CODE HAPPY

DESARROLLO DE APLICACIONES CON EL FRAMEWORK DE PHP LARAVEL PARA PRINCIPIANTES

DAYLE REES & ANTONIO LAGUNA

Laravel: Code Happy (ES)

Desarrollo de aplicaciones con el Framework de PHP Laravel para principiantes.

Dayle Rees and Antonio Laguna

This book is for sale at http://leanpub.com/codehappy-es

This version was published on 2013-02-06

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Dayle Rees and Antonio Laguna by spreading the word about this book on Twitter!

The suggested hashtag for this book is #codehappy.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#codehappy

Índice general

Reconocimiento

Antes que nada, me gustaría agradecer a mi novia Emma, no solo por apoyarme en todas mis ideas frikis, ¡sino por hacer la increíble foto del panda rojo para la portada del libro! ¡Te amo Emma!

A Taylor Otwell, también te quiero tío, pero de forma totalmente varonil. Gracias por hacer un framework que es realmente un placer usar, hace que nuestro código se lea como poesía y por dedicar tanto tiempo y pasión en su desarrollo.

Eric Barnes, Phill Sparks, Shawn McCool, Jason Lewis, Ian Landsman, gracias por todo el apoyo con el framework, y por permitirme ser parte de un proyecto con tanto potencial.

A todos los lectores de mi blog que mostraron interés en mis tutoriales, ¡gracias! Sin vosotros nunca habría tenido la confianza de escribir un libro.

Reconocimientos del traductor

Me gustaría agradecer a mi mujer, Laura, todo el apoyo que me ha ofrecido. Especialmente teniendo en cuenta que me he tomado parte de las vacaciones para ponerme a traducir este libro.

Al señor Dayle Rees, por dejarme ser parte de este proyecto y brindarme esta oportunidad. Ha sido un placer trabajar contigo y todo un desafío el traducir este estilo tan desenfadado que te caracteriza.

Por supuesto a Taylor Otwell, este framework es una gozada. Es una gozada de programar y de leer y estoy seguro de que aun le queda mucho camino por recorrer. No quiero ni imaginar lo que cuece para la versión 4.0.

A todos aquellos que leéis mis artículos en mi blog, gracias por tomaros el tiempo de escribirme comentarios de mandarme correos y de apreciar el trabajo que se hace. Espero que lo disfrutéis.

Erratas

Aunque haya tenido todo el cuidado del mundo para asegurarme de que el libro no tiene errores, algunas veces unos ojos cansados juegan malas pasadas, y los errores escapan al radar. Si te encuentras con algún error en el libro, ya sea de escritura o de código, te agradeceré mucho si pudieras avisarme de su presencia enviándome un correo a me@daylerees.com[1] incluyendo el capítulo y la ubicación del error.

Los errores serán solucionados conforme vayan siendo descubiertos, y las correcciones serán lanzadas con la próxima publicación del libro.

[1] mailto:me@daylerees.com

Feedback

De igual forma, puedes enviarme cualquier comentario que puedas tener sobre el contenido del libro enviándome un correo a me@daylerees.com[2]. Me comprometo a responder todos los correos que reciba sobre el libro.

[2] mailto:me@daylerees.com

Introducción

¡Hola! Soy Dayle Rees, ¡y voy a ser vuestro guía en este mágico viaje hacia el mundo de Laravel! Vale, pensándolo mejor, eso me quedó realmente cursi. Este que ves es mi primer libro, por lo que voy no tengo mucha práctica con toda la jerga literaria de alto nivel, por lo que si te gusta un texto directo, y ser "hablado" como si fueras un ser humano, ¡estamos en el mismo barco entonces!

Os estaréis preguntando "¿Por qué debería confiar en este tío para que me enseñe Laravel? ¡Ni siquiera es un autor experimentado!"

Lo que me falta de experiencia, lo tengo de entusiasmo. Soy un gran fan del desarrollo web, y las herramientas y trucos que nos ahorran tiempo, o hacen nuestro trabajo mucho más fácil. Laravel cumple ambos requisitos, y es una de las piezas de software más útiles que jamás haya descubierto. De hecho, mi entusiasmo y pasión por el framework, que solo tiene rivalidad con el autor del framework me han llevado a ser incluído como miembro del equipo de Laravel, o "Concilio" como nos gusta llamarlo. Suena mejor, ¿verdad?

Estar en el Concilio me otorga ciertos privilegios como ser notificado de nuevas ideas y adiciones planeadas al framework, y contribuyendo al framework estoy constantemente en contacto con el código que está en constante desarrollo. Esto me deja en una gran posición para mantener el libro actualizado, lo cual pretendo hacer con cada futuro lanzamiento del framework.

Aunque no me gusta salirme del tema, me parece obligatorio tener un pequeño párrafo sobre el autor de libros de este tipo, por lo que vamos a hacerlo corto y agradable. Vivo en la costa de Gales (es un país del lado de Inglaterra para los que seáis del otro lado del charco) y trabajo para una gran organización del sector público en Aberystwyth. En mi tiempo libre estoy realmente involucrado con Laravel. Oh, y como he dicho antes, no me considero un genio literario, seamos sinceros mi escritura va a ser penosa, no va a ser tan buena como la de otros libros de programación. Os 'hablaré' como una persona real, a la que podrías responder también, ya lo veremos. Con un poco de suerte, mi pasión por Laravel compensará mi Inglés común (para ser justos, soy Galés). Ya no necesitas saber más sobre mi, centrémonos ahora en Laravel.

Laravel es un soplo de aire fresco en el mundo de PHP. El lenguaje de programación PHP es habitualmente conocido por sus horribles nombres de funciones, y aunque los desarrolladores PHP hemos aprendido a quererlo, la sintaxis puede ser un poco fea comparada con algunos lenguajes Japoneses modernos. ¡Hola Ruby!

Afortunadamente Laravel cambia todo esto, de hecho… creo que la sintaxis de Laravel (que está construido sobre PHP) es tan expresiva y cuidada, que la encuentro mucho más sencilla de leer que Ruby. No es muy compacta, y aunque no se lea como una frase en inglés, se lee como una poesía que solo puede ser leida por los ojos de un programador.

> Pero Dayle…

Digamos que de repente te asalta la idea de que podrías haber gastado tus $4.99 en una bebida divertida.

> Laravel es un framework, ¡no un lenguaje!

Es cierto, me has pillado. Laravel puede que no sea un lenguaje, pero no tiene porqué serlo. Nos gusta PHP, vamos, seamos sinceros, aguantamos su fealdad, disfrutamos escribiendo todas esas llaves y puntos y comas. Laravel simplemente añade los atajos, o cambios sobre el código para hacer la experiencia mucho más agradable.

Creo que el placer de trabajar con el framework, viene dado por sus expresivos métodos que son consistentes en todo el framework. `Str::upper()` Intenta decirme que no sabes lo que hace, intenta decirme que no tiene sentido.

> No puedo.

Sip, eso pensaba. Bueno, podría estar todo el día parloteando sobre todo lo que hace a Laravel maravilloso, veamos, está Eloquent, el mejor ORM que haya visto nunca. Fue lo primero que me trajo al framework.

La opción de usar clausuras (closures) para rutas, pero solo si quieres. ¡Creo que se ven geniales!

No tener que cargar librerías antes de usarlas, sí… me has oído bien. Lo verás en acción más tarde.

No, hay demasiadas características increibles para explicar aquí, creo que sería mejor si nos sumergimos y comenzamos a aprender, después de todo… si gastaste $4.99 en esto en vez de alguna bebida divertida, ya debes de tener interes en el framework ¿verdad? ¡Adelante!

1 Comenzando

Laravel[1] es un framework de aplicaciones para PHP 5.3 escrito por Taylor Otwell[2]. Fue escrito con las características de PHP 5.3 en mente. La combinación de esas características y su expresiva sintaxis le han granjeado al framework bastante popularidad.

En este libro exploraremos Laravel desde los cimientos comenzando con su instalación, y estoy seguro de que coincidiréis conmigo en que es pan comido.

Para poder usar cualquier framework PHP (no solo Laravel) necesitarás tener un servidor web con PHP 5.3 activo, recomendaría instalar un servidor web en tu máquina local de desarrollo, para permitirte probar tu trabajo sin tener que subir los archivos cada vez que hagas un cambio.

Este capítulo asume que:

- Tienes un servidor web, basado en Apache, funcionando.
- Estás familiarizado con el sistema de archivos del servidor, y sabes cómo mover / copiar ficheros.
- Tienes acceso para modificar los archivos de configuración de Apache.

Si estás usando un servidor web diferente, podrás encontrar muchos artículos en la red sobre cómo llevar a cabo las tareas que encontrarás abajo para tu servidor.

Primero, necesitaremos una copia del código fuente del framework, simplemente dirígete a Laravel.com[3] y dale al gran botón naranja de descarga. Para mayor seguridad, recomendaría extraer los contenidos del paquete a algún otro sitio que no fuera la raíz de tu web. Guarda una nota mental de dónde dejaste el código fuente (¡o busca un postit!).

Ahora tenemos dos opciones para permitir al framework ejecutarse como debe. Yo recomendaría intentar el primer método ya que es la forma "real" de instalar el framework y nos permite realizar una configuración más detallada. No obstante, encuentro el segundo método mucho más sencillo al trabajar con muchos proyectos en un servidor de desarrollo.

1.1 Método 1 Crea un nuevo host virtual

Tendremos que crear un nuevo archivo de configuración de Apache. En la mayoría de las instalaciones estándar, creando un archivo `miproyecto.conf` en la carpeta `conf.d` de Apache, lo incluirá por defecto. Por favor, revisa la documentación de tu configuración actual para más información.

Dentro de tu nuevo archivo de configuración, pega/escribe la siguiente declaración del host virtual:

[1] http://laravel.com
[2] https://twitter.com/#!/taylorotwell
[3] http://laravel.com

```
<VirtualHost 127.0.0.2>
        DocumentRoot "/ruta/al/proyecto/de/laravel/public"
        ServerName miproyecto
        <Directory "/ruta/al/proyecto/de/laravel/public">
                Options Indexes FollowSymLinks MultiViews
                AllowOverride all
        </Directory>
</VirtualHost>
```

Tenemos que actualizar la dirección IP a una que no esté actualmente en uso. (Mejor no usar `127.0.0.1`, esta es nuestra dirección loopback, y puede que ya tengas algo que la esté usando.) Cambia ambas rutas para que apunten a la carpeta `public` del código de Laravel. Ahora, reinicia tu servidor web.

Ahora tenemos que crear una nueva entrada en la DNS local para apuntar el nombre del proyecto a tu host virtual. Primero abre el fichero hosts que normalmente se encuentra en `c:\windows\system32\drivers\etc\hosts` en Windows o `/etc/hosts` en sistemas basados en unix.

Añade la siguiente línea usando la dirección IP que usaste en la declaración de tu host virtual, y un nombre para tu proyecto:

```
127.0.0.2               miproyecto
```

Ahora deberías de poder navegar a: `http://miproyecto`

con tu navegador web, y ver la página de bienvenida de Laravel.

1.2 Método 2 Enlace simbólico a la carpeta pública

Si estás familiarizado con el uso de enlaces simbólicos en sistemas basados en unix, este método será bastante sencillo.

Dentro del código que extrajiste (recuerdas dónde lo dejaste, ¿verdad?) encontrarás una sub carpeta llamada `public`. Esta carpeta contiene el archivo de inicialización de Laravel y todos los elementos públicos. Enlazaremos esta carpeta de forma simbólica con la raíz pública de tu web (posiblemente /var/www/html/).

Para crear el enlace simbólico solo tienes que ejecutar el siguiente comando en la terminal de tu elección, reemplazando las rutas donde sea necesario.

```
ln -s /ruta/a/la/carpeta/public/de/laravel /ruta/a/la/carpeta/root/de/la/we\
b/subdirectorio
```

Por ejemplo :

```
1 ln -s /home/dayle/laravel/miaplicacion/public /var/www/html/miaplicacion
```

Nota: También puedes hacer un enlace simbólico a la carpeta pública de la raíz de tu web directamente, pero prefiero usar un subdirectorio para poder trabajar en varios proyectos.

Ahora deberías de poder navegar a: `http://localhost/myapp`

con tu navegador web, y ver la página de bienvenida de Laravel.

1.3 Centrándonos de nuevo

En este punto deberías poder ver la página de bienvenida de Laravel, si es así...

¡Enhorabuena! Ahora tienes un nuevo proyecto de Laravel, ¡ya estás preparado para empezar a programar!

En el próximo capítulo cubriremos la estructura de proyectos de Laravel, y explicaremos cada uno de los archivos y carpetas importantes.

Si encuentras alguno de los temas tratados en este libro algo confusos, puedes usar los siguientes enlaces para encontrar la ayuda y el soporte que necesitas, o simplemente escribir un nuevo comentario en DayleRees.com[4] (en Inglés).

- Web de Laravel[5]
- Documentación de Laravel[6]
- API de Laravel[7]
- Foros de Laravel[8]

¡No dejes de unirte a nuestra comunidad en continua expansión usando un cliente de IRC para conectarte a `irc.freenode.net:6667` y unirte al canal de `#laravel`!

[4] http://daylerees.com
[5] http://laravel.com
[6] http://laravel.com/docs
[7] http://laravel.com/api
[8] http://forums.laravel.com

2 Estructura de proyectos

El paquete fuente de Laravel contiene varios directorios. Vamos a echar un vistazo a la estructura del proyecto para conseguir una mayor comprensión de cómo funcionan las cosas. Puede que use algunos términos para describir algunas características de Laravel que puedan sonar algo confusas si estás comenzando ahora mismo, si es así, no te preocupes porque cubriremos cada característica en posteriores capítulos.

2.1 Estructura del directorio raíz

Echemos un vistazo a la estructura de ficheros y directorios de primer nivel:

```
/application
/bundles
/laravel
/public
/storage
/vendor
/artisan [archivo]
/paths.php [archivo]
```

Ahora centrémonos en uno cada vez:

/application

Aquí es donde estará la mayoría del código de tu aplicación. Contiene tus rutas, modelos de datos y vistas. ¡Pasarás aquí la mayor parte del tiempo!

/bundles

Los Bundles son aplicaciones de Laravel. Se pueden usar para separar aspectos de tu aplicación, o pueden ser lanzados / descargados para compartir código común[1]. Instalando nuevos bundles con artisan, puedes extender la funcionalidad de Laravel para ajustarla a tus necesidades.

Curiosamente, el directorio `/application` es también un bundle conocido como `DEFAULT_BUNDLE`, lo que significa que cualquier cosa que uses en `/application`, ¡puedes usarla también en tus bundles!

/laravel

Aquí es donde se encuentran los archivos del núcleo del framework. Éstos son archivos que necesita para ejecutar una petición. Raramente tendrás que interactuar con este directorio, pero a veces puede resultar útil revisar el código fuente para ver cómo funciona una Clase o Método. De manera alternativa, también puedes revisar el API de Laravel[2].

/public

[1] http://bundles.laravel.com/

[2] http://laravel.com/api

Este es el directorio al que debes aputnar tu servidor web. Contiene el archivo de inicialización `index.php` que hace funcionar el framework de Laravel, y el proceso de enrutado. El directorio público también puede ser usado para almacenar recursos públicamente accesibles como CSS, JavaScript, archivos e imágenes. La subcarpeta `laravel` contiene los archivos necesarios para mostrar la documentación sin conexión correctamente.

/storage

El directorio de almacenamiento se usa como almacén de archivos para servicios que usen el sistema de ficheros como driver, por ejemplo la clase Sessions o Cache. Este directorio debe poder ser escrito por el servidor-web. No tendrás que interactuar con este directorio para construir una aplicación Laravel.

/vendor

Este directorio contiene código usado por Laravel, pero que no fue escrito por el autor del framework o los contribuyentes. La carpeta contiene software de código abierto, o partes de software que contribuyen a las características de Laravel.

/artisan [archivo]

Artisan es la interfaz de línea de comandos de Laravel (CLI). Te permite realizar numerosas tareas[3] en la línea de comandos. ¡Incluso puedes crear tus propias tareas! Para ejecutar Artisan simplemente escribe:

```
1 php artisan
```

/paths.php [archivo]

Este archivo es usado por el Framework para determinar rutas a los directorios importantes arriba mencionado, y para facilitar un atajo para obtenerlos (usando `path()`). No deberías necesitar editar este archivo.

2.2 Estructura del directorio Application

Como mencionamos anteriormente, `/application` es donde ocurre toda la diversión, por lo que echémosle un vistazo a la estructura del directorio `/application`.

```
1 /config
2 /controllers
3 /language
4 /libraries
5 /migrations
6 /models
7 /tasks
8 /tests
9 /views
```

[3] http://laravel.com/docs/artisan/commands

```
/bundles.php [archivo]
/routes.php [archivo]
/start.php [archivo]
```

Y ahora vamos a ver cada uno con detenimiento.

/config

La carpeta config contiene un varios archivos de configuración para cambiar varios aspectos del framework. No hace falta cambiar las configuraciones en la instalación para que funcione el framework 'recién salido del horno'. La mayoría de los archivos de configuración devuelven una matriz de opciones de tipo clave-valor, algunas veces clave-closure que permiten una gran libertad de modificar el funcionamiento interno de algunas de las clases del núcleo de Laravel.

/controllers

Laravel facilita dos métodos para el enrutado, el uso de `controllers` (controladores) y el uso de `routes` (rutas). Esta carpet acontiene las clases Controlador que son usadas para facilitar una lógica básica, interactuar con los modelos de datos, y cargar archivos de vistas para tu aplicación. Los controladores se añadieron al framework posteriormente para ofrecer un entorno más familiar a usuarios que estuvieran migrando desde otros frameworks. Aunque fueron añadidos a posteriori, gracias al potente sistema de enrutado de Laravel, te permiten realizar cualquier acción que pueda ser realizada usando `rutas` a closures.

/language

En este directorio se encuentran archivos PHP con matrices de cadenas para facilitar una traducción sencilla de aplicaciones creadas con Laravel. Por defecto, el directorio contiene un archivo de cadenas para la paginación y validación del formulario en Inglés.

/libraries

Este directorio puede ser usado para 'soltar' librerías PHP de una sola clase para añadir funcionalidad extra a tu aplicación. Para Librerías más grandes es recomendable crear un Bundle. La carpeta Librerías se añade al autocargador al inicio, en el archivo start.php.

/migrations

Esta carpeta contiene clases PHP que permiten a Laravel actualizar el esquema de tu base de datos actual, o rellenarla con valores mientras mantiene todas las versiones de la aplicación sincronizadas. Los archivos de migración no se deben crear a mano, ya que el nombre del archivo contiene una marca de tiempo. En vez de eso, usa el comando de la interfaz CLI de Artisan `php artisan migrate:make <nombre de migracion>` para crear una nueva migración.

/models

Los modelos son las clases que representan los datos de tu proyecto. Normalmente implicarán integración con algún tipo de base de datos u otra fuente de datos. Laravel facilita tres métodos para interactuar con plataformas comunes de bases de datos, incluyendo un constructor de consultas llamado 'Fluent'[4], que te permite crear consultas SQL encadenando métodos PHP, usar el ORM (mapeo de objetos relacional) Eloquent para representar tus tablas como objetos PHP,

[4] http://laravel.com/docs/database/fluent

o las antiguas consultas SQL planas a las que estás acostumbrado. Tanto Fluent como Eloquent usan una sintaxis similar, haciendo que su adopción sea una sencilla transición.

El directorio models se carga de forma automática desde start.php.

/tasks

Creando clases en el directorio tasks, podrás añadir tus propias tareas personalizadas a la interfaz de línea de comandos de Artisan. Las tareas se representan como clases y métodos.

/tests

La carpeta tests facilita una ubicación para que mantengas las pruebas unitarias de tu aplicación. Si estás usando PHPUnit, puedes ejecutar todas las pruebas a la vez usando la interfaz de línea de comandos de Artisan.

/views

El directorio views contiene todos los archivos de plantillas HTML que serán usados por los controladores o las rutas, no obstante usa la extensión .php para los archivos de esta carpeta por favor. De manera alternativa, puedes usar la extensión .blade.php para habilitar el análisis con la librería de plantillas Blade, que será explicada en un capítulo posterior.

/bundles.php [archivo]

Para habilitar un bundle, simplemente añádelo a la matriz en bundles.php. También puedes usar una pareja clave-valor nombre-matriz para definir opciones extra para el bundle.

/routes.php [archivo]

El archivo routes contiene los métodos que permiten que las rutas sean mapeadas a sus respectivas acciones con Laravel. Este tema será tratado en profundidad en posteriores capítulos. Este archivo también contiene declaraciones de varios eventos incluyendo páginas de errores, y puede ser usado para definir creadores de vistas o filtros de rutas.

/start.php [archivo]

El archivo start.php contiene las rutinas de inicialización para el bundle `/application` bundle, como la auto-carga de directorios, carga de configuraciones, ¡y otras cosas maravillosas! No dejes de echarle un vistazo a este archivo.

En el próximo capítulo, cubriremos el sistema de rutas usado por los controladores creando un pequeño sitio dinámico con varias páginas.

3 Usando controladores

En este capítulo crearemos una sencilla web multi-página para mostrar el trabajo del sistema de enrutado de Laravel, sin profundizar en nada demasiado complicado.

Como he mencionado en el anterior capítulo, hay dos opciones disponibles para enrutar las peticiones web en tu código, Controladores y Rutas. En este capítulo usaremos Controladores ya que cualquiera que se haya unido a nosotros desde otro framework, se sentirá más familiarizado con ellos.

3.1 Enrutando controladores

Comencemos echándole un vistazo a un controlador:

```
1  <?php
2
3  // application/controllers/cuenta.php
4  class Cuenta_Controller extends Base_Controller
5  {
6
7          public function action_index()
8          {
9                  echo "Esta es la página del perfil.";
10         }
11
12         public function action_login()
13         {
14                 echo "Este es el formulario de login.";
15         }
16
17         public function action_logout()
18         {
19                 echo "Esta es la acción de cierre de sesión.";
20         }
21
22 }
```

Un controlador es una clase PHP que representa una sección de tu sitio o aplicación web. Sus Métodos o 'Acciones' representan una página individual, o el punto de fin de una petición HTTP.

En el ejemplo anterior, nuestro Controlador de Accoutn representa la sección users de nuestro sitio web, una página de perfil, una página de inicio de sesión, y una página de cierre de sesión. Ten encuenta que el nombre del Controlador lleva al final `_Controller` y que los nombres de las acciones tienen el prefijo `action_`. Los controladores deben extender el `Controlador Base_-Controller`, u otra clase Controlador.

Crearemos nuestro controlador en la carpeta `application/controllers` como archivo en minúsculas coincidiendo con el nombre del controlador. El Controlador de arriba debería ser guardado en:

```
/application/controllers/cuenta.php
```

Antes de que podamos usar nuestro Controlador, tenemos que registrarlo en `/application/routes.php`. Vamos a añadir la siguiente línea:

```
<?php

// application/routes.php
Route::controller('cuenta');
```

Si nuestro controlador está en una subcarpeta del directorio `controllers`, usa puntos (.) para seperar el/los directorio(s) de la siguiente forma:

```
<?php

// application/routes.php
Route::controller('en.una.sub.carpeta.cuenta');
```

Si nuestro controlador se encuentra en un bundle, añade como prefijo el nombre del bundle y el símbolo de dos puntos:

```
<?php

// application/routes.php
Route::controller('mibundle::cuenta');
```

Ahora si visitamos:

```
http://myproject/cuenta/login
```

veremos `Este es el formulario de login.`. Esto es porque ahora que nuestro controlador ha sido mapeado en la clase Route, el primer segmento (entre las barras) de la URL especifica el controlador, y el segundo segmento (sí, de nuevo entre las barras) especifica la acción.

En palabras simples `/cuenta/login` está relacionado con `Account_Controller->action_login()` y lo que vemos es el resultado de nuestro método.

Ahora vamos a probar a visitar `/cuenta`:

```
Esta es la pÃ¡gina del perfil.
```

¿Por qué ocurre esto? La acción `index` es una acción especial que se llama cuando no hay ninguna acción especificada en la URL, por tanto la página de arriba también podría ser "invocada" con la siguiente URL:

```
/cuenta/index
```

3.2 Pasando parámetros

El enrutado simple es interesante, pero no nos ofrece nada que un simple sitio con PHP no pueda.

Intentemos algo un poco más dinámico. Añadiendo parámetros a las acciones de nuestros controladores, podemos pasar datos extra como segmentos a la URL. Vamos a añadir una acción de bienvenida a nuestro controlador:

```php
<?php

// application/controllers/cuenta
public function action_bienvenida($nombre, $lugar)
{
        echo "¡Bienvenido a {$lugar}, {$nombre}!";
}
```

Aquí nuestros parámetros de la acción son los parámetros del método, por lo que el código de arriba debería serte familiar. Vamos a intentar visitar la ruta `/cuenta/bienvenida/Dayle/Gales`...

```
Â¡Bienvenido a Gales, Dayle!
```

Los parámetros pueden ser usados para pasar identificadores de recursos para permitir acciones CRUD (crear, obtener, actualizar y borrar), ¡o cualquier cosa que se te ocurra! Como puedes ver, ofrecen gran flexibilidad a nuestras accioens.

Nota: Puedes asignar valores a las acciones de tus parámetros para hacerlas opcionales en la URL.

3.3 Usando vistas

Haciendo `echo` desde el código de nuestro Controlador nos da un resultado, pero no es una solución elegante. Es probable que las soluciones elegantes sean las que te hayan llevado a aprender Laravel. La naturaleza de MVC sugiere que separemos nuestras capas visuales, de la lógica de la aplicación. Es aquí donde entra en juego la porción 'Vista' del patrón.

Con las vistas de Laravel no podría ser más sencillo, simplemente añade plantillas HTML a tu directorio `/application/views/` con un nombre de archivo en minúsculas, y una extensión `.php`. Por ejemplo:

```
/application/views/bienvenida.php
```

Con el siguiente contenido:

```
<h1>¡Hola!</h1>
<p>Esta es la acción de bienvenida del controlador de cuenta.</p>
```

Ahora tenemos que devolver la vista desde nuestra acción de Bienvenida. Laravel tiene una forma preciosa y expresiva de hacerlo, vamos a verla:

```
<?php

// application/controllers/cuenta.php
public function action_bienvenida($nombre, $lugar)
{
        return View::make('bienvenida');
}
```

Los lectores más empollones se habrán perctado que la sentencia está diciendo a Laravel que haga (make) un objeto View desde el archivo `application/views/bienvenida.php` (la extensión no hace falta) y la devuelva como resultado de la acción de bienvenida.

También habrás notado que el método `make()` busca en la carpeta `application/views` la vista. Si quieres especificar una ruta absoluta a un archivo de vista, usa simplemente el prefijo `path:` , por ejemplo `path: /ruta/a/mi/vista.php`.

Ahora si visitamos `/cuenta/bienvenida/Dayle/Gales` seremos recibidos con la página web que hemos definido en nuestro archivo de la Vista.

Ten en cuenta que también puedes usar la misma estructura de sub-carpetas y prefijos para bundles que hemos visto anteriormente con los controladores, para referirte a Vistas.

Sé lo que estás pensando, ahora nuestro mensaje de bienvenida no es muy dinámico. ¿Verdad? Veamos si podemos arreglarlo. Pasemos parámetros de la acción a la Vista, podemos hacerlo usando el método `with()` y veremos el elegante método de encadenado de Laravel en acción, ¡allá vamos!

```
<?php

// application/controllers/cuenta.php
public function action_bienvenida($nombre, $lugar)
{
        return View::make('bienvenida')
                ->with('nombre', $nombre)
                ->with('lugar', $lugar);
}
```

Usando el método `with()` podemos pasar cualquier valor (u objeto) a la Vista, y darle un 'pseudónimo' para que podamos usarlo en la vista. Hemos usado lo mismo para el nombre del parámetro y el pseudónimo, ¡pero puedes llamarlos como quieras!

Ahora usamos estos datos en nuestra vista:

```
1  <h1>¡Hola!</h1>
2  <p>¡Bienvenido a <?php echo $lugar; ?>, <?php echo $nombre; ?>!</p>
```

Ahora nuestra acción funciona de la misma forma en que lo hacía antes, solo que con mejor formato y un código fuente más bonito, separando toda la lógica de la capa visual.

En vez de usar varios métodos `with()`, puedes pasar una matriz como segundo parámetro a `make()` con parejas clave-valor. Esto puede ahorrarte algo de espacio consiguiendo el mismo resultado. He aquí un ejemplo:

```
1  <?php
2
3  // application/controllers/cuenta.php
4  public function action_bienvenida($nombre, $lugar)
5  {
6          $datos = array(
7                  'nombre'          => $nombre,
8                  'lugar' => $lugar
9          );
10
11         return View::make('bienvenida', $datos);
12 }
```

Nota: A mi me gusta llamar a mi matriz `$datos`, ¡pero puedes llamarla como quieras!

En un capítulo posterior, cubriremos las Vistas con más detalle, incluyendo el sistema de plantillas Blade, vistas anidadas y otras opciones de plantilla avanzadas.

3.4 Controladores RESTful

Las aplicaciones RESTful, responden a verbos HTTP coherentes, con los datos apropiados. Son muy útiles a la hora de crear APIs públicas para tus aplicaciones.

Con Laravel puedes hacer que las acciones de tu controlador respondan a verbos HTTP individuales usando acciones de controladores RESTful. Veámoslo en acción.

```php
<?php

// application/controllers/home.php
class Home_Controller extends Base_Controller
{
    public $restful = true;

    public function get_index()
    {
        //
    }

    public function post_index()
    {
        //
    }

}
```

Simplemente añade un atributo booleano público a la clase llamado `$restful` y establécelo a `true`, luego añade prefijos a tus acciones con los verbos HTTP a los que responder, en vez de `action_`.

Los verbos HTTP comunes son GET, POST, PUT y DELETE.

3.5 El controlador base, Base_Controller

Puedes editar el controlador `Base_Controller`, y extenderlo con otros Controladores para dar funcionalidad global a todos tus controladores. Añadir una acción index por fecto, valores de clases, ¡lo que quieras!

El controlador Base_Controller puede ser encontrado en `/application/controllers/base.php`.

Si no quieres usar un Base_Controller, simplemente haz que tus controladores extiendan la clase 'Controller' en vez de eso.

3.6 Enrutamiento avanzado

Ahora podemos mapear nuestros controladores y acciones a URIs en el formato `/controlador/accion/parametr` lo cual es genial, pero no deberíamos quedar restringidos a usar únicamente este formato. Veamos si podemos romper el molde. Anteriormente pusimos una declaración de controlador en `routes.php` pero ahora vamos a reemplazarla con el siguiente código:

```php
<?php

//application/routes.php
Route::get('superbienvenida/(:any)/(:any)', 'cuenta@bienvenida');
```

Aquí estamos diciendo que vamos a enviar todas las peticiones web con el verbo HTTP `GET`, y la dirección `/superbienvenida/(:any)/(:any)` a la acción `bienvenida` de nuestro controlador `cuenta`. Los segmentos `(:any)` son de relleno para nuestros parámetros y se pasarán en el orden en que sean facilitados. Usar `(:num)` solo permitirá números mientras que usar `(:any?)` creará un segmento opcional.

Ahora, ¡visitemos `/superbienvenida/Dayle/Gales` nos mostrará nuestra página de la vista!

La ventaja de definir rutas, es que podemos tener URLs en el orden que queramos, en el formato que nos apetezca. Por ejemplo también podemos tener:

```php
<?php

//application/routes.php
Route::get('superbievnenida/(:any)/(:any)', 'cuenta@bienvenida');
Route::get('bienvenida/(:any)/a/(:any)', 'cuenta@bienvenida');
```

Ahora tenemos dos rutas diferentes, con la misma página de resultados.

Merece la pena destacar que las Rutas son definidas "más arriba" en el fichero `routes.php` tienen mayor prioridad. Con el siguiente ejemplo ...

```php
<?php

// application/routes.php
Route::get('(:any)/(:any)', 'cuenta@bienvenida');
Route::get('bienvenida/(:any)/a/(:any)', 'cuenta@bienvenida');
```

... la segunda ruta nunca se activará porque `(:any)` en la primera ruta responderá a `bienvenida` en la segunda ruta. Este es un error común al comenzar con Laravel. ¡Asegúrate de tener clara la prioridad de tus rutas!

Cubriremos las rutas con más profundidad en el próximo capítulo, que también cubre enrutado con closures en vez de con controladores.

4 Rutas con closures

En este capítulo usaremos ***Rutas con closures*** en vez de ***Controladores con acciones***. Si aun no has leido el capítulo anterior sobre el uso de controladores, te recomiendo que empieces ahí ya que partiremos de lo que ya hemos aprendido en ese capítulo.

Las rutas nos permitan mapear nuestras URLs del framework a closures, lo cual es una forma muy limpia de contener nuestra lógica sin toda la 'parafernalia de clase'. Las closures son funciones anónimas (`function() {}`). Pueden ser asignadas y tratadas como cualquier otra variable. Para más información sobre Closures, revisa el artículo del API de PHP[1].

4.1 Closures

Vamos a echarle un vistazo a una ruta que enruta a una closure.

```
1  <?php
2
3  // application/routes.php
4  Route::get('/', function()
5  {
6          return View::make('home.index');
7  });
```

En este ejemplo vamos a responder a las peticiones a la raíz que usen el verbo HTTP `GET` con una closure que simplemente devuelve un objeto de vista. La salida es la página de bienvenida por defecto.

Por favor, ten en cuenta que solo necesitas la barra de la raíz para la página de inicio, el resto de las rutas lo omiten, por ejemplo:

```
1  <?php
2
3  // application/routes.php
4  Route::get('cuenta/perfil', function()
5  {
6          return View::make('cuenta.perfil');
7  });
```

Las rutas son REST por naturaleza, pero puedes usar `Route::any()` para responder a cualquier verbo HTTP. He aquí tus opciones:

[1] http://php.net/manual/es/functions.anonymous.php

```php
<?php

// application/routes.php
Route::get();
Route::post();
Route::put();
Route::delete();
Route::any();
```

Para pasar parámetros a tus closures, símplemente añade los comodines de vista habituales a la URI, y define los parámetros en tu closure. Se hacen coincider en orden de izquierda a derecha, por ejemplo:

```php
<?php

// application/routes.php
Route::get('usuario/(:any)/tarea/(:num)', function($usuario, $numero_tarea)
{
        // $usuario será reemplazada con el valor de (:any)
        // $numero_tarea será reemplazada con el valor del entero de (:num)

        $datos = array(
                'usuario'         => $usuario,
                'tarea'                => $numero_tarea
        );

        return View::make('tareas.para_usuario', $datos);
});
```

Los comodines disponibles son:

Comodín	Explicación
(:any)	Cadenas alfanuméricas.
(:num)	Cualquier número completo.
(:any?)	Parámetro opcional.

4.2 Redirecciones y rutas con nombre

Sería un poco estúpido que viéramos las rutas con nombre antes de ver el método que las usa, ¿verdad? VAmos a echar un vistazo a la clase `Redirect`, puede ser usada para redirigir a otra ruta. Podemos usarla de forma similar para devolver una vista.

```php
<?php

// application/routes.php
Route::get('/', function()
{
        return Redirect::to('cuenta/perfil');
});
```

¡Adorable! No podría ser más simple. Espera, puede, ¡y lo es! Vamos a echar un vistazo a una ruta con nombre:

```php
<?php

Route::get('cuenta/perfil', array('as' => 'perfil', 'do' => function()
{
        return View::make('cuenta/perfil');
}));
```

En vez de pasar una closure como segundo parámetro, pasamos una matriz con la clave 'do' apuntando a la closure. Esto nos permite añadir todo tipo de información extra a la ruta. La clave 'as', asigna un apodo par anuestra ruta, esto es de lo que van las rutas con nombre. Veamos cómo podemos usarla para mejorar el `Redirect::` de antes.

```php
<?php

Route::get('/', function()
{
        return Redirect::to_route('perfil');
});
```

Ahora hemos sacado esa URI de nuestro código, muy cuco. Todas las clases o helpers que se refieran a rutas, tienen un método similar para enrutar a una ruta con nombre, lo cual limpia bastante tu código, y hace que se lea como un libro. Además, si más tarde decides cambiar la URI de una página en concreto, ¡no tendrás que volver y cambiar todos los enlaces y redirecciones!

4.3 Filtros

Ok ok... Dije que iba a explicar las rutas en este capítulo, pero sinceramente no se me ocurre un lugar mejor para cubrir los Filtros, y están relacionados, por lo que vamos allá.

Los filtros son exactamente lo que te imaginas, son código, o pruebas que pueden ser realizadas 'antes' o 'después' de una ruta, y otros eventos claves del framework. Laravel tiene cuatro filtros de ruta especiales que son definidos por defecto en `application/routes.php`, vamos a echarles un vistazo.

```php
<?php

Route::filter('before', function()
{
        // Haz algo antes de cada petición a tu aplicación...
});

Route::filter('after', function($respuesta)
{
        // Haz algo después de cada petición a tu aplicación...
});

Route::filter('csrf', function()
{
        if (Request::forged()) return Response::error('500');
});

Route::filter('auth', function()
{
        if (Auth::guest()) return Redirect::to('login');
});
```

Las primeras dos rutas, ejecutan la closure encapsulada antes y después de cada petición (o acción de ruta /) a tu aplicación. Lo que hagas ahí es totalmente cosa tuya. Inicia librerías, facilita datos a 'algo', tu propia creatividad es tu única limitación. Hay filtros especiales que no necesitan ser asignados a rutas individuales.

El filtro 'csrf' se usa para prevenir

The 'csrf' filter is used to prevent 'cross-site-request-forgery[2]' y puede ser aplicado a las rutas que son el resultado de una petición AJAX para mayor seguridad.

El filtro 'auth' puede ser aplicado a cualquier ruta, para prevenir acceso a menos que un usuario haya iniciado sesión, usando el sistema de autentificación de Laravel.

Para aplicar filtros 'csrf' o 'auth' a tus Rutas, añade simplemente una nueva entrada a la matriz del segundo parámetro, tal que así:

```php
<?php

Route::get('/', array('as' => 'perfil', 'before' => 'auth', 'do' => functio\
n()
{
        return View::make('cuenta/perfil');
}));
```

[2] http://es.wikipedia.org/wiki/Cross_Site_Request_Forgery

La clave para la matriz puede ser tanto 'before' para ejecutar el filtro antes de la ruta, o 'after' para ejecutarlo después. Se pueden aplicar múltiples filtros separando sus nombres con una | (tubería). Por ejemplo `auth|csrf`.

Desde Laravel 3.1, si quieres añadir un filtro a un número de peticiones cuya URI coincida con un patrón específico, usa la siguiente línea:

```
<?php

Route::filter('pattern: admin/*', 'auth');
```

Esto aplicará el filtro 'auth' a todas las URIs qu ecomiencen con `admin/`.

4.4 Grupos de rutas

Puede que quieras aplicar un número de ajustes a un conjunto de rutas. Puedes hacerlo fácilmenteusando la opción de agrupado de rutas, echa un vistazo:

```
<?php

Route::group(array('before' => 'auth'), function()
{
        Route::get('panel', function()
        {
                // hacer algo
        });

        Route::get('dashboard', function()
        {
                // hacer algo
        });
});
```

Ahora tanto la ruta `panel` como `dashboard` están protegidas por el filtro `auth`.

Aunque las rutas pueden ser algo muy simple, también pueden ser tan complejas como quieras que sean. Usa los grupos de rutas para evitar duplicar reglas comunes a varias rutas y mantén tu código DRY. (¡No te repitas!)

En el próximo capítulo cubriremos la creación de enlaces, para que podamos movernos de una página de rutas a la siguiente.

5 Enlaces y URLs

Nuestra aplicación puede volverse un poco aburrida si solo tenemos una única página, y estoy seguro de que el usuario se enfadará rápidamente si tiene que andar escribiendo una URI completa para cambiar de páginas. Por suerte, los hiperenlaces están aquí para salvarte el día.

Si no has estado viviendo bajo una roca durante el último par de décadas, ya sabrás lo que son los hiper-enlaces, y no te aburriré con la explicación técnica. Antes de que podamos ver los enlaces, echemos un vistazo a cómo gestiona Laravel sus URLs.

5.1 Obteniendo URLs

Primero, vamos a echarle un vistazo a un problema. Habrás observado que los frameworks tienen estructuras de URL realmente únicas, algunos tienen un `index.php` en ellos y otros no. Otros tendrán rutas complejas. En la mayoría de los casos, usar una URL relativa como lo harías en otro sitios te podría llevar a problemas graves a largo plazo. Si decides dar a todos URLs completas, y decides mover la aplicación más tarde, puede que te encuentres abusando de la función de buscar y reemplazar de tu editor favorito.

¿Por qué no dejamos que Laravel haga todo el trabajo sucio? Laravel sabe la URL completa a tu aplicación, sabe cuándo estás usando reescritura de URLs o no. Incluso sabe sobre tus rutas. Vamos a aprovecharnos de esa información usando la clase URL para generar algunas URLs del sitio.

Comencemos buscando la URL a la raíz de nuestra web. Podemos usar el método `base()` para ello:

```
1  <?php
2
3  echo URL::base();
4  // http://miproyecto/
```

¡Genial! Ahora tenemos la URL completa a nuestro sitio, con o sin el `index.php` al final. Todo depende de tu actual configuración. Qué hay sobre la URL actual, la cual está siendo enrutada ahora mismo, ¿podemos obtenerla? ¡Por supuesto! Simplemente usa el método `current()`.

```
1  <?php
2
3  echo URL::current();
4  // http://miproyecto/esta/pagina
```

Por defecto, Laravel eliminará la cadena de consultas si esque está al final de la URL. Si queremos obtener la URL actual junto a la cadena de consulta, podemos usar el método `full()`.

```php
<?php

echo URL::full();
// http://miproyecto/esta/pagina?otra=cosa
```

Saber la URL base y la URL actual puede ser útil, pero sería más útil si pudiéramos obtener la URL a otras rutas o páginas, con las que podríamos crear enlaces.

Para generar una URL a una ruta, usamos el método `to()`, con la ruta que estamos intentando obtener. Esto es mucho más fácil que especificar la ruta completa, por ejemplo:

```php
<?php

echo URL::to('mi/ruta');
// http://miproyecto/mi/ruta
```

Si queremos enlazar a esta página de forma segura, a través del protocolo HTTPS, podemos usar el método `to_secure()`.

```php
<?php

echo URL::to_secure('mi/ruta');
// https://miproyecto/mi/ruta
```

¿Recuerdas el uso de rutas con nombres en el capítulo anterior? ¡Por supuesto que sí! He aquí un ejemplo una vez más:

```php
<?php

Route::get('login', array('as' => 'login', 'do' => function() {
        // código
}));
```

Aquí tenemos una ruta que hemos nombrado 'login' usando la palabra clave 'as'. Bueno, te dije que nos sería más tarde después, y ahora es el momento para hacer brillar a las rutas. Vamos a hacer un enlace a nuestra ruta con nombre 'login':

```php
<?php

echo URL::to_route('login');
// http://miproyecto/login
```

¡Guau! Creo que estarás de acuerdo en que es muy limpio y expresivo. ¿Qué pasa si necesitamos dar algún parámetro a nuestra ruta? Simple, pasa una matriz de parámetros como segundo parámetro al méotodo `to_route()`. Imaginemos por un segundo que nuestra ruta de login se parece a esto:

```php
<?php

Route::get('mi/(:any)/login/(:any)/pagina')..
```

Es una ruta terrible, por favor uses URLs tan horribles como esta, pero te ayudará a ilustrar lo que quiero explicarte. Como verás, si pasas parámetros al método `to_route()`, Laraval se encargará de establecer el orden en que deberían aparecer en la URL, y devolver la URL con parámetros en el orden correct. ¡Precioso!

```php
<?php

echo URL::to_route('login', array(5, 7));
```

El método de arriba nos dará:

```
http://miproyecto/mi/5/login/7/pagina
```

¡Genial! Ahora nuestras rutas se verán requetelimpias, siempre que no creemos rutas tan complicadas como la anterior.

Aunque ya hemos visto las rutas, no debemos olvidarnos de los controladores. A nadie le gusta quedarse fuera. Afortunadamente, hay una forma limpia y buena de crear un enlace a la acción de un controlador. Símplemente usa el método `to_action()`, por ejemplo:

```php
<?php

echo URL::to_action('dashboard@inicio');
// http://miproyecto/dashboard/inicio
```

Simplemente pasa el nombre del controlador y la acción, separados por una `@` (arroba). De nuevo, puedes pasar una matriz de parámetros adicionales como segundo parámetro al método `to_action()` si lo necesitas.

Si estamos tratando con activos estáticos, una hoja de estilos CSS por ejemplo, en vez de las páginas a rutas necesitaremos una URL muy diferente. No podemos usar `URL::to()` porque eso podría un `index.php` en la URL, o resolverla una de nuestras rutas.

En vez de eso, podemos usar el método `to_asset()` para generar el enlace correcto. Simplemente pasa la ruta relativa de la aplicación a nuestra hoja de estilos, y Laravel se encargará del resto.

```php
<?php

echo URL::to_asset('css/estilo.css');
```

Esta línea nos dará

```
http://miproyecto/css/estilo.css
```

Estos métodos son realmente útiles, pero Laravel va un paso más allá y nos da unos `helpers` más cortos que quedan muy bien al usarlos en nuestras vistas. He aquí una lista de estos "métodos cortos", y sus alternativas más largas.

Helper	Método
url()	URL::to()
asset()	URL::to_asset()
route()	URL::to_route()
action()	URL::to_action()

5.2 Generando enlaces

Ahora que podemos obtener los enlaces de nuestro sitio, el siguiente paso lógico sería usarlos para crear hiper-enlaces. Sé lo que estás pensando, puedo usar algo así:

```
<a href="<?php echo URL::to('mi/pagina'); ?>">Mi página</a>
```

Seguro que eso funcionaría, pero es un poco feo. En Laravel, si algo es un poco feo, siempre hay una mejor forma de hacerlo. Los enlaces no son ninguna excepción.

¿Por qué no usamos la clase HTML para generar un enlace? Después de todo, para eso está la clase HTML. Se usa para generar todo tipo de etiquetas HTML.

```
<?php echo HTML::link('mi/pagina', 'Mi página'); ?>
```

¡Eso se ve aun mejor! Veamos el resultado.

```
<a href="http://miproyecto/mi/pagina">Mi página</a>
```

Si eres un SEO del ninja, y no puedes ver un enlace sin un atributo `title`, pasa simplemente una matriz adicional.

```
<?php echo HTML::link('mi/pagina', 'Mi página', array('title' => '¡Mi págin\
a!')); ?>
```

Lo cual nos da:

```
<a href="http://miproyecto/mi/pagina" title="¡Mi página!">Mi página</a>
```

Una de las grandes características de Laravel es la consistencia que tienen al nombrar los métodos. Muchos de los métodos `HTML::link` siguen un patrón similar de nombres al usado por los métodos `URL::to`, lo cual los hace más fácil de recordar. Echemos un vistazo a cómo podríamos enlazar a una página segura (a través de HTTPS).

```php
<?php

HTML::link_to_secure('mi/pagina', 'Mi página');
// <a href="https://miproyecto/mi/pagina">Mi página</a>
```

También podemos usar `link_to_route`, para crear un enlace a una ruta con nombre, tal y como hicimos con la librería URL.

```php
<?php

HTML::link_to_route('login', '¡Login!');
// <a href="http://miproyecto/login/pagina">¡Login!</a>
```

De nuevo, podemos usar el método `link_to_action()` para enlazar a una pareja controlador-acción. Por ejemplo:

```php
<?php

HTML::link_to_action('cuenta@login', '¡Login!');
// <a href="http://miproyecto/cuenta/login">¡Login!</a>
```

Laravel incluso nos da un método para crear fácilmente, enlaces 'mailto' desde una dirección de correo. Echemos un vistazo.

```php
<?php

HTML::mailto('me@daylerees.com', '¡Mándame un correo!');
// <a href="mailto:me@daylerees.com">¡Mándame un correo!</a>
```

¡Increíble y simple!

Ahora que sabes cómo crear URLs y enlaces, tus aplicaciones comenzarán a crecer en tamaño, cubriendo tantas rutas hasta que consuman nuestro planeta y comiencen la conquista del unive...

¡Tus aplicaciones serán mucho más interesantes!

6 Formularios

Los formularios son una parte importante de cualquier aplicación web. Ayudan a controlar el flujo de la aplicación, nos premiten recibir datos de nuestros usuarios y hacer decisiones que afectan a la funcionalidad de nuestras aplicaciones. También es lo que más odio en el mundo escribir.

Afortunadamente para mi, la clase de formularios de Laravel se encarga de gran parte del trabajo duro dándonos útiles métodos para generar elementos comunes de un formulario. Vamos a usar la clase formulario para crear un simple formulario web en una de nuestras vistas.

6.1 Creando formularios

```
1  // form.php
2  <?php echo Form::open('mi/ruta'); ?>
3
4          <!-- Nombre de usuario -->
5          <?php echo Form::label('usuario', 'Usuario'); ?>
6          <?php echo Form::text('usuario'); ?>
7
8          <!-- Contraseña -->
9          <?php echo Form::label('password', 'Contraseña'); ?>
10         <?php echo Form::password('password'); ?>
11
12         <!-- Botón de login -->
13         <?php echo Form::submit('Login');
14
15 <?php echo Form::close(); ?>
```

Para un momento, echa un vistazo al código fuente del formulario porque no habrás visto nunca nada tan limpio. Dilo alto para ti mismo, vamos. Esperaré.

> Nunca he visto un formulario tan limpio.

Estás en lo cierto, es hermoso. Echemos un vistazo al código generado para asegurarme de que no te estoy enseñando mal, ya sabes... ¿para divertirnos?

```
<form method="POST" action="http://mysite/my/route" accept-charset="UTF-8">

        <!-- Nombre de usuario -->
        <label for="usuario">Usuario</label>
        <input type="text" name="usuario" id="usuario">

        <!-- Contraseña -->
        <label for="password">Contraseña</label>
        <input type="password" name="password" id="password">

        <!-- Botón de login -->
        <input type="submit" value="Login">

</form>
```

Genial, ¡funcionó! Quiero decir, ¡por supuesto que lo hizo! Vamos a verlo línea a línea para ver cómo funciona. En nuestra primera línea tenemos el método `Form::open()`, que crea una etiqueta de apertura de formulario para nosotros.

```
<?php echo Form::open('mi/ruta'); ?>
```

El primer parámetro al método es la URI a la que queremos enviar el formulario. El segundo parámetro es el `METHOD` usado para enviar el formulario. Si no facilitas uno como cadena, Laravel sume que quieres un formulario `POST`, que es lo más común.

El tercer parámetro también es opcional, puedes pasar una matriz de parejas `atributo => valor` para añadir atributos extra a la etiqueta `<form>`. Por ejemplo, si quisieras seleccionar el formulario con JavaScript, querrías pasar `array('id' => 'miformulario')` como tercer parámetro para dar al elemento una `id`.

Para enviar un formulario a una URI segura (https) necesitarás usar el método `open_secure()` en vez de `open()`. Acepta los mismos parámetros.

Si quieres poder enviar ficheros a través de tu formulario, tendrás que usar `multipart/data`, usa `open_for_files()` en vez del método `open()`. Este método también acepta los mismos parámetros.

Finalmente, si quieres enviar un formulario a una URI segura y subir ficheros, tendrás que usar el método `open_secure_for_files()` que, una vez más, acepta los mismos parámetros y es una combinación de `open_secure()` y `open_for_files()`.

6.2 Añadiendo etiquetas

La siguiente línea contiene el método `Form::label()` que es usado para crear un elemento `<label>`. El primer parámetro es el `name` del elemento `input` que será usado para el atributo `for=""`. El segundo parámetro será usado como contenido del elemento `label`. Puedes pasar una matriz como tercer parámetro opcional, para aplicar atributos HTML adicionales.

6.3 Generando campos

Después, tenemos los generadores de campos. Estos métodos ayudan a generar todos los elementos HTMLs que son comunes en los formularios. En el ejemplo anterior usamos los métodos `text()` y `password()` para generar los elementos `<input type="text".` e `<input type="password"...`

El primer parámetro para el método, es el valor del atributo `name`. El segundo parámetro opcional es el valor por defecto del elemento. Una vez más, podemos pasar una matriz de atributos HTML como tercer parámetro opcional. ¿Ya has empezado a ver un patrón?

Los métodos `textarea()` y `hidden()` también aceptan los mismos parámetros.

Las casillas de verificación (checkbox) pueden ser creadas usando el método `checkbox()`. Siendo el primer parámetro el nombre del elemento y el segundo el valor. El tercer parámetro es un booleano opcional para indicar si el elemento está inicialmente marcado o no. El cuarto parámetro opcional, nuevamente, establece atributos adicionales. De hecho, vamos a asumir que todos los elementos futuros aceptan una matriz de atributos como su último parámetro opcional. Vamos a echar un vistazo al generador de casillas de verificación:

```
1 <?php echo Form::checkbox('admin', 'yes', true, array('id' => 'admin-checke\
2 r')); ?>
```

El método `radio()` crea botones de opción (radio) y comparten los mismos parámetros que el método `checkbox()`.

Después, tenemos los select, los elementos más feos de todos los elementos de un formulario. Por suerte, todo lo que tenemos que hacer es pasar un nombre, una matriz de `clave => etiqueta` y un parámetro opcional para decidir qué opción debería estar seleccionada por defecto al método `select()`. Nuestro select será generado para nosotros, por ejemplo:

```
1 <?php
2
3 Form::select('roles', array(
4         0 => 'Usuario',
5         1 => 'Miembro',
6         2 => 'Editor',
7         3 => 'Administrador'
8 ), 2);
```

y obtenemos...

```
<select name="roles">
        <option value="0">Usuario</option>
        <option value="1">Miembro</option>
        <option value="2" selected="selected">Editor</option>
        <option value="3">Administrador</option>
</select>
```

¡Genial! Ahora es momento de enviar nuestro formulario.

6.4 Generando botones

Los métodos generadores `submit()` y `button()`, aceptan ambos los mismos parémtros. Siendo el primero el `value` del elemento HTML y el segundo la matriz abitual de atributos.

```
<?php

Form::submit('Login');
```

6.5 Campos secretos

También hay un número adicional de métodos de generación para campos de formularios menos comunes, que no están cubiertos en la documentación. He aquí una lista, con sus parámetros.

```
<?php

Form::search($name, $value = null, $attributes = array());
Form::email($name, $value = null, $attributes = array());
Form::telephone($name, $value = null, $attributes = array());
Form::url($name, $value = null, $attributes = array());
Form::number($name, $value = null, $attributes = array());
Form::date($name, $value = null, $attributes = array());
Form::file($name, $attributes = array());
```

6.6 Token CSRF

Si pretendes usar el filtro `CSRF` (que cubriremos más adelante) puedes añadir el token CSRF a tu formulario usando el método `token()`, por ejemplo:

```
<?php

Form::token();
```

6.7 Macros de formulario

Laravel facilita muchos métodos para los campos, pero ¿qué pasa si necesitamos algo un poco más personalizado? Por suerte, Laravel nos facilita el método `macro()` para permitirnos crear nuestros propios generadores de campos.

Pasando un nombre de campo y una closure al método `macro()`, podemos definirnuestro propio generador de campos. Echemos un vistazo:

```
<?php

Form::macro('color_zapato', function() {
        return '<input type="color_zapato" />';
});
```

Ahora podemos usar la clase `Form` para generar nuestro campo de color de zapato de la misma forma que con cualquier otro campo. Por ejemplo:

```
<?php echo Form::shoe_size(); ?>
```

Si necesitas usar parámetros, simplemente añádelos como parámetros a la closure. ¡Disfruta creando tus propios generadores de campos!

7 Gestionando la entrada de datos

Ahora que ya sabemos cómo crear formularios, tenemos que aprender cómo gestionar los datos que nos envían a través de ellos. Como siempre, Laravel nos facilita una forma ultra-limpia de manejar esos datos. No hace falta interactuar con matrices de PHP como `$_POST`, `$_GET` ni `$_FILES`. Seamos sinceros, eran horribles.

7.1 Datos de peticiones

Usemos la clase `Input` para hacernos cargo de esto:

```
1  <?php
2
3  $panda = Input::get('panda');
```

¡Ahora tenemos un Panda! Genial... ya tenemos demasiados. Una cosa que tienes que recordar sobre el método `get()` en la clase `Input` es que no se refiere a `$_GET`. El método `get()` es tan solo una forma simple y expresiva para obtener todo tipo de datos. La clase `Input` también responde a `get()` con todo tipo de datos de petición, incluyendo `$_POST`.

Si una parte de la petición no está establecida, la clase Input devolverá `null`. Si pasas un segundo parámetro al método `get()` y el índice no existe, el método devolverá el segundo parámetro. ¡Muy útil!

```
1  <?php
2
3  $panda = Input::get('panda', 'Muffin');
```

Si te gustaría recibir la matriz de petición completa, simpelemente sáltate el índice. Tan fácil como eso.

```
1  <?php
2
3  $maspandas = Input::get();
```

Por defecto, la matriz `get()` no incluye valores de la matriz `$_FILES`, sin embargo si usas `all()` en vez de `get()` contendrá los archivos también.

```
1  <?php
2
3  $pandas_y_archivos = Input::all();
```

Si queres saber si existe una parte en concreto sin obtenerla, simplemente usa el elegante y áltemente expresivo método `has()` que devolverá un resultado booleano.

```php
<?php

$tenemos_un_panda = Input::has('panda');
```

7.2 Archivos

Para acceder a un elemento de la matriz `$_FILES`, simplemente haz una llamada al método `Input::file()`, por ejemplo:

```php
<?php

$archivo = Input::file('cuchara');
```

Si simplemente quieres obtener un atributo del archivo, añade un punto y la clave del atributo al primer parámetro, por ejemplo para obtener arl tamaño del archivo:

```php
$tamano = Input::file('cuchara.size');
```

Una vez más, llamar al método sin un parámetro obtendrá la matriz completa de archivos.

*** Nota: Puedes usar esta sintaxis para acceder a matrices multidimensionales. Simplemente usa un punto para declarar un índice de matriz anidado. ***

```php
$archivos = Input::file();
```

7.3 Datos flash

Los datos flash son una forma útil de almacenar datos para ser utilizados en la próxima petición. Puede ser un método útil para re-rellenar formularios.

Para `flashear` todos los datos de la petición actual a la sesión, para que se accesible en la próxima petición, simplemente usa el método `flash()`.

```php
<?php

Input::flash();
```

Si únicamente quieres flashear una porción de los datos de la actual petición, simplemente pasa `'only'` como primer parámetro al método y una matriz de nombres de campos que quieres 'flashear' como segundo parámetro.

```php
<?php

Input::flash('only', array('maria', 'ramon'));
```

Ahora nos llevamos a María y a Ramón con nosotros a la próxima petición. De manera alternativa podemos especificar una lista de campos que no queremos traernos con nosotros usando la opción `except`. Por ejemplo:

```php
<?php

Input::flash('except', array('tia_francisca'));
```

Ahora podemos olvidarnos de la Tía Francisca, es muy arrogante y no le gusta nuestro panda rojo.

Ahora podemos usar el método habitual, `Redirect::to()`, para movernos a otra petición. Desde ahí podremos usar el expresivo método `Input::old()` para obtener un valor que ha sido 'flasheado' en una petición anterior.

```php
<?php

$maria = Input::old('maria');
```

Como puedes ver, María ha sobrevivido a la transición. Puedes hacerte a la idea de que los datos flash son como esos transportadores de Star Trek, que mueven a Kirk y a sus colegas de una petición a la siguiente.

Una vez más, puedes saltarte los parámetros para obtener una matriz completa de los datos flash.

```php
<?php

$gente = Input::old();
```

Puedes usar el método `had()` para ver si un índice de datos flash existe.

```php
<?php

Input::had('tia_francisca');
```

Por supuesto que no, odiamos a la Tía Francisca.

Laravel no sería el framework que es, sin sus atajos y métodos expresivos. Echemos un vistazo a un ejemplo que lo demuestra en acción.

```php
<?php

return Redirect::to('fiesta')->with_input();
```

El método `with_input()` flasheará todos los datos de la petición por nosotros, también acepta los mismos `only` y `except` que nuestro método `flash()`.

```php
<?php

return Redirect::to('fiesta')->with_input('only', array('maria', 'ramon'));
return Redirect::to('fiesta')->with_input('except', array('tia_francisca'))\
;
```

8 Validación

La validación es una parte importante de muchas aplicaciones web. Nunca puedes fiarte de tus usuarios, llevan tramando tu destrucción semanas abusando de tus formularios con JavaScript malicioso.

No podemos dejarles ganar, no deben destruir nuestras preciosas aplicaicones. Validemos todos los datos facilitados por los usuarios, de esa manera no podrán hacernos nada de daño.

Naturalmente, Laravel tiene una librería, con el acertado nombre de `Validation` que hará todo el trabajo duro por nosotros.

8.1 Estableciendo una validación

Comencemos creando un formulario imaginario, cierra tus ojos e imagina un largo y bonito formulario con muchos campos... ohh ohh... ¿puedes volver a abrir los ojos?

Bien, asumiré que te cansaste de esperar, abriste los ojos y estás de vuelta conmigo y nuestro formulario imaginario. Vamos a obtener los datos de ese formulario.

```
1  <?php
2
3  $campo = Input::get();
```

Ahora bien, normalmente no querrás usar el método `get()`, ya que es una forma fácil de rellenar tu matriz de datos con datos extra que no necesitas. De hecho, el sitio de colaboración en línea Github fue victima de asignación masiva. He usado `get()` para simplificarlo un poco. En tus aplicaciones trae los datos que te interesen únicamente por favor.

Nuestra matriz de datos ahora contiene algo que se parece a esto:

```
1  <?php
2
3  array(
4          'nombre' => 'Juan',
5          'edad'   => 15
6  )
```

Vamos a validar esos campos para asegurarnos de que tienen sentido en nuestra aplicación, antes de que podamos comenzar el proceso de validación tenemos que crear un conjunto de reglas que serán usadas para validar cada campo. Con la clase `Validator` las reglas se definen en formato de matriz. Vamos a echarle un vistazo.

```php
<?php

$reglas = array(
        'nombre'         => 'required|min:3|max:32|alpha',
        'edad'        => 'required|integer|min:16'
);
```

Genial, ahora tenemos algunas reglas. La clave de la matriz es el campo que está siendo validado, y el valor contiene un número de reglas de validación separadas por una pleca |.

En nuestro caso estamos validando que ambos campos contengan un valor, usando la regla `required`. La longitud del nombre de los usuarios debe contener al menos 3 caracteres (`min:3`) y una longitud máxima de 32 caracteres (`max:32`). La regla `alpha` revisará que el campo solo contenga letras.

Nuestro campo `edad` debe contener un `integer` (entero) y el valor debe ser al menos `16`. Verás que la regla `min` se ha adaptado al contenido que está validando, ¡muy inteligente!

No te preocupes, cubriremos todas las reglas de validación después. Ahora veamos la validación en acción, allá vamos:

```php
<?php

$v = Validator::make($campo, $reglas);
```

Hemos creado nuestro objeto de validación con el método `make()`, pasándole nuestra matriz del campo y nuestra matriz de reglas. ¡Veamos si valida!

```php
<?php

if( $v->fails() )
{
        // código para error de validación :(
}
else
{
        // ¡código para éxito al validar!
}
```

Como puedes ver, usamos el método `fails()` para revisar el resultado del intento de validación. Devolverá `true` si la validación falló, y `false` si tuvo éxito.

Si prefieres una perspectiva más positiva en tus validaciones, puedes usar el método `passes()`, que devuelve el valor opuesto:

```php
<?php

if( $v->passes() )
{
        // ¡código para éxito al validar!
}
else
{
        // código para error de validación :(
}
```

Ale, ya somos positivos y podemos bailar sobre arcoíris con unicornios brillantes.

8.2 Errores

Si la validación falla, que será porque nuestro usuario tiene menos de 16 años (lamento destruir tus unicornios), querrás saber qué es lo que fue mal. El validador facilita un objeto de mensajes de errores, llamada `errors`, que nos ayuda a obtener la información que necesitamos.

El objeto `errors` tiene métodos similares a la clase `Input` por lo que no tendré que ir sobre todos los métodos. Vamos a obtener una matriz de errores para un campo específico.

```php
<?php

$errores_edad = $v->errors->get('edad');
```

Ahora tendremos una matriz que contiene todos los errores asociados al campo edad.

```php
<?php

array(
        'The age must be at least 16.'
)
```

La mayoría de las veces, descubro que uso el método `first()` en mis vistas, que devuelve el primer valor de la matriz si existe, o `null` en caso contrario. Por ejemplo:

```php
<?php echo Form::label('usuario', 'Usuario') ?>
<?php echo $errors->first('usuario') ?>
<?php echo Form::text('usuario') ?>
```

Ahora nuestros errores de validación aparecerán para este campo si es que están presentes. También puedes pasar un segundo parámetro a `first()` para dar formato a la salida:

```
<?php echo $errors->first('usuario', '<span class="error">:message</span>')\
 ?>
```

¡Precioso!

También puedes usar `has()` para revisar si existe un error y `all()` para obtener todos los errores como una matriz.

8.3 Reglas de validación

He aquí una lista de reglas de validación y su propósito:

required

Se asegura de que hay un valor presente para el campo, y que no es una cadena vacía.

alpha

La cadena solo debe contener letras (caracteres alfabéticos).

alpha_num

La cadena solo debe contener letras y números. Útil para nombres de usuario.

alpha_dash

La cadena solo debe contener letras, números y guiones bajos. Útil para almacenar slugs de URLs.

size:5

(cadena) La cadena debe tener exactamente cinco caracteres. *(numérico)* El valor debe ser 5.

between:5,10

(cadena) El tamaño de la cadena debe estar entre cinco y diez caracteres. *(numérico)* El valor debe estar entre 5 y 10.

min:5

(cadena) La longitud de la cadena debe ser de cinco caracteres o más. *(numérico)* El valor debe ser igual o superior a 5. *(archivo)* El tamaño del archivo debe ser de 5 kilobytes o más.

max:5

(cadena) La longitud de la cadena debe ser menos o igual a cinco caracteres. *(numérico)* El valor debe ser menor o igual a 5. *(archivo)* El tamaño del archivo debe ser de 5 kilobytes o menos.

numeric

El valor debe ser numérico.

integer

El valor debe ser un entero o un número completo.

in:rojo,verde,azul

Se asegura de que el valor está en la lista de valores facilitados.

not_in:rosa,morado

Se asegura de que ninguno de los valores coincide con el facilitado.

confirmed

El valor del campo debe equivaler a un campo de confirmación, nombrado con el formato ‘

accepted

El valor debe ser igual a ‘yes’ o 1. Útil para validar casillas de verificación.

same:edad

El campo debe ser igual al campo especificado en la misma regla.

different:edad

El campo no debe coincidir con el campo especificado en la misma regla.

match:/[a-z]+/

El campo debe coincidir con la expresión regular facilitada.

unique:usuarios

Este es uno de mis favoritos. El validador buscará en la tabla de la base de datos `users` y se asegurará de que el valor es único en la columna que tiene el mismo nombre. Útil para asegurarte de que no existan los nombres de usuario o correos duplicados.

Si quieres espeficicar un nombre de columna, simplemente pásalo como segundo parámetro:

```
1 unique:usuarios,apodo
```

También puedes forzar la regla para que ignore un determinado `id` pasándolo como tercer parámetro.

```
1 unique:usuarios,apodo,5
```

exists:colores

Actúa al contrario que `unique`. El valor debe existir en la tabla de la base de datos. De nuevo, puedes pasar un segundo parámetro para referirte a otra columna.

before:1984-12-12

La fecha facilitada por el campo, debe haber ocurrido antes de la plantilla facilitada a la regla `before`.

Los filtros `before` y `after` usan `strtotime()` para calcular una marca de tiempo para la comparación. Lo cual implica que puedes usar ciertos trucos como:

```
1 before:next Thursday
```

Por desgracia, yo fui el que añadió esta funcionalidad por lo que si se rompe puedes venir a gritarme... ¡lo siento!

after:1984-12-12

Similar a `before`, solo que la fecha debe haber ocurrido después de la fecha facilitada a la regla `after`.

email

El valor debe ser una dirección de correo válida.

url

El valor debe coincidir con el formato de una URL.

active_url

El valor debe coincidir con una URL válida y activa. Se usará `checkdnsr` para verificar que la URL esté activa.

mimes:png,mp3

El valor debe ser un archivo en `$_FILE` cuyo tipo MIME coincide con las extensiones de archivo facilitadas. Puedes añadir tipos de MIME adicionales a la matriz en `config/mimes.php`.

image

El archivo subido debe ser una imagen.

8.4 Mensajes de error personalizados

Para mi gusto, los mensajes de error son bastante descriptivos, pero tus clientes pueden tener sus propias ideas, o puede que necesites traducirlos. Veamso cómo podemos personalizar nuestros mensajes de error para ajustarlos a nuestras necesidades.

Puedes editar los mensajes de errores de validación modificando el archivo directamente en `application/language/en/validation.php`...

```
1 ...
2 "after"          => "The :attribute must be a date after :date.",
3 "alpha"          => "The :attribute may only contain letters.",
4 "alpha_dash"     => "The :attribute may only contain letters, numbers, and \
5 dashes.",
6 ...
```

Laravel reemplaza el marcador `:attribute` con el nombre del campo. Existen otros marcadores en las reglas aunque su propósito es bastante comprensible a simple vista.

Si prefieres cambiar los mensajes para un único formulario, en vez de editarlos globalmente, puedes pasar una tercera matriz de mensajes al método `Validator::make()`.

```php
<?php

$mensajes = array(
    'same'    => 'El :attribute y :other deben coincidir, ¡idiota!',
    'size'    => 'El :attribute debe ser de :size exactamente, ¡que no te e\
nteras!'
);

$v = Validator::make($campo, $reglas, $mensajes);
```

Genial, ¡ahora tenemos mensajes personalizados! Incluso podemos especificar mensajes de error para campos individuales, estableciendo la clave del mensaje a `campo_regla`. Por ejemplo:

```php
<?php

$mensajes = array(
    'edad_required'    => '¡Debes tener al meno un cumpleaños!'
);
```

8.5 Reglas de validación personalizadas

El validador te permite añadir reglas adicionales que se ajusten a las necesidades de tu aplicación. Vamos a ponernos manos a la obra y ver cómo podemos registrar una nueva regla de validación.

```php
<?php

Validator::register('superdooper', function($atributo, $valor, $parametros)\
{
        return $valor == 'superdooper';
});
```

Nuestra regla de validación recientemente creada, `superdooper` se asegurará de que nuestro coincide con la cadena 'superdooper'. Nuestras validaciones personalizadas deben devolver `true` al validar, o `false` en error.

El valor `$atributo` será el nombre del campo que está siendo validado, y `$valor` contendrá, por supuesto, el valor del campo.

El atributo `$parametros` contiene una matriz de parámetros que hayan sido pasados a la regla tras el símbolo de dos puntos y separados por comas.

Como hemos creado un nuevo validador aun no habrá mensajes de error asociados con él, por lo que tendremos que añadir uno para que Laravel sepa qué decir cuando falle. Podemos añadir un mensaje de error de la misma forma en que lo hicimos antes:

```php
<?php

'superdooper' => 'El :attribute debe ser superdooper, ¡¿vale trooper?!',
```

Una vez más, puedes pasar la matriz de mensajes de error adicional como tercer parámetro al método `Validator::make()`, o simplemente añadirlo al archivo `application/language/en/validation.php` para guardarlo de forma segura.

8.6 Clases de validación

Si queremos ofrecer muchos métodos nuevos de validación, o reusarlos en más proyectos, lo mejor sería crear una clase de validación. Las clases de validación extienden los datos de Laravel, y los sobrecargan con métodos adicionales de validación. La clase se crea en el directorio `application/libraries` para una carga sencilla, pero puedes ponerla donde quieras mientras esté registrada con el `Autoloader` (en un capítulo posterior). Echemos un ojo a la clase.

```php
<?php

// application/libraries/validator.php

class Validator extends Laravel\Validator {

    public function validate_increible($atributo, $valor, $parametros)
    {
        return $valor == 'increible';
    }

}
```

Como puedes ver, nuestra clase `Validator` extiende la clase en el espacio `Laravel\Validator` y facilita validaciones adicionales en la forma de métodos `validate_<nombreregla>`. Los métodos de validación aceptan los mismos parámetros que la closure `Validator::register()`, y funcionan de la misma forma.

Para usar nuestra nueva clase de validación, tenemos que eliminar el alias existente para `Validator` de nuestro archivo `application/config/application.php`. De esta forma, Laravel usará nuestra clase creada en vez de la que hay en la carpeta del código fuente de Laravel.

Puedes usar este método para reemplazar los métodos de validación original con los tuyos propios. Por ejemplo, podrías crear un método `validate_size` y calcular el tamaño de una forma alternativa.

Te sugeriría que añadieras mensajes de error personalizados al archivo de idioma de validación cuando uses clases de Validación. Esto te permitirá una migración mucho más sencilla a otro proyecto y no tendrás que rebuscar en el código para encontrar todos los mensajes usados.

En el próximo capítulo, cubriré algunos escenarios de validación típicos en las aplicaciones web. Esto incluye entrada de datos, edición de datos, re-rellenado de datos y mucho más.

9 Migraciones

Inciialmente, intenté incluir una guía al constructor de consultas Fluent en este capítulo, pero creo que el capítulo se volvió demasiado largo por lo que decidí cubrir únicamente la configuración de la base de datos y las migraciones en este capítulo. Prepárate para una buena y larga explicación sobre Fluent en el próximo capítulo.

Las migraciones son una de las características que más me gustan de Laravel. Odio escribir SQL y la clase `Schema` me permite crear mis tablas de forma sencilla, ¡sin escribir una sola línea de ese repugnante "lenguaje"! Y no solo eso, el código de `Schema` es precioso y se lee como un libro.

Si no te has encontrado con las migraciones antes, son una forma de describir cambios a tu base de datos, en archivos, para que los diferentes entornos/instalaciones de tu aplicación estén al tanto del esquema actual. Los cambios al esquema también pueden ser revertidos o hacerles 'marcha atrás'. Las migraciones también pueden ser usadas para rellenar datos en tablas con datos de ejemplos.

9.1 Configuración de la base de datos

Dirígete al archivo de configuración `application/config/database.php`. Si alguna vez has instalado una aplicación PHP, estarás familiarizado con este tipo de archivos. Tienes tus datos de acceso a la base de datos a mano, ¿verdad? Si no, ¡ve a buscarlos ahora mismo!

¿Ya has vuelto? Genial, vamos a empezar a profundizar.

Desplázate hasta la clave `connections` de la matriz, donde verás un número de opciones para unos pocos de tipos de bases de datos. Rellena los parámetros de conexión para la base de datos de tu elección. Yo voy a empezar con la típica base de datos de MySQL.

```
<?php

'mysql' => array(
        'driver'   => 'mysql',
        'host'     => 'localhost',
        'database' => 'codigodivertido',
        'username' => 'root',
        'password' => 'lospandascomenbamboo',
        'charset'  => 'utf8',
        'prefix'   => '',
),
```

Ya estamos. Ahora querrás subir un poco, y cambiar la clave `default` de la matriz para que refleje la base de datos que estás usando. Para mi, ya está en MySQL.

```
<?php

'default' => 'mysql',
```

Ahora que tenemos la base de datos configurada, necesitamos algunas tablas con las que jugar. ¡Vamos a por las migraciones!

9.2 Migraciones

Comencemos creando un archivo de migración.

Vamos a usar la interfaz de línea de comandos de Laravel, `artisan` para crear nuestra nueva migración. Para ejecutar Artisan tendrás que tener la CLI de PHP instalada (normalmente se instala junto al servidor web). También tendrás que abrir una ventana del terminal en la carpeta de tu paquete de Laravel donde existen los archivos de 'artisan'. Bien, vamos a escribir nuestro primer comando de artisan:

```
php artisan migrate:make crear_usuarios
```

Le estamos diciendo a artisan, que ejecute el método `make` en la tarea `migrate`. Le pasamos un nombre a nuestra migración para identificarlo. Me gusta nombrarlas con el nombre de la acción que estamos llevando a cabo. En este caso estamos creando la tabla de usuarios.

Echemos un ojo al resultado:

```
Great! New migration created!
```

¡Qué entusiasta! Si vemos nuestra carpeta `application/migrations`, veremos un archivo llamado `2012_03_30_220459_crear_usuarios.php`, aunque bueno, ¡la tuya puede que no se llame así! Verás que artisan toma la fecha actual, y añade el tiempo en formato `His` para crear el nombre del archivo. El motivo para ello es que las fechas son muy importante para las migraciones, el sistema necesita poder saber en qué orden debe aplicar los cambios.

Vamos a abrir el fichero y echemos un vistazo a la clase de migración:

```
<?php

class Crear_Usuario {

        /**
         * Make changes to the database.
         *
         * @return void
         */
        public function up()
```

```
11          {
12                  //
13          }
14
15          /**
16           * Revert the changes to the database.
17           *
18           * @return void
19           */
20          public function down()
21          {
22                  //
23          }
24
25  }
```

Como puedes ver, nuestra clase de migración consiste en dos métodos, `up()` es responsable de hacer todos los cambios a la base de datos, mientras que `down()` hace exactamente lo contrario. De esta forma, una migración puede ser realizada y hacerle marcha atrás cuando sea necesario. Si fueras a crear una tabla con el método `up()`, usaríamos `DROP` en la misma tabla en el método `down()`.

Entonces, ¿cómo hacemos cambios en nuestra base de datos? Sin duda con alguna compleja consulta SQL, ¿verdad? Jaja, no. Estamos usando Laravel, si es feo, no está en el framework. Echemos un vistazo a cómo crear una tabla con la clase `Schema` de Laravel.

```
1   <?php
2
3   Schema::create('usuarios', function($tabla) {
4           // id auto incremental (Clave Primaria)
5           $tabla->increments('id');
6
7           // varchar 32
8           $tabla->string('usuario', 32);
9           $tabla->string('correo', 320);
10          $tabla->string('password', 64);
11
12          // int
13          $tabla->integer('rol');
14
15          // boolean
16          $tabla->boolean('activo');
17
18          // created_at | updated_at DATETIME
19          $tabla->timestamps();
20  });
```

Llamamos al método `create()` en la clase `Schema` para crear una nueva tabla, como parámetro pasamos el nombre de la tabla que vamos a crear, y una closure como segundo parámetro. También necesitamos pasar un parámetro a la closure, puedes llamarlo como quieras, pero me gusta usar `$tabla` porque creo que hace que el código se lea mejor.

Dentro de la closure, podemos usar el parámetro `$tabla` para crear nuestros campos con un buen número de métodos bien nombrados. Echémosles un vistazo.

increments()

Añade un ID autoincremental a la tabla. ¡La mayoría de tus tablas lo tendrán!

string()

Crea un campo `VARCHAR`. `string` suena mejor, ¿verdad?

integer()

Añade un campo del tipo entero a la tabla.

float()

Añade un campo del tipo float a la tabla.

boolean()

Añade un campo del tipo booleano a la tabla.

date()

Añade un campo de tipo fecha a la tabla.

timestamp()

Añade un campo de tipo timestamp a la tabla.

timestamps()

Añade los campos `created_at` y `updated_at` a la tabla para indicar cúando se creó y actualizó (respectivamente) el registro.

text()

Añade un campo de tipo texto a la tabla.

blob()

Añade un campo de datos blob a la tabla.

También puedes usar el método encadenable `->nullable()`, para permitir que el campo reciba valores del tipo `NULL`.

Los detalles completos sobre los parámetros disponibles para los métodos arriba detallados, podéis encontrarlos en la documentación oficial[1].

¡Genial, ahora hemos creado nuestra tabla! Teniendo en cuenta que hemos creado la tabla en el método `up()`, ahora tenemos que hacer `DROP` a la tabla en el método `down()`, para que el esquema vuelva a su estado original en caso de marcha atrás. Por fortuna, la clase `Schema` tiene otro método que es perfecto para esta tarea.

[1] http://laravel.com/docs/database/migrations

```
<?php

Schema::drop('usuarios');
```

Sip, no podía ser más fácil. La clase `Schema` también tiene funciones para llevar a cabo otras tareas del esquema, como eliminar columnas `drop_column()`, añadir índices `unique()` y métodos para claves foráneas (foreign key). Creo que cubrirlos todos sería como convertir el libro en un API, y es algo que no me gustaría hacer porque están bien explicados en la documentación oficial[2], y si echas un vistazo allí podremos seguir avanzando en cosas nuevas, como descubrir cómo ejecutar esas migraciones. ¡Vamos a ello!

Antes de que podamos ejecutar nuestras migraciones, tenemos que instalar la tabla `laravel_migrations`, para que Laravel pueda llevar un registro de qué migraciones han sido ejecutadas. Por suerte, artisan tiene un comando que ejecutará el cambio necesario en la base de datos por ti. Vamos a ello:

```
php artisan migrate:install
```

Y el resultado...

```
Migration table created successfully.
```

¡Bien! Ahora que la tabla `laravel_migrations` ha sido creada, podemos ejecutar finalmente nuestra nueva migración. Simplemente usa el nombre de la tarea esta vez. Allá vamos:

```
php artisan migrate
Migrated: application/2012_03_30_220459_crear_usuarios
```

¡Epa! Si miramos nuestra base de datos, la tabla de usuarios habrá sido creada correctamente. Espera, ¡cometimos un error! (no lo hicimos pero me gusta el drama y necesito una excusa para dar marcha atrás). Vamos a hacer marcha atrás y dejemos que el `Schema` se encargue de todos y rehaga los cambios.

```
php artisan migrate:rollback
Rolled back: application/2012_03_30_220459_crear_usuarios
```

Por favor, ten en cuenta que si quieres facilitar datos de ejemplo para rellenar una bae de datos, simplemente creálos con Fluent Query Builder, que lo trataremos en el siguiente capítulo.

¡Facilísimo! ¡Ahora que sabes cómo usar las migraciones, puedes crear tu esquema y rellenar tu base de datos siempre que quieras! En el siguiente capítulo estudiaremos Fluent Query Builder (constructor de consultas Fluent).

[2] http://laravel.com/docs/database/schema

10 Fluent Query Builder

Fluent es otra genial librería que Laravel facilita para ayudarme a esquivar la bala del SQL. Aunque aun puedes escribir consultas SQL 'a pelo' si te gusta el dolor. ¿Que qué es lo mejor de usar Fluent? Aparte de la falta de SQL, es que usa sentencias preparadas, que están completamente protegidas contra ataques de inyección SQL. Fluent es también... bueno, fluido, en varios dialectos SQL por lo que tus métodos funcionarán en una variedad de bases de datos. Antes de que empecemos tendrás que entender el concepto de encadenado. Echa un ojo a este ejemplo.

```
1  <?php
2
3  Class::make()->chain()->chain()->chain()->trigger();
```

Esta es una forma realmente útil de encadenar opciones entre sí, y es una forma adorable de expresar SQL (como verás más tarde). La clase se instancia con el método `make()`, algunas veces también pasarás 'valores de inicialización' a la clase de esta forma. Los métodos `chain()` son usados para modificar la petición con diferentes opciones, y el método `trigger()` es usado para obtener el resultado final. Puede sonar un poco confuso, pero echemos un vistazo a un ejemplo de Fluent.

```
1  <?php
2
3  $usuarios = DB::table('usuarios')->where('usuario', '=', 'dayle')->get();
```

El ejemplo anterior ejecutará un simple...

```
1  SELECT * FROM usuarios WHERE usuario = 'dayle';
```

y nos devolverá una matriz de objetos, representando filas de base de datos que son el resultado de la consulta.

El método `table()` está instanciando el objeto, y estableciendo la tabla con la que queremos trabajar. `where()` es un método encadenado que aplica una cláusula `WHERE` a nuestra consulta, y `get()` es el método final que activa la consulta y obtiene todos los objetos que son resultado de la consulta.

El activador, `get()` devuelve una matriz de resultados por lo que, vamos a empezar a recorrer el conjunto de resultados con un `foreach`.

```php
<?php

foreach ($usuarios as $usuario)
{
        echo $usuario->email;
}
```

Como puedes ver en el ejemplo superior, a los campos de la fila se accede usando el atributo del objeto resultado. El bucle de arriba debería mostrarnos las direcciones de correo de todos nuestros usuarios.

10.1 Obteniendo resultados

Echemos un vistazo a otros de los métodos activadores para obtener resultados.

get()

Acabamos de usarlo. Nos devolverá todos los objetos que son el resultado de la consulta en una matriz.

first()

Este activador devolverá un único objeto de resultado, el primer objeto que cumpla los requisitos de la consulta.

find($id)

Este activador encontrará el elemento por su id de base de datos. Es un atajo útil para `where('id', '=', $id)`. Devolverá un único objeto de resultado.

only($fieldname)

Este traerá el resultado para un único campo que coincida con la consulta.

get(array())

Pasa una matriz de campos al método `get()` para obtener solo esos campos.

10.2 Clausulas WHERE

Ahora que tenemos los métodos que necesitamos para obtener nuestros resultados de la base de datos, ¿cómo podemos aplicar condiciones a esas consultas SQL? Bien, en el ejemplo de arriba me viste usar el método `where()`. Vamos a mirarlo con más atención.

```php
<?php

$usuarios = DB::table('usuarios')->where('usuario', '=', 'dayle')->get();
```

Aquí tenemos el mismo código de nuevo, pero vamos a concentrarnos en la parte `where()` de la cadena:

```php
<?php

where('usuario', '=', 'dayle')
```

Lo genial sobre cómo Laravel gestiona la cláusula WHERE, es que la cadena se parece un poco al SQL que se está generando. En el método encadenado de arriba, estamos diciendo `WHERE usuario = 'dayle'`. El primer parámetro para el método indica el campo que estamos comparando, el segundo parámetro indica el operador a usar en la comparacion y el tercer parámetro es el valor contra el que estamos comparando. También podemos usar:

```php
<?php

where('edad', '>', '18')
// WHERE edad > '18'
// ¡Ya puedes beber!
```

¿Qué pasa si queremos más condiciones? Primero tendremos que decidir si necesitamos 'AND WHERE', o 'OR WHERE'. Para incluir una clausula 'AND' simplemente usa el método `where()` nuevamente en la cadena. Por ejemplo:

```php
<?php

$usuarios = DB::table('usuarios')
                        ->where('usuario', '=', 'dayle')
                        ->where('atractivo', '>', 5000)
                        ->get();
```

Como puedes ver en el ejemplo superior, pongo cada método de la cadena en una nueva línea. Encuentro que es más sencillo de leer y nos evita tener líneas de código terriblemente largas. Esta cadena ejecutará la siguiente consulta SQL:

```sql
SELECT * FROM usuarios WHERE usuario = 'dayle' AND atractivo > 5000;
```

Si preferimos usar una condición 'OR', simplemente usaremos el método `or_where()`, que aceptará los mismos parámetros. Por ejemplo:

```php
<?php

$usuarios = DB::table('usuarios')
                        ->where('usuario', '=', 'dayle')
                        ->or_where('cara', 'LIKE', '%modelomasculino%')
                        ->get();
```

Lo cual nos daría:

```
SELECT * FROM usuarios WHERE usuario = 'dayle' OR cara LIKE '%modelomasculi\
no%';
```

Ahora no necesito explicar cómo funciona cada característica de SQL. Hay multitud de libros para ello, pero voy a nombrar los métodos usados para cumplir tareas comunes:

Usa los métodos `where_in()`, `where_not_in()`, `or_where_in()` y `or_where_not_in()` para comparar un campo con una matriz de elementos.

Los métodos `where_null()`, `where_not_null()`, `or_where_null()`, y `or_where_not_null()` para comparar un campo con un valor `NULL`.

A veces querrás anidar cláusulas `where` entre sí. Laravel provee funcionalidad para esto en forma de 'Cláusulas Where anidadas'. Vamos a ver un código de ejemplo de la documentación oficial:

```
<?php

$usuarios = DB::table('usuarios')
        ->where('id', '=', 1)
        ->or_where(function($query)
        {
                $query->where('edad', '>', 25);
                $query->where('votos' '>', 100);
        })
        ->get();
```

Pasando una closure que contiene cláusulas where adicionales a otro método where, podemos crear cláusulas where anidadas. El resultado se puede ver en SQL:

```
SELECT * FROM "usuarios" WHERE "id" = ? OR ("edad" > ? AND "votos" > ?)
```

Bonito, ¿verdad?

¡Y ahora una característica más interesante! (Ya te conté lo mucho que odio SQL, ¿verdad?) Las clausulas where dinámicas te ofrecen una manera realmente original de definir cláusulas where. Echa un ojo a esto...

```
<?php

where_tamano(5)->get();
```

Aquí estamos especificando un campo a comparar en el nombre del método. Fluent es tan listo que se encargará de todo, ¡sin problemas!

Incluso entiende `AND`y `OR`. ¡Mira esto!

```php
<?php

where_tamano_and_altura(700, 400)->get();
```

Expresivo, limpio. Laravel en su máxima expresión.

10.3 Joins de tablas

Echemos un vistazo a los `JOIN` usando Fluent.

```php
<?php

DB::table('tareas')
        ->join('proyectos', 'tareas.id', '=', 'proyectos.id_tarea')
        ->get(array('tareas.nombre', 'proyectos.nombre'));
```

Pasamos el nombre de la tabla join como primer parámetro, y usamos los otros tres parámetros para realizar una cláusula 'ON' de manera similar a como lo hemos hecho con 'WHERE'.

Luego pasamos los métodos que queremos obtener al método `get()` como una matriz.

Podemos hacer un `left_join()` exactamente de la misma manera. De hecho tiene los mismos parámetros. Fácil, ¿no?

¿Te acuerdas de las cláusulas where anidadas? Bien, puedes usar una sintaxis similar para añadir más condiciones a la clausula `ON` de un `JOIN`. Echemos un ojo:

```php
<?php

DB::table('tareas')
        ->join('proyecto', function($join) {
                $join->on('tareas.id', '=', 'proyecto.id_tarea');
                $join->or_on('tareas.id_autor', '=', 'proyecto.id_autor');
        })
        ->get(array('tareas.name', 'proyecto.name'));
```

En este caso, pasamos una closure como segundo parámetro del método `join()`, luego usamos los métodos `on()`, `or_on()` y `and_on()` para establecer las condiciones.

10.4 Ordenación

El orden es bastante importante. No querrás desperdiciar recursos haciéndolo con PHP, aunque podrías... claro... multitud de horribles `array_sort()`, muchos bucles... No sería divertido. Pasemos esa responsabilidad a Fluent.

```php
<?php

DB::table('zapatos')->order_by('tamano', 'asc')->get();
```

Humm... ¿zapatos? Las mujeres están pensadas para pensar sobre zapatos pero eso es todo lo que me ha venido a la mente... Escalofriante. De cualquier forma... es tan fácil como pasar el nombre de un campo y `asc` para ascendente o `desc` para descendiente. Para ordenar en más de una columna, simplemente repite la cadena `order_by()`.

10.5 Limitando... no, cogiendo

¿Qué pasa si solo queremos un cierto número de resultados? En SQL usaríamos `LIMIT`. Bah, eso suena estúpido. Laravel nos da `take()`.

```php
<?php

DB::table('zapatos')->take(10)->get();
```

¿Ahora quiero 10 zapatos? Debería estar más preocupado. Pero queda claro, ¡limitar es muy sencillo!

10.6 Saltándonos resultados

No necesitamos los 5 primeros pares de zapatos, ¿verdad? Son zapatos de cuero, me gusta vestir zapatos de skate porque tengo los pies muy grandes... Vamos a saltárnoslos.

```php
<?php

DB::table('zapatos')->skip(5)->get();
```

Y ahí lo tienes, ahora podemos saltarnos los 5 primeros resultados con `skip()`. ¡Facilísimo!

10.7 Agregados

A veces es útil ejecutar operaciones matemáticas básicas en las consultas. AVG, MIN, MAX, SUM, y COUNT son usados a menudo para obtener rápidamente el resultado que queremos con SQL y están disponibles en fluente. Echemos un vistazo.

```php
<?php

$val = DB::table('zapatos')->avg('tamano');
$val = DB::table('zapatos')->min('tamano');
$val = DB::table('zapatos')->max('tamano');
$val = DB::table('zapatos')->sum('tamano');
$val = DB::table('zapatos')->count();
```

¡Simple! No olvides que estos métodos son activadores por lo que no necesitamos el método `get()` aquí. ¡También puedes añadir condiciones con `where()` o lo que quieras!

10.8 Expresiones

Los métodos que hemos estado usando hasta ahora escapan y entrecomillan los parámetros que le pasamos, de forma automática. Pero, ¿qué pasa si necesitamos algo realmente personalizado? ¿qué pasa si no queremos añadir nada más? Para eso tenemos el método `DB::raw()`, he aquí un ejemplo:

```php
<?php

DB::table('zapatos')->update(array('vestidos' => DB::raw('NOW()')));
```

En esta consulta, `NOW()` no será escapado ni entrecomillado. Gran libertad aquí, pero no te olvides de la cita de Spiderman, sé responsable con este poder.

10.9 ++ (o decremento)

¿Qué ocurre si únicamente queremos aumentar o reducir un valor? ¡Facilísimo!

```php
<?php

DB::table('zapatos')->increment('tamano');
DB::table('zapatos')->decrement('tamano');
```

Simplemente pasa el nombre de un campo y... ¡voilá!

10.10 Insertar

Finalmente, vamos a guardar algunos datos. Todo este tiempo hemos visto cómo leer datos. ¡Esto será más divertido! Es realmente fácil insertar una nueva fila. Todo lo que tenemos que hacer es facilitar una matriz clave-valor al método `insert()`, ¡que es en sí un disparador!

```php
<?php

DB::table('zapatos')->insert(array(
        'color'        => 'rosa fucsia',
        'tipo'        => 'tacón',
        'tamano'        => '12'
));
```

Espera, vamos a hacernos con el `id` que ha sido creado con esta nueva fila, puede que nos sirva después. Podemos usar `insert_get_id()` con los mismos parámetros para ello.

```php
<?php

$id = DB::table('zapatos')->insert_get_id(array(
        'color'        => 'rosa fucsia',
        'tipo'        => 'tacón',
        'tamano'        => '12'
));
```

Este es mi par del fin de semana, que quede entre nosotros... ahora tenemos un bonito par de tacones de color rosa fucsia, con tamaño 12 en la tabla `zapatos` de nuestra base de datos.

10.11 Actualizar

Espera, ¿dije tacones en la última sesión? Eran botines de color rosa fucsia. Si tan solo pudiera volver y arreglar nuestro error, podría actualizar la tabla y nadie vería mi vergüenza. Oh... podríamos usar probablemente el método `update()`, verás que recibe una matriz con la misma sintaxis que `insert()`.

```php
<?php

DB::table('zapatos')->update(array(
        'tipo'        => 'botines'
));
```

Espera un segundo, no hemos especificado ningún registro aquí. No quiero cambiar todos mis registros a botines, arruinaría mis maravillosas chanclas de Laravel. Usemos el método `where()` con la `$id` que obtuvimos para reducirla al registro que queremos. ¡Hora de la cadena!

```php
<?php

DB::table('zapatos')
        ->where('id', '=', $id)
        ->update(array(
                'tipo'        => 'botines'
        ));
```

Ea, hemos arreglado el problema antes de que nadie se diera cuenta.

10.12 Borrar

Para borrar una fila (por favor, no los zapatos, ¡son inocentes!) podemos usar el método `delete()` con una cláusula `where()`, o simplemente le pasamos un `id` directamente para borrar una única fila. Veamos estos métodos en acción.

```php
<?php

DB::table('no_zapatos')->where('textura', '=', 'peluda')->delete();
```

Simple, ¡los peludos se fueron!

```php
<?php

DB::table('zapatos')->delete($id);
```

¡NO! No los botines rosa fucsia. Obviamente has aprendido mucho, por favor no te olvides del poder y la responsabilidad... no queremos más masacres de zapatos.

En el próximo capítulo, dejaremos atrás los zap... Fluent, y le echaremos un vistazo a Eloquent. Eloquent nos permite tratar nuestras filas de la base de datos como objetos y nos facilita una solución elegante... o elocuente, de gestionar las relaciones.

11 ORM Eloquent

Un ORM es un paquete realmente útil. Son las siglas para Mapeo objeto-relacional (Object Relational Mapper). Suena muy complicado, ¿verdad? Vayamos por partes. La parte de mapeo, es porque estamos mapeando nuestros objetos PHP o clases a tablas y filas de la base da datos. La parte Relacioinal quedará más clara en la sección de relaciones.

Hay muchas soluciones de ORM, pero ninguna tan elocuente como… bueno, Eloquent. Eloquent viene con Laravel y puede ser usado sin modificar nada. Para esta parte, voy a asumir que configuraste tu base da datos, como describimos en el capítulo de `Migraciones`, y que estás familiarizado con los métodos y encadenado del capítulo de `Fluent Query Builder`. Vamos a basarnos en ellos.

Verás que Eloquent es una alternativa a usar Fluent, pero comparte mucho de sus métodos. La única diferencia es que vamos a interactuar con modelos de Eloquent, pero comparten muchos de los métodos, en vez de los `stdObject` que obtenemos de Fluent. Va a hacer que nuestro código parezca incluso más claro. Vamos a sumergirnos directamente y revisemos el modelo de Eloquent.

11.1 Creando y usando modelos de Eloquent

```
1  <?php
2
3  class Usuario extends Eloquent
4  {
5
6  }
```

De hecho, eso es todo… de verdad. Estoy serio. Verás, Eloquent confía en ti y sabe que ya creaste una bonita migración para la tabla `users`. No necesita decirte sobre los campos que existen, confía en que los conoces. Deberías, porque escribiste la migración e hiciste la table. Oh, por cierto… pusiste un `increments('id')` en la tabla ¿verdad? Es una práctica común y Eloquent lo necesita para funcionar.

Otra cosa de la que puedes haberte percatado, es que en el modelo la clase se llama `Usuario` y la tabla se llama `usuarios`. No es un error. ELoquent es tan inteligente, que puede detectar los plurales en Inglés. Nuestro objeto es singular, por lo que lo definimos como `Usuario`, pero nuestra tabla de la base de datos, `usuarios`, contendrá cualquier número de usuarios, por lo que Laravel sabe cómo buscar una tabla con un plural del nombre del objeto.

Puede que te hayas quedado con el hecho de que solo detecta los plurales para el idioma Inglés. Pero, ¿qué pasa si usamos otro idioma o un determinado plural no te funciona? Simplemente añade la palabra, y su versión en plural a la matriz en el archivo `application/config/strings.php`. ¡Ahora funcionará como esperabas!

Quizá no tengas la forma plural del nombre de la tabla- No es problema. Simplemente usa el atributo estático de la clase, `$table` para especificar el nombre de la tabla. Por ejemplo:

```php
<?php

class Usuario extends Eloquent
{
        public static $table = 'usuarios_aplicacion';
}
```

Bueno, escribir una definición de modelo tan grande debe haberte dejado sin aliento (por favor, nota el sarcasmo). Vamos allá y usemos nuestro nuevo modelo.

```php
<?php

$usuario = Usuario::find(1);
```

Ey... ¡espera! Hemos visto esto antes. ¿Te acuerdas del método `find()` de nuestro capítulo de Fluent? Nos devuelve un único resultado por su clave primaria. Verás que Eloquent usa muchos de los métodos usados por Fluent. Es realmente conveniente porque puedes usar una combinación de ambas librerías de Base de Datos sin olvidar qué método es para qué. ¡Increíble!

Para Eloquent, simplemente usa el nombre del objeto y facilita un método estático. No tenemos que usar `DB::table()` para este.

Puedes usar el objeto resultado de forma similar. La verdad es que ahora estás interactuando con un objeto de Eloquent. Pero no notarás la diferencia en este punto. Vamos a obtener el nombre del usuario.

```php
<?php

echo $usuario->nombre;
```

Sip. Es lo mismo que la última vez. Bonito y fácil.

¿Qué pasa si queremos obtener nuestro objeto `User` como matriz? Simplemente llama al método `to_array()` en cualquier método Eloquent para recibir una matriz en vez de un objeto.

```php
<?php

$usuario = $usuario->to_array();
```

Si quieres excluir ciertos campos de esta matriz, puedes añadir un atributo estático `$hidden` a tu modelo Eloquent, con una matriz de todos los campos a exlcuir. Por ejemplo:

```php
<?php

class Usuario extends Eloquent
{
        public static $hidden = array('apodo');
}
```

Digamos que queremos múltiples resultados de vuelta. Podemos usar el método `all()` para obtener todos los usuarios y así podremos obtener una fiesta de usuarios curiosa.

```php
<?php

$usuarios = Usuario::all();
```

Lo que tenemos ahora, es una matriz de objetos de `Usuario`. Podemos iterar sobre ella para acceder a cada uno de forma individual. Realmente simple.

```php
<?php

foreach ($usuarios as $usuario)
{
        invitar_a_fiesta($usuario);
}
```

Bien, ya empezamos a menear el esqueleto. Aunque hay muchos chicos... aumentemos nuestras posibilidades y deshagámonos del resto de chicos.

```php
<?php

$usuarios_sexys = Usuario::where('genero', '=', 'mujer')->get();
```

¡Yeheee, mucho mejor! Hemos mejorado bastante. Veras que acabamos de usar un método `where()` encadenado, y `get()` al igual que hicimos con Fluent. Nada nuevo aquí. Solo claridad aumentada.

Podría cubrir nuevamente todos los métodos de consulta para devolver resultados, pero realmente no me hace falta. Simplemente vuelve un capítulo atrás y cambia mentalmente la palabra Fluent por Eloquent. Pillarás la idea.

En vez de eso, vamos a ver qué ha cambiado. ¡Crear y actualizar objetos (filas) se volvió mucho más fácil!

```php
<?php

$usuario = new Usuario();

$usuario->nombre = 'Capitán Gallumbos';
$usuario->mojo = '100%';

$usuario->save();
```

Mejor no invitar al Capitán Gallumbos a nuestra fiesta. No tendremos oportunidades con él cerca.

Verás que simplemente hemos creado un nuevo objeto `Usuario`, establecido atributos de clase para los campos que existen (no tenemos que establecer un ID, Eloquent se encargará) y guardar nuestro objeto con `save()` para escribir nuestros cambios a la base de datos. Este patrón de diseño es comunmente conocido como registro activo, `Active Record`. Es una forma realmente buena de interactuar con nuestras filas, tratándolas de la misma forma que cualquier otro objeto en nuestra aplicación. ¡Es como si la base de datos y todo ese horrible SQL ni siquiera existieran!

Si ya tenemos una matriz de clave-valor describiendo a nuestro amigo el 'Capitán Gallumbos', podemos pasársela al parámetro del constructor del nuevo objeto, o usar el método de asignación masiva, `fill()` para añadirlo a la base de datos. Mira esto...

```php
<?php

$usuario = new Usuario(array(
        'nombre' => 'Capitán Gallumbos',
        'mojo' => '100%'
));

$usuario->save();
```

o

```php
<?php

$arr = array(
        'nombre' => 'Capitán Gallumbos',
        'mojo' => '100%'
);

$usuario = new Usuario();
$usuario->fill($arr);
$usuario->save();
```

Por favor, ten cuidado con no dejarte ninguna información extra al usar la asignación masiva ya que podría resultar en un riesgo de seguridad potencial.

Y entonces, ¿cómo acutalizamos una fila? Es muy sencillo, solo que primero tenemos que consultar el usuario que queremos cambiar. Veamos...

```php
<?php

$usuario = Usuario::where('nombre', '=', 'Capitán Gallumbos')->first();
```

Aquí usamos el método `first()` para asegurarnos de que solo obtenemos un objeto de vuelta. No podemos cambiar los valores en la matriz, tendríamos que iterar sobre cada uno, ¡y eso haría el nuestro ejemplo fuera más largo de lo que necesitamos!

```php
<?php

$usuario->mojo = '5%';
$usuario->save();
```

Bueno, no podía quitarle todo su mojo. No sería justo. Al menos puede venir a la fiesta ahora. Como ves, la actualización se hace de la misma forma que la inserción, excepto que primero necesitamos encontrar el objeto que queremos actualizar primero.

¿Recuerdas haber usado `$table->timestamps()` en el capítulo de migraciones para crear los campos `updated_at` y `created_at`? Eloquent actualizará estos de manera automática para ti. Insertando una marca de tiempo cuando se crea un objeto, y actualizando el campo `updated_at` cada vez que lo guardas. Si quisieras desactivar esta característica, simplemente añade el atributo de clase `$timestamps` a tu modelo y establécelo a `false`.

```php
<?php

class Usuario extends Eloquent
{
        public static $timestamps = false;
}
```

11.2 Relaciones

Las relaciones son bonitas. No, no me he puesto moña. Quiero decir las relaciones entre tablas. En Eloquent son bonitas. Nada de basura de 'JOIN'. Podemos definir relaciones 'uno-a-uno', 'uno-a-muchos' y 'muchos-a-muchos' simplemente añadiendo algunos métodos sencillos a nuestros modelos de Eloquent. Vamos a meternos de lleno con algo de código.

Uno a uno

Allá vamos...

```php
<?php

class Usuario extends Eloquent
{
        public function invitacion()
        {
                return $this->has_one('Invitacion');
        }
}
```

Vamos a asumir que tenemos una tabla llamada `Invitaciones` y un modelo `Invitacion` para almacenar las invitaciones a nuestra fiesta.

Ahora podemos hacernos con la invitación de un usuario con una sintaxis realmente clara y expresiva, echemos un ojo:

```php
<?php

$invitacion = Usuario::find(1)->invitacion()->first();
```

De nuevo, estamos usando `first()` porque solo queremos un resultado. Verás que añadiendo `->invitacion()->` a la cadena, que es por supuesto el nombre del método de nuestra relacion, nos dará el objeto `Invitacion` que está relacionado con nuestro `usuario`.

La relación ejecutará dos consultas:

```
SELECT * FROM "usuarios" WHERE "id" = 1;
SELECT * FROM "invitaciones" WHERE "usuario_id" = 1;
```

De la consulta, verás que Eloquent busca `user_id` de forma automática como clave foránea. Por lo que si queremos usar nuestra relación, tendremos que crear un campo entero llamado `usuario_id` en nuestra migración, en la tabla `Invitaciones`. ¿Qué pasa si queremos llamar a nuestra clave foránea de otra manera? Sin problemas. Simplemente pasamos un segundo parámetro al método `has_one()` (u otros métodos de relación) para especificar el nuevo nombre del campo. Por ejemplo:

```php
<?php

class Usuario extends Eloquent
{
        public function invite()
        {
                return $this->has_one('Invitacion', 'la_id_de_la_invitacion');
        }
}
```

Pero, ¿qué pasa con la inversa de esta relación? ¿Qué pasa si tengo una Invitación, pero quiero saber a quién pertenece? Aquí es donde entra en juego el método `belongs_to()`. Echemos un vistazo al modelo.

```php
<?php

class Invitacion extends Eloquent
{
        public function usuario()
        {
                return $this->belongs_to('Usuario');
        }
}
```

Una sintaxis similar, pero usando `belongs_to()` para señalar la clave foránea, que existe en esta tabla.

Ahora podemos usar...

```php
<?php

$usuario = Invite::find(1)->usuario()->first();
```

¡Fácil!

Uno a muchos

¿Qué pasa si queremos obtener muchos elementos relacionados? Bueno, como habrás podido deducir hay un método para esto. Vamos a echar un vistazo (Lo digo demasiado, ¿verdad? Podría ser mi muletilla.)

```php
<?php

class Usuario extends Eloquent
{
        public function sombreros()
        {
                return $this->has_many('Sombrero');
        }
}
```

De nuevo, verás que pasamos el nombre del objeto en formato de cadena de texto, capitalizándolo, al método `has_many()`.

En este ejemplo, la tabla `sombreros` necesitará una clave foránea `usuario_id` para poder usar:

```php
<?php

$sombreros = Usuario::find(1)->sombreros()->get();
```

Ten en cuenta que también podemos usar una propiedad dinámica con el mismo nombre, para obtener los mismos resultados con una sintáxis más corta. Por ejemplo:

```php
<?php

$sombreros = Usuario::find(1)->sombreros;
```

¡Mejor estilo!

Como antes, puedes pasar otra clave foránea como segundo parámetro para usarlo. ¡Sigamos adelante!

Muchos a muchos

Aquí las cosas se vuelven un poco más complicadas, pero no te preocupes... Estoy aquí para ayudarte a pillarlo. Agarra mi mano, imagina que tenemos dos modelos de Eloquent, `Usuario` y `Tarea`. Un usuario puede tener muchas tareas, y una tarea puede tener muchos usuarios. Para este tipo de relaciones necesitamos una tercera tabla.

Esta tabla es conocida por muchos nombre: Una tabla pivote, una tabla de referencia, una tabla intermedia o El Pivote Grande.

Es simplemente una tabla con dos valores de entero `usuario_id` y `tarea_id` que enlaza ambas tablas, formando una relación de muchos a muchos.

La tabla se nombra en función de las dos tablas referenciadas en plural, en orden alfabético. Suena un poco complicado pero es simple. Mira esto:

`tareas_usuarios`

Genial. Ahora tenemos nuestra tabla, vamos a formar la relación:

```php
<?php

class Usuario extends Eloquent
{
        public function tareas()
        {
                return $this->has_many_and_belongs_to('Tareas');
        }
}
```

De nuevo, sintaxis similar, con un nombre un poco más largo.

Esta vez, podemos pasar un segundo parámetro opcional para especificar una tabla pivote diferente. Simple.

11.3 Insertando modelos relacionados

Cuando estamos insertando modelos relacionados, podemos establecer las claves foráneas `usuario_id` o `sombrero_id` nosotros mismos, pero no es demasiado bonito. ¿Por qué no le pasamos el objeto?:

```
<?php

$sombrero = Sombrero::find(1);
$usuario = Usuario::find(1);
$usuario->sombreros()->insert($sombrero);
```

Esto pinta mucho mejor. Es como hacer que los objetos se agarran de la mano. Mucho más limpio que lidiar con esas claves foráneas con valor entero directamente.

Con las relaciones `has_many()` puedes pasar una matriz de pareja clave-valor al método `save()` para insertar o actualizar los modelos relacionados. Por ejemplo:

```
<?php

$sombrero = array(
        'nombre'        => 'Dennis',
        'estilo'        => 'Fedora'
);

$usuario = Usuario::find(1);

$usuario->sombreros()->save($sombrero);
```

Limpio :)

Cuando estamos creando un nuevo elemento relacionado con una relación de muchos a muchos, si usamos el método `insert()` Eloquent no solo creará el nuevo objeto sino que también actualizará la tabla pivote con la nueva entrada. Por ejemplo:

```
<?php

$sombrero = array(
        'nombre'        => 'Dennis',
        'estilo'        => 'Fedora'
);

$usuario = Usuario::find(1);

$usuario->sombreros()->insert($sombrero);
```

Pero ¿qué pasa si el objeto ya existe y únicamente necesitamos crear una relación entre ambos? Podemos hacerlo fácilimente con el método `attach()` pasándole la `$id` (o el objeto) del objeto a ser relacionado. Veamos el código.

```php
<?php

$usuario->sombreros()->attach($sombrero_id);
```

¡La tabla pivote habrá sido actualizada para ti!

Lo siguiente es un método que acabo de descubrir. Asumiré que ha llegado con Eloquent en la versión 3.1. Podemos usar el método `sync()` con una matriz de `ids` y una vez que el método se haya ejecutado, solo las `ids` en la matriz quedarán en la tabla pivote. ¡Muy útil! He aquí un fragmento de código.

```php
<?php

$usuario->sombreros()->sync(array(4, 7, 8));
```

11.4 Tablas pivote

¿Qué pasa si queremos acceder a nuestra tabla pivote directamente, en vez de solo las tablas que están relacionadas? Eloquent nos facilita el método `pivot()` para cumplir esa tarea.

```php
<?php

$pivot = $usuario->sombreros()->pivot();
```

Ahor atenemos una matriz de objetos de resultado, como cualquier otra consulta, pero aquellos que relacionan al usuario con los sombreros.

¿Qué ocurre si queremos obtener la fila exacta que ha sido usada para relacionar un objeto? Fácil, con el atributo dinámico `pivot`. La documentación oficial tiene un gran ejemplo sobre esto y voy a robarlo porque soy una persona horrible. No te preocupes, haré que se vea un poco diferente.

```php
<?php

$usuario = Usuario::find(1);

foreach ($usuario->sombreros as $sombrero)
{
        echo $sombrero->pivot->created_at;
}
```

Ahora podemos acceder al campo `created_at` en la tabla pivote, para ver cuándo se hizo nuestro sombrero.

Ya vale, me estoy empezando a cansar de los sombrero. De hecho quiero borrar mis sombreros. ¡Todos! Por suerte conozco mi id de usuario. Es el número 7, el número de la suerte. Vamos a hacerlo ahora mismo, borremos todos mis sombreros.

```php
<?php

Usuario::find(7)->sombreros()->delete();
```

Hecho. Tendremos la cabeza fría a partir de ahora, pero merece la pena que sepas cómo usar el método `delete()`.

11.5 Carga anticipada

Eloquent nos ofrece la opción de 'Carga anticipada' para ayudar a solucionar la incidencia tan discutida de N+1. Si no sabes lo que es, déjame resumírtela usando lo que ya he aprendido. Echa un vistazo al siguiente trozo de código.

```php
<?php

$usuarios = Usuario::all();

foreach ($usuarios as $usuario)
{
        echo $usuario->sombrero->tamano();
}
```

Por cada iteración del bucle, Eloquent tiene que realizar otra consulta SQL para obtener el objeto `Sombrero` de ese usuario y encontrar su tamaño. Esto no supone mucho problema en conjuntos de datos pequeños, pero con cientos de filas podría afectar drásticamente al rendimiento. Con la carga anticipada, le podemos decir a Eloquent que ejecute `SELECT * FROM sombreros;` cuando obtenga el usuario `Usuario` para tener todos los datos que necesitamos de antemano. Esto reduce la cantidad de consultas a solo dos consultas SQL. ¡Mucho mejor!

Veamos cómo le podemos decir a Eloquent que cargue de forma anticipada una relación:

```php
<?php

$usuarios = Usuario::with('Sombrero')->get();
```

Ahora, la relación es cargada de forma anticipada. Si quieres cargar de manera anticipada varias relaciones, simplemente tienes que pasar una matriz al método `with()`. Merece la pena destacar que el método `with()` es estático y como tal debe estar siempre al inicio de la cadena.

Incluso puedes cargar de forma anticipada relaciones anidadas. Asumamos que el objeto `Sombrero` tiene una relación `estanteriasombrero` para permitirnos saber en qué estantería está. Podemos cargar ambas relaciones de forma anticipada de la siguiente forma:

```php
<?php

$usuarios = Usuario::with(array('Sombrero', 'sombrero.estanteriasombrero'))\
->get();
```

Simplemente añade un prefijo con el nombre de la relación, con el objeto anidado.

¿Qué ocurre si solo queremos añadir algunas condiciones a nuestra carga anticipada? Quizá solo queramos ver sombreros azules. ¡Por supuesto podemos hacerlo con Eloquent! Simplemente pasa el nombre de la relación como clave a la matriz, y una closure que contenga las condiciones como parámetro. Lo explicamos mejor con código.

```php
<?php

$usuarios = Usuario::with(array('Sombrero' => function($consulta) {
        $consulta->where('color', '=', 'azul');
}))->get();
```

Simple. ¡Puedes añadir tantas consultas como quieras!

11.6 Setters y Getters

Los setters son métodos útiles que nos permiten dar formato a los nuevos datos de nuestro modelo de cierta forma cuando es asignado. Vamos a echar un vistazo a un trozo de código.

```php
<?php

public function set_password($password)
{
    $this->set_attribute('hashed_password', Hash::make($password));
}
```

Creamos un método con el nombre del setter, con el prefijo `set_`. Pásale una variable que recibirá el valor del campo. Ahora podemos usar `set_attribute()` para establecer un campo con su nuevo valor. Esto nos permite modificar el valor pasado de cualquier forma que queramos. El ejemplo anterior provocará que el campo `hashed_password` sea actualizado con una versión con hash del valor facilitado al llamarlo:

```php
<?php

$usuario->password = "panda_ninja_secreto";
```

¡Muy útil!

Los getters son exactamente lo contrario. Pueden ser usados para ajustar un valor cuando esté siendo leído. Por ejemplo:

```php
<?php

public function get_nombre_panda()
{
    return 'Panda'.$this->get_attribute('nombre');
}
```

Ahora si nuestro nombre es `Dayle`, podemos usar...

```php
<?php

echo $usuario->nombre_panda;
```

Que nos dará 'PandaDayle'.

En el siguiente capítulo cubriremos otra de las increíbles características de Laravel, los eventos (todas son increíbles, ¿lo pillas?).

12 Eventos

Los eventos son una forma de permitir a otras porciones de código extender tus aplicaciones de forma elegante. Algunas aplicaciones se refieren a este tipo de extensiones como 'Hook'. Si eres un 'Gurú PHP' puede que encuentres parecido al patrón de diseño de 'Observador / Observable'.

Los eventos pueden tener cualquier número de 'escuchadores', y ejecutarlos puede devolver un variado número de respuestas. Creo que la mejor forma de aprender será verlo en acción. Vamos a ver con más detalle los Eventos en Laravel.

12.1 Activa un evento

```
1  <?php
2
3  $respuestas = Event::fire('hora.de.cenar');
```

En este ejemplo 'activamos' o 'ejecutamos' el evento `hora.de.cenar` que alerta a todos los que estén pendientes del evento de que es hora de ir a por el papeo. La variable `$respuestas` contiene una matriz de respuestas que ofrezcan aquellos que escuchan el evento. Ahora profundizamos sobre esto.

También podemos obtener la primera respuesta de un evento, en vez de todas. Simplemente llamando al método `first()` en vez de `fire()`. Ahora recibimos un único resultado como este.

```
1  <?php
2
3  $respuesta = Event::first('hora.de.cenar');
```

Merece la pena destacar, que aunque solo estamos recibiendo una respuesta, todos aquellos que estén escuchando serán informados por lo que el evento será activado de la misma forma que si usáramos el método `fire()`, pero todas las respuestas aparte de la primera serán ignoradas.

La tercera opción para disparar un evento es usar el método `until()`. Este método disparará el evento, notificando a cada uno de los que escuchan hasta que se reciba un valor `'NON-NULL'`. El valor `'NON-NULL'` se pasa a la variable `$respuesta`.

```
1  <?php
2
3  $respuesta = Event::until('hora.de.cenar');
```

Ahora que ya sabemos cómo disparar un evento, echemos un vistazo al otro lado del espectro. Tenemos que saber cómo registrarnos nosotros mismos como escuchador. ¡No queremos perdernos la cena!

12.2 Escucha un Evento

Para escuchar un evento, usamos... probablemente lo hayas adivinado. El método `listen()`. Le damos el nombre del evento que estamos buscando como cadena y una closure que se ejecute cuando el evento haya sido activado. El resultado de la closure será pasado de vuelta como respuesta. Echemos un vistazo..

```
1  <?php
2
3  Event::listen('hora.de.cenar', function() {
4          return '¡Gracias por el papeo!';
5  });
```

Si ahora llamamos al `$val = Event::first('hora.de.cenar');` el valor de `$val` será `¡Gracias por el papeo!`. Realmente fácil.

12.3 Eventos con parámetros

Se puede pasar información extra a aquellos que escuchen un evento que haya sido activado, pasando una matriz de valores a la closure del evento. De esta forma.

```
1  <?php
2
3  $respuestas = Event::fire('hora.de.cenar', array(1, 2, 3));
```

Podemos añadir parámetros a la closure en en la closure para 'capturar' esos valores de la siguiente forma:

```
1  <?php
2
3  Event::listen('hora.de.cenar', function($uno, $dos, $tres) {
4          return '¡Gracias por el papeo!';
5  });
```

¡Realmente fácil!

12.4 Eventos de Laravel

Estos son los eventos que pueden ser encontrados dentro del núcleo de Laravel. Normalmente no tendrás que interactura con ellos, pero una vez que te hayas convertido en un 'Gurú de Laravel' puede que sientas la necesidad de moldear el framework a tu voluntad y usar esos eventos es una de las formas de hacerlo.

```php
<?php

Event::listen('laravel.log', function($tipo, $mensaje){});
```

Se ha creado una nueva entrada en el log de Laravel.

```php
<?php

Event::listen('laravel.query', function($sql, $bindings, $time){});
```

Se ha ejecutado una consulta SQL.

```php
<?php

Event::listen('laravel.done', function($respuesta){});
```

El trabajo de Laravel está hecho y la respuesta está preparada para ser enviada.

```php
<?php

Event::listen('404', function(){});
```

No se pudo encontrar una página-

y muchos más.

> ***Nota:*** Lo mejor es añadir un prefijo a tu evento con el nombre de la aplicación para evitar conflictos con otros eventos. Es esto por lo que los eventos de Laravel comienzan con `laravel`.

12.5 Ejemplo de uso

Entonces, ¿dónde pueden ser útiles los eventos? Bueno, imaginemos que tenemos una aplicación de gestión de tareas muy sencilla. Este tipo de aplicaciones obviamente tendrán un objeto del tipo `Usuario` que permitirá a nuestros usuarios iniciar sesión y usar la aplicación. Los objetos `Usuario` también pueden ser creados para añadir más usuarios al sistema. Vamos a añadir un evento al proceso de creación del usuario.

```php
<?php

$nuevo_usuario = array(.. detalles del usuario ..);
$u = new Usuario($nuevo_usuario);
$u->save();

Event::fire('miaplicacion.nuevo_usuario', array($u->id));
```

Como puedes ver, le pasamos la ID del nuevo usuario al evento `miaplicacion.nuevo_usuario` para que todos los que estén escuchando el evento estén al tanto de qué usuario ha sido creado.

Ahora creemos una extensión a nuestra aplicación. La extensión no debería interferir con el código existente de la aplicación. Podría ser una librería o un bundle, cualquier cosa fuera de la aplicación pricnipal.

En nuestra extensión, vamos a añadir un prefijo al nombre del usuario (de alguna forma es útil para nuestra extensión). Hagámoslo escuchando al evento del usuario.

```php
<?php

Event::listen('miaplicacion.nuevo_usuario', function ($uid) {
        $usuario = Usuario::find($uid);
        $usuario->nombre = 'Miaplicacion_'.$user->nombre;
        $usuario->save();
});
```

Comenzamos a escuchar el evento, que recibe la `id` del usaurio. Usándola podemos obtener el nuevo usuario de la base de datos y aplicar los cambios.

Ahora hemos añadido funcionalidad extra a la aplicación, sin editar los archivos del núcleo. Esto es responsabilidad del autor de la aplicación, insertar Eventos en áreas de la aplicación que sean susceptibles de ser extendidas. Sin estos eventos, el código fuente de la aplicación tendría que ser modificado.

13 Plantillas Blade

El motor de plantillas de Laravel, Blade, te permite usar una bonita sintaxis para incluir código PHP en tus vistas. También incluye un número de atajos que permiten el usao de características existentes de Laravel. Las plantillas Blade son cacheadas por defecto, ¡lo cual las hace muy rápidas!

Como siempre, ¡vamos al lío!

13.1 Lo básico

Para activar las plantillas de Blade, tendrás que nombrar tus vistas con la extensión `.blade.php` en vez de `.php`. ¡Tan sencillo como eso!

Cuando usamos los archivos de vista en los frameworks PHP, a menudo te encontrarás haciendo esto...

```
1  <?php echo $val; ?>
```

Activando las etiquetas cortas de PHP podemos recortarlo un poco...

```
1  <?=$val?>
```

... no obstante aun podemos mejorar. Echemos un vistazo a cómo podríamos hacerlo con Blade.

```
1  {{ $val }}
```

¡Precioso! El espacio entre las llaves es opcional, pero creo que se ve mejor con el espacio. El contenido de las dobles llaves es evaluado y volcado. Puedes usar cualquier PHP que quieras dentro. Por ejemplo...

```
1  {{ 5 * time() }}
```

Este código funcionará también. Todo lo que hace Laravel es convertir `{{ $val }}` a `<?php echo $val; ?>`. ¡Asegúrate de considerar esto si te metes en problemas con Blade!

13.2 Lógica

¿Qué pasa con los bucles `foreach()`?, ¡uso montones de ellos! También uso muchos `if` y `else`. Blade simplifica todas estas sentencias condicionales con el uso del mágico signo `@`. Echemos un vistazo.

```
<?php

@foreach ($usuarios as $usuario)
        <div class="usuario">{{ $usuario->nombre }}</div>
@endforeach
```

No solo nos deshacemos de todas esas feas etiquetas de PHP, ¡sino que son mucho más rápidas de escribir! ¿Qué hay sobre los `if`s y `elseif`s? Bueno, si estás acostumbrado a la sintaxis alternativa de PHP, podrás averiguar el resultado. Simplemente reemplaza el `<?php` con `@` y sáltate el ' : ?>'. Lo que nos queda es...

```
@if ($usuario->nombre == 'Dave')
        <p>Â¡Bienvenido Dave!</p>
@else
        <p>Â¡Bienvenido invitado!</p>
@endif
```

Muy simple. He aquí otros operadores que puedes usar con Blade. Deben serte familiares.

```
<?php

@for ($i =0; $i < 100 - 1; $i++)
    Número {{ $i }}<br />
@endfor
```

y

```
<?php

@forelse ($usuarios as $usuario)
    {{ $user->nombre }}
@empty
    <p>No hay usuarios.</p>
@endforelse
```

El último es un poco especial. Es un bucle `foreach()` pero con un `@empty` adicional que nos mostrará el resultado de abajo si la matriz facilitada está vacía. Muy útil y nos evita una sentencia `if` adicional.

13.3 Distribuciones de Blade

Blade nos ofrece otro método de escribir distribuciones complejas o anidadas. Usando su limpia sintaxis, bien podría ser la mejor implementación de una distribución disponible en el framework. Simplemente tienes que usarla para comprender su belleza. Echemos un ojo a nuestra plantilla principal, que es una vista de blade (en este caso nombrada `template.blade.php`) como cualquier otra.

```
<!DOCTYPE HTML>
<html lang="en-GB">
<head>
        <meta charset="UTF-8">
        <title>@yield('titulo')</title>
</head>
<body>
        <div class="header">
                <ul>
                @section('navegacion')
                        <li><a href="#">Inicio</a></li>
                        <li><a href="#">Blog</a></li>
                @yield_section
                </ul>
        </div>

        @yield('contenido')

</body>
</html>
```

Nuestra plantilla principal usa el método `@yield()` para definir una región de contenido que puede ser rellenada con una vista que use esta plantilla. Simplemente pasa una cadena al método para facilitar un apodo para esa región de contenido que será usada para identificarla más tarde.

`@section()` y `@yield_section` definen una región de contenido que contienen datos por defecto pero que pueden ser reemplazados posteriormente. Echemos un ojo a una vista (`page.blade.php`) que hace uso de la plantilla que acabamos de crear.

```
@layout('template')

@section('titulo')
Â¡PÃ¡gina de Dayle!
@endsection

@section('navegacion')
        @parent
        <li><a href="#">Sobre mi</a></li>
@endsection

@section('contenido')
        <h1>Â¡Bienvenido!</h1>
        <p>Â¡Bienvenido a la pÃ¡gina de Dayle!</p>
@endsection
```

En esta vista usamos el método `@layout()` para especificar que queremos usar una vista llamada `template` como plantilla. Ahora podemos usar `@section()` y `@endsection` para reemplazar el

contenido de regiones de contenido, con el código que se encuentre entre ambos métodos.

En el caso de la sección de `navegacion`, descubrirás que hay una etiqueta `@parent` dentro del area de contenido. Blade la reemplazará con el contenido de la plantilla básica.

Si devolvemos..

```
return View::make('page');
```

Desde nuestra ruta/acción, ahora podemos ver nuestra página dentro de la plantilla `layout`. Algo como esto...

```
<!DOCTYPE HTML>
<html lang="en-GB">
<head>
        <meta charset="UTF-8">
        <title>Â¡PÃ¡gina de Dayle!</title>
</head>
<body>
        <div class="header">
                <ul>
                        <li><a href="#">Inicio</a></li>
                        <li><a href="#">Blog</a></li>
                        <li><a href="#">Sobre mi</a></li>
                </ul>
        </div>

        <h1>Â¡Bienvenido!</h1>
        <p>Â¡Bienvenido a la pÃ¡gina web de Dayle!</p>

</body>
</html>
```

¡Genial! También podemos usar cuantas plantillas como queramos. ¡Son vistas normales de Blade!

¿Qué pasa si queremos facilitar contenidos a una `@section` desde nuestra Ruta/Acción? Simplemente llama al método `Section::inject()` con el nombre de la sección, y una cadena representando el contenido de la sección para que sea inyectado en la vista.

```
Route::get('/', array('do' => function()
{
        Section::inject('title', 'Mi sitio');

        return View::make('page');
}));
```

¡Y eso es todo! Ahora puedes usar Blade para hacer que tus vistas se vean realmente limpias y eficientes. Tu diseñador te amará por ello.

14 Autentificación

Muchas aplicaciones querrán una capa de autentificación. Si estás escribiendo un blog, no querrás que tus lectores puedan publicar nuevos artículos. O si estás trabajando con datos sensibles, no querrás que los usuarios sin autorización tengan acceso a estos.

Por suerte, Laravel tiene una clase de autentificación simple, segura y altamente personalizable. Vamos a echar un ojo a cómo podemos interactuar con ella.

14.1 Configuración

Antes de que empecemos, tendrás que crear una nueva tabla para guardar los detalles de nuestros usuarios. Podemos nombrar esta tabla como queramos, pero si la nombramos `users` no tendremos que cambiar el archivo de configuración. He aquí como crear una tabla adecuada con el constructor de esquema.

```
1  <?php
2
3  Schema::create('users', function($table) {
4          $table->increments('id');
5          $table->string('username', 128);
6          $table->string('password', 64);
7  });
```

Puedes añadir tantos campos adicionales como quieras, pero esto nos permitirá avanzar. Vamos a crear un usuario de ejemplo que podemos usar para probar el proceso de autentificación. Primero debería explicar cómo funciona la clase `Hash`.

Puedes usar la clase `Hash` para añadir un hash a una contraseña usando el algoritmo `bcrypt` que es altamente seguro. Es muy sencillo de usar, he aquí un ejemplo.

```
1  <?php
2
3  $pass = Hash::make('mi_cadena_de_contraseña');
```

En el ejemplo superior, creamos un hash de bcrypt de nuestra contraseña. ALmacenando el hash en la base de datos en vez de la contraseña en texto plano ofrece a los usuarios algo de seguridad adicional. Descubrirás que esta es una práctica habitual con las aplicaciones web.

Si quisieras comparar una contraseña hasheada con un un valor, usa el método `check()`. Por ejemplo:

```php
<?php

Hash::check('mi_contraseña', $pass);
```

Esto devolverá un valor booleano, `true` si coinciden y `false` en caso contrario. Ahora que sabemos como hashear una contraseña, podemos crear nuestro usuario de ejemplo. Voy a llamarlo Dexter, porque estoy viendo la serie de televisión de Dexter mientras escribo este capítulo. Es genial escribir con algo de ruido de fondo. ¡Pruébalo mientras programas! ¡Funciona!

Sigamos con Dexter...

```php
<?php

DB::table('users')->insert(array(
        'username'        => 'Dexter',
        'password'        => Hash::make('cuchillo')
));
```

Ahora debemos elegir qué driver de autentificación queremos usar. Tenemos que elegir entre 'eloquent' o 'fluent'.

El driver de 'Fluent' usará Fluent para interactuar con la base de datos y devolverá un objeto que representa la fila de la tabla de usuarios al llamar a `Auth::user()`. El driver `eloquent` devolverá un modelo de Eloquent representando al usuario.

La configuración para el driver de autentificación, tabla o nombre de objeto y nombres de campos se pueden encontrar en 'application/config/auth.php'.

```php
<?php

return array(

        'driver' => 'eloquent',

        'username' => 'email',

        'model' => 'User',

        'table' => 'users',
);
```

Cambiemos el 'driver' a `fluent` para usar el constructor de consultas de Fluent como driver de autentificación y cambiemos el elemento 'username' a `username` para que nuestros usuarios puedan iniciar sesión en la aplicación usando sus usuarios en vez de una dirección de correo.

```
<?php

return array(

        'driver' => 'fluent',

        'username' => 'username',

        'model' => 'User',

        'table' => 'users',
);
```

14.2 Configurando el formulario

¡Derechos a ello! Vamos a necesitar un formulario de inicio de sesión. Hagamos uno bonito con Blade. Nos encanta Blade, ¿verdad? Hagámoslo creando `login.blade.php`.

```
{{ Form::open('login') }}

        <!-- Revisemos si tenemos errores de login -->
        @if (Session::has('login_errors'))
                <span class="error">Usuario o contraseÃ±a incorrectos.</span>
        @endif

        <!-- Campo de usuario -->
        <p>{{ Form::label('username', 'Usuario') }}</p>
        <p>{{ Form::text('username') }}</p>

        <!-- Campo de contraseÃ±a -->
        <p>{{ Form::label('password', 'ContraseÃ±a') }}</p>
        <p>{{ Form::text('password') }}</p>

        <!-- BotÃ³n de envÃ-o -->
        <p>{{ Form::submit('Login') }}</p>

{{ Form::close() }}
```

> Es precio...

Ssssh, eso fue hace unos pocos capítulos. No voy a intentar lavarte el cerebro, puedes dejarlo. ¡Aunque es un bonito formulario! Vamos a crear una bonita ruta para mostrar el formulario.

```php
<?php

Route::get('login', function() {
        return View::make('login');
});
```

Hagamos una variable `POST` de esta ruta. Podemos usarla para cuando enviemos el formulario, de esta forma podemos usar una única URL para ambas.

```php
<?php

Route::post('login', function() {
        return 'Formulario de login enviado';
});
```

Vamos a asegurarnos de que nuestras rutas funcionan correctamente. De hecho estoy bastante seguro de que lo harán, pero es solo una manía que tengo. Si visitamos 'http://localhost/login', introducimos algunos datos y enviamos el formulario, deberíamos obtener `Formulario de login enviado`.

14.3 Gestionando el inicio de sesión

Ten en cuenta de que antes de que podamos iniciar sesión, tendremos que configurar un driver de sesión. Esto es porque los logins de Laravel son guardados en sesión. Si te vas a `application/config/session.php` listará todas tus opciones. Escoge la que te convenga.

¡Genial! Vamos a ver cómo gestionamos ese intento de inicio de sesión y trabajemos en esa ruta `POST`. Primero vamos a obtener los datos que han sido enviados a través del formulario.

```php
<?php

Route::post('login', function() {

        // Obtener datos POST
        $userdata = array(
                'username'              => Input::get('username'),
                'password'               => Input::get('password')
        );

});
```

¡Genial! Tenemos los datos, vamos a usarlos. Usaremos el método `Auth::attempt()` para revisar si el usuario y la contraseña se pueden usar para iniciar sesión. Lo genial de este método es que creará automáticamente la sesión 'logged-in' para nosotros! Très utile! (Nunca aprendí Francés, lo siento.).

```php
<?php

Route::post('login', function() {

        // Obtener datos POST
        $userdata = array(
                'username'                => Input::get('username'),
                'password'                 => Input::get('password')
        );

        if ( Auth::attempt($userdata) )
        {
                // Ya hemos iniciado sesión, vamos al inicio
                return Redirect::to('home');
        }
        else
        {
                // error de autentificación! Volvamos al login!
                return Redirect::to('login')
                        ->with('login_errors', true);
                // Pasa las notificaciones de error que quieras
                // me gusta hacerlo así
        }

});
```

Ahora, si nuestra autentificación tuvo éxito, se creó una sesión de login y nos redirigirá a la ruta `home`. Perfecto, pero antes de que profundicemos más, creemos una ruta `logout` para que podamos cerrar la sesión para continuar con las pruebas.

```php
<?php
```

Route::get(‘logout’, function() {

```php
        Auth::logout();
        return Redirect::to('login');
});
```

Aquí usamos el método `Auth::logout()` para destruir la sesión de login, y volvemos a la página de login. Ahora hemos cerrado la sesión.

Perfecto, ahora que tenemos funcionando nuestro inicio de sesión a la perfección, vamos a crear nuestra página de inicio super secreta. Añadamos un enlace de cierre de sesión y demos la bienvenida a nuestro usuario.

```php
<?php

//---- LA RUTA ----

Route::get('home', function() {
        return View::make('home');
});

//---- LA VISTA (home.blade.php) ----

<div class="header">
        Bienvenido de nuevo, {{ Auth::user()->username }}!<br />
        {{ HTML::link('logout', 'Logout') }}
</div>

<div class="content">
        <h1>Información de las Ardillas</h1>
        <p>Esta es nuestra super página de información de las ardillas rojas.</p>
        <p>Ten cuidado, las ardillas grises están vigilando.</p>
</div>
```

Ahora podemos probar nuestro bucle de inicio de sesión. Inicia sesión, recibe la bienvenida, cierra sesión. Repite. Hazlo al menos 800 veces para dejar a tu mente descansar. No te preocupes, es algo normal. Además, te pagan por horas, ¿verdad gurú PHP?

Te habrás dado cuenta de que en el ejemplo superior usamos `Auth::user()`. Esto nos devolverá un objeto de resultado representando el usuario que ha iniciado sesión. Muy útil para encontrar su `id`, o mostrar información de bienvenida.

14.4 Protegiendo rutas

Vale, ¡ya casi hemos acabado! Hay un pequeño bug, un fallo de seguridad del que tenemos que ocuparnos primero. Cierra la sesisón, (no, no del ordenador en el que estés leyendo esto, solo de la web) y ve a la URL de `home` a mano.

Oh oh... Las ardillas grises pueden ver nuestros planes de ataques, ¡sin haber iniciado sesión siquiera! Tenemos que arreglarlo. Además, debido a que no tenemos un usuario que haya iniciado sesión en el sistema, obtenemos un error de `undefined index` (o algo similar) cuando intentamos acceder al nombre del usuario.

Vuelve atrás. La solución está ahí, acechando en las sombras cerca del principio del libro. ¿Qué? ¿Nada de sombras? ¡Pagué al editor para que las pusiera! ... olvídate y sigamos. ¿Te acuerdas de los filtros de ruta? Usando los filtros podemos ejecutar una porción de código antes de que ejecutemos la ruta, y si devolvemos algo será usado en vez de lo que devolvamos en nuestra ruta. ¡Qué de potencial!

Es más fácil que eso. Taylor tiene un master en lectura de mentes amateur. El sabía que estaríamos escribiendo este capítulo y sabía que tendríamos que proteger una ruta de usuarios que no hayan

iniciado sesión. Eso es por lo que creó el filtro 'auth', que está incluido en Laravel como estándar. Echemos un vistazo al filtro en sí que puedes encontrarlo en `routes.php`.

```
<?php

Route::filter('auth', function()
{
        if (Auth::guest()) return Redirect::to('login');
});
```

¡Precioso!

¿Ves el método `Auth::guest()`? Es un método expresivo que devuelve `true` solo si la petición actual no tiene a un usuario que haya iniciado sesión. ¡Muy útil! También puedes usar `Auth::check()` para realizar la comprobación contraria, para ver si un usuario ha iniciado sesión. Sabemos que esos métodos hacen exactamente lo mismo, pero facilitando nombres de métodos limpios y expresivos, usar el correcto parecerá mucho más limpio en tu código fuente.

Como puedes ver, si no hay un usuario logado, el filtro `auth` redirige a la página de inicio de sesión, sobreescribiendo la vista facilitada a nuestra ruta. Todo lo que tenemos que hacer es asociarla a nuestra ruta `home`.

```
<?php

Route::get('home', array('before' => 'auth', 'do' => function() {
        return View::make('home');
}));
```

Ahí lo tienes. Ahora la ruta `home` está protegida. Los errores de `undefined` no se verán y las ardill... los usuarios sin autorización no serán capaces de ver la página de inicio. Recuerda no aplicar el filtro `auth` a la URI de `login` ya que de lo contrario ¡te enfrentarás a un bucle infinito!

14.5 Personalización

Sé lo que estás pensando. ¿Qué pasa si no quiero usar Eloquent o Fluent? ¡Parece muy limitado!

Por favor, esto es Laravel. ¡Ya deberías haber aprendido! Laravel te permite crear clases personalizadas conocidas como 'Drives de Autorización' para que puedas modificar los parámetros o engancharte a cualquier sistema de autentificación que prefieras. Simplemente crea una nueva clase. Me gusta poner las mías en `application/libraries`, ¡para que sean auto-cargadas para mi!

```php
<?php

// application/libraries/myauth.php
class Myauth extends Laravel\Auth\Drivers\Driver {

        public function attempt($arguments = array())
        {

        }

        public function retrieve($id)
        {

        }

}
```

Tu driver de autentificación debe extender `Laravel\Auth\Drivers\Driver` y contener los dos métodos arriba listados. El primer método acepta la matriz de `username` y `password` y es usada para identificarte usando tu propio método. En una autentificación con éxito deberías hacer una llamada al método `login()` del padre para informar a Laravel de que la identificación tuvo éxito. Por ejemplo:

```php
<?php

public function attempt($argumentos = array())
{
        $usuario = $argumentos['username'];
        $contrasena = $argumentos['password'];

        $resultado = mi_metodo($usuario, $contrasena);

        if($resultado)
        {
                return $this->login($resultado->id, array_get($argumentos, 'remember'));
        }

        return false;
}
```

El método de login acepta un identificador (que puede ser usado más tarde para obtener el usuario) y el valor de la clave `remember` de nuestra matriz de argumentos. Si hay un error de autentificación, el método debería devolver siempre `false`.

El método `retrieve()` recibe el identificador que previamente le pasaste al método `login()`. Puedes usar esto para devolver un objeto que represente al usuario actual. Por ejemplo:

```php
<?php

public function retrieve($id)
{
        return obtener_mi_objeto_de_usuario($id);
}
```

¡Genial! Ahora tenemos un driver de autentificación funcionando. Antes de que podamos usarlo necesitamos registrarlo en la clase `Auth`. Añade el siguiente código a tu `application/start.php`.

```php
<?php

Auth::extend('myauth', function() {
        return new Myauth();
});
```

Pasa un identificador de driver como primer parámetro a `Auth::extend()`. El segundo parámetro es una closure que es usada para devolver una nueva instancia de la clase.

Todo lo que nos queda es actualizar tu archivo `application/config/auth.php` para que apunte al nuevo driver:

```php
<?php

'driver' => 'myauth',
```

¡Y ahora disfruta de tu método de autentificación como siempre!

15 El tutorial del Blog

Finalmente ha llegado el momento de escribir nuestra primera aplicación completa. Si no has leído los otros capítulos, puede que encuentres esto un poco difícil. Vamos a ver lo que vamos a usar para crear el blog.

- Rutas
- Migraciones
- Eloquent
- Plantillas de Blade
- Autentificación
- Filtros

Veamos el plan y organicemos lo que vamos a crear.

15.1 El diseño

Vamos a crear un blog de tres páginas. La primera página consistirá en un listado de posts, similar a la primera página de un blog de Wordpre… Wordpush. La siguiente página será la página a `vista-completa` a la que llegas haciendo click a un artículo que quieres leer. La página final es aquella en la que escribimos nuestros artículos. Normalmente, tendrías la habilidad de editar o borrar artículos, pero podemos añadir todo eso más tarde. No estoy contando la página habitual de login, ya que es usada en cualquier aplicación que implemente autentificación.

Pensemos en el esquema de la base de datos un momento. Sabemos que vamos a necesitar una tabla para almacenar nuestros artículos. También tendremos una tabla para almacenar nuestros usuarios, y vamos a necesitar alguna forma de relación entre ambos. Vamos a pensar en los campos:

`Tabla : posts`

Campo	Tipo
id	INTEGER
title	VARCHAR(128)
body	TEXT
author_id	INTEGER
created_at	DATETIME
updated_at	DATETIME

`Tabla : users`

Campo	Tipo
id	INTEGER
username	VARCHAR(128)
nickname	VARCHAR(128)

password	VARCHAR(64)
created_at	DATETIME
updated_at	DATETIME

Pinta bien. Podemos usar el campo `author_id` en el objeto `Post` para hacer referencia al objeto `User`que representa el autor de un artículo.

Bien, ¡a trabajar!

15.2 Configuración básica

Antes que nada, tendrás que configurar tu base de datos y el driver de sesión. Deberías saber hacerlo en este punto del libro. Usaré MySQL como base de datos como siempre.

Ahora creemos migraciones para nuestras tablas usando Artisan como hicimos en el capítulo de migraciones.

```
1  php artisan migrate:make create_users
```

Nuestro esquema para `up()`con la cuenta de usuario por defecto:

```
1  <?php
2
3  Schema::create('users', function($table) {
4          $table->increments('id');
5          $table->string('username', 128);
6          $table->string('nickname', 128);
7          $table->string('password', 64);
8          $table->timestamps();
9  });
10
11 DB::table('users')->insert(array(
12         'username'        => 'admin',
13         'nickname'        => 'Admin',
14         'password'        => Hash::make('password')
15 ));
```

y actualicemos nuestro método `down()` por si acaso.

```
1  <?php
2
3  Schema::drop('users');
```

Vamos a crear nuestra migración para `posts`. Apuesto q eu no te lo imaginabas, ¿verdad?

```
php artisan migrate:make create_posts
```

El esquema para `up()`:

```
<?php

Schema::create('posts', function($table) {
        $table->increments('id');
        $table->string('title', 128);
        $table->text('body');
        $table->integer('author_id');
        $table->timestamps();
});
```

y carguémonos esta tabla en `down()` también.

```
<?php

Schema::drop('posts');
```

Ejecutemos nuestras migraciones y dejemos las tablas configuradas>

```
php artisan migrate:install
php artisan migrate
```

Finalmente, vamos a actualizar nuestra configuraci'on de autentificaci'on en `application/config/auth.php` para usar `fluent` como driver de autentificación y `username` como identificador de inicio de sesión.

```
<?php

return array(

        'driver' => 'fluent',
        'username' => 'username',
        'model' => 'User',
        'table' => 'users',
);
```

Ahora ya estamos preparados para crear nuestros modelos de Eloquent.

15.3 Modelos Eloquent

Bueno, ya sabes cómo crear modelos ELoquent, por lo que echémosle un vistazo al código:

```php
<?php

class User extends Eloquent
{
        public function posts()
        {
                return $this->has_many('Post');
        }
}

class Post extends Eloquent
{
        public function author()
        {
                return $this->belongs_to('User', 'author_id');
        }
}
```

Bonito y simple. Tenemos nuestro objeto `User` que 'tiene muchos' (`has_many`) artículos y nuestro objeto `Post` que 'pertenece a' (`belongs_to`) un usuario.

15.4 Rutas

Tenemos nuestras tablas. Tenemos modelos y teniendo en cuenta que estamos siguiendo mi flujo de trabajo, vamos a crear las rutas que vamos a necesitar.

```php
<?php

Route::get('/', function() {
        // Nuestra lista de artículos
});

Route::get('view/(:num)', function($post) {
        // Esta es nuestra vista de artículo
});

Route::get('admin', function() {
        // Muestra el formulario de creación de nuevo artículo
});

Route::post('admin', function() {
        // Gestiona el formulario de creación de un nuevo artículo
});

```

```
Route::get('login', function() {
        // Muestra el formulario de login
});

Route::post('login', function() {
        // Gestiona el formulario de login
});

Route::get('logout', function() {
        // Cierra la sesión en el sistema
});
```

Bien, ya las tenemos. Como puedes ver estoy usando GET/POST para mostrar y gestionar formularios con la misma URI.

15.5 Vistas

Comencemos creando una nueva distribución de Blade como base para nuestra aplicación.

`templates/main.blade.php`

```
<!DOCTYPE HTML>
<html lang="en-GB">
<head>
        <meta charset="UTF-8">
        <title>Wordpush</title>
        {{ HTML::style('css/style.css') }}
</head>
<body>
        <div class="header">
                <h1>Wordpush</h1>
                <h2>Code is Limmericks</h2>
        </div>

        <div class="content">
                @yield('content')
        </div>
</body>
</html>
```

Como puedes ver, tenemos una plantilla muy sencilla en HTML5, con una hoja de estilos CSS (que no vamos a cubrir, ya que sirve para hacer tu blog bonito, pero este no es un libro de diseño) y un área de contenido para el contenido de nuestra página.

Vamos a necesitar un formulario de login, para que los autores de nuestros artículos puedan crear nuevas entradas. Voy a robar esta vista del tutorial anterior y personalizarla un poco con la

distribución de blade. De hecho esta es una buena práctica, re-usa todo lo que puedas y te harás con tu propio kit de desarrollo de Laravel.

pages/login.blade.php

```
@layout('templates.main')

@section('content')
        {{ Form::open('login') }}

                <!-- check for login errors flash var -->
                @if (Session::has('login_errors'))
                        <span class="error">Username or password incorrect.</span>
                @endif

                <!-- username field -->
                <p>{{ Form::label('username', 'Username') }}</p>
                <p>{{ Form::text('username') }}</p>

                <!-- password field -->
                <p>{{ Form::label('password', 'Password') }}</p>
                <p>{{ Form::password('password') }}</p>

                <!-- submit button -->
                <p>{{ Form::submit('Login') }}</p>

        {{ Form::close() }}
@endsection
```

Como puedes ver, nuestro formulario de login está usando nuestra distribución recién creada. Vamos a crear el formulario de 'Crear nuevo artículo'.

pages/new.blade.php

```
<?php

@layout('templates.main')

@section('content')
        {{ Form::open('admin') }}

                <!-- title field -->
                <p>{{ Form::label('title', 'Title') }}</p>
                {{ $errors->first('title', '<p class="error">:message</p>') }}
                <p>{{ Form::text('title', Input::old('title')) }}</p>

                <!-- body field -->
```

```
14                  <p>{{ Form::label('body', 'Body') }}</p>
15                  {{ $errors->first('body', '<p class="error">:message</p>') }}
16                  <p>{{ Form::textarea('body', Input::old('body')) }}</p>
17
18                  <!-- submit button -->
19                  <p>{{ Form::submit('Create') }}</p>
20
21          {{ Form::close() }}
22  @endsection
```

15.6 A programar

Finalmente ha llegado la hora de ponernos manos a la obra. Comencemos con la rutina de autentificación. Primero tenemos que enlazar nuestra ruta de login a la vista con el formulario.

```
1  <?php
2
3  Route::get('login', function() {
4          return View::make('pages.login');
5  });
```

No fue tan difícil. Ahora vamos a gestionar la autentificación en la ruta `POST`, de la forma habitual.

```
1  <?php
2
3  Route::post('login', function() {
4
5          $userdata = array(
6                  'username' => Input::get('username'),
7                  'password' => Input::get('password')
8          );
9
10         if ( Auth::attempt($userdata) )
11         {
12                 return Redirect::to('admin');
13         }
14         else
15         {
16                 return Redirect::to('login')
17                         ->with('login_errors', true);
18         }
19 });
```

Ahora podemos iniciar sesión en nuestro sistema. Si tienes algún problema entendiendo estos temas, dirígete a los capítulos anteriores ya que no estamos cubriendo nada nuevo aquí.

Vamos a crear la ruta de cierre de sesión para que podamos probar el proceso de inicio de sesión al completo.

```
<?php

Route::get('logout', function() {
        Auth::logout();
        return Redirect::to('/');
});
```

Vamos a añadir una pequeña sección de perfil a la cabecera de nuestra plantilla principal. Podemos usarla para iniciar o cerrar sesión en el sistema.

```
<div class="header">
        @if ( Auth::guest() )
                {{ HTML::link('admin', 'Login') }}
        @else
                {{ HTML::link('logout', 'Logout') }}
        @endif
        <hr />

        <h1>Wordpush</h1>
        <h2>Code is Limmericks</h2>
</div>
```

Ahora tenemos que añadir el filtro `auth` a ambas rutas `admin`. Te habrás dado cuenta de que `admin` es la única ruta que necesita protección, teniendo en cuenta que queremos que la gente pueda navegar por el blog pero no escribir nada.

```
<?php

Route::get('admin', array('before' => 'auth', 'do' => function() {
        // muestra el formulario de creación de artículo
}));

Route::post('admin', array('before' => 'auth', 'do' => function() {
        // gestiona el formulario de creación de artículo
}));
```

Vamos a asociar también el crear nuevo artículo a la ruta `GET` de `admin` mientras estamos aquí. También deberíamos enviar al usuario que haya iniciado sesión a esta vista. De esta forma podemos usar la `id` de los objetos `User` para identificar al autor.

```php
<?php

Route::get('admin', array('before' => 'auth', 'do' => function() {
        $user = Auth::user();
        return View::make('pages.new')->with('user', $user);
}));
```

Ahora que ya tenemos controlado al usuario que ha iniciado sesión, añadamos esta información a la vista para 'crear nuevo artículo' para identificar nuestro autor.

```php
<?php

...

{{ Form::open('login') }}

        <!-- author -->
        {{ Form::hidden('author_id', $user->id) }}

        <!-- title field -->
        <p>{{ Form::label('title', 'Title') }}</p>
        <p>{{ Form::text('title') }}</p>

...
```

Ahora podemos identificar a nuestro autor y nuestra página para 'crear nuevo artículo' está preparada. No necesitamos enlazarlo, lo queremos oculto. Simplemente podemos escribir la URL `/admin` si queremos crear un nuevo artículo.

Vamos a gestionar la creación de nuevos artículos.

```php
<?php

Route::post('admin', function() {

        // Obtengamos el nuevo artículo de los datos POST
        // mucho más seguro que usar asignación masiva
        $new_post = array(
                'title'         => Input::get('title'),
                'body'                  => Input::get('body'),
                'author_id' => Input::get('author_id')
        );

        // Configuremos algunas reglas para nuestros datos
        // Estoy seguro de que puedes conseguir alguna mejor
        $rules = array(
```

```
                'title'         => 'required|min:3|max:128',
                'body'                  => 'required'
        );

        // Hagamos la validación
        $v = Validator::make($new_post, $rules);

        if ( $v->fails() )
        {
                // redirigimos al formulario con los errores
                // entrada de datos y el usuario que
                // ha iniciado sesión actualmente
                return Redirect::to('admin')
                                ->with('user', Auth::user())
                                ->with_errors($v)
                                ->with_input();
        }

        // Crea el nuevo artículo
        $post = new Post($new_post);
        $post->save();

        // Redirigimos a la vista de nuestro artículo
        return Redirect::to('view/'.$post->id);

});
```

Ahora deberíamos poder crear algunos artículos en el blog. ¡Ánimo! Escribe algunos artículos para que tengamos algo que ver en nuestra vista de 'lista de todos los artículos'.

Hablando de eso, vamos a trabajar sobre esa vista.

```
<?php

Route::get('/', function() {
        // obtengamos nuestros artículos y carguemos
        // de antemano el autor
        $posts = Post::with('author')->all();

        return View::make('pages.home')
                ->with('posts', $posts);
});
```

También necesitamos una vista para listar todos nuestros artículos. Allá vamos.

`pages/home.blade.php`

```
<?php

@layout('templates.main')

@section('content')
        @foreach ($posts as $post)
                <div class="post">
                        <h1>{{ HTML::link('view/'.$post->id, $post->title) }}</h1>
                        <p>{{ substr($post->body,0, 120).' [..]' }}</p>
                        <p>{{ HTML::link('view/'.$post->id, 'Read more &rarr;') }}</p>
                </div>
        @endforeach
@endsection
```

Finalmente, necesitamos una vista completa para nuestros artículos del blog. Vamos a llamarla...

pages/full.blade.php

```
<?php

@layout('templates.main')

@section('content')
        <div class="post">
                <h1>{{ HTML::link('view/'.$post->id, $post->title) }}</h1>
                <p>{{ $post->body }}</p>
                <p>{{ HTML::link('/', '&larr; Back to index.') }}</p>
        </div>
@endsection
```

También necesitamos añadir una ruta para activar la vista:

```
Route::get('view/(:num)', function($post) {
        $post = Post::find($post);
        return View::make('pages.full')
                ->with('post', $post);
});
```

Ahora tenemos un blog completamente funcional, con solo lo básico. Echemos un ojo a lo que podríamos hacer para mejorarlo. Estas mejoras podrían venir en futuros capítulos.

15.7 El futuro

Paginación

Podríamos añadir algo de paginación a la página que lista los artículos lo cual podría ser útil cuando la cantidad de artículos sea demasiado elevada.

Editar / Borrar artículos

Añadir la funcionalidad de editar o eliminar artículos no sería demasiado trabajo, y nos permitiría que se realizara mantenimiento en el blog.

Slugs de la URL

Podríamos crear slugs de las URLs de los artículos para usarlas en vez de las `ids` en nuestras URLs lo cual nos dejaría una mejor optimización para buscadores.

16 Pruebas unitarias

Las pruebas unitarias pueden ser muy útiles para un desarrollador web. Se pueden usar para revisar si añadir una nueva característica o modificar el código base de alguna forma, ha alterado de manera accidental alguna otra característica, provocando que esta falle. Algunos desarrolladores incluso practican un desarrollo orientado a las Pruebas, donde las pruebas son escritas antes que el código para asegurar que el código escrito cumple con todos los requisitos.

Laravel nos ofrece el directorio `tests` para albergar todas las pruebas de tu aplicación e incluso añade un comando de ayuda a la interfaz de Artisan para ejecutar pruebas de PHPUnit.

No solo podemos probar la aplicación sino que los bundles también pueden contener sus propias pruebas. De hecho, Laravel tiene un bundle dedicado a las pruebas de características del núcleo del frameowrk. Puede ser encontrado en el repositorio de pruebas de Github de Laravel[1].

16.1 Instalación

No, ¡no vamos a cubrir nuevamente la instalación de Laravel!

Laravel usa el software PHPUnit para ejecutar sus pruebas. Antes de poder usar las características de pruebas unitarias de Laravel, tendremos que instalar este software.

La instalación de PHPUnit puede variar entre distintos sistemas operativos, por lo que creo que sería mejor mirar la documentación oficial de PHPUnit para encontrar las instrucciones de instalación apropiadas. Puedes encontrar la página de instalación aquí[2].

16.2 Creando una prueba

Echemos un vistazo a un caso de prueba. Por suerte, ¡Laravel ha incluido un caso de ejemplo para nosotros!

```
1  <?php
2
3  // application/tests/example.test.php
4
5  class TestExample extends PHPUnit_Framework_TestCase {
6
7          /**
8           * Test that a given condition is met.
9           *
10          * @return void
11          */
12         public function testSomethingIsTrue()
```

[1] https://github.com/laravel/tests

[2] http://www.phpunit.de/manual/current/en/installation.html

```
13          {
14                  $this->assertTrue(true);
15          }
16
17  }
```

Como puedes ver, creamos nuestros casos de prueba en un archivo con la extensión `test.php`, el nombre de la clase debe comenzar con la palabra clave `Test` y debe extender la clase `PHPUnit_Framework_TestCase`. Estas limitaciones no están impuestas por Laravel si no por PHPUnit.

Una prueba de PHPUnit puede contener cualquier número de prueba, como acciones camelcase, con la palabra `test` como prefijo. Nuestras pruebas pueden contener un distinto número de aserciones que deciden si nuestras pruebas pasan o fallan.

Puedes encontrar una lista completa de aserciones en la documentación de PHPUnit[3].

16.3 Ejecutando pruebas

Las pruebas pueden ser ejecutadas usando la interfaz de línea de comandos Artisan. Simplemente usa el comando `test` para ejecutar todas las pruebas.

```
1  php artisan test
```

y el resultado

```
1  PHPUnit 3.6.10 by Sebastian Bergmann.
2
3  Configuration read from /home/daylerees/www/laravel/develop/phpunit.xml
4
5  .
6
7  Time: 0 seconds, Memory: 6.25Mb
8
9  OK (1 test, 1 assertion)
```

Con la parte `OK` en verde, estamos seguros de que todas las pruebas han tenido éxito.

Por supuesto sabíamos que la prueba tendría éxito porque estamos usando el método `assertTrue()` para revisar el valor `true`. No hay forma de que eso pudiera fallar.

Vamos a fastidiar la prueba para que falle. Vamos a cambiar el parámetro a `false`.

[3] http://www.phpunit.de/manual/current/en/writing-tests-for-phpunit.html#writing-tests-for-phpunit.assertions

```
...
$this->assertTrue(false);
...
```

y el resultados:

```
PHPUnit 3.6.10 by Sebastian Bergmann.

Configuration read from /home/daylerees/www/laravel/develop/phpunit.xml

F

Time: 0 seconds, Memory: 6.50Mb

There was 1 failure:

1) TestExample::testSomethingIsTrue
Failed asserting that false is true.

/home/daylerees/www/laravel/develop/application/tests/example.test.php:12
/usr/bin/phpunit:46

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

Ahora tenemos líneas brillantes en rojo para indicar que las pruebas han fallado incluyendo algunos detalles de porqué las pruebas han fallado.

Si quisiéramos probar un bundle, simplemente pasaríamos un parámetro al comando ‘test. Por ejemplo:

```
php artisan test mybundle
```

16.4 Probando el núcleo

El nucleo de Laravel está bien probado unitariamente pero si quieres ejecutar las pruebas por ti mismo, aquí tienes cómo hacerlo.

Instala el bundle de pruebas

Descarga el bundle `tests` desde github[4] y extráelo en el directorio `bundles/laravel-tests`. O usa `php artisan bundle:install laravel-tests` para cumplir el mismo objetivo.

Ahora simplemente ejecuta el comando `test:core` para ejecutar las pruebas del núcleo del paquete.

[4] https://github.com/laravel/tests

```
php artisan test:core
```

17 Caché

Laravel ofrece una clase muy simple para usar la Caché, permitiéndote cachear todo lo que necesites durante tanto tiempo como necesites. ¿Confuso? Echémosle un vistazo en acción.

17.1 Configuración

Hay muchas formas de almacenar tus datos cacheados pero debes establecer un driver que elija el método que quieres usar en `application/config/cache.php`. Las opciones son `file`, `memcached`, `apc`, `redis` y `database`. Puede que obtengas mejor rendimiento con `apc` o `memcached` pero voy a usar la caché basada en `file` por la simplicidad de su configuración.

```
1  <?php
2
3  'driver' => 'file',
```

17.2 Estableciendo valores

Puedes usar los métodos `put()` o `forever()` para almacenar datos en la caché.

El método `put()` te merite almacenar un valor durante un determinado lapso de tiempo. Echemos un ojo a este código de ejemplo.

```
1  <?php
2
3  $datos = 'datos_generados_de_forma_complicada';
4  Cache::put('misdatos', $datos, 10);
```

Vamos a imaginarnos que `$datos` no es una simple cadena, pero el resultado de un complejo algoritmo que tardaríamos algo de tiempo en procesar. No queremos procesar estos datos cada vez, por lo que los mantenemos en caché.

El primer parámetro al método es la clave, una cadena usada para identificar los datos cacheados. El segundo parámetro son los datos en sí mismos y el tercero es la cantidad de tiempo en minutos para guardar los datos en caché.

El método `forever()` solamente recibe dos parámetros y hace lo que sugiere su nombre. Los datos son almacenados en caché para siempre.

17.3 Obteniendo valores

Para obtener un elemento de la caché, usa el método `get()`y facilita la clave. Por ejemplo:

```php
<?php

$datos = Cache::get('misdatos');
```

Por defecto, si elemento no existe o ya ha expirado, el método devolverá `NULL`. No obstante, puedes pasar un segundo parámetro opcional que nos facilita un valor por defecto alternativo.

```php
<?php

$datos = Cache::get('misdatos', 'datos complicados');
```

También puedes pasar una closure como segundo parámetro y el valor devuelto por la closure será usado si el dato de la caché no existe. La closure será únicamente ejecutada si la clave no existe.

17.4 Una forma mejor

Puede que te descubras a ti mismo usando a menudo el método `has()` para revisar si existe un elemento en la caché y recurriendo a este patrón que te resultará familiar:

```php
<?php

if (! Cache::has('misdatos'))
{
        $datos = 'datos_generados_de_forma_complicada';
        Cache::put('misdatos', $datos, 10);
}

return $datos;
```

Sin embargo, puedes usar el método `remember()` como una solución mucho más elegante. El método `remember()` devolverá el valor si existe en la caché o almacenará y devolverá el valor del segundo parámetro durante un determinado lapso de tiempo. Por ejemplo:

```php
<?php

return Cache::remember('misdatos', function () {
        return 'datos_generados_de_forma_complicada';
}, 10);
```

Por supuesto, la closure será únicamente ejecutada si la clave no existe o ha expirado.

¡Diviértete usando la Caché!

18 Autocarga de clases

Con muchos frameworks, saber dónde poner tus archivos y cómo cargar definiciones de clases puede ser un tema complicado. Sin embargo, con Laravel no hay reglas estrictas que se apliquen a la estructura de tu aplicación. El auto loader de Laravel es una librería inteligente que simplifica la carga de clases con convenciones de nombrado de subcarpetas. Es lo suficientemente flexible para gestionar la adición de librerías o paquetes complejos con facilidad. Echemos un vistazo a las funciones que tenemos disponible.

18.1 Asociación

La asociación es el método más simple de cargar clases. Puedes pasar al auto loader una matriz de nombres de clases con ubicaciones de archivos en forma de parejas clave-valor y Laravel se encargará del resto. El eficiente autoloader solo cargará la clase requerida cuando se use la clase. Por defecto, las asociaciones del Autoader se establecen en el archivo `start.php`. Sin embargo puedes usar la clase desde cualquier parte, pero el archivo `start.php` es una buena elección debido a que se carga muy pronto. Echemos un vistazo a la asociación:

```
<?php

// application/start.php

Autoloader::map(array(
        'Cloud'                 => path('app').'libraries/ff/cloud.php',
        'Tifa'                 => path('app').'libraries/ff/tifa.php',
));
```

El método `path('app')` es muy útil para obtener la ruta absoluta a la carpeta `application`. También puedes obtener rutas absolutas a otras carpetas usando el método `path()`, he aquí una lista corta.

Método	Directorio
path(‘app’)	application
path(‘sys’)	laravel
path(‘bundle’)	bundles
path(‘storage’)	storage

En el ejemplo de asociación, verás que hemos especificado el nombre de la clase como índice de la matriz y el archivo y la ubicación como valor. Ahora, si quisieramos usar la clase `Cloud`...

```
<?php

$c = new Cloud();
```

El auto loader de Laravel 'detectará' (métodos mágicos de php) que hace falta cargar una definición de clase. Mirará el mapeo de definiciones para ver si existe ahí nuestra clase y procederá a hacer el apropiado `include()` de la clase.

18.2 Carga de directorios

Si tienes un número de clases que siguen el patrón de tener el nombre del archivo igual al nombre de la clase en minúscula, puede que quieras especificar el directorio en vez de cada archivo de forma individual. Puedes hacerlo usando el método `directories()` de la clase `Autoloader`. Sin embargo, esta vez tendrás que facilitar una matriz de valores como ubicaciones y nombres de archivos. Por ejemplo:

```
1  <?php
2
3  Autoloader::directories(array(
4          path('app').'pitufos'
5  ));
```

Ahora todas las clases dentro del directorio `applications/pitufos` serán autocargados siempre que el nombre del archivo coincida con el nombre en minúscula de la clase.

18.3 Asociación por espacio de nombre

PHP 5.3 marcó la llegada de los espacios de nombre. Aunque están lejos de la perfección, esta característica usa la convención PSR-0 de la carga de ficheros. Bajo la PSR-0, los espacios de nombre son usados para indicar la estructura del directorio y los nombres de las clases son usados para identificar el nombre del archivo, por lo que podemos asumir que...

```
1  <?php namespace Paladin\Mago;
2
3  class Princesa
4  {
5          // que bonito
6  }
```

Estará en el archivo...

`paladin/mago/princesa.php`

Antes de poder usar esta convención, Laravel necesita saber dónde está la raíz de nuestro espacio de nombre. Podemos ayudarle a encontrar nuestros ficheros usando el método `namespaces()` de la clase `Autoloader`. Por ejemplo:

```
<?php

Autoloader::namespaces(array(
        'Paladin'        => path('libraries').'paladin'
));
```

Como puedes ver, pasamos una matriz al método. La clave de la matriz representa el nombre del nombre de espacio raíz, en este caso `Paladin` y el valor de la matriz se usa para indicar la carpeta raíz que coincide con ese nombre de espacio, en mi ejemplo `application/libraries/paladin`.

18.4 Asociando guiones bajos

Bueno, realmente no necesitamos asociar el guion bajo. Está ahí en nuestro teclado, siempre observando, juzgando... Mira, guion bajo, puede que abuse de ti un poco pero sin amor. ¡Usar camelcase me parece terrible!

Siento el arrebato, lo hablaré más tarde en privado. El motivo de que el título sea asociación de guines bajos es porque en los viejosh diash de PHP, antes de que los espacios de nombre (y de Laravel) hubieran llegado, muchos desarrolladores decidieron usar guiones bajos en los nombres de los ficheros para separar sub directorios. Laravel ofrece el método `underscored` para acomodarse a este tipo de cargado de clases. Una vez más, pasa una matriz con la clave representando el prefijo de la clase y el valor representando el directorio raíz. Por ejemplo:

```
<?php

Autoloader::underscored(array(
        'Paladin'        => path('app').'jobs/pld'
));
```

y ahora la clase `Paladin_Caballero_Luminoso` cargará su definición del archivo `application/jobs/pld/caballe`

Ahora que ya sabes cómo cargar de forma automática tus clases con Laravel, ¡no tendrás que volver a inundar tu código fuente con sentencias `include()`!

19 Configuración

Laravel tiene muchos archivos de configuración en `application/config` para configurar casi todas las características que ofrece el framework. ¿No sería genial si tú pidieras crear tus propios archivos de configuración de la misma forma? Bueno, hoy es tu día de suerte, ¡porque puedes!

19.1 Creando nuevos archivos de configuración

Los archivos de configuración son simplemente archivos PHP que se encuentran en `application/config` o en un subdirectorio, y devuelven una matriz PHP. Por ejemplo:

```
1  <?php
2
3  // application/config/nuestraconfig.php
4
5  return array(
6
7          'tamano'        => 6,
8
9          'comer'                 => true,
10 );
```

Puedes usar comentarios para hacer que tus archivos de configuración sean más descriptivos. Me gusta usar el estilo de comentarios que usa Laravel. Por ejemplo

```
1  <?php
2
3  // application/config/nuestraconfig.php
4
5  return array(
6
7          /*
8          |------------------------------
9          | Tamaño
10         |------------------------------
11         |
12         | Este es el tamaño de nuestra cosa
13         |
14         */
15
16         'tamano'        => 6,
17 );
```

Estoy seguro de que puedes hacer una descripción mejor. Te habrás dado cuenta de que los archivos de configuración de Laravel están en parejas de clave-valor. El índice representa la clave, y el valor... su valor.

El valor del ajuste puede ser cualquier valor u objeto que PHP soporte. Incluso puede ser una closure. Facilitando una closure puedes hacer fácil al usuario cambiar la configuración haciendo que se cargue desde otra fuente. Por ejemplo:

```
<?php

// application/config/nuestraconfig.php

return array(

        /*
        |------------------------------
        | Tamaño
        |------------------------------
        |
        | Este es el tamaño de nuestra cosa
        |
        */

        'tamano'        => function() {
                $tamano = file_get_contents('ruta/a/archivo.json');
                $tamano = json_decode($tamano);
                return $tamano;
        },
);
```

Ahora nuestra configuración de `tamano` es leída desde un archivo JSON. ¡Simple!

19.2 Leyendo configuración

Podemos leer un ajuste de configuración usando el método `get()`:

```
<?php

$opcion = Config::get('nuestraconfig.tamano');
```

Simplemente pasa una cadena al método con el nombre del archivo, un punto (.) y el nombre de la clave de configuración. El valor será devuelto. Si el archivo está en un subdirectorio, tendrás que usar puntos adicionales para indicar el subdirectorio. Por ejemplo:

```php
<?php

$opcion = Config::get('nuestraconfig.sub.directorio.tamano');
```

A vece, es útil obtener toda la matriz de configuración. Para hacerlo, simplemente especifica el nombre del archivo sin la opción. Para obtener toda la matriz de nuestro archivo usaríamos:

```php
<?php

$opcion = Config::get('nuestraconfig');
```

19.3 Estableciendo la configuración

Para establecer un elemento de configuración, usa el método `set()`. El primer parámetro representa el archivo y el nombre del elemento, en el mismo formato que el método `get()`. El segundo parámetro es el valor que quieres establecer.

```php
<?php

Config::set('nuestraconfig.tamano', 7);
```

Si un elemento de configuración existe en un archivo, será sobrescrito en tiempo de ejecución cuando usas `set()`, pero no será sobrescrito en el archivo de configuración en si. Si un elemento de configuración no existe, una llamada al método `set()` lo creará, pero de nuevo no estará en el archivo de configuración.

Intenta poner tantos ajustes configurables como puedas en archivos de configuración. Hará mucho más sencillo el configurarla si tienes que trasladarte o redistribuir la aplicación.

20 El contenedor IoC

El contenedor IoC es un tema complicado. Mucha gente está confundida por su descripción en la documentación y por un corto período de tiempo, yo estaba entre esa gente. Tras mucha investigación y el apoyo de la fantástica comunidad de Laravel (únete a nosotros en #laravel en el IRC de freenode) he conseguido aclarar el tema con éxito. Espero poder arrojar algo de luz sobre este misterioso tema.

IoC son las siglas para Inversión de Control. No quiero complicar las cosas con una descripción completa. Hay demasiados artículos en la red que cubrirán el lado más complicado de este tema. En vez de eso, piensa en el contenedor como 'Invertir el Control' o 'Devolver el control a Laravel' para resolver nuestros objetos.

Esto es de lo que trata el contenedor, de resolver objetos. Aparte de su uso para inyectar dependencias para usarlos en pruebas unitarias, puedes hacerte a la idea de que el Contenedor IoC es un 'atajo' para resolver objetos complejos o seguir un patrón singleton sin la habitual clase asociada al patrón. Hablaremos más de singleton más tarde. Echemos un vistazo a cómo registrar objetos en el contenedor.

20.1 Registrando objetos

Usemos nuestra imaginación. Como el gran dinosaurio morado nos enseñó en la tele. Vamos a imaginarnos una clase llamada 'Luzambiental' que será usada por toda nuestra aplicación para algún oscur... luminoso propósito.

Por desgracia, nuestra clase de `Luzambiental` necesita mucha configuración antes de poder usarla. Echémosle un vistazo.

```
1  <?php
2  $la = new LuzAmbiental(LuzAmbiental::BRILLANTE);
3  $la->configurar_brillo('max');
4  $la->duracion('10min');
```

¡Wow! Qué de configuración. Pronto nos cansaremos de configurar LuzAmbiental cada vez que queramos usarla. Dejemos que el IoC la instancie por nosotros y vamos a ver un ejemplo de código directamente.

Me gusta poenr este código dentro de `start.php` pero lo puedes poner donde quieras siempre que tus objetos sean registrados antes de que intentes resolverlos.

```php
<?php

// application/start.php

IoC::register('LuzAmbiental', function() {

        // Instanciamos la clase como antes
        $la = new LuzAmbiental(LuzAmbiental::BRILLANTE);
        $la->configurar_brillo('max');
        $la->duracion('10min');

        // Devolvemos el objeto como resultado de la closure
        return $la;
});
```

Usamos el método `IoC::register()` para registrar nuestro objeto en el controlador. El primer parámetro es una cadena que usaremos para resolver el objeto más tarde. Usé la palabra 'LuzAmbiental' ya que es lo que tenía más sentido para mi. El segundo parámetro es una closure que podemos usar para instanciar nuestro objeto.

Dentro de la closure verás el código de configuración habitual de `LuzAmbiental` y devolveremos el objeto `LuzAmbiental` condigurado de la closure.

¡Genial! Nuestro objeto está registrado y eso es todo para el Io... solo bromeaba. Echemos un vistazo a cómo podemos usar nuestro objeto registrado.

20.2 Resolviendo objetos

Ahora tenemos nuestra luz ambiental registrada. Veamos cómo podemos hacerla volver. Hagamos una llamada a `resolve`.

```php
<?php
$la = IoC::resolve('LuzAmbiental');
```

¡Y eso es todo! En vez de crear y configurar una nueva instancia de nuestra `LuzAmbiental` cad avez, hacemos una llamada al método `resolve()` pasando la cadena que identifica al objeto y el contenedor IoC ejecutará la closure creada en la primera sección y devolverá nuestro objeto `LuzAmbiental` instanciado y configurado.

Útil, ¡y ahorra muchas líneas de código!

Puedes registrar y resolver tantos objetos como quieras. Anímate y pruébalo. Por ahora pasemos a Singleton.

20.3 Singletons

Resolver nuestra `LuzAmbiental` es útil, pero ?qué ocurre si instanciar `LuzAmbiental` es realmente costoso en recursos o solo debería ser instanciada una vez? El método `register` no será útil en este caso teniendo en cuenta que la closure es ejecutada con cada llamada a `resolve()` y se devuelve una nueva instancia del objeto cada vez. Es aquí donde entra en juego el patrón singleton.

El patrón de diseño Singleton, implica escribir tus clases de una determinada forma para que puedan ser llamadas usando un método estático y siempre devolverá la misma instancia de sí mismo. De esta forma la clase solo queda instanciada una vez.

Para más información sobre el patrón de diseño Singleton, te sugeriría que hicieras una rápida búsqueda en Google o revises el API de PHP que tiene un artículo sobre el tema.

Singleton puede ser útil, pero necesita una cierta estructura de clase para poder usarlo. El contenedor IoC tiene un método `singleton()` que hace el proceso mucho más sencillo y no necesita ningún tipo especial de clase. Registremos `LuzAmbiental` como singleton:

```
1  <?php
2  // application/start.php
3
4  IoC::singleton('LuzAmbiental', function() {
5
6          // Instanciamos la clase como antes
7          $la = new LuzAmbiental(LuzAmbiental::BRILLANTE);
8          $la->configurar_brillo('max');
9          $la->duracion('10min');
10
11         // Devolvemos el objeto como resultado de la closure
12         return $la;
13 });
```

Como puedes ver, el proceso es prácticamente idéntico al registro de un objeto excepto porque usamos `singleton()` que acepta los mismos parámetros.

Cuando resolvemos nuestra `LuzAmbiental`, la closure solo será ejecutada la primera vez que llamemos `resolve()`. El objeto resultado será almacenado y cualquier llamada futura al método `resolve()` devolverá la misma instancia. Por ejemplo:

```php
<?php

// Se ejecuta la closure y se devuelve una
// instancia de LuzAmbiental
$la = IoC::resolve('LuzAmbiental');

// La misma instancia de LuzAmbiental
// de la sentencia anterior es devuelta
$otra = IoC::resolve('LuzAmbiental');
```

¡Genial! Merece la pena destacar que puedes pasar un objeto ya instanciado como segundo parámetro al método `singleton()` y será devuelto por todas las futuras peticiones a `resolve()`. Por ejemplo:

```php
<?php

$la = new LuzAmbiental(LuzAmbiental::SHINY);
IoC::singleton('LuzAmbiental', $la);

// Obtengamos nuestra LuzAmbiental
$luz = IoC::resolve('LuzAmbiental');
```

En un capítulo futuro discutiremos el uso del contenedor IoC y la inyección de dependencias en combinación con las pruebas unitarias.

21 Encriptación

A veces necesitas proteger datos importantes. Laravel facilita dos métodos diferentes para ayudarte a hacerlo. Encriptación en un sentido y en ambos sentidos. Echemos un vistazo a esos métodos.

21.1 Encriptación en un sentido

La encriptación en un sentido es la mejor forma de almacenar contraseñas de usuarios u otros datos sensibles. En un sentido significa que tus datos pueden ser convertidos a una cadena encriptada, pero debido a un complejo algoritmo con aburridas matemáticas, realizar el proceso inverso (desencriptarlo) no es posible.

¡Eso hace que almacenar contraseñas esté tirado! Tus clientes no tendrán que preocuparse de que sepas sus contraseñas, pero aun podrás compararlas (hasheando la contraseña que facilitan) o cambiar la contraseña si te hace falta.

`Ten en cuenta que hashear es el proceso de crear un hash o una cadena encriptada de otra cadena.`

Echemos un vistazo a cómo funciona la encriptación de contraseñas en un sentido.

```
1  <?php
2
3  $pass = Input::get('password');
```

Ahora hemos obtenido la contraseña de nuestro formulario 'crear usuario', ¡pero está en texto plano! Vamos a encriptarla para que podamos almacenarla de forma segura en nuestra base de datos.

```
1  <?php
2
3  $pass = Hash::make($pass);
```

Hemos usado otro de los altamente expresivos métodos de Laravel. En esta ocasión `make` un nuevo `Hash`. Nuestro valor `$pass` contendrá una versión encriptada con bcrypt de nuestra contraseña. ¡Genial!

Digamos que nuestro usuario ha introducido su contraseña para iniciar sesión, y ahora tenemos qu erevisar si es auténtica antes de que pueda iniciar sesión en el sistema. Simplemente podemos comparar nuestro hash con el valor almacenado en la base de datos con el método `check()`.

```php
<?php

$pass = Input::get('password');
if ( Hash::check($pass, $user->password) )
{
        // éxito
}
```

El método `check()` acepta dos parámetros. El valor en texto plano facilitado por tu usuario, y la contraseña encriptada que tienes almacenada. Devuelve un valor booleano para indicar si los valores coinciden o no.

¿Qué ocurre si quisiéramos desencriptar los datos más tarde? Hagamos la encriptación en dos sentidos.

21.2 Encriptación en ambos sentidos

La encriptación en ambos sentidos te permite obtener los datos encriptados en su forma original, ¡parecido a aquellas hojas de espía con las que jugaste cuando eras niño!

La clase `Crypter` de Laravel usa encriptación AES-256 que es facilitada por la extensión `Mcrypt` de PHP. Asegúrate de que esta extensión de PHP haya sido instalada antes de intentar usar esta clase.

Esta clase usa dos métodos sencillos, `encrypt()` y `decrypt()`. Echemos un vistazo a cómo encriptar una cadena.

```php
<?php

$secreto = Crypter::encrypt('Me gusta Hello Kitty');
```

Ahora nuestro oscuro secreto ha sido encriptado con AES-256, y nos heos hecho con el resultado. Esto no tendría uso si no pudiéramos descifrar el secreto más tarde. Echemos un ojo a cómo puedes desencriptar unos datos encriptados.

```php
<?php

$secreto_descifrado = Crypter::decrypt($secreto);
```

¡Realmente fácil! Símplemente dale la cadena al método `decrypt()` y el resultado descifrado te será devuelto.

¡Disfruta usando la clase `Crypter` para simular el sentimiento de usar anillos super secretos de decodificación que conseguiste una vez en una caja de cereales!

22 Contenido AJAX

Las aplicaciones web modernas no tienen el tiempo para esperar a la siguiente petición HTTP. JavaScript ha cambiado la forma en la que navegamos, queremos que nuestro contenido se actualice de forma automática. Queremos publicar información sin tener que recargar la página.

En el canal IRC de Laravel somos preguntados a menudo cómo usar AJAX con Laravel, lo cual parece confuso porque la respuesta es realmente simple 'como cualquier otra petición HTTP'. Vamos allá y veamos algunas peticiones AJAX usando el framework. Vamos a necesitar hacer algunas peticiones HTTP con JavaScript. Como puede ser algo feo, he decidido usar el framework de JavaScript, jQuery en estos ejemplos.

22.1 Plantilla de la página

Necesitaremos una vista, o plantilla de página que sirva como primera petición. Dejemos algo muy básico.

```
<!-- application/views/template.php -->
<!DOCTYPE HTML>
<html>
<head>
        <meta charset="UTF-8">
        <title>Mi página Ajax</title>
</head>
<body>
        <div id="content">
                <p>¡Hola y bienvenido al sitio!</p>
        </div>
        <a href="#" id="load-content">Cargar contenido</a>
</body>
</html>
```

Ahora tenemos un área de contenido definida en un elemento `<DIV>` con la `id` de `content`. Es ahí donde vamos a cargar nuestro futuro contenido. También tenemos un enlace con la `id` de `load-content` que podemos usar como activador para cargar nuestro nuevo contenido dentro de la `<DIV>` superior.

Antes de que podamos ver esta vista, necesitaremos una ruta para cargarla. Voy a hacer que sea mi ruta básica:

```
<?php

// application/routes.php
Route::get('/', function() {
        return View::make('template');
});
```

Ahora, si visitamos `http://localhost` recibiremos una calurosa bienvenida, pero hacer click en el enlace no cargará nada sin algo de javaScript. Aunque no necesitaremos JavaScript si no tenemos nada que cargar. Vamos a crear una nueva ruta y vista de contenido que vaya a ser cargada en la `<DIV>`.

First the view..

```
<!-- application/views/content.php -->
<h1>Nuevo contenido</h1>
<p>Este es nuestro contenido cargado con AJAX.</p>
```

y una ruta para servir el contenido. Ten en cuenta que no estamos embebiendo el contenido en una plantilla. Si hicieramos eso, la plantilla sería repetida por segunda vez cuando AJAX cargara nuestro nuevo contenido.

```
<?php

// application/routes.php
Route::get('content', function() {
        return View::make('content');
});
```

Genial, ahora tenemos nuestra plantilla principal, y tenemos una ruta secundaria con algo de contenido que cargar con AJAX por lo que, comencemos con JavaScript.

22.2 El JavaScript

Bueno, soy un desarrollador PHP por lo que mis habilidades en JavaScript no van a mostraros un gran nivel de maestría. Si puedes en contrar una forma mejor de llevar a cabo esas acciones (y hay muchas alternativas), ve y pruébalas. ¡Estoy seguro de que puedes hacerlo mejor!

Primero, creemos un nuevo archivo llamado `script.js` y déjalo en `public/js` junto a la última versión de jQuery que, en mi caso es simplemente llamado `jquery.js`. Editemos nuestra plantilla principal y añadamos referencias a esos archivos usando el método `HTML::script()`.

```
<!-- application/views/template.php -->
<!DOCTYPE HTML>
<html>
<head>
        <meta charset="UTF-8">
        <title>Mi página Ajax</title>
</head>
<body>
        <div id="content">
                <p>¡Hola y bienvenido al sitio!</p>
        </div>
        <a href="#" id="load-content">Cargar contenido</a>

    <!-- javascript files -->
    <script type="text/javascript">var BASE = "<?php echo URL::base(); ?>";\
</script>
    <?php echo HTML::script('js/jquery.js'); ?>
    <?php echo HTML::script('js/script.js'); ?>
</body>
</html>
```

Como puedes ver, he hecho referencia a mis archivos JavaScript antes de la etiqueta de cierre para que las peticiones HTTP de carga no bloqueen la carga de la página. Es una buena práctica que seguir. Comencemos y añadamos algo de JavaScript a nuestro archivo `public/js/script.js`.

También creamos una nueva variable `BASE` para que JavaScript sepa la URL base de nuestra aplicación. La necesitaremos luego para crear las URLs de petición.

```
// public/js/script.js
jQuery(document).ready(function($) {
        // Ejecutemos el código cuando
        // la página haya sido cargada por completo
});
```

Aquí estamos usando el objeto `jQuery()` para que se haga con el documento actual, y añadimos un evento `ready()` con una closure para que contenga nuestro código. Esperando aque `document` esté listo, podemos asegurarnos de que todos los objetos del `DOM` han sido cargados y que la librería de jQuery ha sido cargada.

Puede que veas otros ejemplos como este...

```
// public/js/script.js
$(document).ready(function() {
        // Ejecutemos el código cuando
        // la página haya sido cargada por completo
});
```

Está bien, pero podría crearte problemas si otras librerías de JavaScript deciden usar el objeto `$` más tarde. Mi método usa la clase jQuery y recibe `$` en la closure que ha sido asignada al objeto jQuery. Esto evita colisiones.

Comencemos creando una función gestora del evento `click` para nuestro enlace que carga contenido. Hay muchas formas de hacer esto con jQuery pero voy a usar el método `.click()`. Allá vamos:

```
// public/js/script.js
$(document).ready(function() {

        $('#load-content').click(function(e) {
                e.preventDefault();
        })

});
```

Ahora ya estamos escuchando al evento. Facilitando el parámetro `event` llamado `e` a la closure, podemos usar el método `e.preventDefault();` para evitar que la acción de click por defecto tenga lugar. En este caso, el link no actuará como un hiper-enlace. Ahora necesitamos hacerotra petición HTTP para obtener el nuevo contenido y cargarlo en nuestra zona `#content`. Usemos el método `.get()` de jQuery para llevar a cabo esta tarea.

```
// public/js/script.js
$(document).ready(function() {
        $('#load-content').click(function(e) {
                // evitamos que la acción por defecto
                // del enlace tenga lugar
                e.preventDefault();

                // intentamos obtener el nuevo contenido
                $.get(BASE+'content', function(data) {
                        $('#content').html(data);
                });
        })
});
```

¿Recuerdas la variable `BASE` que establecimos antes? Podemos usarla para construir nuestra URL de petición, y creamos un método callback que capture los datos que obtenemos. Inyectaremos la respuesta de nuestra petición `GET` en el elemento `#content` usando el método `.html()`.

Eso fue mucho trabajo, ¿verdad? Al menos ahora podemos ver todo nuestro trabajo en acción, carguemos la aplicación en `http://localhost` y hacer click en el enlace de carga de contenido. ¡Espero que haya funcionado!

Como puedes ver, usar AJAX con Laravel es lo mismo que usar cualquier otro framework o PHP plano. Solo que tienes que asegurarte de dar formato a las vistas para tus rutas AJAX de la forma adecuada.

22.3 Envío de datos

A veces tendrás que enviar datos adicionales con tus peticiones. Creamos una nueva petición `POST` HTTP con jQuery para demostrar cómo puede hacerse. Priemro tenemos que definir una ruta que responda a la petición `POST`.

```
<?php

// application/routes.php
Route::post('content', function() {
    // gestiona los datos POST
});
```

Usando el método `Route::post()` en vez de `get()` nuestra ruta de contenido responderá a nuestra petición `POST`. Comencemosc on JavaScript.

```
// Intentemos una petición POST con
// algunos datos adicionales
$.post(BASE+'content', {
        nombre: "Dayle",
        edad: 27
}, function(data) {
        $('#content').html(data);
});
```

Estamos usando el método `POST` para enviar algunos datos a la ruta de contenido, podemos recibir los datos con Laravel de la misma forma en que lo hacemos con un formulario usando la clase `Input`.

```php
<?php

// application/routes.php
Route::post('content', function() {
    echo Input::get('nombre');         // Dayle
    echo Input::get('edad');          // 27
});
```

¡Tan fácil como eso!

22.4 Respuestas JSON

Al interactuar con JavaScript, es útil poder devolver datos en formato JSON. Una matriz basada en una cadena con la que JavaScript está familiarizado.

> Para devolver una respuesta JSON, primero tenemos que codificar nuestra matriz PHP y luego enviar las cabeceras apropiadas para informar al cliente de nuestro contenido de tipo JSON.

La cita de arriba solía ser cierta, pero el método `Response::json()` fue añadido con Laravel 3.2 que consigue el mismo objetivo.

Creemos una nueva ruta.

```php
<?php

// application/routes.php
Route::get('content', function() {

        // nuestra matriz de datos, pronto será JSON
    $data = array(
            'nombre               => 'Dummy',
            'tamano               => 'XL',
            'color'         => 'Azul'
    );

    return Response::json($data);

});
```

Finalmente devolvemos un objeto de respuesta, pasándole nuestra matriz de datos al método `json()` que es codificado en JSON para nosotros. Podemos pasar opcionalmente un código de estao diferente al 200 como segundo parámetro.

22.5 Detectando una petición AJAX

A veces es útil detectar en la ruta si una petición viene por AJAX. Por suerte, Laravel facilita un método para detectar peticiones AJAX. Echemos un vistazo.

```
<?php

// application/routes.php
Route::get('content', function() {

        // vemos si la petición
        // es AJAX
        if (Request::ajax())
        {
                // facilita contenido AJAX
                return View::make('solo_contenido');
        }

        // facilita todo el contenido
    return View::make('contenido_con_disposicion');

});
```

`Request::ajax()` devuelve un booleano. `true` si la petición es AJAX y `false` en caso contrario.

23 Debugueando Aplicaciones

Las aplicaciones escritas usando Laravel son mejores sin bugs. Sin embargo, todos sabemos lo fácil que es que aparezcan cuando tenemos fechas límites que cumplir y jefes enfadados encima nuestra.

PHP tiene distintos métodos para el debug. Desde los sencillos, `var_dump()`s, `print_r()`s, el famoso `die()` y las avanzadas características del debug con la extensión xdebug de PHP. No voy a cubrir esos métodos básicos en detalle, porque este es un libro sobre Laravel, no lo básico sobre PHP. En vez de eso, echemos un ojo a las características que Laravel noss ofrece para cazar esos bugs.

23.1 Gestor de errores

Laravel incluye un gestor de errores personalizado que sobrescribe los errores por defecto de una sola línea de PHP y en vez de ello nos ofrecen más detalles con una traza completa. Echemos un ojo:

```
Unhandled Exception

Message:

View [page.panda] doesn't exist.
Location:

/Users/daylerees/www/panda/laravel/view.php on line 151
Stack Trace:

#0 /Users/daylerees/www/panda/laravel/view.php(93): Laravel\View->path('pag\
e.panda')
#1 /Users/daylerees/www/panda/laravel/view.php(199): Laravel\View->__constr\
uct(' page.panda ', Array)
#2 /Users/daylerees/www/panda/application/routes.php(35): Laravel\View::mak\
e(' page.panda ')
#3 [internal function]: {closure}()
#4 /Users/daylerees/www/panda/laravel/routing/route.php(163): call_user_fun\
c_array(Object(Closure), Array)
#5 /Users/daylerees/www/panda/laravel/routing/route.php(124): Laravel\Routi\
ng\Route->response()
#6 /Users/daylerees/www/panda/laravel/laravel.php(125): Laravel\Routing\Rou\
te->call()
#7 /Users/daylerees/www/panda/public/index.php(34): require('/Users/daylere\
e...')
#8 {main}
```

Este es el error mostrado si solicitamos una vista que no existe en tiempo de ejecución. Como puedes ver, tenemos un error informativo de Laravel, así como el error en que se encontró el error y el número de la línea. Además, tenemos una traza que muestra el error inicial. Siguiendo la llamada del método a través de la capa `View` hasta al sistema de rutas.

La mayoría de las veces no te hará falta esta traza pero podría ser útil para desarrolladores con más experiencia. Por ejemplo, cuando el error ocurre en una librería compleja.

23.2 Configuración de errores

No siempre queremos que los errores se muestren de esta forma, especialmente en un sitio en producción, mostrar una traza podría ser un riesgo de seguridad importante.

Por suerte, y como siempre, Laravel nos hace fácil el cambiar la configuración para mostrar errores. El archivo de configuración para los errores se puede encontrar en `application/config/error.php`. Echemos un vistazo a las opciones de configuración que están en esta matriz.

```
1  'ignore' => array(),
```

La matriz `ignore` contiene una lista de errores que serán ignorados. Aunque esos errores no se mostrarán al encontrarlos, siempre quedarán registrados. Ten esto en cuenta al usar esta matriz.

Para añadir un tipo de error a esta matriz, añade la constante de tipo de error de PHP, o un valor entero a la matriz.

```
1  'ignore' => array(E_ERROR);
```

Puedes encontrar una lista completa de las constantes de los tipos de errores de PHP en el API de PHP. Sin embargo, he aquí algunas de las más útiles.

E_ERROR Este error coincidirá con todos los errores fatales en tiempo de ejecución.

E_WARNING Esta constante coincide con todas las advertencias o errores no fatales.

E_PARSE Esta constante coincide con todos los errores de análisis o errores de sintaxis.

E_NOTICE Esta constante coincidirá con todas las notificaciones en tiempo de ejecución.

E_ALL Esta constante coincidirá con todos los de arriba, excepto los errores `E_STRICT`.

```
1  'detail' => true,
```

La opción `detail` puede ser usada para activar/desactivar el detalle de errores detallado. Cuando esté activado (`true`) mostrará el error completo junto a la traza como has visto arriba. Desactivando esta opción (`false`) provocará una página de error 500 que será mostrada.

```
'log' => false,
```

Si la opción `log` está en `true`, la closure contenida en la opción `logger` será ejecutada con cada error y recibirá un objeto de excepción.

```
'logger' => function($exception)
{
        Log::exception($exception);
},
```

Por defecto, la closure dentro de la opción `logger` escribirá una entrada a un archivo de log dentro de `storage/logs`. Sin embargo, facilitando una closure tenemos un alto grado de flexibilidad, permitiéndonos sobrescribir el método de registro por defecto con cualquier cosa que e te ocurra. Quizá prefieras guardarlo en una base de datos. ¡Adelante!

23.3 Registro

Poder mostrar errores y guardarlos en archivos es útil, pero ¿qué pasa si queremos escribir nuestra propia información? La clase `Log` de Laravel contiene dos métodos diferents para registrar información útil de tu aplicación. Echemos un ojo a esos métodos.

El método `Log::write()` acepta un tipo de registro y una cadena que será escrita en el archivo. Por ejemplo:

```
<?php

Log::write('miaplicacion', 'He aquí algo de información útil.');
```

Esto provocará que la siguiente línea sea escrita en el archivo de registro:

```
2012-05-29 19:10:17 MIAPLICACION - He aquÃ algo de informaciÃ³n Ãºtil
```

Por supuesto, la entrada será escrita en el archivo activo dentro del directorio `storage/logs` en un archivo con el nombre del día actual. Por ejemplo `2011-02-05.log`. Esto permite la rotación de registros y evita que tengamos archivos enormes.

También puedes usar métodos mágicos para establecer el tipo del error. Por ejemplo:

```
<?php

Log::zapato('mi entrada');
```

Tendrá el mismo efecto que...

```
1 <?php
2
3 Log::write('zapato', 'mi entrada');
```

¡Muy útil!

¡Haz uso de todo lo que has aprendido en este capítulo para ayudarte a diagnosticar tu aplicación si algo va mal!

Y ahora, como prometí, una lista de nombres/mensajes para aquellos que han comprado el libro por más de $9.99.

> Please don't forget to talk about the IoC container and how this can be useful with unit tests. You made a great choice with Laravel. - Jorge Martinez

Gracias por comprar el libro Jorge, cubriremos más sobre el contenedor IoC en seccioens posteriores una vez hayamos cubierto la base sobre la creación de una aplicación.

> We at @HelpSpot really support what you're doing to expand the Laravel community, keep up the great work! - Ian Landsman

¡Gracias Ian! La comunidad de Laravel realmente apoya lo que estáis haciendo para expandir el Framework por lo que ahora, ¡estamos empatados! Para aquellos que no lo sepais, Ian Landsman es el presidente de UserScape, una compañía que le dio al autor del Framework un trabajo que le permitió expandir el Framework. UserScape también patrocina el sitio de bundles de Laravel, ¡y muchas más cosas! Por favor, mirad sus productos Helpspot, un programa de helpdesk maravilloso, flexible y a un precio que te puedes permitir[1]!

Me gustaría agradecer a Max Schwanekamp por comprar el libro y también a Douglas Grubba que dijo lo siguiente:

> Keep up the great work Dayle! A wonderful resource. Thank you to the @laravel team. @taylorotwell you are really changing the php world.

¡Gracias Doug! Estoy seguro de que Taylor también se alegrará de oir que está cambiando el mundo de PHP, ¡un método cada vez!

Me gustaría dar las gracias a Enrique Lopez, que rompió el límite de caracteres porque fue lo suficientemente inteligente para comentar de forma bilingüe. Thanks Enrique!

[1] http://www.helpspot.com/

> With Laravel I think what I want and not how to write it. I enjoy programming. (Con Laravel pienso qué es lo que quiero y no como escribirlo. Me divierto programando.)

Gracias a Andrew Smith a Julien Tant (AoSix) por comprar el libro. Julien dijo lo siguiente...

> Hey Dayle, thanks a lot for this book, the Laravel framework is just awesome ! If any french guys read this, visit http://www.laravel.fr/ !

¡Gracias chicos y genial trabajo el traducir los tutoriales al Francés!

Gracias también a Proger_XP por comprar el libro y traducir mis tutoriales al Ruso, que están disponibles en Laravel.ru. ¡Gran trabajo!

Me gustaría decir gracias a las siguientes personas por comprar una copia de Code Happy!, ¡Gracias

- William Manley
- Nate Nolting
- Jorijn Schrijvershof
- Caleb Griffin
- Florian Uhlrich
- Stefano Giordano
- Simon Edwards

Me gustaría también dar las gracias a todos los que enviasteis amables correos sobre el libro, ¡gracias por todo el apoyo!

Me gustaría agradecer a esta gente maravillosa por comprar una copia del libro!

- Marc Carson
- David Wosnitza AKA _druuuuuuuu
- Mark van der Waarde from Alsjeblaft!
- Joseph Wynn
- Alexander Karisson
- Victor Petrov
- Josh Kennedy
- Alexandru Bucur

Hay más compras que pagaron más de $9.99 de las que aun no he recibido un correo. Recuerda que si pagas más de $9.99 y quieres que tu nombre aparezca en el libro, ¡envíame un correo a me@daylerees.com! Por desgracia no puedo ver los nombres de aquellos que compraron en la pestaña de ventas.