↪
use Cake\
↪Core\
↪PluginApplicationInterfac
↪
use Cake\
↪Console\
↪CommandCollection;
↪
use Cake\
↪Http\
↪MiddlewareQueue;
↪
use Cake\
↪Routing\
↪RouteBuilder;
↪

class␣
↪ContactManagerPlugin␣
↪extends␣
↪BasePlugin
{

    /**
     *␣
↪@inheritDoc
     */
    public␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪function␣
↪middleware(MiddlewareQueu
↪
↪$middleware):␣
↪MiddlewareQueue
    {
        //␣
↪Add␣
↪middleware␣
↪here.

↪$middleware␣
↪=␣
↪parent::middleware(
↪$middleware);
↪


        ␣
↪return
↪$middleware;
↪
    }

    /**
     *␣
↪@inheritDoc
     */
    public␣
↪function␣
↪console(CommandCollectio
↪
↪$commands):␣
↪CommandCollection
    {
        //␣
↪Add␣
↪console␣
↪commands␣
↪here.

↪$commands␣
↪=␣
↪parent::console(
↪$commands);
↪


        ␣
↪return
↪$commands;
    }

    /**
```

(continué en la próxima página)

(proviene de la página anterior)

```
     *␣
↪@inheritDoc
     */
    public␣
↪function␣
↪bootstrap(PluginApplicati
↪$app):␣
↪void
    {
        //␣
↪Add␣
↪constants,
↪ load␣
↪configuration␣
↪defaults.
        //␣
↪By␣
↪default␣
↪will load␣
↪`config/
↪bootstrap.
↪php` in␣
↪the␣
↪plugin.
        ␣
↪parent::bootstrap(
↪$app);
    }

    /**
     *␣
↪@inheritDoc
     */
    public␣
↪function␣
↪routes(RouteBuilder␣
↪$routes):␣
↪void
    {
        //␣
↪Add␣
↪routes.
        //␣
↪By␣
↪default␣
↪will load␣
↪`config/
↪routes.
↪php` in␣
↪the␣
↪plugin.
        ␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪parent::routes(
↪$routes);
    }

    /**
     *␣
↪Register␣
↪application␣
↪container␣
↪services.
     *
     *␣
↪@param \
↪Cake\Core\
↪ContainerInterface␣
↪
↪$container␣
↪The␣
↪Container␣
↪to update.
     *␣
↪@return␣
↪void
     *␣
↪@link␣
↪https://
↪book.
↪cakephp.
↪org/5/en/
↪development/
↪dependency-
↪injection.
↪html
↪#dependency-
↪injection
     */
    public␣
↪function␣
↪services(ContainerInterfa
↪
↪$container):␣
↪void
    {
        //␣
↪Add your␣
↪services␣
↪here
    }
}
```

Rutas del Plugin

Los plugins pueden proporcionar archivos de rutas que contienen sus propias rutas. Cada plugin puede contener un archivo **config/routes.php**. Este archivo de rutas se puede cargar cuando se agrega el plugin o en el archivo de rutas de la aplicación. Para crear las rutas del plugin ContactManager, coloca lo siguiente en **plugins/ContactManager/config/routes.php**:

```
<?php
use Cake\
↪Routing\
↪Route\
↪DashedRoute;
↪

$routes->
↪plugin(

↪'ContactManager
↪',
    ['path'␣
↪=> '/
↪contact-
↪manager'],
  ␣
↪function (
↪$routes) {

↪$routes->
↪setRouteClass(DashedRoute
↪


↪$routes->
↪get('/
↪contacts',
↪ [
↪'controller
↪' =>
↪'Contacts
↪']);

↪$routes->
↪get('/
↪contacts/
↪{id}', [
↪'controller
↪' =>
↪'Contacts
↪', 'action
↪' => 'view
↪']);

↪$routes->
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪put('/
↪contacts/
↪{id}', [
↪'controller
↪' =>
↪'Contacts
↪', 'action
↪' =>
↪'update
↪']);
    }
);
```

Lo anterior conectará las rutas predeterminadas para tu plugin. Más adelante, puedes personalizar este archivo con rutas más específicas.

También puedes cargar las rutas del plugin en la lista de rutas de tu aplicación. Hacer esto te proporciona un mayor control sobre cómo se cargan las rutas del plugin y te permite envolver las rutas del plugin en ámbitos o prefijos adicionales:

```
$routes->
↪scope('/',
↪ function␣
↪($routes)
↪{
    //␣
↪Connect␣
↪other␣
↪routes.
    $routes-
↪>scope('/
↪backend',␣
↪function (
↪$routes) {

↪$routes->
↪loadPlugin(
↪'ContactManager
↪');
    });
});
```

Lo anterior resultaría en URLs como `/backend/contact-manager/contacts`.

Controladores del Plugin

Los controladores para nuestro plugin ContactManager se almacenarán en **plugins/ContactManager/src/Controller/**. Dado que la principal tarea que realizaremos es gestionar contactos, necesitaremos un ContactsController para este plugin.

Entonces, colocamos nuestro nuevo ContactsController en **plugins/ContactManager/src/Controller** y se verá así:

```
// plugins/
↪ContactManager/
↪src/
↪Controller/
↪ContactsController.
↪php
namespace␣
↪ContactManager\
↪Controller;
↪

use␣
↪ContactManager\
↪Controller\
↪AppController;
↪

class␣
↪ContactsController␣
↪extends␣
↪AppController
{
    public␣
↪function␣
↪index()
    {
        //..
↪.
    }
}
```

También crea el controlador `AppController` si aún no lo tienes:

```
// plugins/
↪ContactManager/
↪src/
↪Controller/
↪AppController.
↪php
namespace␣
↪ContactManager\
↪Controller;
↪

use App\
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪Controller\
↪AppController␣
↪as␣
↪BaseController;
↪

class␣
↪AppController␣
↪extends␣
↪BaseController
{
}
```

El `AppController` de un plugin puede contener lógica de controlador común a todos los controladores en un plugin, pero no es obligatorio usarlo.

Si deseas acceder a lo que hemos hecho hasta ahora, visita `/contact-manager/contacts`. Deberías obtener un error de «Modelo faltante» porque aún no hemos definido un modelo Contact.

Si tu aplicación incluye el enrutamiento predeterminado que proporciona CakePHP, podrás acceder a los controladores de tu plugin utilizando URLs como:

```
// Access␣
↪the index␣
↪route of␣
↪a plugin␣
↪controller.
↪
/contact-
↪manager/
↪contacts

// Any␣
↪action on␣
↪a plugin␣
↪controller.
↪
/contact-
↪manager/
↪contacts/
↪view/1
```

Si tu aplicación define prefijos de enrutamiento, el enrutamiento predeterminado de CakePHP también conectará rutas que utilizan el siguiente patrón:

```
/{prefix}/
↪{plugin}/
↪{controller}
↪
/{prefix}/
↪{plugin}/
↪{controller}
↪/{action}
```

Consulta la sección sobre *Configuración de Hooks del Plugin* para obtener información sobre cómo cargar archivos de rutas específicos del plugin.

Modelos del Plugin

Los modelos para el plugin se almacenan en **plugins/ContactManager/src/Model**. Ya hemos definido un Contacts-Controller para este plugin, así que creemos la tabla y entidad para ese controlador:

```
// plugins/
↪ContactManager/
↪src/Model/
↪Entity/
↪Contact.
↪php:
namespace␣
↪ContactManager\
↪Model\
↪Entity;

use Cake\
↪ORM\
↪Entity;

class␣
↪Contact␣
↪extends␣
↪Entity
{
}

// plugins/
↪ContactManager/
↪src/Model/
↪Table/
↪ContactsTable.
↪php:
namespace␣
↪ContactManager\
↪Model\
↪Table;

use Cake\
↪ORM\Table;

class␣
↪ContactsTable␣
↪extends␣
↪Table
{
}
```

Si necesitas hacer referencia a un modelo dentro de tu plugin al establecer asociaciones o definir clases de entidad, debes incluir el nombre del plugin con el nombre de la clase, separados por un punto. Por ejemplo:

```
// plugins/
↪ContactManager/
↪src/Model/
↪Table/
↪ContactsTable.
↪php:
namespace␣
↪ContactManager\
↪Model\
↪Table;

use Cake\
↪ORM\Table;

class␣
↪ContactsTable␣
↪extends␣
↪Table
{
    public␣
↪function␣
↪initialize(array␣
↪$config):␣
↪void
    {

↪$this->
↪hasMany(
↪'ContactManager.
↪AltName');
    }
}
```

Si prefieres que las claves del array para la asociación no tengan el prefijo del plugin, puedes utilizar la sintaxis alternativa:

```
// plugins/
↪ContactManager/
↪src/Model/
↪Table/
↪ContactsTable.
↪php:
namespace␣
↪ContactManager\
↪Model\
↪Table;

use Cake\
↪ORM\Table;

class␣
↪ContactsTable␣
↪extends␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪Table
{
    public␣
↪function␣
↪initialize(array␣
↪$config):␣
↪void
    {

↪$this->
↪hasMany(
↪'AltName',
↪ [

↪'className
↪' =>
↪'ContactManager.
↪AltName',
        ]);
    }
}
```

Puedes utilizar `Cake\ORM\Locator\LocatorAwareTrait` para cargar las tablas de tu plugin utilizando la familiar *Sintaxis de plugin*:

```
//␣
↪Controllers␣
↪already␣
↪use␣
↪LocatorAwareTrait,
↪ so you␣
↪don't need␣
↪this.
use Cake\
↪ORM\
↪Locator\
↪LocatorAwareTrait;
↪

$contacts =
↪$this->
↪fetchTable(
↪'ContactManager.
↪Contacts
↪');
```

Plantillas de Plugin

Las vistas se comportan exactamente como lo hacen en las aplicaciones normales. Solo colócalas en la carpeta correcta dentro de la carpeta `plugins/[NombreDelPlugin]/templates/`. Para nuestro plugin ContactManager, necesitaremos una vista para nuestra acción `ContactsController::index()`, así que incluyámosla también:

```
// plugins/
↪ContactManager/
↪templates/
↪Contacts/
↪index.php:
<h1>Contacts
↪</h1>
<p>
↪Following␣
↪is a␣
↪sortable␣
↪list of␣
↪your␣
↪contacts</
↪p>
<!-- A␣
↪sortable␣
↪list of␣
↪contacts␣
↪would go␣
↪here....--
↪>
```

Los plugins pueden proporcionar sus propios diseños. Para añadir diseños de plugin, coloca tus archivos de plantilla dentro de `plugins/[NombreDelPlugin]/templates/layout`. Para usar un diseño de plugin en tu controlador, puedes hacer lo siguiente:

```
$this->
↪viewBuilder()-
↪>
↪setLayout(
↪'ContactManager.
↪admin');
```

Si se omite el prefijo del plugin, el archivo de diseño/vista se ubicará de forma normal.

Nota: Para obtener información sobre cómo usar elementos de un plugin, consulta *Elementos*

Sobrescribir Plantillas de Plugin desde dentro de tu Aplicación

Puedes sobrescribir cualquier vista de un plugin desde dentro de tu aplicación usando rutas especiales. Si tienes un plugin llamado "ContactManager", puedes sobrescribir los archivos de plantilla del plugin con lógica de vista específica de la aplicación creando archivos utilizando la siguiente plantilla **templates/plugin/[Plugin]/[Controlador]/[vista].php**. Para el controlador Contacts podrías crear el siguiente archivo:

```
templates/
↪plugin/
↪ContactManager/
↪Contacts/
↪index.php
```

Crear este archivo te permitiría sobrescribir **plugins/ContactManager/templates/Contacts/index.php**.

Si tu plugin está en una dependencia de Composer (por ejemplo, "Company/ContactManager"), la ruta a la vista "index" del controlador Contacts será:

```
templates/
↪plugin/
↪TheVendor/
↪ThePlugin/
↪Custom/
↪index.php
```

Crear este archivo te permitiría sobrescribir **vendor/elproveedor/elplugin/templates/Custom/index.php**.

Si el plugin implementa un prefijo de enrutamiento, debes incluir el prefijo de enrutamiento en las sobrescrituras de plantillas de tu aplicación. Por ejemplo, si el plugin "ContactManager" implementara un prefijo "Admin", la ruta de sobrescritura sería:

```
templates/
↪plugin/
↪ContactManager/
↪Admin/
↪ContactManager/
↪index.php
```

Recursos de Plugin

Los recursos web de un plugin (pero no los archivos PHP) se pueden servir a través del directorio `webroot` del plugin, al igual que los recursos de la aplicación principal:

```
/plugins/
↪ContactManager/
↪webroot/
                ␣
↪           ␣ ␣
↪           ␣
↪css/
                ␣
↪           ␣ ␣
↪           ␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪js/
                ␣
↪               ␣ ␣
↪               ␣
↪img/
                ␣
↪               ␣ ␣
↪               ␣
↪flash/
                ␣ ␣
↪               ␣ ␣
↪               ␣
↪pdf/
```

Puedes colocar cualquier tipo de archivo en cualquier directorio, al igual que en un webroot regular.

Advertencia: Manejar recursos estáticos (como imágenes, archivos JavaScript y CSS) a través del Dispatcher es muy ineficiente. Consulta *Mejora el Rendimiento de tu Aplicación* para obtener más información.

Enlazar a Recursos en Plugins

Puedes utilizar la *Sintaxis de plugin* al enlazar a recursos de plugins utilizando los métodos script, image o css del `HtmlHelper`:

```
//␣
↪Generates␣
↪a URL of /
↪contact_
↪manager/
↪css/
↪styles.css
echo $this->
↪Html->css(
```

(continué en la próxima página)

(proviene de la página anterior)

```
    ↪'ContactManager.
    ↪styles');

//␣
    ↪Generates␣
    ↪a URL of /
    ↪contact_
    ↪manager/
    ↪js/widget.
    ↪js
echo $this->
    ↪Html->
    ↪script(
    ↪'ContactManager.
    ↪widget');

//␣
    ↪Generates␣
    ↪a URL of /
    ↪contact_
    ↪manager/
    ↪img/logo.
    ↪jpg
echo $this->
    ↪Html->
    ↪image(
    ↪'ContactManager.
    ↪logo');
```

Los recursos de los plugins se sirven por defecto utilizando el middleware `AssetMiddleware`. Esto solo se recomienda para desarrollo. En producción, debes *crear enlaces simbólicos para los recursos del plugin* para mejorar el rendimiento.

Si no estás usando los ayudantes (helpers), puedes agregar /nombre-del-plugin/ al principio de la URL para un recurso dentro del plugin y servirlo de esa manera. Enlazar a "/contact_manager/js/some_file.js" serviría el recurso **plugins/ContactManager/webroot/js/some_file.js**.

Componentes, Helpers y Behaviours

Un plugin puede tener Componentes, Helpers y Behaviours, al igual que una aplicación de CakePHP. Incluso puedes crear plugins que consistan solo en Componentes, Helpers y Behaviours, lo que puede ser una excelente manera de construir componentes reutilizables que se pueden integrar en cualquier proyecto.

La construcción de estos componentes es exactamente igual a construirlos dentro de una aplicación regular, sin ninguna convención de nomenclatura especial.

Hacer referencia a tu componente desde dentro o fuera de tu plugin solo requiere que agregues el nombre del plugin antes del nombre del componente. Por ejemplo:

```
//␣
    ↪Component␣
    ↪defined␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪in
↪'ContactManager
↪' plugin
namespace␣
↪ContactManager\
↪Controller\
↪Component;

use Cake\
↪Controller\
↪Component;

class␣
↪ExampleComponent␣
↪extends␣
↪Component
{
}

// Within␣
↪your␣
↪controllers
public␣
↪function␣
↪initialize():␣
↪void
{
   ␣
↪parent::initialize();
↪
    $this->
↪loadComponent(
↪'ContactManager.
↪Example');
}
```

La misma técnica se aplica a Ayudantes y Comportamientos.

Comandos

Los plugins pueden registrar sus comandos dentro del gancho `console()`. Por defecto, todos los comandos de consola en el plugin se descubren automáticamente y se añaden a la lista de comandos de la aplicación. Los comandos de los plugins llevan el prefijo del nombre del plugin. Por ejemplo, el `UserCommand` proporcionado por el plugin `ContactManager` se registraría tanto como `contact_manager.user` como `user`. El nombre sin prefijo solo será tomado por un plugin si no es utilizado por la aplicación o por otro plugin.

Puedes personalizar los nombres de los comandos definiendo cada comando en tu plugin:

```
public␣
↪function␣
↪console(
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪$commands)
{
    //␣
↪Create␣
↪nested␣
↪commands

↪$commands-
↪>add(
↪'bake␣
↪model',␣
↪ModelCommand::class);
↪

↪$commands-
↪>add(
↪'bake␣
↪controller
↪',␣
↪ControllerCommand::class
↪

    return
↪$commands;
}
```

Probar tu Plugin

Si estás probando controladores o generando URL, asegúrate de que tu plugin conecte las rutas en `tests/bootstrap.php`.

Para obtener más información, consulta la página de *pruebas de plugins*.

Publicar tu Plugin

Los plugins de CakePHP deben publicarse en Packagist[137]. De esta manera, otras personas pueden usarlo como dependencia de Composer. También puedes proponer tu plugin a la lista de awesome-cakephp[138].

Elige un nombre semánticamente significativo para el nombre del paquete. Idealmente, este debería llevar el prefijo del framework, en este caso «cakephp» como el framework. El nombre del proveedor generalmente será tu nombre de usuario de GitHub. **No** uses el espacio de nombres de CakePHP (cakephp), ya que está reservado para los plugins propiedad de CakePHP. La convención es usar letras minúsculas y guiones como separadores.

Entonces, si creaste un plugin «Logging» con tu cuenta de GitHub «FooBar», un buen nombre sería *foo-bar/cakephp-logging*. Y el plugin propiedad de CakePHP llamado «Localized» se puede encontrar bajo *cakephp/localized*, respectivamente.

[137] https://packagist.org
[138] https://github.com/FriendsOfCake/awesome-cakephp

Archivo de Mapeo del Plugin

Cuando instalas plugins a través de Composer, es posible que notes que se crea **vendor/cakephp-plugins.php**. Este archivo de configuración contiene un mapa de nombres de plugins y sus rutas en el sistema de archivos. Permite que los plugins se instalen en el directorio estándar del proveedor que está fuera de las rutas de búsqueda normales. La clase `Plugin` utilizará este archivo para localizar plugins cuando se carguen con `addPlugin()`. En general, no necesitarás editar este archivo manualmente, ya que Composer y el paquete `plugin-installer` se encargarán de gestionarlo por ti.

Gestiona tus Plugins usando Mixer

Otra forma de descubrir y gestionar plugins en tu aplicación de CakePHP es a través de Mixer[139]. Es un plugin de CakePHP que te ayuda a instalar plugins desde Packagist. También te ayuda a gestionar tus plugins existentes.

Nota: IMPORTANTE: No uses esto en un entorno de producción.

[139] https://github.com/CakeDC/mixer

CAPÍTULO 23

REST

Muchos de los nuevos programadores de aplicaciones se están dando cuenta de la necesidad de abrir el núcleo de la funcionalidad a un mayor público. Proporcionando acceso fácil y sin restricciones al núcleo de su API puede ayudar a que su plataforma sea aceptada, y permite realizar mashups y fácil integración con otros sistemas.

Si bien existen otras soluciones, REST es una excelente manera de proporcionar un fácil acceso a la lógica que ha creado para su aplicación. Es simple, generalmente basado en XML (estamos hablando de simple XML, nada como un envoltorio de SOAP), y depende de los encabezados HTTP por dirección. Exponer una API utilizando REST en CakePHP es simple.

La Configuración Simple

La forma más rapida para empezar a utilizar REST es agregar unas líneas para configurar la *resource routes* <*resource-routes*> en su archivo **config/routes.php**.

Una vez que la ruta se ha configurado para mapear las solicitudes REST a cierto controlador de acciones, se puede proceder a crear la lógica de nuestro controlador de acciones. Un controlador básico podría visualizarse de la siguiente forma:

```
// src/
↪Controller/
↪RecipesController.
↪php
class␣
↪RecipesController␣
↪extends␣
↪AppController
{
    public␣
↪function␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪initialize():␣
↪void
    {
        ␣
↪parent::initialize();
↪

↪$this->
↪loadComponent(
↪'RequestHandler
↪');
    }

    public␣
↪function␣
↪index()
    {

↪$recipes␣
↪= $this->
↪Recipes->
↪find('all
↪');

↪$this->
↪set(
↪'recipes',
↪
↪$recipes);

↪$this->
↪viewBuilder()-
↪>
↪setOption(
↪'serialize
↪', [
↪'recipes
↪']);
    }

    public␣
↪function␣
↪view($id)
    {

↪$recipe =
↪$this->
↪Recipes->
↪get($id);

↪$this->
↪set(
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪'recipe',
↪$recipe);

↪$this->
↪viewBuilder()-
↪>
↪setOption(
↪'serialize
↪', [
↪'recipe
↪']);
    }

    public␣
↪function␣
↪add()
    {

↪$this->
↪request->
↪allowMethod([
↪'post',
↪'put']);

↪$recipe =
↪$this->
↪Recipes->
↪newEntity(
↪$this->
↪request->
↪getData());
↪
        if (
↪$this->
↪Recipes->
↪save(
↪$recipe))
↪{

↪$message␣
↪= 'Saved';
        }␣
↪else {

↪$message␣
↪= 'Error';
        }

↪$this->
↪set([

↪'message'␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪=>
↪$message,

↪'recipe'␣
↪=>
↪$recipe,
        ]);

↪$this->
↪viewBuilder()-
↪>
↪setOption(
↪'serialize
↪', [
↪'recipe',
↪'message
↪']);
    }

    public␣
↪function␣
↪edit($id)
    {

↪$this->
↪request->
↪allowMethod([
↪'patch',
↪'post',
↪'put']);

↪$recipe =
↪$this->
↪Recipes->
↪get($id);

↪$recipe =
↪$this->
↪Recipes->
↪patchEntity(
↪$recipe,
↪$this->
↪request->
↪getData());
↪
        if (
↪$this->
↪Recipes->
↪save(
↪$recipe))
↪{
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪$message␣
↪= 'Saved';
        }␣
↪else {

↪$message␣
↪= 'Error';
        }

↪$this->
↪set([

↪'message'␣
↪=>
↪$message,

↪'recipe'␣
↪=>
↪$recipe,
        ]);

↪$this->
↪viewBuilder()-
↪>
↪setOption(
↪'serialize
↪', [
↪'recipe',
↪'message
↪']);
    }

    public␣
↪function␣
↪delete(
↪$id)
    {

↪$this->
↪request->
↪allowMethod([
↪'delete
↪']);

↪$recipe =
↪$this->
↪Recipes->
↪get($id);

↪$message␣
↪= 'Deleted
↪';
```

(continué en la próxima página)

(proviene de la página anterior)

```
        if␣
↪(!$this->
↪Recipes->
↪delete(
↪$recipe))
↪{

↪$message␣
↪= 'Error';
        }

↪$this->
↪set(
↪'message',
↪
↪$message);

↪$this->
↪viewBuilder()-
↪>
↪setOption(
↪'serialize
↪', [
↪'message
↪']);
    }
}
```

Los controladores RESTful a menudo usan extensiones parseadas para mostrar diferentes vistas, basado en diferentes tipos de solicitudes. Como estamos tratando con solicitudes REST, estaremos haciendo vistas XML. Puedes realizar vistas en JSON usando el CakePHP *Vistas JSON y XML*. Mediante el uso de `XmlView` se puede definir una opción de `serialize`. Esta opción se usa para definir qué variables de vistas `` XmlView`` deben serializarse en XML.

Si se quiere modificar los datos antes de convertirlos en XML, no se debería definir la opción `serialize`, y en lugar de eso, se debería usar archivos plantilla. Colocaremos las vistas REST de nuestro RecipesController dentro de **templates/Recipes/xml**. también podemos utilizar el `Xml` para una salida XML rápida y fácil en esas vistas. De esta forma, así podría verse nuestra vista de índice:

```
//␣
↪templates/
↪Recipes/
↪xml/index.
↪php
// Realizar␣
↪un␣
↪formateo␣
↪y␣
↪manipulacion␣
↪en
// $recipes␣
↪array.
$xml =␣
↪Xml::fromArray([
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪'response
↪' =>
↪$recipes]);
↪
echo $xml->
↪asXML();
```

Al entregar un tipo de contenido específico usando `Cake\Routing\Router::extensions()`, CakePHP busca automáticamente un asistente de vista que coincida con el tipo. Como estamos utilizando XML como tipo de contenido, no hay un asistente incorporado, sin embargo, si creara uno, se cargaría automáticamente para nuestro uso en esas vistas.

El XML procesado terminará pareciéndose a esto:

```
<recipes>
    <recipe>
        <id>
↪234</id>

↪<created>
↪2008-06-13
↪</created>

↪<modified>
↪2008-06-14
↪</
↪modified>

↪<author>

↪<id>23423
↪</id>

↪<first_
↪name>Billy
↪</first_
↪name>

↪<last_
↪name>Bob</
↪last_name>
        </
↪author>

↪<comment>

↪<id>245</
↪id>

↪<body>
↪Yummy␣
↪yummmy</
↪body>
        </
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪comment>
    </
↪recipe>
    ...
</recipes>
```

Crear la lógica para la acción de edición es un poco más complicado, pero no mucho. Ya que se está proporcionando una API que genera XML como salida, es una opción natural recibir XML como entrada. No te preocupes, las clases `Cake\Controller\Component\RequestHandler` y `Cake\Routing\Router` hacen las cosas mucho más fáciles. Si un POST o una solicitud PUT tiene un tipo de contenido XML, entonces la entrada se ejecuta a través de la clase de CakePHP `Xml`, y la representación del arreglo de los datos se asigna a `$this->request->getData()`. Debido a esta característica, el manejo de datos XML y POST se hace en continuamente en paralelo: no se requieren cambios en el controlador o el código del modelo. Todo lo que necesita debe terminar en `$this->request->getData()`.

Aceptando Entradas en otros formatos

Por lo general, las aplicaciones REST no solo generan contenido en formatos de datos alternativos, sino que también acepta datos en diferentes formatos. En CakePHP, el `RequestHandlerComponent` ayuda a facilitar esto. Por defecto, decodificará cualquier entrada de datos en JSON / XML para solicitudes POST / PUT y proporcionar una versión del arreglo de esos datos en `$this->request->getData()`. También puedes conectar deserializadores adicionales para formatos alternativos si los necesitas, usando: `RequestHandler::addInputType()`.

Enrutamiento RESTful

El enrutador de CakePHP fácilita la conexión de rutas de recursos RESTful. Ver la sección *resource-routes* para más información.

CAPÍTULO 24

Seguridad

CakePHP provee algunas herramientas para hacer que tu aplicación sea segura. Las siguientes secciones cubren estas herramientas:

Security

class Cake\
Utility\
Security

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[140] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[140] https://github.com/cakephp/docs

Protección CSRF

Las Solicitudes CSRF son una clase de ataque donde se realizan comandos no autorizados en nombre de un usuario autenticado sin su conocimiento o consentimiento.

CakePHP ofrece dos formas de protección CSRF:

- `SessionCsrfProtectionM` almacena tokens CSRF en la sesión. Esto requiere que tu aplicación abra la sesión en cada solicitud. Los beneficios de los tokens CSRF basados en sesiones son que están vinculados a un usuario específico y sólo son válidos durante la duración de una sesión activa.
- `CsrfProtectionMiddlewa` almacena tokens CSRF en una cookie. Usar una cookie permite realizar verificaciones CSRF sin

ningún estado en el servidor. Los valores de las cookies se verifican en términos de autenticidad mediante una comprobación HMAC. Sin embargo, debido a su naturaleza sin estado, los tokens CSRF se pueden reutilizar entre usuarios y sesiones.

Nota: No puedes usar ambos enfoques a la vez, debes elegir sólo uno. Si usas ambos, ocurrirá un error de incompatibilidad de tokens CSRF en cada solicitud *PUT* y *POST*.

Middleware CSRF

La protección CSRF se puede aplicar a toda tu aplicación o a ámbitos de enrutamiento específicos. Al aplicar un middleware CSRF a tu cola de middlewares, proteges todas las acciones en la aplicación:

```
// En src/
↪Application.
↪php
// Para␣
↪tokens␣
↪CSRF␣
↪basados␣
↪en␣
↪cookies.
use Cake\
↪Http\
↪Middleware\
↪CsrfProtectionMiddleware
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪

// Para␣
↪tokens␣
↪CSRF␣
↪basados␣
↪en␣
↪sesiones.
use Cake\
↪Http\
↪Middleware\
↪SessionCsrfProtectionMidd
↪

public␣
↪function␣
↪middleware(MiddlewareQueu
↪
↪$middlewareQueue):␣
↪MiddlewareQueue
{

↪$opciones␣
↪= [
        // .
↪..
    ];
    $csrf =␣
↪new␣
↪CsrfProtectionMiddleware(
↪$opciones);
↪
    // o
    $csrf =␣
↪new␣
↪SessionCsrfProtectionMidd
↪$opciones);
↪


↪$middlewareQueue-
↪>add(
↪$csrf);

    return
↪$middlewareQueue;
↪
}
```

Al aplicar la protección CSRF a los ámbitos de enrutamiento, puedes aplicar condicionalmente CSRF a grupos específicos de rutas:

```
// en src/
↪Application.
↪php
use Cake\
↪Http\
↪Middleware\
↪CsrfProtectionMiddleware;
↪

public␣
↪function␣
↪routes(RouteBuilder␣
↪$routes)␣
↪: void
{

↪$options␣
↪= [
        // .
↪..
    ];
    $routes-
↪>
↪registerMiddleware(
↪'csrf',␣
↪new␣
↪CsrfProtectionMiddleware(
↪$options));
↪
   ␣
↪parent::routes(
↪$routes);
}

// en␣
↪config/
↪routes.php
$routes->
↪scope('/',
↪ function␣
↪(RouteBuilder␣
↪$routes) {
    $routes-
↪>
↪applyMiddleware(
↪'csrf');
});
```

Opciones del middleware CSRF basado en cookies

Las opciones de configuración disponibles son:

- `cookieName`: El nombre de la cookie que se enviará. Por defecto es `csrfToken`.
- `expiry`: Cuánto tiempo debe durar el token CSRF. Por defecto es por la sesión del navegador.
- `secure`: Si la cookie se establecerá o no con la bandera Secure. Es decir, la cookie sólo se establecerá en una conexión HTTPS y cualquier intento sobre HTTP normal fallará. Por defecto es `false`.
- `httponly`: Si la cookie se

establecerá o no con la bandera HttpOnly. Por defecto es `false`. Antes de la versión 4.1.0, usa la opción `httpOnly`.

- `samesite`: Te permite declarar si tu cookie debe estar restringida a un contexto de primer partido o mismo sitio. Los valores posibles son `Lax`, `Strict` y `None`. Por defecto es `null`.
- `field`: El campo del formulario a comprobar. Por defecto es `_csrfToken`. Cambiar esto también requerirá configurar FormHelper.

Opciones del middleware CSRF basado en sesiones

Las opciones de configuración disponibles son:

- `key`: La clave de sesión a utilizar. Por defecto es *csrfToken*.
- `field`: El campo del formulario a comprobar. Cambiar esto también requerirá configurar FormHelper.

Cuando está habilitado, puedes acceder al token CSRF actual en el objeto de Request:

```
$token =
↪$this->
↪request->
↪getAttribute(
↪'csrfToken
↪');
```

Si necesitas rotar o reemplazar el token CSRF de la sesión, puedes hacerlo con:

```
$this->
↪request =␣
↪SessionCsrfProtectionMidd
↪$this->
↪request);
```

Nuevo en la versión 4.3.0: Se añadió el método `replaceToken`.

Omitir comprobaciones CSRF para acciones específicas

Ambas implementaciones de middleware CSRF te permiten usar la función `skip check` para un control más preciso sobre las URL para las cuales se debe hacer la comprobación de tokens CSRF:

```
// en src/
↪Application.
↪php
use Cake\
↪Http\
↪Middleware\
↪CsrfProtectionMiddleware;
↪

public␣
↪function␣
↪middleware(MiddlewareQueu
↪
↪$middlewareQueue):␣
↪MiddlewareQueue
{
    $csrf =␣
↪new␣
↪CsrfProtectionMiddleware(
↪

    // La␣
↪comprobación␣
↪del token␣
↪se␣
↪omitirá␣
↪cuando el␣
↪callback␣
↪devuelva␣
↪`true`.
    $csrf->
↪skipCheckCallback(functio
↪(
↪$request)
↪{
        //␣
↪Omitir la␣
↪comprobación␣
↪del token␣
↪para las␣
↪URL de la␣
↪API.
        if (
↪$request->
↪getParam(
↪'prefix')␣
↪=== 'Api
↪') {
```

(continué en la próxima página)

(proviene de la página anterior)

```
                ␣
↪return␣
↪true;
            }
    });

    //␣
↪Asegúrate␣
↪de que el␣
↪middleware␣
↪de␣
↪enrutamiento␣
↪se añada␣
↪a la cola␣
↪antes del␣
↪middleware␣
↪de␣
↪protección␣
↪CSRF.

↪$middlewareQueue-
↪>add(
↪$csrf);

    return
↪$middlewareQueue;
↪
}
```

Nota: Debes aplicar el middleware de protección CSRF solo para rutas que manejen solicitudes con estado que utilicen cookies/sesiones. Por ejemplo, al desarrollar una API, las solicitudes sin estado que no usan cookies para la autenticación no se ven afectadas por CSRF, por lo que el middleware no necesita aplicarse para esas rutas.

Integración con FormHelper

El `CsrfProtectionMiddleware` se integra perfectamente con `FormHelper`. Cada vez que creas un formulario con `FormHelper`, se insertará un campo oculto que contiene el token CSRF.

Nota: Cuando uses protección CSRF, siempre debes empezar tus formularios con `FormHelper`. Si no lo haces, deberás crear manualmente los campos ocultos en cada uno de tus formularios.

Protección CSRF y Solicitudes AJAX

Además de los parámetros de datos de la solicitud, los tokens CSRF se pueden enviar a través de un encabezado especial `X-CSRF-Token`. Usar un encabezado a menudo facilita la integración de un token CSRF con aplicaciones JavaScript, o *endpoints* de API basados en XML/JSON.

El Token CSRF se puede obtener en JavaScript a través de la Cookie `csrfToken`, o en PHP a través del atributo del objeto de Request llamado `csrfToken`. Usar la cookie puede ser más fácil cuando tu código JavaScript reside en archivos separados de las plantillas de vista de CakePHP, y cuando ya tienes funcionalidad para analizar cookies mediante JavaScript.

Si tienes archivos JavaScript separados, pero no quieres ocuparte de manejar cookies; podrías, por ejemplo, configurar el token en una variable global de JavaScript en tu diseño, mediante la definición de un bloque de script como este:

```
echo $this->
↪Html->
↪scriptBlock(sprintf(
    'var␣
↪csrfToken␣
↪= %s;',
    json_
↪encode(
↪$this->
↪request->
↪getAttribute(
↪'csrfToken
↪'))
));
```

Luego puedes acceder al token como `csrfToken` o `window.csrfToken` en cualquier archivo de script que se cargue después de este bloque de script.

Otra alternativa sería poner el token en una metaetiqueta personalizada como esta:

```
echo $this->
↪Html->
↪meta(
↪'csrfToken
↪', $this->
↪request->
↪getAttribute(
↪'csrfToken
↪'));
```

Que luego se podría acceder en tus scripts buscando el elemento `meta` con el nombre `csrfToken`, que podría ser tan simple como esto cuando se usa jQuery:

```
var␣
↪csrfToken␣
↪= $(
↪'meta[name=
↪"csrfToken
↪"]').attr(
↪'content
↪');
```

Content Security Policy Middleware

The `CspMiddleware` makes it simpler to add Content-Security-Policy headers in your application. Before using it you should install `paragonie/csp-builder`:

```
composer␣
↪require␣
↪paragonie/
↪csp-
↪builder
```

You can then configure the middleware using an array, or passing in a built `CSPBuilder` object:

```
use Cake\
↪Http\
↪Middleware\
↪CspMiddleware;
↪

$csp = new␣
↪CspMiddleware([
    'script-
↪src' => [

↪'allow' =>
↪ [

↪'https://
↪www.
↪google-
↪analytics.
↪com',
        ],

↪'self' =>␣
↪true,

↪'unsafe-
↪inline' =>
↪ false,

↪'unsafe-
↪eval' =>␣
↪false,
    ],
]);


↪$middlewareQueue-
↪>add(
↪$csp);
```

If you want to use a more strict CSP configuration, you can enable nonce based CSP rules with the `scriptNonce` and `styleNonce` options. When enabled these options will modify your CSP policy and set the `cspScriptNonce`

and `cspStyleNonce` attributes in the request. These attributes are applied to the `nonce` attribute of all script and CSS link elements created by `HtmlHelper`. This simplifies the adoption of policies that use a nonce-base64[141] and `strict-dynamic` for increased security and easier maintenance:

```
$policy = [
    // Must␣
↪exist␣
↪even if␣
↪empty to␣
↪set nonce␣
↪for for␣
↪script-src
    'script-
↪src' =>␣
↪[],
    'style-
↪src' =>␣
↪[],
];
// Enable␣
↪automatic␣
↪nonce␣
↪addition␣
↪to script␣
↪& CSS␣
↪link tags.
$csp = new␣
↪CspMiddleware(
↪$policy, [

↪'scriptNonce
↪' => true,

↪'styleNonce
↪' => true,
]);

↪$middlewareQueue-
↪>add(
↪$csp);
```

Middleware SecurityHeader

El `SecurityHeaderMiddleware` te permite agregar headers relacionados con la seguridad a tu aplicación. Una vez configurado, el middleware agregará los siguientes headers a las respuestas:

- `X-Content-Type-Options`
- `X-Download-Options`

[141] https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src

- `X-Frame-Options`
- `Referrer-Policy`

Este middleware debe ser configurado antes de agregarlo a la cola de middlewares:

```
use Cake\
↪Http\
↪Middleware\
↪SecurityHeadersMiddleware
↪


↪$securityHeaders␣
↪= new␣
↪SecurityHeadersMiddleware
↪

↪$securityHeaders
    ->
↪setReferrerPolicy()
    ->
↪setXFrameOptions()
    ->
↪noOpen()
    ->
↪noSniff();


↪$middlewareQueue-
↪>add(
↪$securityHeaders);
↪
```

Aqui está uina lista de Headers HTTP comunes[142], y la configuración recomendada[143] de Mozzilla para aplicaciones web seguras.

HTTPS Enforcer Middleware

If you want your application to only be available via HTTPS connections you can use the `HttpsEnforcerMiddleware`:

```
use Cake\
↪Http\
↪Middleware\
↪HttpsEnforcerMiddleware;
↪

// Always␣
↪raise an␣
```

(continué en la próxima página)

[142] https://en.wikipedia.org/wiki/List_of_HTTP_header_fields
[143] https://infosec.mozilla.org/guidelines/web_security.html

(proviene de la página anterior)

```
    ↪exception␣
    ↪and never␣
    ↪redirect.
    $https =␣
    ↪new␣
    ↪HttpsEnforcerMiddleware(

    ↪'redirect
    ↪' =>␣
    ↪false,
    ]);

    // Send a␣
    ↪302␣
    ↪status␣
    ↪code when␣
    ↪redirecting
    $https =␣
    ↪new␣
    ↪HttpsEnforcerMiddleware(

    ↪'redirect
    ↪' => true,

    ↪'statusCode
    ↪' => 302,
    ]);

    // Send␣
    ↪additional␣
    ↪headers␣
    ↪in the␣
    ↪redirect␣
    ↪response.
    $https =␣
    ↪new␣
    ↪HttpsEnforcerMiddleware(
        'headers
    ↪' => ['X-
    ↪Https-
    ↪Upgrade'␣
    ↪=> 1],
    ]);

    // Disable␣
    ↪HTTPs␣
    ↪enforcement␣
    ↪when␣
    ↪``debug``␣
    ↪is on.
    $https =␣
    ↪new␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪HttpsEnforcerMiddleware(

↪'disableOnDebug
↪' => true,
]);

// Only␣
↪trust␣
↪HTTP_X_␣
↪headers␣
↪from the␣
↪listed␣
↪servers.
$https =␣
↪new␣
↪HttpsEnforcerMiddleware(

↪'trustProxies
↪' => [
↪'192.168.
↪1.1'],
]);
```

If a non-HTTP request is received that does not use GET a `BadRequestException` will be raised.

NOTE: The Strict-Transport-Security header is ignored by the browser when your site has only been accessed using HTTP. Once your site is accessed over HTTPS with no certificate errors, the browser knows your site is HTTPS capable and will honor the Strict-Transport-Security header.

Adding Strict-Transport-Security

When your application requires SSL it is a good idea to set the `Strict-Transport-Security` header. This header value is cached in the browser, and informs browsers that they should always connect with HTTPS connections. You can configure this header with the `hsts` option:

```
$https =␣
↪new␣
↪HttpsEnforcerMiddleware(
    'hsts'␣
↪=> [
        //␣
↪How long␣
↪the␣
↪header␣
↪value␣
↪should be␣
↪cached␣
↪for.

↪'maxAge'␣
↪=> 60 *␣
↪60 * 24 *␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪365,
        //␣
↪should␣
↪this␣
↪policy␣
↪apply to␣
↪subdomains?
↪

↪'includeSubDomains
↪' => true,
        //␣
↪Should␣
↪the␣
↪header␣
↪value be␣
↪cacheable␣
↪in google
↪'s HSTS␣
↪preload
        //␣
↪service?␣
↪While not␣
↪part of␣
↪the spec␣
↪it is␣
↪widely␣
↪implemented.
↪

↪'preload'␣
↪=> true,
    ],
]);
```

CAPÍTULO 25

Sesiones

CakePHP proporciona una envoltura y una serie de funciones de utilidad sobre la extensión nativa de sesión de PHP. Las sesiones te permiten identificar usuarios únicos a lo largo de las solicitudes y almacenar datos persistentes para usuarios específicos. A diferencia de las cookies, los datos de sesión no están disponibles en el lado del cliente. En CakePHP, se evita el uso de *$_SESSION* y se prefiere el uso de las clases de Sesión.

Configuración de Sesión

La configuración de sesión generalmente se define en **/config/app.php**. Las opciones disponibles son:

- `Session.timeout` - El número de *minutos* antes de que el manejador de sesiones de CakePHP expire la sesión.
- `Session.defaults` - Te permite

usar las configuraciones de sesión predeterminadas incorporadas como base para tu configuración de sesión. Consulta a continuación para ver las configuraciones predeterminadas.

- `Session.handler` - Te permite definir un manejador de sesiones personalizado. Los manejadores de sesiones de base de datos y caché utilizan esto. Consulta a continuación para obtener información adicional sobre los manejadores de sesiones.

`Session.ini` - Te permite establecer configuraciones adicionales de ini de sesión para tu configuración. Esto, combinado con `Session.handler`, reemplaza las características de manejo de sesiones personalizadas de las versiones anteriores.

- `Session.cookie` - El nombre de la cookie que se utilizará. De forma predeterminada, se establece en el valor configurado para `session.name` en php.ini.
- `Session.cookiePath` - La ruta URL para

la cual se establece la cookie de sesión. Se mapea a la configuración `session.cookie_path` de php.ini. De forma predeterminada, se establece en la ruta base de la aplicación.

Las configuraciones predeterminadas de CakePHP establecen `session.cookie_secure` en `true`, cuando tu aplicación está en un protocolo SSL. Si tu aplicación se sirve tanto desde protocolos SSL como no SSL, podrías tener problemas con las sesiones que se pierden. Si necesitas acceder a la sesión en dominios SSL y no SSL, deberás deshabilitar esto:

```
Configure::write(
↪'Session',
↪ [

↪'defaults
↪' => 'php
↪',
    'ini' =>
↪ [

↪'session.
↪cookie_
↪secure' =>
↪ false
    ]
]);
```

A partir de la versión 4.0 de CakePHP, también se establece el atributo SameSite[144] en `Lax` de forma predeterminada para las cookies de sesión, lo que ayuda a proteger contra ataques CSRF. Puedes cambiar el valor predeterminado configurando la opción `session.cookie_samesite` en php.ini:

```
Configure::write(
↪'Session',
↪ [

↪'defaults
↪' => 'php
```

(continué en la próxima página)

[144] https://owasp.org/www-community/SameSite

(proviene de la página anterior)

```
↪',
    'ini' =>
↪ [

↪'session.
↪cookie_
↪samesite'␣
↪=> 'Strict
↪',
    ],
]);
```

La ruta de la cookie de sesión se establece de forma predeterminada en la ruta base de la aplicación. Para cambiar esto, puedes usar el valor de `session.cookie_path` en ini. Por ejemplo, si quieres que tu sesión persista en todos los subdominios, puedes hacerlo así:

```
Configure::write(
↪'Session',
↪ [

↪'defaults
↪' => 'php
↪',
    'ini' =>
↪ [

↪'session.
↪cookie_
↪path' =>
↪'/',

↪'session.
↪cookie_
↪domain' =>
↪ '.
↪tudominio.
↪com',
    ],
]);
```

De forma predeterminada, PHP configura la cookie de sesión para que caduque tan pronto como se cierre el navegador, independientemente del valor configurado en `Session.timeout`. El tiempo de espera de la cookie está controlado por el valor de `session.cookie_lifetime` en ini y se puede configurar así:

```
Configure::write(
↪'Session',
↪ [

↪'defaults
↪' => 'php
↪',
    'ini' =>
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪ [
        //␣
↪Invalidar␣
↪la cookie␣
↪después␣
↪de 30␣
↪minutos␣
↪sin␣
↪visitar
        //␣
↪ninguna␣
↪página en␣
↪el sitio.

↪'session.
↪cookie_
↪lifetime'␣
↪=> 1800
    ]
]);
```

La diferencia entre `Session.timeout` y el valor de `session.cookie_lifetime` es que este último depende de que el cliente diga la verdad acerca de la cookie. Si necesitas una comprobación de tiempo de espera más estricta, sin depender de lo que el cliente informa, deberías usar `Session.timeout`.

Ten en cuenta que `Session.timeout` corresponde al tiempo total de inactividad para un usuario (es decir, el tiempo sin visitar ninguna página donde se utilice la sesión), y no limita el total de minutos que un usuario puede permanecer en el sitio.

Manejadores de Sesiones Incorporados y Configuración

CakePHP viene con varias configuraciones de sesión incorporadas. Puedes usar estas configuraciones como base para tu configuración de sesión o crear una solución completamente personalizada. Para usar las configuraciones predeterminadas, simplemente configura la clave "defaults" con el nombre del predeterminado que deseas utilizar. Luego puedes anular cualquier configuración secundaria declarándola en tu configuración de Sesión:

```
Configure::write(
↪'Session',
↪ [

↪'defaults
↪' => 'php'
]);
```

Lo anterior usará la configuración de sesión "php" incorporada. Puedes agregar partes o la totalidad de ella haciendo lo siguiente:

```
Configure::write(
↪'Session',
↪ [
```

(continué en la próxima página)

(proviene de la página anterior)

```
    'defaults' => 'php',
    'cookie' => 'mi_app',
    'timeout' => 4320, // 3 días
]);
```

Lo anterior anula el tiempo de espera y el nombre de la cookie para la configuración de sesión "php". Las configuraciones incorporadas son:

- `php` - Guarda sesiones con las configuraciones estándar en tu archivo php.ini.
- `cake` - Guarda sesiones como archivos dentro de `tmp/sessions`. Esta es una buena opción cuando estás en hosts que no te permiten escribir fuera de tu propio directorio de inicio.
- `database` - Utiliza las sesiones de base de datos

incorporadas. Consulta a continuación para obtener más información.

- `cache` - Utiliza las sesiones de caché incorporadas. Consulta a continuación para obtener más información.

Manejadores de Sesiones

Los manejadores de sesiones también se pueden definir en el array de configuración de la sesión. Al definir la clave de configuración "handler.engine", puedes nombrar la clase o proporcionar una instancia del manejador. La clase/objeto debe implementar la interfaz nativa de PHP `SessionHandlerInterface`. Implementar esta interfaz permitirá que `Session` mapee automáticamente los métodos para el manejador. Tanto los manejadores de sesiones de base de datos como de caché utilizan este método para guardar sesiones. Las configuraciones adicionales

Para el manejador deben colocarse dentro del array del manejador. Luego puedes leer esos valores desde dentro de tu manejador:

```
'Session' =>
↪ [
    'handler
↪' => [

↪'engine'␣
↪=>
↪'DatabaseSession
↪',

↪'model' =>
↪
↪'SesionesPersonalizadas
↪',
    ],
]
```

Lo anterior muestra cómo podrías configurar el manejador de sesiones de base de datos con un modelo de aplicación. Al utilizar nombres de clases como tu "handler.engine", CakePHP esperará encontrar tu clase en el es-

pacio de nombres `Http\Session`. Por ejemplo, si tenías una clase `AppSessionHandler`, el archivo debería estar en **src/Http/Session/AppSessionHandler.php**, y el nombre de la clase debería ser `App\Http\Session\AppSessionHandler`. También puedes usar manejadores de sesiones desde dentro de plugins, estableciendo el motor en `MyPlugin.PluginSessionHandler`.

Sesiones de Base de Datos

Si necesitas usar una base de datos para almacenar los datos de tu sesión, configúralo de la siguiente manera:

```
'Session' =>
↪ [

↪'defaults
↪' =>
↪'database'
]
```

Esta configuración requiere una tabla de base de datos con este esquema:

```
CREATE␣
↪TABLE␣
↪`sessions`␣
↪(
  `id`␣
↪char(40)␣
↪CHARACTER␣
↪SET ascii␣
↪COLLATE␣
↪ascii_bin␣
↪NOT NULL,
  `created`␣
↪datetime␣
↪DEFAULT␣
↪CURRENT_
↪TIMESTAMP,
↪ --␣
↪Opcional
 ␣
↪`modified`␣
↪datetime␣
↪DEFAULT␣
↪CURRENT_
↪TIMESTAMP␣
↪ON UPDATE␣
↪CURRENT_
↪TIMESTAMP,
↪ --␣
↪Opcional
  `data`␣
↪blob␣
↪DEFAULT␣
↪NULL, --␣
↪para␣
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪PostgreSQL,
↪ usa␣
↪bytea en␣
↪lugar de␣
↪blob
  `expires`␣
↪int(10)␣
↪unsigned␣
↪DEFAULT␣
↪NULL,
  PRIMARY␣
↪KEY (`id`)
)␣
↪ENGINE=InnoDB␣
↪DEFAULT␣
↪CHARSET=utf8;
↪
```

Puedes encontrar una copia del esquema para la tabla de sesiones en el esqueleto de la aplicación[145] en **config/schema/sessions.sql**.

También puedes usar tu propia clase de `Table` para manejar el guardado de las sesiones:

```
'Session' =>
↪ [

↪'defaults
↪' =>
↪'database
↪',
    'handler
↪' => [

↪'engine'␣
↪=>
↪'DatabaseSession
↪',

↪'model' =>
↪
↪'SesionesPersonalizadas
↪',
    ],
]
```

Lo anterior le dirá a Session que use las configuraciones predeterminadas de "database" y especifica que una tabla llamada `SesionesPersonalizadas` será la encargada de guardar la información de la sesión en la base de datos.

[145] https://github.com/cakephp/app

Sesiones de Caché

La clase Cache se puede utilizar para almacenar sesiones también. Esto te permite almacenar sesiones en una caché como APCu o Memcached. Hay algunas advertencias al usar sesiones en caché, ya que si agotas el espacio de la caché, las sesiones comenzarán a caducar a medida que se eliminan registros.

Para usar sesiones basadas en caché, puedes configurar tu configuración de Sesión así:

```
Configure::write(
↪'Session',
↪ [

↪'defaults
↪' =>
↪'cache',
    'handler
↪' => [

↪'config'␣
↪=>
↪'session',
    ],
]);
```

Esto configurará Session para usar la clase `CacheSession` como el delegado para guardar las sesiones. Puedes usar la clave "config" para especificar qué configuración de caché usar. La configuración de caché predeterminada es `'default'`.

Bloqueo de Sesiones

El esqueleto de la aplicación viene preconfigurado con una configuración de sesión como esta:

```
'Session' =>
↪ [

↪'defaults
↪' => 'php
↪',
],
```

Esto significa que CakePHP manejará las sesiones según lo que esté configurado en tu `php.ini`. En la mayoría de los casos, esta será la configuración predeterminada, por lo que PHP guardará cualquier sesión recién creada como un archivo en, por ejemplo, `/var/lib/php/session`.

Pero esto también significa que cualquier tarea computacionalmente intensiva, como consultar un gran conjunto de datos combinado con una sesión activa, **bloqueará ese archivo de sesión**, lo que bloqueará a los usuarios para, por ejemplo, abrir una segunda pestaña de tu aplicación para hacer algo más mientras tanto.

Para evitar este comportamiento, tendrás que cambiar la forma en que CakePHP maneja las sesiones utilizando un manejador de sesiones diferente como *Sesiones de Caché* combinado con el *Motor Redis* u otro motor de caché.

Truco: Si deseas leer más sobre el Bloqueo de Sesiones, consulta aquí[146].

[146] https://ma.ttias.be/php-session-locking-prevent-sessions-blocking-in-requests/

Configuración de Directivas de ini

Las configuraciones predeterminadas incorporadas intentan proporcionar una base común para la configuración de sesiones. Es posible que necesites ajustar flags de ini específicos también. CakePHP expone la capacidad de personalizar las configuraciones de ini tanto para las configuraciones predeterminadas como para las personalizadas. La clave `ini` en las configuraciones de sesión te permite especificar valores de configuración individuales. Por ejemplo, puedes usarlo para controlar configuraciones como `session.gc_divisor`:

```
Configure::write(
↪'Session',
↪ [

↪'defaults
↪' => 'php
↪',
    'ini' =>
↪ [

↪'session.
↪cookie_
↪name' =>
↪'MiCookie
↪',

↪'session.
↪cookie_
↪lifetime'␣
↪=> 1800, /
↪/ Válido␣
↪por 30␣
↪minutos

↪'session.
↪gc_divisor
↪' => 1000,

↪'session.
↪cookie_
↪httponly'␣
↪=> true
    ]
]);
```

Creación de un Manejador de Sesiones Personalizado

Crear un manejador de sesiones personalizado es sencillo en CakePHP. En este ejemplo, crearemos un manejador de sesiones que almacene sesiones tanto en la Caché (APC) como en la base de datos. Esto nos brinda lo mejor de ambas opciones: la entrada/salida rápida de APC, sin tener que preocuparnos por las sesiones que desaparecen cuando la caché se llena.

Primero necesitamos crear nuestra clase personalizada y ponerla en **src/Http/Session/ComboSession.php**. La clase debería verse algo así:

```
namespace␣
↪App\Http\
↪Session;

use Cake\
↪Cache\
↪Cache;
use Cake\
↪Core\
↪Configure;
use Cake\
↪Http\
↪Session\
↪DatabaseSession;
↪

class␣
↪ComboSession␣
↪extends␣
↪DatabaseSession
{
    ␣
↪protected
↪$cacheKey;

    public␣
↪function _
↪_
↪construct()
    {

↪$this->
↪cacheKey␣
↪=␣
↪Configure::read(
↪'Session.
↪handler.
↪cache');
        ␣
↪parent::__
↪construct();
↪
    }
```

(continué en la próxima página)

(proviene de la página anterior)

```
    // Lee␣
↪datos de␣
↪la sesión.
    public␣
↪function␣
↪read(
↪$id):␣
↪string
    {

↪$result =␣
↪Cache::read(
↪$id,
↪$this->
↪cacheKey);
        if (
↪$result) {
            ␣
↪return
↪$result;
        }

        ␣
↪return␣
↪parent::read(
↪$id);


    }

    //␣
↪Escribe␣
↪datos en␣
↪la sesión.
    public␣
↪function␣
↪write($id,
↪ $data):␣
↪bool
    {
        ␣
↪Cache::write(
↪$id,
↪$data,
↪$this->
↪cacheKey);

        ␣
↪return␣
↪parent::write(
↪$id,
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪$data);
    }

    //␣
↪Destruye␣
↪una␣
↪sesión.
    public␣
↪function␣
↪destroy(
↪$id): bool
    {
        ␣
↪Cache::delete(
↪$id,
↪$this->
↪cacheKey);

        ␣
↪return␣
↪parent::destroy(
↪$id);
    }

    //␣
↪Elimina␣
↪sesiones␣
↪caducadas.
    public␣
↪function␣
↪gc(
↪$expires␣
↪= null):␣
↪bool
    {
        ␣
↪return␣
↪parent::gc(
↪$expires);
    }
}
```

Nuestra clase extiende el `DatabaseSession` incorporado para no tener que duplicar toda su lógica y comportamiento. Envolvemos cada operación con una operación de *Cake\Cache\Cache*. Esto nos permite obtener sesiones de la caché rápida y no tener que preocuparnos por lo que sucede cuando llenamos la caché. En **config/app.php** haz que el bloque de sesión se vea así:

```
'Session' =>
↪ [

↪'defaults
↪' =>
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪'database
↪',
    'handler
↪' => [

↪'engine'␣
↪=>
↪'ComboSession
↪',

↪'model' =>
↪ 'Session
↪',

↪'cache' =>
↪ 'apc',
    ],
],
//␣
↪Asegúrate␣
↪de␣
↪agregar␣
↪una␣
↪configuración␣
↪de caché␣
↪apc
'Cache' => [
    'apc' =>
↪ ['engine
↪' => 'Apc
↪']
]
```

Ahora nuestra aplicación comenzará a usar nuestro manejador de sesiones personalizado para leer y escribir datos de sesión.

class Sesión

Acceso al Objeto de Sesión

Puedes acceder a los datos de sesión en cualquier lugar donde tengas acceso a un objeto de solicitud. Esto significa que la sesión es accesible desde:

- Controladores
- Vistas
- Ayudantes (Helpers)

- Celdas (Cells)
- Componentes

Un ejemplo básico de uso de sesión en controladores, vistas y celdas sería:

```
$nombre =
↪$this->
↪request->
↪getSession()-
↪>read(
↪'Usuario.
↪nombre');

// Si␣
↪accedes a␣
↪la sesión␣
↪varias␣
↪veces,
//␣
↪probablemente␣
↪querrás␣
↪una␣
↪variable␣
↪local.
$sesion =
↪$this->
↪request->
↪getSession();
↪
$nombre =
↪$sesion->
↪read(
↪'Usuario.
↪nombre');
```

En los ayudantes, usa `$this->getView()->getRequest()` para obtener el objeto de solicitud; en los componentes, usa `$this->getController()->getRequest()`.

Lectura y Escritura de Datos de Sesión

`Sesión::read`(*$clave*, *$predeterminado = null*)

Puedes leer valores de la sesión utilizando una sintaxis compatible con `Hash::extract()`. Ejemplo:

```
$sesion->
↪read(
↪'Config.
↪idioma',
↪'es');
```

Sesión::**readOrFail**(*$clave*)

Lo mismo que una envoltura de conveniencia alrededor de un valor de retorno no nulo:

```
$sesion->
↪readOrFail(
↪'Config.
↪idioma');
```

Esto es útil cuando sabes que esta clave debe estar configurada y no deseas tener que comprobar su existencia en el código mismo.

Sesión::**write**(*$clave*, *$valor*)

`$clave` debería ser la ruta separada por puntos a la que deseas escribir `$valor`:

```
$sesion->
↪write(
↪'Config.
↪idioma',
↪'es');
```

También puedes especificar uno o varios hashes así:

```
$sesion->
↪write([
  'Config.
↪tema' =>
↪'azul',
  'Config.
↪idioma' =>
↪ 'es',
]);
```

Sesión::**delete**(*$clave*)

Cuando necesitas eliminar datos de la sesión, puedes usar `delete()`:

```
$sesion->
↪delete(
↪'Algo.
↪valor');
```

static Sesión::**consume**(*$cla*

Cuando necesitas leer y eliminar datos de la sesión, puedes usar `consume()`:

```
$sesion->
↪consume(
↪'Algo.
↪valor');
```

Sesión::**check**(*$clave*)

Si deseas ver si los datos existen en la sesión, puedes usar `check()`:

```
if ($sesion-
↪>check(
↪'Config.
↪idioma'))
↪{
    //␣
↪Config.
↪idioma␣
↪existe y␣
↪no es␣
↪nulo.
}
```

Destrucción de la Sesión

Sesión::**destroy**()

Destruir la sesión es útil cuando los usuarios cierran sesión. Para destruir una sesión, usa el método `destroy()`:

```
$sesion->
↪destroy();
```

Destruir una sesión eliminará todos los datos del lado del servidor en la sesión, pero **no** eliminará la cookie de la sesión.

Rotación de Identificadores de Sesión

Sesión::**renew**()

Mientras que el `Plugin de Autenticación` renueva automáticamente el ID de sesión cuando los usuarios inician sesión y cierran sesión, es posible que necesites rotar los ID de sesión manualmente. Para hacerlo, usa el método `renew()`:

```
$sesion->
↪renew();
```

Mensajes Flash

Los mensajes flash son pequeños mensajes que se muestran a los usuarios una vez. A menudo se utilizan para presentar mensajes de error o confirmar que las acciones se realizaron con éxito.

Para establecer y mostrar mensajes flash, debes usar el *Componente Flash* y *Ayudante Flash*.

CAPÍTULO 26

Testing

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[147] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

Running Tests

[147] https://github.com/cakephp/docs

CAPÍTULO 27

Validation

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[148] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[148] https://github.com/cakephp/docs

CAPÍTULO 28

La clase App

class Cake\Core**App**

La clase App se encarga de la localización de recursos y de la administración de rutas.

Búsqueda de clases

static Cake\Core\App::**class**

Éste método se utiliza para resolver el nombre completo de una clase en todo Cakephp. Como parámetros del método entran los nombre cortos que usa CakePHP y devuelve el nombre completo (La ruta relativa al espacio de trabajo):

```
// Resuelve␣
↪el nombre␣
↪de clase␣
↪corto␣
↪utilizando␣
↪el nombre␣
↪y el␣
↪sufijo.
```

(continué en la próxima página)

(proviene de la página anterior)

```
App::classname(
↪'Auth',
↪'Controller/
↪Component
↪',
↪'Component
↪');
// Salida:␣
↪Cake\
↪Controller\
↪Component\
↪AuthComponent

// Resuelve␣
↪el nombre␣
↪de plugin.
App::classname(
↪'DebugKit.
↪Toolbar',
↪'Controller/
↪Component
↪',
↪'Component
↪');
// Salida:␣
↪DebugKit\
↪Controller\
↪Component\
↪ToolbarComponent

// Nombres␣
↪con '\' se␣
↪devuelven␣
↪inalterados.
↪
App::classname(
↪'App\
↪Cache\
↪ComboCache
↪');
// Salida:␣
↪App\Cache\
↪ComboCache
```

A la hora de resolver clases, primero se prueba con el espacio de nombres de `App`, si no existe, se prueba con el espacio de nombres de `Cake` . Si no existe ninguno, devuelve `false`.

Búsqueda de rutas al espacio de nombres

static Cake\Core\App::**path**

Se usa para la búsqueda de rutas basada en convenio de nombres de CakePHP:

```
// Buscar
↪la ruta
↪de
↪Controller/
↪ en tu
↪aplicación
App::path(
↪'Controller
↪');
```

Se puede utilizar para todos los espacios de nombres de tu aplicacón. Además puedes extraer rutas de plugins:

```
// Devuelve
↪la ruta
↪del
↪'Component
↪' en
↪DebugKit
App::path(
↪'Component
↪',
↪'DebugKit
↪');
```

`App::path()` sólo devuelve la ruta por defecto,no mostrará ningún tipo de información sobre las rutas adicionales configuadas en autoloader.

static Cake\Core\App::**core**

Se usa para buscar rutas de paquetes dentro del core de Cakephp:

```
// Devuelve
↪la ruta
↪de engine
↪de cake.
App::core(
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪'Cache/
↪Engine');
```

Búsqueda de plugins

static Cake\Core\Plugin::**pa**

Los plugins se localizan con el método Plugin. Por ejemplo, `Plugin::path('DebugKit');` devuelve la ruta completa al plugin DebugKit:

```
$path =␣
↪Plugin::path(
↪'DebugKit
↪');
```

Localización de temas (nota:"themes")

Dado que los temas (nota:"themes") son también plugins, se localizan con el método anterior, «Plugin». (nota:"Aquí se refiere a los themes que se pueden crear para modificar el comportamiento del bake, generador de código.")

Cargar archivos externos (nota: "vendor")

Lo ideal es que los archivos externos ("vendor") se carguen automáticamente usando `Composer`, si necesita archivos externos que no se pueden cargar automáticamente o no se pueden instalar con el Composer, entonces hay que usar `require` para cargarlos.

Si no puede instalar alguna librería con el Composer, debería instalar cada librería en el directorio apropiado, siguiendo el convenio del Composer: `vendor/$author/$package`. Si tiene una librería de autor "Acme" que se llama "AcmeLib", la tiene que instalar en: `vendor/Acme/AcmeLib`. Asumiendo que la librería no usa nombres de clase compatibles con "PSR-0", puede cargar las clases definiéndolas en el `classmap`, dentro del archivo: `composer.json` en su aplicación:

```
"autoload":
↪{
    "psr-4
↪": {

↪"App\\":
↪"App",

↪"App\\
↪Test\\":
↪"Test",
        "":
↪"./Plugin"
```

(continué en la próxima página)

(proviene de la página anterior)

```
    },

↪"classmap
↪": [

↪"vendor/
↪Acme/
↪AcmeLib"
    ]
}
```

Si la librería no usa clases y sólo proporciona métodos,puede configurar el Composer para que cargue esos archivos al inicio de cada petición(“request”), usando la estrategia de carga automática de ficheros `files`, como sigue:

```
"autoload":
↪{
    "psr-4
↪": {

↪"App\\":
↪"App",

↪"App\\
↪Test\\":
↪"Test",
        "":
↪"./Plugin"
    },
    "files
↪": [

↪"vendor/
↪Acme/
↪AcmeLib/
↪functions.
↪php"
    ]
}
```

Después de la configuración de las librerías externas, tiene que regenerar el autoloader de su aplicación usando:

```
$ php␣
↪composer.
↪phar dump-
↪autoload
```

Si no usa Composer en su aplicación, tendrá que cargar manualmente cada librería en su aplicación.

CAPÍTULO 29

Collections

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[149] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[149] https://github.com/cakephp/docs

CAPÍTULO 30

Hash

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[150] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[150] https://github.com/cakephp/docs

CAPÍTULO 31

Http Client

class Cake\Network\Http**Cli**

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[151] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[151] https://github.com/cakephp/docs

CAPÍTULO 32

Inflector

class Cake\
Utility\
Inflector

La clase *Inflector* toma una cadena y puede manipularla para manejar variaciones de palabras como pluralización o conversión a formato camello (camelCase). Por lo general, se accede a esta clase de manera estática. Por ejemplo:

`Inflector::pluralize('example')` devuelve «examples».

Puedes probar las inflecciones en línea en inflector.cakephp.org[152] or sandbox.dereuromark.de[153].

[152] https://inflector.cakephp.org/
[153] https://sandbox.dereuromark.de/sandbox/inflector

Métodos integrados en Inflector y su resultado

Los métodos integrados en el Inflector y los resultados que generan al proporcionarles un argumento compuesto por varias palabras:

Método	Argumento	Resultado
plu	BigApple	BigApples
	big_apple	big_apples
sin	BigApples	BigApple
	big_apples	big_apple
cam	big_apple	BigApples
	big apple	BigApple
und	BigApples	big_apples
	Big Apples	big apples
hum	big_apples	Big Apples
	bigApple	BigApple
cla	big_apple	BigApples
	big apple	BigApple
das	BigApples	big-apples
	big apple	big apple
tab	BigApple	big_apples
	Big Apple	big apples
var	big_apple	bigApple
	big apples	bigApples

Generando formas Plural y Singular

```
static Cake\Utility\Inflect

static Cake\Utility\Inflect
```

Both `pluralize` and `singularize()` work on most English nouns. If you need to support other languages, you can use *Configurando las Inflecciones* to customize the rules used:

```
// Apples
echo␣
↪Inflector::pluralize(
↪'Apple');
```

Nota: `pluralize()` no debería ser usado en un nombre que ya está en su forma plural.

```
// Person
echo␣
↪Inflector::singularize(
↪'People');
```

Nota: `singularize()` no debería ser usado en un nombre que ya está en su forma singular.

Generando formas CamelCase y under_scored

```
static Cake\Utility\Inflect

static Cake\Utility\Inflect
```

Estos métodos son útiles cuando creas nombres de clases o de propiedades:

```
// ApplePie
Inflector::camelize(
↪'Apple_pie
↪')

// apple_pie
Inflector::underscore(
↪'ApplePie
↪');
```

Nótese que el método *underscore* sólo convertirá palabras en formato *CamelCase*. Palabras que contengan espacios serán transformadas a minúscula pero no contendrán un guión bajo.

Generando formas legibles por humanos

```
static Cake\Utility\Inflect
```

Este método es útil cuando se quiere convertir una palabra de la forma *under_scored* al formato «Título» para que sea legible por un ser humano:

```
// Apple Pie
Inflector::humanize(
↪'apple_pie
↪');
```

Generando formas de tabla y nombre de clase

```
static Cake\Utility\Inflect
```

```
static Cake\Utility\Inflect
```

```
static Cake\Utility\Inflect
```

Cuando se genera código, o usando las convenciones de CakePHP, puedes necesitar generar inflecciones para los nombres de tabla o de clase:

```
//␣
↪UserProfileSetting
Inflector::classify(
↪'user_
↪profile_
↪settings
↪');

// user-
↪profile-
↪setting
Inflector::dasherize(
↪'UserProfileSetting
↪');

// user_
↪profile_
↪settings
Inflector::tableize(
↪'UserProfileSetting
↪');
```

Generando Nombres de Variables

static Cake\Utility\Inflect

Los nombres de variable son a menudo útiles cuando se hacen tareas de meta-programación que involucran generar código o hacer trabajo basado en convenciones:

```
// applePie
Inflector::variable(
↪'apple_pie
↪');
```

Configurando las Inflecciones

Las convenciones de nomenclatura de CakePHP pueden ser muy útiles: puedes nombrar tu tabla de base de datos como `big_boxes`, tu modelo como `BigBoxes`, tu controlador como `BigBoxesController`, y todo funcionará automáticamente juntos. La forma en que CakePHP sabe cómo vincular las cosas es *inflectando* las palabras entre sus formas singular y plural.

Existen ocasiones (especialmente para nuestros amigos que no hablan inglés) en las que podrías encontrarte con situaciones donde el inflector de CakePHP (la clase que pluraliza, singulariza, utiliza notación camello y subrayados) puede no funcionar como deseas. Si CakePHP no reconoce tus «Foci» o «Fish», puedes indicarle a CakePHP acerca de tus casos especiales.

Cargando Inflecciones Personalizadas

static Cake\Utility\Inflect

Define nuevas reglas de inflexión y transliteración para que Inflector las utilice. A menudo, este método se utiliza en tu archivo **config/bootstrap.php**:

```
Inflector::rules(
↪'singular
↪', ['/^
↪(bil)er$/i
↪' => '\1',
↪ '/^
↪(inflec|contribu)tors
↪$/i' => '\
↪1ta']);
Inflector::rules(
↪'uninflected
↪', [
↪'singulars
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪']);
Inflector::rules(
↪'irregular
↪', [
↪'phylum'␣
↪=> 'phyla
↪']); //␣
↪The key␣
↪is␣
↪singular␣
↪form,␣
↪value is␣
↪plural␣
↪form
```

Las reglas suministradas se fusionarán en los conjuntos de inflexión respectivos definidos en `Cake/Utility/Inflector`, y las reglas añadidas tendrán prioridad sobre las reglas principales del núcleo. Puedes usar `Inflector::reset()` para eliminar las reglas y restaurar el estado original del Inflector.

CAPÍTULO 33

Clase Number

class Cake\
I18n**Number**

Si necesitas las funcionalidades de `NumberHelper` fuera de una vista, utiliza la clase `Number`:

```
namespace␣
↪App\
↪Controller;
↪

use Cake\
↪I18n\
↪Number;

class␣
↪UsersController␣
↪extends␣
↪AppController
{
    public␣
↪function␣
↪initialize():␣
↪void
    {
        ␣
↪parent::initialize();
↪

↪$this->
↪loadComponent(
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪'Authentication.
↪Authentication
↪');
    }

    public␣
↪function␣
↪afterLogin()
    {

↪$identity␣
↪= $this->
↪Authentication-
↪>
↪getIdentity();
↪

↪$storageUsed␣
↪=
↪$identity-
↪>storage_
↪used;
        if (
↪$storageUsed␣
↪>␣
↪5000000) {
            ␣
↪//␣
↪Notificar␣
↪a los␣
↪usuarios␣
↪de su␣
↪cuota

↪$this->
↪Flash->
↪success(__
↪('Estás␣
↪usando {0}
↪ de␣
↪almacenamiento
↪',␣
↪Number::toReadableSize(
↪$storageUsed)));
↪
        }
    }
}
```

Todas estas funciones devuelven el número formateado; no imprimen automáticamente la salida en la vista.

Formato de Valores Monetarios

Cake\I18n\Number::**currency**(

Este método se utiliza para mostrar un número en formatos de moneda comunes (EUR, GBP, USD), basándose en el código de moneda de tres letras ISO 4217. Su uso en una vista se ve así:

```
// Llamado␣
↪como␣
↪NumberHelper
echo $this->
↪Number->
↪currency(
↪$value,
↪$currency);
↪

// Llamado␣
↪como␣
↪Number
echo␣
↪Number::currency(
↪$value,
↪$currency);
↪
```

El primer parámetro, `$value`, debería ser un número de punto flotante que representa la cantidad de dinero que estás expresando. El segundo parámetro es una cadena utilizada para elegir un esquema de formato de moneda predefinido:

$currenc	1234,56, formateado por tipo de moneda
EUR	€1.234,56
GBP	£1.234,56
USD	$1.234,56

El tercer parámetro es un arreglo de opciones para definir aún más la salida. Las siguientes opciones están disponibles:

Opción	Descripción
before	Texto para mostrar antes del número formateado.
after	Texto para mostrar después del número formateado.
zero	El texto a usar para los valores cero; puede ser una cadena o un número, por ejemplo, 0, "¡Gratis!".
places	Número de lugares decimales a usar, por ejemplo, 2
precision	Número máximo de lugares

Si el valor de `$currency` es `null`, la moneda predeterminada se recuperará de `Cake\I18n\Number::defaultCurrency()`. Para formatear monedas en un formato de contabilidad, debes establecer el formato de la moneda:

```
Number::setDefaultCurrencyF
↪CURRENCY_
↪ACCOUNTING);
↪
```

Configurar la Moneda Predeterminada

Cake\I18n\Number::**setDefaul**

Configura la moneda predeterminada. Esto evita la necesidad de pasar siempre la moneda a *`Cake\I18n\Number::currency()`* y cambiar todas las salidas de moneda configurando otro valor predeterminado. Si `$currency` se establece en `null`, se eliminará el valor almacenado actualmente.

Obtener la Moneda Predeterminada

Cake\I18n\Number::**getDefaul**

Obtén la moneda predeterminada. Si la moneda predeterminada se configuró anteriormente utilizando `setDefaultCurrency()`, se devolverá ese valor. De forma predeterminada, recuperará el valor de la ini de `intl.default_locale` si está configurado y `'en_US'` si no lo está.

Formato de Números de Punto Flotante

Cake\I18n\Number::**precisio**

Este método muestra un número con la cantidad especificada de precisión (lugares decimales). Se redondeará para mantener el nivel de precisión definido.

```
// Llamado␣
↪como␣
↪NumberHelper
```

(continué en la próxima página)

(proviene de la página anterior)

```
echo $this->
↪Number->
↪precision(456.
↪91873645,␣
↪2);

// Salida
456.92

// Llamado␣
↪como␣
↪Number
echo␣
↪Number::precision(456.
↪91873645,␣
↪2);
```

Formato de Porcentajes

Cake\I18n\Number::**toPercent**

Opción	Descripción
multiply	Booleano para indicar si el valor debe ser multiplicado por 100. Útil para porcentajes decimales.

Al igual que *Cake\I18n\Number::precision()*, este método formatea un número según la precisión proporcionada (donde los números se redondean para cumplir con la precisión dada). Adicionalmente, también expresa el número como un porcentaje y agrega un signo de porcentaje a la salida.

```
// Llamado␣
↪como␣
↪NumberHelper.
↪ Salida:␣
↪45.69%
echo $this->
↪Number->
↪toPercentage(45.
↪691873645);
↪

// Llamado␣
↪como␣
↪Number.␣
↪Salida:␣
↪45.69%
echo␣
↪Number::toPercentage(45.
↪691873645);
↪

// Llamado␣
↪con␣
↪multiplicar.
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪ Salida:␣
↪45.7%
echo␣
↪Number::toPercentage(0.
↪45691, 1,␣
↪[

↪'multiply
↪' => true
]);
```

Interactuar con Valores Legibles para Humanos

Cake\I18n\Number::**toReadabl**

Este método formatea tamaños de datos en formas legibles para humanos. Proporciona una forma abreviada de convertir bytes a KB, MB, GB y TB. El tamaño se muestra con un nivel de precisión de dos dígitos, de acuerdo con el tamaño de los datos suministrados (es decir, los tamaños más altos se expresan en términos más grandes):

```
// Llamado␣
↪como␣
↪NumberHelper
echo $this->
↪Number->
↪toReadableSize(0);
↪ // 0 Byte
echo $this->
↪Number->
↪toReadableSize(1024);
↪ // 1 KB
echo $this->
↪Number->
↪toReadableSize(1321205.
↪76); // 1.
↪26 MB
echo $this->
↪Number->
↪toReadableSize(536870912
↪ // 5 GB

// Llamado␣
↪como␣
↪Number
echo␣
↪Number::toReadableSize(0
↪ // 0 Byte
echo␣
↪Number::toReadableSize(1
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪ // 1 KB
echo␣
↪Number::toReadableSize(1
↪76); // 1.
↪26 MB
echo␣
↪Number::toReadableSize(5
↪ // 5 GB
```

Formato de Números

Cake\I18n\Number::**format**(*m*
$v
lu
a
$o
ti
=
[]

Este método te brinda mucho más control sobre el formato de números para usar en tus vistas (y se utiliza como el método principal por la mayoría de los otros métodos de NumberHelper). Usar este método puede verse así:

```
// Llamado␣
↪como␣
↪NumberHelper
$this->
↪Number->
↪format(
↪$value,
↪$options);

// Llamado␣
↪como␣
↪Number
Number::format(
↪$value,
↪$options);
```

El parámetro `$value` es el número que estás planeando formatear para la salida. Sin opciones proporcionadas, el número 1236.334 se mostraría como 1,236. Ten en cuenta que la precisión predeterminada es cero decimales.

El parámetro `$options` es donde reside la verdadera magia para este método.

- Si pasas un entero, este se convierte en la cantidad de precisión o lugares

para la función.

- Si pasas un arreglo asociado, puedes usar las siguientes claves:

Opción	Descripción
places	Número de lugares decimales a usar, por ejemplo, 2
precision	Número máximo de lugares decimales a usar, por ejemplo, 2.
pattern	Un patrón de número ICU para usar para formatear el número, por ejemplo, #,###.00.
locale	El nombre de la localidad a usar para formatear el número, por ejemplo, «es_ES».
before	Texto para mostrar

Ejemplo:

```
// Llamado como NumberHelper
echo $this->Number->format('123456.7890', [
    'places' => 2,
    'before' => '¥ ',
    'after' => ' !'
]);
// Salida ¥123,456.79 !'

echo $this->Number->format('123456.7890', [
    'locale' => 'fr_FR'
]);
// Salida '123 456,79 !'

// Llamado como Number
echo Number::format('123456.7890', [
    'places' => 2,
    'before' => '¥ ',
    'after' => ' !'
]);
// Salida ¥123,456.79 !'

echo
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪Number::format(
↪'123456.
↪7890', [
    'locale
↪' => 'fr_
↪FR'
]);
// Salida
↪'123 456,
↪79 !'
```

Cake\I18n\Number::**ordinal**(

Este método mostrará un número ordinal.

Ejemplos:

```
echo␣
↪Number::ordinal(1);
↪
// Salida
↪'1st'

echo␣
↪Number::ordinal(2);
↪
// Salida
↪'2nd'

echo␣
↪Number::ordinal(2,
↪ [
    'locale
↪' => 'fr_
↪FR'
]);
// Salida '2e
↪'

echo␣
↪Number::ordinal(410);
↪
// Salida
↪'410th'
```

Diferencias en el Formato

Cake\I18n\Number::formatDel

Este método muestra diferencias en el valor como un número con signo:

```
// Llamado␣
↪como␣
↪NumberHelper
$this->
↪Number->
↪formatDelta(
↪$value,
↪$options);

// Llamado␣
↪como␣
↪Number
Number::formatDelta(
↪$value,
↪$options);
```

El parámetro `$value` es el número que estás planeando formatear para la salida. Sin opciones proporcionadas, el número 1236.334 se mostraría como 1,236. Ten en cuenta que la precisión predeterminada es cero decimales.

El parámetro `$options` toma las mismas claves que *Number::format()* en sí:

Opción	Descripción
places	Número de lugares decimales a usar, por ejemplo, 2
precision	Número máximo de lugares decimales a usar, por ejemplo, 2.
locale	El nombre de la localidad a usar para formatear el número, por ejemplo, «es_ES».
before	Texto para mostrar antes del número formateado.
after	Texto para mostrar después del número

Ejemplo:

```
// Llamado como NumberHelper
echo $this->Number->formatDelta('123456.7890', [
    'places' => 2,
    'before' => '[',
    'after' => ']'
]);
// Salida '[+123,456.79]'

// Llamado como Number
echo Number::formatDelta('123456.7890', [
    'places' => 2,
    'before' => '[',
    'after' => ']'
]);
// Salida '[+123,456.79]'
```

Configurar Formateadores

Cake\I18n\Number::**config**(*st*
$l
co
le
in
$t
pe
=
N
be
Fo
m
te
ar
$c
tie
=
[]

Este método te permite configurar valores predeterminados del formateador que persisten en llamadas a varios métodos.

Ejemplo:

```
Number::config(
↪'es_ES', \
↪NumberFormatter::CURRENCY
↪ [
    'pattern
↪' => '#,#
↪#,##0'
]);
```

CAPÍTULO 34

Registry Objects

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[154] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[154] https://github.com/cakephp/docs

CAPÍTULO 35

Text

```
class Cake\
Utility\Text
```

```
static Cake\
Utility\
Text::uuid
```

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[155] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[155] https://github.com/cakephp/docs

CAPÍTULO 36

Time

```
class Cake\
Utility\Time
```

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[156] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[156] https://github.com/cakephp/docs

CAPÍTULO 37

Xml

```
class Cake\
Utility\Xml
```

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[157] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[157] https://github.com/cakephp/docs

CAPÍTULO 38

Constants & Functions

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[158] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[158] https://github.com/cakephp/docs

CAPÍTULO 39

Debug Kit

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[159] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[159] https://github.com/cakephp/docs

CAPÍTULO 40

Migrations

Nota: La documentación no es compatible actualmente con el idioma español en esta página.

Por favor, siéntase libre de enviarnos un pull request en Github[160] o utilizar el botón **Improve this Doc** para proponer directamente los cambios.

Usted puede hacer referencia a la versión en Inglés en el menú de selección superior para obtener información sobre el tema de esta página.

[160] https://github.com/cakephp/docs

CAPÍTULO 41

Apéndices

En los apéndices encontrarás información relacionada con las nuevas características introducidas en cada versión, así como también las guías de migración entre versiones.

Guía de Migración a 5.x

5.0 Guía de actualización

En primer lugar, compruebe que su aplicación se está ejecutando en la última versión de CakePHP 4.x.

Arreglar avisos de obsolescencia

Una vez que su aplicación se ejecuta en la última version de CakePHP 4.x, activar advertencias de obsoletos en **config/app.php**:

```
'Error' => [

↪'errorLevel
↪' => E_
↪ALL,
]
```

Ahora que puede ver todas las advertencias, asegúrese de que están corregidas antes de proceder con la actualización.

Algunas obsolescencia potencialmente impactantes que debes asegurarte de haber abordado son:

- `Table::query()`

was deprecated in 4.5.0. Use `selectQuery()`, `updateQuery()`, `insertQuery()` and `deleteQuery()` instead.

Actualiza a PHP 8.1

Si no estas ejecutando en **PHP 8.1 o superior**, tendrás que actualizar PHP antes de actualizar CakePHP.

Nota: CakePHP 5.0 requiere **un mínimo de PHP 8.1**.

Usar la herramienta de actualización

Nota: La herramienta de actualización sólo funciona en aplicaciones que se ejecutan en cakePHP 4.x. No puedes ejecutar la herramienta de actualización después de actualizar a CakePHP 5.0.

Debido a que CakePHP 5 aprovecha los tipos de unión y `mixed`, existen muchos cambios incompatibles con versiones anteriores relativas a las definiciones de los métodos y cambios de nombre archivos. Para ayudar a acelerar los arreglos de estos cambios tediosos, existe una herramienta CLI de actualización:

```
# Instalar la herramienta de actualización
git clone https://github.com/cakephp/upgrade
cd upgrade
git checkout 5.x
composer install --no-dev
```

Con la herramienta de actualización instalada, ahora puedes ejecutarla en su aplicación o plugin:

```
bin/cake upgrade
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪rector --
↪rules␣
↪cakephp50
↪<path/to/
↪app/src>
bin/cake␣
↪upgrade␣
↪rector --
↪rules␣
↪chronos3
↪<path/to/
↪app/src>
```

Actualizar dependencias de CakePHP

Después de aplicar las refactorizaciones de Rector necesitas actualizar CakePHP, sus plugins, PHPUnit y tal vez otras dependencias en el `composer.json`. Este proceso depende de gran medida de tu aplicación por lo que te recomendamos que compares el `composer.json` con el que está presente en cakephp/app[161].

After the version strings are adjusted in your `composer.json` execute `composer update -W` and check its output.

Actualiza los archivos de la aplicación basándose en las últimas plantillas

A continuación, asegúrate de que el resto de tu aplicación esté actualizado basándose en la última version de cakephp/app[162].

5.0 Guía de migración

CakePHP 5.0 contiene cambios importantes, y no es compatible con versiones anteriores de 4.x. Antes de intentar actualizar a la version 5.0, primero actualice a la version 4.5 y resuelva todas las advertencias de obsolescencia.

Consulte *5.0 Guía de actualización* para obtener instrucciones paso a paso de como actualizar a la versión 5.0.

Características obsoletas eliminadas

Todos los métodos, propiedades y funcionalidades que emitían advertencias de obsolencias a partir de la versión 4.5 se han eliminado.

[161] https://github.com/cakephp/app/blob/5.x/composer.json
[162] https://github.com/cakephp/app/blob/5.x/

Cambios importantes

Además de la eliminación de características obsoletas, se han realizado cambios importantes:

Global

- Se han añadido declaraciones de tipo a todos los parámetros de función y devoluciones siempre que ha sido posible. Estos están pensados para que coincidan con las anotaciones de docblock, pero incluyen correcciones para anotaciones incorrectas.
- Se han añadido declaraciones de tipo a todas las propiedades de clase siempre que ha sido posible. También se han

corregido algunas anotaciones incorrectas.

- Se han eliminado las constantes `SECOND`, `MINUTE`, `HOUR`, `DAY`, `WEEK`, `MONTH`, `YEAR`.
- Las funciones globales son ahora opcionales. Si tu aplicación utiliza alias de funciones globales, asegúrase de añadir `require CAKE . 'functions.php'` al `config/bootstrap.php` de tu aplicación.
- **Se ha eliminado el uso de `#[\AllowDynamicProperties]` en todas las partes. Se utilizaba para las siguientes clases:**

- `Command/Command`
- `Console/Shell`
- `Controller/Component`
- `Controller/Controller`
- `Mailer/Mailer`
- `View/Cell`
- `View/Helper`
- `View/View`

- **Se han actualizado las versiones compatibles del motor de base de datos:**
  - MySQL (5.7 o superior)
  - MariaDB (10.1 o superior)

- PostgreSQL (9.6 o superior)
- Microsoft SQL Server (2012 o superior)
- SQLite 3

Auth

- *Auth* ha sido eliminado. Usa los plugins cakephp/authentication[163] y cakephp/authorization[164] en su lugar.

[163] https://book.cakephp.org/authentication/2/es/index.html
[164] https://book.cakephp.org/authorization/2/es/index.html

Cache

- El motor `Wincache` ha sido eliminado. La extension wincache no es compatible con PHP 8.

Consola

- `BaseCommand::__construc` ha sido eliminado.
- Se ha eliminado `ConsoleIntegrationTest` porque ya no es necesario.
- `Shell` Ha sido eliminado y debe ser sustituido por Command[165]
- Ahora `BaseCommand` emite los eventos `Command.beforeExecute` and `Command.afterExecute` cuando el método `execute()` del comando es invocado

[165] https://book.cakephp.org/5/es/console-commands/commands.html

por el framework.

Connection

- Se ha eliminado `Connection::prepare()`. En su lugar, puede utilizar `Connection::execute()` para ejecutar una consulta SQL especificando en la cadena SQL los parámetros y los tipos en una sola llamada.
- Se ha eliminado `Connection::enableQuer` Si no ha habilitado el registro a través de la configuración de conexión, puedes configurar más adelante la instancia del registrador para que el controlador habilite el registro de consultas `$connection->getDriver`

Controlador

- La firma del método para `Controller::__construc` ha cambiado. Por lo tanto, tienes que ajustar el código en consecuencia si estás sobreescribiendo el constructor.
- Después de la carga, los componentes ya no se establecen como propiedades dinámicas. En su lugar `Controller` usa `__get()` para proporcionar acceso a las propiedades de los componentes. Este cambio puede afectar a las aplicaciones que usan `property_exists()` en los componentes.
- Se ha re-

nombrado la devolución de llamada del evento `Controller.shutdown` de los componentes de `shutdown` a `afterFilter` para que coincida con el del controlador. Esto hace que las devoluciones de llamada sean más coherentes.

- `PaginatorComponent` ha sido eliminado y tienes que reemplazarlo llamando a `$this->paginate()` en tu controlador o usando `Cake\Datasource\Paging\NumericPaginator` directamente.
- `RequestHandlerComponer` ha sido eliminado. Consulte la guía 4.4 mi-

gration[166] para saber como actualizarlo.

- Se ha eliminado `SecurityComponent`. Usa `FormProtectionComponer` para la protección contra la manipulación de formularios o `HttpsEnforcerMiddlewar` para forzar el uso de solicitudes HTTPS en su lugar.
- `Controller::paginate()` ya no acepta opciones de consulta como `contain` para su argumento `$settings`. En su lugar debes usar la opción `finder` `$this->paginate($this-` `['finder' => 'published'])`. O puede crear la consulta requerida de an-

[166] https://book.cakephp.org/4/es/appendices/4-4-migration-guide.html#requesthandlercomponent

temano y luego pasarla a `paginate()` `$query = $this->Articles->find( => true]); $this->paginate($query .`

Core

- La función `getTypeName()` ha sido desechada. En su lugar usa `get_debug_type()` de PHP.
- La dependencia de `league/container` se actualizó a `4.x`. Esto requerirá la adición de typehints a tus implementaciones de `ServiceProvider`.
- `deprecationWarning()` ahora tiene un parámetro `$version`.
- La opción de configuración `App.uploadedFilesAsObjects` se ha eliminado junto con

el soporte para arrays con forma carga de archivos PHP en todo el framework.

- `ClassLoader` ha sido eliminado. En su lugar, utiliza composer para generar archivos de carga automática.

Base de datos

- `DateTimeType` y `DateType` ahora siempre devuelven objetos inmutables. Además, la interfaz para los objetos `Date` refleja la interfaz `ChronosDate` que carece de todos los métodos relacionados con el tiem-

po que estaban presentes en CakePHP 4.x.

- `DateType::setLocaleFor` ya no acepta array.
- `Query` ahora solo acepta parámetros `\Closure` en lugar de `callable`. Los callables se pueden convertir a closures usando la nueva sintaxis de array de primera clase de PHP 8.1.
- `Query::execute()` ya no ejecuta el resultado de la ejeción de la consulta. Debe utilizar `Query::all()` en su lugar.
- `TableSchemaAwareInterf` fue eliminado.
- `Driver::quote()`

fue eliminado. En su lugar, utiliza declaraciones preparadas.

- `Query::orderBy()` fue añadido para reemplazar `Query::order()`.
- `Query::groupBy()` fue añadido para reemplazar `Query::group()`.
- `SqlDialectTrait` se ha eliminado y toda su funcionalidad se ha movido a la propia clase `Driver`.
- `CaseExpression` ha sido eliminado y debe ser reemplazado por `QueryExpression::case(` o `CaseStatementExpressio`
- `Connection::connect()` ha sido eliminado. Usar `$connection->getDriver` en su lu-

gar.

- `Connection::disconnect` ha sido eliminado. Usar `$connection->getDriver` en su lugar.
- `cake.database.queries` ha sido añadido como alternativa al scope `queriesLog`.

Datasource

- El método `getAccessible()` ha sido añadido a `EntityInterface`. Las implementaciones que no son ORM tienen que implementar este método ahora.
- El método `aliasField()` ha sido añadido a `RepositoryInterface`. Las implementaciones que no son ORM tienen que implementar este

método ahora.

Eventos

- Las cargas útiles de eventos deben ser un array. Otros objetos como `ArrayAccess` ya no se convierten en array y ahora lanzarán un `TypeError`.
- Se recomienda ajustar los handlers de eventos para que sean métodos void y usar `$event->setResult()` en lugar de devolver el resultado.

Error

- `ErrorHandler` y `ConsoleErrorHandler` han sido eliminados. Consulte la guía 4.4 migration[167] para saber

[167] https://book.cakephp.org/4/es/appendices/4-4-migration-guide.html#errorhandler-consoleerrorhandler

como actualizarlo.

- `ExceptionRenderer` ha sido eliminado y debe ser reemplazado por `WebExceptionRenderer`
- `ErrorLoggerInterface::` ha sido eliminado y debe ser reemplazado por `ErrorLoggerInterface::`
- `ErrorLoggerInterface::` ha sido eliminado y debe ser reemplazado por `ErrorLoggerInterface::`

Filesystem

- El paquete de Filesystem se ha eliminado, y la clase `Filesystem` se ha movido al paquete de Utility.

Http

- `ServerRequest` ya no es compatible con `files` como arrays. Este behavior se ha deshabilitado de forma predeterminada desde la version 4.1.0. Los datos `files` ahora siempre contendrán objetos `UploadedFileInterfaces`

I18n

- Se cambió el nombre de `FrozenDate` a *Date* y el de `FrozenTime` a *DateTime*.
- `Time` ahora extiende de `Cake\Chronos\ChronosTime` y. por lo tanto, es inmutable.
- `Date::parseDateTime()`

ha sido eliminado.

- `Date::parseTime()` ha sido eliminado.
- `Date::setToStringForma` y `Date::setJsonEncodeFor` ya no aceptan un array.
- `Date::i18nFormat()` y `Date::nice()` ya no aceptan un parámetro de zona horaria.
- Los archivos de traducción en la carpeta de vendor con prefijo como (`FooBar/Awesome`) ahora tendrán ese prefijo en el nombre del archivo de traducción, por ejemplo, `foo_bar_awesome.po` para evitar colisiones con otro fichero `awesome.po` corres-

pondiente con el plugin (`Awesome`).

Log

- La configuración del motor de registros ahora utiliza `null` en lugar de `false` para desactivar los scopes. Así que en lugar de `'scopes' => false` necesitas usar `'scopes' => null` en la configuración de tu log.

Mailer

- Se ha eliminado `Email`. Usar Mailer[168] en su lugar.
- `cake.mailer` se ha añadido como alternativa al scope `email`.

[168] https://book.cakephp.org/5/en/core-libraries/email.html

ORM

- `EntityTrait::has()` ahora devuelve `true` cuando existe un atributo y es estable en `null`. En versiones anteriores de CakePHP esto devolvía `false`. Consulte las notas de la version 4.5.0 para saber como adoptar este comportamiento en 4.x.
- `EntityTrait::extractOr` ahora devuelve solo los campos existentes, similar a `extractOriginalChanged`
- Ahora se requiere que los argumentos de un *Finder* sean arrays asociativos, como siempre se esperó que

fueran.

- `TranslateBehavior` ahora tiene como valor predeterminado la estrategia `ShadowTable`. Si está utilizando la estrategia `Eav` deberá actualizar la configuración de tu behavior para conservar el comportamiento anterior.
- La opción `allowMultipleNulls` para la regla `isUnique` ahora es true de forma predeterminada, coincidiendo con el comportamiento original de 3.x.
- `Table::query()` se ha eliminado en favor de funciones específicas de tipo de consulta.

- `Table::updateQuery()`, `Table::selectQuery()`, `Table::insertQuery()`, y `Table::deleteQuery()` se añadieron y ahora devuelven los nuevos objetos de consulta de tipo específico.
- Se añadieron `SelectQuery`, `InsertQuery`, `UpdateQuery` y `DeleteQuery` que representan solo un tipo de consulta y no permiten cambiar entre tipos de consulta, sin llamar a funciones no relacionadas con el tipo de consulta específico.
- `Table::_initializeSche` ha sido eliminado y debe ser reemplazado llamando a `$this->getSchema()` dentro del

método `initialize()`.

- `SaveOptionsBuilder` ha sido eliminado. En su lugar, utilice un array normal para las opciones.

Enrutamiento

- Los métodos estáticos `connect()`, `prefix()`, `scope()` y `plugin()` del `Router` han sido eliminados y deben ser reemplazados llamando a sus variantes de método no estáticos a través de la instancia `RouteBuilder`.
- `RedirectException` ha sido eliminado. Usar `\Cake\Http\Exception\RedirectException`

en su lugar.

TestSuite

- `TestSuite` fue eliminado. En su lugar, los usuarios deben usar variables de entorno para personalizar la configuración de las pruebas unitarias.
- `TestListenerTrait` fue eliminado. PHPUnit dejó de dar soporte a estos listeners. Ver documentación /appendices/phpunit10
- `IntegrationTestTrait::` ahora fusiona la configuración cuando se llama varias veces en lugar de reemplazar la configuración

actualmente presente.

Validaciones

- `Validation::isEmpty()` ya no es compatible con la subida de ficheros en forma arrays. El soporte para la subida de ficheros en forma de array también se ha eliminado de `ServerRequest` por lo que no debería ver esto como un problema fuera de las pruebas.
- Anteriormente, la mayoría de los mensajes de error de validacion de datos eran simplemente `El valor proporcionado no es válido`. Ahora, los mensajes de error de vali-

dación de datos están redactados con mayor precisión. Por ejemplo, `El valor proporcionado debe ser mayor o igual que \`5\`.`

Vistas

- Las opciones de `ViewBuilder` ahora son verdaderamente asociativas (string keys).
- `NumberHelper` y `TextHelper` ya no aceptan la configuración de `engine`.
- `ViewBuilder::setHelper` el parámetro `$merge` fue eliminado. Usar `ViewBuilder::addHelper` en su lugar.
- Dentro `View::initialize()`, preferen-

temente usar `addHelper()` en lugar de `loadHelper()`. De todas formas, todas las configuraciones de helpers se cargarán después.

- `View\Widget\FileWidget` ya no es compatible con la subida de ficheros en forma arrays. Esto está alineado con los cambios en `ServerRequest` y `Validation`.
- `FormHelper` ya no estable `autocomplete=off` en los campos de token CSRF. Esto fue una solución para un error de Safari que no es relevante.

Obsolescencias

A continuación se muestra una lista de métodos, propiedades y comportamientos en desuso. Estas características seguirán funcionando en la versión 5.x y se eliminarán en la versión 6.0.

Base de datos

- `Query::order()` ha quedado obsoleto. Utiliza `Query::orderBy()` en su lugar ahora que los métodos `Connection` ya no son proxy. Esto alinea el nombre de la función con la instrucción SQL.
- `Query::group()` ha quedado obsoleto. Utiliza `Query::groupBy()` en su lugar ahora que los métodos `Connection` ya no son proxy. Esto alinea el nombre de la función con la instrucción SQL.

ORM

- Llamar a `Table::find()` con opciones de array está obsoleto. Utiliza named arguments[169] en su lugar. Por ejemplo, en lugar de `find('all', ['conditions' => $array])` usar `find('all', conditions: $array)`. De manera similar, para las opciones de finders personalizados, en lugar de `find('list', ['valueField' => 'name'])` usar `find('list', valueField: 'name')` o varios argumentos como `find(type: 'list', valueField: 'name',`

[169] https://www.php.net/manual/en/functions.arguments.php#functions.named-arguments

`conditions: $array`).

Nuevas características

Comprobación de tipos mejorada

CakePHP 5 aprovecha la función de sistema de tipos expandidos disponible en PHP 8.1+. CakePHP también usa `assert()` para proporcionar mensajes de error mejorados y una solidez de tipo adicional. En el modo de producción, puede configurar PHP para que no genere código para `assert()` lo que mejora el rendimiento de la aplicación. Consulte *Mejora el Rendimiento de tu Aplicación* para saber cómo hacerlo.

Colecciones

- Se añadió `unique()` que filtra el valor duplicado especificado por la devolución de llamada proporcionada.
- `reject()` ahora soporta una devolución de llamada predeterminada que filtra los valores verdaderos, que es el inverso del comportamiento predeterminado de `filter()`

Core

- El método `services()` se añadió a `PluginInterface`.
- `PluginCollection::addP` se ha añadido a *simplify plugin loading*.

Base de datos

- **`ConnectionManager` ahora soporta roles de conexión de lectura y escritura. Los roles se pueden configurar** con claves de `read` y `write` en la configuración de conexión que anulan la configura-

ción compartida.

- Se añadió `Query::all()` que ejecuta devoluciones de llamada del decorador de resultados y devuelve un conjunto de resultados para consultas seleccionadas.
- Se añadió `Query::comment()` para agregar un comentario SQL a la consulta ejecutada. Esto facilita la depuración de consultas.
- `EnumType` fue añadido para permitir el mapeo entre enumeraciones respaldadas por PHP y una cadena o columna entera.

- `getMaxAliasLength()` y `getConnectionRetries()` se añadieron a `DriverInterface`.
- Los drivers compatibles ahora agregan automáticamente el incremento automático solo a las claves primarias enteras denominadas «id» en lugar de a todas las claves primarias enteras. Si se establece "autoIncrement" como false, siempre se deshabilita en todos los drivers compatibles.

Http

- Se ha añadido soporte para "factories interface" PSR-17[170]. Esto permite `cakephp/http` proporcionar una implementación de cliente a bibliotecas que permiten la resolución automática de interfaces como php-http.
- Se añadieron `CookieCollection::__ge` y `CookieCollection::__is` para añadir formas ergonómicas de acceder a las cookies sin excepciones.

[170] https://www.php-fig.org/psr/psr-17/

ORM

Campos de entidad obligatorios

Las entidades tienen una nueva funcionalidad de opt-in que permite hacer que las entidades manejen propiedades de manera más estricta. El nuevo comportamiento se denomina "required fields". Cuando es habilitado, el acceso a las propiedades que no están definidas en la entidad generará excepciones. Esto afecta a los siguientes usos:

```
$entity->get();
$entity->has();
$entity->getOriginal();
isset($entity->attribute);
$entity->attribute;
```

Los campos se consideran definidos si pasan `array_key_exists`. Esto incluye valores nulos. Debido a que esta puede ser una característica tediosa de habilitar, se aplazó a 5.0. Nos gustaría recibir cualquier comentario que tenga sobre esta función, ya que estamos considerando hacer que este sea el comportamiento predeterminado en el futuro.

Typed Finder Parameters

Los finders de las tablas ahora pueden tener argumentos escritos según sea necesario en lugar de un array de opciones. Por ejemplo, un finder para obtener publicaciones por categoría o usuario:

```
public function findByCategoryOrUser(Sele
$query, array $options)
{
    if (isset($options['categoryId'])) {

$query->where(['category_id' => $options['categoryId']]);
    }
```

(continué en la próxima página)

(proviene de la página anterior)

```
    if␣
↪(isset(
↪$options[
↪'userId
↪'])) {

↪$query->
↪where([
↪'user_id'␣
↪=>
↪$options[
↪'userId
↪']]);
    }

    return
↪$query;
}
```

Ahora se pueden escribir como:

```
public␣
↪function␣
↪findByCategoryOrUser(Sele
↪$query, ?
↪int
↪$categoryId␣
↪= null, ?
↪int
↪$userId =␣
↪null)
{
    if (
↪$categoryId)␣
↪{

↪$query->
↪where([
↪'category_
↪id' =>
↪$categoryId]);
↪
    }
    if (
↪$userId) {

↪$query->
↪where([
↪'user_id'␣
↪=>
↪$userId]);
    }
```

(continué en la próxima página)

(proviene de la página anterior)

```
    return
↪$query;
}
```

El finder puede ser llamado como `find('byCategoryOrUser', userId: $somevar)`. Incluso puedes incluir los argumentos con nombre especial para establecer cláusulas de consulta. `find('byCategoryOrUser', userId: $somevar, conditions: ['enabled' => true])`.

Un cambio similar se ha aplicado al método `RepositoryInterface::get()`:

```
public␣
↪function␣
↪view(int
↪$id)
{
    $author␣
↪= $this->
↪Authors->
↪get($id, [

↪'contain'␣
↪=> ['Books
↪'],

↪'finder'␣
↪=> 'latest
↪',
    ]);
}
```

Ahora se pueden escribir como:

```
public␣
↪function␣
↪view(int
↪$id)
{
    $author␣
↪= $this->
↪Authors->
↪get($id,␣
↪contain: [
↪'Books'],␣
↪finder:
↪'latest');
}
```

TestSuite

- Se ha añadido `IntegrationTestTrait::`
  para establecer encabezados JSON para la siguiente solicitud.

Instalador de plugins

- El instalador de plugins se ha actualizado para manejar automáticamente la carga automática de clases para los plugins de tu aplicación. Por lo tanto, puedes eliminar el espacio de nombres para las asignaciones de rutas de tus plugins del `composer.json` y simplemente ejecutar `composer dumpautoload`.

Retrocompatiblidad

Si quieres utilizar funcionalidades de 3.x o 4.x o ir migrando poco a poco el Plugin Shim[171] puede ayudarte a mitigar algunos de los cambios que rompen la compatibilidad.

Antecompatiblidad

La Antecompatibilidad puede preparar tu aplicación 4.x para la próxima versión 5.x.

Si quieres utilizar una funcionalidad de 5.x en 4.x, revisa el Plugin Shim[172]. Este plugin te ayuda a mitigar los problemas de compatibilidad y a llevar funcionalidades de 5.x a 4.x.

General Information

CakePHP Development Process

Los proyectos de CakePHP en general siguen semver[173]. Ésto significa que:

- Las versiones se numeran en el formato **A.B.C**
- Las versiones **A** son *lanzamientos principales*. Contienen cambios importantes y requerirán una cantidad significativa de trabajo para actualizar desde una version **A** inferior.

[171] https://github.com/dereuromark/cakephp-shim
[172] https://github.com/dereuromark/cakephp-shim
[173] https://semver.org/

- Las versiones **A.B** son *lanzamientos de mejoras*. Cada versión será compatible con las anteriores, pero puede marcar algunas características como **obsoletas**. Si es absolutamente necesario realizar un cambio que rompa la compatibilidad, se indicará en la guía de migración para ese lanzamiento.
- Las versiones **A.B.C** son *lanzamientos de parches*. Deben ser compatibles con el lanzamiento de parche anterior. La excepción a esta regla es si se

descubre un problema de seguridad y la única solución es romper una API existente.

Consulta el :doc:/contributing/backwards-compatibility para ver lo que consideramos como compatible con versiones previas y cambios que rompen la compatibilidad.

Lanzamientos Principales

Los lanzamientos principales introducen nuevas características y pueden eliminar funcionalidades que se hayan marcado como obsoletas en un lanzamiento anterior. Estos lanzamientos se encuentran en las ramas `next` que coinciden con su número de versión, como `5.next`. Una vez que se lanzan, se promocionan a la rama `master` y luego la rama `5.next` se utiliza para futuros lanzamientos de características.

Lanzamientos de Mejoras

Los lanzamientos de mejoras son donde se envían nuevas funcionalidades o extensiones a las funcionalidades existentes. Cada serie de lanzamientos que recibe actualizaciones tendrá una rama `next`, por ejemplo, `4.next`. Si deseas contribuir con una nueva característica, por favor dirígete a estas ramas.

Lanzamientos de Parches

Los lanzamientos de parches corrigen errores en el código/documentación existente y siempre deben ser compatibles con los lanzamientos de parches anteriores de la misma serie. Estos lanzamientos se crean a partir de las ramas estables. Las ramas estables a menudo se nombran según la serie de lanzamientos, como `3.x`.

Frecuencia de Lanzamiento

- Los *Lanzamientos Principales* se entregan aproximadamente cada dos o tres años. Este período de tiempo nos obliga a ser deliberados y considerados con

los cambios que rompen la compatibilidad, y brinda tiempo a la comunidad para ponerse al día sin sentir que se están quedando atrás.

- Los *Lanzamientos de Mejoras* se entregan cada cinco a ocho meses.
- Los *Lanzamientos de Parches* se entregan inicialmente cada dos semanas. A medida que un lanzamiento de características madura, esta frecuencia se relaja a una entrega mensual.

Política de Obsolescencia

Antes de que una característica pueda ser eliminada en un lanzamiento principal, necesita ser marcada como obsoleta. Cuando una funcionalidad se marca como obsoleta en el lanzamiento **A.x**, seguirá funcionando durante el resto de todos los lanzamientos **A.x**. Las obsolescencias generalmente se indican mediante advertencias en PHP. Puedes habilitar las advertencias de obsolescencia agregando `E_USER_DEPRECATED` al valor de `Error.level` de tu aplicación.

El comportamiento marcado como obsoleto no se elimina hasta el próximo lanzamiento principal. Por ejemplo, un comportamiento marcado como obsoleto en `4.1` se eliminará en `5.0`.

Glosario

arreglo de enrutamiento
: Un arreglo de atributos que son pasados a `Router::url()`. Típicamente se ve algo así:

```
[
↪'controller
↪' =>
↪'Posts
↪',
↪'action
↪' =>
↪'view
↪', 5]
```

Atributos HTML
: Un array con claves => valores que son colocados en los atributos HTML. Por ejemplo:

```
// Dado
['class
↪' =>
↪'mi-
↪clase
```

(continué en la próxima página)

(proviene de la página anterior)

```
↪',
↪'target
↪' =>
↪'_
↪blank
↪']

//␣
↪Generará
class=
↪"mi-
↪clase
↪"␣
↪target=
↪"_
↪blank
↪"
```

Si una opción puede usar su nombre como valor, entonces puede ser usado `true`:

```
// Dado
[
↪'checked
↪' =>␣
↪true]

//␣
↪Generará
checked=
↪"checked
↪"
```

Sintaxis de plugin

La sintáxis de plugin se refiere a el punto que separa los nombres de clases indicando

que la clase es parte de un plugin:

```
// El␣
↪plugin␣
↪es
↪"DebugKit
↪", y␣
↪el␣
↪nombre␣
↪de␣
↪la␣
↪clase␣
↪es
↪"Toolbar
↪".

↪'DebugKit.
↪Toolbar
↪'

// El␣
↪plugin␣
↪es
↪"AcmeCorp/
↪Tools
↪", y␣
↪el␣
↪nombre␣
↪de␣
↪clase␣
↪es
↪"Toolbar
↪".

↪'AcmeCorp/
↪Tools.
↪Toolbar
↪'
```

Notación de punto

La notación de punto define un array de rutas, separando los niveles anidados

con . Por ejemplo:

```
Cache.
↪default.
↪engine
```

Apuntará al siguiente valor:

```
[
↪'Cache
↪' =>␣
↪[
    ␣
↪
↪'default
↪' =>␣
↪[
    ␣
↪
↪'engine
↪' =>
↪'File
↪'
    ␣
↪ ]
  ]
]
```

CSRF
: *Cross Site Request Forgery*. Previene los ataques de replay o playback, peticiones duplicadas y peticiones falsificadas desde otros dominios.

CDN
: *Content Delivery Network*.

Le puedes pagar a un proveedor para que ayude a distribuir el contenido a centros de datos alrededor del mundo. Esto ayuda a poner elementos estáticos más cerca de tus usuarios geográficamente.

routes.php
: Un archivo en el directorio `config` que contiene las configuraciones de enrutamiento. Este archivo se incluye antes de que cada petición sea procesada. Se deben conectar todas las rutas que necesita tu aplicación para que cada petición sea enrutada correcta-

mente al controlador + acción.

DRY
: *Don't repeat yourself*. Es un principio de desarrollo de software orientado a reducir la repetición de la información de todo tipo. En CakePHP, DRY se utiliza para que se pueda escribir las cosas una vez y reutilizarlos a través de su aplicación.

PaaS
: *Platform as a Service*. La plataforma como servicio proporcionará recursos de hosting, bases de datos y almacenamiento en caché basado en la nube.

Algunos proveedores populares incluyen Heroku, EngineYard y PagodaBox.

DSN
: *Data Source Name*. Una cadena de conexión formateada para que sea como una URI. CakePHP soporta conexiones DSN para Caché, Base de datos, Registro y de E-mail.

PHP Namespace Index

C

Índice

Símbolos

A

B

C

D

E

F

G

S

T

U

V

W

X